Is HTTP/2 the future of web browsing?

Matt Taylor

April 2018

Abstract

HTTP/2 is a new protocol that defines how web browsers communicate with websites. Standardised in 2015, it is a major revision of the widely used HTTP/1.1 protocol. In this project I will be exploring the advantages of using HTTP/2 over HTTP/1.1, such as faster page load times, bandwidth savings through improved encoding, and stricter security requirements. To investigate these improvements I have written a minimal web server that uses HTTP/2 and I report my findings on the performance of HTTP/2 compared to a commercial-grade HTTP/1.1 server. I also compare HTTP/2 to some competing protocols, concluding that HTTP/2 is advantageous enough that websites can only benefit from supporting it.

Contents

1 Introduction to HTTP

In 1996, the Hyper Text Transfer Protocol (HTTP) version 1.0 was standardised by Tim Berners-Lee and the Internet Engineering Task Force (IETF). This

protocol defines the precise way in which a 'client' (which is usually a web browser, such as Internet Explorer) can request resources stored on a remote computer known as a 'server'. Clients open a connection, send a 'request' for a given resource (identified by a URL), and the server sends a 'response' containing the requested resource. From the perspective of the HTTP protocol itself, the type of resource that is transferred is arbitrary: it could be a text file, an image file, or even some dynamically generated content such as the current time. HTTP is one of many network protocols that is used to transmit data between two computers, however when a browser is requesting a resource from a server this is done exclusively over HTTP.

The IETF issued a minor revision of the HTTP/1.0 specification, publishing the HTTP/1.1 specification in 1999 [1]. In 2015, the IETF finalised and published the HTTP/2 specification [2], which is a major revision of the HTTP protocol. As of January 2018, out of 2.8 million of the most popular websites, 5% fully supported HTTP/2 [3]. All major desktop browsers and most mobile browsers now support HTTP/2 [4].

This project will explore HTTP/2 and evaluate the benefits that HTTP/2 offers in comparison to HTTP/1.1 for both users and server operators. To assist me in this, I have programmed a web server from scratch that supports HTTP/2 which I will be referencing and using for demonstration purposes. The source code for the server can be found at [5].

2 Why does performance matter?

For companies like Google, who are estimated to service 40 million requests per second [6], small delays can incur very large costs. In 2006, they found that a delay of half a second on their search engine caused traffic and revenue from affected users to drop by 20% [7]. Amazon also found that every 100ms of delay on their website costs them 1% in sales [8]. This highlights just how important it is for users to have an experience that is as smooth as possible, and how important it is for large companies to use high performance software which takes full advantage of the hardware capabilities. Therefore, it is in the best interests of both users and website operators to push for the introduction of technological advancements such as HTTP/2 which was designed to address some of the performance issues that are present in previous iterations of the protocol.

3 A Changing Web

Back when HTTP/1.1 was standardised in 1999, web pages generally consisted of a single file that included text and some directives to indicate how the page should be laid out when displayed by a browser, possibly accompanied by an image or two. Looking at the Wikipedia home page from 2001 [9], we can see

that this is true: a single image is displayed at the top right hand corner of the page, and everything else is simple text styling and some hyperlinks to other similar pages. A possible layout file (known as an HTML file) that could be used to display a single image is as follows:

```
<img src="logo.png"/>
```

This is the exact data[1] that the browser might receive from a server after requesting the homepage of a website. As you can see, the `<img>` tag simply references an image (specifying the source at a location called `logo.png`); it does not provide the actual image data. The browser must now request the image from the location specified in the `src` attribute before it can be displayed. So, after the HTML has been retrieved, the browser initiates another request for the image at the URL `https://some-website.com/logo.png`. When this second request finishes, the browser takes the image data and displays it to the user.

