Are HTTP/2 Servers Ready Yet?

Muhui Jiang*, Xiapu Luo*§, Tungngai Miu†, Shengtuo Hu*, and Weixiong Rao‡
*Department of Computing, The Hong Kong Polytechnic University
†Nexusguard Limited
‡School of Software Engineering, Tongji University

***Abstract*—Superseding HTTP/1.1, the dominating web protocol, HTTP/2 promises to make web applications faster and safer by introducing many new features, such as multiplexing, header compression, request priority, server push, etc. Although a few recent studies examined the adoption of HTTP/2 and evaluated its impacts, little is known about whether the popular HTTP/2 servers have correctly realized the new features and how the deployed servers use these features. To fill in the gap, in this paper, we conduct the first systematic investigation by inspecting six popular implementations of HTTP/2 servers (i.e., Nginx, Apache, H2O, Lightspeed, nghttpd and Tengine) and measuring the top 1 million Alexa web sites. In particular, we propose new methods and develop a tool named H2Scope to assess the new features in those servers. The results of the large-scale measurement on HTTP/2 web sites reveal new observations and insights. This study sheds light on the current status and the future research of HTTP/2.**

I. Introduction

To remedy the shortcomings in HTTP/1.1, the dominating web protocol, IETF published the standard of HTTP/2 (i.e., RFC 7540 [1]) based on the SPDY [2] protocol proposed by Google in 2015. HTTP/2 introduces a number of new features that will lead to faster web applications, more efficient transmissions, and better web security. In particular, HTTP/2 supports request multiplexing that permits sending multiple HTTP requests over a single TCP connection, thus mitigating the head-of-line blocking (HoLB) problem. In addition, the client can prioritize the multiple HTTP requests to the same domain and alleviate the effect of HoLB issue. Using binary format instead of plain text, HTTP/2 simplifies the implementation. Furthermore, it can drastically reduce header size by using header compression.

More and more web sites are adopting HTTP/2. A recent measurement on the top 1 million Alexa web sites reported the H2 announce rate increased by 50% from Nov. 2014 to Aug. 2015 and HTTP/2 could reduce page load time [3]. They further created a web [4] to show daily statistics about the number of web sites supporting HTTP/2 and the reduction in page load time due to HTTP/2.

Although these studies were based on the deployed HTTP/2 web servers, little is known about whether these servers have carefully realized the new features in HTTP/2 and how the servers use these features. In this paper, we conduct the *first* systematic investigation to fill in the gap by inspecting six popular implementations of HTTP/2 servers (i.e., Nginx, Apache, H2O, Lightspeed, nghttpd and Tengine) and measuring the top 1 million Alexa web sites. More precisely, we propose new methods to characterize how the servers support six new features, including multiplexing, flow control, request priority, server push, header compression, and HTTP/2 ping. We have realized these methods in a tool named H2Scope for conducting the large-scale measurement. The measurement results uncover new observations and insights. For example, not all implementations strictly follow RFC 7540 as shown in Table III. Some new features, like server push and priority mechanism, have not been well implemented and fully used by web sites. One possible reason is that RFC 7540 does not provide detailed instructions for realizing such features. Moreover, some features (e.g., flow control, priority mechanism, etc.) may be exploited by adversaries to launch DoS attacks. In summary, our major contributions are as follows:

1) To our best knowledge, we are the first to investigate the new HTTP/2 features realized in servers.
2) We design a set of new methods to characterize how these features are realized and used in HTTP/2 servers.
3) We not only examine six popular implementations of HTTP/2 server (i.e., Nginx, Apache, H2O, Lightspeed, nghttpd and Tengine) but also conduct a large scale measurement on the top 1 million Alexa web sites with new observations and insights.

The rest of this paper is organized as follows. Section II introduces the background knowledge about HTTP/2 with an emphasise on its new features. We detail our assessment methods in Section III and the implementation in Section IV, respectively. The extensive evaluation results are reported in Section V. After introducing the related work in Section VII, we conclude with future work in Section VIII.

II. Background

In this section, we first introduce the basic concepts in HTTP/2 (Section II-A), and then detail the new features under examination (Section II-B - Section II-G).

A. Basic Concepts

HTTP/2 re-uses the same application semantics of HTTP/1.1, such as, the HTTP methods, status codes, header fields, etc., but changes the way how the requests and responses are formatted, transmitted, and processed [1], [5].

HTTP/2 adopts binary framing instead of delimited plain text to delivery messages. A *frame* is the basic unit of communication with a 9-byte frame header. There are ten types

§ The corresponding author.

of frames serving different purposes. Each HTTP *message*, including a request or a response, comprises one or more frames. HTTP/2 introduces the concept of *stream*, which could be regarded as a virtual channel carrying messages exchanged between two ends. Each stream has an unique identifier embedded in the frame header. A client and a server establish one TCP connection shared by all streams.

B. Request Multiplexing

HTTP/2 achieves request multiplexing by having each HTTP request and the corresponding response in one stream and letting streams be independent of each other by default. Hence, a blocked request or a stalled response will not impede the progress of other streams. Since an endpoint may not be able to handle many concurrent streams, HTTP/2 allows either endpoint to adjust the SETTINGS_MAX_CONCURRENT _STREAMS parameter in the SETTINGS frames, which indicates the maximum number of concurrent streams to be supported by the sender.

C. Flow Control

Although request multiplexing allows the server to handle requests simultaneously, multiple streams may cause contention over the underlying TCP connection, thus leading to blocked streams. HTTP/2 provides the flow control to mitigate such effect by using the WINDOW_UPDATE frame. A receiver announces how many data they will receive in the WINDOW_UPDATE frame. Note that only DATA frames are subject to the flow control according to RFC 7540. HTTP/2 applies the flow control to both individual streams and the whole connection. For the former, the WINDOW_UPDATE frame's stream identifier is set to the ID of the affected stream. For the latter, the stream identifier is set to 0. The receiver of the WINDOW_UPDATE frame must respect the flow-control limits set by the sender. Note that the initial window size for the connection is 65,535 and it can only be changed by sending WINDOW_UPDATE frames according to RFC 7540.

D. Stream Priority

The priority mechanism in HTTP/2 is designed for distributing more resources to important streams. For example, the server can select streams for sending frames according to the priority under limited transmission capacity. Note that priority is just a suggestion to the server that will determine the processing and the transmission order of all streams.

A client can add the prioritization information in the HEADERS frame when opening a stream or use the PRIORITY frame to modify a stream's priority. There are two approaches for a client to prioritize a stream. One is to let the stream be dependent on another stream, which is called the parent stream. In this case, the parent stream will be given more resource. The other one is to assign a relative weight to streams that are dependent on the same parent stream. In this case, the resources will be allocated proportionally according to the weights of streams having the same parent stream. The PRIORITY frame can be used to change the stream priorities.

Based on the prioritization information, the client can construct a prioritization tree indicating how it would like to receive the responses. The server will generate the same tree and then use this information to allocate resources, such as CPU, memory, etc.

