HTTP/3's Extensible Prioritization Scheme in the Wild

Joris Herbots
Robin Marx
{name.lastname}@uhasselt.be
Hasselt University, EDM
Diepenbeek, Belgium

Maarten Wijnants
Peter Quax
{name.lastname}@uhasselt.be
Hasselt University, Flanders Make
Diepenbeek, Belgium

Wim Lamotte
wim.lamotte@uhasselt.be
Hasselt University, EDM
Diepenbeek, Belgium

	Fastly	QUIC.cloud	Akamai	Cloudflare	Google Cloud CDN	Google Gstatic	jsDelivr (Fastly)	jsDelivr (Cloudflare)	Shopify	Amazon CloudFront	NGINX	Caddy
Default scheduler	SEQ	SEQ	SEQ	SEQ	INC	INC	SEQ	SEQ	SEQ	INC	INC	INC
Priority request header field	●	●	▲	●	▲	▲	●	▲	▲	▲	▲	▲
PRIORITY_UPDATE frame	●	●	●	○†	●	●	●	▲	▲	▲	▲	▲
Reprioritization	●	●	●	▲	●	●	●	▲	▲	▲	▲	▲
urgency parameter	●	●	●	●	●	●	●	▲	▲	▲	▲	▲
incremental parameter	●	●	▲	●	▲	▲	●	▲	▲	▲	▲	▲
Incremental chunk size (packets)	1	>1	/	1	>1	>1	1	/	/	1	>1	1

Table 1: Comparison of prioritization subfeature support across 12 HTTP/3 endpoints. Green circles indicate support; red triangles, no support. SEQ = sequential, INC = incremental. † Cloudflare ignores PRIORITY_UPDATE frames sent after the request, precluding reprioritization.

ABSTRACT

For HTTP/2 and HTTP/3, multiple (Web page) resources are loaded by multiplexing them onto a single TCP or QUIC connection. A "prioritization system" is used to properly schedule the order in which the resources are sent. As HTTP/2's "prioritization tree" underperformed, a more straightforward setup called the Extensible Prioritization Scheme (EPS) was proposed for HTTP/3. This paper represents the first real-world measurement study into how this new scheme is supported and employed in practice by the three main browser engines and 12 different popular servers and cloud/CDN deployments. We find considerable heterogeneity in overall EPS (sub)feature support and even fundamental differences in approach/philosophy between the stacks. As incorrect prioritization can have a negative effect on (Web) performance metrics, our work not only provides essential insights for browser vendors and server deployments but also offers recommendations for future improvements.

CCS CONCEPTS

• **Networks** → **Application layer protocols**; **Packet scheduling**; • **Information systems** → **Browsers**.

KEYWORDS

QUIC, HTTP, Prioritization, Browsers, Servers, CDN, Resource Loading, Web Performance Optimization

1 INTRODUCTION & RELATED WORK

When loading a Web page's constituent resources, both HTTP/2 (H2) and HTTP/3 (H3) use a single TCP or QUIC connection respectively; each resource is transferred on a single "stream" and many in-flight streams can be multiplexed on the single connection. At any given time, each connection has limited bandwidth available, which is determined by the congestion controller. Therefore, a scheduling algorithm is needed to determine how to distribute bandwidth across active streams. Since not all resources are equally important for Web page performance (e.g., "render blocking" JavaScript (JS) and CSS in the <head> should be loaded before images in the <body>), the scheduler should make sure more important resources are loaded first. Purely fair, round-robin scheduling between the streams is rarely optimal in this context [16, 37]. Since the server often does not have enough context to deduce the resources' importance (e.g., image immediately visible in viewport versus loaded "lazily" at the bottom of the page), the client traditionally determines the resource importance and loading order deemed appropriate and signals this to the server using a "prioritization system".

For example, H2 had a complex "prioritization tree" [1, 37] where each stream was represented by a node in said tree. Bandwidth was distributed across the nodes depending on their position and relationship to their siblings. Nodes could be inserted and moved around at will by clients, leading to a flexible yet difficult to (efficiently) implement server-side system [16, 31]. Consequently, even today there continue to be many inconsistencies and bugs in browser and server implementations of this system [18, 37]. It was subsequently removed in an updated version of the H2 specification [35].

