Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC

Peer-reviewed author version

MARX, Robin; De Decker, Tom; QUAX, Peter & LAMOTTE, Wim (2019) Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In: Bozzon, Alessandro; Domínguez Mayo, Francisco; Filipe, Joaquim (Ed.). Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019),p. 130-143.

DOI: 10.5220/0008191701300143
Handle: http://hdl.handle.net/1942/29787

Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC

Robin Marx[1], Tom De Decker[1], Peter Quax[1,2] and Wim Lamotte[1]
[1]*Hasselt University – tUL – EDM, Diepenbeek, Belgium*
[2]*Flanders Make, Belgium*
{*robin.marx, peter.quax, wim.lamotte*}*@uhasselt.be, tom.dedecker@student.uhasselt.be*

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

Abstract: Not even five years after the standardization of HTTP/2, work is already well underway on HTTP/3. This latest version is necessary to make optimal use of that other new and exiting protocol: QUIC. However, some of QUIC's unique characteristics make it challenging to keep HTTP/3's functionalities on par with those of HTTP/2. Especially the efforts on adapting the prioritization system, which governs how multiple resources can be multiplexed on a single QUIC connection, have led to some difficult to answer questions. This paper aims to help answer some of those questions by being the first to provide experimental evaluations and result comparisons for 11 different possible HTTP/3 prioritization approaches in a variety of simulation settings. We present some non-trivial insights, discuss advantages and disadvantages of various approaches, and provide results-backed actionable advice to the standardization working group. We also help foster further experimentation by contributing our complete HTTP/3 implementation, results dataset and custom visualizations to the community.

1 INTRODUCTION

A revolution is coming to the internet in the form of the nearly standardized transport-layer QUIC protocol (Langley, 2017). Sometimes called "TCP 2.0", QUIC combines over 30 years of practical internet protocol experience into one neat package, using UDP as a flexible substrate. QUIC re-imagines loss detection and recovery, adds full transport-layer end-to-end encryption, allows for 0-RTT overhead connection setups and, of main importance to this work, solves the TCP Head-Of-Line (HOL) blocking problem.

Some of TCP's main strengths, namely reliability and in-order delivery, can lead to severe performance problems in the event of heavy jitter or packet loss (Goel et al., 2017). This is because a TCP connection considers all data transmitted over it as a single, opaque bytestream; it has no knowledge of a higher-layer application protocol, such as the ubiquitous HTTP. This is problematic if those application layer protocols multiplex data from various, independent resources on the single TCP connection. For example, when loading a web page using the HTTP/2 (H2) protocol (RFC7540, 2015) we typically download several separate resources at the same time (e.g., HTML, JavaScript (JS), images). As H2 uses a single underlying TCP connection, data for these distinct resources is scheduled and multiplexed onto this connection, allowing them to share the available bandwidth.

As such, if a TCP packet containing data for just one of these resources is delayed or lost, there should be no reason that succeeding packets containing data for the other *independent* resources, can not simply be processed by the H2 layer. However, this is not what happens in practice. As TCP is unaware of the various HTTP resources, if a packet is lost, subsequent packets cannot be processed until a retransmit of the lost packet arrives. This is called HOL-blocking, see the top part of Figure 1. While this may seem a mild issue, it has been shown to be one of the major downsides of the H2 protocol running on top of TCP (Goel et al., 2017). The key contribution of QUIC in this area is that it moves this concept of independent resources (more generally referred to as 'streams') away from the application level down into the transport layer protocol. QUIC is inherently aware of several streams being multiplexed on its conceptual single connection, and will not block data from stream A or C if there is loss on stream B. Thus it solves TCP's HOL-blocking problem, see the bottom part of Figure 1. It is important to note though, that within a single resource stream, all data is still deliv-

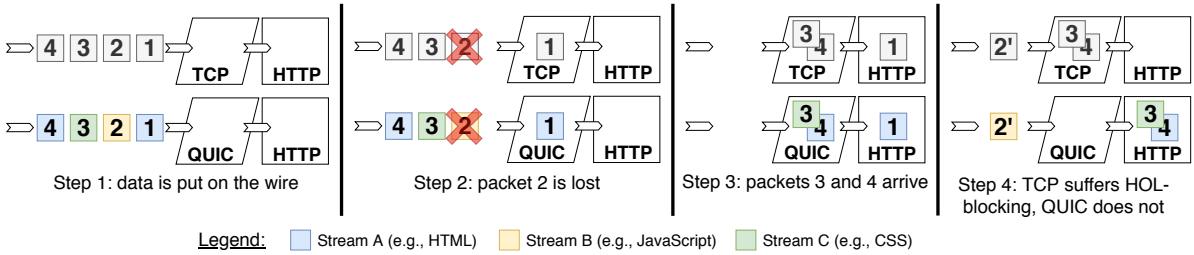

Figure 1: Head-Of-Line blocking in TCP vs QUIC. Lacking knowledge of the three independent streams, TCP is forced to wait for the retransmit of packet 2 (2'). QUIC can instead pass packets 3 and 4 to HTTP immediately, where they are processed before packet 2'.

ered in order and thus there is still HOL-blocking on that level.

QUIC incorporating the concept of streams into its transport layer design leaves H2 in a weird position, as it also strongly defines stream semantics on the application layer. Running H2 on QUIC directly without changes would thus lead to two separate and competing multiplexers. As this 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 HTTP/3 (H3). Despite the higher version number, the intent is that H2 and H3 will exist side-by-side, the first over TCP, the latter over QUIC. Currently, H3 is still just a relatively straightforward mapping of H2 onto QUIC; the main change is that all of H3's stream-specific amenities have been removed in favor of QUIC's streams. However, this seemingly simple mapping introduces some subtle issues, as several concepts in H2 rely on a strict ordering between several control messages. Due to QUIC stream data now potentially being passed onto the H3 layer out-of-order, some H2 approaches no longer hold and need to be revised. Main among them is the prioritization setup, which orchestrates the aforementioned stream data scheduling and multiplexing logic.

