An Analysis of HTTP/2: The Present and The Future

Anthony Nguyen - CSC579 - V00915696

April 2020

1 Introduction

The Hypertext Transfer Protocol (HTTP) is one of the most popular application protocols on the Internet since its first HTTP/0.9 version in 1991 with simplicity in mind. In 2009, Google introduced an experimental protocol SPDY as an alternative to HTTP to solve the low efficiency of HTTP [1]. HTTP/2 which is based on SPDY was standardized in 2015 to be the next major revision of the HTTP [2]. HTTP/2 inherits all advantages from SPDY and addresses several limitations in HTTP/1.1 as well as enhance the protocol performance and robustness.

In regards to the HTTP/2 adoption of websites on the Internet, a study in 2017 showed that there were only 6.43% websites in Alexa top 1 million sites support HTTP/2 protocol [needref]. Since then, there was no other study to update the HTTP/2 adoption of websites. The results are likely outdated and not useful anymore. Therefore, in this project, we attempt to conduct an investigation on the current HTTP/2 adoption and discuss the future of the HTTP/2 protocol. Contributions of the project are as follows:

1. Examine the current HTTP/2 adoption of Alexa top 511,850 websites.
2. Reveal useful insights and observations on how websites are using HTTP/2.
3. Discuss the development of HTTP/2 and its potential successor HTTP/3

The rest of this report is structured as follows: In section 2, we introduce background concepts of HTTP/2 and its features. We describe the implementation of the examination tool in section 3. Results and relevant observations are reported in section 4. In section 5, we discuss the current status quo and future development directions. Finally, we conclude the project in section 6.

2 Background

2.1 Core Concepts

HTTP/1.1 was officially released in 1997 and enabled the unprecedented growth of World Wide Web. Appearing in almost every internet device, HTTP has been serving the Internet well for over two decades. Nowadays, web applications become more complicated and are required to transfer a large amount of data reliably with as low latency as possible. Therefore, there were attempts to improve HTTP protocol.

HTTP/2, which was standardized in 2015, is the next major version of HTTP to improve web applications efficiency and reliability. The maintainer of HTTP decided to increase the version to 2 because HTTP/2 introduces a new binary framing layer that is not backward compatible with HTTP/1.x. Existing semantics of HTTP remain unchanged. HTTP/2 changes how messages are formatted, transmitted and processed. Additionally, HTTP/2 adds many more advanced features to current HTTP protocol such as but not limited to multiplexing, flow control, stream priority, server push, header compression, ping. We will now describe and discuss core concepts of the above features.

HTTP/2 defines a new term `stream`. A stream is a virtual channel which carries HTTP messages. According to the specification, one TCP connection serves all streams in a communication between a client and a server.

Multiplexing allows multiple streams on the same TCP connection. Technically, multiplexing eliminates the need of multiple connections and enables interleaving requests and responses in parallel without blocking others. Figure 1 illustrate the main difference between HTTP/1.1 multiple connections and HTTP/2 single connection.

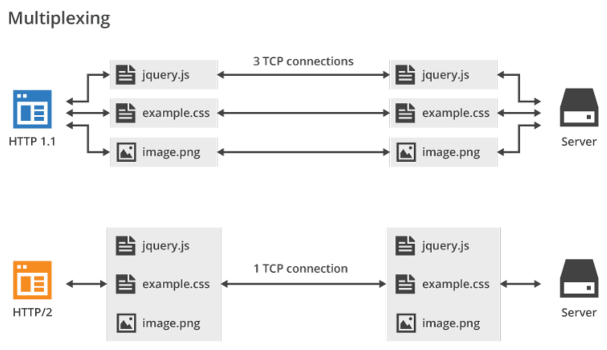


Figure 1: Comparison between HTTP/1.1 and HTTP/2 connection. Source: https://blog.cloudflare.com/content/images/2015/12/http-2-multiplexing.png

Flow control is a feature which uses window concept to limit how much data an endpoint can receive. This setting can happen at stream-level or the whole connection level, and each endpoint can define this parameter on its side. A study [3] examined a possible attack vector can be done by abusing flow control in HTTP/2 . Malicious clients can set a connection's window size to a very small value thus lead to a denial of service attack.