Given the issues with H2's approach, a simpler system was designed for H3 [2]. This Extensible Prioritization Scheme (EPS) [28] does away with the complexities of maintaining a tree and simply assigns two parameters to each resource stream. The first is Urgency (u), an integer $\in [0,7]$ where 0 is highest conceptual "priority" and 7 lowest, with a default of 3. Resources with a low urgency value should *all* be sent before those with higher values, while those with the same urgency should be sent in FIFO order. The second parameter is Incremental (i), a boolean flag (0 or 1), which indicates if a resource can/should share bandwidth with other in-flight resources *with the same urgency*, leading to a sequential (0, default) or incremental (1) send order. As a simplified example, imagine two resources A and B consisting of three "chunks" each. If requested concurrently, Table 2 shows the expected resulting server send order per the EPS RFC [28]. These two parameters can be signaled from client to server in two ways. First, via the "priority" HTTP request header field, mainly intended to set the resource's initial priority. Second, via the "PRIORITY_UPDATE" frame, primarily to allow for

A	B	Send order	Comments
u=1	u=5	AAABBB	Sequential lowest u value first
u=2	u=2	AAABBB	Sequential FIFO
u=3, i	u=2, i	BBBAAA	Sequential lowest u value first
u=4, i	u=4, i	ABABAB	Incremental fair bandwidth sharing
u=4, i	u=4	undefined	Not specified in RFC.

Table 2: Illustrative example of the Extensible Prioritization Scheme with different combinations of u and i

reprioritization after the initial request (e.g., a low priority image halfway down the page becomes higher priority as the user scrolls down).

At the time of writing, it has been two years since the publication of both the H3 and EPS IETF RFCs [2, 28] and H3 now represents a significant share of global traffic (e.g., over 27% of Cloudflare traffic in the first week of April 2024 [3]). As such, we feel it is important to conduct a measurement study across prevalent H3 stacks to gauge their support and use of the EPS system, and to see if it has led to more consistent support than H2's setup (we did not yet aim to assess the impact of EPS on Web performance metrics; this is future work). This text presents our results for both browsers (§2) and servers (§3). We find substantial differences between the implementations, with several still lacking full or lacking partial support, indicating that the new approach might not be as successful in its goals as originally hoped. Our complete test setup and results are publicly available [6, 7, 14].

2 BROWSER OBSERVATIONS

A browser uses heuristics to determine a likely optimal loading order for a Web page's subresources. These heuristics are driven by many inputs, including the resources' types, order of resource tags in the HTML and the use of "loading method" modifiers that developers can employ to explicitly provide hints to the browser (e.g., preload, lazy, async/defer JS [20, 21, 23, 24]). The browser then employs the prioritization system to signal its preferred transmission order to the server.

However, neither H2 nor EPS mandate how the load order should map to the priority signals (e.g., no guidance is given on how to construct a well-performing prioritization tree in H2); this is left as an implementation choice. In practice, this has contributed to browsers choosing divergent ways of using prioritization (for both H2 [37] and H3 (this paper)). This is further complicated by the recent introduction of the fetchpriority API [22]. This allows developers to influence the underlying priority signals indirectly by nudging the browser's heuristical value either slightly up (fetchpriority="high") or conversely down (fetchpriority="low"). This API however does not provide fine-grained control (under EPS, it only influences u but not i) and is inconsistently implemented.

2.1 Experimental Setup

To assess how browsers utilize the EPS, we created multiple HTML pages [14] that load a wide variety of resource types using most available loading methods, including `fetchpriority` modifiers; Table 3 lists the most common combinations.

We evaluated the EPS usage of three popular browser engines (Chromium, WebKit and Gecko) by loading our test pages in their main implementing browsers (Chrome Canary, Safari Technology Preview, Firefox Nightly). The pages were hosted on a London-based Linode VM via the existing aioquic server [6], which we modify slightly to add EPS support to its qlog logging format output [17]. We loaded all pages three times from a Belgian domestic network using a 100Mbps wired link. We repeated the experiments over a period of 1.5 years in three main periods with recent browser versions (YYYY-MM): 2022-12, 2023-08, and 2024-03. Each test produced 9-18 qlogs which we manually analyzed. Since most results remained stable, in §2.2 we report the results from 2024, indicating longitudinal changes where appropriate.

