How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push

Torsten Zimmermann, Jan Rüth, Benedikt Wolters, Oliver Hohlfeld
Chair of Communication and Distributed Systems, RWTH Aachen University
{zimmermann, rueth, wolters, hohlfeld}@comsys.rwth-aachen.de

***Abstract*—The simplicity of HTTP made it the default building block for desktop and mobile apps, yet it suffers from inherent inefficiencies in the modern web. HTTP/2 was designed to address these inefficiencies and its adoption remarks a major protocol shift in the Internet. Despite this relevance, its Internet-wide adoption remains unknown. Especially, the adoption and use of *server push*—advertised as a key feature to further reduce page load times—is completely unexplored. To answer both questions, we provide large-scale measurements of the HTTP/2 adoption and usage of server push in the wild, probing the entire IPv4 address space and the complete set of .com/.net/.org domains. We find 5.38M HTTP/2 enabled domains hosted by only few infrastructures driving this adoption. While we find the overall HTTP/2 adoption to increase, only few hundred domains utilize server push. We examine pushed content, push strategies and identify the use of currently undocumented push strategies. Moreover, we discover large sources of overheads through server push for reoccurring page visits. By measuring page load times, we show that while push *can* speed up webpages, it also *can* slow them down—motivating the need for optimized push strategies.**

I. Introduction

Currently, the Hypertext Transfer Protocol (HTTP) is the de-facto standard protocol for realizing a large set of desktop and mobile applications. Traffic shares of HTTP $> 50\,\%$, e.g., in a residential access link [1], an Internet Exchange Point [2], or a backbone link [3], express this dominance. Despite this relevance, HTTP-based applications are built on a protocol that was designed 18 years ago and suffers from a set of inefficiencies in the modern web, e.g., head of line blocking.

To largely address these inefficiencies, the IETF standardized HTTP/2 (H2 for the remainder) in 2015 as the H1 successor. Its deployment denotes a major protocol shift with the potential to accelerate the web. Unlike IPv6, H2 has the potential to be adopted quickly since no support by the network core is required and client-side support is already provided by all major browsers. However, despite this potential, its adoption by the Alexa listed popular domains is known to be comparably low [4]. Since the overall H2 adoption almost entirely depends on server-side support (e.g., by large content distribution infrastructures that have not yet widely enabled H2), the H2 adoption rate can substantially change at *any time*. This motivates further studies observing and analyzing this major protocol shift to provide recommendations for an optimal use of the new protocol.

A broadly anticipated component is the H2 server-side push feature, transforming the *pull-only* web into a *push-enabled* web. By enabling servers to send resources without an explicit client request (push), server push promises the potential to speed up the H2 web by saving unnecessary round trips. However, an incorrect usage of push can have detrimental effects, e.g., *i)* prolonging page load times by pushing objects in orders that stall the browsers' rendering pipeline or *ii)* wasting resources when pushing cached or unnecessary objects. The correct usage of push thus depends on appropriate *push strategies* that determine which objects are being pushed and in which order. While H2 implementations already support push, no such strategies are defined in the standard [5], leaving it to developers and providers how to implement and use this feature. Furthermore, as this feature is typically configured explicitly, an empirical understanding of current push (ab)use is missing, which in turn is necessary to provide guidelines on *how* to correctly utilize this feature.

To answer the above questions, we conduct a large-scale measurement study, i.e., we probe the entire IPv4 address space, the Alexa 1M list, as well as the complete set of .com/.net/.org domains. This represents the first comprehensive study of the overall H2 adoption and enables us to tackle the main question of this paper: obtaining an *empirical* understanding of current (ab)use of server-push in the wild. In the absence of standardized push strategies, we hereby aim to inform the current standardization activities with an understanding on push use and aim at paving the way for optimized push strategies. The contributions of this paper are as follows:

1) We present the *first large-scale* assessment of H2 by *i)* scanning the entire IPv4 address space and *ii)* the complete 151.4M .com/.net/.org domains. Thus, we are able to paint a comprehensive picture of the H2 adoption. This complements earlier work focusing on a longitudinal adoption study on a smaller set of the Alexa top list [4] and provides the insights that only few infrastructures currently drive the H2 adoption. We found that, e.g., Akamai can drastically increase their customer's H2 adoption by at least a factor of 210 in our datasets. We find that H2 is enabled for 5.38M of the probed domains.
2) We provide the *first* analysis of H2 server push usage in the wild. While millions already use H2, server push is only used by 595 domains. In the absence of standardization and recommendations for concrete push strategies (e.g., which resources to push and when), we identified current undocumented push practices each having a potential

ISBN 978-3-901882-94-4 © 2017 IFIP

influence on page load time and network resource usage. Interestingly, we found that simply pushing *some* content does not necessarily improve the page load time, indeed only 50.92 % webpages show performance benefits and using server push can even drastically degrade page load time by up to 67 %.

3) We make our dataset publicly available for the research community and provide additional information at: `https://push.comsys.rwth-aachen.de`

Paper Structure. We first present background information on the history of H2, its characteristics, and features in Section II. We then summarize related work regarding the H2 adoption in the Internet and server push approaches in Section III. Section IV then presents our measurement of the Internet-wide usage of H2 required to identify and further evaluate the usage of server push in Section V. Finally, we conclude this paper and discuss further research directions in Section VI.

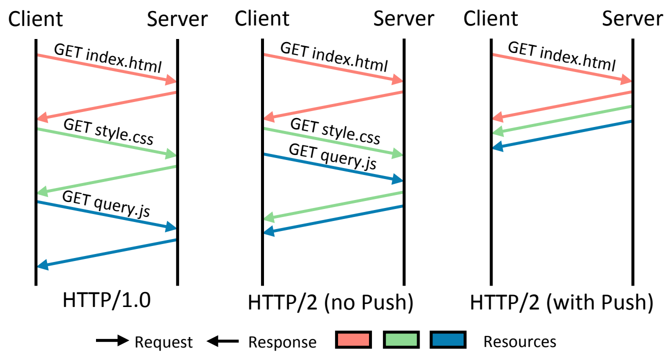


Fig. 1: H2 Server Push compared to H1 and H2 without push. In H1, requests are served sequentially. As H2 offers multiplexed streams, multiple requests can be issued in parallel. When server push is active, it enables servers to preemptively *push* embedded objects without explicit client requests. This procedure *can* optimize page load times by saving round trips.