E. Server Push

HTTP/2 proposes server push to enable a server to speculatively send data to its clients, which is expected to be requested by the client. This mechanism could reduce the potential latency on the cost of some network usage. For example, after receiving and processing a request from a client, the server may infer that the client will also need some additional resources (e.g., figures), and therefore send them to the client.

The server will first send a PUSH_PROMISE frame to the client, which includes a header block with a complete set of request header fields. Moreover, the PUSH_PROMISE frame also includes the promised stream identifier for a new stream, through which the server will push the data to the client. The client can refuse the pushed response by sending a RST_STREAM frame or prevent the server from creating a stream for pushed data by advertising a SETTING_MAX_CONCURRENT_STREAMS with value zero.

F. Header Compression

HTTP/2 uses HTTP header compression (HPACK) [6] to compress header fields and pack them into HEADERS or PUSH_PROMISE frames for reducing the overhead and improving performance. Two techniques are adopted to compress the header fields. One uses the Huffman code to encode the header fields that have not been seen before. The other one helps the client and the server maintain and update a compression context that records previously seen header fields. Then, the compression context is employed as a reference to encode and reconstruct header keys and values. In particular, HTTP/2 defines a static table listing common HTTP header fields. Moreover, both ends maintain a dynamic table that is initially empty and updated according to the exchanged data within the entire connection. The data that are already in the static or dynamic tables will be replaced with the table indexes.

G. HTTP/2 PING

HTTP/2 proposes the PING frame, a new type of frame for measuring the round trip time (RTT) and conducting liveness checks. Either ends can send a PING frame without ACK flag, and the other end must reply a PING frame with ACK flag and identical payload. Moreover, HTTP/2 suggests giving PING response higher priority than any other frames. The payload size of PING frame is fixed to 8 bytes.

III. Measurement Methodology

It is worth noting that RFC 7540 neither describes how to determine whether a server supports these new features nor details how to implement them. We design measurement approaches, detailed in the following subsections, to characterize how remote servers realize the new HTTP/2 features.

A. Testing Request Multiplexing

1) Multiple Requests: HTTP/2 empowers the server to process multiple requests at the same time. To verify this feature, H2Scope will send N requests to the server simultaneously, each of which will download a large file to be carried by several DATA frames. Note that N is less than the value of SETTINGS_MAX_CONCURRENT_STREAMS announced by the server. If the server handles these requests in parallel, we will observe interleaved responses from N streams. Otherwise, we will receive all response frames for the ith request before getting response frames for the $(i+1)$th request. This approach may not work for the servers that do not have large web objects or can quickly finish processing each request. The reason is if the server quickly processes each request and sends back the responses (e.g., the requested web object is small), we will observe that the response frames follow the same sequence as the requests even if the server supports request multiplexing. Therefore, we only conduct such experiments in the testbed where large web objects are located in the HTTP/2 server.

2) Maximum Concurrent Streams: Both the server and the client can limit the number of concurrent streams using the SETTINGS_MAX_CONCURRENT_STREAMS parameter in the SETTINGS frame. RFC 7540 suggests that this value should not to be smaller than 100. The rationale is that a limited number of streams may delay the downloading of web objects because it is very common that a web page contains tens of embedded web objects. Therefore, in the measurement, we will record the value of SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame from servers.

B. Checking Flow Control

While flow control is designed for preventing one end from overwhelming the other end, an adversary could exploit such features to launch Denial-of-Service (DoS) attacks on the other end, such as preventing the other end from sending back the responses in order to consume its memory. We test whether a client can control the DATA frames and the HEADERS frames from a server and set incorrect values into window update.

1) Controlling DATA Frames: HTTP/2 employs connection level and stream level flow-control window and the WINDOW_UPDATE frame to realize the flow control. We verify whether a server will adjust the frame size according to the flow-control window. More precisely, we change the initial window size of each stream to a small value, denoted as S_{frame}, by setting the value of SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame. Then, we send requests and check whether the payload size of the response is equal to S_{frame}. Note that if a server allows a client to limit its frame size to a very small value, the server is vulnerable to DoS attacks.

2) The Effect of Zero Initial Window on HEADERS Frames: RFC 7540 specifies that *only* DATA frames are subject to the flow control mechanism. In other words, the server could still send back other frames. To check the compliance of HTTP/2 servers, we first set the value SETTINGS_INITIAL_WINDOW_SIZE to 0, and then send a request through HEADERS frame to the server. If the server follows RFC 7540, it will send back a HEADERS frame without any DATA frames, showing that the request has been processed but the data cannot be returned.

Algorithm 1 Measuring Priority Mechanism

```
1. session ← HTTP/2 session
2. settings ← initial window size = 2^31 − 1
3. url ← the front page of the domain
4. received_length ← 0
5. request_num ← 1
6. submit_settings_frame(session, settings)
7. submit_headers_frame(session, url)
8. on_frame_recv_callback:
9. if frame type == data frame then
10.         received_length += frame.length
11.         if frame.end ==True then
12.                 request_num = 65535/received_length + 1
13.         end if
14. end if
15. for i=0;i lt request_num-1;i++ do
16.         submit_headers_frame(session, url)
17. end for
18. sleep(3)
19. for i=0;i lt request_num;i++ do
20.         submit_rst_frame(session, url)
21. end for
22. submit_header_frame(session, stream A)
23. submit_header_frame(session, stream B)
24. submit_header_frame(session, stream C)
25. submit_header_frame(session, stream D)
26. submit_header_frame(session, stream E)
27. submit_header_frame(session, stream F)
28. submit_priority_frame(session, Frame 1)
29. sleep(3)
30. submit_window_update_frame(session, 2^31 − 1)
```

3) Zero Window Update: Since zero window size increment is meaningless, the receiver of a WINDOW_UPDATE with value 0 should regard it as a stream error according to RFC 7540. We send such kind of frames to the server and check whether it will reply with RST_STREAM frame.

4) Large Window Update: Besides rejecting zero window update, the sender should also prevent the flow-control window from exceeding 2^{31}-1 bytes. If this happens, the sender must terminate either the stream or the connection as suggested by RFC 7540. To verify such behavior, we will send more than one WINDOW_UPDATE frames and let the summarization of the window size increment be larger than 2^{31}-1.

C. Characterizing Priority Mechanism

1) Stream Priority: HTTP/2 provides the priority mechanism for a client to suggest a server how to allocate resources to concurrent streams. Since RFC 7540 does not specify how to realize the priority mechanism, different servers may have diverse strategies.

It is non-trivial to remotely infer whether or not the server has realized the priority mechanism because we could only observe the responses from the server and their order may be affected by many factors (e.g., response size, flow control mechanism, the dynamics of the dependency tree in the server, etc.). For example, if stream A has higher priority than stream

B, stream A should be allocated more resource and hence its packets will arrive at the client side before that from stream B. However, if there is no enough stream window for stream A, server will start to handle stream B instead of waiting for the window update frames on stream A. In this case, we will first observe packets from stream B instead of that from stream A. To tackle this problem, we propose a novel measurement approach, as shown in Algorithm 1, to determine whether a remote server has realized the priority mechanism.

