Can SPDY Really Make the Web Faster?

[1]Yehia Elkhatib, [2]Gareth Tyson and [3]Michael Welzl

[1]Lancaster University, UK [2]Queen Mary, University of London, UK [3]University of Oslo, Norway

Abstract—HTTP is a successful Internet technology on top of which a lot of the web resides. However, limitations with its current specification have encouraged some to look for the next generation of HTTP. In SPDY, Google has come up with such a proposal that has growing community acceptance, especially after being adopted by the IETF HTTPbis-WG as the basis for HTTP/2.0. SPDY has the potential to greatly improve web experience with little deployment overhead, but we still lack an understanding of its true potential in different environments. This paper offers a comprehensive evaluation of SPDY's performance using extensive experiments. We identify the impact of network characteristics and website infrastructure on SPDY's potential page loading benefits, finding that these factors are decisive for an optimal SPDY deployment strategy. Through exploring such key aspects that affect SPDY, and accordingly HTTP/2.0, we feed into the wider debate regarding the impact of future protocols.

I. INTRODUCTION

Web pages are increasing in complexity and size. The HTTP Archive reports that the global average web page size surpassed 1MB in April 2012 [1]. By January 2014, visiting one of the top 1000 sites incurs, on average, 1607kB of web page resources over 112 separate requests [1]. Such growth has been fuelled by the emergence of advanced web-based services (Web 2.0, SaaS cloud services, etc.), enhanced client device capabilities (JavaScript browser runtimes, display), and increased downlink speeds [4]. This growing complexity, however, can dramatically slow down page retrieval. Unfortunately, this has negative consequences, and very real ones in the case of commercial websites: most users cannot tolerate more than 2 seconds of page load delay [20], and increments of just 100ms on shopping websites can decrease sales by 1% [18]. The converse is similarly true: decreasing delay can have a powerful enhancing effect, with Google claiming to have increased ad revenue by 20% through cutting 500ms from load times. To reduce page load times, various extensions to HTTP have been proposed. However, in practice, little progress has been made, with many web servers, proxies and browsers being slow to adopt these new tweaks (e.g. pipelining [10]).

In light of these observations, some have proposed developing a new web protocol. Such efforts include Microsoft's Speed+Mobility [27] and HTML5 Websockets; most prominent, however, is Google's SPDY [3]. This has already begun to see deployment by prominent organisations such as Google, Twitter, Akamai and Facebook, whilst also being adopted as the base for HTTP/2.0 [13] by the HTTPbis Working Group. Despite this, we still possess a limited understanding of its behaviour, overheads and performance: does it offer a fundamental improvement or just further tweaking? A number of early stage studies have explored the topic in an attempt to answer this question. They offer a range of results, with some claiming significant gains and (curiously) others claiming

rather negative results. This paper seeks to resolve these issues by analysing the circumstances under which SPDY improves page load times and those where the opposite is true.

To achieve this, we perform a large-scale evaluation of SPDY both in the wild and in a controlled setup. We crawl the Alexa top 10k websites to discover those domains that have deployed SPDY, finding that 208 of them (2.1%) in October'12 and 271 (2.7%) in April'13 support SPDY. Using some of these websites, we execute a large number of probes to measure the performance of SPDY in the wild. Confirming our suspicions, we find highly variable results between different websites and samples: SPDY has the potential to both benefit and damage page load times. Motivated by this, we perform a large body of controlled experiments in an emulated testbed to understand the reasons behind these performance variations. We permute different factors to identify how website types and network characteristics affect SPDY behaviour. This offers insight to guide SPDY deployment in terms of page design and provider-side infrastructural decisions.

The rest of the paper is organised as follows. §II provides background and highlights related work. §III describes our measurement toolkit and environments. §IV presents the results of comparing SPDY to HTTPS on live websites. We then dissect the factors affecting SPDY in an emulated network testbed, namely network characteristics (§V) and infrastructure setup (§VI). §VII concludes and discusses future work.

II. BACKGROUND AND RELATED WORK

A. SPDY

SPDY is an application-layer web protocol that reuses HTTP's semantics [10]. As such, it retains all features including cookies, ETags and Content-Encoding negotiations. SPDY only replaces the manner in which data is written to the network. The purpose of this is to reduce page load time, which it does by introducing the following mechanisms:

- *Multiplexing*: A framing layer multiplexes streams over a single connection, removing the need to establish separate TCP connections for transferring different page resources.
- *Compression*: All header data is compressed to reduce the overheads of multiple related requests.
- *Universal encryption*: SPDY is negotiated over SSL/TLS and thus operates exclusively over a secure channel in order to address the increasing amounts of traffic sent over insecure paths (e.g. public WiFi).
- *Server Push/Hint*: Servers can proactively *push* resources to clients (e.g. scripts and images that will be required). Alternatively, SPDY can send *hints* advising clients to pre-fetch content.
- *Content prioritisation*: A client can specify the preferred order in which resources should be transferred.

SPDY consists of two components. The first provides framing of data, thereby allowing things like compression and multiplexing. The framing layer works on top of secure (SSL/TLS) persistent TCP connections that are kept alive as long as the corresponding web pages are open. Clients and servers exchange *control* and *data* frames, both of which contain an 8 bytes header. Control frames are used for carrying connection management signals and configuration options, while data frames carry HTTP requests and responses. The second component maps HTTP communication into SPDY data frames. Multiple logical HTTP streams can be multiplexed using interleaved data frames over a single TCP connection.

