Web Performance Characteristics of HTTP/2
and comparison to HTTP/1.1

Robin Marx, Maarten Wijnants, Peter Quax, Axel Faes, and Wim Lamotte

UHasselt-tUL-imec, EDM, Hasselt, Belgium {first.last}@uhasselt.be

Abstract. The HTTP/1.1 protocol has long been a staple on the web, for both pages and apps. However, it has started to show its age, especially with regard to page load performance and the overhead it entails due to its use of multiple underlying connections. Its successor, the newly standardized HTTP/2, aims to improve the protocol's performance and reduce its overhead by (1) multiplexing multiple resources over a single TCP connection, (2) by using advanced prioritization strategies and by introducing new features such as (3) Server Push and (4) HPACK header compression. This work provides an in-depth overview of these four HTTP/2 performance aspects, discussing both synthetic and realistic experiments, to determine the gains HTTP/2 can provide in comparison to HTTP/1.1 in various settings. We find that the single multiplexed connection can actually become a significant performance bottleneck in poor network conditions with high packet loss and that HTTP/2 rarely improves much on HTTP/1.1, except in terms of reduced overhead. Prioritization strategies, Server Push and HPACK compression are found to have a relatively limited impact on web performance, but together with other observed HTTP/2 performance problems this could also be due to faulty current implementations, of which we have discovered various examples.

1 Introduction

As the web grows more mature in terms of availability and features, so does its complexity. Websites have evolved from collections of simple individual document pages into complex user experiences and even full "apps". Even though internet connection speeds have also been steadily increasing in this time frame, the traditional internet protocols HTTP/1.1 and TCP have struggled to keep up with these developments and are in many cases unable to provide fast web page load performance[12]. This is detrimental to the overall viability of the web platform for complex use cases such as e-commerce, since a multitude of studies has shown that web performance is a core tenet in ensuring user satisfaction [9, 13].

Most of the performance problems with HTTP/1.1 stem from the fundamental limitation to only request a single resource per underlying TCP connection at the same time. This means that slow or large resources can delay others, which is called "Head-Of-Line (HOL) blocking". As modern websites consist of tens to even hundreds of individual resources, browsers typically open several parallel HTTP (and thus also TCP) connections (up to six per hostname and 30 in total in most browser implementations). However, these heavily

parallelized setups induce large additional overheads (e.g., in terms of server-side connection count) while not providing extensive performance benefits in the face of ever more complex websites [12] (see also Sect. 4). Note that while the HTTP/1.1 specification does include the *pipelining* technique (which does allow multiple requests to be queued on a connection), it is not enabled by default in the major modern browsers due to various practical issues [14].

In order to tackle these challenges, the new HTTP/2 protocol [3] (h2) was standardized in 2016, after evolving from Google's SPDY protocol since 2009. While keeping full backwards compatibility with the semantics of the HTTP/1.1 protocol (h1) (e.g., types of headers, verbs and overall setup), h2 nevertheless introduces many low-level changes, primarily with the goal of improving web page load performance. For example, all h2 traffic is ideally sent over a single TCP connection (making use of multiplexing and inter-resource prioritization algorithms to eliminate HOL blocking), there is support for server-initiated traffic (Server Push) and headers are heavily compressed in the HPACK format. The details of these aspects are discussed in Sect. 4 to 7.

In theory, h2 should solve most of the problems of h1 and improve web page load times by up to 50% [12]. In practice however, the gains from h2 are limited by other factors and implementation details. Firstly, the use of a single TCP connection introduces a potential single-point-of-failure when high packet loss is present (and so it might actually be better to also use multiple TCP connections for h2). Secondly, correctly multiplexing multiple resources over this connection is heavily dependent on the used resource prioritization scheme and the interleaving of resource chunks might introduce its own implementation overhead as these chunks need to be aggregated before processing. Finally, complex inter-dependencies between resources and late resource discovery might also lessen the gains from h2 [22]. The fact that h2 is not a simple drop-in replacement with consistently better performance than h1 is also clear from previous studies, which often find cases where h2 is significantly slower than h1 (see Sect. 2).

In this text, we continue the groundwork from our previous publications [17, 18]. We discuss four HTTP/2 performance-related aspects and test their impact, both in synthetic and realistic test scenarios, in comparison with HTTP/1.1's performance (Sect. 4 to 8). Our main contributions are as follows:

- We extend the **Speeder framework for web performance measurement** [18], combining a large number of off-the-shelf software packages to provide various test setup permutations, leading to a broad basis for comparison and interpretation of experimental results.
- We compare h2 to h1 in both synthetic and realistic experiments and find that **while h2 rarely significantly improves performance over h1, it is also rarely much slower**. Additionally, in most cases, bad network conditions do not seem to impact h2 much more than they impact h1. Using multiple parallel TCP connections can help both h2 and h1. Prioritization, Server Push and HPACK compression seem to contribute only sparingly to page load time improvements.
- We find that many current **h2 implementations (both on the server and browser sides) are not yet fully mature** and that some (default) implementations lead to sub-optimal performance, especially concerning the time it takes to start rendering the web page.

2 Related Work

Various authors have published works comparing the performance of `h2` and its predecessor SPDY to `h1`.

In "How Speedy is SPDY?" [27] the authors employ isolated test cases to better assess the impact of various parameters (latency, throughput, loss rate, initial TCP window, number of objects and object sizes). They observe that SPDY incurs performance penalties when packet loss is high (mainly due to the single underlying TCP connection) but helps for many small objects, as well as for many large objects when the network is fast. For real pages, they find that SPDY improves page load performance for up to 80% of pages under low throughput conditions, but only 55% of pages under high bandwidth.

"Towards a SPDY'ier Mobile Web?" [8] performs an analysis of SPDY over a variety of real networks and finds that underlying cellular protocols can have a profound impact on its performance. For 3G, SPDY performed on a par with `h1`, with LTE showing slight improvements over `h1`. A faster 802.11g network did yield improvements of 4% to 56%. They further conclude that using multiple concurrent TCP connections does not help SPDY.

"Is The Web HTTP/2 Yet?" [26] measures page load performance by loading real websites over real networks from their original origin servers. They find that most websites distribute their resources over multiple backend hosts and as such use `h2` over multiple concurrent connections, which "makes `h2` more resilient to packet loss and jitter". They conclude that 80% of the observed pages perform better over `h2` than over `h1` and that `h2`'s advantage grows in mobile networks. The remaining 20% of the pages suffer a loss of performance.

"HTTP/2 Performance in Cellular Networks" [10] introduces a novel network emulation technique based on measurements from real cellular networks. They use this technique to specifically assess the performance impact of using multiple concurrent TCP connections for `h2`. They find that `h2` performs well for pages with large amounts of small and medium sized objects, but suffers from higher packet loss and larger file sizes. They demonstrate that `h2` performance can be improved by using multiple connections, though it will not always reach parity with `h1`.

"HTTP/1.1 Pipelining vs HTTP2 In-The-Clear: Performance Comparison" [7] compares the cleartext (non-secure) versions of `h1` and `h2` (`h1c` and `h2c` respectively) (even though `h2c` is not currently supported by any of the main browsers, see Sect. 3). They disregard browser computational overhead and find that on average `h2c` is 15% faster than `h1c` and "`h2c` is more resilient to packet loss than `h1c`".

Additional academic work [15] found that for a packet loss of 2%, "`h2` is completely defeated by `h1`" and that even naive Server Push schemes can yield up to 26% improvements. Others [25] conclude that `h2` is mostly interesting for websites with large amounts of images, showing up to a 48% decrease in page load time, with an additional 10% when using `h2` Server Push, and that `h2` is resilient to higher latencies but not to packet loss. Further experiments [23] indicate that `h2` Server Push seems to improve page load times under almost all circumstances. Finally, Carlucci et al. [6] state that packet loss has a very high impact on SPDY, amounting to a 120% increase of page load time compared to `h1` on a high bandwidth network.

Content Delivery Network (CDN) companies have also measured `h2` performance on their networks. Gooding et al. [11] from Akamai find that using multiple TCP connections is

best avoided for critical resources on `h2`. A presentation by Fastly [2] states that `h2` mostly outperforms `h1` on fast networks, but loses on networks with higher packet loss.

Our review of related work clearly shows that the current state of the art is often contradictory in its conclusions regarding `h2` performance. It is not clear whether using multiple TCP connections provides significant benefits, whether `h2` is resilient to poor network conditions and what degrees of improvement developers might expect when migrating from `h1` to `h2`.

In this work, we try to assess why these contradictions exist by running a wide variety of tests on several heterogeneous test setups (see Sect. 3). We look at four performance-related aspects of `h2`, first in isolation to assess their relative impacts and then in combination to evaluate the protocol's impact on typical realistic web page loads. We are thus able to confirm some of the findings reported by the related work, while showing that many of the contradictory findings can be attributed to inefficiencies in current `h2` implementations.

3   Experimental Setup with the Speeder Framework

As discussed in Sect. 2, there are many cases of contradictory results regarding the performance of `h2`. As we suspect that one of the reasons for these discrepancies are differences in the underlying `h2` implementations (both client/browser-side and server-side) and utilized test configurations, we aim to employ as many test setup permutations as possible. We argue that if the results show similar trends across all or a large part of the test setups, they are most likely attributable to the protocol itself. If however the results vary widely, they are typically dependent on specific implementations.