At the time of writing, QUIC and H3 are still being standardized[1] within the dedicated IETF QUIC working group. Recently, there have been many discussions on how to approach prioritization in H3. This is only partly due to the out-of-order streams issue though. Another important component is that several implementations of H2's prioritization approach seem to severely underperform in real-world deployments (see Section 2.3). As H2's prioritization system was originally added without much practical experience or proof of validity, the working group is weary of making the same mistake twice. It is torn between wanting to retain as much consistency as possible between H3 and H2 on one hand, and attempting to fix some of H2's most glaring prioritization issues on the other. This work explores both options. Firstly, we explain the subtle issues and background underlying the prioritization systems (Sections 2 and 3). Secondly, we compare different proposed approaches on their various merits (Sections 3.3 and 4). Thirdly, we perform experiments for 11 different prioritization schemes on realistic websites in various conditions (Section 5). Lastly, we make several actionable recommendations to the wider QUIC community in an in-depth discussion (Section 6). An extended version of this text, our source code, dataset, results and visualizations are made publicly available at https://h3.edm.uhasselt.be.

[1] tools.ietf.org/html/draft-ietf-quic-http

2 HTTP/2 PRIORITIZATION

2.1 Background: Web Page Loading

Web pages typically consist of different (types of) resources (e.g., HTML, JavaScript (JS), CSS, font, image files), which have very distinct characteristics during the loading process. For example, HTML can be parsed, processed and rendered incrementally. This is different from JS and CSS files, which can be parsed as data comes in but have to be *fully* downloaded to be executed and applied. Additionally, CSS files are HTML render-blocking: the browser engine cannot just continue rendering any HTML after a new CSS file is included, as this CSS might impact what that following HTML should look like. JS is even worse; it is HTML parser-blocking, as it might programmatically change the HTML structure, removing or adding elements. Consequently, JS and CSS files referenced early in the HTML should be downloaded as soon as possible.

Another issue is that not all the needed resources are known up-front, as they are discovered incrementally during the page load. Most are mentioned in the HTML markup directly, but many (e.g., fonts, back-

ground images) are often imported from within CSS or JS files, and are only discovered when those files are fully executed.

A final aspect is that the user typically does not get to see or interact with the full web page immediately, as it often extends below the current screen height. The immediately visible part is typically called "Above The Fold" (ATF). As such, resources that are ATF are conceptually more important than those "Below The Fold". Thus, resources that appear first in the HTML (and their direct children) are usually considered the most important ones.

Combining all these points, it is clear 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. As much of this information is unknown to the browser before the page load starts, user agents typically resort to complex heuristics for determining relative resource importance in practice. To complicate things even more, all browser implementations employ (subtly) different heuristics, see (Wang et al., 2013) and (Wijnants et al., 2018).

2.2 Dependency Tree: What and Why?

This idea that the client should use heuristics to steer the server's resource scheduling underpins H2's prioritization system. H2 provides the client with so called PRIORITY frames, control messages that it can use to communicate its desired per-resource scheduling setup to the server. The practical system by which this scheduling is accomplished on the server is in the form of a "dependency tree", in which each individual resource stream is represented as a single 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, ideally resulting in the following scheduled packet sequence: AABAABAAB....

As such, the browsers have to map their internal heuristics onto this type of tree structure. While the tree setup is tremendously flexible and allows for an abundance of approaches (Section 4.1), it is non-trivial to define a good mapping for the heuristics in practice. For example, it is unclear up-front what this dependency tree should look like, as its form can change frequently during the page load. If newly discovered resources are of a higher priority than other,

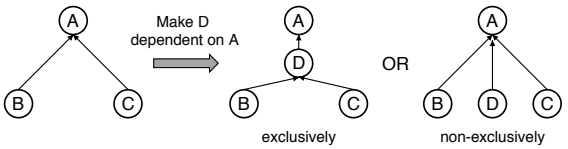

Figure 2: HTTP/2 dependencies: exclusivity.

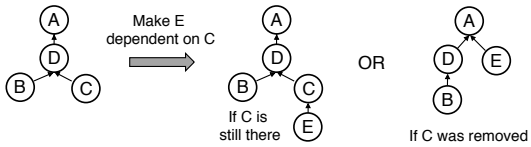

Figure 3: HTTP/2 behaviour when referenced parent does not exist. E is added as a sibling of D on the root, (unintentionally) sharing its bandwidth.

previously requested resources (which already have a node in the tree), the browser might wish to initiate a re-prioritization. This means the new, high-priority resource node needs to somehow be added to the tree so that it will (immediately) receive more bandwidth than the already present, but lower-priority resource nodes. As such, the tree's structure can become very volatile and complex.

H2 adds to this complexity by allowing various ways for clients to (re-)prioritize resources. Firstly, nodes can be added as children to a parent in two ways: exclusively and non-exclusively. As can be seen from Figure 2, non-exclusive addition is the 'normal', less-invasive way of adding nodes to the tree. Exclusive addition however, changes all of the potential siblings beneath a 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. Secondly, as nodes can depend upon other nodes, it is also possible to group nodes together under conceptual 'placeholder' nodes: these do not necessarily represent a real resource stream, but rather just serve as anchors for other streams.

Now, whenever the server has the ability to send packets, it re-processes the dependency tree, determining which resource data should be put on the wire. Depending on the implementation, the frequent (re-)processing of the tree to determine the proper next resource can be non-trivial and computationally expensive. Additionally, there is the memory cost of maintaining the tree structure. To combat this, servers are allowed to remove nodes from the tree once their resource is transmitted fully. However, this can lead to problems if the client attempts to add a new node to a parent node that was already removed. At this point, H2 specifies that the server should fall back to the conceptual root of the tree as a parent instead. This can unintentionally promote the importance of a re-

source, as it is now a sibling of more important nodes under the root. See for example Figure 3, where on the right side E should conceptually have been added as a child of D instead. Note that while in this example there is only one mis-prioritized node (E), its possible that there are many at the same time, exacerbating the problem.

Given this complexity, one might start to wonder why we decided it was the client that had to determine the resource priorities in the first place. Could we not make a similar argument that the server (usually) already has all the resources and thus has a good overview from the start? We could also say that the server is controlled by the web developers, who have knowledge of the intended resource priorities up front. While these seem like solid arguments, in practice there are many reasons why this is typically much easier said than done. Still, to support such use cases, H2 also allows the server to simply ignore the client's PRIORITY messages and instead decide upon the proper resource scheduling itself. As such, our road to flexibility seems complete, being able to choose either client-side or server-side prioritization and scheduling.