Our approach consists of three steps. First, we will prepare the server's context for the testing (line 6 - 21). Second, we send M requests with stream priority information (e.g., stream dependencies, weights, etc.) to let the server build the dependency tree for these requests (line 22 - 27). We also send the PRIORITY frame to test the functionality of reprioritization (line 28). Third, we allow the server to send back the responses and analyze the order of HEADERS frame or DATA frame to infer whether or not the server supports the priority mechanism (line 30). We detail these steps as follows.

In the first step, we prepare the server's context to avoid the disturbance from the flow control mechanism and the arrival sequence of requests. More precisely, the flow control mechanism may let dependent streams be processed first. For example, if the flow-control window of a parent stream has been used up, this stream will be blocked, and the server may first process its dependent streams that have enough flow-control window. To address this issue, we set the value of SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame to be a very large value(e.g. 2^{31} - 1) so that each stream has enough flow-control window to send data(line 2 - 6).

The second disturbance may come from the arrival sequence of requests. Since the server usually adopts the first-come-first-served (FCFS) policy to process requests, it may process the requests one-by-one if it can quickly finish all of them. In this case, we can neither check whether the server takes into account the stream dependencies among requests nor verify whether the server supports re-prioritization. To address this issue, we first block all requests that are used for conducting the priority testing, and then change their dependencies through the HEADERS frames and PRIORITY frames before allowing the server to send back the response data.

We exploit the connection level flow control mechanism to block all requests used for testing the priority mechanism. According to RFC 7540, the initial window size for both connection and stream level is 65,535, and the initial window size for new streams can be changed by setting SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame. However, the connection flow-control window can only be adjusted by sending WINDOW_UPDATE frames. Note that if the connection flow-control window becomes zero, none of the streams can send DATA frames even if the stream level flow-control window is sufficient. Taking advantage of this feature, we first deplete the initial HTTP/2 connection level window (i.e., 65,535 octets) by downloading objects from the server (line 15 - 16). We calculate how many streams are required in the callback function (line 9 - 14). Once this window is used up, we reset these streams to avoid any interfere.

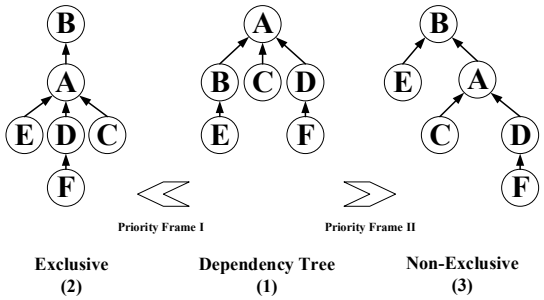

Fig. 1. Changing the stream dependency tree.

In the second step, we send M requests with pre-defined stream dependencies to the server. Since the connection level flow-control window has been drained, the server cannot send back the corresponding DATA frames. Our measurement result (Section V-D) shows that some servers even do not send back the HEADERS frames if the connection level flow-control window is zero. In the mean time, the server will build up the dependency tree for these streams. To check the functionality of re-prioritization, we send the PRIORITY frames to change the dependency tree.

One example is demonstrated in Figure 1. The original dependency tree is shown in the sub-figure (1). Stream A is the parent stream of streams B, C, and D. Streams E and F are the dependent stream of streams B and D, respectively. The server will construct such a dependency tree after we send the frames according to the information in Table I. If we send a PRIORITY frame to the server, the dependency tree will be changed. For example, if the PRIORITY frame follows the information in the first row of Table II (i.e., exclusive flag is set to true), the dependency tree will become the one shown in the sub-figure (2). If the PRIORITY frame uses the setting in the second row of Table II (i.e., exclusive flag is set to false), the dependency tree will become the one shown in the sub-figure (3). The major difference between these two dependency trees is the new parent stream of stream B. If the PRIORITY frame has the exclusive flag, stream A will become the sole dependency of its parent stream (i.e., stream B), and other dependent streams become the dependencies of stream A.

TABLE I
STREAM DEPENDENCIES.

Stream	Parent Stream	Weight	Exclusive
A	0	1	false
B	A	1	false
C	A	1	false
D	A	1	false
E	B	1	false
F	D	1	false

TABLE II
SETTINGS OF THE PRIORITY FRAME FOR THE EXAMPLE IN FIG.1.

Index	Stream	Parent Stream	Weight	Exclusive
1	A	B	1	true
2	A	B	1	false

In the last step, we send a WINDOW_UPDATE frame to increase the connection level flow-control window to a very

large value (e.g., $2^{31}-1$) so that the DATA frames in all streams can be sent back. By analyzing the DATA frames and HEADERs frames if any, we could infer whether the server has supported the priority mechanism.

2) Depending on Itself: A stream should not depend on itself. RFC 7540 suggests that an endpoint must treat such dependency as a stream error and reply with RST_STREAM. To check whether a remote server is compliant to the standard, we send a PRIORITY frame to make a stream depend on itself, which is named as self-dependent stream, and then observe the server's response.

D. Measuring Server Push

Since server push is optional, when scanning a web site, we first enable this functionality by setting the value of SETTINGS_ENABLE_PUSH in the SETTINGS frame to 1, and then browse different pages. If we receive the PUSH_PROMISE frame, the server will start server push.

E. Assessing Header Compression

HTTP/2 adopts HPACK to compress headers for decreasing the transmission time. Besides the static table, both endpoints will maintain the same dynamic table for tracking and recording previously seen key-value pairs. For a HEADERS frame, if a value is in the static or dynamic table, it will be replaced by the corresponding table index.

Since the static table is fixed, we will check whether a server properly realizes the dynamic table. More precisely, we send H identical request headers to the server. Since both endpoints will learn and index the values from previously seen HEADERS frames, the HEADERS frame of the first response should be usually larger than that of the following responses because the repeated values are replaced by indexes.

We define a compression ratio to evaluate the HPACK mechanism realized by a server as follows:

$$r = \frac{\sum_{i=1}^{H} S_{header}^{i}}{S_{header}^{1} \times H}, \tag{1}$$

where H is the number of response HEADERS frames and S_{header}^{i} denotes the size of ith response HEADERS frame. If the server carefully implements HPACK, r should be small. Otherwise, the value will be large and even around 1.

F. Evaluating HTTP/2 PING

HTTP/2 proposes the PING frame to measure the round-trip time (RTT). Either endpoint can send a PING frame to the other end, and the receiver sends back a PING frame with the same payload. However, little is known about the accuracy of the RTT measurement. We compare the values measured by HTTP/2 PING with the results from three other measurement approaches. One employs the ICMP Ping tool to collect the RTT samples. The other one exploits the packet exchange in TCP three-way handshaking to estimate the RTT (i.e, from sending a TCP SYN packet to receiving a TCP SYN/ACK packet [7], [8]. The third one uses the interval between sending an HTTP/1.1 request and receiving the corresponding HTTP/1.1 response to estimate RTT [9].

IV. Implementation

We implement the measurement methods described in Section III in a tool named H2Scope. It consists of two major components. One is to determine whether a web site supports HTTP/2 and establish a TLS connection with the web site if it supports HTTP/2 (Section IV-A). The other one is to send customized frames over the TLS connection and make decision after receiving the response frames (Section IV-B).

