HTTP/2: Assumptions vs Reality for Mobile

Ethan Petuchowski
The University of Texas at Austin
ethanp@utexas.edu

1 ABSTRACT

HTTP/2.0's (H2) performance characteristics over mobile are not well understood. We test both the validity of its underlying assumptions and the performance benefits available via existing implementations. In particular, we evaluate H2's design decision to multiplex all data transfer between server and client through a single TCP connection. We also evaluate performance differences between HTTP/1.1 (H1) and H2. We find that H2 does not live up to its goals of measurable improvement over H1 via an LTE connection, and the single TCP connection may claim part of the blame. We suggest that further optimizations such as server push and QUIC may be necessary to obtain noticeable improvements.

Keywords: HTTP, mobile phones, WiFi, cellular networks, LTE, TCP.

2 INTRODUCTION

2.1 Beginning with HTTP/1.1

2.1.1 HTTP/1.1 is from an earlier Web

For the past several years, Google, Mozilla, Akamai, the Internet Engineering Task Force (IETF),[1] the academic research community, and others have been engaged in efforts to reduce page load times (PLT)[2] experienced by users of the World Wide Web (Web). One bottleneck in apparent need of an update is the now-"ancient" HTTP/1.1 (H1), first defined in (IETF RFC 2068, 1997). H1's design decisions are anachronistic given the realities of the modern Web.

At the time of its inception, the major design goal of HTTP was simplicity to implement and adopt. This was to encourage growth of the Web, which at the time was one of multiple competing applications built atop the Internet. Evidently, this technique worked, and the Web's meteoric growth in popularity is unprecedented.

However, over the past 18 years, the way people create, distribute, and view pages on the Web has changed drastically. Pages today contain more Javascript, CSS, images, and other content to go along with the vanilla HTML (de Saxce, 2015). In the course of this evolution, unnecessary latency introduced by the numerous round-trips H1 requires to fully download the data for each web page has caused page load time to suffer. Nowadays, increasing bandwidth is not nearly as important to PLTs as reducing latency (Grigorik, 2013). Web businesses know that reduced PLTs have been repeatedly shown to translate to more content being seen and products being bought.

In response to evolving web pages, over the past 16 years, countermeasures, "optional" protocol alterations, and application level workarounds have been suggested and implemented to reduce the round-trip count and therefore latency of pages retrieved using H1. These will be discussed in more detail below.

2.1.2 How does HOL blocking impact HTTP?

Head-of-line (HOL) blocking reduces performance in situations in which multiple messages are being sent through a first-in-first-out (FIFO) ordered message queue. If the message at the front of the queue takes a long time to be prepared for transfer, the messages queued behind it must wait in order to maintain FIFO. If the messages do not actually need to be FIFO ordered, this increase in transfer latency is unnecessary.

Since H1 transfers requests and responses over TCP as plain ASCII-encoded text, no method of multiplexing multiple responses was ever added to the protocol. In fact, there are *four different ways* of specifying the end of an HTTP response, and they require text parsing, which has led to situations in which two implementations are incompatible for communication due to parsing conflicts.

Note that TCP provides reliable, ordered, error-checked byte delivery; i.e. it *is* a FIFO-ordered message queue. Now suppose the H1 server is sending three resources in response to a client request, and the first one is a large image file (e.g. JPEG), and the other two are small text files (e.g. HTML & Javascript). Because of HOL blocking, the small text files will not be sent until the image is read from disk and reliably sent through the socket, even though the user would rather just receive the text files first and partially render the content. H2 will solve this problem as we'll discuss later.

2.1.3 HTTP/1.1 Page Load Time optimizations

Optimizations servers and browsers use to lower PLTs perceived by clients include the following (Grigorik, 2013):

1) **Open multiple TCP connections** to request and download multiple required web page resources in parallel.

a. This would prevent the HOL blocking scenario discussed above if (e.g.) the small text files were

[1] The IETF is a standards organization for the Internet, which produces "RFC"s (requests for comment) specifying some of the crucial Internet protocols, such as HTTP, TCP, and TLS.

[2] The duration between the time at which a user submits a request for a web page, and the time at which the last byte of data needed to represent that web page correctly is received.

downloaded over a different TCP connection than the large image. However, this further optimization is not performed.

2) **Inlining and concatenating** scripts and stylesheets to reduce the total number of requests.