One might make the argument that this flexible setup is too complex just to support the requirements of the web page loading use case. Indeed, as we will see in Section 3, there are several proposals for H3 that dramatically simplify this setup (e.g., by not constructing a dynamic tree), while still supporting fine-grained bandwidth distribution. Why then was this complex system chosen in the first place? As with many protocol design decisions, a lot of the finer details are lost in the sands of time. From what we were able to piece together from various H2 mailing list threads[2] and conversations with original contributors, it seems that it was mainly meant to support more advanced use cases for cross-connection prioritization. For example, some parties envisioned multiplexing multiple (H2) connections onto a single H2 connection. This is for example interesting in the case of a general purpose proxy/VPN server or a load balancing/edge server in a Content Delivery Network (CDN). Another use case was for browsers to have multiple tabs/windows open of the same web site, which can share the same, underlying H2 connection.

Surely, you might think, if the dependency tree scheduling was added to H2 to support these use cases, they must be implemented and deployed at scale? Sadly, this is not the case. To the best of our knowledge and as indicated to us by many of the involved companies, no browsers, CDNs, proxy or web servers implement these advanced use cases today. In fact, even the simpler use cases of fine-grained bandwidth sharing for a single web page load are barely utilized or improperly implemented and deployed in practice.

[2] lists.w3.org/Public/ietf-http-wg/2019AprJun/0113.html

2.3 Related Work: Theory vs Practice

(Wijnants et al., 2018) 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 and prioritization scheme, using multiple levels of placeholders and complex weight distributions (see Figure 9). Google's Chrome instead opts for a purely sequential model where all resources are added to a parent exclusively. Apple's Safari goes the other route with a purely interleaved model where all resources are added non-exclusively to the root, using different weights to achieve proper scheduling. Microsoft's Edge browser (before its move to the Chromium engine) neglected to specify any priorities at all, relying on H2's default behaviour of adding all the resources to the root with a weight of 16 (leading to a Round-Robin bandwidth distribution). They reviewed these various approaches, and concluded that H2's default Round-Robin behaviour is actually the worst case scenario (see also Section 3.3), while the other browsers' approaches are also suboptimal.

Next, Patrick Meenan and Andy Davies investigated how well various H2 implementations actually (re-)prioritize resources in practice (Davies and Meenan, 2018). They first request some low-priority resources. After a short delay, they then request a few high priority resources, expecting them to re-prioritize the dependency tree and be delivered as soon as possible. They find that out of 35 tested CDN services and H2 web server implementations, only 9 actually properly support (re-)prioritization. They posit that these problems arise for various reasons. Firstly, some implementations simply have faulty H2 implementations or servers do not adhere to the client's PRIORITY messages. Secondly, implementation inefficiencies cause data to be mis-prioritized[3]. Thirdly, they identify various forms of 'bufferbloat' as the main culprit. If deployments use too large buffers, the risk exists that these buffers will be filled with low-priority data before the high-priority requests arrive. It is often difficult or impossible to clear these buffers to re-fill them with high-priority data when needed. (Patrick Meenan, 2018) suggests limiting the application-level buffers' size, and to use the BBR

[3] blog.cloudflare.com/nginx-structural-enhancements-for-http-2-performance

congestion control mechanism as solutions. Finally, akin to (Wijnants et al., 2018), they indicate that the browsers' heuristics and their mapping to the H2 dependency mechanism are not optimal, and propose a better scheme in (Patrick Meenan, 2019a), which we will refer to as *bucket* later.

Next to this H2 specific work, there are also contributions looking at optimal browser heuristics and prioritization in general. The WProf paper (Wang et al., 2013) looks at resource dependencies and their impact on total page load performance. They instrument the browser to determine the 'critical resource path' for a page load and use that to up-front determine optimal resource ordering. Similarly, Polaris (Netravali et al., 2016), Shandian (Wang et al., 2016) and Vroom (Ruamviboonsuk et al., 2017) collect very detailed loading information (down to the level of the JS memory heap) and construct complex resource transmission and computation scheduling schemes. Polaris and Shandian claim speedups of 34% - 50% faster page load times at the median, while Vroom even reports a flat median 5s load time reduction. However, while their approaches are perfect candidates for H2's server-side prioritization, none of these implementations choose that option. Instead they use custom, JS-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 (Patrick Meenan, 2019a). This scheme aligns more closely with their server implementation and is preferred over the web browser's PRIORITY messages. They claim improvements of up to 50% for the original Edge browser. Overall, we can conclude that advanced server-side prioritization remains relatively unproven in practice.

3 HTTP/3 PRIORITIZATION

While the tenet of the QUIC working group has (so far) mainly been to keep H3 as close to H2 as possible, lately there have been discussions on whether to introduce major changes into the prioritization system. Firstly, the dependency tree setup is quite complex and little of its full potential is being used in the wild. Secondly, due to QUIC's independent streams, the system can not be ported over to H3 in a trivial manner. The working group has long struggled with this latter aspect and has only very recently taken steps to solve some of the issues that arise from the QUIC mapping. We will now first discuss which problems were originally identified and which solutions where included up until draft version 20[4] of the H3 Internet-Draft document. Then we look at how and why those solutions were changed in draft version 22 in July 2019[5] (due to a mistake in the process, draft version 21 was never officially published).

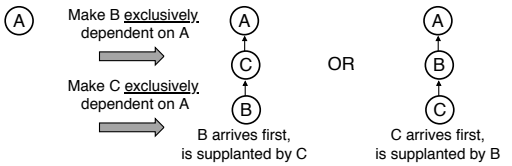

Figure 4: Exclusive dependency end state in HTTP/3 for two concurrent operations is non-deterministic.

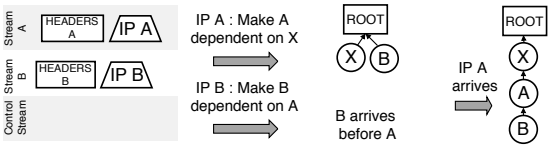

Figure 5: HTTP/3 before draft-22: B 'steals' bandwidth from X. (IP: Initial PRIORITY message).

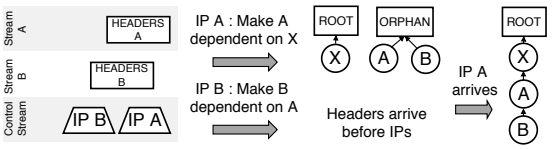

Figure 6: HTTP/3 after draft-22: A and B no longer steal bandwidth from X and are sent only once X has finished.