A. Establishing HTTP/2 Connection

Although HTTP/2 does not require encryption, the majority of browsers only support HTTP/2 over an encrypted connection [10], [11]. When using an unencrypted connection, the client can send an HTTP/1.1 request with *Upgrade* header field and the *h2c* value to the server. If the server supports HTTP/2 and accepts the upgrade, it will send back a 101 response (i.e., switching protocols), and then start sending HTTP/2 frames.

If TLS is used to establish an encrypted connection, we could use either ALPN (Application Layer Protocol Negotiation) [12] or NPN (Next Protocol Negotiation) [13] to determine whether or not the server supports HTTP/2, because HTTPs, SPDY, and HTTP/2 are all listening on port 443. NPN is a TLS extension for application layer protocol negotiation, and is used for SPDY. It is being replaced by ALPN for HTTP/2 because of security consideration. The main difference between NPN and ALPN is that using ALPN the client will send a list of supported application protocols to the server in the *ClientHello* message and waits for the server's selection in the *ServerHello* message whereas NPN asks the server to send a list of supported protocols for the client to select. H2Scope uses both NPN and ALPN to negotiate the application layer protocol with server.

B. Sending and Receiving HTTP/2 Frames

We use Nghttp/2 C library [14] to construct and receive HTTP/2 frames because it provides well-documented APIs. If the server supports HTTP/2, we will use nghttp2_session_callbacks_set_on_frame_send_callback() and nghttp2_ session_callbacks_set_on_frame_recv_callback() to set the callbacks so that we can know when a frame is sent and received in order to trigger other activities.

Then, we send customized SETTINGS frame and HEADERS frame to the server according to different purposes, and wait for the frames from the server. We leverage an event loop through poll() to get notified of receiving and sending data. When a request is replied by the server, we will store the request and the response into a database for further study. To speed up the scanning, we construct a thread pool with configurable number of threads, each of which will test a web site. After a thread finishes one site, it will check next one.

V. Measurement Results

We first characterize six popular HTTP/2 web servers, including Nginx (v1.9.15) [15], LiteSpeed (v5.0.11) [16], H2O (v1.6.2) [17], nghttpd (v1.12.0) [14] , Tengine (v2.1.2) [18] and Apache (v2.4.23) [19] in our testbed (Section V-A). We

have conducted the measurement on the top 1 million web sites twice to determine whether they support HTTP/2 (Section V-B) and characterize the changes. One measurement was conducted in Jul. 2016 (i.e., the first experiment) and the other one was performed in Jan. 2017 (i.e., the second experiment). For those HTTP/2 servers, we report their settings and features in Section V-C - Section V-H.

A. Characterizing Popular HTTP/2 Servers

We install the six popular web servers on a PC in our testbed, and use H2Scope to characterize their features. We verify the measurement result by inspecting the code of open-source servers (i.e., Nginx, Apache, H2O, nghttpd, and Tengine). This manual verification assures that the measurement result from H2Scope is correct, and then we apply H2Scope to scan the top 1 million websites. Table III lists the measurement results of different HTTP/2 servers.

Most servers support both ALPN and NPN for negotiating HTTP/2 except that Apache doesn't support NPN over TLS. RFC 7540 says "*implementations that support HTTP/2 over TLS MUST use protocol negotiation in TLS [TLS-ALPN]*". They also support request multiplexing. When testing their flow control mechanisms, we observe that the regulation on DATA frames is effective. In particular, when we set SETTINGS_INITIAL_WINDOW_SIZE to a small value S_{frame}, the response DATA frame's size is S_{frame}. However, LiteSpeed also applies the regulation on HEADERS frames and does not send back HEADERS frames if the SETTINGS_INITIAL_WINDOW_SIZE is set to 0. According to RFC 7540 the flow control should only affect DATA frames.

By sending unexpected WINDOW_UPDATE frame, we find that Nginx and Tengine will ignore the zero window update whereas Litespeed and H2O will send back RST_STREAM frame if the window is for stream as suggested by RFC 7540. Note that nghttpd and Apache send back GOAWAY frame, which is used to shutdown the connection or signal serious errors, even when the WINDOW_UPDATE frame is for stream. If we let the window increment exceed the largest value, all servers will send back error messages. More precisely, if the window increment is for the connection, they reply with GOAWAY. Otherwise, they respond with the RST_STREAM frame.

Unfortunately, some useful mechanisms have not been realized in popular web servers. For example, Nginx, Litespeed and Tengine don't support server push. H2O, nghttpd and Apache have provided such functionalities. For the priority mechanism, we examine the sequence of DATA frames returned from server, and notice that only H2O, nghttpd and Apache pass the testing of Algorithm 1. Note that RFC 7540 doesn't clearly define what kind of resource will be allocated for streams with high priority.

When sending streams that depend on itself, we find that LiteSpeed will ignore it whereas H2O, nghttpd and Apache regard it as a connection error and send back GOAWAR. Nginx and Tengine follow RFC 7540 to send back RST_STREAM. Although all the servers support header compression, they do not have the same efficiency. We observe that Nginx and Tengine do not add the fields in the responses headers to the dynamic table. A good news is that all these servers have supported HTTP/2 PING. By leveraging this feature, web applications can measure network performance and then provide adaptive services.

Since Nginx and Tengine support adjusting the maximum number of concurrent HTTP/2 streams, we set the value of SETTINGS_MAX_CONCURRENT_STREAMS to 0 or 1. Both servers have the same result. When this value is set to 0, we receive the RST_STREAM frame from the servers after sending a new request to it. If this value is set to 1 and we send two requests to the servers simultaneously, the second request will trigger an RST_STREAM frame from the servers.

B. Adoption

1) ALPN and NPN: In the first experiment, we find that 49,334 sites support establishing HTTP/2 connections through NPN whereas 47,966 sites use ALPN for initializing HTTP/2 connections. This number increases to 78,714 and 70,859, respectively, in the second experiment. Moreover, we receive HEADERS frames from 44,390 sites in the first experiment, and the number increases to 64,299 in the second experiment. This observation indicates that more and more web sites adopt HTTP/2. The other tests were conducted on the sites that return HEADERS frames. We also find that more than one hundred types of servers just speak NPN. It may be because ALPN is not available until OPENSSL 1.0.2.

2) Server: We parse the HEADERS frame returned from server to obtain the server name. According to this information, we observe 223 and 345 different kinds of servers in the first and the second experiments, respectively. Note that since the information of server type can be set to an arbitrary value by the web server, the obtained result may not be very accurate. But it illustrates a big picture of server adoption.

Table IV lists seven servers that have been adopted by more than 1,000 sites in each experiment. We can see that Litespeed, Nginx and GSE are the most widely used web servers in both two experiments, and more sites are using Nginx for HTTP/2 according to the result of the second experiments. GSE is Google's proprietary web server. Tengine and cloudflare-nginx are variants of Nginx. A new server named Tengine/Aserver appears in the second experiment. It is due to the fact that sites in the domain of tmall.com change their original server name from Tengine to Tengine/Aserver.