In order to obtain these diverse test setup permutations, we use the Speeder framework for web performance measurement, previously introduced in [18]. Speeder provides pre-installed versions of a large amount of existing software packages (e.g., servers, browsers, network emulation tools, automated testing tools) that can be freely coupled to each other through the use of Docker containers[1]. Users simply need to select the desired setup permutations and the framework collects and aggregates a multitude of key metrics. Users can then utilize various visualization tools to compare the results.

For this work, we have expanded Speeder in a variety of ways. We have upgraded the supported browser versions of Chrome and Firefox to v60 and v54 respectively, updated webpagetest[2] to v3 and now also support the H2O webserver [3], which was heavily optimized for `h2` from the ground up. We have also created and integrated the H2Vis visualization tool. H2Vis directly takes the low-level *.pcap* packet capture files recorded during a test run (using `tcpdump`[4]) to produce a number of insightful graphical representations. For example, we can plot both TCP-level and `h2`-level packets on a graphical timeline to help verify how data is actually sent by the `h2` server, how the various `h2` streams are interleaved on a single TCP connection (see Sect. 4 and 5) and what the practical impact of packet loss is on the connection. Additionally, support for graphically visualizing the generated `h2` priority dependency trees (see Sect. 5) allows us to quickly assess the impact of various prioritization

---

[1] https://www.docker.com/

[2] https://www.webpagetest.org/

[3] https://h2o.examp1e.net/

[4] http://www.tcpdump.org/

Table 1. Software, metrics and visualizations supported in the Speeder framework (August 2017).

Protocols	HTTP/1.1 (cleartext), HTTPS/1.1, HTTPS/2
Browsers	Chrome (v51 - v60), Firefox (v45 - v54)
Test drivers	Sitespeed.io (v3), Webpagetest (v3.0)
Servers	Apache (v2.4.20), NGINX (v1.10), NodeJS (v6.2.1), H2O (v2.1)
Network	- `DUMMYNET` (cable and cellular) (provided by Webpagetest) - fixed `TC NETEM` (cable and cellular) - dynamic `TC NETEM` (cellular) [10]
Metrics	All Navigation Timing values [28], SpeedIndex [19], firstPaint, visualComplete, other Webpagetest metrics [20]
Visualizations	Packet timeline (TCP and `h2`), `h2` priority dependency trees. Boxplots, linegraphs and CDFs of recorded metrics

strategies in use by browsers. Table 1 provides an overview of the features of the Speeder framework at the time of writing.

Unless indicated otherwise, the results in this work were generated in an experimental setup using NGINX v1.10 as web server and Google Chrome v54 as browser, driven by Webpagetest v2.19 and the dynamic cellular network model. This dynamic network model uses previous work [10] which introduced a model based on real-life cellular network observations. The model has six levels of "user experience (UX)": NoLoss, Good, Fair, Passable, Poor and VeryPoor. Each UX level contains a time series of values for bandwidth, latency and loss. The model changes these parameters at 70ms intervals to simulate a real network. This implies, for example, that applied packet loss is more bursty than with the fixed model. For details, please see [10] or the original source code[5].

Our results will be presented using two distinct metrics, namely `loadEventEnd` and `SpeedIndex`. `loadEventEnd` from the Navigation Timing API [28] gives a good indication of the total time (in milliseconds (ms)) a page needed to load, but does not say anything about how progressively it was rendered in that time frame. In other words: a page that stays empty for 5s and only renders content during the last 0.6s (page A) will have a better observed `loadEventEnd` performance than a page that finishes loading at 7.5s, but that had its main content drawn by 2.5s (page B), while the latter arguably yields the better end-user experience. In order to capture the degree to which the page loads progressively, Google introduced the `SpeedIndex` metric [19], which measures how fast a page renders, not just loads. Inconsistencies between `loadEventEnd` and `SpeedIndex` results can indicate that a resource was fast to load but slow to have visual impact. Like `loadEventEnd`, `SpeedIndex` is expressed in ms and so for both metrics lower values mean better performance.

Finally, we performed most of our tests using three versions of the HTTP protocol: the secure HTTPS/2 (`h2s`) and HTTPS/1.1 (`h1s`) and also the unencrypted HTTP/1.1 (`h1c`), because many websites still use this "cleartext" version. We do not include `h2c`, as modern

---

[5] https://github.com/akamai/cell-emulation-util

browsers choose to only support `h2s` for security reasons. Note additionally that switching from `h1c` to a secure setup (either `h1s` or `h2s`) could have its own performance impact as TLS connections typically require additional network round-trips to setup. In the following sections, we will use `h2` to refer to `h2s`, and `h1` refers to both `h1s` and `h1c`.

Most of our graphs will show `loadEventEnd` on the Y-axis. Individual data points will typically represent aggregates (e.g., median, average) of 10 to 100 page loads. Each experiment was repeated at least five times. Unexpected datapoints and anomalies across runs were analyzed further by manually checking the collected output of individual page loads (e.g., screenshots/videos, .har files, waterfall charts, .pcap files). The line plots will show the median values under Good network conditions, as do the Cumulative Distribution Functions (CDFs). The box plots will show the median as a horizontal bar and the average as a black square dot, along with the 25th and 75th percentiles and min and max values as the whiskers. Some box plots use a logarithmic scale on the Y-axis to allow for large values. To be able to compare our results using the SpeedIndex metric, we make sure our loaded resources have a strong visual impact on the visible "above the fold" part of the website.

Some of our results were obtained using hand-crafted experiments on synthetic data. These test cases are intended to demystify the underlying behavior of the protocols and their implementations, and so are often not entirely realistic or involve extreme circumstances. However, most of our results were obtained using more realistic data based on existing websites. We expect that, compared to the experiments on synthetic pages, these test cases will show similar but more nuanced results and trends.

Readers are encouraged to review our full dataset (which encompasses results not presented in this paper (e.g., for other browser/server combinations and test pages)), setup details and source code via `https://speeder.edm.uhasselt.be`.

4   Multiplexing Over a Single TCP Connection

4.1   Background

One of the major downsides of HTTP/1.1 is that it only allows a single resource to be requested and sent on an individual TCP connection at any given time. As such, the problem of Head-Of-Line (HOL) blocking is introduced, where the delivery of the initial resource(s) can block later resources (e.g., if the initial resource is very slow to be generated, is very small (so it does not take up the full possible bandwidth) or is very large). To work around this problem, modern browsers typically open up to six parallel HTTP/TCP connections to a single origin server. This way, even if one or more of the connections suffer from HOL blocking, the others can serve key resources as soon as possible. In tandem, developers have adopted the practice of merging several smaller resources into larger files, a practice called "concatenation" or "bundling". This approach causes the number of individual resources to go down and with them the number of needed TCP connections and HTTP requests. On the other hand, concatenation has the adverse effect that it reduces the fine-grained cacheability of individual, smaller resources.

Another `h1` best practice is that of "hostname sharding". Web developers will typically distribute their resources over a number of individual servers with different hostnames (for example by using a CDN). The browser will open up to six connections per hostname,

resulting in a total of 17 - 60 parallel `h1` connections across all hostnames[6] per page load. This leads to massively parallel page loads, but also introduces significant overheads on the server side in order to support this large amount of connections and their state management. The downsides of both concatenation and sharding (reduced cacheability and higher overhead, respectively) do not always outweigh their observed page load performance benefits [12].

In response, HTTP/2 tries to solve the root issue of HOL blocking by delivering multiple resources over a single TCP connection concurrently, using multiplexing. In practice, smaller chunks of individual resources are encapsulated in conceptual "streams" and are then interleaved on the single connection. Section 5 discusses in detail how `h2` decides on the resources' interleaving order with a priority-based dependency tree. The HTTP/2 specification [3] actively encourages this single connection setup. For example, it includes a mechanism for coalescing requested HTTP connections to separate hostnames onto a single TCP connection if the hosts use the same HTTPS certificate and resolve to the same IP address, this way effectively "undoing" a typical sharded `h1` setup. HTTP/2 Server Push can also only be used for resources on the same domain (see Sect. 6).

In theory, `h2`'s approach should render the `h1` best practices of concatenation and sharding obsolete [12]. In practice however, the single TCP connection might also be more susceptible to adverse network conditions than `h1`'s parallel approach. With `h1`, if one or more of the parallel connections would incur packet loss or high jitter, the possibility exists that the other connections would remain unimpaired. With just a single `h2` connection, all resources will be impacted when the network deteriorates. In effect, this could introduce transport-layer HOL blocking, induced by TCP's guarantee of in-order delivery combined with re-transmits when packet loss is present [21]. If the impact of packet loss is significant, `h2` might in practice also benefit from sharding on multiple connections (see Sect. 4.2).

4.2 Head-of-Line Blocking in Practice with Images

In order to assess the impact of concatenation and sharding on both `h1` and `h2` page load performance in varying network conditions, the experiments in Fig. 1 compare three cases: (left) concatenated into a single resource on one host, (middle) non-concatenated on one host, (right) non-concatenated on four hosts ("sharded"). In practice, for the sharded case, for `h1` the browser will open the maximum amount of connections (24, six per hostname) and a single connection per hostname for `h2` (four in our case). The observed `h1` connections are all configured with `Keep-Alive` and do not use pipelining.