II. Background

H2. The initial starting point and some of the key features for the first draft of H2 [5] are rooted in Google's attempt to replace H1 by the SPDY protocol [6]. Meanwhile, SPDY has converged to H2, which has been finalized and standardized in May 2015. Since SPDY will therefore be no longer supported, we entirely focus on H2 in the following.

A major difference compared to H1 is that H2 is a binary instead of an ASCII protocol, which enables easier framing and more efficient processing [7]. Moreover, H2 is able to *multiplex* requests and responses, which results in multiple, parallel *streams*—identified by their stream IDs—over a single TCP connection. This interleaving of requests and responses mitigates the head of line blocking on the application layer, when compared to H1.

Furthermore, clients can signal request importance to the server by assigning *priorities* to streams that they initiated. Exemplary, a browser could signal to prioritize requests for images that are rendered next when scrolling, as compared to content out of the currently rendered area. In addition, a client is able to update these priorities dynamically. Note that these priorities express preferences rather than demands and that it is up to the server how to handle them. H2 preserves the H1 paradigm of a stateless protocol, meaning header information for the same connection can be repetitive. To reduce the overhead caused by this redundancy, H2 enables *header compression* [8].

H2 over TLS. Although not explicitly required by the standard, all major server and client implementations use H2 over TLS. To distinct between different protocols on top of TLS, Application Layer Protocol Negotiation (ALPN) [9] is used. We remark that, while we find its predecessor Next Protocol Negotiation (NPN) [10] to be still in use, it will, however, not longer be supported by browsers such as Chrome[1]. Using ALPN, a client is able to inform the server about supported protocols, ordered by priority. Upon negotiation, the server selects the protocol and signals the decision to the client.

H2 Server Push. Finally, aiming to optimize the user experience by reducing the page load time, H2 adds a feature called *server push*. We point out that this feature, e.g., compared to the stream feature, has to be configured and applied by the server explicitly. To understand the benefits added by push, recall the procedure used by H1: the browser first requests the body document, parses it, and then individually requests all discovered embedded objects. In contrast, an H2 push-enabled server can preemptively *push* embedded objects without an explicit client request, illustrated in Figure 1 [5].

Thus, push can optimize the page load time by omitting these requests and thus saving the required round trip. This way, push enables to transfer resources before the browser completed parsing and processing the embedding document. In order to push resources to a client, the server announces a push using a `PUSH_PROMISE` frame over a client initiated stream. Besides the actual header information for the data to be pushed, it contains a stream ID on which the server will send the actual data. Following the `PUSH_PROMISE` frame, the server starts sending the data over the previously announced server initiated stream. As a requirement by most major browsers [7], pushed resources have to follow the *same-origin* policy, i.e., the server can not push content from other domains. If the client decides that the push should not be sent after the `PUSH_PROMISE`, it can reset the promised stream using a `RST_STREAM` frame. Moreover, the client itself can disable the server push feature completely for the current connection by setting `SETTINGS_ENABLE_PUSH` in a specific settings header to `0`.

The potential benefits on page load times thus render server push a key feature of H2. Additionally, the fact that the standard does not define any strategy on how to push content motivates us to study its configuration and usage in the wild.

[1]blog.chromium.org/2016/02/transitioning-from-spdy-to-http2.html

III. Related Work

H2 Adoption and Performance. The first empirical understanding of H2 and SPDY adoption for *popular* domains is provided by the scans of the Alexa Top 1M list by Varvello et al. [4][2]. Using a cumulatively evolving Alexa list—totaling 1.8M domains over the course of one year—these results indicate a growth of H2 availability from 1.4 % in November 2014 to 3.6 % in October 2015 (see [4]). Their adoption analysis is followed by an in-depth analysis of the websites' structure and the influence on loading times. We complement this ongoing effort by *i)* broadly analyzing the H2 adopting in large-scale measurements covering the entire IPv4 address space and the complete set of .com/.net/.org domains and *ii)* by investigating the server structure driving this adoption. Further, by including their Alexa domain list, we independently confirm their measurements and verify our scan setup.

In addition, related work focused on the performance of H2 or its predecessor SPDY compared to H1 in various network setups or real world traces [4], [11], [12], [13], [14]. As the outcome of these works are partly contradictory, i.e., showing faster and slower page load times, we are keen to investigate if these performance gains or deficits can be attributed to correct or false usage of server push in the Internet.

H2 Server Push. While the H2 standardization finished and a first empirical understanding of its adoption by *popular* domains is available, ongoing efforts to define push strategies have not yet converged and no insights into push usage in the wild are available. We again point out that the current standard itself does not define *any* strategies at all.

Concretely, related work focused on how to optimize push for specific setups and resource characteristics. To not waste bandwidth in mobile networks, the approaches presented in [15], [16] inform the server about *what* to push, based on the state of the client's current cache. In [17], an approach that decides *when* to push or not depending on the size of the respective object is presented and evaluated.

Further, means to quickly deliver objects with high priority at an early stage, based on tracking object dependencies and push, is presented in [11], [18]. Recent work utilized server push in order to allow the server to choose the path on which to send resources, in presence of multi-interface connectivity [19].

While these works *conceptually* propose or utilize push mechanisms, no empirical understanding exists on how and if at all push is used in the wild. By measuring and analyzing push usage in the wild we provide the first *empirical* understanding of push as key feature of H2. Besides a classification of the pushed content, our measurements identify several non-published push strategies and potential inefficiencies. By this we aim to empirically elucidate push usage and inform the ongoing debate on defining push mechanisms.

IV. Availability and Usage of HTTP/2

We start by broadly exploring the server-side availability of H2 by large-scale active measurements of the *www* infrastructure. These measurements currently represent the largest assessment of the global H2 deployment and enable our study of H2 server push in Section V.

[2]More information available at http://isthewebhttp2yet.com, currently containing data from Nov. 11, 2014 to Nov. 16, 2016

A. Set of Probed Infrastructures

To probe for H2 availability, our measurements utilize multiple input datasets covering a large portion of the Internets' server infrastructure. These data sets fall into two categories: *1)* domain lists and *2)* IP lists resulting from port scanning the entire IPv4 address space.

Domain Lists. We utilize *i)* the Alexa Top 1M list, *ii)* complete set of .com and .net domains available at Verisign [20], and the complete set of .org domains available at PIR [21]. At the time of writing, the .com/.net/.org zone alone contained 151.4M distinct domains (see Table I), representing $\approx 50\,\%$ of the domain name space [22].