C. SETTINGS

The SETTINGS frame contains several important parameters, including SETTINGS_HEADER_TABLE_SIZE with default value 4,096 suggesting a peer the maximum size of the header compression table, SETTINGS_MAX_CON CURRENT_STREAMS indicating the maximum number of concurrent streams acceptable to the sender, SETTINGS_INITIAL_WINDOW_SIZE with default value 65,535 denoting the sender's initial window size for stream-level flow control, SETTINGS_MAX_FRAME_SIZE with default

TABLE III
CHARACTERIZING POPULAR HTTP/2 WEB SERVERS IN TESTBED. COLUMN RFC 7540 LISTS THE CORRESPONDING SPECIFICATION. SUPPORT* REPRESENTS PARTIALLY SUPPORT.

	Nginx	LiteSpeed	H2O	nghttpd	Tengine	Apache	RFC 7540
ALPN	support	support	support	support	support	support	support
NPN	support	support	support	support	support	no support	does not require
Request Multiplexing	support	support	support	support	support	support	support
Flow Control on DATA Frames	yes	yes	yes	yes	yes	yes	yes
Flow Control on HEADERS Frames	no	yes	no	no	no	no	no
Zero Window Update on stream	ignore	RST_STREAM	RST_STREAM	GOAWAY	ignore	GOAWAY	RST_STREAM
Zero Window Update on connection	ignore	GOAWAY	GOAWAY	GOAWAY	ignore	GOAWAY	GOAWAY
Large Window Update (Connection)	GOAWAY	GOAWAY	GOAWAY	GOAWAY	GOAWAY	GOAWAY	GOAWAY
Large Window Update (Stream)	RST_STREAM	RST_STREAM	RST_STREAM	RST_STREAM	RST_STREAM	RST_STREAM	RST_STREAM
Server Push	no	no	yes	yes	no	yes	yes
Priority Mechanism Testing (Algorithm 1)	fail	fail	pass	pass	fail	pass	pass
Self-dependent Stream	RST_STREAM	ignore	GOAWAY	GOAWAY	RST_STREAM	GOAWAY	RST_STREAM
Header Compression	support*	support	support	support	support*	support	support
HTTP/2 PING	support	support	support	support	support	support	support

TABLE IV
SERVERS THAT HAVE BEEN USED BY MORE THAN 1,000 SITES IN THE FIRST (I.E., 1ST EXP.) AND THE SECOND (I.E., 2ND EXP.) MEASUREMENTS.

Server name	Num. in 1st Exp.	Num. in 2nd Exp.
Litespeed	12,637	13,626
Nginx	11,293	27,394
GSE	9,928	9,929
Tengine	2,535	674
cloudflare-nginx	1,197	1,766
IdeaWebServer/v0.80	1,128	1,261
Tengine/Aserver	0	2,620

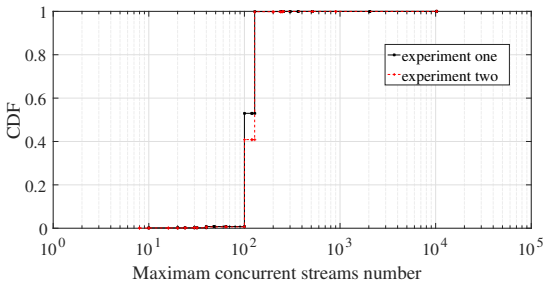

Fig. 2. The distribution of SETTINGS_MAX_CONCURRENT_STREAMS.

value 16,384 indicating the maximum payload size acceptable to the sender, and SETTINGS_MAX_HEADER_LIST_SIZE notifying the remote endpoint the maximum size of header list acceptable to the sender.

The measurement results show that all servers use the default value of SETTINGS_HEADER_TABLE_SIZE. This is reasonable since large table size may consume more system resource if an attacker keeps sending different headers to exhaust the available dynamic table size. For SETTINGS_MAX_ CONCURRENT_STREAMS, RFC 7540 suggests that it should not be smaller than 100. As shown in Figure 2, 100 and 128 are popular values selected by the web sites in two experiments, and the majority of web sites use a value larger than or equal to 100.

Table V lists the values of initial window size. It is interesting to see that some servers (e.g., Nginx) set the initial window size to 0 and then immediately send WINDOW_UPDATE frames to increase the window size. We further verify Nginx in the testbed, and have the same observation. Table VI

TABLE V
THE DISTRIBUTION OF THE VALUES OF SETTINGS_INITIAL_WINDOW_SIZE OBTAINED IN THE FIRST (I.E., 1ST EXP.) AND THE SECOND (I.E., 2ND EXP.) EXPERIMENTS. NULL MEANS THAT THE SETTINGS FRAME DOES NOT CONTAIN THIS ITEM.

SETTINGS_INITIAL_WINDOW_SIZE	1st Exp.	2nd Exp.
NULL	1,050	1,015
0	3,072	7,499
32,768	3	59
65,535	49	106
65,536	20,477	40,612
131,072	1	1
262,144	1	1
1,048,576	10,799	10,929
16,777,216	11	15
20,000,000	1	0
2,147,483,647	8,926	4,062

TABLE VI
THE DISTRIBUTION OF THE VALUES OF SETTINGS_MAX_FRAME_SIZE OBTAINED IN THE FIRST (I.E., 1ST EXP.) AND THE SECOND (I.E., 2ND EXP.) EXPERIMENTS.

Maximum Frame Size	1st Exp.	2nd Exp.
NULL	1,050	1,015
16,384	24,781	25,987
1,048,576	27	81
16,777,215	18,532	37,216

TABLE VII
THE DISTRIBUTION OF THE VALUES OF SETTINGS_MAX_HEADER_LIST_SIZE OBTAINED IN THE FIRST (I.E., 1ST EXP.) AND THE SECOND (I.E., 2ND EXP.) EXPERIMENTS.

Maximum Header List Size	1st Exp.	2nd Exp.
NULL	1,050	1,015
unlimited	32,568	52,311
16,384	10,717	10,806
32,768	3	59
81,920	2	3
131,072	24	25
1,048,896	26	80

lists the values of maximum frame size. More than half of the servers adopt the default value (i.e., 16,384) in the first experiment while more sites adopt a larger value (i.e., 1,048,576 or 16,777,215) in the second experiment. Moreover, Table VII lists the distribution of the values of SETTINGS_MAX_HEADER_LIST_SIZE. It shows that 73.4%

sites in the first experiment and 81.3% sites in the second experiment use the suggested value (i.e., unlimited).

D. Flow Control

1) Controlling DATA Frames: We let the value of SETTINGS_INITIAL_WINDOW_SIZE be 1 to test whether it can control DATA frames. The result shows that 37,525 sites in the first experiment and 44,204 sites in the second experiment sent back DATA frames with 1-byte payload, meaning that these servers follow RFC 7540. 2,433 sites in the first experiment and 8,056 sites in second experiment return DATA frames of *zero* length. Moreover, 4,432 sites did not send back any response. This number increases to 12,039 in the second experiment and 10,472 sites among them are using LiteSpeed. It is worth noting an adversary could launch DoS attacks like malicious TCP receiver [20] by setting SETTINGS_INITIAL_WINDOW_SIZE to a small value so that the server cannot quickly send out the response frames and release the corresponding memory.