2.2 Browser Results

Our main results are summarized in Table 3. To aid in interpretation, we map each browser's raw urgency values to a 5-tuple of [highest, high, medium, low, lowest] (except for Firefox, which only utilizes 4 values, so we omit "lowest").

EPS parameter usage. Despite EPS only defining two parameters (u and i) the three browsers still use them in very different ways. Firefox never sets the incremental flag (defaulting to 0), while Safari always sets it to 1. Chrome originally never set the flag, but changed this in August 2023 to setting the flag for all resources types except JS, CSS and Fonts [30]. Firefox and Chrome also do not send an explicit signal for resources that would be assigned the EPS default (u=3, i=0), while Safari does. Chrome clearly has a more fine-grained approach, setting distinct urgencies for most resource type and loading method combinations. This is in contrast to Firefox's coarse urgency groupings; Safari is in the middle. Safari's approach is likely suboptimal for Web performance, while Chrome is plausibly the best of the three [13, 37].

EPS signaling method. Despite EPS also only defining two signaling methods (request header and PRIORITY_UPDATE frame), the three browsers still use these differently. Firstly, while Firefox and Safari both signal initial priorities with the header, Chrome instead originally used the frame for this. Interestingly it sent the frame before the actual request, requiring servers to buffer the frame until the request arrived. Since February 2024 [27] Chrome *also* sends the `priority` request header, *in addition to the frame*, which is still sent first. Chrome likely keeps sending the frame to retain interoperability with its own servers that ignore the header (see

Resource (load method)	Chrome	Safari	Firefox
Main resource (HTML)	highest, i	highest, i	highest
Font (preload)	high	medium, i	medium
Font (preload fp@high)	high	high, i	medium
Font (preload fp@low)	low	low, i	medium
Font (font-face)	highest	medium, i	low
JS (preload)	high	high, i	medium
JS (preload fp@high)	high	highest, i	medium
JS (preload fp@low)	low	medium, i	medium
JS (prefetch)	lowest, i	undefined	low
JS (head)	high	high, i	high
JS (head fp@high)	high	highest, i	high
JS (head fp@low)	high	medium, i	high
JS (async)	low	medium, i	medium
JS (async fp@high)	high	high, i	medium
JS (async fp@low)	low	low, i	medium
JS (async blocking)	high	medium, i	medium
JS (defer)	low	high, i	medium
JS (defer fp@high)	high	highest, i	medium
JS (defer fp@low)	low	medium, i	medium
JS (defer blocking)	high	high, i	medium
JS (module)	high [12]	high, i	medium
JS (head inserted)	low	high, i	medium
JS (bottom of body)	medium	high, i	medium
JS (bottom fp@high)	high	highest, i	medium
JS (bottom fp@low)	low	medium, i	medium
CSS (preload)	highest	high, i	high
CSS (preload fp@high)	highest	highest, i	high
CSS (preload fp@low)	high	medium, i	high
CSS (head)	highest	high, i	high
CSS (head fp@high)	highest	highest, i	high
CSS (head fp@low)	high	medium, i	high
CSS (media=print)	lowest	lowest, i	low
CSS (bottom of body)	medium	high, i	high
CSS (bottom fp@high)	high	highest, i	high
CSS (bottom fp@low)	low	medium, i	high
Image (preload)	low, i	low, i	low
Image (preload fp@high)	high, i	medium, i	low
Image (preload fp@low)	low, i	lowest, i	low
Image	low, i	medium, i	low
Image (first 5)	medium, i	medium, i	low
Image (first 5 fp@high)	high, i	high, i	low
Image (first 5 fp@low)	low, i	low, i	low
Image (lazy)	low, i	medium, i	low
Image (lazy fp@high)	high, i	high, i	low
Image (lazy fp@low)	low, i	low, i	low
Fetch	high, i	medium, i	low
Fetch (fp@high)	high, i	high, i	medium
Fetch (fp@low)	low, i	low, i	low
Fetch (manual header)	accepted	accepted	ignored

Table 3: Browser EPS signals for the most common resource types and loading methods. i indicates the incremental flag was true. fp@high|low indicates the `fetchpriority` attribute was explicitly set.