Websites have significantly increased in complexity over the last two decades. For example, a full page load of `https://amazon.com/` consists of 173 requests, transferring over 3.3 megabytes of data [10]. However, forcing the browser to initiate too many requests can quickly introduce large delays: on a single connection, only one request may be outstanding at any time, meaning that if a browser needs to request two images it must wait for the whole of the first response before it can initiate the second request, as opposed to sending off the two requests and receiving the two responses in parallel[2]. In the former case, the total page load time is directly proportional to the number of requests made, whereas in the latter case the total page load time is independent of the number of requests and therefore scales much better on websites like Amazon that force the browser to start hundreds of requests upon every page load.

Due to this inability to process parallel requests on a single connection, browsers began doing the next logical thing: opening more connections. HTTP/1.1 specified a limit of 2 connections at a time [1, § 8.1.4], but today most browsers ignore this limit, using a maximum of 4 to 8 connections depending on the browser [12].

This is the first of the major differences between HTTP/1.1 and HTTP/2. In the former, parallelism is achieved through opening multiple connections to the server. In the latter, all operations are instead conducted over a single connection in such a way that requests and responses can be interleaved in whichever way is most convenient to the client or server [2, § 1]. In cases where the browser is performing more requests than the maximum number of connections it is permitted to open, using HTTP/2 can yield very high performance gains.

[1] Most likely in the ASCII encoding, which means that each character of text is represented by one byte, which is equivalent to a number between 0 and 255 inclusive.

[2] This issue is known as 'head of line blocking' and was already recognised as a problem when HTTP/1.1 was being standardised, but a method that was introduced to alleviate it was not implemented in any major browsers due to some misbehaving servers not handling these parallel requests properly [11].

Another reason why using one connection is preferable to using multiple connections is due to the time it takes to open a connection. If the client and server want to communicate using encryption, the client and server must perform a 'handshake' to negotiate the ciphers and cryptographic keys that will be used, which has been shown to increase the median time it takes to open a connection by roughly 3 times the round trip time[3] [13], potentially adding a 50 to 500 milliseconds delay depending on the distance between the server and the client.

As a practical demonstration of the performance gains that are to be made, I have set up a web page (using the web server I have created) at `https://mattst.me/tiles.html` which consists of a large image split into 180 smaller images. This forces the browser to very quickly max out the number of parallel requests that it can use, which gives HTTP/2 a significant advantage since by default there is no limit to the number of parallel requests that can be active at a time. You can click on the buttons to switch between using HTTP/1.1 and HTTP/2, and observe the time taken at the bottom of the page. On my computer, the 'time taken' to fully load the page varies with round trip time as follows (all results averaged over 5 readings):

Round trip time/ms	HTTP/1.1 time taken/s	HTTP/2 time taken/s
14	0.914	0.399
30	1.449	0.535
60	2.430	0.694
100	3.746	0.930
200	6.909	1.515
500	16.508	3.176
1000	32.577	6.205

Table 1: Mean page load time versus latency for HTTP/1.1 and HTTP/2
(Arch Linux x86_64, Chrome 64-bit 63.0.3239.108)

Here one can see that the head of line blocking effect in HTTP/1.1 becomes significantly more pronounced when high latencies are involved, since the client has to wait for full responses before sending off the next batch of requests. This feature is undoubtedly a great benefit to companies, especially given that there are now more people on mobile phones accessing the internet than there are people on desktop PCs accessing the internet [14], and the WiFi or 3G/4G connections that phones use often incur higher latencies and have a higher chance of data needing to be re-transmitted due to (among other things) poor reception [15].

Additionally, it may be interesting to consider the effects that this will have on the loading of advertisement. The Internet Advertising Bureau found that in February 2017, 22.1% of UK adults were using ad blocker software [16], which is software that blocks HTTP requests to servers that are known to distribute

[3]Also known as RTT, latency or ping: this is a measure of how long it takes for a single byte of data to be sent from a client to a server and back.