In `h1` the problem of HOL blocking is most apparent when trying to download many smaller files, as browsers only open six parallel connections. Since these smaller files do not fully take up the available bandwidth and each individual resource request requires a full Round-Trip-Time (RTT) delay, this overhead quickly adds up. For this, we consider three experiments in Fig. 1: (a) a large number (i.e., 380) of small files, (b) a medium number (i.e., 42) of medium sized files and (c) a medium number (i.e., 30) of large files. We choose images because they typically incur a low processing overhead from the browser. We look at more complex JavaScript/CSS cases in the next section.

For Fig. 1(a) we observe that `h2` significantly outperforms `h1` when there is no concatenation (middle), but that using a single concatenated image largely reduces `h2`'s benefit

---
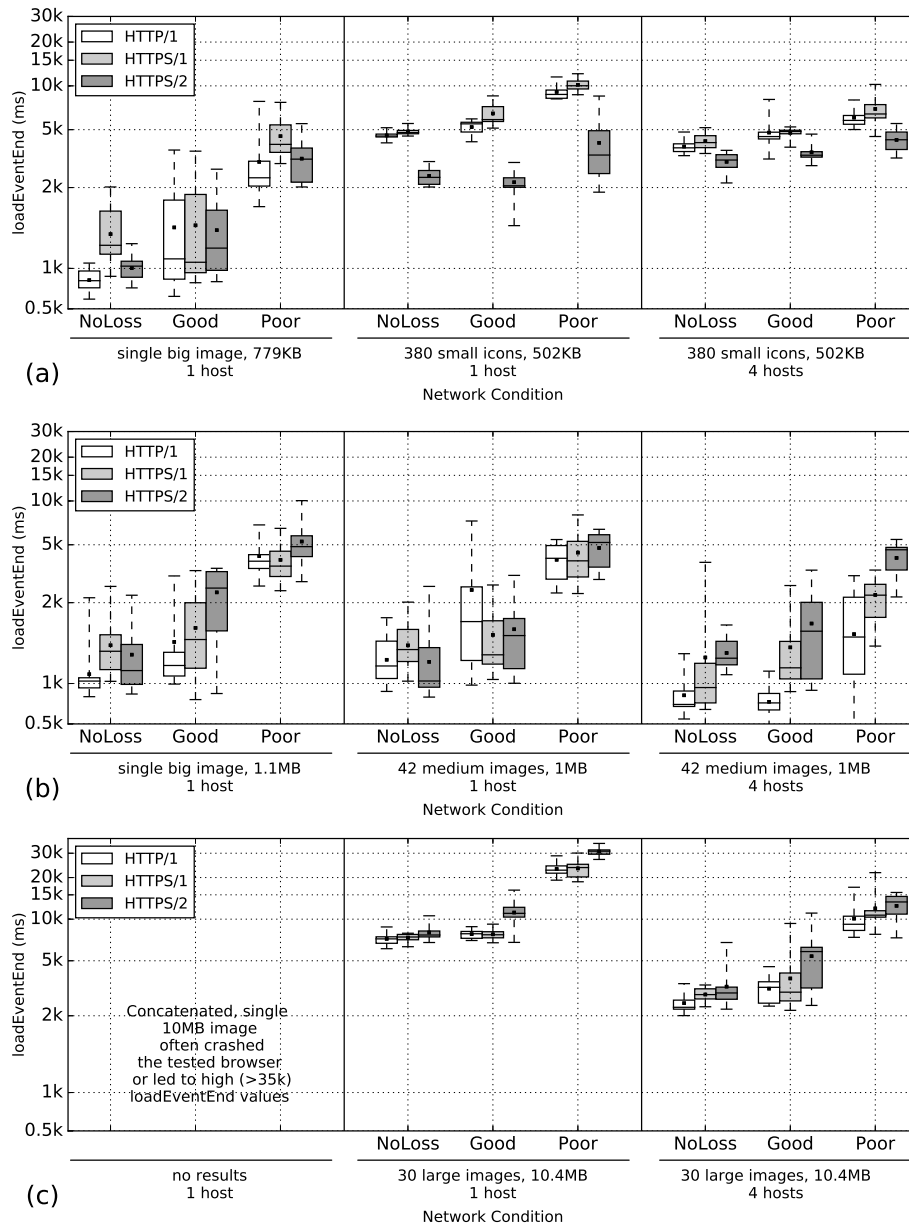
Fig. 1. Synthetic test cases concerning HOL blocking with images. `h2` performs well for many small files but deteriorates for less or larger files. Sharding only helps `h2` for larger files.

and brings it somewhat on a par with `h1` (left). This is expected as the single `h2` connection can efficiently multiplex the many small files. It is of note that the concatenated version is two to five times faster overall, even though (in a rare compression fluke) its file size is much higher than the sum of the individual file sizes. Additionally, we see that sharding deteriorates `h2`'s performance, while only marginally benefiting `h1` (right). Because the files are that small, `h2`'s multiplexing was at its best in the single host case and maximized the single connection's throughput, while in the sharded setting it has less data to multiplex per connection. Conversely, sharding empowers `h1` to open up more connections, but still suffers from HOL blocking on the small files.

Figure 1(b) shows relatively little differences and no clear consistent winners between the concatenated (left) and separate files (middle) over one host. This is somewhat expected for `h2`, as in both cases it sends the same amount of data over the same connection, but not for `h1`. We would expect the six parallel connections to have more impact, but it seems they can actually hinder on good network conditions. This is probably because of the limited bandwidth in our emulated cellular network, where the six connections contend with each other, while a single connection can consume the full bandwidth by itself. Unlike `h2`'s behaviour in (a), we see that here `h2` does not get significantly faster for the concatenated version. This indicates that the higher measurements in (a)(middle) are in large part due to the overhead of handling the many individual requests. Similarly, sharding (right) shows inconsistent behavior: sometimes it helps and sometimes it hurts `h2`; it shows impressive benefits for `h1c` but smaller gains for `h1s`. We posit that the additional overhead of setting up extra secured HTTPS connections (both for `h1s` and `h2s`) limits the effectiveness of the higher parallel throughput. Overall, we can state that there is no clear winner here, nor for the three different setups, nor for the three protocols.

Lastly, in Fig. 1(c) we see that `h2` struggles to keep up with `h1` for the larger files and performs significantly worse under bad network conditions (note the y-axis' log scale). Due to the much larger amount of data, `h1`'s larger amount of parallel connections do help here, while packet loss impacts the fewer `h2` connections more. This is immediately apparent when comparing the NoLoss and Good network conditions in Fig. 1(c)(middle): the `h1` measurements are very similar while the single `h2` connection is almost 80% slower in this case (note that the NoLoss and Good conditions are identical except for the amount of packet loss introduced). As expected, utilizing additional parallel connections (right) benefits both `h1` and `h2`, helping mitigate HOL blocking for `h1` and lessening the impact of loss when compared to a single `h2` host. The SpeedIndex measurements (not included here) show very similar trends for all of the experiments discussed in Fig. 1.

In conclusion, we can say that while `h2` indeed helps for many smaller files, it still loses to concatenated versions of those files, both over `h1` and `h2`. This indicates that the current `h2` implementations can incur heavy costs for handling individual resources, though this primarily poses a large problem for many (>42) files (see also Sect. 4.3). We can also conclude that `h2`'s single connection setup seems to suffer from bad network conditions, but not excessively more than `h1`, and the performance drop largely depends on the observed case. Similarly, we have observed that using multiple parallel connections for `h2` can help mitigate this problem (especially for websites with large objects), but that it can also lead to slower load times (for many, smaller objects). These findings are consistent with the previous work of Goel et al. [10], who overall observed that *if* sharding helps for `h2`, sharding over

more hosts helps more, but there are diminishing returns with each increase in the amount of hosts. Additionally, Mi et al. [21] decisively show that large files can increase the time to download smaller files by 99% over a single `h2` connection. They propose an extension to `h2` that allows migrating resource requests between parallel TCP connections (also in a multipath TCP setting). Interestingly, Manzoor et al. [16] have shown empirically that various browsers are already using multiple parallel connections for `h2` in the wild (although this was never observed during our tests). This might indicate that the browser vendors are aware of the beneficial nature of this practice. However, to the best of our knowledge, the browser vendors have yet to present their own results on this issue.

4.3 HOL Blocking in Practice with CSS and JavaScript.

HOL Blocking with CSS and JavaScript with `loadEventEnd`. The discussion in 4.2 has clearly shown the impact of network conditions and the amount of parallel connections of `h2`'s performance. It has also shown that due to HOL blocking, `h2` seems to shine when loading a large amount of smaller files, but that it is not necessarily faster when the amount of files is lower. In order to investigate this property further and determine the point where HOL blocking is overcome, we observe two experiments in Fig. 2: 500 `<div>`-elements are styled using (left) simple CSS files (single CSS rule per `<div>`) and (right) complex JS files (multiple statements per `<div>`). We vary the degree of CSS and JS code concatenation, from one file (full concatenation) to 500 files (no concatenation). Figure 2 plots full results in (a) and shows more detail for one to 30 files in (b). We resorted to CSS and JS files in these experiments instead of images because they typically include additional processing from the browser, which can also impact page load time performance, as we will see. The data shown here is from tests using the Good network condition.