IP Lists. Lastly, we obtain the set of all publicly reachable servers on H2 port 443 by scanning the entire IPv4 address space. For the latter part, we extend ZMap [23] to support scanning for TLS ALPN and NPN on top of simply enumerating open ports. Thus we can already scan for H2 announcing hosts and thereby drastically reduce the set of hosts that we need to scan for *full* H2 support.

These two viewpoints enable us to broadly comment on the current H2 adoption and to complement prior work which focused on the Alexa Top 1M list [4]. We summarize our data sets in Table I and discuss the methodology used to assess the H2 adoption next.

B. Methodology for Assessing H2 Adoption

Given these three datasets, we want to determine *full H2 support*, i.e., if we are able to fetch the landing page by using H2 from these domains. To realize this probing, we utilize the H2-capable Nghttp2[3] library to establish H2 connections to all entries in our datasets. Since this step requires to perform a large number of probes, we distribute the experiment to a set of workers located in the same network. We prefix all domains with a `www.`, if not already present as we found greater coverage of A records in the DNS for these domains. We instruct the Nghttp2 library to timeout connections after 10 s to exclude unresponsive hosts. In case of a successfully established H2 connection, we issue a `GET` request for the `/` page of the domain/host. We follow up to 10 redirects and once we downloaded the requested page, we parse its HTML content, count objects from external domains (non-same origin) and additionally download all same-origin objects, i.e., content that is potentially pushable. Additionally, we identify push indications by the server and also download these if present (to be used for push analysis in the next section). We summarize our datasets and results in Table I and next discuss their details.

C. Global H2 Adoption

We base our assessment of the global H2 adoption on two measurements performed in September 2016 and January 2017. Measurements for all data sets were completed over the course

[3]https://nghttp2.org/

	Alexa 1M		Varvello [4]		.com/.net/.org		ZMap IP	
	Sep. 2016	Jan. 2017	Sep. 2016	Jan. 2017†	Sep. 2016	Jan. 2017	Sep. 2016	Jan. 2017
# Domains	1M (100.00%)	1M (100.00%)	236.7K (100.00%)	241.9K (100.00%)	153.1M (100.00%)	151.4M (100.00%)	–	–
# IPs	–	–	–	–	–	–	669.6K* (100.00%)	849K* (100.00%)
Full H2 sup.	99.3K (9.93%)	125.4K (12.54%)	165.6K (69.97%)	168.4K (69.62%)	3.1M (2.03%)	5.3M (3.47%)	598.1K (89.31%)	766.6K (90.3%)
Using Push	125 < (0.01%)	117 < (0.01%)	98 < (0.01%)	100 < (0.01%)	6.4K < (0.01%)	7K < (0.01%)	95 < (0.01%)	118 < (0.01%)
No H2 neg.	534.8K (53.48%)	521.1K (52.11%)	1.4K (0.6%)	6K (2.5%)	44M (28.73%)	43.8M (28.92%)	4.2K (0.62%)	5.1K (0.6%)
Con. timeout	81K (8.1%)	71.9K (7.19%)	1.6K (0.66%)	1.7K (0.71%)	41.9M (27.39%)	38.8M (25.65%)	8.1K (1.2%)	4.7K (0.56%)
Con. failed	143.5K (14.35%)	131.7K (13.17%)	183 (0.08%)	1.4K (0.58%)	29.8M (19.45%)	29.7M (19.61%)	3.7K (0.55%)	971 (0.11%)
TLS error	74.5K (7.45%)	68.4K (6.84%)	361 (0.15%)	2.5K (1.04%)	13.5M (8.8%)	13.9M (9.16%)	1.1K (0.16%)	1.9K (0.22%)
Redirect H1	37.6K (3.76%)	40.7K (4.07%)	62.3K (26.32%)	53.5K (22.12%)	700.5K (0.46%)	1M (0.67%)	40K (5.98%)	46.3K (5.45%)
DNS failed	26.1K (2.61%)	37.2K (3.72%)	203 (0.09%)	3.5K (1.44%)	20M (13.07%)	18.8M (12.45%)	1.9K (0.29%)	3K (0.35%)
App. timeout	1.4K (0.14%)	1.5K (0.15%)	3K (1.26%)	2.8K (1.14%)	66.1K (0.04%)	82.7K (0.05%)	1.3K (0.19%)	4.8K (0.56%)
Miscellany	1.8K (0.18%)	2K (0.2%)	2.1K (0.87%)	2.1K (0.86%)	36.3K (0.02%)	40.9K (0.03%)	11.3K (1.69%)	15.7K (1.85%)

TABLE I: Summary of H2 availability scan over different datasets. Top rows (i.e., #Domains or #IPs) define the relative basis for the reported figures. *Please note, these are the amount of IPs after ZMap ALPN/NPN enumeration over 2^{32} IPs of which 60 M are TLS-enabled on TCP port 443. †At time of writing, the latest available data set was provided on November 16, 2016.

of one week in the respective measurement period. We next discuss the obtained H2 adoption figures. If not mentioned otherwise, the reported statistics in the remainder of this Section correspond to our latest measurement performed in January 2017 (we summarize both measurements in Table I).

Alexa Top 1M. We start by exploring the Alexa Top 1M domain list. We chose this dataset since it contains popular domains typically hosted on large infrastructures (e.g., CDNs) that are likely to have a higher adoption rate than unpopular sites and it further allows comparisons with prior work [4]. The list contains 1M domains of which 52.11 % announce no H2 support in the NPN/ALPN negotiation (TLS handshake), 31.3 % fail to connect (various reasons, see Table I), and 4.07 % redirect to H1. The remaining 125.4K (12.54 %) are H2-capable. These H2-enabled domains are served by 64.6K distinct IPs located in 1.8K ASes. This grouping by AS shows that H2 support is significantly driven by a small set of infrastructures which enabled H2 support. Concretely, the top 10 ASes already account for 88.8K domains and the top 2 ASes (i.e., Cloudflare and Google) account for 66.9K domains. In other words, 53.4% of the H2-capable domains are served by only two infrastructures. Out of 125.4K H2-enabled domains, we find 117 domains to use server push on their landing page.