Stream priority allows clients to define a dependency graph indicates streams that they want servers to allocate more resources and response first. Further discussion on this feature is in section 5.

Server Push is a feature which was designed to allow server to preemptively send resources to clients to reduce latency. It is useful in case the requests are heavy and time consuming. Typically, developers need to implement additional techniques to send back data to client side. For instance, common techniques like short polling, long polling or a web socket are used to notify client about the response. However, the use of this feature is low compared to the HTTP/2 adoption rate. There are possible factors causing this such as: 1, Server Push is not mandatory when enabling HTTP/2 and 2, sub-optimal push strategies can slow down page load time [4]. Figure 2 shows a simple case where server proactively send css files to client without being requested. Server Push is quite useful in these cases where unnecessary latency is eliminated.

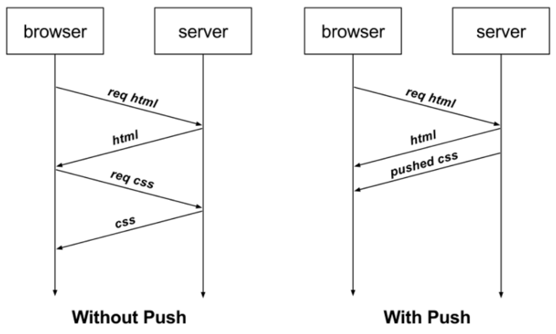


Figure 2: HTTP/2 with Server Push. Source: https://www.tutorialdocs.com/upload/2018/07/server-push-1.png

For header compression, HTTP/2 uses an algorithm named HPACK to compress header fields using Huffman encoding, and maintain a dynamic table on both client and server to cache frequent header fields. There was no header compression in HTTP/1.1. This made requests and responses in a web application contain redundant header fields increasing bandwidth consumption and round trip latency. SPDY addressed this concerned by introducing DEFLATE algorithm to compress header fields. HTTP/2 then presented a new dedicated to improve DEFLATE and eliminate the reported attack CRIME in DEFLATE [5]. In a nut shell, HPACK consists of three methods:

1. Static Table: This contains 61 frequently used header fields such as accept-encoding, content-type, method: GET, method: POST, etc.

2. Dynamic Table: This limited sized table is empty upon connection establishment. Frequent header fields are added to the table as the connection lasts.

3. Huffman encoding: This algorithm is applied to compress the header list.

HTTP/2 Ping was also introduced as a helper to measure round trip time (RTT) of the connection. It can be very helpful to keep a connection alive.

2.2 Adoption

In regards to adoption, we discuss the HTTP/2 adoption from three perspectives: end-users, developers and server maintainers. Typically, end-users do not aware of what happen on the high level, and what they might only notice is faster web page load. All the new, low-level framing is performed by the client and server. As of March 24, 2020, 96.77% of global users from desktop to mobile environment are using HTTP/2 supported web browsers [6]. In fact, web browser have been actively supporting HTTP/2 development since its early days.

Developers only need little knowledge and effort to enable HTTP/2 on web servers. They could just update the web server to a newer version and configure the correct settings without code changes in most cases. It's worth noting that HTTP/2 is fully backward compatible with previous HTTP versions. More particularly, the protocol negotiation process used in HTTP/2 will determine the appropriate protocol for the communication. However, in real system environment, HTTP/2 is not always better than an optimized HTTP/1.1 solution. Developers might need to pay attention to measure whether turning on HTTP/2 is giving better overall performance. A naive HTTP/2 enabling can cause significant problems that can lead to extra work needed to fix or even re-architecture application [7]. Additionally, to utilize advanced features such as Stream Prioritization or Server Push, developers need to spend effort to implement and test on both client side and server side.

From the server perspective, most of work needed to realize HTTP/2 resides here. Popular web servers and Content Delivery Networks (CDNs) such as nginx, Litespeed, CloudFlare, Netlify, etc have their own implementation of HTTP/2. Because they do all low-level work, developers do not really need to worry about the details unless they want to build a custom HTTP/2 implementation or client. Nevertheless, a concrete implementation on a CDN or server engine may be different from the others as long as it follows the HTTP/2 specification. E.g., a server engine can configure different setting parameters than default values, drop support for optional features or handle error frames in their own way.