B. Related Studies

There are a number of preliminary studies on the performance of SPDY. The first was a Google white paper [2] which showed significant performance benefits over both HTTP (27-60%) and HTTPS (39-55%). Somewhat conflicting accounts followed from Akamai [22] and Microsoft [21]. Akamai's test showed a marginal benefit over HTTPS (4.5%) alongside a decrease in performance (-3.4%) when compared to HTTP. Microsoft offered slightly more positive results, but still did not attain the high levels reported by Google.

An Internet Draft [31] found a mix of results highlighting that SPDY's performance is dependent on a number of factors. Nevertheless, no further insight is offered into such variance. More recent studies [22], [26], [29], [6], [9] reported that SPDY provides minimal improvement, if any, over HTTP. Such gain was found to increase for high latency connections [29], [6], but is also reported to be lost over 3G networks due to the interplay between different layers (browser, TCP, cellular network) [9]. All these studies, however, were carried out either via a SPDY/HTTP proxy between client and server or on a single-server setup which offers no insight into the effect of infrastructure parameters.

In light of the above findings, the only clear conclusion is that SPDY has variable performance. Therefore, it is necessary to ascertain exactly how network and/or infrastructural settings affect SPDY's performance.

III. MEASUREMENT METHODOLOGY

A. Measurement Toolkit

SPDY is designed to reduce page load time for end users. We therefore focus on client-side measurements, for which we have built a toolkit based on the Chromium browser. This is the logical choice as Chromium constitutes Google's reference implementation of SPDY. Chromium also offers sophisticated logging features that allow us to extract statistics via automated scripting. We use Chromium 25 (running over Ubuntu Desktop 12.04.2) via the Chrome-HAR-capturer [5] package, which interacts with Chromium through its remote debugging API. To ensure authenticity, we maintained all of Chromium's default settings, apart from disabling DNS pre-fetching in order to include DNS lookup time in all measurements.

When invoked, our measurement toolkit instructs Chromium to fetch a particular webpage. Once this is completed, the toolkit extracts detailed logs in the form of HTTP Archives (HAR) and Wireshark network traces. It then processes them to calculate metrics of interest. Traditionally, page load time has been measured by the Document Object Model (DOM) being fully loaded, (e.g. [29]). However, this is not suitable for our purposes, as it also captures browser processing time that is strictly HTML-related, e.g. arbitrating the style hierarchy. Such factors are not relevant to SPDY, as it solely deals with data transmission. Instead, we evaluate this *transport* protocol by measuring the time spent performing network interactions. Thus, we use an alternate metric which we coin the *Time on Wire* (ToW), which is calculated from the Wireshark traces as the period between the first request and last response packets, giving us precise timestamps for the page transmission delay.

Using this toolkit, we employ Chromium in a non-obtrusive manner to retrieve a number of webpages in a range of different environments. The collated measurements allow us to explore the performance of SPDY. The rest of this section details the environments we utilised the measurement toolkit in.

B. Measurement Setups

The measurements are separated into two groups, both using the above toolkit. First, we briefly perform *live tests*, probing real-world deployments to understand the level of performance variation in existing deployments. Second, we expand on these results using *emulated network tests*, creating our own controlled SPDY deployment in a local testbed. The latter allows us to analyse performance in a deterministic fashion by varying and monitoring the impact of various key factors (namely network conditions and website infrastructure setup). In both cases, a large number of samples are taken to ensure statistical significance. Overall, we have collected over 70,000 probes (12,000 live and 58,000 controlled).

1) Live Tests: First, we probe (from various vantage points) web sites that have deployed SPDY in their infrastructures. For this we implemented a crawler that probes the top 10k Alexa websites[1] recording their individual protocol support. We then select the top 8 Alexa websites that implement SPDY. We choose only the highest Alexa ranked website from every distinct online presence and disregard similar sites (i.e. facebook.com (#1) but not fbcdn.net (#202); google.com (#2) but not google.co.in (#12), google.com.hk (#22), etc.; and so on). The list is shown in Table I.

The selected websites provide a range of resource sizes, resource count, and domain count. In terms of their respective delivery infrastructures, we note that all appear to use CDNs with the exception of WordPress. We confirm this using `whois` as well as other means (e.g. trying to directly access a CloudFlare IP address with a browser yields an error message *from* CloudFlare). It seems, unsurprisingly, that the employed CDN dictates the supported SPDY version and the TCP Initial Window (IW).[2] We therefore posit that our results for these sites are representative of the performance for other customers of the same respective CDN.

---

[1] All Alexa ranks henceforth are of April 23,[rd] 2013.

[2] Note that increasing the IW size is another closely related component in Google's "Make the Web Faster" project. We determined IW by sending self-crafted TCP packets, carrying out a successful handshake, followed by sending a HTTP GET for a large resource (e.g. a static image). We then noted the number of ensuing packets (which go unacknowledged) as the server's IW.