Varvello et al. To allow comparability with prior work [4] we include the list of domains which announce H2 support via NPN/ALPN obtained from http://isthewebhttp2yet.com. This list was obtained by scanning the list of Alexa Top 1M domains accumulated over the course of one year. We remark that the latest update to this list was on November 16, 2016, scanned by us in January 2017. To verify our scan setup, we reproduce these related results by probing for H2 support expressed via NPN/ALPN in the TLS handshake (and thus replicate the original scan setup). Full agreement would yield an NPN/ALPN failure rate of 0% since this list was originally generated out of domains that announce H2 support during the NPN/ALPN negotiation. Probing this domain list with our scan setup yields agreement with this related finding as indicated by the low amount of NPN/ALPN failures. In total, we find 69.62 % of the probed domains to be fully H2 capable (69.97 % in our September measurement). A larger fraction of 22.12 % domains is not H2 capable since the server redirects to H1—even though H2 support is announced in the NPN/ALPN negotiation. We thus conclude that our scan setup is able to verify related results and indicates the correct functioning of our setup. Out of 168.4K H2-enabled domains, we find 100 domains to actively use server push.

.com/.net/.org. To asses the H2 support on a larger dataset that is not influenced by page popularity and to possibly obtain a larger corpus of sites that utilize server push, we next probe the complete list of .com/.net/.org domains. This list contains 151.4M domains, of which 565.0K are also included in the scanned Alexa Top 1M list. H2 usage is not advertised by 28.92 % of the probed domains, 66.9 % fail to connect via TLS, and 0.67 % redirect to H1. The surprisingly high failure rate can be explained by two aspects: *i)* TLS is not configured (28.77 % connection refused + TLS error) and *ii)* unreliable/unreachable or misconfigured infrastructure (25.65% connection timeouts (10 s) and 13.07 % DNS failure of which 66 % are unresponsive authoritative DNS server (SRVFAIL) and 32.8 % are missing A records (NXDOMAIN)). We find 5.3M H2-enabled domains (3.47 %). These domains are served by 252.6K distinct IPs located in 3.3K ASes. Grouping domains by the serving AS again reveals that only few infrastructures dominate the H2 adoption. That is, 72.5 % of the domains are served by the top 10 ASes and 59.3 % by the top 3 ASes (i.e., 27.6 % Squarespace, 15.9 % Automatic, and 15.8 % Cloudflare). This skewed distribution again highlights that few but large infrastructures adopted H2 early. The adoption rate of 3.47 % is lower than for the Alexa Top 1M list (12.54 %) due to the larger diversity in unpopular sites. Notably, while the number of registered .com/.net/.org domains *declined* from 153.1M domains in September to 151.4M domains in January, the number of H2-enabled sites *increased* from 3.1M to 5.3M domains. This highlights the increasing H2 adoption. Out of 5.3M H2-enabled domains, we find 7K to use push.

ZMap IPs. Moreover, we probe the entire IPv4 address space (4 B IPs) using ZMap for servers announcing H2 support in their ALPN or NPN negotiation. Out of 60 M hosts open on port 443 (TLS), 849K distinct IPs responded to our probes and announced H2 support, out of which 170.1K IPs are also included in our .com/.net/.org dataset. We remark that our ZMap module does not capture the entire set of H2-capable servers due to missing Server Name Indication (SNI) information (i.e., forward DNS names for the probed IPs). This is because SNI-capable servers require a domain name to be present in the request to successfully complete the negotiation. In the absence of domain names the ALPN/NPN negotiation fails (recall that we are now port scanning IPs without knowledge of their forward DNS names). Internet-wide we find 766.6K H2 capable IPs. Note that these IPs may deliver multiple websites (e.g., CDNs) and are thus an indicator for available *infrastructure* rather than H2-enabled *websites*. Out of 766.6K H2 enabled IPs, 118 are using push.

CDN Support. We now focus on analyzing Akamai as one of the largest CDNs. Akamai enabled H2 for a subset of its customers in November 2015 [24]. Due to its distributed nature, it does not dominate our AS top list. To identify domains *fully* hosted by Akamai (i.e., having their *www.* A record point to Akamai), we analyze the CNAME chain observed during DNS resolution. By filtering our data with the identified Akamai CNAMEs, we find 230.5K .com/.net/.org domains and 12.7K Alexa listed domains to be served by Akamai. Out of these, only 0.5% of the .com/.net/.org domains and 7.7% Alexa listed domains deliver their landing page over H2. This is because Akamai H2 support is currently enabled only for some of its customers [24]. However, since Akamai requires no H2-capable customer infrastructure to deliver H2 [24], Akamai can drastically increase the H2 adoption by a factor of 13 (Alexa) to 210 (.com/.net/.org) at *any time* by enabling H2 for all of its customers (not yet counting the much larger set of sites *partially* using Akamai, e.g., to deliver images). The fact that such a drastic increase happens is reflected in our measurements: the H2 enabled domains served by Akamai in the Alexa Top 1M List (.com/.net/.org) increased from 523 (349) in September to 976 (1,097) in January, respectively.

Server Software. Last, we briefly comment on server software driving the H2-capable web as identified by the server field in the response header. Grouped by IP over all datasets, few server software dominate: Nginx 51.0%, IdeaWebServer 18.5%, LiteSpeed 9.2%, Apache 4.3%, and Microsoft IIS 5.4% for all probed IPs, respectively. The server software distribution of the H2 web thus differs from the H1 web, for which Apache is found on $\approx 20\,\%$ and Nginx on $\approx 8\,\%$ of all IPs.

D. Conclusion

We broadly assessed the H2 adoption by utilizing large-scale measurements providing us with a unique view on the current Internet-wide H2 adoption. During the analysis of our measurements (in Sep'16 and in Jan'17), we already experienced an increase in H2-enabled sites by 65.83 % from 3.24M to 5.38M. Our analysis shows that while H2 already is deployed on $< 12.54\,\%$ of the probed domains, its current adoption is mainly driven by only a few early adopters operating large infrastructures (e.g., Cloudflare). These infrastructures have the potential to continue to increase the H2 adoption, e.g., Akamai alone can drastically increase the H2 adoption by a factor of 210 once enabled for all of its customers.

V. Usage of HTTP/2 Push

Motivated by its potential to improve Page Load Time (PLT), we study the usage of H2 server push in the wild. While there is currently interest in the research community on *proposing* new push strategies (see e.g., [16], [17], [18]), no strategy is defined in the H2 *standard* [5]. It further remains unclear if push is already deployed. In the absence of this empirical understanding of push usage, we start by identifying pushing servers and continue by analyzing pushed content, its influence on PLT, and by identifying concrete (re-)push strategies.