3 Implementation

We wrote a program in Python to check the HTTP/2 support status of Alexa top 511,850 websites. All websites are sorted by rank in a csv file. Each line contains two parts separated by a comma: the rank of the site and its domain without http schema prefix (http:// or https://) e.g. facebook.com, google.com. The program uses nghttp2 command-line interface (CLI) to send and receive HTTP/2 requests. nghttp2 is one of the most efficient and mature HTTP/2 implementations [8]. Responses from websites are in form of strings. We obtain useful fields of this data using regular expressions, and then store results in a `sqlite` database for further analysis.

We use `ThreadPool` in Python speed up the experiment up to 5 requests/s. It means that it takes nearly 28 hours and 25 minutes to complete requesting all the sites. Not to mention when the Internet is unstable or the website is unreachable, we need to query failed sites in the database and run the tool again for those sites.

A lot of sites do not handle unknown HTTP protocols in the handshake phase. It caused the tool wait for several minutes to get an error response. To avoid this, we added a timeout 3s to nghttp command. There is a trade-off by doing this, for some sites, they responses the whole web page content and the end to end latency may be longer than 3s. If the timeout is large, it takes much more time to finish the experiment.

4 Results

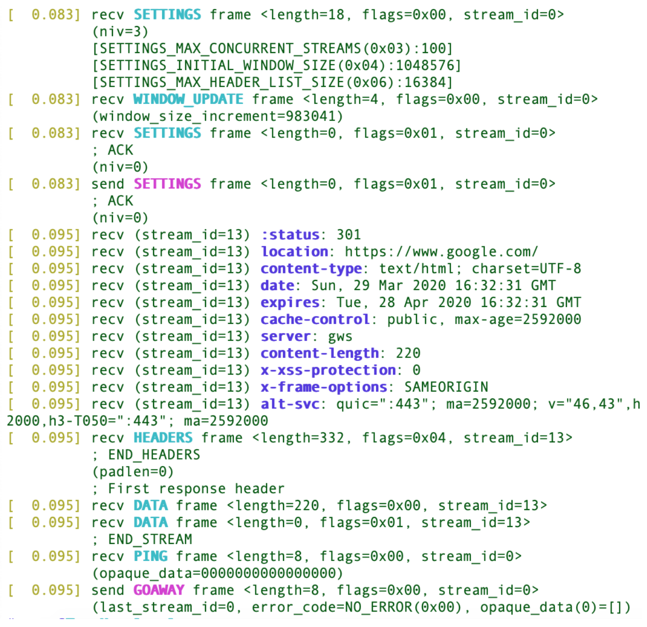
We assess a website is support HTTP/2 if it responses HEADER frames to a GET request from client side through Application-Layer Protocol Negotiation (ALPN). A sample response for a GET request from `google.com` is shown in Figure 3. If the response contains `recv HEADERS`, it indicates that protocol HTTP/2 is being used.

We can also extract other useful data such as `SETTINGS_MAX_CONCURRENT_STREAMS`, `SETTINGS_INITIAL_WINDOW_SIZE`, `SETTINGS_MAX_HEADER_LIST_SIZE`, `SETTINGS_MAX_FRAME_SIZE`, `SETTINGS_HEADER_TABLE_SIZE` and web server name.

4.1 Current Adoption

Apart from HEADERS frames, we record a website supports HTTP/2 if it uses ALPN as negotiation protocol. We observed many websites are still using SPDY or Next Protocol Negotiation (NPN) as negotiation protocol. Although some studies consider NPN as acceptable to support HTTP/2, we only consider ALPN. Firstly, NPN is not officially documented in HTTP/2 specification. Secondly, many web browsers and frameworks dropped NPN support [9, 10, 11].