3) **HTTP pipelining** — multiple requests are sent by the browser over a single TCP connection before waiting for responses to any, then the server sends all of the responses back in the same order.
   a. This was abandoned by major browsers because the maintaining ordering of responses increased problems with HOL blocking; and it introduced issues with older proxies.

2.2 Enter HTTP/2.0

As mentioned above, H1 being an ASCII protocol makes it difficult to simply and precisely specify how to demultiplex responses. To address this difficulty, HTTP/2 (H2) is a *binary* protocol that can easily de/multiplex multiple requests and responses through a single TCP connection. It does this using a new networking layer added to the stack: the *binary framing layer*. This layer breaks individual HTTP requests and responses into separate *frames* and interleaves them through the TCP socket. In this way, from the application's perspective, H2 retains the interface and semantics that have become so pervasive across the Web (e.g. `GET`, `POST`, etc.).

These *frames* are packaged up with relevant frame-header information to make them easy to parse and disambiguate (e.g. length in bytes), and sent one-at-a-time through a *single* persistent client–server TCP socket per client. This is in contrast to the first PLT optimization listed above, in which *multiple* TCP connections are used in parallel.

The frames associated with a single resource are called a *stream*. Frames belonging to a stream are sent in-order through the socket. To demultiplex a potentially large number of streams of incoming data, the receiver reads the header information off the socket, and uses that to correctly parse the rest of the frame's (binary) content and order it after previous data received for that stream.

Web designers have discovered that to yield the best user experience, the server should transmit certain most-important resources first, as soon as they are available to send (e.g. loaded from disk). This is why H2 frames (and streams) have (optional) priority and dependency fields to allow the application developer (or server writer) to bias the ordering of frames sent through the socket using a *priority queue* and *dependency tree*. For example, one may want to ensure that HTML files have a higher priority than JPEG files because JPEGs are large and not as crucial for showing the user a barebones version of the page they requested, alleviating the issue presented above.

2.2.1 Brief history, from SPDY to beyond

The current H2 specification (IETF RFC 7540) is based primarily on the SPDY protocol developed at Google. Google holds a unique position to conduct research on this topic because they write both browsers and popular web sites. This enables them to conduct true *controlled* experimental studies on *user* traffic. Also, Google and Akamai both stand to drive increased business value over a faster Web.

Many have voiced concerns that 'giant corporation' Google's ideas were welcomed too quickly by the IETF standards committee. However, supporters were eager to get the ball rolling on anything that would stimulate more research, and Google already had experimental *results*.

As of November 2015, in my Chrome browser, using the "HTTP/2 and SPDY indicator" browser extension, I can see for example that Twitter currently loads over H2. The "indicator" also reveals that Google's search results and YouTube videos are served over SPDY version 3 on top of their still-in-research QUIC protocol. QUIC is a protocol over UDP, meant to replace TCP as the transport layer protocol beneath HTTP. Sadly, time and space constraints do not permit further exploration of the QUIC protocol.

2.2.2 Why open only *one* TCP connection?

TCP was developed specifically to optimize throughput for full-duplex connections between two hosts over the Internet. The fact that implementers of H1 resort to multiple TCP connections is as a *hack* to reduce the HOL blocking that would occur with a single pipe. However, this misuse of TCP leads the separate TCP connections to compete with each other for bandwidth. In addition, each must do its own handshake, go through the Slow Start and ramp up to full network utilization, and when they produce congestion, each leads the others to drop packets and cut their throughputs at unpredictable intervals. On top of that, for TLS connections, each must go through the TLS handshake's two additional round trips. A single long-lived TCP connection is used in H2 to alleviate these problems, while the HOL blocking is taken care of by the framing layer. We will explore the impact of this in our experiments.

2.3 HTTP for mobile

A growing share of people's access to the Web is mediated by mobile devices (de Saxce et al., 2015). The long list of requests–response pairs required by H1 for modern websites especially damages performance on mobile phones because cellular network connections typically experience longer latencies (Sommers and Barford, 2012). Sommers and Barford found via large-scale analysis that WiFi provides better (factor of two) and more consistent throughput than cellular, and WiFi provides lower but *less* consistent latency than cellular.

3 PREVIOUS WORK

Please note that some results in the papers cited here were obtained with SPDY, not H2, but both protocols are essentially the same. A common trope in the papers states that research is still needed to uncover the practical impact of H2. We look to test the impact of H2 *on* mobile phones over wireless connections. Few papers focus on H2 for mobile, and we couldn't find any with experiments run on an *actual* mobile phone as part of their testbed.