The big-picture trends in Fig. 2(a) look very similar to Fig. 1(a)(left and middle): `h2` again clearly outperforms `h1` as the number of files rises and shows a much better progression towards larger file quantities than the quasi linear growth of `h1`. Interesting is also the performance of Firefox: while its `h1` results (not shown in Fig. 2 for clarity) look almost identical to Chrome, its `h2` values are much lower, indicating that it has a more efficient implementation that scales better to numerous files.

Looking at the zoomed-in data in Fig. 2(b), we do see somewhat different patterns. For the simple CSS files the trends are relatively stable, with `h1c` outperforming `h2` and `h2` beating `h1s`. This changes at about 30-40 files, where `h2` finally takes the overall lead. For the more complex JS files (right), this tipping point comes much later around 100 files. The measurements for one to ten JS files are also much more irregular when compared to CSS. Because `h1` shows the same incongruous data as `h2`, we can assume this can be accounted to the way the browser handles the computation of the larger incoming files. The performance of a multithreaded or otherwise optimized handling of multiple files can depend on how many files are being handled at the same time. This would also explain the very high `h2` measurements for a single JS file in Firefox (consistent over multiple runs of the experiment). In additional tests, smaller JS files and larger CSS files also showed much more stable trends, indicating that especially large JS files incur a large computational overhead. Note as well that the timings for a smaller amount of JS files are sometimes higher than those for the larger amounts, indicating that concatenation might not always be optimal here (for none of the
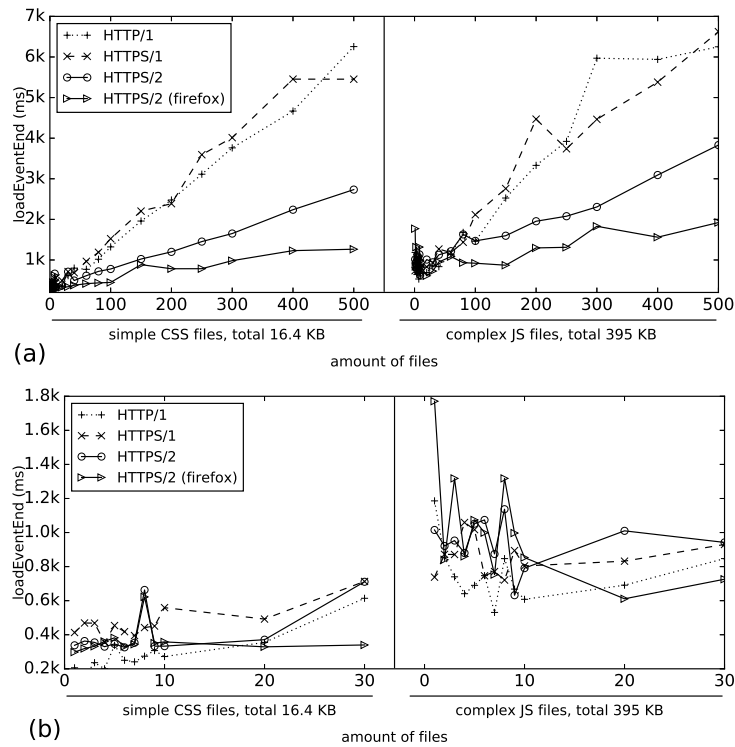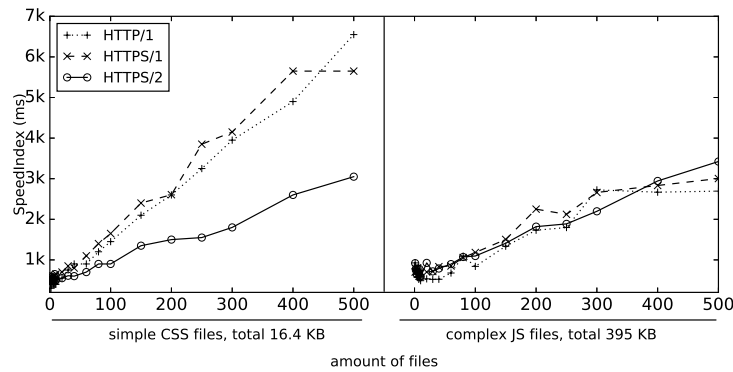
Fig. 2. Synthetic test cases for HOL blocking with CSS/JS files. `h2` performs well for many files but there is no clear winner for the more concatenated cases. Image taken from our previous work [18].

protocols). Poor network conditions (not shown here) show similar trends to Good networks, but the `h2` tipping points come later: 40-50 files for simple CSS, 150 for complex JS.

All in all, we can see that `h2` only overcomes `h1`'s HOL blocking problems at a relatively large amount of individual files (30+ in the best case). While most websites do include that many resources, our results also show that concatenating files together (thus again reducing the total resource count) can overall be faster than sending individual files for all protocols (especially for CSS files and many images, see Fig. 1(a)). This again confirms our earlier thesis that browsers introduce a lot of overhead per individual resource/request, regardless of the actual size of the data (though Firefox seems to have a more efficient implementation than chrome, at least for `h2`) and that this issue needs to be resolved first before `h2` can overtake `h1` and its best practices.

HOL Blocking with CSS and JavaScript with `SpeedIndex`. For the tests in the previous section 4.3, the `SpeedIndex` results were significantly different from the `loadEventEnd` measurements and merit separate discussion. Figure 3 shows the same experiment but depicts `SpeedIndex` for Google Chrome. We notice that the data for the simple CSS files (left) looks very similar to Fig. 2, but the results for the complex JS files (right) do not. Since

the `SpeedIndex` metric gives an indication of how progressively a page renders (Sect. 3) and because we know from Fig. 2 that `h1` takes much longer than `h2` to load large amounts of small files, we can only conclude that under `h2` the JS files take much longer to have an effect on the page rendering, to skew the `SpeedIndex` in this way.

Fig. 3. Synthetic test cases for HOL blocking with CSS/JS files (`SpeedIndex` metric). `h2 SpeedIndex` for JavaScript indicates that it is much slower to start rendering than `h1`. Image taken from our previous work [18].

We manually checked this assumption using screenshots and found that for `h1` the JS was indeed progressively executed as soon as a file was downloaded, but with `h2` the JS code was applied in "chunks": in larger groups of 50 to 300 files at a time and mostly towards the end of the page load. We first assumed this was because of erroneous multiplexing: if all the files are given the same priority and weight, their data will be interleaved, delaying the delivery of all files (see Sect. 5). Captures of `h2` frame data in Google Chrome however showed that each file `i` was requested as dependent on file `i - 1`, and that file data was fully delivered in request order (consistent with the behaviour described in Sect. 5). We can once more only conclude that the browser implementation somehow delays the processing of the files, either because of their JS complexity or because the handling of many concurrent `h2` streams is not optimized yet. This argument is supported by the `SpeedIndex` results for Firefox (not shown here, for clarity), as its `h2` values are much lower than those of `h1`, indicating that Firefox has a more efficient `h2` implementation than Chrome.

If the browsers' handling of CSS code would be similar to that of JS code, we would expect to see similar results in Fig. 3 (left) and (right). However, if the CSS files would also be applied individually as soon as they were downloaded, the `h1` SpeedIndex values would be much lower than the observed measurements. We found that the browser delays execution of *all* CSS until they have all been downloaded and processed for both `h1` and `h2`, despite our experiments having been built specifically to prevent this. This is again unexpected browser behaviour (though probably not directly related to the `h2` implementation) and we plan to look deeper into this in future work, as discussed in [18].

5 Resource Prioritization

5.1 Background

As demonstrated in Sect. 4, `h2` solves the `h1` Head-Of-Line blocking problem by allowing multiple resources to be sent on the same connection at the same time. To make this possible, each resource is assigned to its own conceptual "stream" and these streams are then multiplexed over the single underlying TCP connection. The data from the individual files is split up in chunks and can thus be interleaved with chunks from other files. This is especially interesting for resources that are partly directly available but that need slow I/O operations to complete (e.g., an HTML template that fetches content from a database). Using multiplexing, chunks from other resources can be sent while the data of the "delayed" resource is being fetched, resulting in less idle time on the TCP connection. Alternatively, when concurrently sending a very large and a very small file, the data from the small file might be multiplexed with parts of the larger file so the receiver does not need to wait for the larger file to be fully downloaded to receive the smaller resource [21].

To this end, the `h2` specification [3] details the concept of a "dependency tree". Nodes in this dependency tree represent individual `h2` streams, while the root of the dependency tree denotes the underlying TCP connection. New nodes are added to the tree as new resources are requested and nodes can be removed when their corresponding resources have been fully downloaded. A parent-child relationship between nodes indicates that the child's transmission should be postponed until its ancestor has been downloaded completely (or until it is temporarily impossible to make progress on the parent resource). Conversely, a sibling relationship between nodes allows bandwidth to be distributed among the siblings proportionally to their "weight" (i.e., ∈ [1,256]), thus allowing multiple resources to be interleaved in a very fine-grained way. The `h2` buildup of the tree is decided by the browser at runtime and communicated to the server using `HEADERS` or `PRIORITY` frames. This general setup allows for a lot of flexibility in how the dependency tree is effectively constructed and maintained by the browser during the page load.