We found that 285,450 (55.76%) of Alexa top 511,850 websites are supporting HTTP/2. In favour of top 1000, 584 websites are supporting HTTP/2.



```
[  0.083] recv SETTINGS frame <length=18, flags=0x00, stream_id=0>
          (niv=3)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):1048576]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):16384]
[  0.083] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=983041)
[  0.083] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.083] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.095] recv (stream_id=13) :status: 301
[  0.095] recv (stream_id=13) location: https://www.google.com/
[  0.095] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.095] recv (stream_id=13) date: Sun, 29 Mar 2020 16:32:31 GMT
[  0.095] recv (stream_id=13) expires: Tue, 28 Apr 2020 16:32:31 GMT
[  0.095] recv (stream_id=13) cache-control: public, max-age=2592000
[  0.095] recv (stream_id=13) server: gws
[  0.095] recv (stream_id=13) content-length: 220
[  0.095] recv (stream_id=13) x-xss-protection: 0
[  0.095] recv (stream_id=13) x-frame-options: SAMEORIGIN
[  0.095] recv (stream_id=13) alt-svc: quic=":443"; ma=2592000; v="46,43",h
2000,h3-T050=":443"; ma=2592000
[  0.095] recv HEADERS frame <length=332, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
[  0.095] recv DATA frame <length=220, flags=0x00, stream_id=13>
[  0.095] recv DATA frame <length=0, flags=0x01, stream_id=13>
          ; END_STREAM
[  0.095] recv PING frame <length=8, flags=0x00, stream_id=0>
          (opaque_data=0000000000000000)
[  0.095] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Figure 3: A response from google.com in nghttp2

4.2 Server

We recorded 1363 different server names in 201953 (39.45%) websites which returned server name in the response's header. However, the real number of server names is significant lower what we recorded because many web servers put the version after server name (e.g., nginx/1.14.0, openresty/1.15.8.2, Apache/2.4.41). We ignore versions as we do not intend to discuss different server versions. By removing the version info, there were 792 server names. Table 1 shows top 10 common servers.

Server Name	Count
cloudflare	70516
nginx	63421
Apache	37507
LiteSpeed	14547
GSE	7310
openresty	5538
Microsoft-IIS	5376
Squarespace	4138
ghs	1715
AmazonS3	1689

Table 1: Top 10 most common servers

In favour of servers used by top 1000 websites, the results are shown in table 2. There are some differences in what servers are popular in top 1000 websites. nginx is the most popular with 278 sites whereas cloudflare only ranks second with 199 sites. Interestingly, Tengine is one of the most popular server in the top 1000 sites. It is used by only Tmall, Aliexpress and some other Chineses websites. gws also appears in this list because Google homepage domains for countries are in the top 1000 such as google.com.hk, google.co.jp, google.cn, etc. Although Varnish is used by many top websites but it does not used widely.

Server Name	Count
nginx	278
cloudflare	199
Apache	91
Tengine	83
AkamaiGHost	47
gws	44
Varnish	37)
Microsoft-IIS	32
openresty	20
AmazonS3	17

Table 2: Top 10 most common servers in top 1000 websites

4.3 Settings

The SETTINGS frame is used to set configuration parameters that affect how clients and servers behave and handle resources. Both endpoints must exchange a SETTING frame to each other upon establishing a connection. Later on during communication, either endpoint can update setting parameters. In this project, we observed and retrieved the most 5 common parameters in a SETTING frame sent by servers. For each setting, we report results for two range of websites: one is the whole dataset and one is top 1000 sites. Because they receive much larger traffic than the rest We believe that it would be interesting and insightful to analyze top 1000 sites to see how they configure their servers.

1. `SETTINGS_MAX_CONCURRENT_STREAMS`: Indicates the maximum number of concurrent streams that the sender will allow. It is recommended to not be smaller than 100. The default value is infinity.

2. `SETTINGS_INITIAL_WINDOW_SIZE`: Indicates the sender's initial window size (in octets) for stream-level flow control. The default value is 65,535.