Among the research papers published so far on the topic, the most influential appears to be(Wang et al., 2014). This paper comes to the same conclusion yielded by most researchers: the results have been contradictory and major wins for H2 on normal (non-pathological) websites are elusive. They found that H2 generally outperforms H1, except when transmitting large objects over a high-loss connection.

One of the goals in defining H2 is that it is supposed to be easy to swap-in to replace H1. The findings of (Wang et al., 2014) and others indicate that especially with respect to mobile phones, the H1 optimization techniques of inlining and concatenating web page resources is the wrong way to improve performance for H2. This suggests that H2 will not quite be as simple as a swap-in replacement for H1, because other elements of the application server pipeline will have to be altered as well.

The IETF working group for H2 ("httpbis") has a charter that suggests the desire for mobile users to see "measurable" improvements to perceived latency via "improved use of TCP" [Fig 1].

> It is expected that HTTP/2.0 will:
>
> - substantially and **measurably improve** end-user **perceived latency** in most cases, over HTTP/1.1 using TCP
> - Not require multiple connections to a server to enable parallelism, thus **improving its use of TCP, especially regarding congestion control**
>
> The resulting specification(s) are expected to meet these goals for common existing deployments of HTTP; in particular, Web browsing (desktop and **mobile**), non-browsers, [etc.].
> (extraneous text removed; IETF, 2012)

Figure 1: H2 was designed with mobile in mind

However, (Erman et al., 2013) found several problems with H1 with respect to mobile devices that are not addressed by H2. Most crucially and most widely documented, the TCP congestion window part of Congestion Avoidance (viz. `cwnd` and `ssthresh`) makes it so that dropping a single TCP datagram is assumed to be due to network congestion, so TCP tries to avoid further congestion by exponentially reducing throughput. In reality though, the packet could have been dropped for reasons specific to transfer over the shared wireless medium (viz. multipath propagation, short term fading, hidden terminals, etc.).

Secondly, bad interactions between both the 3G and LTE MAC state machine timeouts, and modern default TCP timeouts and mechanisms, lead to datagram retransmissions with further resets on congestion window state variables, resulting in a drastically negative impact on throughput (Erman et al., 2013).

(de Saxce et al., 2015) served downloaded copies of Alexa's top globally popular websites. They found (like Wang et al.) that H2 outperforms H1 more for loading websites that involve a larger number of requests, both over WiFi and 3G connections. They also found that the higher packet loss of 3G does lead to lower benefits of switching H2, but H2 still usually outperforms H1 over 3G because the packet loss doesn't outweigh the benefits. However, unlike our experimental setup, they connected to 3G using a computer, whereas we are connecting to LTE using an actual smartphone.

3.1 Further research is needed

From the system administrator's perspective, it is still difficult to decide whether enabling H2 on one's servers is going to have a positive impact on performance at all (Varvello et al., 2015). According to the research we could find, the following questions still do not have good enough answers to make answering that decision easy.

1) What changes to the infrastructure and optimization pipeline are needed to provide a significant benefit over H1?
2) What are the best algorithms for leveraging H2's new capabilities for server push and prioritization? (de Saxce et al., 2015)
3) Why do experiments by different research groups yield such a wide disparity in results? (Wang et al., 2014)
4) Are we unlikely to see any real PLT improvements until we improve the underlying transport layer protocol? (Google QUIC)
5) In what ways is the problem different for mobile? (Erman et al., 2013)
   - I.e. where interference, low bandwidth, high latency, packet loss, and battery life considerations are major concerns?

4 EXPERIMENTS

We set up a mobile client and a static server to run benchmarking of the new protocol. Jetty was found to be the most suitable way to implement the testbed's HTTP server. This is because it is a popular server framework that provides the same API to secure H2 and secure H1, and also implements both of them over the same basic pipeline infrastructure. The intention is that these features make the performance comparison more *"fair"*, although there is perhaps no objective measure of performance evaluation equitability in this context.[3]

The Jetty server runs on a 2014 Apple MacBook Pro laptop with 16GB RAM. The laptop is connected to the Internet through a standard wireless home router. (Sommers and Barford, 2012) analyzed 3 million `Speedtest.net` results from around the world and found only very mild difference in average speed at different times of the day for both WiFi and cellular connections. Simple experimentation indicates the same conclusion about our connections.