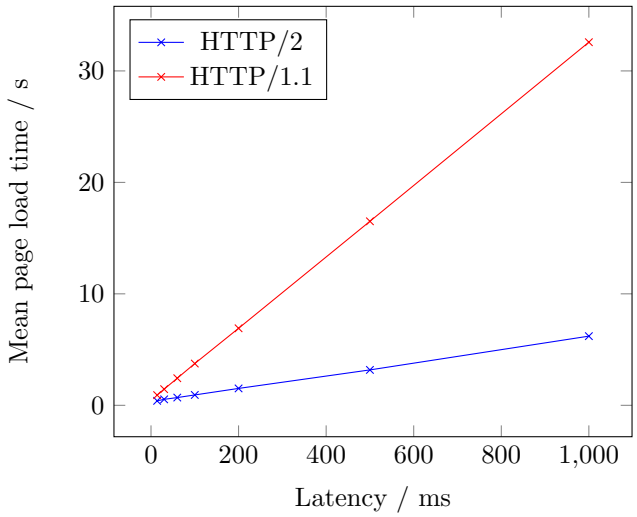

Figure 1: Mean page load time versus latency for HTTP/1.1 and HTTP/2 *(Arch Linux x86_64, Chrome 64-bit 63.0.3239.108)*

advertising. One reason for this is that blocking these unnecessary requests decreases page load time significantly [17], but with HTTP/2 this effect is reduced. Since ad blocking is forecast to cause revenue losses to UK publishers of £1.3 billion by 2020 [18], decreasing the appeal of ad blocking software is in their interest.

4 Pre-emptive Loading

Another feature of HTTP/2 is that responses can be sent before a corresponding request is received. On a typical website, the HTML file that the browser initially receives will contain a number of other resources (images, code to style the page, advertising) that also need to be requested before the page can be fully displayed. Assuming that all these other request can be done in parallel, this means that the time taken for the page to load is effectively twice that of the time taken to complete a single request.

Using HTTP/1.1, many websites avoided this problem by embedding the secondary resources inside the primary HTML file (known as 'inlining'), so that the browser does not need to make a second full request to get the extra resources. However HTTP/2 has a feature called 'server push' which allows a server to pre-emptively send the secondary resources at the same time as the primary resource, before having received the request for them. So for example on a HTML file that contained two images, upon receiving the request for that HTML file, the server would send both the HTML file and then the two images

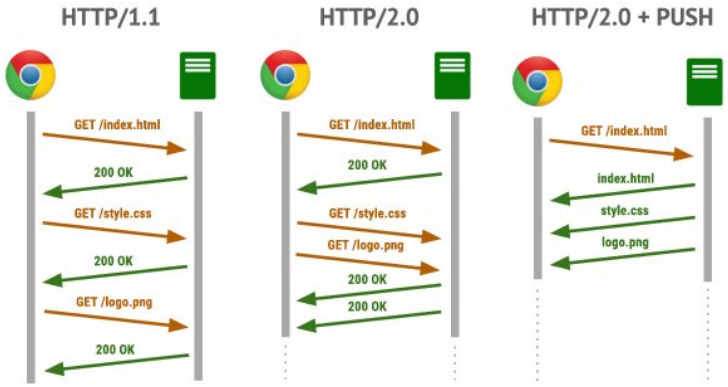

Figure 2: HTTP/2 server push saves round trips from the client to the server.

all at the same time[4], avoiding the time needed for the client to receive the response and dispatch another two requests for the images. The behaviour of HTTP/1.1 without inlining, HTTP/2 without server push, and HTTP/2 with server push is shown in figure 2.

For the web developer, this means that inlining is in theory no longer necessary, as server push achieves much the same function. Additionally server push is more 'cache friendly'. Browsers typically cache (meaning save locally) resources that they receive so that they are able to retrieve the file from the local cache instead of having to make another network request[5]. With server push, a server could in theory remember that a client has image X cached but not image Y, and send only image Y in the pre-emptive push. This is superior to inlining where both X and Y would get sent despite X being in the cache, using up unnecessary bandwidth. Additionally, some resources (like the results from a search engine) are dynamically generated and cannot realistically be inlined [19]. However with server push, the performance benefits are retained since the server can send the resource as it would any other. Overall, server push provides major benefits since it not only takes off some of the burden of optimising the web page from the designer, but also improves performance in the case where optimisation has been applied.