3. `SETTINGS_MAX_HEADER_LIST_SIZE`: Informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The default value is infinity.

4. `SETTINGS_MAX_FRAME_SIZE`: Indicates the size of the largest frame payload that the sender is willing to receive, in octets. The default value is 16,384 and the maximum should be allowed is 16,777,215 according to the protocol specification.

5. `SETTINGS_HEADER_TABLE_SIZE`: Informs the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The default value is 4,096.

Max Concurrent Streams	Count
100	225005
128	54940
256	2595
8	1309
250	1083

Table 3: The distribution of max concurrent streams values that are used by at least 1000 sites

For the parameter `SETTINGS_MAX_CONCURRENT_STREAMS`, we recorded 25 different values defined by websites. However, we only discuss values that are used by at least 1000 sites. It is unsurprising that 225005 sites are using the default value 100. In the rank 2nd and 3rd, many websites are using 128 and 256 respectively. Interestingly, there are 1083 sites are running with only max 8 concurrent streams. We found that all these websites are using Apache server. Perhaps that Apache version configured max 8 concurrent stream by default. Table 4 show the distribution of initial window size values. 137,881 (39.59%) servers use the default value whereas 40,167 sites (11.53%) adjusted the default value to 65,536. Interestingly, 6,047 sites configure a different initial window size value 1,048,576 which use Microsoft-IIS server.

Max Header List Size	Count
NULL	262,960
16,384	12,752
8,192	4,339
1,048,896	1,932
131,072	1,689
32,768	1,304

Table 5: The distribution of max header list size values that are used by at least 1000 sites

For the setting parameter `SETTINGS_MAX_HEADER_LIST_SIZE`, we recorded that 262,960 (92.12%) sites do not specify this parameter in the SETTINGS frame indicating that they use the default value - infinity. Table 5 shows the

Initial Window Size	Dataset
0	306
4096	1
8192	1
32768	213
65535	137,881
65536	40,167
128000	2
131072	5
262143	3
262144	3
1048576	6,047
5120000	2
8388608	1,350
33554432	2
2147483647	336

Table 4: The distribution of initial window size values in the whole dataset

distribution of max header list size values that are used by at least 1000 sites. Most of sites using max header list size of 16,384 use gws or Google Servlet Engine (GSE). Most of site using the 8,192 value use Squarespace server. The value 1,048,896 is used by many versions of nginx and openresty, whereas most versions of Netlify use the value 131,072. A majority of sites using the value 32,768 use AkamaiGhost server.

For the setting parameter `SETTINGS_MAX_FRAME_SIZE`, we found that 184,444 (64.61%) websites configure their server to support maximum possible frame size 16,777,215. Whereas 71,222 sites skip defining this parameter and 22,104 sites explicitly defines 16,384 in the SETTING frame which ends up 94,326 (33.04%) sites use the default value 16,384 in total. The more detail distribution can be found in table 6.

Conversely, 283,439 (99.29%) sites use the default parameter of setting `SETTINGS_HEADER_TABLE_SIZE`. There are 1,992 sites use the value 4,096, whereas 17 sites use the value 16,384. Only 1 site configures 5,096 and 1 site configures 65,536.

Max Frame Size	Count
16777215	184,444
NULL	71,222
16,384	23,104
65,536	3,706
1048576	1,931

Table 6: The distribution of max frame size values that are used by at least 1000 sites

5 The future of HTTP/2

5.1 The status quo

In this subsection, we discuss about the implementation of the HTTP/2 in real systems nowadays. From these discussions, we draw observations on the impact of HTTP/2 and its limitations.