`Speedtest` Results	Ping (ms)	Download (Mbps)	Upload (Mbps)
Laptop WiFi	10-13	120-150	23
iPhone WiFi	23	14	22
iPhone LTE	49	33	12

Table 1 Experimental Component Connection Speeds according to `Speedtest.net`

For the mobile half of the testbed, we use an Apple iPhone 6S running iOS 9.1. For the "WiFi" test condition, we are connecting the mobile phone to the same wireless local area network (WLAN) as the server. For cellular, we use the AT&T LTE network. Under both the WiFi and "cellular" test conditions, we connect the client to the server at the LAN's external IP address, with port-forwarding configured on the local router. The Speedtest results are listed in [Table 1].

4.1 Effect of single TCP over wireless

First, we shall test whether mobile experiences the same expected issues noted in a non-mobile setting over a wireless connection, viz. the situation in which using one TCP connection makes TCP's congestion reduction algorithm reduce overall bandwidth too drastically compared to multiple parallel connections.

[3] All code used in this paper is available at https://github.com/ethanp/wireless-http2-experiments

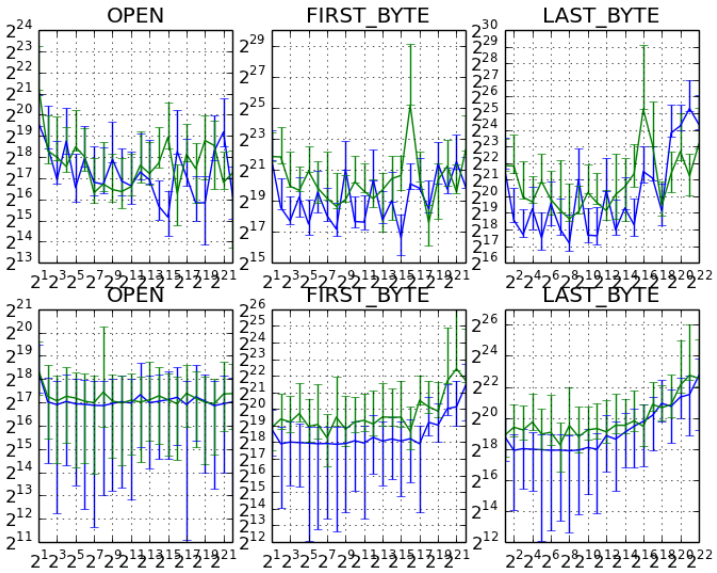

Figure 2 **TCP Connection Experiment Results**. **x-axis** is number of bytes transferred, **y-axis** is number of microseconds (10^{-6}) from connection initiation until event. **Events** are listed above each graph. **Green** line is for five concurrent connections, **blue** is for single connection. **Line** is average over four trials; **upper** and **lower** bars are maximum and minimum. **Top** row of graphs is over WiFi, **bottom** row is over LTE.

4.1.1 Experimental Setup

Our test involves downloading a set number of bytes to an iPhone from a Java SE 8 over TCP. This is done when the phone is connected via either a household-grade IEEE 802.11n at 5 GHz, or AT&T's LTE network in Austin. We open either 1 or 5 simultaneous TCP connections to the server. On the server side, each connection is fed random bytes by a different Java thread. This is basically a Java implementation of the UNIX `netcat` program. On the client side, we install an event handler for `NSStream` events for all (1 or 5) connections on the `mainRunLoop`. This event handler receives events when

a) **`OPEN`** — The input `NSStream` is *open* (implying that the TCP connection is *open*),
b) **`FIRST_BYTE`** — The 1st byte is *available* to be read by the application from `NSStream`'s buffer,
c) **`LAST_BYTE`** — The last byte is read from `NSStream`

We record the time it takes for these 3 events to happen from the time the connections are initialized.

Based on previous research, we expect to see that for transmitting a small total number of bytes, a single connection should be faster because of the reduced hand-

shake time. We also expect that for larger transfers, multiple connections will be faster because of the bad interactions between wireless performance and TCP's assumptions stated above.

4.1.2 Results

Our results agree with the conclusions of (Sommers and Barford, 2012) that latency is higher over cellular than WiFi, but latency *variance* is higher over WiFi.