TABLE I: Live SPDY-enabled Websites

| Site | Alexa Rank | SPDY Rank | Resources | | | SPDY | | | | Av. RTT |
			Count	Av. Size (kB)	Domains	Web Server	Version	CDN	IW	(ms)
Facebook	1	1	20	12.56	4	OpenCompute	2	Akamai	7	92
Google	2	2	7	41.29	2	Google Web Server	3	Google	7	8
YouTube	3	3	50	10.63	4	Apache	3	Google	7	8
Blogspot	11	4	31	5.03	6	Google Servlet Engine	3	Google	7	17
Twitter	13	6	7	46.40	3	Twitter Frontend	3	EdgeCast	10	158
WordPress	23	8	13	7.92	4	nginx	2	(none)	10	91
imgur	96	24	133	11.78	58	nginx	2	CloudFlare	10	8
youm7	485	65	270	11.07	54	nginx (via Varnish)	2	CloudFlare	10	150

Using our measurement toolkit, we periodically probe each website using HTTP, HTTPS and SPDY. In this paper, we focus on HTTPS as a baseline comparison as, like SPDY, it encrypts its data. However, where possible, we also include HTTP, considering that many websites have no interest in securing their connections. In both cases, when HTTP and HTTPS are used, we avoid bias by forcing Chromium to pursue the Next Protocol Negotiation (NPN) handshake as SPDY does. The probes are carried out for each website in an alternating sequence of protocols with 2 seconds between each run. For SPDY, we select the highest non-experimental version that the server supports (listed in Table I). Tests were carried out from different sites: Lancaster, Dublin, and Tokyo. Due to space, we only discuss the Lancaster set as the other results provide very similar outcomes. The Lancaster tests ran on weekdays between 12pm and 5pm BST between 20 and 23/5/2013, totalling 1.06 million GET requests with 500 samples taken for each website and protocol combination.

2) Emulated Network Tests: The above live experiments provide useful context to the current state of affairs, but are somewhat limited in what they can tell us. Although the comparison they provide is fair, it would be difficult to definitely and neutrally ascertain SPDY's performance as it is subject to variations in the network, and tightly bound to the particular deployment under test: its characteristics (e.g. web server, SPDY module/proxy) and its status (e.g. server load). To address these issues, we extend our tests by creating our own SPDY deployment in an emulated testbed interconnected via a LAN. This allows us to control the various network parameters to understand how they impact performance.

Our testbed consists of a client and server setup. The client runs our measurement toolkit and is connected via 100Mbps Ethernet. We then emulate various network conditions: the Linux `tc` utility is used to throttle bandwidth by shaping traffic with Hierarchy Token Bucket queuing [8], and NetEm [15] is used (at the server) to specify a deterministic round trip time (RTT) and packet loss ratio (PLR). The server runs Ubuntu Server 12.04 with Apache 2.2.22 web server supporting both HTTP and HTTPS, as well as spdy/3 via the mod_spdy 0.9.3.3-386 module (the reference implementation provided by Google). Using this server, we clone a set of SPDY websites discovered in the wild; these are each intended to be representative of a broader class of top Alexa websites[3]:

- *Twitter*: A simple page with few (7) resources (average size 46.4KB). This is comparable to other top Alexa

websites such as Google and Blogspot and their regional versions, Wikipedia, and Soso (Chinese search engine).
- *YouTube*: A relatively complicated page, with a fair number (50) of resources (average size 10.63KB). Examples of similar websites include Amazon's regional websites, AOL, Alibaba (Chinese e-commerce portal), About.com, and DailyMotion.
- *imgur*: A complicated page, with a large number (133) of images and flash (average size 11.78KB). Similar websites include QQ (Chinese messaging website), TaoBao (equivalent to eBay), The NY Times, CNN, and MSN.

The described setup enables us to have a single server-side SPDY implementation and a single SPDY-capable web browser, which rules out software discrepancies (note that we also later use multiple servers). This method also allows us to control the network characteristics in order to experiment with SPDY under different network conditions. Moreover, server and connection load are also controlled. Using this testbed, we apply the same methodology as in the live experiments, generating repeated page requests for the chosen webpages using HTTPS and SPDY. The exact details of the parameters investigated will be presented in §V and §VI.

IV. LIVE RESULTS

We begin by performing experiments with existing deployments of SPDY. The aim here is not an exhaustive study but, rather, to form a general idea of the benefits being gained by some of those who have so far adopted SPDY. To achieve this, each website in Table I is probed 500 times to calculate its ToW. Figure 1 displays the cumulative distribution functions (CDFs) of these measurements, and Table II summarises them. Note that the HTTP results are not included for websites that redirect such requests to HTTPS.

Confirming our analysis of past studies, the results are *not* conclusive. We find no clear winner among the three protocols. Instead, we observe large performance variations between different websites, as well as between different samples for the same website. We find that notable improvements are, indeed, gained in some cases. On average, ToW is reduced by 7% for Facebook, 4.7% for YouTube, and 9.7% for youm7. The biggest winner is the Twitter front-page, with an average ToW reduction of 10.6%. This, however, is not a universal observation. In other cases, improvements are far more modest; for example, imgur only achieves a meager improvement of 0.8%. Moreover, we find websites that suffer from their use of SPDY; an average ToW increase of 6.0% for Blogspot and 15.1% for Wordpress. Ironically, the biggest sufferer is Google with a 20.2% increase in ToW for their search homepage.