§3.2). Secondly, only Chrome and Safari use the frame for actual stream reprioritization, and only very sparingly. On Chrome, this was only observed to give an image a "high" priority (up from the default "medium") once it was discovered to be in the viewport. On Safari, we only saw frames to reflect the priority bump from `fetchpriority="high"`, and this again only for visible images (other `fetchpriority` modifiers seem to be reflected in the initial header instead).

Heuristics differences. The browsers are also inconsistent in their use of heuristics to determine priority signals. For example, Font files loaded via `@font-face` are highest u in Chrome, medium in Safari, but only low in Firefox (while loading fonts with preload lowers their u in Chrome, but increases it in Firefox). Other discrepancies can be seen for JS files, where Chrome and Firefox treat `async/defer` JS as lower priority than "normal" `<head>` JS, while Safari only lowers the priority for `async`. Similarly, Chrome is more fine-grained in how it handles various JS types in the `<body>`. Finally, while all images are simply assigned a low u by Firefox, Safari and Chrome employ more complex logic. Chrome even changed from assigning all images an initial low u, to marking the first 5 images on a page as medium instead [26] to help improve the Largest Contentful Paint metric.

Fetchpriority impact. While Firefox does not have support for the `fetchpriority` API *yet* [5], Safari applies the API in a very predictable way: where possible, "high" increases the priority one level, while "low" decreases it by one. Chrome however is much more restrictive, often ignoring either high or low and keeping the resource at its "default" urgency instead (e.g., Fetch (fp@high) has no impact), thus arguably diminishing the (already limited) usefulness of the API.

Direct control with fetch(). We tested if it is possible to manually override the EPS "priority" header using the JS `fetch()` API. Originally, Firefox sent two `priority` headers, one with its own heuristics, and one with the manual signals [11]; it now simply ignores the manual override. Safari and Chrome both apply the manual version directly, even if it contains invalid values (e.g., u=42). Notably though, Chrome only changes the header signal; the PRIORITY_UPDATE frame it sends before the request still contains its original values, giving somewhat contradicting info to the server.

Comparison to HTTP/2. Interestingly, both Chrome and Firefox's EPS logic is nearly the opposite of their H2 approach [37]. Firefox has the most complex tree of the three browsers in H2 with complex bandwidth sharing between resources, but employs the simplest EPS signals without the incremental flag. In contrast, Chrome has a purely sequential scheduler in H2, but chooses to set the incremental flag for various resource types in H3. Only Safari uses conceptually similar approaches for the two protocol versions.

3 SERVER OBSERVATIONS

3.1 Experimental Setup

Given the heterogeneity in the browsers' use of EPS, it is evident that servers should implement all or most EPS subfeatures. To assess if servers can indeed support the divergent clients, we ran custom experiments where we request ten resources of the same size (either all at the same time, or slightly staggered), using a variety of u and i values, communicated via the HTTP "priority" header field and/or PRIORITY_UPDATE frame, to assess specific subfeature support. Table 4 details our experimental categories (A-G). For each category, we permutate header vs frame ordering, delays between requests, and when signals are sent, to properly verify the underlying logic in a large variety of conditions.

The design of the experiments had to account for QUIC's non-deterministic ordering between individual streams, as QUIC tries to prevent inter-stream head-of-line-blocking [8, 16] (i.e., by design, messages X and Y sent on different streams in order X/Y, may well arrive at and be applied by the receiver in reversed Y/X order). This is an issue since the EPS header is sent on a resource's "request stream", while the PRIORITY_UPDATE frame is sent on a separate "control stream". As such, these signals can conceptually become inverted, leading to inconsistent results. To combat and detect this problem, we employed various delays (e.g., 200ms, 100ms, 20ms, 0ms) between sending the frame and the request headers.