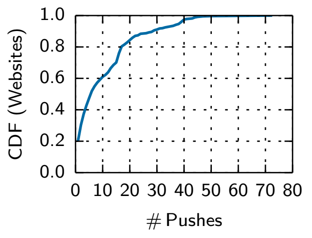
Push Resource Selection. Since the H2 standard does not define how resources that should be pushed are specified, we investigated how current setups decide what to push and when. We first investigated how major CDNs found in our data set, i.e., Akamai and Cloudflare enable server push. Akamai, currently allows manual configuration of resources to be pushed by the CDN edge servers. On the other hand, Cloudflare relies on the origin server to insert a `Link: </resource/>; rel=preload;` header in the HTTP response or corresponding prefetch links in the HTML content (as implemented by `nghttpx`). Identifying pushable objects is either done manually or automatically by analyzing static webpages for resources. We observed that already for popular content management systems (e.g., Joomla, Drupal, or WordPress) plugins exist that automate server push resource selection. Here, again the `Link` header is used to define resources that should be pushed by the frontend. While Cloudflare's solution offers greater flexibility, it requires the initial resource causing the push to be available at the edge server. It thus prolongs server pushes until the origin resource is fetched, which can be avoided when the server push mapping is known up front as used by Akamai. `Link` header inspection for server push is also implemented in H2 web servers by current Apache releases using `mod_http2`, the `h2o` and `nghttpx` web servers, or NodeJS `node-http2`. However, many web servers allow explicit direct configuration of pushed objects without having to wait for the `Link` header saving push latency.

Dataset. We base our analysis on H2 push-enabled pages identified in Section IV. That is, out of 5.38M H2 supporting sites, 7K are using push on their *landing page*. Of these push-enabled sites, 6.4K belong to a domain parker (domainstaff.com) which broadly registers domains for resell, each hosting the same advertisement page. To not bias our results, we exclude them from further analysis, leaving us with 595 sites.

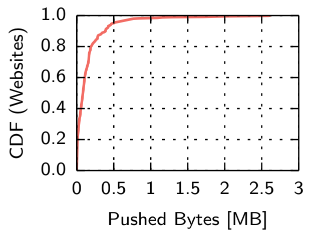
H2 usage increasing, push remains low & stable. While we observed the overall H2 adoption to increase from September to January, we remark that in contrast the share of push-enabled webpages remains rather stable. One reason for this trend can be the fact that H2 can be enabled by upgrading the web server software, while enabling server push needs explicit configuration. This need for explicit configuration can thus render push usage as indicator for *true* H2 adoption.

A. Identifying Pushes

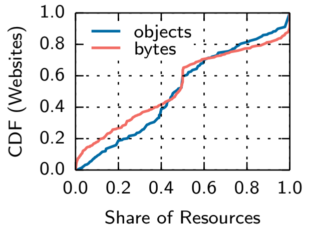
Analysis of Pushed Objects. We start by providing an empirical understanding on *i)* how often push is already being



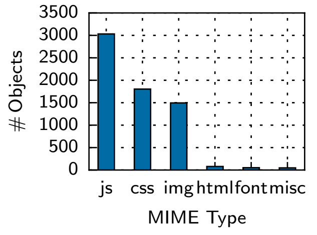
(a) CDF: # pushed objects/page



(b) CDF: pushed bytes/page

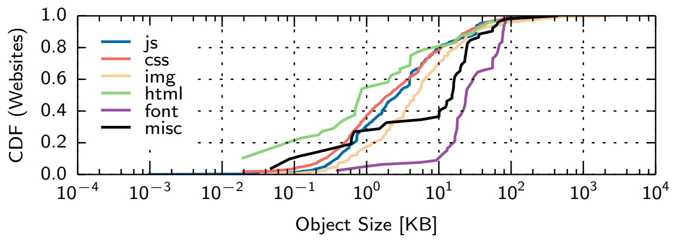


(c) Share of pushed resources from all resource.

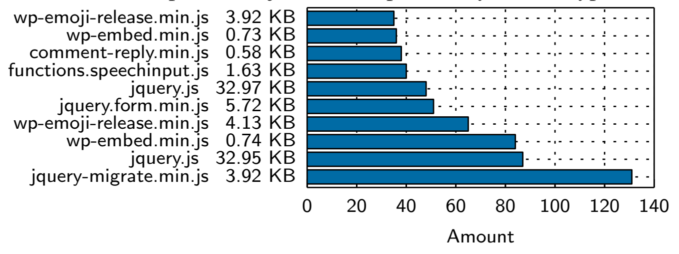


(d) Pushed content by MIME type.

Fig. 2: Amount and type of pushed objects from the dataset of 595 domains.



(a) CDF of pushed object sizes separated by MIME types.



(b) Top ten pushed objects (grouped by hash) across webpages.

Fig. 3: Content analysis of of pushed objects.

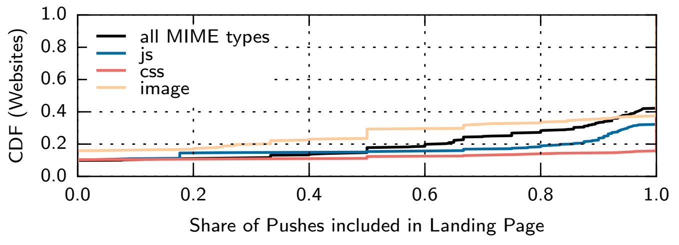


Fig. 4: Pushed resources referenced in the landing's page. Each line shows the fraction of overall resources of the same type.

used and *ii)* what content is being pushed. Therefore, we first analyze the number of objects pushed by each of the 595 pages and show the distribution in Figure 2(a). We observe that 50 % of these pages push no more than 6 objects, yet around 20 % already push at least 17 objects, with a maximum of 72 objects pushed by a single website. Moreover, Figure 2(b) depicts the total amount of bytes pushed per site as CDF. We detect that 50 % (80 %) of pushing sites push less than 81.75 KB (192.90 KB), with the maximum being 2,606 KB. Since this view omits site complexity (i.e., the total number of objects), we next analyze the ratio of pushed to total objects from the same origin (i.e., excluding external objects such as ads) as CDF in Figure 2(c). We observe that 50 % of all websites push at least 46.67 % (48.14 %) of their objects (bytes). Moreover, already 59.96 % (63.83 %) of websites push 50 % of their resources, respectively. In other words, many sites deploy push to send a large fraction of their landing page for potential faster loading due to omitted client requests.

Next, we turn our attention to the analysis of pushed content. We therefore plot the push object frequency by MIME type in Figure 2(d). The top three pushed object types are JavaScripts (46.57 %), CSS (27.71 %), and images (22.96 %). They are good push candidates since they are required to render the landing page and are cacheable. We show their object size distribution by MIME type as CDF in Figure 3(a). Altogether, we observe that 97.91 % (76.21 %) of the pushed objects have a size of $\leq$ 100 KB (10 KB). The MIME types of JavaScript, CSS, and images show a quite similar size distribution while HTML pages are typically rather small in comparison.