3.1 Before Draft-22

One of the major problems in bringing H2 prioritization to H3 is in the concept of exclusive dependencies, which can move multiple nodes in the tree (see Section 2, Figure 2). This approach relies heavily on the correct ordering of PRIORITY messages. As they are sent on the resource stream they are meant to provide priority information on, this can lead to problems. Due to packet loss or jitter on the network, in QUIC these PRIORITY messages sent on different streams can now arrive out-of-order, leading to non-deterministic dependency tree layouts, see Figure 4. The original "solution" to this was simply to remove exclusive dependencies from the protocol.

However, the non-deterministic ordering of QUIC streams leads to other problems. For example, let's say A and B are requested immediately after each other, with B indicating A as its parent. If B arrives before A, the server does not yet have A in its dependency tree. It then has two options: either append B

[4] tools.ietf.org/html/draft-ietf-quic-http-20
[5] tools.ietf.org/html/draft-ietf-quic-http-22

to the root of the tree (default fallback), or create a "non-initialized" node for A and hope its PRIORITY message will arrive soon. However, even in the second case, A would have to be added to the root, since we don't know its real parent yet. This leads to the problem discussed in 2.2 and Figure 3, where these new streams potentially compete for bandwidth with streams of much higher importance, see Figure 5. The "solution" for this problem was to simply ignore it, as in this case A should in fact be arriving pretty soon. However, note that in the case of a packet loss on a long-fat network (high bandwidth, high latency) a retransmit of A's PRIORITY message could keep B mis-prioritized for over 1 Round-Trip-Time (RTT). If B is relatively small and the connection's congestion window is large, B could potentially be fully transmitted or put into buffers before A's retransmitted request arrives.

In order to partially alleviate this problem in the case of *updates* to the priority of existing nodes (an additional PRIORITY message is sent), H3 uses a separate "control stream". As this is a single, conceptual stream, all messages sent on it are fully-ordered, and the updates are applied in the expected order. As such, in draft-20, normal H3 modus operandi is to send the initial PRIORITY message as the very first data on the resource stream itself, and subsequent PRIORITY messages for that resource on the separate control stream. However, this does not completely eliminate all edge cases. As a potentially better solution, the text also allows implementations to send the *initial* PRIORITY message on the control stream (see the leftmost part of Figure 6). While this provides deterministic tree buildup, it again suffers from the same problem as above: if the request stream's HTTP headers arrive before the *initial* PRIORITY message is received on the control stream, the request stream is (temporarily) added as direct child of the root node.

In an attempt to prevent these issues from happening in practice, the concept of placeholder nodes was revisited. In H2, servers could potentially remove the placeholders from the tree prematurely, as they were merely simulated using idle resource streams. As such, in H3 these nodes are explicitly made separate entities in the tree. They are created up-front at the start of the connection to create a harness for the prioritization setup, and are never removed. As such, if resource nodes only depend on placeholders, those parents will always be in the tree and these issues do not occur. However, other edge cases still remained.

3.2 Draft-22

Given the suboptimal state of prioritization in draft-20, working group members had their choice of two main directions to continue in: Either attempt to move the design even closer to H2 (e.g., by re-introducing exclusive priorities) or introduce more impactful changes to the setup (e.g., moving away from the dependency tree setup). As proof was not yet available that the second option would lead to performance on par with or better than the H2 status quo, for the time being, the working group decided to bring H3's prioritization system closer to the original H2 setup. This was accomplished by two main changes: Firstly, *all* PRIORITY messages are now required to be sent on the control stream[6], where before the Initial PRIORITY messages could be sent on the resource stream itself. As now all PRIORITY altering information is fully ordered again, this allows for the re-introduction of exclusive dependencies into the specification[7]. The downside of this change however is that, as before, problems can arise if PRIORITY messages on the control stream are lost: unprioritized request streams are added directly to the root with a weight of 16 (the default H2 fallback behaviour), where they can unintentionally "steal" bandwidth from higher-priority streams.

The best solution to that issue was deemed to change this default fallback behaviour. Partly in thanks to the early results of this work, the concept of an "orphan placeholder" was introduced[8] to help resolve this issue. This special purpose placeholder replaces the dependency tree root as the default parent for unprioritized nodes, but is not part of the normal dependency tree and has special semantics. The text states that children of the Orphan Placeholder can only be allotted bandwidth if none of the streams in the main dependency tree can make progress (or there are no more open prioritized streams under the root). This means that unprioritized streams will never get to send data as long as there is data available for prioritized streams, thus preventing the unintentional bandwidth "stealing" (Figure 6).

3.3 Alternatives for HTTP/3 Priorities

Next to H3 draft-22, there also existed several proposals for H3 that aim to introduce alternative schemes to the one defined in H2. Several of these flowed from the aforementioned insight that a Round-Robin (RR) bandwidth distribution scheme is undesirable in most

[6] github.com/quicwg/base-drafts/issues/2754
[7] github.com/quicwg/base-drafts/pull/2781
[8] github.com/quicwg/base-drafts/pull/2690

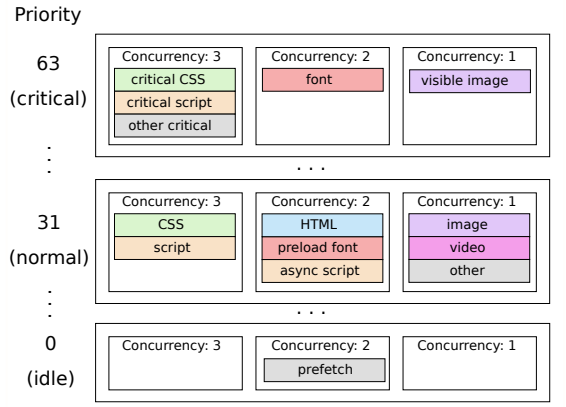

Figure 7: Proposal for HTTP/3 prioritization based on priority buckets, from (Patrick Meenan, 2019b).

web page loading use cases (Wijnants et al., 2018) and (Patrick Meenan, 2019a). This is mainly because, as discussed in Section 2.1, for many high-priority resources (e.g., JS, CSS, fonts) it is imperative that they are downloaded *in full* as soon as possible. As discussed in detail in Section 5 and as can be seen at the top of Figure 8, RR bandwidth interleaving leads to resource downloads being *completed* very late. As such a more sequential scheduler, which for example sends a single resource at a time, is a much better approach for many important resources, while a RR scheme is more apt for lower-priority resources that can be incrementally used (e.g., progressive images). Since at the time, H3 did not support exclusive addition of nodes anymore, this type of sequential prioritization was more difficult to obtain, and so many of the proposals focus on ways to make this easier to accomplish. A second main issue was the potential overhead of placeholder nodes. As they are created up-front and cannot be removed while the connection remains alive, an attacker who sets up a large amount of placeholders could potentially execute a memory-based Denial of Service attack on the server. As a response, servers can limit the amount of placeholders a client is allowed to open. The question then becomes: how many is enough[9]? Some schemes might require large amounts of placeholders for legitimate reasons. As such, various proposals attempt to limit the amount of placeholders needed[10] or eliminate the need for them altogether. Thirdly, many feel H2's scheme is overly complex and would prefer to see simpler schemes. Finally, a combination of client and server-side scheduling, where both parties contribute importance information at the same time, might have some merits.