Unfortunately in practice it is difficult for servers to determine which resources the client currently has cached, and realistically not that many resources are cacheable. This means that if applied incorrectly, server push can even be detrimental to performance as the server can push resources that are already in the browser's cache, wasting unnecessary bandwidth [20]. For this reason, HTTP/2 server push support is not used by many servers (even though all major browsers support it [21]). In a study of 5.32 million HTTP/2 enabled

[4]The server operator would need to configure beforehand which secondary resources should be sent for a given primary resource.

[5]This is why pages load so fast when you press the back button in a browser

websites, only 595 supported server push on their landing page [20]. It seems that server push will remain largely unused in the future, although there is an extension to HTTP/2 [22] being introduced that allows for the client to more effectively inform the server which resources are cached, so that the server does not unnecessarily push already cached resources.

For this reason I have chosen not to implement server push in my web server, but you can find a demonstration at [23].

5 Improved Encoding

HTTP/1.1 is a text-based protocol, which means that all the request and response metadata uses a specific encoding called ASCII. This metadata[6] includes things such as the content length (how much data is about to be transmitted), the content type (whether the data is text, audio, video etc.), which type of device the request originated from (including OS and browser), and authentication data. In ASCII, each byte (8 'bits' – can also be thought of as a number between 0 and 255 inclusive) represents a single character of text (such as a letter or punctuation mark). For example the word 'HTTP' is encoded as the sequence of bytes 72, 84, 84, 80.

Additionally, any numbers that need to be sent (such as the content length) are also represented digit by digit: '1234' would be represented as 49, 50, 51, 52. Considering that each individual byte can have 256 discrete possible values, this is quite wasteful, since it means that a four-byte sequence can have $(2^8)^4 = 2^{32} = 4294967296$ possible values. All the numbers between zero and 1234 (inclusive) can be stored in $\lceil \log_2 1235 \rceil = 11$ bits, which is around a third of the length that it would be in the ASCII encoding.

As well as being very wasteful, text-based protocols in general suffer from issues surrounding handling of whitespace (which just refers to spaces, 'new line' or 'tab' characters) and capitalisation. Overall, text-based protocols are both wasteful in terms of bandwidth and are often slower for a computer program to parse [24]. The major benefit of using text protocols is that they are easier for humans to read which in turn makes them easier to debug, but in practice this is rarely useful (due to tools which can format the data for you).

Therefore HTTP/2 adopts a binary protocol (that is to say one that is merely not textual), and additionally adopts a header compression scheme known as HPACK [25] [7]. This is especially useful for large header fields such as authentication data – for example when you are 'logged in' to a website, often all that is happening is that along with every request, your browser is sending a secret

[6] Known as the 'headers', as opposed to the actual data being transferred which is known as the 'body'.

[7] HTTP/1.1 previously specified a method for compressing the *body* of requests and responses, which is retained in HTTP/2, but previously there was no mechanism for compressing the *headers*.

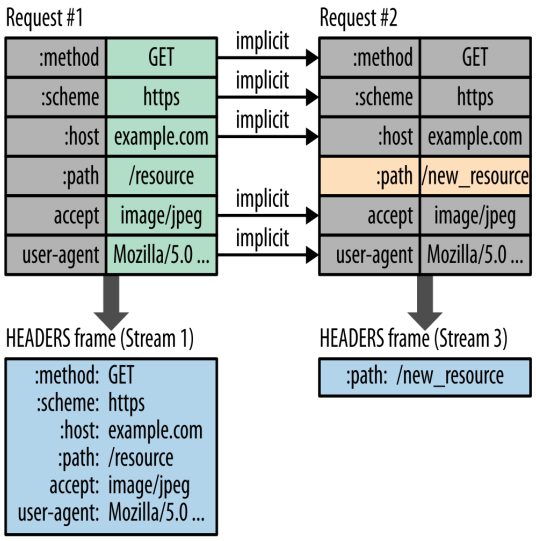

Figure 3: In HTTP/2, repeated headers are not re-transmitted in full across requests on the same connection.

authentication 'token' that it was issued by the server upon clicking the 'log in' button. Since such data may only change every few hours or days, it is very wasteful to repeat these tokens in full for every request, especially as they may be hundreds of bytes long. HTTP/2 instead defines a mechanism for one endpoint (which means a client or server) to specify which header fields should be stored between requests or responses, and upon subsequent transmissions the endpoint is able to refer to that value by only using a single byte, regardless of the original length of the data. This mechanism is known as the 'dynamic table', in contrast to the 'static table' (which is constant and predefined by the HTTP/2 specification) that includes some common header fields so as to not waste space in the dynamic table. This behaviour is depicted in figure 3.