Figure 4 shows example dependency trees from Google Chrome (a) and Mozilla Firefox (b) respectively. It is apparent that Chrome chooses a very sequential setup, where each node is the only child of its parent (rendering individual stream weights effectively useless). It does however maintain an internal "priority order" depending on the type and location of the resource (e.g., a CSS file in the `<head>` will be given a `Highest` priority level, while an image in the `<body>` will have a `Low` overall importance). If a new resource is discovered, it will be not be added at the end of the full tree, but rather after the last existing resource with the same priority level. Firefox utilizes similar priority bins internally (indicated by e.g., `leaders`, `followers`, `unblocked`) but chooses to build its priority tree in a radically different way from Chrome. Firefox adds "ghost" nodes for each of these priority levels (which do not directly represent an `h2` resource or stream) to be able to group the `h2` streams that belong to this category as siblings. This allows Firefox to use a more complex prioritization strategy for its `h2` implementation.

5.2 Evaluation of Prioritization Strategies

As Chrome's and Firefox's approaches for the `h2` dependency trees are fundamentally different (Fig. 4), it is difficult to directly compare both options and see which one performs
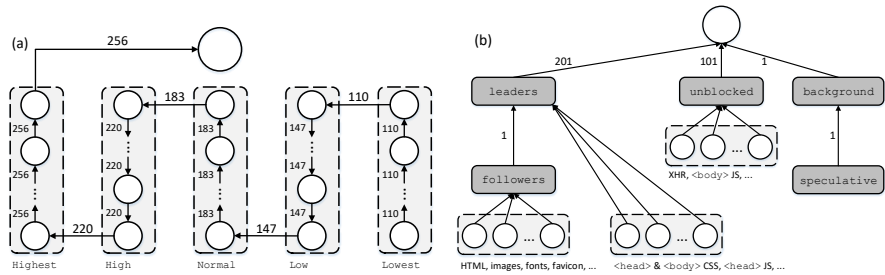
Fig. 4. HTTP/2 dependency tree layout of (a) Chrome and (b) Firefox. Numbers indicate node weights.

best. This is due to the fact that the both browsers are internally optimized for their specific strategy, implying that using a different strategy will skew the results. Instead, we implement two alternative, less complex prioritization strategies to see how much better (or worse) the browser's more advanced approach works.

The first alternative algorithm, Round Robin (RR), is the default behaviour specified in the `h2` specification [3]. All `h2` streams are made siblings under the root node and each is given an equal weight. In effect, this causes all active resources to be given an equal share of the bandwidth and leads to heavy multiplexing. The second algorithm, First-Come-First-Served (FCFS), approaches the way `h1` works. The dependency tree is purely linear and each new node is added as the bottom leaf node; FCFS is thus a much less advanced version of Chrome's strategy as it will never inject new nodes between two existing nodes. FCFS entails that the current resource has to be sent fully before (any part of) the next resource can be sent, effectively disabling multiplexing.

These two alternative algorithms are implemented by modifying the H2O server source code. The server simply ignores the priority directives from the browser (which is allowed behaviour as per the `h2` specification [3]) and builds its own dependency tree using the rules described above.
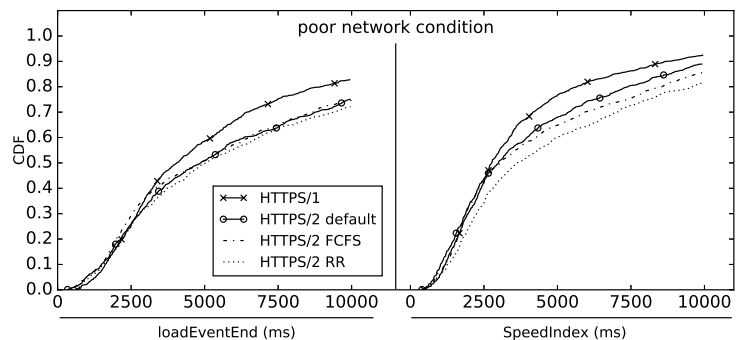
Fig. 5. HTTP/2 prioritization strategies in Firefox for a Poor network condition. Round Robin is clearly detrimental for SpeedIndex and `h1s` is consistently faster than `h2`.

Figure 5 shows the CDF results for the Firefox browser in Poor network conditions. We take a corpus of 40 websites (see Sect. 8 for details) and load them 20 times with each protocol/prioritization strategy; the median values are used in the CDF.

We can deduce that the impact of the different `h2` prioritization strategies is moderate for the `loadEventEnd` metric, with `h2` measurements being very similar. However, the `SpeedIndex` metric clearly shows worse performance when using the Round Robin strategy. This is expected, as it will take longer for resources to be fully downloaded and because, as we have discussed in Sect. 4.3, browsers will often wait for a full resource (or group of resources) to be downloaded before re-rendering the page. This is a remarkable result because, as mentioned before, Round Robin is the default prioritization behaviour prescribed by the `h2` specification and in our tests it was seen in effect in both Microsoft's Edge and Apple's Safari browsers (which do not seem to employ a custom prioritization strategy at this time).

We performed similar tests for Google Chrome and on the various other network emulation settings detailed in Sect. 3. The results displayed in Fig. 5 are among the most distinct of all our evaluated data, meaning that other tests showed even less differences between the different strategies, especially for improved network conditions. This indicates that the adopted prioritization strategy is not a major influencer of `h2` page load performance, except on poor networks and then only when used in the most straightforward way. This reflects previous work by Bergan [4], who found that Chrome's implementation only clearly outperforms a completely random strategy in 31% of the observed pages.

Finally, when looking at the browsers' prioritization strategies, we found that their implementations are often still lacking in their support of cutting-edge web technologies. For example, the new Service Worker concept[7] allows developers to register a JS-based "client-side proxy" that can intercept and perform custom processing on all requests the browser emits. We found that all `h2` requests passing through such a Service Worker lost all of their intelligent prioritization information, leading to Chrome defaulting to a FCFS-alike strategy, while Firefox exhibited pure RR behaviour. This is probably an implementation oversight and we expect this to be fixed in the future. As another example, developers can indicate to the browser that certain JS files are less important using the `async/defer` attributes. Chrome correctly assigns a `Low` priority to those resources, but in Firefox they are regarded as normal, high-priority JS files. We believe that these and similar browser implementation errors could be responsible, at least partly, for some contradictory results in other work (Sect. 2).

6 Server Push

6.1 Background

In HTTP/1.1, the browser can only receive resources that it has explicitly requested. Typically, the user agent first fetches the HTML page (e.g., index.html), which it then parses to discover other referenced resources. As such, it takes at least one RTT before the browser can start requesting critical CSS/JS files and a minimum of two RTTs before they are downloaded. Especially on slow networks, this can have a large performance impact. In response, the HTTP/2 specification [3] describes a novel mechanism called "Server Push". This allows the

---

[7] https://developers.google.com/web/fundamentals/getting-started/primers/service-workers

server to decide to send along additional resources with previously requested resources, not having to wait for the browser to request them first, thus potentially saving a full RTT.

In theory, developers could push all necessary resources of a website along with the original HTML request and thus eliminate additional RTTs completely. In practice however, Server Push is limited by TCP's congestion control mechanisms. For example, in its "slow start" phase, TCP sends only a small amount of data at the beginning of the connection and then exponentially ramps up its speed if no packet loss or delays are present. In practice, the TCP congestion window starts at about 14 KB for modern Linux kernels (as used in our experiments) [17], severely limiting the amount of resource data we can push during the first RTT. Given this behaviour, `h2` Server Push's benefits should increase the longer the TCP connection stays open (i.e., the congestion window grows as the connection gets "warmer"), as more data can be pushed in a single RTT.

Server Push could be a good fit for the popular modern Single Page App (SPA) setup. In this paradigm, the loaded page routinely requests additional data from the server using a (REST) API, thus keeping the TCP connection active. The API's response will then no longer just consist of the structured `xml/json` data, but can also contain the pushed subresources (e.g., images) mentioned in the data. Another interesting use case is to deploy Server Push from a network intermediary, such as a CDN proxy. In this setup, the browser typically connects to the proxy, which in turn connects to the origin server. The proxy can then "warm up" its connection to the browser by pushing static assets (mainly CSS/JS/font files) while it waits for the dynamic HTML and other data to arrive from the origin. This use case is discussed in-depth by Zarifis et al. [30], who show up to 27% web page load time improvements when using Server Push in this fashion.

However, the page load performance of Server Push can be very dependent on knowledge of the correct priority of the resources it wants to push and how they fit into the page loading process. The main reason for this is that large parts of common network stack implementations use buffered I/O, both on the OS-level and in the network itself [5]. Once data is queued in these intermediate buffers, it is often impossible to remove it or replace it with other data. If we then, for example, would immediately push three very large image files along with the initiating request and their data fills up all buffers, the browser's request for a more critical CSS file will be delayed because we cannot re-prioritize the less important image data in the buffers. This makes for large practical difficulties in determining which resources to push and when [30]. This problem is enlarged by the fact that the `h2` specification [3] does not include a mechanism for the browser to signal to the server which files it has already cached. Consequently, the server will potentially push files which the browser already has, wasting bandwidth and delaying other resources.