Second of all, our results suggest (similar to previous studies using laptop-based hosts) that for small total transfer sizes, using one TCP connection leads to lower average download times than five concurrent TCP connections. This can be seen in the `LAST_BYTE` column of graphs in [Fig 2]. In a sense, this validates one of the main assumptions on which the H2 protocol is based. However, the improvement is negligible over LTE for transfers larger than 1KB, and performance is *worse* over WiFi for transfers larger than 250KB.

From the `OPEN` column of graphs in [Fig 2], we can see the connection initiation latencies. For the five connection case, we are looking at the average time to finish the TCP handshake for *each* of the five connections. Our results indicate that opening five connections simultaneously does *not* significantly affect handshake time per-connection.

From the `FIRST_BYTE` column of graphs in [Fig 2], we see that on average, when five TCP connections are open, it takes measurably longer for the iPhone operating system to buffer and pass the first bytes received from the network interface to a particular connection in the transport layer. The reason for this is not clear, but worth noting because it reveals hard evidence of the advantage of H2. This difference could be due to specifics of the iOS implementation, or interactions between the concurrent TCP connections.

4.2 H1 vs H2 example page Benchmark

Akamai's H2 benchmark page[4] is (roughly) an HTML page consisting of references to a bunch of images. This is a paragon of the advantages of H2, because it requires many separate requests from the client in order to download all data required to display the webpage. Over WiFi, iOS 9's Safari loads this page 3x *slower* over H2 than H1. Over LTE, Safari loads this page 5x *faster* over H2 than H1. Meanwhile, from the MacBook laptop, H2 outperforms H1 by >4x.

However, using Akamai's demo does not give us sufficient visibility into what is happening on the server side, so we implement a similar benchmark page server.

4.2.1 Experimental setup

We started with a test program to verify that at our discretion, our iPhone CocoaTouch code will indeed connect to our Jetty server via H2 when we want it to. iOS 9 makes H2 the default application layer protocol used when HTTP requests are made using the new `NSURLSession` API and infrastructure. Our session object is configured for benchmarking purposes to not cache results (or cookies or credentials), use at most 5 underlying TCP connections per host for requests over H1, and disable pipelining. Unfortunately, iOS 9's `NSURLSession` does not support the *server push* feature of H2 (Ishizawa, Twitter, 2015), so we were not able to benchmark its effects.

The Java 1.8.0_25, Jetty 9.3.6 server uses a self-signed certificate to serve both H1 & H2 requests over TLS. This meant we were not permitted to use Apple's `UIWebView` browser API because its internal security chain cannot be overridden. `NSURLSession` does allow overriding the authentication procedure, so we implemented a simple browser simulation using that API. This "browser" sends an HTTP request for a given URL, parses the response text (assumed to be HTML) to find `<img src="…">`, and subsequently sends HTTP requests for the URLs in quotes in those tags. These requests reuse the same `session` object, so they will occur over parallel TCP connections under the H1 condition, or be multiplexed through a single connection under H2; this is just like a standard Web browser would do.

Our experiment tests the total time between the initial request for the base URL from the `session` object, and when the download completion handler is called by the `session` object for the last of the 50 embedded images. The entire page is 785 KB in size, of which the HTML (which must be downloaded alone and first) is 1 KB.

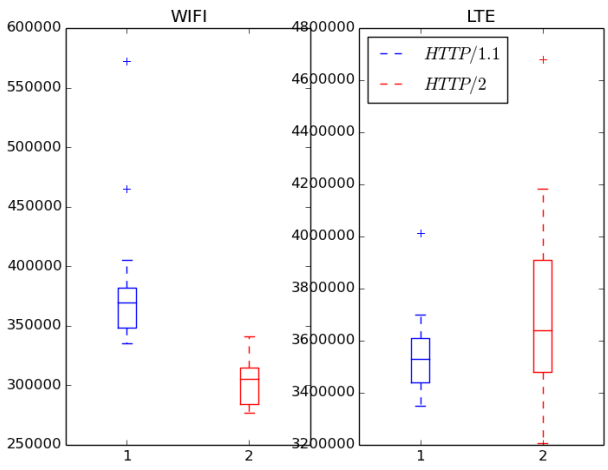

Figure 3 HTTP 1 vs 2 Experiment Results. **y-axis**: total microseconds (10^{-6}) to download benchmark webpage HTML and associated image files. Boxplot of 20 trials for each of the four conditions.