Finally, HPACK also defines a static 'Huffman code' which is a primitive form of compression that involves replacing characters with variable-length sequences of bits such that the most frequent characters have shorter sequences and infrequent characters have longer sequences [26]. Since the 'A' character appears more frequently than the 'Z' character, it gets replaced by a specific sequence of 6 bits, whereas the 'Z' gets replaced by a specific sequence of 8 bits. Cloudflare, one of the first large companies to deploy HTTP/2, found that using this Huffman code alone (and even neglecting the use of the dynamic table for optimal compression) results in around 30% header size savings compared with HTTP/1.1 [27].

6 Security

HTTP/2 has a number of features which are designed to make it more difficult for an attacker to decrypt the communications between a browser and web server. As with HTTP/1.1, if secure communication is negotiated between the client and the server, the encryption and decryption is handled by a separate protocol called 'TLS' (Transport Layer Security)[8]. Much like HTTP/1.1, HTTP/2 permits transmission over unencrypted cleartext (called '`h2c`' when sending HTTP/2 traffic) or transmission over TLS (called '`h2`' when sending HTTP/2 traffic) [2, § 3.1]. Currently however, `h2c` is not supported by major browsers, meaning that TLS is a *de facto* requirement [28]. Of the 2.8 million most popular websites, only 1210 of them had true `h2c` support [3] as of November 2016. Since setting up TLS is relatively complicated, I would argue that this non-support is a benefit since it means that server operators can't simply pick `h2c` for their own convenience, which ultimately makes the web in general more secure.

The HTTP/2 specification also defines stricter requirements for what ciphers clients and servers are allowed to use within TLS. It has a minimum version requirement of TLS 1.2, which is negligible in impact since around 90% of TLS communication already uses this [29]. The specification also defines a cipher suite blacklist [2, §9.2.2], which prevents servers from using some ciphers that are deemed to have inadequate insecurity. This blacklist is unfortunately a very tough requirement to meet, which causes some HTTP/2 servers to ignore this list entirely. I have also experienced issues with TLS compatibility when connecting to my own web server from some devices. So overall these additional TLS requirements do not do much to improve HTTP/2's security.

The predecessor to HTTP/2, called 'SPDY', served as the basis for the HTTP/2 specification until its official standardisation in 2015. One of the major changes between SPDY and HTTP/2 is the header compression mechanism that is used. SPDY used an algorithm that was later discovered to be vulnerable to the 'CRIME' exploit (Compression Ratio Info-leak Made Easy) [30], which is essentially a chosen-plaintext attack[9]. HTTP/2 mitigates this issue by using a different header compression algorithm which is not vulnerable to such attacks. HTTP/1 has no such vulnerability, since it does not use header compression. Additionally, due to the request and response multiplexing that HTTP/2 now mandates, this in itself increases the difficulty of a chosen-plaintext attack (compared with HTTP/1.1), since network traffic is much less predictable.

HTTP/2 incorporates a method of padding data (which is simply the insertion of random bytes) which is designed to further reduce the possibility of a chosen-plaintext attack. However I would argue that this is of limited use, since this mechanism is already employed by TLS, and using padding bytes on top of this

[8] Formerly known as 'SSL' (Secure Sockets Layer).

[9] This involves an attacker guessing the contents of an encrypted message, and working backwards to derive the encryption keys in order to read all encrypted messages that are sent.