Network Resource Waste? One relevant question is if pushed objects are page-specific or if they are common frameworks used across multiple sites and thus could benefit from caching. We therefore cluster objects by their SHA-256 hash and show the frequency of same-hash objects in Figure 3(b). 3.49K (53.59 %) of 6.51K pushed objects are indeed unique by their hash. The remaining (46.41 %) object are pushed by multiple sites and can be attributed to web frameworks in different versions (e.g., 184 sites (30.92 %) push versions of jQuery). Recall that a generally recommended practice is to serve these frameworks from an external provider to render them cacheable over multiple sites. Thus, they should never appear in any push. However, since they do, this practice of unnecessary pushes can be interpreted as waste of network resources.

Of particular interest are pushed objects which are not directly referenced in the HTML of the landing page, but are later incrementally discovered by a browser requiring multiple round trips without push. We find that a large portion of pages (57.87 %) only pushes resources that are directly embedded on the landing page (c.f. Figure 4). We attribute this to content management system plugins that automatically deduce the resources to push by parsing the generated HTML output. Additionally, 10.02 % of sites only push objects not referenced in the landing page's HTML, which indicates the use of a push strategy that is based on other factors.

Push Order. In lack of current standardization or best practices on which resources types should be prioritized, we examined the order in which resources are pushed. In this scenario there are multiple possible arrival dates to investigate: *i)* the order in which the `PUSH_PROMISE` frames are received, *ii)* the order when a particular pushed stream was completely received. For the first received `PUSH_PROMISE` (first fully received pushed object) the share between JavaScript 40.98 % (38.32 %), and style sheets 36.42 % (38.32 %) are equally distributed, while the share of images is 18.72 % (18.66 %). In fact, we observe that 71.93 % of pages clearly define a type driven-push strategy, i.e. they first announce to push resources of a certain type and then proceed to the next MIME type (e.g., first push all style sheets, then push all JavaScripts). However, we note that from the reception of the second push onward the distribution in which resources are received and announced start to differ, due to stream multiplexing. Indeed we observe that in cases where the page pushes more than one object, 68.93% of pushed streams were interleaved, i.e., the timestamp of the first byte and the last byte received were overlapping with other pushed streams. In fact, we observe that for 69.41 % of the pages the order in which pushed streams arrive is different from the order in which push streams are announced. This multiplexing effects might appear counterintuitive to a user explicitly specifying push resources, i.e., because the receive-order does not always coincide to announced order in this scenario. Furthermore, 14.45 % of pages deliver all their pushed objects strictly before the landing page's HTML, while 79.33 % pages pushes arrive strictly only after the landing page. Again, we observe that in 47.4 % pages the order of `PUSH_PROMISE` frames is in line with the resource order of the HTML, indicating that pushed resources were automatically obtained by parsing the HTML.

Temporal Push Stability. After initially discovering 595 pages that push objects on the landing page, we immediately started sampling the push behavior of every pages in an interval of 5 mins for 5 days using the same setup as described in Section IV-B in order to examine how stable pushes can be observed. Note that our sampling setup does not maintain any state per page, such as storing cookies or local storage. We observe that 506 pages (85.04 %) show a very deterministic push behavior, i.e., pushes could always be observed, while 74 pages (12.44 %) pushed not regularly (however, for 33 of those pages we could at least observe pushes regularly every hour). We attribute this behavior to load-balancing effects and web servers that control their push behavior based on the client's IP-address. Moreover, after 5 days, 15 pages (2.52 %) were no longer reachable or completely stopped pushing content.

B. Page Load Time

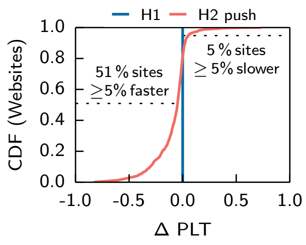
Methodology. Motivated by its potential to optimize the PLT, we first analyze the influence of H2 vs. H1 on the push-enabled webpages and additionally analyze the influence of server push. To perform our probes with realistic H2-capable browsers, we utilize the Selenium[4] framework to automate Chromium to fetch the push-enabled webpages. Each individual measurement is conducted with a fresh Chromium instance with a cold-cache, a fresh user profile and no per-site state (such as service workers, cookies, local storage etc.). We use Chromium 56 and we ensure H2 over TLS/TCP is used by disabling QUIC. Moreover, we measure the PLT as the time difference between the `connectEnd` (*start*) and the `loadEventEnd` events (*end*) obtained through the W3C Navigation Timing API. This measure excludes the connection establishment time, i.e., excluding influencing noise factors such as DNS resolution and the TCP and TLS handshake. This enables to focus on the protocol's influence on PLT alone, i.e., including all differences between H1, H2 and H2 with push. As the comparison metric between measurements, we use ΔPLT of two measurements, i.e., the relative difference for one webpage fetched with two assessed protocol variants (e.g., H1 vs. H2). To analyze the influence of push by enabling and disabling push support, we modified a Chromium variant to announce no push support by setting `SETTINGS_ENABLE_PUSH` to `0` in the initial `SETTINGS` frame, disallowing the server to push. Furthermore, we verified that the pages under consideration adhere to this setting. This permits to compare the PLT of a webpage that pushes objects with the PLT where no resources are pushed. Hence, for the no push case, all embedded resources are requested by the browser only upon detection when parsing the initial HTML document. These delayed requests require additional round trips and thus should in theory increase the PLT as compared to a push version where objects can be pushed without a request.

We continue analyzing the set of 595 websites that have full H2 support and push objects. We visit each page 30 times, each single iteration through the set of pages takes 4 hours (5 days in total), i.e., every page is visited within this interval having *H2 with push*, *H2 without push*, and *H1* enabled. We conducted this measurement in parallel to our stability sampling described in Section V-A, however, from independent machines with different IP addresses, but within the same network. In the following we present our findings for the set of 506 pages with a stable push behavior in order to present precise results.