6.2 Experimental Evaluation

While a full in-depth evaluation of the discussed characteristics of `h2` Server Push is out of scope for this work, we can nevertheless demonstrate several of the discussed aspects using a very simple example. We use the existing Push demo by Bradley Fazon[8], based on the *www.eff.org* frontpage. This page has a single "critical CSS" file (which is responsible for the main look-and-feel of the site and thus a good Push target) and a good mix of additional

---

[8] https://github.com/bradleyfalzon/h2push-demo

CSS, JS and image files without being too complex. We make sure the initial HTML code is smaller than 14 KB (by removing some metadata and enabling gzip compression), reducing the on-network HTML size from 42 KB to 9 KB.

Figure 6 shows the results from tests using the Apache webserver because NGINX does not yet support `h2` Server Push. The SpeedIndex results are displayed because these should be most affected. We observe five different experiments: (1) push the single "critical CSS" file, (2) push all the CSS/JS files (10 files), (3) push all (images + CSS/JS + one font), (4) push all images (18 files) and (5) the reference measurement (original, no push). We see that pushing the one "critical CSS" file does indeed improve the `SpeedIndex` measurements, but not excessively so. It is unexpected however that pushing all CSS/JS performs a little better than just the "critical CSS" in the Good network condition. We found that in practice the initial data window is often a bit larger than 14 KB, so it can accommodate more than just the single CSS file. However, this is not always the case and other runs of the same experiment show less optimal results (which can also be seen in the Poor network condition). Given a larger data window, the "Push all" test case should perform similarly to pushing the CSS/JS resources, but it is consistently a bit slower. It turned out that we pushed the single font file at the very end, after all the images. The font data should have been given a higher `h2` priority than the images, but due to an Apache bug[9] this was not the case and the font data had to wait, delaying the final render. This also explains why pushing just the images performs worse than the reference: the much more important CSS and JS is delayed behind the image data.
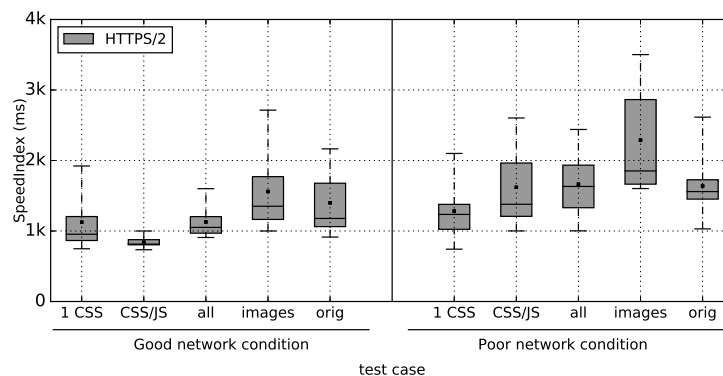
Fig. 6. Realistic test case for HTTP/2 Push. Pushing the wrong assets or in the wrong order can deteriorate performance.

The discussed aspects and challenges make Server Push difficult to fine-tune to achieve optimal performance. Due to this and the fact that many popular `h2` server implementations do not yet support Server Push, Zimmerman et al. [31] found that out of their observed 5.38 million HTTP/2 enabled domains, only 595 actively used Server Push.

---

[9] https://icing.github.io/mod_h2/nimble.html

7 HPACK Header Compression

7.1 Background

HTTP uses the concept of *Headers* to convey various types of metadata about its requests and responses between the user agent and the server. These headers are typically prepended to the actual message body. Some popular header names are `Content-Type`, `Keep-Alive`, `Cache-Control` and `Cookie`. This last header is useful to bind multiple requests and responses to the same conceptual "user session", allowing applications to provide stateful interactions. Cookies typically contain a numeric user ID or session token but can also include more complex (serialized) data, which can make them relatively large in practice [29].

The headers are often repeated with each individual message, which can be wasteful with respect to bandwidth usage, especially in the case of large metadata like `Cookie`. HTTP/2 attempts to solve this deficiency by introducing HPACK [24], a compression algorithm specifically tuned to the HTTP header format. HPACK combines a pre-defined dictionary of known prolific header names and values with a dynamic shared dictionary per connection that is built up at runtime (at both the server and browser side), based on the header data that is actually being sent during the session. As such, HPACK will perform better with large amounts of similar files or cases with large dynamic metadata, as it learns the repeating data on-the-fly. For example, the first time a header of the form `Cookie: value` is sent on the connection, it is stored in the dynamic dictionary. The next time this specific header would be sent, it can be wholly replaced by a reference to the dictionary entry, which is identical on both client and server.

7.2 Experimental Evaluation

To demonstrate the behaviour of HPACK, we use data gathered during our experiments from Sect. 4, which include only typical HTTP headers and no cookies are set. Table 2 details three cases: (a) 10 large images, (b) 42 medium images and (c) 400 complex JS files. The `BytesOut` measurements consist of all data that was sent by the browser and thus include primarily HTTP request headers and TLS connection setup data. The actual header-induced overhead is even larger if we also consider HTTP response headers.

For the cases with one host server, we can clearly see that HPACK significantly reduces the overall header size when compared to `h1`, with a factor of more than five for case (c). It is also apparent that the header overhead is typically relatively low but can grow to 27% for many individual files on `h1s` (as each of those files requires a separate request and response message). Looking at the sharded setup with four host servers, we see that while both protocols produce more overhead from the extra connections, `h2s` relatively suffers more than `h1s`, especially for (a) and (b) (note that the `h1s` overhead is mainly due to the TLS overhead from opening 24 connections compared to the six for the single host case and four for sharded `h2s`). This is expected, as `h2s` now has less data to learn from on each individual connection and optimize its dynamic compression scheme. This is another argument of `h2` to favor using only a single underlying TCP connection.

It is of note that these observed header compression results are arguably too low to have a significant impact on the performance of any individual page load of a realistic website. However, when viewed on a larger scale (e.g., cumulatively across all the servers in a data

center or CDN) these savings can add up and make a significant difference in the overall bandwidth usage of popular websites. Related work from Cloudflare [1] indicates that on average HPACK reduces HTTP header size by 30% and overall HTTP/2 egress traffic by 1.4%, with outliers of up to 15% for individual websites.

Table 2. Total bytes sent by Google Chrome ($\sim$ HTTP headers) and ratio to total page size. For many small files, the HTTP header overhead is significant. Sharding over multiple hosts decreases the effectiveness of HPACK header compression.

File count	Protocol	Total page size	1 host		4 hosts	
			BytesOut	% of total page size	BytesOut	% of total page size
(a) **10 large files**	h2s	2177600	504	0.02%	1227	0.05%
	h1s	2177600	2419	0.1%	2993	0.1%
(b) **42 medium files**	h2s	1075000	649	0.06%	1362	0.1%
	h1s	1075000	2786	0.2%	3346	0.3%
(c) **400 small files**	h2s	610000	29580	4%	38680	6%
	h1s	610000	165300	27%	177600	29%

8 HTTP/2 Performance for Realistic Web Pages

8.1 Experimental Setup

While the synthetic test cases from the previous sections (excluding Sect. 5) are useful to assess the individual `h2` performance techniques in isolation, they are not always representative for real websites. We will now look at some more realistic test cases. We will first present results for a corpus of nine manually selected website landing pages (corpus A), which all contain either many smaller images (e.g., media/news sites) or fewer but larger images (e.g., product landing pages with large images taking up most of the "above the fold" space). The composition of this corpus is motivated by the goal of enabling easy and meaningful comparison with our synthetic experiments in Sect. 4.2. As we will see however, while the resulting findings showed clear trends, it was difficult to pinpoint their underlying causes. In response, we executed additional tests on a second, larger corpus of 40 landing pages (corpus B) taken from the Alexa Top 50 and Moz Top 500 rankings[10]. These pages were selected primarily on their total filesize, with 10 pages being low-weight ($<$ 500KB), 10 pages medium-weight ($\geq$ 500KB, $\leq$ 1MB) and 20 pages heavy-weight ($>$ 1MB). All pages were cloned using the `wget` tool[11] so that they could be served locally in the Speeder experimental setup (Sect. 3).

---

[10] http://www.alexa.com/topsites, https://moz.com/top500
[11] https://www.gnu.org/software/wget/

The experimental setup is meant to simulate what would happen if a developer would switch their `h1` site to `h2` by naively moving all their own assets over to a single server (disabling sharding) but still downloading some external assets from third party servers (e.g., Google analytics, some JS libraries). This approach is similar to the one adopted in [27]. We expect to see good `h2` performance compared to `h1`, as the latter has only six parallel connections to work with and `h2` can optimally use its single TCP connection.

The results for corpus A are from server NGINX v1.10, browsers Google Chrome v54 and Mozilla Firefox v49 and test runner webpagetest v2.19. Each page was loaded at least 10 times. The results for corpus B were obtained later during our research through the standard H2O server v2.1, Google Chrome v58, Mozilla Firefox v54 and webpagetest v3.0. Each page was loaded at least 20 times. We will display the median values. For more details on both test corpora and the Speeder setup, we refer to our website (see Sect. 3).