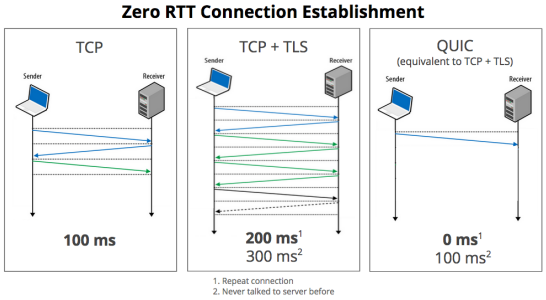

Figure 4: Round trips are decreased significantly in QUIC

leads to an unnecessary increase in network bandwidth.

Finally, on a more practical note, the advent of a new protocol leads to a greater attack surface due to the implementation of new software. The HTTP/1.1 security pitfalls are reasonably well known by developers and have been incrementally ironed out of software over the years. With HTTP/2, there is a greater possibility of programmer error simply because the software has not been around as long, so bugs are likely to exist in implementations. Inevitably over time this will improve, but it is certainly something to be wary of in the short term.

7 Alternatives To HTTP/2

In determining whether HTTP/2 is likely to increase in popularity in the future, it is important to take a brief look at some competing protocols.

One such protocol that I mentioned previously is called QUIC (Quick UDP Internet Connections) [31]. This aims to gain some performance benefits by combining 'layers' of protocols. Protocols usually do one specific function, so it is necessary to arrange them in sort of a stack in order to achieve the required functionality. For example, one protocol might be used to ensure that any data that is lost during transmission gets re-transmitted, another protocol might be used to encrypt data, and another protocol (such as HTTP) might be used to determine how to interpret the data that is received. Due to their modularity it is then possible to replace or remove different parts of that protocol stack with ease. A simple analogy for this would be an message in English - you could choose to write it down, send it via text, dictate it etc. You could also opt for using a different language, say French. The choice of language is totally independent of the method that is used for transporting the message (and *vice versa*), so any of these components can be swapped at will.

In the case of an encrypted HTTP/2 connection, there are essentially 3 protocols: TCP, TLS and HTTP/2. TCP is a protocol that guarantees that data arrives in the correct order and is re-transmitted if necessary, while TLS handles the encryption and HTTP/2 handles the semantics. QUIC essentially combines

these three protocols into one single protocol that still serves the same overall function. By combining them you do lose some modularity but some optimisations can be made as there are some redundant components. I mentioned previously that HTTP/2 has a mechanism for adding 'padding' bytes to data that is sent, in order to enhance the security. However TLS also has this mechanism, so it is effectively implemented twice. Another optimisation that can be made is the combination of 'handshakes'. A handshake is a pre-defined exchange of information upon initial communication. Each protocol does this differently, with HTTP/2 sending a specific ASCII-encoded string followed by an exchange of settings so that the other end knows which features of the protocol are in use. TCP and TLS both do something similar, however the TLS handshake cannot start until the TCP handshake is complete, and the HTTP/2 handshake cannot start until the TLS handshake is complete. If each handshake is taking a very optimistic 3 round trips, this would add up to 9 round trips in total, which can easily add up to a delay of over a second before any meaningful data is transmitted. Combining these handshakes (as seen in figure 4) can therefore lead to significant decreases in delays starting up connections, and through use of some other optimisations the handshake is effectively one round trip. However, as mentioned previously, starting new connections in the context of HTTP/2 is fairly rare anyway so the overall savings that accumulate over many requests are fairly negligible.

There are a number of other reasons why performance is greatly increased in QUIC but the empirical data does show a clear boost in performance. The Google homepage saw a 3.6-8% drop in search latency and a mean 3% decrease in the search page load time when using QUIC as opposed to normal HTTP. Google also reports that the slowest 1% of connections load 1 second faster using QUIC [32].

QUIC was initially designed by Google, and adoption of the protocol is fairly unimpressive. The only major browser to support QUIC is Google Chrome and server support for QUIC is around 0.8% of all websites [33] making up a respectable 7% of all internet traffic [34], with most of the popular websites that support it being ones owned by Google. This is perhaps the largest disadvantage of QUIC compared to HTTP/2 - it is not easy for server operators to add support for it due to the lack of both client and server support. Realistically, only large companies such as Google are going to be able to support such a protocol, at least in the short term while the software ecosystem for QUIC is relatively new.