As we need full control and determinism in both the timing and contents of the prioritization signals, we were unable to utilize existing Web browsers directly. Instead, we employ the full-featured [33] and extensible aioquic client, which we modify [6] to add support for EPS client-side signaling and to extend the verbosity of its qlog logging format output [17].

We manually selected 12 server stacks to test, based on perceived popularity and prevalence on the Web [38], as well as expected maturity [33]. This includes the major CDNs (Akamai, Cloudflare and Fastly), distributed clouds (QUIC.cloud, Google, Amazon), popular (resource) hosting services (jsDelivr, Shopify), and off-the-shelf open source packages (NGINX, Caddy). For all but the final two, we manually identified multiple publicly accessible H3-capable [19] URLs (e.g., from company homepages or from other test setups [4]), all between 102KB - 3.1MB in size, and of various types (JS, CSS, images). We disregarded some deployments (e.g., Microsoft Azure) as we could not find public H3-capable URLs for them. For NGINX and Caddy, we hosted their latest docker images on a London-based Linode VM, copying over one of the other deployments' resources to use as the experimental object.

We conducted all experiments from a Belgian University network using a symmetrical 1Gbps wired link. We repeated experiments six times over a period of seven weeks

ID	Experiment	Logic and goals
A	nothing	No EPS signals are sent. Test default scheduling behavior.
B	incremental	All resources have the same urgency and incremental set to 1. Test bandwidth sharing.
C	late high prio	Resource 5 and 6 are requested after the others with a lower urgency. Test urgency precedence.
D	late high prio inc.	Same as late high prio with incremental resources. Test sequential to incremental changeover.
E	mixed bucket	Both incremental and non-incremental in the same urgency. Test the unspecified edge case in Table 2.
F	mixed method	Send both request header and `PRIORITY_UPDATE` frame with conflicting info. Test signaling method support.
G	reprioritization	Send `PRIORITY_UPDATE` frame after 50ms with lower urgency value (higher priority). Test reprioritization.

Table 4: Different server experiments request ten resources on the same connection with varying EPS signals.

in 2024 (MM-DD): 02-14, 02-21, 02-29, 03-28 (only NGINX and Caddy), 04-03, and 04-05. Each test run produced 1276 to 1680 qlog traces (experiments F and G were added later, as were NGINX v1.25.4 and Caddy v2.7.6, and only tested during the final three runs). We manually analyzed the qlogs using the qvis tool suite [15], longitudinally comparing results both across experiments for the same server, as well as across servers for the same experiment. As we incrementally added experiment categories and new server deployments over time, we derive a server's core behavior for a specific experiment primarily from the first obtained result thereof. We then validated our assessments with the final full runs of all experiments over all servers in April 2024 [7].

3.2 Server Results

Table 1 summarizes the results for the main EPS subfeatures that showed clear discrepancies across implementations. While we reached out to multiple stack engineers for comments, most declined. We have included other replies where appropriate.

Full support. Among all the global server deployments tested, only Fastly and QUIC.cloud demonstrated full support for EPS[1]. However, they exhibit minor differences in how streams with the incremental flag set to true are scheduled. Fastly adopts a fine-grained round-robin (RR) approach, alternating streams with each QUIC packet, while QUIC.cloud transmits chunks of 1 to 90 packets before switching streams. It is unclear if these chunk sizes are consciously chosen or if they are a consequence of, for example, the buffering logic.

Partial support. Akamai, Cloudflare and Google each support a different subset of EPS features. Firstly, in terms of signaling methods, Akamai and Google both lack support for the HTTP "priority" header field and only adhere to signals received in a `PRIORITY_UPDATE` frame, while Cloudflare (largely) adheres to both. This is likely because Google's Chrome browser originally only sent the frame and not the header (see §2.2). Consequently, as Akamai utilizes a modified version of Google's stack to implement QUIC and H3 on its CDN [38], it is probably constrained by their feature

set. This is an important limitation, as only Chromium-based browsers utilize the frame to signal (initial) priorities. As such, both Akamai and Google will not (fully) adhere to Firefox's and Safari's EPS signaling, instead falling back to their default schedulers.