The first proposal, termed *bucket* by us, is one by Patrick Meenan from Cloudflare (Patrick Meenan, 2019b). He proposes to drop the dependency tree setup and replace it with a simpler scheme of "priority buckets", see Figure 7. 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 "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 the 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 7, this allows a nice and fine-grained mapping to typical web page asset loading needs. This scheme was deployed for H2 as well on Cloudflare's edge servers and they claim impressive speedups (Patrick Meenan, 2019a). Overall, this scheme is also easier to implement: 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.

A second proposal by Ian Swett from Google[11] called "strict priorities" attempts to integrate the semantics of Patrick Meenan's *bucket* proposal with the existing priority tree setup. Nodes can now have both a priority value and a weight, and siblings with a higher priority are sent before others. By disallowing streams to depend on each other (i.e., streams can only have placeholders as parents), this proposal also side-steps many of the issues discussed before, while allowing sequential sending without needing exclusive dependencies. With this scheme, as with the previous one, placeholders could also be bypassed completely. While we could describe this proposal as a "best of both worlds" endeavour, it is also relatively complex.

Thirdly, our own proposal[12] called "*zeroweight*" has an aim to stay quite close to the default H2 setup. 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 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 2.2). Finally, if all other siblings are processed,

[9] github.com/quicwg/base-drafts/issues/2734
[10] github.com/quicwg/base-drafts/pull/2761
[11] github.com/quicwg/base-drafts/pull/2700
[12] github.com/quicwg/base-drafts/pull/2723

do zero-weighted nodes get bandwidth, again sequentially in the lowest stream ID order. Note that draft-22's Orphan Placeholder could thus be implemented as a zero-weighted placeholder under the root. The resulting tree can be viewed in Figure 10. While this proposal requires just a few semantic changes to the H2 system, and is thus easy to integrate in existing implementations, it does represent a potentially large placeholder overhead. To get fully similar behaviour to the previous two proposals, one would need three placeholders per priority bucket (i.e., nine for the example in Figure 7), as opposed to zero in their proposals. However, a simpler practical setup in the zero weighting scheme, such as the one used in our evaluations and in Figure 10, requires no placeholders.

Note that (our implementations of) both *bucket* and *zeroweight* rely, at least in part, on some additional inside knowledge that is typically not found in browser heuristics. For example, Figure 7 mentions a "visible image" while in practice, browsers have no way of definitively knowing which images will eventually be visible or not. As such, these schemes partially emulate the previously mentioned case of a combination of client and server-side priorities, where the web developer explicitly indicates some up-front priorities. 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.

Various other setups were proposed, among which there was one suggesting to go back to the prioritization scheme of the SPDY protocol (SPDY, 2014). SPDY was the predecessor of H2 and had just "eight levels of strict priorities". As Chrome's default H2 behaviour can be seen as a sequential version of SPDY's setup, for this work we have also created a Round-Robin version of SPDY's general concept, termed *spdyrr*.

It should be noted that none of these proposals introduce radical new ideas. The basic concepts remain those of sequential versus Round-Robin. All the discussed schemes mainly differ in how easy it is to implement them, in their runtime overhead, their support of resource re-prioritization and in how fine-grainedly they allow resource importance to be specified. Even so, it is not immediately apparent that all these options will provide similar or better performance than H2's status quo. In fact, it is not even fully clear if the current H2 prioritization schemes of the various browsers are optimal. The results of (Wijnants et al., 2018) at least seem to indicate that many browsers use clearly sub par prioritization schemes. Given this complex situation, the working group deemed that working H3 implementations and evaluations of the various schemes were required, which this work aims to provide.

4 EXPERIMENTAL SETUP

4.1 Prioritization Schemes

For this work, we have implemented and evaluated 11 different prioritization schemes. Their main approaches are described in Table 1 and Figure 8 shows to what kind of data scheduling they lead in practice. For example, as expected the Round-Robin *rr* clearly has a very spread out way of scheduling data for the various streams. 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 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). Note that we did not implement Ian Swett's proposal, as it should function identically to *bucket* in our evaluation.

4.2 Evaluation Parameters

For easiest comparison with other work, we test the 11 prioritization schemes on the test corpus of (Wijnants et al., 2018). This corpus consists of 41 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). See the original paper for more details. 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 (Davies and Meenan, 2018) (Section 2.3, Figure 8). 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 (Robin Marx, Tom De Decker, 2019). We have exhaustively tested the implementation to make sure any inefficiencies stemming from the underlying JavaScript engine did not lead to performance issues. We choose Quicker because the high level language makes it easy to add support for H3 and to implement our various prioritization schemes. We test the valid-

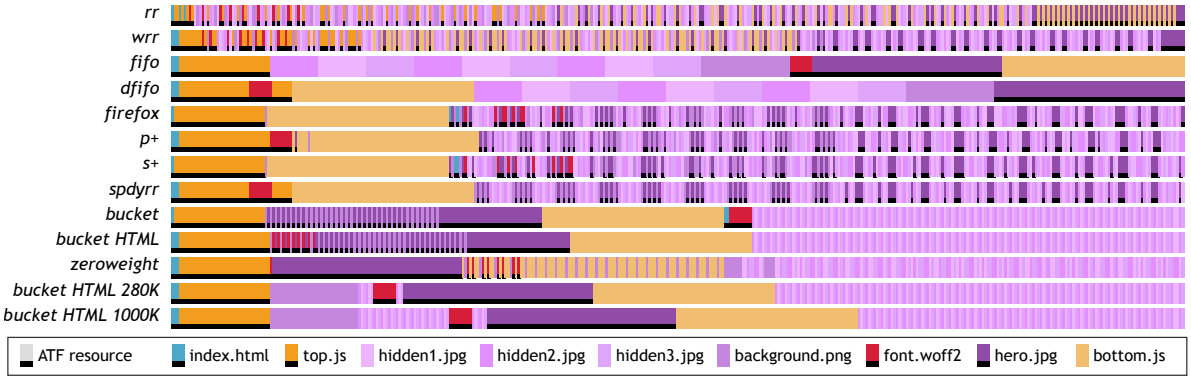

Figure 8: Scheduling behaviour of various prioritization schemes for a single, synthetic test page from (Davies and Meenan, 2018). Each individual colored rectangle represents a single QUIC packet of 1400 bytes. Packets arrive at the client from left to right. The bottom two lines show results with non-zero send buffers. Resources in the legend are listed in request-order from left to right.