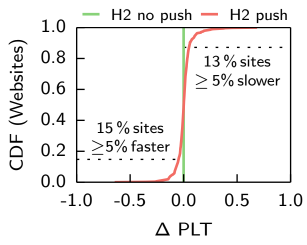
Influence on Page Load Time. We start by comparing the median PLTs for H1 and H2. We show the relative increase ($\Delta PLT < 0$ on the left) and decrease ($\Delta PLT > 0$ on the right) in PLT as compared to H1 in Figure 5(a). Since the PLT measurements contain noise (e.g., jitter), we selected a threshold of $\Delta PLT \leq 5\,\%$ for which we consider webpages to load equally fast. We observe that 50.92 % of the websites served with H2 are above this threshold, i.e., they load faster than their H1 counterparts and achieve reductions for the PLT of up to 83 %. However, 5.19 % of the H2 websites experience *slower* loading times of up to 72 %. This highlights that the majority of websites benefit from enabling H2, however in some cases H2 does not speed up the page load time, which complies with results given in related work (cf. Section III).

To evaluate how effective server push is used, we compare H2 PLT with and without push being enabled, see Figure 5(b). We again use a ΔPLT threshold of 5 % to indicate pages that are not impacted by push. As promised, push *can* indeed yield PLT improvements. Concretely, 14.75 % of the studied websites are above the threshold: here server push *speeds up* the PLT of up to 63 % compared to H2 without push. A large body of pages is not influenced by push in our setup ($\Delta PLT \approx 0$), i.e., push neither helps nor harms. To our surprise, a fraction of

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

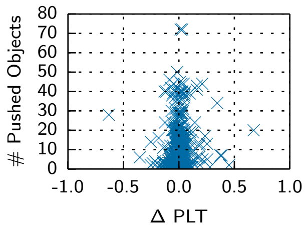


(a) H2 vs H1.

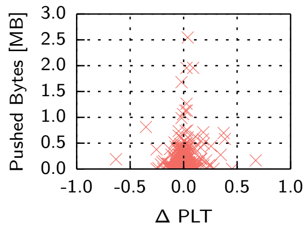


(b) H2 push vs. H2 w/o push.

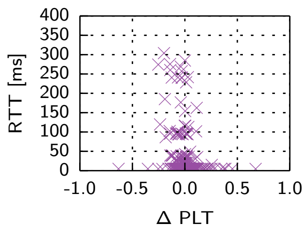
Fig. 5: Speedup of H2 with *push* vs. H1 and H2 *without* push.



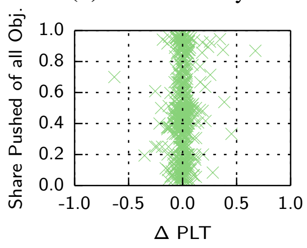
(a) Amount of pushed objects



(b) Amount of bytes



(c) Latency



(d) Fraction of all objects

Fig. 6: Distribution of ΔPLT under various aspects, with H2 *without* push as baseline.

12.84 % pages suffer from server push as they load up to 67 % *slower* with a median (average) of 9.75 % (13.25 %). Thus, the use of server push does not always decrease the PLT per se and cannot be considered a silver bullet. In fact, our results show that only 48.08 % of the pages had a $\Delta PLT \leq 0$. We could not attribute simple reasons for these performance drops to individual properties such as *i)* the total number of pushed objects/bytes (Figure 6(a) and Figure 6(b)), *ii)* network latency of the web server (Figure 6(c)), *iii)* share of pushed objects in comparison to the total number of objects included in the landing page (Figure 6(a)), *iv)* types of pushed resources, or *v)* resource type ordering. Hence, we acknowledge that the performance-determining factors of server push are either very page-specific, or are indeed more complex and require further analysis which however goes beyond the scope of this paper.

Conclusion. H2 keeps its promise to reduce PLT on a large set of the considered domains. We observed that push is used to deliver mostly medium sized JavaScripts, CSS, or images. Interestingly, the practice to push common web frameworks actually limits their cross-website cacheability. While some pages' PLT profit from server push, an equal share of pages is subject to reduced performance at the same time. We find that push performance cannot easily be attributed to how push is used and on what resources, motivating further research.

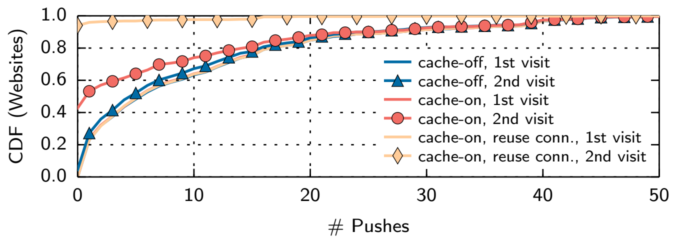


Fig. 7: Pushes in subsequent visits for various configurations. To improve visibility, we cut-off at 50 pushes (cf. Figure 2(a)).

C. Push Behavior on Reoccurring Visits

A practical challenge to be addressed by push strategies is avoiding to repeatedly push same resources over multiple visits by the same client, as these may already be in the cache. Pushing such resources is wasteful, moreover, upon a second visit the server could push other resources that are not yet in the cache. Currently, the H2 standard does not provide mechanisms to enable a cache-aware server push. Thus, we investigate whether or not this issue is present in our dataset.

Methodology. We use mitmproxy[5] to capture the traffic between browser and web server. Again, we automate the browser to visit a page, then visit a dummy page, and to visit the page again. We use different configurations (cache-enabled, cache-disabled) of Chromium to examine push in reoccurring visits. Figure 7 depicts the number of pushes observed in the first and the second visit. We detected that Chromium does not close the TCP connection to the server for up to 180 s of inactivity, even if the user has left the page. If the connection is reused, 5.96% of pages push responses on the second visit. To simulate a revisit after this timeout, we enforce a termination of the TCP connection in the proxy. We found that enabling the browser's cache does affect the push behavior: With caching enabled 57.23% of pages push objects on the second visit, compared to 95.53% of pages with disabled cache. Next, we present results for the cache-enabled scenario, where connections are not reused. Comparing the SHA-256 hashes we rediscover 97.39 % of all objects are pushed again, i.e., redundant transfers and thus waste of network resources. Moreover, 95.76 % of the re-pushing sites push solely objects that already were pushed in the previous visit. In 95.53 % of all re-pushing sites the amount of pushed objects was the same for both visits; additionally considering hash equality in 89.0 % of pages the encountered pushes were exactly the same. During our measurements, we discovered strategies we could not account to the aforementioned cache settings. Thus, we inspected the respective websites and derived further strategies, verified in individual evaluations. We describe these strategies (**S**) for reoccurring visits and illustrate possible disadvantages.

IP-based (S1). A server identifies clients by their IP and avoids to push on re-visits within a certain time. This is especially fragile in presence of NAT boxes or proxies, where multiple clients share one IP. Quantifying the use of this strategy is challenging, since we cannot reliably distinguish load-

[5]https://mitmproxy.org/

balancing effects and server configuration. However, during our stability measurement (Section V-A) we found 33 pages that are potential candidates.