8 My Experiences with HTTP/2

Note 5,000 word draft: I am a little unsure what to do with this section. It currently exists as sort of a miscellaneous collection of my thoughts, but it would also be possible for me to remove this entirely. Considering the marking criteria, I think it would be a good idea for me to focus on my own work more in some way, otherwise I only really have the page load time results and demonstration

to show for my work. I am aware that it does deviate from the 'scholarly' tone somewhat.

In this section I will briefly talk about some of my experiences and observations from writing a HTTP/2 server and what it means for the protocol as a whole.

First of all the specification [2] is well-written and almost entirely self contained, which is a major benefit to anyone who finds themselves writing an implementation of the protocol. There is only 1 verified erratum [35], which is a single typographic error.

Secondly, I found that most client side implementations I interacted with were robust. Most of them sent back useful error messages when I was doing something incorrectly and any issues I encountered never persisted longer than about a day of debugging due to this.

Most devices I tested my server on all interacted in the same manner, which is again a good thing. One of the problems with HTTP/1.1 is that there is such a wide range of behaviour across clients which means that in some cases you need to go against the specification in order to achieve interoperability - so having all implementations on the same page from the start is a benefit, and I only found one edge case where I decided to not follow the specification for my own convenience. I noticed that the school computers either blocked entirely or didn't support HTTP/2, which was disappointing, but I suspect this is due to them using a very old version of Google Chrome (pre-2014 at least).

Unfortunately it is quite difficult for me to discuss many of the details of my code or to analyse specific snippets since I chose to program it in C which is quite a low-level language (meaning that simpler tasks can often take many lines of code and involve much error checking). This is partly due to my comfortability with the language but also since I wanted to ensure that the server would run fast enough for me to get meaningful results from - I would not consider it to be a fair test if I was comparing a slow HTTP/2 implementation of mine with a highly optimised HTTP/1.1 implementation, so performance was certainly a goal of mine. I also chose C because I felt like I would learn more about low-level networking if I wrote it in C than if I had written it in a higher level language like Python.

The server (which I have named HH) is currently 2,437 lines of code and the source code can be found online at [5]. The majority of the actual code can be found in the `src` folder.

Overall I am very happy with the project and I would consider it a testament to the success of HTTP/2 that I have been able to create such a server from scratch with minimal prior experience and still demonstrate large performance improvements over highly optimised HTTP/1.1 implementations.

References

[1] Henrik Frystyk Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt.

[2] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://rfc-editor.org/rfc/rfc7540.txt.

[3] Matteo Varvello et al. "Is The Web HTTP/2 Yet?" In: *Passive and Active Measurements Conference (PAM)*. 2016.

[4] *Can I use... HTTP/2 protocol*. URL: https://caniuse.com/#search=http2 (visited on Jan. 10, 2018).

[5] Matthew Staveley-Taylor. *HH: An HTTP/2 server written in C*. URL: https://github.com/64/hh.

[6] *Google Data Center FAQ*. Mar. 16, 2017. URL: http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq (visited on Jan. 10, 2018).

[7] *Marissa Mayer at Web 2.0*. Nov. 9, 2006. URL: http://glinden.blogspot.co.uk/2006/11/marissa-mayer-at-web-20.html (visited on Jan. 10, 2018).

[8] *Amazon found every 100ms of latency cost them 1% in sales*. Aug. 13, 2008. URL: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (visited on Jan. 10, 2018).

[9] *Wikipedia: HomePage*. June 2001. URL: https://web.archive.org/web/20010727112808/https://wikipedia.org.

[10] *HTTP Requests Checker*. URL: https://www.giftofspeed.com/request-checker/.

[11] *Mozilla Bug 264354: Enable HTTP pipelining by default*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=264354 (visited on Jan. 4, 2018).

[12] Steve Souders. *Roundup on Parallel Connections*. Mar. 20, 2008. URL: http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/ (visited on Jan. 4, 2018).