Multiplexing attempts to resolve the downsides of TCP slow-start in HTTP/1.1. With TCP slow-start, a connection is initialized with a small window size which is increased over time as long as the connection is good [12]. Thus, a connection is likely to be congested initially. This issue also relates to head of line blocking where clients have to wait for the first request to complete before the next one could be sent [13]. HTTP/2 seems to solve these problems efficiently. Nevertheless, one might wonder if HTTP/2 performs better than HTTP/1.1 at every case. In fact, the answer is no, when there is packet loss on the network, it may perform worse than its predecessor. According to a test performed by Twilio, there are situations where HTTP/2 will perform worse than HTTP/1. They explained that: "Congestion controls at the TCP layer will throttle the HTTP/2 streams that are multiplexed within fewer TCP connections. Additionally, because of TCP retry logic, packet loss affecting a single TCP connection will simultaneously impact several HTTP/2 streams while retries occur. In other words, head-of-line blocking has effectively moved from layer 7 of the network stack down to layer 4" [14].

There are controversial discussions on Stream Prioritization efficiency. In fact, this is a complicated feature. Wijnants et al. [15] reported that designing such an optimal prioritization scheme is a non-trivial task. A sub-optimal algorithm can lead to slower page load time. There is no officially suggested algorithm for stream priority, this can cause difficulties for developers to implement it. Servers are also prone to algorithmic attacks where clients force servers to construct the dependency graph intensively. [16]. Many studies also attempted to examine implementations of popular web browsers [17] and web sever engine [18]. Prioritization in Chrome was the most effective one which is shown in Figure 4, whereas Firefox, Edge and Safari come with dependency graphs not as good as the one in Chrome or do not support prioritization at all [19]. On the other hand, most servers technically support HTTP/2 prioritization but there

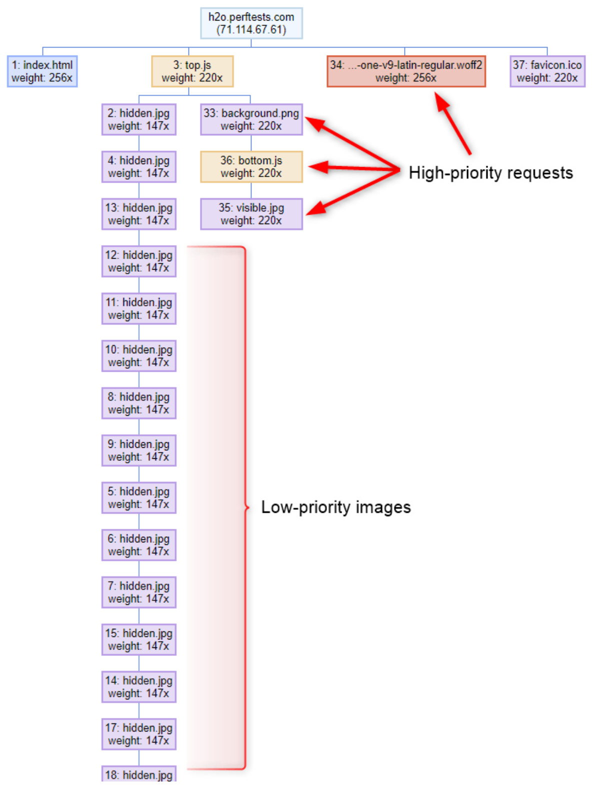

h2o.perftests.com (71.114.67.61)
1: index.html weight: 256x
3: top.js weight: 220x
34: ...-one-v9-latin-regular.woff2 weight: 256x
37: favicon.ico weight: 220x
2: hidden.jpg weight: 147x
33: background.png weight: 220x
4: hidden.jpg weight: 147x
36: bottom.js weight: 220x
13: hidden.jpg weight: 147x
35: visible.jpg weight: 220x
High-priority requests
12: hidden.jpg weight: 147x
11: hidden.jpg weight: 147x
10: hidden.jpg weight: 147x
8: hidden.jpg weight: 147x
9: hidden.jpg weight: 147x
5: hidden.jpg weight: 147x
6: hidden.jpg weight: 147x
7: hidden.jpg weight: 147x
15: hidden.jpg weight: 147x
14: hidden.jpg weight: 147x
17: hidden.jpg weight: 147x
18: hidden.jpg
Low-priority images

Figure 4: Stream Prioritization in Chrome. Adopted from [19]

is a need of extra tuning work to get it work efficiently [20]. That said, there is also good prioritization implemented in CDNs such as Akamai, Cloudflare, Fastly, etc, and developers can provide prioritization just by putting the website behind one of those CDNs.

In regards to HTTP/2 Ping, it is used to approximate the latency of MP-H2 - a client-only multipath solution for HTTP/2 [21]. The MP-H2's authors explained that HTTP/2 was a good choice to do that job because i. HTTP/2 Ping happens on application layer, the same layer as HTTP/2 messages, and ii. PING frames are negligible that they do not cause much impact on the overall system performance. Another good use case of this feature is to maintain connections on an Apple Push Notification Server (APNS) [22]. This can help avoid generating too many connections on the APNS which is considered as a denial of service attack.

As mentioned before, although HTTP/2 solves performance bottleneck on application layer, it operates badly when there is packet loss in the network due to congestion control in TCP. This is a motivation of recent improvements on TCP. TCP Fast Open (TFO) is one of researches to improve TCP. TFO was first introduced in 2011 as a method to speed up connection establishment between clients and servers by skipping a round trip delay. From an article by Craig Andrews [23], although being received support from popular operating systems such as Linux (2012), Android (2013), iOS (2015) and Windows 10 (2016), TFO is no longer enabled by default nowadays because of a variety reasons from integration to performance issues. The author also explained that HTTP/2 multiplexing does not work really well together with TFO.