---

[3]We plotted the resource count and size of the top 1000 Alexa sites, and chose a website from each of the three prominent clusters in the graph (not included due to space).
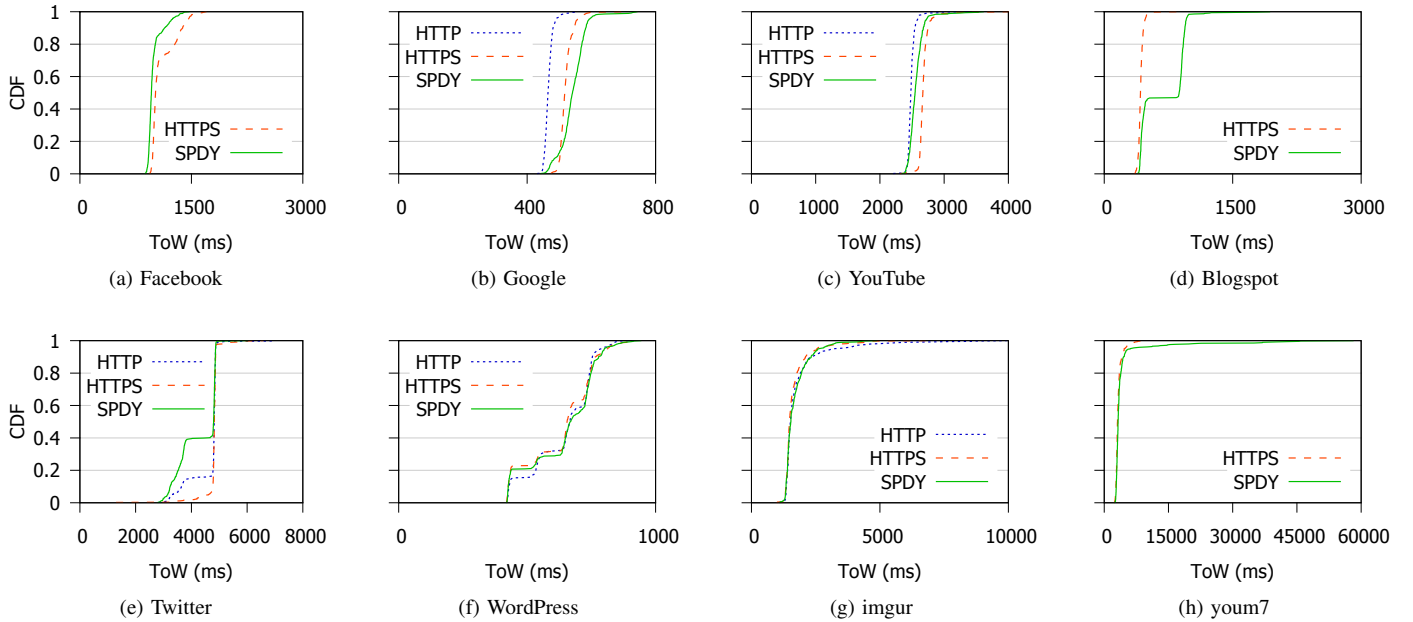
Fig. 1: ToW of Live SPDY-enabled Websites

There is certainly no one-size-fits-all operation with SPDY, as all websites alternate between SPDY and HTTP optimality.

TABLE II: Gain in ToW for Live SPDY-enabled Websites

Site	Average Gain in ToW (SPDY over HTTPS)
Facebook	7.0%
Google	-20.2%
YouTube	4.7%
Blogspot	-6.0%
Twitter	10.6%
WordPress	-15.1%
imgur	0.8%
youm7	9.7%

These experiments therefore raise some interesting (yet serious) questions. From a research perspective, one might ask why these notable variations occur? From an administrator's perspective, the next logical question would then be how to maximise the benfit of deploying SPDY? The remainder of this paper now explores these questions using emulated experiments. Whereas the live experiments limit our control to the client-side, emulated experiments allow us to dissect all aspects to understand the causes of such variations.

V. INVESTIGATING THE NETWORK EFFECT

To explore SPDY's performance variations, we now perform controlled experiments. We aim to understand the effect of different network conditions on the performance gains of SPDY over HTTPS. We mirror the representative websites (Twitter, YouTube, imgur) by employing `wget` to retrieve all resources (images, stylesheets, scripts, etc.) and converting their links accordingly. We measure ToW under a variety of delay, bandwidth, and loss settings. As a guideline, we use typical cellular network conditions due to SPDY's focus on mobile communications [30].

A. Delay

First, we inspect the impact that round trip time (RTT) has on SPDY's performance. In a real environment, this varies a lot between different requests due to client locations and path characteristics [17]. To remove any variance, we fix bandwidth (BW) at 1Mbps[4] and Packet Loss Ratio (PLR) at 0%, whilst changing the RTT between the client and server in a range from 10ms to 490ms. Following this, we perform 20 requests for each website using each configuration with both SPDY and HTTPS. The results are presented in Figure 2 as the average percentage improvement in ToW of SPDY over HTTPS.
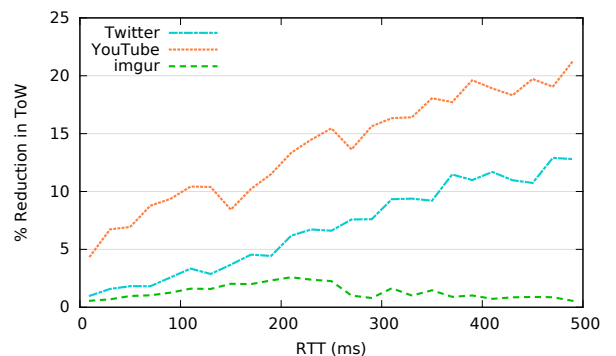

Fig. 2: Effect of Round Trip Time (BW=1Mbps, PLR=0%).