8.2 Experimental Results

Figure 7 shows the median `loadEventEnd` and `SpeedIndex` measurements for corpus A over Good and Poor networks. Globally, we can state that `loadEventEnd` and `SpeedIndex` are often similar for the three protocols on the Good network, indicating that the page load times of the tested pages are mostly network dependent, with the rendering having to wait for assets to come in. This explains why Poor network conditions can have a very large impact on page load time performance (see Fig. 7(right)). In various cases, `h2`'s SpeedIndex is far above that of `h1` even if their `loadEventEnd` values are similar, indicating that `h2` is slower to start rendering, consistent with our observations in Sect. 4.3. `h1c` is faster than `h2` in almost all of the cases and `h2` is almost never much faster than `h1s`. Note that this is somewhat against our hypothesis, as `h1s` has to make due without the benefits of sharding. A more in-depth discussion of some of the outliers in Fig. 7 can be found in [18].

Looking more closely at the results for Poor networks in Fig. 7, we see that `h2` is sometimes much slower than `h1` but sometimes is also relatively similar. Given the limited size of corpus A, it was difficult to pinpoint the underlying reasons for this inconsistent behaviour. Suspecting that the total page size and amount of objects on the page had a large influence (both from the corpus A results and our synthetic tests in Sect. 4), we ran additional tests on the larger corpus B. The results in Fig. 8 show that our thesis was indeed correct: the low-weight pages (left) have similar `SpeedIndex` performance for `h1s` and `h2s` even on the Poor network, while for the heavier pages (right) `h2` clearly suffers. The results for the `loadEventEnd` metric showed similar though less pronounced trends for the Poor network and only small differences in the three protocols' measurements on the Good network.

Finally, it is difficult to directly compare our results for realistic pages to related work, since few authors present results from a large corpus of locally cloned web pages over various network conditions with modern `h2` implementations or for the `SpeedIndex` metric. The closest related work loads pages directly on the internet via various networks and shows more positive `loadEventEnd` results for `h2` than our tests do, for example that 80% of pages on faster networks clearly benefit from `h2` [26]. This percentage is lower on slower networks but there `h2` typically also has a higher benefit. We are unable to confirm their findings with our measurements. The most recent related work [31] loads pages over a high speed link and concludes that 51% of the tested pages are $\geq$ 5% faster over `h2` when compared to `h1s`, which is also contradictory to our realistic test case results.

(a)

(b)

(c)

(d)
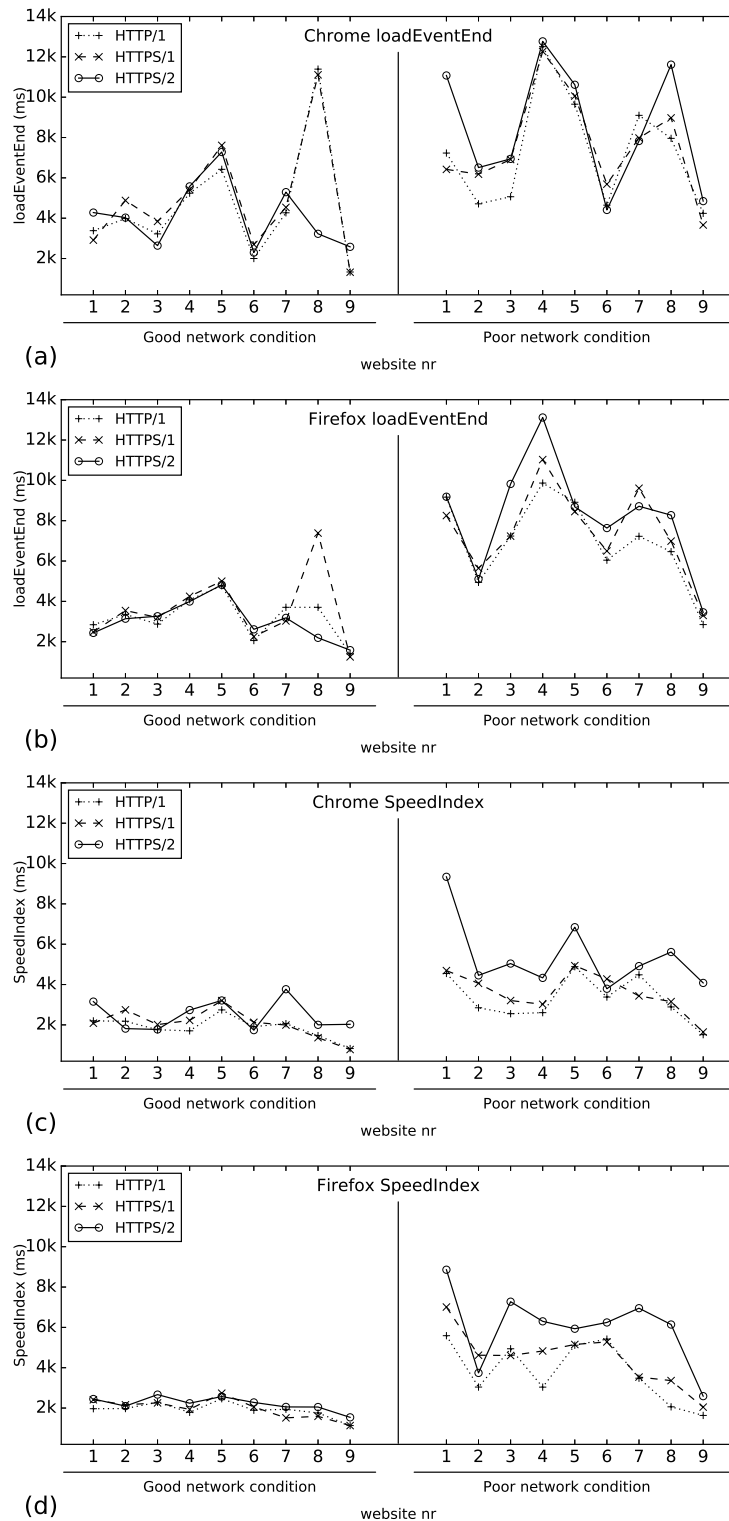
Fig. 7. Nine realistic websites from corpus A on the dynamic Good and dynamic Poor network models. There is very similar performance under Good network conditions, but `h2` clearly suffers from Poor conditions. Image taken from our previous work [18].
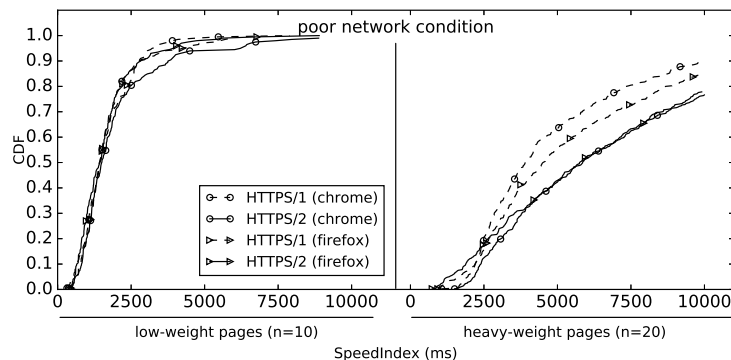
Fig. 8. Differences in SpeedIndex for low-weight and heavy-weight pages from corpus B in Poor network conditions. Heavy-weight pages clearly cause `h2` to suffer more and are faster to load under `h1s`.

9 Discussion

Conceptually, the ideal HTTP/2 setup will use a single TCP connection to multiplex a large amount of small and individually cacheable site resources. This mitigates the HTTP/1.1 application-layer HOL blocking issue and helps to reduce the `h1` overhead of many parallel connections, while also maximizing the efficiency of the underlying TCP protocol. Together with advanced resource prioritization strategies, Server Push and HPACK header compression, this can lead to (much) faster load times than are possible today over `h1`, with less overhead.

Unfortunately, as our experiments have shown, this ideal setup is not yet viable. While `h2` is indeed faster than `h1` when loading many small files (Fig. 1 and 2), it is still often slower than loading concatenated versions of those files over `h2` (Sect. 4.2). Looking at the `SpeedIndex` metric results (Fig. 3, 7 and 8) also shows that `h2` is frequently later to start rendering the page than `h1`. HTTP/2 also struggles when downloading large files (Fig. 1 and 8) and its performance can quickly deteriorate when used in bad network conditions. In our observations, `h2` is in most cases currently either a little slower than or on a par with `h1` and shows both the most improvement and worst deterioration in extreme circumstances.

The good news is that almost all of the encountered problems limiting `h2`'s performance seem to be due to inefficient implementations in the used server and browser software. Firstly, while loading many smaller files incurs its own considerable browser overhead, the comparison of Chrome and Firefox in Fig. 2 tells us that this overhead can be reduced, as Firefox seems to have especially optimized its pipeline for large amounts of files. Secondly, the fact that `h2` is later to start rendering than `h1` is also due to ineffective processing of the `h2` data, since we have confirmed that resources are received well in time to enable faster first paints (Sect. 4.3). Thirdly, several cases in which `h2` underperformed could be attributed to the server or browser not correctly (re-)prioritizing individual assets (Sect. 5 and 6). As these implementations mature, we can expect many of these issues to be resolved.