Secondly, in terms of this default scheduling behavior, we see that in the absence of EPS signals, Akamai and Cloudflare correctly opt for a sequential FIFO send order (u=3 and i=0, the EPS defaults), but that Google instead employs an incremental scheduler. The latter being suboptimal [16, 37] is why the default recommended behavior switched from incremental in H2's "prioritization tree" [1] to sequential in EPS (`i` should be false (0) by default [28]). It is interesting that Akamai does deviate from its underlying Google-based stack for this aspect, indicating this discrepancy was noted.

Thirdly, we find that the incremental parameter is simply ignored by both Akamai and Google. This means that a browser like Safari cannot override Akamai's default sequential behavior. Similarly, Google's servers cannot be made to send resources sequentially, contrasting with Google Chrome's many requests for sequential loads (see Table 3). While this is less of a problem for Akamai as it applies proper defaults, it can have noticeable (Web) performance implications for Google-hosted properties. Note that the problem is somewhat mitigated because Google does adhere to the urgency parameter and so higher priority resources will still be sent before lower priority ones; they are just round-robined with other, equally high priority resources.

Fourthly, considering reprioritization, only Cloudflare lacks support for this functionality. Specifically, it disregards all `PRIORITY_UPDATE` frames received after the HTTP request headers, regardless of whether a priority has been specified before (either by the "priority" request header or an earlier `PRIORITY_UPDATE` frame). In contrast, all other surveyed EPS-capable deployments appropriately update the stream's priority in response to subsequent frames as expected. While reprioritization is not (yet) extensively used by browsers for Web page loading (see §2.2) and cannot be manually triggered by developers yet, it can be an important aspect of other use cases (e.g., HTTP-based video streaming [10, 29]). When discussing this deficiency with Cloudflare engineers, they highlighted "the well-documented race conditions with reprioritization that can affect its usefulness for real-world

---

[1]In two Fastly experiments, priority signals were ignored and random stream sequencing occurred, but only in one URL and not reproduced in later data points from April 2024.

workloads [31]" as a reason for not supporting the subfeature. They expressed interest in "seeing data demonstrating strong potential benefits to offset the additional complexity that supporting reprioritization would add".

Indirect support. We investigated two deployments that offer services on top of/via another deployment. Concretely, jsDelivr (a service to load popular JS libraries) is backed by among others Fastly and Cloudflare [9], while the Shopify platform is built on top of Cloudflare [34]. As such, we expected them to exhibit the same EPS behavior as their underlying deployments, but this only seems to hold for jsDelivr's Fastly backend. Strangely, the Cloudflare backend for both jsDelivr and Shopify behaves in a radically different fashion than when tested directly, seemingly offering no support for EPS at all, only inheriting the default sequential scheduling behavior from the parent stack. This is especially concerning for jsDelivr, as it somewhat undermines the goal of optimized delivery through inconsistent and unpredictable resource loading behavior. When discussing this with Cloudflare engineers, they stated that "H3 prioritization is still in the process of being rolled out to all customers", and confirmed the intent to use sequential scheduling by default.

No support. As far as we can determine, neither Amazon CloudFront, nor NGINX or Caddy implement any support for EPS. Furthermore, like Google, they implement the opposite of the recommended default sequential behavior and send all resources incrementally. Amazon CloudFront and Caddy furthermore utilize per-packet Round-Robin logic, while NGINX chunks data by 64 KB. Asked for comments, NGINX and Caddy developers indicated that EPS support is on the roadmap, but not currently prioritized [7, 36].

Other common behaviors. As per the specification, all implementations supporting PRIORITY_UPDATE frames buffer them before stream opening. This is important for Chrome, which sends the frame before the headers (§2.2). Our observations also indicate that all servers correctly maintain FIFO send order within the same urgency "bucket" (if i=0).

For experiment E, which explored the unspecified edge case (see Table 2), unexpectedly consistent behavior was observed across all EPS-capable stacks. *All* non-incremental resources are first dispatched sequentially in FIFO order, before the fair multiplexing of incremental resources. However, this systematic approach can lead to resource starvation if new non-incremental streams keep arriving during incremental resource transmission (though we did not verify this during our experiments).