In contrast to the live experiments, we see that SPDY *always* achieves better performance than HTTPS in this setup. With low RTTs, these benefits are marginal: requests with RTTs below 150ms achieve under 5% improvement on average. These benefits, however, increase dramatically as the RTT

---

[4]Internationally, cellular connectivity speeds hover around 1Mbps [4], [28].

goes up. In the best case (490ms RTT for YouTube), SPDY beats HTTPS by 21.26%. The results effectively highlight the key benefit of SPDY: stream multiplexing. As RTT goes up, it becomes increasingly expensive for HTTPS to establish separate connections for each resource. *Each* HTTPS connection costs one round trip on TCP handshaking and a further two on negotiating SSL setup. SPDY does this only once (per server) and hence reduces such large waste by multiplexing streams over a single connection. By inspecting the HAR logs, we find that SPDY saves between 66% and 94% of SSL setup time, creating significant gains in high delay settings.

There is also a notable variation between the different webpages. For Twitter and YouTube, SPDY's ability to multiplex is well exploited by retrieving Twitter's 7 resources and YouTube's 50 resources in parallel. YouTube is by far the greatest beneficiary with an average improvement of 13.81% over HTTPS, whilst Twitter comes second with 6.87%. The benefits for Twitter are less pronounced because there are fewer streams that can be multiplexed, therefore reducing the benefits of SPDY over HTTPS's maximum of 6 parallel TCP connections (note that this limit of six is hard coded, based on the amendment [16] to the limit set by RFC 2616 [10]).

Perhaps more interesting, though, is the fairly steady behaviour exhibited by imgur across all delay values. At first, one would imagine imgur to benefit greatly from SPDY due to its ability to multiplex imgur's large number of resources (133). However, performance is very subdued: its overall average is 1.32%. To understand this, we inspect the HAR logs to see what is occurring 'under the bonnet'. Figure 3 depicts a breakdown of the HTTPS and SPDY retrieval times for Twitter and imgur at RTT=490ms and BW=2Mbps. We choose this particular subset of our experiments as it provides network conditions where both websites achieve equal SPDY-induced improvement ($\approx$15%), and hence provides a fair comparison. The figure shows the fraction of time spent in the five key stages of page retrieval: connect, send, wait, receive and SSL. We notice that the make-up of these retrievals is remarkably different. As expected, HTTPS spends a lot of time in the connect and SSL phases, establishing TCP and SSL connections (respectively). This increases for imgur, which has 19 times as many resources as Twitter. On the other hand, SPDY greatly reduces the connect and SSL stages but spends a huge proportion of time in wait. This phase begins when the browser issues a request for a resource, and ends when an initial response is received back. The receive phase is time spent receiving the response data until it is loaded into the browser's memory. In the case of SPDY, wait includes not just the network latency between the client and server but also the time requests are blocked until multiplexed onto the wire. For imgur, SPDY cuts connect time by 94% but inflates wait time by more than 9 times. As emulated RTT was the same for both protocols, it appears that this inflation in wait time is an unfortunate product of SPDY's multiplexing, but we are unable to exactly ascertain *why* multiplexing is creating this much delay. In other words, SPDY's savings in establishing new connections is compromised with multiplexing overhead for highly complex webpages served over a single connection.

B. Bandwidth

Next, we inspect the impact that client bandwidth has on performance. Once again, this parameter spans a wide range
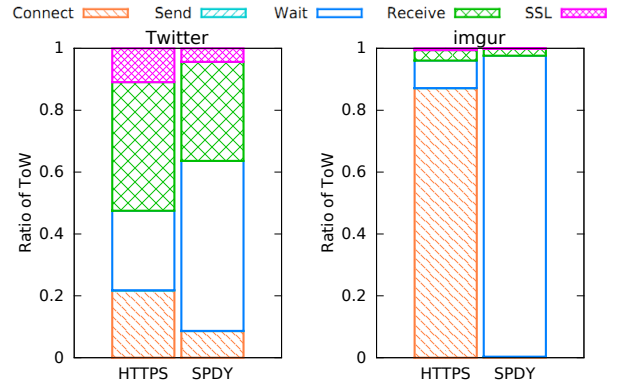

Fig. 3: Breakdown of HAR Times for Twitter and imgur at RTT=250. SPDY spends 49% in Wait for Twitter, and 97% for imgur due to its high resource count.

of values across the globe [25], [4]. This time we fix RTT at 150ms and PLR at 0.0%, while setting client bandwidth to values between 64Kbps and 8Mbps. A first in first out tail-drop queue of 256 packets length is used to emulate commodity routers commonly used as residential and public gateways [19], [12]. The results are presented in Figure 4.
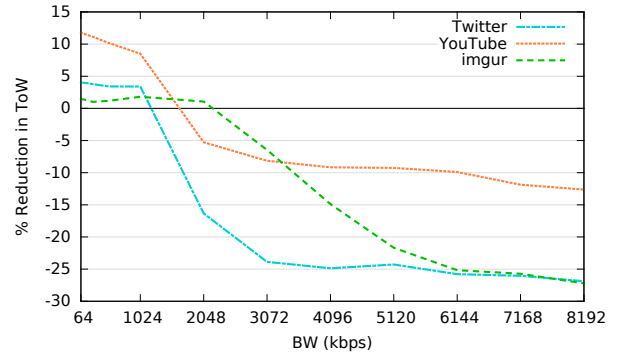

Fig. 4: Effect of Bandwidth (RTT=150ms, PLR=0%).