Conditional Request (S2). The server decides to push objects based on the advertised modification date of the page in the `If-Modified-Since` HTTP request header, which is set by the browser if the requested page is cached. However, if the page is dynamic (indicated not to be cached), the browser will not send this header and thus resources are always pushed. We observed this strategy for Cloudflare-hosted websites.

Cookie-based (S3). A cookie is set by the server indicating which resources have been pushed. This is not ideal since the server assumes that client cache state is stable, while only the client can tell which resources are still cached.

Client-side Code (S4). A website executes JavaScript code that installs a HTML5 service worker upon the first visit. On the next visit, request are then answered by the service worker instead of being sent to the server. This interception enables to realize custom behavior, e.g., prevent extra pushes.

Conclusion. Although some pages deploy non-standardized measures to avoid pushing the same objects multiple times, which in fact would waste network resources (especially in a mobile scenario), there is enormous potential for improvement either by avoiding pushing objects that are already cached or pushing new fresh resources in the second visit.

VI. Discussion and Conclusion

This paper presents the first broad assessment of HTTP/2 (H2) adoption and server push in the wild. We study infrastructure as well as domains by probing *i)* the entire IPv4 address space, *ii)* the Alexa Top 1M list, and *iii)* the complete set of 151.4M .com/.net/.org domains. By this, we complement and validate prior work that solely focuses on the Alexa 1M list. In light of the H2 standardization in 2015, we find H2 still not to be widely deployed, but also observe an increase in deployment by 65.83 % from Sep'16 to Jan'17, totalling to 5.38M H2 domains in our datasets. This rising adoption can be explained by out of the box support in major web servers, which can provide H2 upon software upgrade. We also see that the H2 adoption is driven by few large infrastructures (e.g., CDNs). Since they have not yet enabled H2 for their entire customer base, they are capable of drastically increasing the adoption at *any time*, e.g., Akamai alone can increase their customer's H2 adoption by a factor of 210 in our datasets.

Analyzing H2 server push, regarded as key feature promising to further reduce PLT, we observe only hundreds of domains using it. This massive difference to the overall H2 adoption may be explained by the fact that server push needs to be actively configured, which can pose an enormous challenge as we have shown. Regarding its promise in PLT reduction, we observe that push *can* speedup PLT (true for $\approx 50\,\%$ of the sites), however, it *can* also slow down PLT (true for the other $\approx 50\,\%$). This effect cannot be attributed to the number of pushed objects, their size, nor the fraction of all objects that are pushed. We only observe a trend for extreme high latency links where server push can indeed reduce PLT. The observation that an approach designed to improve PLT can easily yield detrimental effects, motivates further research to better understand the complex nature of server push—especially towards its interplay with the underlying transport protocol. We further find some push practices to arguably waste network resources, either by preventing cacheability or by unnecessary transmissions. Without a better understanding of the effects of server push and the used push strategies, server push clearly remains behind its promised potential. In the absence of standardized push practice, we argue that it is now the right time to optimize push before inappropriate strategies find widespread adoption.

Acknowledgements

This work has been funded by the DFG as part of the CRC 1053 MAKI and the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains. Furthermore, we would like to thank Jens Hektor and Bernd Kohler (RWTH Aachen IT Center) for enabling and supporting our work.

References

[1] G. Maier *et al.*, "On Dominant Characteristics of Residential Broadband Internet Traffic," in *ACM IMC*, 2009.
[2] B. Ager *et al.*, "Anatomy of a Large European IXP," in *ACM SIGCOMM*, 2012.
[3] P. Borgnat *et al.*, "Seven Years and One Day: Sketching the Evolution of Internet Traffic," in *IEEE INFOCOM*, 2009.
[4] M. Varvello *et al.*, "Is The Web HTTP/2 Yet?" in *PAM*, 2016.
[5] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540, Nov. 2015.
[6] "SPDY: An experimental protocol for a faster web," http://www.chromium.org/spdy/spdy-whitepaper, retrieved: 10/05/2016.
[7] I. Grigorik, *High Performance Browser Networking*. O'Reilly, 2013.
[8] R. Peon and H. Ruellan, "HPACK: Header Compression for HTTP/2," RFC 7541, Dec. 2015.
[9] S. Friedl *et al.*, "TLS Application-Layer Protocol Negotiation Extension," RFC 7301, Oct. 2015.
[10] A. Langley, "TLS Next Protocol Negotiation Extension," IETF, Internet-Draft draft-agl-tls-nextprotoneg-04, Nov. 2012, Work in Progress.
[11] X. S. Wang *et al.*, "How speedy is SPDY?" in *NSDI*, 2014.
[12] H. de Saxcé, I. Oprescu, and Y. Chen, "Is HTTP/2 really faster than HTTP/1.1?" in *IEEE INFOCOM (WKSHPS)*, 2015.
[13] J. Erman *et al.*, "Towards a SPDY'Ier Mobile Web?" *IEEE/ACM Trans. Netw.*, vol. 23, no. 6, pp. 2010–2023, Dec. 2015.
[14] U. Goel *et al.*, "HTTP/2 Performance in Cellular Networks," in *ACM MobiCom*, 2016.
[15] J. Khalid *et al.*, "Improving the performance of SPDY for mobile devices," in *HotMobile Poster*, 2015.
[16] B. Han, S. Hao, and F. Qian, "MetaPush: Cellular-Friendly Server Push For HTTP/2," in *All Things Cellular Workshop*, 2015.
[17] I. N. de Oliveira *et al.*, "Should I Wait or Should I Push? A Performance Analysis of Push Feature in HTTP/2 Connections," in *ACM LANCOMM Workshop*, 2016.
[18] M. Butkiewicz *et al.*, "KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices," in *NSDI*, 2015.
[19] B. Han, F. Qian, and L. Ji, "When Should We Surf the Mobile Web Using Both Wifi and Cellular?" in *All Things Cellular Workshop*, 2016.
[20] Verisign, "Zone Files For Top-Level Domains (TLDs)," verisign.com.
[21] Public Interest Registry, "Zone File Access," http://pir.org/.
[22] R. van Rijswijk-Deij *et al.*, "A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements," *IEEE JSAC*, vol. 34, no. 6, pp. 1877–1888, Jun. 2016.
[23] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide Scanning and Its Security Applications," in *USENIX Security*, 2013.
[24] "Akamai http/2 support," https://http2.akamai.com/.