Although HTTP/2 was not adopted widespread as expected, it has brought innovative ideas into the development of the internet. Its presence is result of a good approach from the HTTP maintainers and influences upcoming technologies to some extent.

5.2 QUIC and HTTP/3

In this subsection, we will briefly introduce QUIC and HTTP/3 as alternatives to HTTP/2.

QUIC is a transport layer network protocol introduced by Google in 2013 as an experiment to increase performance of network applications that are using TCP. QUIC can be considered a more efficient combination of UDP, TLS and HTTP/2. According to [24], QUIC still outperforms the protocol stack: TCP + TLS + HTTP/2 even with tuned variant of TCP and various optimizations. This is because QUIC not only inherits advantages of HTTP/2: header compression, multiplexing and server push but also addresses major limitations of the protocol. First of all, streams in HTTP/2 are prone packet losses which is discussed as the head-of-line blocking issue above because of TCP congestion control. Conversely, in QUIC, loss of a UDP packet only affect the streams contained within that packet. In other words, one slow request does not affect other requests. Secondly, thanks to UDP, QUIC is able to establish connection more quickly within fewer round trips. Although TFO promised to improve TCP connection establishment latency, it was not widely adopted as expected due to its change requirement at the OS level. This is not an issue to QUIC because QUIC is built at the software level, and it can deploy progressive updates quickly and effectively. For packet loss restranmission, QUIC implemented this feature at the level of QUIC. QUIC is also designed with security at its core i.e. encryption and authentication are provided by default. It is also proved that the latency to establish an encrypted session in QUIC is faster than that of TCP with TLS [25]. Nevertheless, QUIC still has its limitations need addressed. QUIC causes CPU overhead compared to HTTP/2 which may become a bigger problem on mobile devices. In addition, in network stacks with middle boxes like in enterprise network, UDP is usually throttled, leading to web performance degradation.

In 2018, the Internet Engineering Task Force (IETF) officially approved HTTP/3 which is former HTTP-over-QUIC to base on QUIC as the next major version of HTTP. This is a reimplementation of HTTP protocol using QUIC instead of TCP. HTTP/3 is subjected to remove or change some unnecessary or over complicated features in HTTP/3 such as HPACK (replaced by QPACK), stream prioritization. More particularly, stream prioritization in HTTP/3 is proposed with a more simpler scheme call "Extensible Prioritization Scheme for HTTP" [26]. Nevertheless, this is still an ongoing work and more improvements are expected to come in future. In terms of adoption, by the begin of 2020, HTTP/3 is implemented in stable Chrome build 79 and stable Firefox build 72. Cloudflare also started supporting HTTP/3 [27] and maintain open source HTTP/3 tools to grow the development community [28]. HTTP/3 is growing and being actively developed. More and more big players on the internet are supporting HTTP/3 for a faster, reliable and secure World Wide Web.

6 Conclusions

Since its first standardization in 2015, HTTP/2 has gained lots of attention and support. Many researches were also conducted to analyze and improve HTTP/2 further. We examined Alexa top 511,850 websites and found that 285,450 (55.76%) are supporting HTTP/2 properly at the time writing this project. We also reveal insightful observations from the examination results to see how websites are using HTTP/2, especially top 1000 websites that consume much larger web traffic than the rest. The tool we wrote for the experiment can be reused easily for an up-to-date analysis. Additionally, we discuss further the limitations of the HTTP/2 and why it is planned to be replaced by HTTP/3. HTTP/3 which is built on top of QUIC is proved to be much more efficient with lots of advantages and resolve many issues of HTTP/2. An analysis of HTTP/3 is left as a future work for this project.

7 References

References