This graph reveals a very different story to that of delay. Confirming the findings of the live experiments, we see that SPDY *does* have the potential to lower performance, and significantly so. This occurs with a clear trend that favours lower capacities. At 64Kbps, on average, clients witness a 5.75% improvement over HTTPS, compared to a 22.24% decrease at 8Mbps. Initial impressions suggest that bandwidth variations have a larger detrimental impact on SPDY's performance.

We now have two dimensions of impact — RTT and bandwidth — where SPDY prefers high delay, low bandwidth (<1Mbps) environments. As previously discussed, the reason behind SPDY's sensitivity to RTT is relatively easy to measure by inspecting the HAR logs. However, its relationship with bandwidth is rather more complicated. To understand this, we turn our attention to the network traces. We find that the separation between RTT and bandwidth is not particularly distinct. This is because HTTPS tends to operate in a somewhat network-unfriendly manner, creating queueing delays where bandwidth is low. The bursty use of HTTPS' parallel connections creates congestion at the gateway queues, causing upto

3% PLR and inflating RTT by upto 570%[5]. In contrast, SPDY causes negligible packet loss at the gateway.

The network friendly behaviour of SPDY is particularly interesting as Google has recently argued for the use of a larger IW for TCP [7]. The aim of this is to reduce round trips and speed up delivery — an idea which has been criticised for potentially causing congestion. One question here is whether or not this is a strategy that is specifically designed to operate in conjunction with SPDY. To explore this, we run further tests using IW=$\{3, 7, 10, 16\}$ and bandwidth fixed at 1Mbps (all other parameters as above). For HTTPS, it appears that the critics are right: RTT and loss increase greatly with larger IWs. In contrast, SPDY achieves much higher gains when increasing the IW without these negative side effects. It therefore seems that Google have a well integrated approach in their "Make the Web Faster" project. Interestingly, we observe that the key reason that this increase in RTT and loss adversely affects HTTPS is that it slows down the connection establishment phase, creating a similar situation to that presented earlier in Figure 2. Obviously, this congestion also severely damages window ramping over the HTTPS connections. We can tangibly observe this by inspecting the client's TCP window size, which scales far faster with SPDY than any one of the parallel HTTPS connections; this alone leads to an average of $\approx$10% more throughput than that of HTTPS.

While this explains SPDY's superior performance at low bandwidths, it does not explain its poor performance as capacities increase. As soon as bandwidth becomes sufficient to avoid the increased congestion caused by HTTPS, the benefits of SPDY begin to diminish. This is particularly the case for websites with fewer resources, like Twitter. To understand this, we breakdown the operations performed by SPDY and HTTPS. Figure 5 presents the results for YouTube as an example. Again, the two protocols have very different constitutions. HTTPS spends a large proportion of its time in the connect phase, setting up TCP and SSL. In contrast, SPDY spends the bulk of its time in the wait phase. Deep inspection reveals streams blocking until the connection is free to transmit. In line with our previous findings, this highlights that SPDY does not always do an effective job of multiplexing. Whereas, previously, this was caused by the complexity of the webpage, here it appears that high capacity transmission is also a challenge. Thus, as bandwidth increases, HTTPS can amortise the costs of TCP and SSL setup by exploiting the higher raw throughput afforded by opening parallel TCP sockets. We also observe that this situation occurs particularly when dealing with larger resources (e.g. images in Twitter), as window size can be scaled up before each connection ends in HTTPS. In contrast, SPDY appears to struggle to fill TCP's pipe as the server waits for new requests for each object from the client. Indeed, the Wireshark traces show TCP throughput reductions between 2% and 10% in the case of SPDY due to this problem compared to that of the parallel HTTPS connections. It would therefore seem that SPDY's default use of a single TCP connection might be unwise in circumstances of high bandwidth.

Fig. 5: HAR Times for different Bandwidth values (YouTube, RTT=150ms, PLR=0%).

C. Packet Loss Ratio

Finally, we inspect the impact of packet loss on SPDY's performance. We fix RTT at 150ms and BW at 1Mbps, varying packet loss using the Linux kernel firewall with a stochastic proportional packet processing rule between 0 and 3%[6]. Figure 6 presents the results.
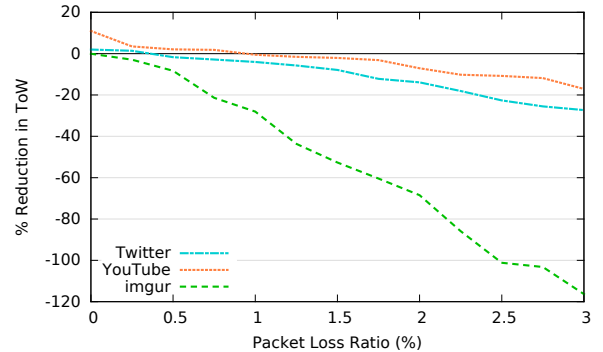
Fig. 6: Effect of Packet Loss (RTT=150ms, BW=1Mbps).