2) The Effect of Zero Initial Window on HEADERS Frames: After we set the value of SETTINGS_INITIAL_WINDOW_SIZE to zero, the server should return HEADERS frames on the receipt of new requests without sending back the DATA frames. However, we only received HEADERS frames from 17,191 sites in the first experiment and 23,834 sites in the second experiment. The remaining sites do not follow RFC 7540 and applied flow control to the HEADERS frames mistakenly.

3) Zero Window Update: We get diverse responses after sending a WINDOW_UPDATE frame with value 0. 23,673 sites in the first experiment and 26,156 sites in the second experiment sent back RST_STREAM frames whereas 20,717 sites and 38,143 sites in the two experiments did not regard zero window update as a stream error. We also find that 31 sites and 162 sites in the two experiments consider it as a connection error and send back GOAWAY frames. Moreover, 26 sites in the first experiment and 42 sites in the second experiment sent us clear error information (i.e., the window update shouldn't be zero in the additional debug data field). We also conduct the measurement on the connection level, and find that nearly all the websites return connection error.

4) Large Window Update: When sending two WINDOW_UPDATE frames for the entire connection and let the summarization of their window size increment be larger than 2^{31}-1, we received GOAWAY frames from 40,567 sites and 62,668 sites in the two experiments, respectively. When sending the same large window update for the streams, we got 36,619 and 44,057 RST_STREAM frames in the first and the second experiments, individually. 7,771 sites and 20,242 sites in the two experiments did not send back the RST_STREAM frames, individually.

E. Priority Mechanism

1) Stream Priority: Since the priority mechanism provides a server suggestions on resource allocation for different streams, RFC 7540 neither defines how to realize it nor provides criteria for assessing the mechanism. Therefore, we use the sequence order of the first and the last DATA frames in each stream to infer the priority. More precisely, if a stream has higher priority, the server should first send out its DATA frames or finish sending all its DATA frames.

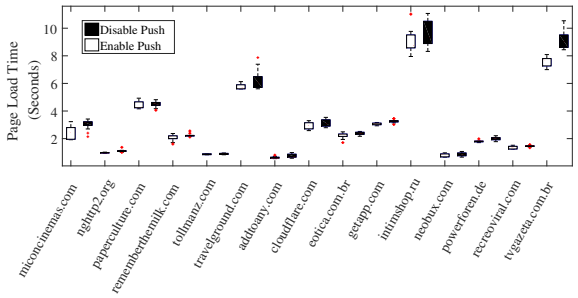

Fig. 3. The Page Load Time when server push is enabled and disabled

For the ease of explanation, we construct and send requests following the example in the section 5.3.3 of RFC 7540. More precisely, after receiving PRIORITY frames, the server will let stream D be the parent stream of stream A, which is the parent stream of streams B, C and F. Moreover, stream E is the dependent stream of stream C. Therefore, we expect to receiving DATA frames from stream D before the DATA frames from other streams. Similarly, stream A's DATA frames should arrive at the client before the DATA frames from all other streams except stream D. Moreover, the client should receive the DATA frames from stream C before the DATA frames from stream E.

If we use the sequence order of the last DATA frames in each stream to infer the priority, we find that 1,147 sites and 2,187 sites in the two measurements follow the above rules. If we employ the order of the first DATA frames in each stream to infer the priority, 46 sites and 117 sites in the two experiments obey the rules, respectively. If we take into account both the first and the last DATA frames to determine the priority, 38 sites and 111 sites act as what we expect in the two experiments. These results suggest that the priority mechanism has not been well designed and deployed.

2) Depend on self: After sending a PRIORITY frame to let a stream depend on itself, we received RST_STREAM frames, as suggested by RFC 7540, from 18,237 sites and 53,379 sites in the two experiments, respectively. Other sites either sent back GOAWAY or ignore the frames. It may suggest that the servers are getting better implementation.

F. Server Push

When visiting the front page of those web sites, we only received PUSH_PROMISE frames from six sites in the first experiment, and got PUSH_PROMISE frames from additional nine sites in the second experiment. They usually push objects like javascript, css, figures, etc. When requesting URLs other than the front page, we do not receive pushed objects. We further measure and compare the performance when server push is enabled and disabled. For a fair comparison, we visit these sites for 30 times using Firefox, because it allows us to

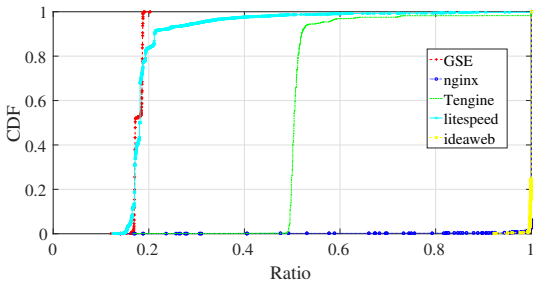

Fig. 4. The HPACK compression ratio of five popular HTTP/2 servers in the first experiment (Jul. 2016).

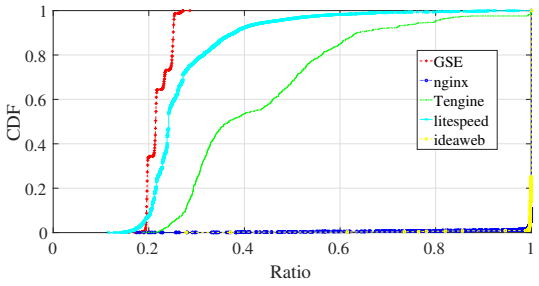

Fig. 5. The HPACK compression ratio of five popular HTTP/2 servers in the second experiment (Jan. 2017).

enable/disable server push through configuration, and collect the page load time for comparison following [21]. Fig. 3 shows the distribution of page load time when the server push is enabled/disabled. It shows that enabling server push could reduce the page load time in most cases.

G. HPACK

After sending identical requests H times in different streams, we get the response and compute the HPACK compression ratio (i.e., r) defined in Section III-E. We filter out the data with $r > 1$, because a few web sites will insert new cookies into the 2nd to the Hth HEADERS frames, making $S^1_{header} < S^i_{header}$ $(i = 2, \ldots, H)$. Eventually, we collect data from 37,849 sites in the first experiment, including 2,449 Tengine servers, 12,764 Nginx servers, 9,929 GSE servers, 873 IdeaWebServer and 11,834 litespeed servers while we get 46,948 sites in second experiment, including 619 Tengine servers, 22,548 Nginx servers, 9,925 GSE servers, 1,000 IdeaWebServer and 12,856 litespeed servers.