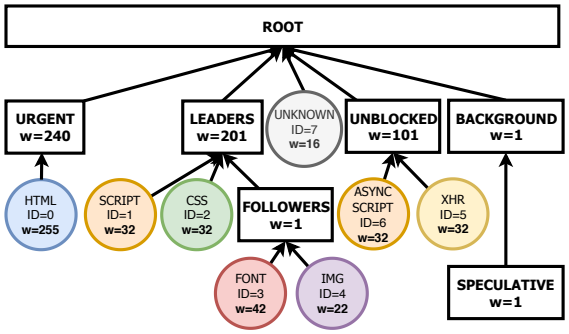

Figure 9: Firefox's HTTP/2 dependency tree.

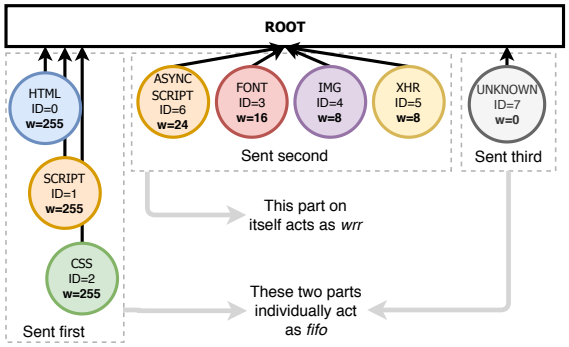

Figure 10: Tree for our HTTP/3 zero weighting proposal.

ity of our H3+QUIC implementation by achieving full interoperability with seven other implementations.

On the client side, there is currently sadly no browser available that supports H3. This also prevents us from doing qualitative user studies at this time. 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[13], an easy to use implementation of the concepts from the original WProf paper (Wang et al., 2013). We host the test corpus on a local optimized webserver (H2O) and load the pages via the Google Chrome-integrated WProfX software. From this load, the tool can extract detailed resource interdependencies (e.g., was an image referenced in the HTML directly or from inside a CSS file) and request timing information. Our H3 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.

None of the open source QUIC stacks (including Quicker) currently has a performant congestion control implementation that has 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 QUIC 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). As such, our results represent an "ideal" upper bound of how well prioritization could perform in the absence of network congestion and retransmits.

While we abstract away from fine-grained congestion control, we do simulate other behaviours. Firstly, we experiment with the effect of small and larger application-level send buffers, to determine if we see the same detrimental "bufferbloat" effects as in (Patrick Meenan, 2018). Secondly, to illustrate QUIC's resilience to HOL-blocking, we add a mode

[13] wprofx.cs.stonybrook.edu

Table 1: Prioritization schemes. The top seven are from browser H2 implementations and (Wijnants et al., 2018). The bottom four are proposals for H3.

Name	Description
rr (Edge)	Fully fair Round-Robin. Each resource gets equal bandwidth.
wrr (Safari)	Weighted Round-Robin. Resources are interleaved 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 9.
p+	Parallel+. Combines *dfifo* for high-priority with separate *wrr* for medium and low-priority resources (Wijnants et al., 2018).
s+	Serial+. Combines *dfifo* for high and medium-priority with *firefox* for low-priority resources (Wijnants et al., 2018).
spdyrr	Five strict priority sequential buckets, each performing *wrr* on their children. The Round-Robin counterpart of *dfifo*.
bucket	Patrick Meenan's proposal, Figure 7.
bucket HTML	Our variation on *bucket* (HTML content is in bucket 63 instead of 31 in Figure 7).
zeroweight	Our proposal, Figure 10.

to Quicker that simulates TCP's behaviour (i.e., packets are only processed in-order). As we do not use congestion control or retransmits, we instead employ network jitter rather than packet loss to demonstrate how QUIC profits from having independent streams (Section 5.2).

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. As discussed in Section 2.1, these resources are either on the browser's critical render path or 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 8), as they should have less of an impact on user experience.

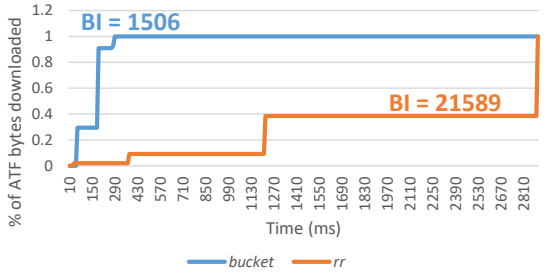

Figure 11: ByteIndex (BI) for *bucket* and *rr* schemes. *Bucket* is clearly faster for ATF resources. Looking at these schemes in Figure 8, it is immediately clear why.

However, 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 (Bocchi et al., 2016). 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 11. 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 proposed qlog standard logging format[14] for QUIC and H3. 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 8, 12 and 13).

5 RESULTS

5.1 Prioritization Schemes

Our main results are presented in Figure 12 and Table 2. Like (Wijnants et al., 2018), we remark that

[14] github.com/quiclog/internet-drafts

Table 2: Mean speedup ratios compared to *rr* per other prioritization scheme from Figure 12. Higher mean values are better. #PH = number of placeholders used in this scheme.

Scheme	#PH	Mean All	Mean ATF	Mean 1000K
wrr	0	**1.05**	1.49	**1.28**
fifo	0	**1.27**	1.93	1.57
dfifo	5	**1.27**	2.30	1.72
firefox	6	1.07	**1.22**	1.25
p+	3	1.17	2.20	1.64
s+	8	1.14	1.45	1.56
spdyrr	5	1.14	1.96	1.57
bucket	0	1.20	2.13	**1.82**
bucket HTML	0	1.20	**2.49**	**1.83**
zeroweight	0	1.15	**2.8**	**1.9**

the *rr* scheme is by far the worst performing of all tested setups, with almost no data points performing worse. As such, 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, 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.