4 DISCUSSION & RECOMMENDATIONS

Despite EPS' more straightforward approach compared to H2's prioritization tree, our results still show considerable heterogeneity in how different servers and browsers support and use its (sub)features in practice. These discrepancies can likely be attributed to several factors, of which we list a few. Firstly, the EPS specification *leaves a lot of (intentional) flexibility to implementers* (most guidelines are a SHOULD rather than a MUST), which is clearly exercised to its full extent. Secondly, the decision to make EPS its own specification decoupled from the main H3 RFC, might lead some implementers to *consider it a recommended, yet fundamentally optional* feature. Thirdly, despite the simplicity of the signals, *implementing the underlying priority-driven scheduler remains a complex task* (as suggested by Cloudflare engineers). Finally, there is the lack of clear guidance (for both H2 and EPS) on how to employ client-side load order heuristics and how these should drive the prioritization signals. That the browsers have decided to fill this void in such divergent ways (even between their H2 and H3 stacks, with Firefox and Chrome evolving in nearly opposite directions), to us is *a strong indicator that (some) browser vendors have not tested the performance impact of their approaches in the wild* and also do not know the optimal approach.

While actually measuring the impact of these differences on Web performance metrics was a non-goal of this study, extrapolating previous results for both H2 and H3 indicates that end-users might face suboptimal and highly volatile experiences [16, 25, 32, 37]. As such, we feel that further research into loading heuristics and prioritization strategies is essential to improving long-term end-user experiences. Additionally, the community should explore *allowing more manual control over priority signaling through developer APIs* (e.g., by extending fetchpriority to allow changing incrementality or to manually trigger reprioritization). This will allow developers to not only work around browser heuristic deficiencies, but also enables more complex Web applications (e.g., fine-grained media streaming for AR/VR use cases). For this to work however, it is *essential that servers implement the EPS and all its subfeatures* in their entirety. This is true now, but also in the future, if we plan to actually put the "Extensible" aspects of EPS into practice by adding additional parameters and features.

As such, to make EPS and its evolution easier to test and evaluate going forward, we suggest that major deployments *offer a range of publicly available realistic test resources of standard size and behavior*. This would ensure consistent overall testing conditions for both researchers and developers, allowing for a more accurate simulation of real-world scenarios. This will enable further thorough validation of all HTTP/3 features and implementations, leading to improved standards and better alignment with the needs of modern Web applications and users. As a first step, we open source all artifacts for this work as well [6, 7, 14].

REFERENCES

[1] Mike Belshe, Roberto Peon, and Martin Thomson. 2015. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540. https://doi.org/10.17487/RFC7540

[2] Mike Bishop. 2022. HTTP/3. RFC 9114. https://doi.org/10.17487/RFC9114

[3] Cloudflare. 2024. Cloudflare Radar: Adoption and Usage. https://radar.cloudflare.com/adoption-and-usage.

[4] Cloudflare. 2024. Cloudflare radar performance testing endpoint script. https://performance.radar.cloudflare.com/beacon.js.

[5] Valentin Gosu. 2024. Bugzilla: [meta] Fetch Priority (was Priority Hints). https://bugzilla.mozilla.org/show_bug.cgi?id=1797715.

[6] Joris Herbots and Robin Marx. 2024. Aioquic fork for testing HTTP/3 prioritization. https://github.com/http3-prioritization/aioquic.

[7] Joris Herbots and Robin Marx. 2024. *HTTP/3's Extensible Prioritization Scheme in the Wild - Experiment Results.* https://doi.org/10.5281/zenodo.12544401

[8] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[9] jsDelivr. 2024. jsDelivr Documentation. https://www.jsdelivr.com/documentation#Multi-CDN.

[10] Daniele Lorenzi, Minh Nguyen, Farzad Tashtarian, Simone Milani, Hermann Hellwagner, and Christian Timmerer. 2021. Days of future past: an optimization-based adaptive bitrate algorithm over HTTP/3. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC* (Virtual Event, Germany) *(EPIQ '21)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3488660.3493802

[11] Robin Marx. 2023. Bugzilla: Double RFC9218 HTTP Priority header field when set via fetch(). https://bugzilla.mozilla.org/show_bug.cgi?id=1809403.