Figure 4 and 5 illustrate the HPACK compression ratio of the top five popular HTTP/2 servers in the two experiments, namely GSE, nginx, Tengine, litespeed, and ideaweb. We can see that GSE achieves the best compression ratio, all of which are less than 0.3. Nginx and IdeaWebServer have the worst performance. In particular, the compression ratio of 93.5% Nginx servers is 1, meaning that all response headers have the same size. By investigating the responses, we find that Nginx only puts the fields in the request headers into the dynamic table without storing the fields in the response headers into the dynamic table. For LiteSpeed, 80% servers have HPACK compression ratios less than 0.3, indicating effective compression. For Tengine, in the first experiment, since sites in the domain of tmall.com may have similar resources, their HPACK compression ratio are almost the same. In the second experiment, since some sites changed their original server names from Tengine to Tengine/Aserver, sites using Tengine have diverse HPACK compression ratio.

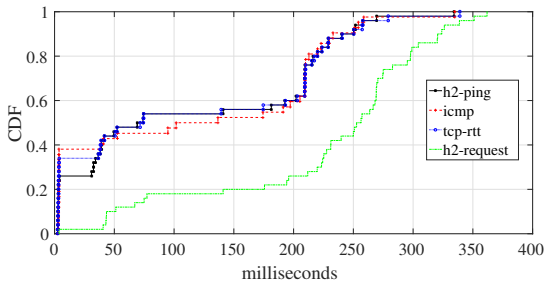

Fig. 6. RTT measured by ICMP, TCP, HTTP/1.1, and HTTP/2.

H. HTTP/2 PING

To evaluate the accuracy of RTT measurement by HTTP/2 PING, we compare the results with that from three other methods, namely, ICMP ping, TCP based approach that exploits its three-way handshaking, and HTTP/1.1 based method that leverages the interval between HTTP request and response.

The measurement client runs in a machine with the OS Ubuntu 14.04 in our campus, and we randomly select 10 sites for each top popular servers. Figure 6 shows the CDF of RTT values for the web sites. We can see that the RTT measured by HTTP/2 PING frame and that by TCP three-way handshaking are very close. The result obtained from ICMP is also similar to that from HTTP/2 PING. However, the RTT measured by HTTP/1.1 is longer than that from the other three methods. The reason may be that the HTTP/1.1 server needs time to handle the requests, generate the response and send it back. By contrast, HTTP/2 PING can obtain quite accurate results.

VI. Discussion

Based on the investigation of popular HTTP/2 implementations and the measurement of deployed HTTP/2 servers, we have some concerns and suggestions about the development and deployment of HTTP/2.

First, request multiplexing could address the head-of-line blocking issue [22] because the server can process several requests simultaneously and send back the response frames. However, since HTTP/2 uses one TCP connection, its performance may be significantly affected in a lossy environment (e.g., mobile network), because the congestion window of the TCP connection will be decreased in the presence of packet losses and thus limit the throughput. Using more than one TCP connection could mitigate such problem.

Second, while flow control can prevent the sender from overwhelming the receiver, it could also be exploited by adversaries to launch DoS attacks. For example, the TCP's flow control mechanism has been exploited by malicious receiver to

conduct various attacks [23]. One possible defense approach is to define lower bounds for the values of SETTINGS_INITIAL _WINDOW_SIZE and WINDOW_UPDATE. Moreover, although RFC 7540 suggests that flow control should only be applied to DATA frames, many servers also control the HEADERS frames and may mislead HTTP/2 clients that are expecting the HEADERS frames.

Third, although the priority mechanism is very useful, RFC 7540 does not suggest any algorithm for HTTP servers and clients. Since the dependencies of web objects are complex [24], [25], algorithms for servers and clients should be carefully designed and evaluated. On one hand, server side algorithms should take into account both performance and security, because malicious clients may exploit this mechanism to launch algorithmic complexity attacks [26] (e.g., force the server to frequently reconstruct the dependency tree, etc.). On the other hand, HTTP/2 clients may need more information to better select the priorities and weights for streams. For example, without knowing how much time or resource will be needed by the server to process requests, the parameters selected by the client may be ignored by the server.

Fourth, while server push could speed up the downloading [27], [28], only a few web sites support it. Moreover, existing HTTP/2 servers only allow users to statically list which resources will be pushed. To further improve the performance, new algorithms and the support from HTTP/2 servers are desired to dynamically determine which resources should be pushed (e.g., the most likely next page, etc.). A potential issue with server push is the waste of network bandwidth. For example, since a server can push web objects after sending the PUSH_PROMISE frame, if the client already caches these web objects, the pushed data wastes the network bandwidth. Moreover, if the client shutdowns the connection before receiving the push data, such transmissions are useless.

Fifth, it is expected that header compression will reduce the data exchanged between a server and a client. However, Figure 4 and 5 show that it is not easy to correctly realize this feature for improving the performance. Moreover, attackers might exploit this feature to launch DoS attacks, such as setting SETTINGS_HEADER_TABLE_SIZE, which affects the size of dynamic table, to a large value, and then using randomly-generated headers to fill up the table, etc.

VII. Related Work

Since HTTP/2 is a new protocol standardized in 2015, there are not many studies on it yet. Varvello et al. reported their regular measurement on top 1 million web sites to check whether they support HTTP/2 or not [3]. However, they did not examine whether or not those web sites support the new features in HTTP/2.

Chowdhury et al. [29] and Saxce et al. [30] conducted measurement to determine whether HTTP/2 will be more energy efficient and faster than HTTP/1.1, respectively. Chowdhury et al. [29] found that when RTT is above 30ms and TLS is used, HTTP/2 saves energy. Saxce et al. [30] observed that packet loss will significantly affect HTTP/2. Sanae et al. investigated HTTP/2 Server Push in mobile networks [21], and found that server push can improve performance, especially when latency is high, because it saves one round-trip. Our observation is in consistent with them. Kim et al. examined the impact of HTTP/2 on the performance of three types of typical web sites in Korea in different network environments [31]. The result suggested that HTTP/2 doesn't improve the performance because portal sites are already well-optimized. Another possible reason is that sites with multiple domains will increase the time spent on the establishment of HTTP/2 connection. Enrico et al. compared the user experience between HTTP/2 and HTTP/1.1 and found that HTTP/2 is not better than HTTP/1.1 [32].

Adi et al. examined how to exploit some new frames in HTTP/2 to conduct DDoS attacks [33]. Mi et al. proposed SMig to allow a client or server to migrate an HTTP/2 stream from one connection to another [34]. Han et al. proposed Meta-Push that is cellular-friendly and has better performance [35]. They adopt server push to get the meta files containing the resource URLs and use prefetch in the link header standardized in HTML 5 to download all the resources in one round-trip. Researchers have exploited the new features, especially the server push mechanism, to improve the performance of online streaming [36]–[41]. Since HTTP/2 roots from SPDY, the performance of SPDY has also been studied [42], [43].