4.2.2 Results

We can see from [Fig 3] that these results are nominally *opposite* from the results we obtained from the Ak-

[4] https://http2.akamai.com/demo

amai benchmark, and the differences between H1 and H2 are notably less stark. It was possibly less stark because the Akamai page is 1.5 MB, roughly 2x the size of our page, because it's embedded images are bigger. It is difficult to investigate the Akamai case further without more information about the Akamai server setup. Being *Akamai*, it is probably aggressively cached at edge servers, and optimized in many ways that our home-built server is not.

Furthermore, it is quite interesting that the results in [Fig 3] do not agree with the raw TCP test for transfers of this size (~2^{19} bytes) above. For that test, we saw that 5 connections outperformed 1 on WiFi but not LTE. From this, one would predict H1 to outperform H2 on WiFi but not LTE, the opposite of what did occur. This is difficult to reconcile, and may simply be an artifact of the unpredictability of wireless connections. Alternatively, it could have to do with the fact that our webpage was not a raw 785 KB transfer, but was 51 separate request–response pairs, with 50 of size ~16 KB and 1 of size 1 KB. This means that both uplink and downlink data was being used, in contrast to the TCP benchmark. In retrospect, it would have been helpful to include a separate benchmark for uplink event latencies as well, so that we could get further insight into this discrepancy.

5 CONCLUSION

This research is the first we know of that runs experiments comparing H1 & H2 on a real smartphone. Our results show that using currently available technology, H2 does not offer a material performance gain over H1 over wireless connections to an iPhone 6S. We see this as evidence that H2 needs to offer more drastic changes with respect to H1 to make a noticeable improvement.

The server push technology already included in the H2 spec is one potential source of major performance improvements that we did not leverage because it is not yet implemented in iOS 9. If Apple does include this feature in a future version of `NSURLSession`, its impact should be further explored.

Not surprisingly, people assume that after a long time without much change to the HTTP protocol, it must be ripe with low-hanging fruit (Jetty Forum). Unfortunately this does not seem to be the case.

Browser vendors have been anxious to publish and implement H2 as the new HTTP standard. This is probably because H2 does better against H1 on a PC.

QUIC will be the protocol to watch out for in the future, as its cross-layer optimizations may yield the substantial improvements the world is waiting for.

6 BIBLIOGRAPHY

[1] Erman, Jeffrey, et al. "Towards a SPDY'ier mobile web?." Proceedings of the ninth ACM conference on Emerging networking experiments and technologies. ACM, 2013.

[2] Fielding, R., et al. "IETF RFC 2068 hypertext transfer protocol-HTTP 1.1, 1999."

[3] Grigorik, Ilya. High Performance Browser Networking: What every web developer should know about networking and web performance. "O'Reilly Media, Inc.", 2013.

[4] IETF, *charter-ietf-httpbis-07*. Available at http://datatracker.ietf.org/wg/httpbis/charter/. Last updated October 12, 2012.

[5] Jetty Forum, http://comments.gmane.org/gmane.comp.ide.eclipse.jetty.devel/1935

[6] Padhye, Jitu, and Henrik Frystyk Nielsen. *A comparison of SPDY and HTTP performance*. Microsoft Technical Report MSR-TR-2012-102, 2012.

[7] J. Roskind. Multiplexed Stream Transport over UDP, 2013. [*QUIC*]

[8] Roth, Gregor. "HTTP/2 for Java Developers". Retrieved from http://www.javaworld.com/article/2916548/java-web-development/http-2-for-java-developers.html on October 8, 2015.

[9] de Saxce, Hugues, Iuniana Oprescu, and Yiping Chen. "Is HTTP/2 really faster than HTTP/1.1?" *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*. IEEE, 2015.

[10] Sommers, Joel, and Paul Barford. "Cell vs. WiFi: on the performance of metro area mobile connections." *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012.

[11] Stenberg, Daniel aka. bagder. *http2 explained*, retrieved October 8, 2015; downloaded as MOBI.

[12] Ishizawa, Twitter, 2015 https://twitter.com/summerwind/status/608592686648414208?ref_src=twsrc%5Etfw

[13] Varvello, Matteo, et al. "To HTTP/2, or Not To HTTP/2, That Is The Question" arXiv preprint arXiv:1507.06562 (2015).

[14] Wang, Xiao Sophia, et al. "How speedy is SPDY." Proc. of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI). 2014.