Immediately, we see that SPDY is far more adversely affected by packet loss than HTTPS is. This has been anticipated in other work [26] but never before tested. It is also contrary to what has been reported in the SPDY white paper [2], which states that SPDY is better able to deal with loss. The authors suggest because SPDY sends fewer packets, the negative effect of TCP backoff is mitigated. We find that SPDY does, indeed, send fewer packets (upto 49% less due to TCP connection reuse). However, SPDY's multiplexed connections persist far longer compared to HTTPS. Thus, a lost packet in a SPDY connection has a more profound setback on the long term TCP throughput than it would in any of HTTPS' ephemeral connections, the vast majority of which do not last beyond the TCP slow start phase [24]. Furthermore, packet loss in SPDY affects all following requests and responses that are multiplexed over the same TCP connection. In contrast, a packet loss in one of the parallel HTTPS connections

---

[5]We also experimented with different gateway queue sizes. Generally, increasing queue size caused longer delays and more loss: upto a 920% RTT increase and 5% PLR with a 512 packets queue size, but only 296% maximum RTT inflation and 1% PLR with a queue of 64 packets.
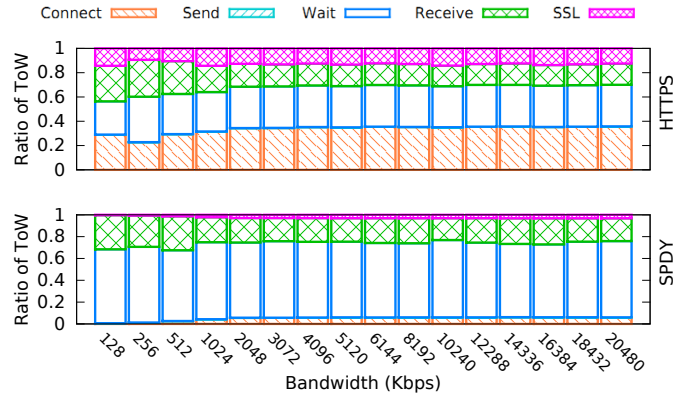
[6]Reports of cellular packet loss vary: $\approx$ 0.2%–1.9% in the US, 2–3% in Europe, and $\geq$ 3% in other regions [14]. It is also quite high for WiFi [23]. We therefore consider 0–3% to be an appropriate parameter range.

would not affect the other connections, neither concurrent nor subsequent (assuming HTTP pipelining is not used, which is commonly the case). In essence, HTTPS 'spreads the risk' across multiple TCP connections. On average, we found that SPDY's throughput is affected by packet loss up to 7 times more than HTTPS (all experiments were performed using the default Linux CUBIC congestion avoidance algorithm).

It is also important to note that the probability of packet loss is higher in SPDY. According to [11], the probability of experiencing loss increases in proportion to the position of the packet in a burst chain. Hence, the chance of experiencing a packet tail drop is much higher for longer lived connections such as SPDY's. Thus, not only does SPDY react badly to packet loss, the chance of it experiencing loss is also higher. This is effectively highlighted in Figure 6; imgur, which has the longest transfer time (by far), exhibits extremely poor performance under packet loss.

Finally, these results indicate that SPDY may not perform that well in mobile settings, one of its key target environments [30]. Whilst both SPDY's high delay and low bandwidth support is desirable in this environment, the benefits can be undone by relatively low levels of packet loss (e.g. 0.5%).

VI. IDENTIFYING THE IMPACT OF THE INFRASTRUCTURE

The previous section has investigated the performance of SPDY under different network conditions between a single client and server. However, our original crawling of the Alexa Top 10k highlighted a tendency for providers to implement a practice known as *domain sharding*. This is the process of distributing page resources across multiple domains (servers), allowing browsers to open more parallel connections to download page resources. Figure 7 presents a CDF of the number of shards we discovered. We find that apart from front-less domains (such as CDN endpoints like akamaihd.net), all websites employ some degree of domain sharding. Here, we investigate domain sharding in order to understand the implications of this infrastructural design choice on SPDY.
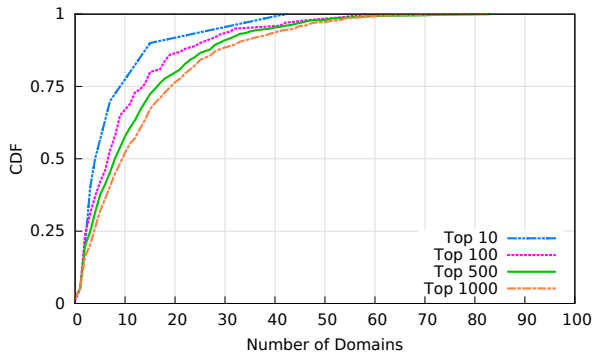

Fig. 7: Number of Alexa Websites Resource Domains

A. Number of Shards

To inspect the impact of sharding, we recreate the earlier experimental setup but mirror the webpages across multiple servers, as occurs in real setups. We consider 7 shards, i.e. servers, an appropriate upper limit as our measurements find that 70 of the top 100 Alexa websites have 7 or fewer shards.