[12] Robin Marx. 2023. Chromium: JavaScript modules are loaded with higher HTTP priority than defer scripts. https://issues.chromium.org/issues/40279703.

[13] Robin Marx. 2023. Resource Loading at the Cutting Edge. https://perfnow.nl/speakers#robin.

[14] Robin Marx. 2024. Prioritization Test Pages. https://github.com/http3-prioritization/prioritization-test-page.

[15] Robin Marx. 2024. qvis: tools and visualizations for QUIC and HTTP/3. https://qvis.quictools.info/.

[16] Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. 2019. Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*. INSTICC, SciTePress, 130–143. https://doi.org/10.5220/0008191701300143

[17] Robin Marx, Luca Niccolini, Marten Seemann, and Lucas Pardue. 2024. *Main logging schema for qlog.* Internet-Draft draft-ietf-quic-qlog-main-schema-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/08/ Work in Progress.

[18] Robin Marx, Maxime Piraux, Peter Quax, and Wim Lamotte. 2020. Debugging Modern Web Protocols with qlog. In *Proceedings of the Applied Networking Research Workshop (ANRW 2020)*. https://qlog.edm.uhasselt.be/anrw/

[19] MDN. 2024. MDN Web Docs - Alt-svc. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Alt-Svc.

[20] MDN. 2024. MDN Web Docs - async. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script#async.

[21] MDN. 2024. MDN Web Docs - defer. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script#defer.

[22] MDN. 2024. MDN Web Docs - fetchPriority. https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement/fetchPriority.

[23] MDN. 2024. MDN Web Docs - Lazy loading. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img#loading.

[24] MDN. 2024. MDN Web Docs - rel=preload. https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/rel/preload.

[25] Patrick Meenan. 2019. Better HTTP/2 Prioritization for a Faster Web. https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web.

[26] Patrick Meenan. 2023. Chromium: Schedule load of first N images sooner. https://issues.chromium.org/issues/40263406.

[27] Patrick Meenan. 2024. Chromium: Add priority: request header. https://issues.chromium.org/issues/40252001.

[28] Kazuho Oku and Lucas Pardue. 2022. Extensible Prioritization Scheme for HTTP. RFC 9218. https://doi.org/10.17487/RFC9218

[29] Roger Pantos. 2023. [Hls-interest] iOS 17 and Low-Latency HLS. https://mailarchive.ietf.org/arch/msg/hls-interest/RcZ2SG8Sz_zZEcjWnDKzcM_-TJk/.

[30] Lucas Pardue. 2023. Chromium: HTTP/3 download priority value causes serialized downloads. https://issues.chromium.org/issues/40864006.

[31] Netflix Research. 2019. CVE-2019-9513: HTTP/2 resource loop attack. https://nvd.nist.gov/vuln/detail/CVE-2019-9513.

[32] Constantin Sander, Ike Kunze, and Klaus Wehrle. 2022. Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance.. In *Proceedings of the 6th Network Traffic Measurement and Analysis Conference (TMA Conference 2022)*. IFIP.

[33] Marten Seemann. 2024. QUIC Interop Runner. https://interop.seemann.io/.

[34] Shopify. 2024. Shopify Documentation. https://shopify.dev/docs/themes/best-practices/performance/platform#shopify-cdn.

[35] Martin Thomson and Cory Benfield. 2022. HTTP/2. RFC 9113. https://doi.org/10.17487/RFC9113

[36] Chanh Tran. 2023. quic-go support for http3: RFC 9218 - Extensible Priority Scheme. https://github.com/quic-go/quic-go/issues/3470.

[37] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. 2018. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. ACM, 1755–1764. https://doi.org/10.1145/3178876.3186181

[38] Johannes Zirngibl, Florian Gebauer, Patrick Sattler, Markus Sosnowski, and Georg Carle. 2024. QUIC Hunter: Finding QUIC Deployments and Identifying Server Libraries Across the Internet. In *Passive and Active Measurement*, Philipp Richter, Vaibhav Bajpai, and Esteban Carisimo (Eds.). Springer Nature Switzerland, Cham, 273–290.