Our main results are presented in Figure 12 and Table 2. Like (Wijnants et al., 2018), we remark that the *rr* scheme is by far the worst performing of all tested setups, with almost no data points performing worse. As such, 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, 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.

A few things are immediately clear from Figure 12: 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 larger 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 the slowdowns of /2 reported in (Patrick Meenan, 2018).

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, the 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 (Patrick Meenan, 2018), it has a simple explanation in two parts. Firstly, larger send buffers mainly impact the ability of the scheme to reprioritize 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 of (Davies and Meenan, 2018) themselves (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. This is clearly visible in Figure 8. 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 by Davies and Meenan for faulty prioritizations in the wild might be less due to "bufferbloat" and more due to misconfigured or badly implemented H2 servers, or that the observed impact is enlarged due 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 12. 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

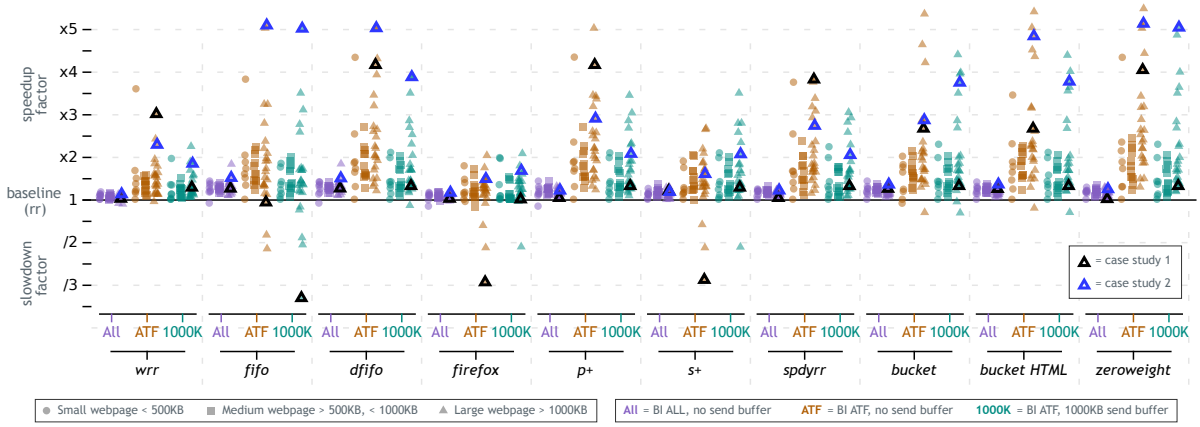

Figure 12: ByteIndex (BI) speedup and slowdown ratios for 10 prioritization schemes compared to the baseline *rr* scheme. Each datapoint represents a single web page, split out by total page byte size. Higher y values are better.

hero image that is relatively small in byte size. Next to this, 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 9), 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 12. 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., *pmeenan* 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 lead 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, leading to relatively low gains for RR-alike schemes.

Finally, looking at Table 2, we can see that the schemes using the most placeholders are partially also those that showed sub par performance in various conditions. Contrarily, the two best performing schemes both use zero placeholders. With regards to overall implementation complexity, *bucket (HTML)* is the only scheme we actually implemented completely separately and this was indeed far easier than the complex dependency tree implementation. However, defining new schemes such as *zeroweighting* or *spdyrr* for the dependency tree was also relatively straightforward.

5.2 QUIC's HOL-blocking Resilience

As mentioned in Section 4, we also try to determine the practical impact of QUIC's absence of HOL-blocking. We induce the HOL-blocking by introducing jitter for semi-random packets: about one packet in four is delayed until 1-3 other packets have been sent. For normal QUIC (*jitter only*), the 1-3 later packets can just be processed and passed on to H3 upon arrival. To determine how much this matters in practice, we implement a HOL-blocking mode in the Quicker client. In this mode, the 1-3 later packets are instead kept in a buffer until the delayed packet arrives, simulating normal TCP behaviour. Partial results for both approaches can be seen in Figure 13.

In opposition to Figure 8 we can now clearly see empty areas where no packets arrived. Packets that arrive together, or that are HOL-blocked and then delivered to the H3 layer together, are drawn stacked vertically. Comparing the two *rr* setups, we can clearly see the beneficial impact of QUIC's independent streams: *rr jitter only* has far fewer stacked packets (maximum of two) than *rr HOL-blocking*, as packets containing independent stream data can be processed directly. In opposition, *rr HOL-blocking* shows various instances of data from (critical) resources being blocked behind

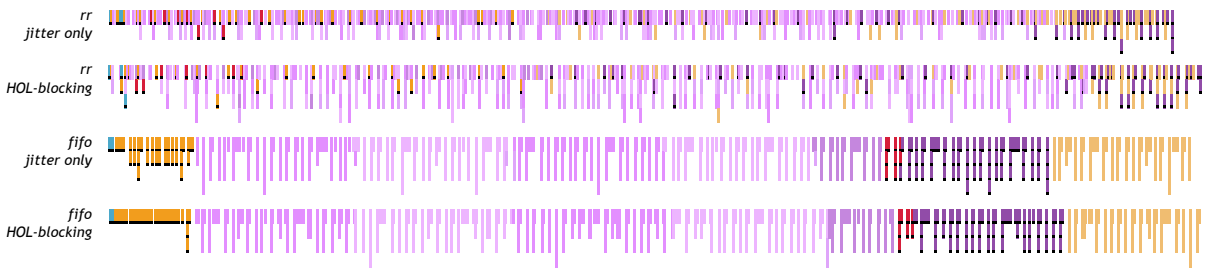

Figure 13: Scheduling behaviour under jitter and HOL-blocking conditions for the same test page as Figure 8. Packets that are stacked vertically are passed from QUIC to HTTP/3 at the same time. The color legend and other semantics are the same as Figure 8.

packets of other (non-critical) streams, leading to frequent stacks of four packets.

However, we do not see similar HOL-blocking resilience for the *fifo* scheme. The reason for this is simple: while QUIC removes inter-stream HOL-blocking, data within a single stream still needs to be delivered in-order. As in *fifo* there is always only a single stream in progress at a time, this stream will always HOL-block itself, undoing one of QUIC's main promised improvements.