However, `h2` still retains some core limitations, mostly due to its single underlying TCP connection, which seems to simultaneously be its greatest strength and weakness. TCP's congestion control algorithms can lead `h2` to suffer significantly from packet loss on Poor

networks (most obvious when downloading multiple large files (Fig. 1 and 8)) and can heavily impact the effectiveness of `h2` on newly established connections (Sect. 6). We have to nuance these statements however, as in practice `h2` actually performs quite admirably and usually does not suffer more from bad networks than `h1`, despite using fewer connections. Additionally, we have found that `h2` can also benefit from using multiple connections in bad networks, especially in the cases where its performance problems are greatest.

The other discussed `h2` performance aspects do not seem to have as large an impact as the use of the single TCP connection. While prioritization is certainly important, the exact strategy that is used seems to have relatively little impact in most cases. Chrome and Firefox use wildly different algorithms to build their dependency trees (Sect. 5). Similarly, HPACK has only a limited impact on the total used bandwidth for most normal cases and will probably not directly affect individual page load times (Sect. 7). Finally, `h2` Server Push sounds like a powerful optimization but takes a lot of work and special network setup (e.g., CDN intermediaries) to save more than a single RTT on a page load (Sect. 6). Further work is needed to determine how to optimize both `h2` resource prioritization and Server Push.

Recognizing that the core `h2` performance problems stem primarily from the use of TCP, the new QUIC protocol [6] implements its own application-layer reliability and congestion control logic on top of UDP. QUIC removes the transport-layer HOL blocking by allowing out-of-order delivery of packets, differently handles re-transmits in the case of loss, reduces the amount of round-trips needed to establish a new connection and allows larger initial data transmissions. Running `h2` on top of QUIC could greatly benefit `h2`'s multiplexing setup.

As such, we can conclude that the HTTP/2 protocol specification is a solid foundation for the next steps in bringing better page load performance to the web and reducing overall overhead. It will however take some time for implementations to mature and the QUIC protocol to be finalized before we will see its largest benefits in practice.

10   Conclusion

In this work we have discussed and evaluated four salient performance-related aspects of the new HTTP/2 protocol: using a single underlying TCP connection (Sect. 4), prioritization of multiple resources over this single connection (Sect. 5), the new Server Push construct (Sect. 6) and HPACK header compression (Sect. 7). Our evaluation was comprehensive and varied, looking both at synthetic and realistic test cases, over a variety of software, performance metrics and emulated network conditions.

Our results have shown that the switch to the single multiplexed TCP connection has by-and-large the biggest performance impact when comparing `h2` to `h1`'s multiple parallel connections. While in most cases `h2` performs similarly to or slightly better than `h1` (while inducing much less overhead), poor network conditions coupled with large files can cause `h2`'s performance to deteriorate. The emerging QUIC protocol might help `h2` overcome these problems by switching to UDP, while in the mean time using multiple concurrent `h2` connections can also help.

Other discovered performance problems, such as `h2` delaying the time to start rendering web page content, were likely to stem primarily from incomplete or erroneous `h2` implementations and are expected to be solvable in time. Similarly, prioritization and Server Push both have potential but require future work to determine their best practices.

ACKNOWLEDGEMENTS

References

1. Alpichi, K.: HTTP pipelining. `https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/` (2017), last visited: 2017-08-08
2. Beheshti, H.: HTTP/2: What no one's telling you. `http://www.slideshare.net/Fastly/http2-what-no-one-is-telling-you` (2016), last visited: 2017-03-01
3. Belshe, M., Peon, R., Thomson, M.: HyperText Transfer Protocol Version 2. `https://tools.ietf.org/html/rfc7540` (2015), last visited: 2017-03-01
4. Bergan, T.: Benchmarking HTTP/2 Priorities. Online, `https://docs.google.com/document/d/1oLhNg1skaWD4_DtaoCxdSRN5erEXrH-KnLrMwEpOtFY/` (October 2016)
5. Bergan, T., Pelchat, S., Buettner, M.: Rules of Thumb for HTTP/2 Push. `https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0` (2016), last visited: 2017-03-01
6. Carlucci, G., De Cicco, L., Mascolo, S.: HTTP over UDP: an Experimental Investigation of QUIC. In: Proceedings of the ACM Symposium on Applied Computing. pp. 609–614. ACM (2015)
7. Corbel, R., Stephan, E., Omnes, N.: HTTP/1.1 pipelining vs HTTP2 in-the-clear: Performance comparison. In: 2016 13th International Conference on New Technologies for Distributed Systems (NOTERE). pp. 1–6 (July 2016)
8. Erman, J., Gopalakrishnan, V., Jana, R., Ramakrishnan, K.K.: Towards a SPDY'Ier Mobile Web? In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies. pp. 303–314. CoNEXT '13, ACM, New York, NY, USA (2013), `http://doi.acm.org/10.1145/2535372.2535399`
9. Everts, T., Kadlec, T.: WPO stats. `https://wpostats.com/` (2017), last visited: 2017-08-03
10. Goel, U., Steiner, M., Wittie, M.P., Flack, M., Ludin, S.: HTTP/2 Performance in Cellular Networks: Poster. In: Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking. pp. 433–434. MobiCom '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2973750.2985264`
11. Gooding, M., Garza, J.: Real world experiences with HTTP/2. `https://www.slideshare.net/JavierGarza18/real-world-experiences-with-http2-michael\-gooding-javier-garza-from-akamai` (2016), last visited: 2017-03-01
12. Grigorik, I.: High Performance Browser Networking. O'Reilly Media, Inc. (2013)
13. Kohavi, R., Deng, A., Longbotham, R., Xu, Y.: Seven rules of thumb for web site experimenters. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1857–1866. ACM (2014)
14. Krasnov, V.: HPACK: the silent killer (feature) of HTTP/2. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x#HTTP_pipelining` (2017), last visited: 2017-08-08
15. Liu, Y., Ma, Y., Liu, X., Huang, G.: Can HTTP/2 Really Help Web Performance on Smartphones? In: Services Computing (SCC), 2016 IEEE International Conference on. pp. 219–226. IEEE (2016)
16. Manzoor, J., Drago, I., Sadre, R.: The curious case of parallel connections in HTTP/2. In: Network and Service Management (CNSM), International Conference on. pp. 174–180. IEEE (2016)

17. Marx, R.: HTTP/2 Push : the details. http://calendar.perfplanet.com/2016/http2-push-the-details/ (2016), last visited: 2017-03-01
18. Marx, R., Quax, P., Faes, A., Lamotte, W.: Concatenation, Embedding and Sharding: Do HTTP/1 Performance Best Practices Make Sense in HTTP/2? In: Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST17). pp. 160–173. INSTICC, ScitePress (2017)
19. Meenan, P.: Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index (2012), last visited: 2017-03-01
20. Meenan, P.: Webpagetest. https://webpagetest.org (2016), last visited: 2017-03-01
21. Mi, X., Qian, F., Wang, X.: SMig: Stream Migration Extension for HTTP/2. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT16). pp. 121–128 (2016)
22. Netravali, R., Goyal, A., Mickens, J., Balakrishnan, H.: Polaris: faster page loads using fine-grained dependency tracking. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16) (2016)
23. de Oliveira, I.N., Endo, P.T., Melo, W., Sadok, D., Kelner, J.: Should I Wait or Should I Push? A Performance Analysis of Push Feature in HTTP/2 Connections. In: Proceedings of the workshop on Fostering Latin-American Research in Data Communication Networks. ACM (2016)
24. Peon, R., Ruellan, H.: HPACK: Header Compression for HTTP/2. https://www.rfc-editor.org/rfc/rfc7541.txt (2015), last visited: 2017-08-07
25. de Saxcé, H., Oprescu, I., Chen, Y.: Is HTTP/2 really faster than HTTP/1.1? In: Computer Communications Workshops (INFOCOM), IEEE Conference on. pp. 293–299. IEEE (2015)
26. Varvello, M., Schomp, K., Naylor, D., Blackburn, J., Finamore, A., Papagiannaki, K.: Is the Web HTTP/2 Yet? In: International Conference on Passive and Active Network Measurement. pp. 218–232. Springer (2016)
27. Wang, X.S., Balasubramanian, A., Krishnamurthy, A., Wetherall, D.: How Speedy is SPDY? In: NSDI. pp. 387–399 (2014)
28. Wang, Z.: Navigation Timing API. https://www.w3.org/TR/navigation-timing (2012), last visited: 2017-03-01
29. Yue, C., Xie, M., Wang, H.: An automatic HTTP cookie management system. Computer Networks 54(13), 2182–2198 (2010)
30. Zarifis, K., Holland, M., Jain, M., Katz-Bassett, E., Govindan, R.: Making Effective Use of HTTP/2 Server Push in Content Delivery Networks. Tech. rep., University of Southern California, Networked Systems Laboratory (01 2017)
31. Zimmermann, T., Rüth, J., Wolters, B., Hohlfeld, O.: How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push. In: 2017 IFIP Networking Conference and Workshops (2017)