VIII. Conclusion

We conduct the first systematic investigation on whether the popular HTTP/2 servers have correctly realized the new features in HTTP/2 and how the deployed servers use these features. We not only inspect six popular implementations of HTTP/2 server but also propose new measurement methods to characterize HTTP/2 web sites. We realize the new methods in a tool named H2Scope and conduct a large scale measurement on the top 1 million Alexa web sites. The results reveal new observations and insights. In short, existing HTTP/2 web servers have not taken full advantage of the new features in HTTP/2, and more research is desired to leverage these features for better performance. To foster the research, we will release H2Scope in this link https://github.com/valour01/H2Scope. In future work, we will perform regular scanning on popular web sites to characterize how HTTP/2 and its features are adopted, and examine the interaction between HTTP/2 and other protocols/techniques (e.g., TCP, HTML5) and the usage [44].

Acknowledgement

This work is supported in part by the Hong Kong ITF (No. UIM/285), Hong Kong GRF (No. PolyU 5389/13E), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), National Natural Science Foundation of China (61602371, 61572365), China Postdoctoral Science Foundation (No. 2015M582663), and Science and Technology Commission of Shanghai Municipality (No. 15ZR1443000).

REFERENCES

[1] IETF, "Hypertext transfer protocol version 2 (HTTP/2)," https://tools.ietf.org/html/rfc7540, May 2015.

[2] Google Inc., "SPDY," https://www.chromium.org/spdy.

[3] M. Varvello, K. Schomp, D. Naylor, J. Blackburn, A. Finamore, and K. Papagiannaki, "Is the web http/2 yet?" in *Proc. International Conference on Passive and Active Measurement (PAM)*, 2016.

[4] ——, "Http/2 dashboard: Monitoring the adoption and performance of http/2 on the web," http://isthewebhttp2yet.com/.

[5] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, 2013.

[6] IETF, "Hpack: Header compression for HTTP/2," https://tools.ietf.org/html/rfc7541, May. 2015.

[7] H. Jiang and C. Dovrolis, "Passive estimation of tcp round-trip times," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, 2002.

[8] X. Luo, E. Chan, and R. Chang, "Design and implementation of tcp data probes for reliable and metric-rich network path monitoring," in *Proc. USENIX ATC*, 2009.

[9] L. Xue, X. Ma, X. Luo, L. Yu, S. Wang, and T. Chen, "Is what you measure what you expect? factors affecting smartphone-based mobile network measurement," in *Proc. IEEE INFOCOM*, 2017.

[10] C. Blog, "Transitioning from SPDY to HTTP/2," http://goo.gl/oiZ3BA, Feb. 2016.

[11] mozilla wiki, "Networking/http2," https://wiki.mozilla.org/Networking/http2, Aug. 2014.

[12] S. Friedl, A. Popov, A. Langley, and E. Stephan, "Transport layer security (tls) application-layer protocol negotiation extension," https://tools.ietf.org/html/rfc7301, July 2014.

[13] A. Langley, "Transport layer security (tls) next protocol negotiation extension," https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-04, May 2012.

[14] T. Tsujikawa, "Nghttp2: Http/2 c library," May. 2016.

[15] "nginx," http://nginx.org/en/download.html, 2016.

[16] "Litespeed," https://www.litespeedtech.com/http2-ready, 2016.

[17] "H2o," https://h2o.examp1e.net/, 2016.

[18] "Tengine," http://tengine.taobao.org, 2016.

[19] "Apache: Http server project," https://httpd.apache.org/, 2016.

[20] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "Tcp congestion control with a misbehaving receiver," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, 1999.

[21] S. Rosen, B. Han, S. Hao, Z. M. Mao, and F. Qian, "Push or request: An investigation of http/2 server push for improving mobile performance," in *Proc. WWW*, 2017.

[22] F. Qian, V. Gopalakrishnan, E. Halepovic, S. Sen, and O. Spatscheck, "Tm3: Flexible transport-layer multi-pipe multiplexing middlebox without head-of-line blocking," in *Proc. AMC CoNEXT*, 2015.

[23] R. Sherwood, B. Bhattacharjee, and R. Braud, "Misbehaving tcp receivers can cause internet-wide congestion collapse," in *Proc. ACM CCS*, 2005.

[24] X. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with shandian," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design & Implementation*, 2016.

[25] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster page loads using fine-grained dependency tracking," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design & Implementation*, 2016.

[26] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium*, 2003.

[27] V. Krasnov, "Announcing support for http/2 server push," https://blog.cloudflare.com/announcing-support-for-http-2-server-push-2/, April 2016.

[28] A. Jayaprakash, "Are you ready for http/2 server push?!" https://blogs.akamai.com/2016/04/are-you-ready-for-http2-server-push.html, April 2016.

[29] S. Chowdhury, V. Sapra, and A. Hindle, "Client-side energy efficiency of http/2 for web and mobile app developers," in *Proc. IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[30] H. Saxce, I. Oprescu, and Y. Chen, "Is http/2 really faster than http/1.1?" in *Proc. 18th Global Internet Symposium*, 2015.

[31] H. Kim, J. Lee, I. Park, H. Kim, D. Yi, and T. Hur, "The upcoming new standard http/2 and its impact on multi-domain websites," in *Proc. Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2015.

[32] E. Bocchi, L. De Cicco, M. Mellia, and D. Rossi, "The web, the users, and the mos: Influence of http/2 on user experience," in *International Conference on Passive and Active Network Measurement*. Springer, 2017, pp. 47–59.

[33] E. Adi, Z. A. Baig, P. Hingston, and C.-P. Lam, "Distributed denial-of-service attacks against http/2 services," *Cluster Computing*, vol. 19, no. 1, 2016.

[34] X. Mi, F. Qian, and X. Wang, "Smig: Stream migration extension for http/2," in *Proc. ACM CoNEXT*, 2016.

[35] B. Han, S. Hao, and F. Qian, "Metapush: Cellular-friendly server push for http/2," in *Proc. 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, 2015.

[36] R. Huysegems, J. van der Hooft, T. Bostoen, P. Rondao Alface, S. Petrangeli, T. Wauters, and F. De Turck, "Http/2-based methods to improve the live experience of adaptive streaming," in *Proc. ACM MM*, 2015.

[37] M. Xiao, V. Swaminathan, S. Wei, and S. Chen, "Evaluating and improving push based video streaming with http/2," in *Proc. ACM NOSSDAV*, 2016.

[38] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan, "Modeling http/2 speed from http/1 traces," in *Proc. International Conference on Passive and Active Measurement (PAM)*, 2016.

[39] M. Xiao, V. Swaminathan, S. Wei, and S. Chen, "Dash2m: Exploring http/2 for internet streaming to mobile devices," in *Proc. ACM MM*, 2016.

[40] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori, "Dash fast start using http/2," in *Proc. ACM NOSSDAV*, 2015.

[41] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, T. Bostoen, and F. De Turck, "An http/2 push-based approach for low-latency live streaming with super-short segments," *Journal of Network and Systems Management*, pp. 1–28.

[42] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan, "Towards a spdy'ier mobile web?" *IEEE/ACM Transactions on Networking*, vol. 23, no. 6, pp. 2010–2023, 2015.

[43] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is spdy?" in *Proc. 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[44] X. Luo, L. Xue, C. Shi, Y. Shao, C. Qian, and E. Chan, "On measuring one-way path metrics from a web server," in *Proc. IEEE ICNP*, 2014.