6 DISCUSSION & CONCLUSION

Looking back on some of the questions the QUIC working group had about changing H3's prioritization system in Section 3.3, we believe we can now answer most of them.

Firstly, it is indeed a good goal to make sequential behaviour easier to accomplish. As was shown time and time again in our results in Section 5, more sequential schemes generally outperform more Round-Robin-alike schemes. As such, we encourage the working group to adopt *fifo* as the default fallback behaviour, instead of *rr*.

Secondly, we immediately need to nuance our previous point in the case of networks with high packet loss or jitter. There the Round-Robin-alike schemes might actually outperform the more sequential schemes when there are many parallel streams, benefitting fully from QUIC's HOL-blocking resilience (Section 5.2). More experiments on actual lossy networks with functioning congestion control are needed however, to confirm this hypothesis.

Thirdly, 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, do not require placeholders and seem to provide good baseline performance for most sites.

Yet, we have a problem with the "most" in the previous sentence. As our results and case studies have also 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 why some systems (e.g., (Netravali et al., 2016)) aim to automatically determine the exact optimal scheme and why efforts such as "Priority Hints"[15] give developers options to manually indicate resource priorities. However, we feel both these complex automated systems and manual intervention approaches require a lot of effort and do not scale well. In summary, we want to get better performance for individual web pages than default heuristics can provide, but are unwilling to pay high automation or manual labor costs.

So, Fourthly, we propose a different way forward. We suggest that all H3 clients should ideally implement and support several more than one prioritization scheme at the same time. Developers can then use a low-overhead, easily automated "optimal scheme finder" test to find the scheme that performs best for their specific page. They simply need to load their page a few times per scheme using any compliant H3 client. The optimal scheme(s) can then be stored server-side and communicated to new clients during their H3 connection setup. While the chosen scheduler might be less optimal than what a more advanced system could provide, it should perform better than general purpose heuristics, treading an attractive middle ground. Additionally, this approach is still complementary to manual interventions such as priority hints. The ideal combination with a good default client-side scheme (such as *bucket HTML*) ensures that even web servers that do not specify a preferred scheme fall back to decent behaviour. This option would require the working group to provide guidance as to which schemes clients should support and how to best tweak heuristics to them.

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

Finally, note that if we indeed want clients to support a wide array of schemes, this will probably only be possible using a flexible underlying system, such as the dependency tree setup. The high flexibility is probably well worth the added complexity in the long run. Additionally, as most H2 implementations already support this more flexible base framework, our proposed approach of multiple schemes per client could be recommended and implemented for existing H2 stacks as well. Note that this proposal does limit the options for the combination of client and server-side prioritization. As discussed in Section 3.3, in such a flexible system it is difficult to infer the client's semantics, especially if it is now choosing between multiple schemes. However, we feel that this is an inherent problem of how we communicate priority information from the client to the server at the moment. To make proper client and server-side combinations possible, the client would need to send additional metadata (e.g., if a resource is critical, render/parser-blocking, can be processed incrementally, etc. (Section 2.1)), rather than/next to building a dependency tree directly. As this is a heavy departure from H2, this approach is unlikely to make it into H3, but it is worth further investigation. For now, we remark that server-side directives can also be communicated to the client, allowing it to apply them properly at client-side while building the tree, as opposed to the server changing the tree. This is the route taken by the aforementioned Priority Hints proposal, and fits nicely with our proposal of having the server send the client its preferred scheme.

As our general conclusion, we recommend to the QUIC working group to remain with the existing H2 dependency tree system and to possibly even extend it with new capabilities. The provided flexibility is, in our opinion, well worth the additional implementation complexity. Future work can assess QUIC's actual HOL-blocking resilience on lossy networks, look at the dynamics of cross-connection or multipath prioritization, discuss new forms of PRIORITY metadata from client to server and implement a proof-of-concept of the proposed 'optimal scheme finder'.

ACKNOWLEDGEMENTS

Robin Marx is a SB PhD fellow at FWO, Research Foundation Flanders, #1S02717N.

REFERENCES

Bocchi, E., De Cicco, L., and Rossi, D. (2016). Measuring the quality of experience of web users. In *Proceedings of the 2016 Workshop on QoE-based Analysis of Data Communication Networks*, Internet-QoE '16, pages 37–42. ACM.

Davies, A. and Meenan, P. (2018). HTTP/2 priorities test page. Online, https://github.com/andydavies/http2-prioritization-issues.

Goel, U., Steiner, M., Wittie, M. P., Ludin, S., and Flack, M. (2017). Domain-sharding for faster http/2 in lossy cellular networks. *arXiv preprint arXiv:1707.05836*.

Langley, A. e. a. (2017). The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 183–196. ACM.

Netravali, R., Goyal, A., Mickens, J., and Balakrishnan, H. (2016). Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI'16, pages 123–136.

Patrick Meenan (2018). Optimizing HTTP/2 prioritization with BBR and tcp_notsent_lowat. Online, https://blog.cloudflare.com/http-2-prioritization-with-nginx.

Patrick Meenan (2019a). HTTP/2 priorities test page. Online, https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web.

Patrick Meenan (2019b). HTTP/3 prioritization proposal. Online, https://github.com/pmeenan/http3-prioritization-proposal.

RFC7540 (2015). HTTP/2. Online, https://tools.ietf.org/html/rfc7540.

Robin Marx, Tom De Decker (2019). Quicker: TypeScript QUIC and HTTP/3 implementation. Online, https://github.com/rmarx/quicker.

Ruamviboonsuk, V., Netravali, R., Uluyol, M., and Madhyastha, H. V. (2017). Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proc. of the ACM SIG on Data Communication*, pages 390–403. ACM.

SPDY (2014). SPDY Protocol. Online, https://www.chromium.org/spdy/spdy-protocol.

Wang, X. S., Balasubramanian, A., Krishnamurthy, A., and Wetherall, D. (2013). Demystifying Page Load Performance with WProf. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 473–486.

Wang, X. S., Krishnamurthy, A., and Wetherall, D. (2016). Speeding Up Web Page Loads with Shandian. In *Proc. of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI'16, pages 109–122.

Wijnants, M., Marx, R., Quax, P., and Lamotte, W. (2018). Http/2 prioritization and its impact on web performance. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1755–1764. ACM.