[1] *SPDY: An experimental protocol for a faster web.* URL: `https://www.chromium.org/spdy/spdy-whitepaper` (visited on 31/01/2020).

[2] *Hypertext Transfer Protocol Version 2.* URL: `https://tools.ietf.org/html/rfc7540` (visited on 31/01/2020).

[3] *HTTP/2: In-depth analysis of the top four flaws of the next generation web protocol.* URL: `https://www.imperva.com/docs/Imperva_HII_HTTP2.pdf` (visited on 14/03/2020).

[4] T. Zimmermann et al. "How HTTP/2 pushes the web: An empirical study of HTTP/2 server push". In: *2017 IFIP Networking Conference (IFIP Networking) and Workshops.* 2017, pp. 1–9.

[5] *Wikipedia - CRIME.* URL: `http://en.wikipedia.org/w/index.php?title=CRIME&oldid=660948120` (visited on 08/04/2020).

[6] *Can I use HTTP/2 protocol.* URL: `https://caniuse.com/#feat=http2` (visited on 10/03/2020).

[7] *Why Turning on HTTP/2 Was a Mistake.* URL: https://www.lucidchart.com/techblog/2019/04/10/why-turning-on-http2-was-a-mistake/ (visited on 02/04/2020).

[8] *Nghttp2: HTTP/2 C Library.* URL: https://nghttp2.org/ (visited on 14/03/2020).

[9] *NPN support has been removed.* URL: https://www.fxsitecompat.dev/en-CA/docs/2017/npn-support-has-been-removed/ (visited on 24/03/2020).

[10] *API Deprecations and Removals in Chrome 51.* URL: https://developers.google.com/web/updates/2016/04/chrome-51-deprecations (visited on 24/03/2020).

[11] *Removing Legacy SPDY Protocol Support.* URL: https://webkit.org/blog/8569/removing-legacy-spdy-protocol-support/ (visited on 27/03/2020).

[12] *HTTP/2 Theory and Practice in NGINX Stable, Part 1.* URL: https://www.nginx.com/blog/http2-theory-and-practice-in-nginx-stable-13 (visited on 14/03/2020).

[13] *Remember HTTP/2?* URL: https://http3-explained.haxx.se/en/why-h2 (visited on 02/04/2020).

[14] *Discovering Issues with HTTP/2 via Chaos Testing.* URL: https://www.twilio.com/blog/2017/10/http2-issues.html (visited on 14/03/2020).

[15] Maarten Wijnants et al. "HTTP/2 Prioritization and its Impact on Web Performance". In: Apr. 2018, pp. 1755–1764. DOI: 10.1145/3178876.3186181.

[16] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12.* SSYM'03. Washington, DC: USENIX Association, 2003, p. 3.

[17] *http2priorities.* URL: https://github.com/pmeenan/http2priorities (visited on 28/03/2020).

[18] M. Jiang et al. "Are HTTP/2 Servers Ready Yet?" In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* 2017, pp. 1661–1671.

[19] *HTTP/2 Prioritization.* URL: https://calendar.perfplanet.com/2018/http2-prioritization/ (visited on 01/04/2020).

[20] *Optimizing HTTP/2 prioritization with BBR and $tcp_notsent_lowat$.* URL: https://blog.cloudflare.com/http-2-prioritization-with-nginx/ (visited on 24/03/2020).

[21] Ashkan Nikravesh et al. "MP-H2: A Client-Only Multipath Solution for HTTP/2". In: *The 25th Annual International Conference on Mobile Computing and Networking*. MobiCom '19. Los Cabos, Mexico: Association for Computing Machinery, 2019. ISBN: 9781450361699. DOI: `10.1145/3300061.3300131`. URL: `https://doi.org/10.1145/3300061.3300131`.

[22] *Communicating with APNs*. URL: `https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/CommunicatingwithAPNs.html` (visited on 20/03/2020).

[23] *The Sad Story of TCP Fast Open*. URL: `https://squeeze.isobar.com/2019/04/11/the-sad-story-of-tcp-fast-open/` (visited on 15/03/2020).

[24] Konrad Wolsing et al. "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC". In: *Proceedings of the Applied Networking Research Workshop*. ANRW '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 1–7.

[25] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. ISBN: 9781450346535. DOI: `10.1145/3098822.3098842`. URL: `https://doi-org.ezproxy.library.uvic.ca/10.1145/3098822.3098842`.

[26] *Extensible Prioritization Scheme for HTTP*. URL: `https://github.com/httpwg/http-extensions/blob/master/draft-ietf-httpbis-priority.md` (visited on 22/03/2020).

[27] *HTTP/3: the past, the present, and the future*. URL: `https://blog.cloudflare.com/http3-the-past-present-and-future/` (visited on 22/03/2020).

[28] *Savoury implementation of the QUIC transport protocol and HTTP/3*. URL: `https://github.com/cloudflare/quiche` (visited on 10/03/2020).