[13] Ilker Nadi Bozkurt et al. "Why Is the Internet so Slow?!" In: *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings*. Ed. by Mohamed Ali Kaafar, Steve Uhlig, and Johanna Amann. Cham: Springer International Publishing, 2017, pp. 173–187. ISBN: 978-3-319-54328-4. DOI: 10.1007/978-3-319-54328-4_13. URL: https://doi.org/10.1007/978-3-319-54328-4_13.

[14] *Mobile and tablet internet usage exceeds desktop for first time worldwide*. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (visited on Feb. 16, 2018).

[15] *Ethernet vs WiFI: Ping, Packets & Playing better*. URL: https://na.leagueoflegends.com/en/page/ethernet-vs-wifi-ping-packets-playing-better (visited on Feb. 16, 2018).

[16] *Ad blocking levels have stabilised*. URL: https://www.iabuk.net/about/press/archive/ad-blocking-levels-have-stabilised (visited on Feb. 16, 2018).

[17] *Brave Mobile Speedtests*. July 2017. URL: https://brave.com/Brave_mobile_speedtests_July.pdf (visited on Feb. 16, 2018).

[18] *Ad blocking forecast to cost 35 billion by 2020*. URL: https://digiday.com/uk/uh-oh-ad-blocking-forecast-cost-35-billion-2020/ (visited on Feb. 16, 2018).

[19] *Caching Tutorial*. URL: https://www.mnot.net/cache_docs/ (visited on Apr. 17, 2018).

[20] Torsten Zimmerman et al. *How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push*. URL: http://dl.ifip.org/db/conf/networking/networking2017/1570332989.pdf (visited on Apr. 17, 2018).

[21] *HTTP/2 PUSH vs HTTP Preload*. URL: https://dexecure.com/blog/http2-push-vs-http-preload/ (visited on Apr. 17, 2018).

[22] K. Oku et al. *Cache Digests for HTTP/2*. URL: https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest-04 (visited on Apr. 17, 2018).

[23] *HTTP/2 Server Push Demo*. URL: https://http2.golang.org/serverpush (visited on Apr. 17, 2018).

[24] *HTTP/2 Frequently Asked Questions: Why is HTTP/2 binary?* URL: https://http2.github.io/faq/#why-is-http2-binary (visited on Feb. 15, 2018).

[25] Roberto Peon and Herve Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. May 2015. DOI: 10.17487/RFC7541. URL: https://rfc-editor.org/rfc/rfc7541.txt.

[26] David Huffman. *A Method for the Construction of Minimum-Redundancy Codes*. Sept. 1951. URL: https://www.ic.tu-berlin.de/fileadmin/fg121/Source-Coding_WS12/selected-readings/10_04051119.pdf.

[27] *HPACK: The silent killer feature of HTTP/2*. URL: https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/ (visited on Feb. 16, 2018).

[28] *HTTP/2 for Web Application Developers*. Sept. 2015. URL: https://www.nginx.com/wp-content/uploads/2015/09/NGINX_HTTP2_White_Paper_v4.pdf.

[29] *Qualys SSL Pulse*. URL: https://www.nginx.com/wp-content/uploads/2015/09/NGINX_HTTP2_White_Paper_v4.pdf (visited on Feb. 16, 2018).

[30] *The CRIME attack*. 2012. URL: https://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf.

[31] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-11. Work in Progress. Internet Engineering Task Force, Apr. 2018. 105 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-11.

[32] *Taking a Long Look at QUIC*. URL: https://lwn.net/Articles/745590://www.sjero.net/pubs/2017_IMC_QUIC.pdf (visited on Apr. 16, 2018).

[33] *Usage of QUIC for websites*. URL: https://w3techs.com/technologies/details/ce-quic/all/all (visited on Apr. 16, 2018).

[34] *QUIC as a solution to protocol ossification*. URL: https://lwn.net/Articles/745590/ (visited on Apr. 16, 2018).

[35] *RFC 7540 Errata*. URL: https://www.rfc-editor.org/errata_search.php?rfc=7540 (visited on Apr. 17, 2018).