Each shard is configured as in §III-B2. In order to control the number of shards per experiment, we use a script to adapt the HTML to configurations between 1 and 7 shards. The client is configured with 1Mbps bandwidth, 150ms RTT and 0% PLR. Figure 8 presents the results of 100 runs at each configuration.
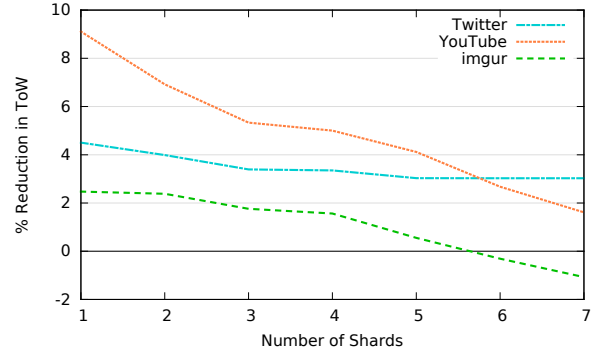

Fig. 8: Effect of the Number of Shards

We first note that sharding distinctly decreases SPDY's gain for YouTube and imgur. As the number of shards increases, so does the maximum number of parallel HTTPS connections. SPDY, too, is forced into creating multiple parallel TCP connections (one to each server). Hence, *both* protocols are allowed to capitalise on increased parallelism. However, the benefits achieved by HTTPS outweigh those of SPDY as the former gains 6 new TCP connections per shard, a large performance boost that, in essence, offers SPDY-like multiplexing. This, therefore, reduces the overall improvement offered by SPDY. Another ramification of sharding evident from the examples of YouTube and imgur, is that as SPDY opens more connections, it multiplexes fewer streams per connection. This diminishes the returns of multiplexing which is SPDY's main competitive advantage over HTTPS. This suggests, based on findings in §V-B, that increasing the servers' IW would give SPDY an advantage and the potential to tip the balance in its favour.

The case of Twitter provides a different insight. Here, fairly steady results are achieved across all sharding levels. SPDY gains marginal improvements over HTTPS by reducing the number of round trips, which is dictated by the number of resources in a page. For such a page with only 7 resources, SPDY saves between one and two round trips at 1 shard (depending on whether all resources were requested together or at different times as the page is rendered). With more shards, the number of round trips that SPDY potentially saves is reduced to only one, if any, due to its reduced ability to multiplex. Whereas, in the case of HTTPS, more shards means fewer resources (and hence fewer parallel connections) per shard. This has the effect of gradually decreasing HTTPS' parallelism as the number of shards increase, hence allowing SPDY to continue to retain an edge.

In summary, we deduce that SPDY loses its performance gains as a website is sharded more. However, these negative results are not ubiquitous and vary remarkably depending on the number of page resources. This raises a few questions about SPDY deployment. Are the benefits enough for designers and admins to restructure their websites to reduce sharding? What about third party resources that cannot be consolidated, e.g. ads and social media widgets? Can SPDY be redesigned to

multiplex across domains? Is proxy deployment [26] rewarding and feasible as a temporary solution? The success of SPDY (and thereupon HTTP/2.0) is likely to be dependent on the answers to precisely these questions.

B. Number of Multiplexed Streams

So far, we have seen that sharding can create a significant challenge to SPDY's performance by forcing it into HTTP-like behaviour and by limiting its ability to perform stream multiplexing. To further inspect this, we now directly study the impact of this multiplexing by artificially changing the maximum number of streams allowed per connection. This allows us to control exactly the degree of multiplexing afforded by SPDY. We vary this value from 1 to 100 (which is the default in Apache, as recommended by the SPDY draft [3]) whilst mirroring the three websites on a single server. We perform these retrievals for a variety of RTTs. We choose to vary RTT because of the discovery that many of the bandwidth impacts are actually products of the inflated RTTs caused by queuing. Bandwidth is fixed at 1Mbps and PLR at 0%.

In Figure 9, the average improvement in ToW (over HTTPS) for each multiplexing degree is displayed as a trend line, whilst the ToW reduction over different RTT values is shown as a heatmap to elicit more generalisable results. In all cases, SPDY's multiplexing has the potential to improve the ToW. For YouTube and imgur, we see a direct relationship between these benefits and the level of multiplexing afforded by SPDY. Here, these benefits plateau at 10 streams for YouTube and at 30 for imgur. In contrast, the results for Twitter remain relatively steady for all levels of multiplexing.

To explore the different results for each page, we inspect the nature of their resources, as well as SPDY's recorded behaviour when accessing them. We confirm that these results are a product of the complexity of the webpages in terms of their resources. Twitter benefits little from increasing the multiplexing degree, as it only possesses 7 resources, i.e. no further benefits can be achieved when multiplexing beyond this level. The inverse case is found with YouTube (50 resources) and imgur (133 resources), which clearly can exploit multiplexing levels beyond 7 streams. Preventing this from happening has dire ramifications: when allowing SPDY to multiplex fewer than 6 streams for YouTube and imgur, it performs worse then HTTPS. This therefore confirms the negative impact that sharding will have on SPDY's deployment, where multiplexing capabilities could be severely undermined. We found these observations to be true for even more complicated websites, e.g. the New York Times website (148 resources).

To better understand the relationship between performance and page complexity, we perform regression analysis to look at the multiplexing level (m) required to outperform HTTPS for a website with a given number of resources (r). This is done for all websites under test in addition to three other websites we experimented with. We find that $m \approx r/4$, with a very strong fit ($R^2 = 0.98537$, p-value$= 8.0684 \times 10^{-5}$). This is not a robust model and is not intended to be so; it effectively highlights the impact that page type will have on SPDY's performance. Another interesting point here is that intuition would perhaps lead towards a $r/6$ relationship, due to the maximum number of parallel HTTPS connections. Instead, m is found to be of greater value. We are not able to pinpoint

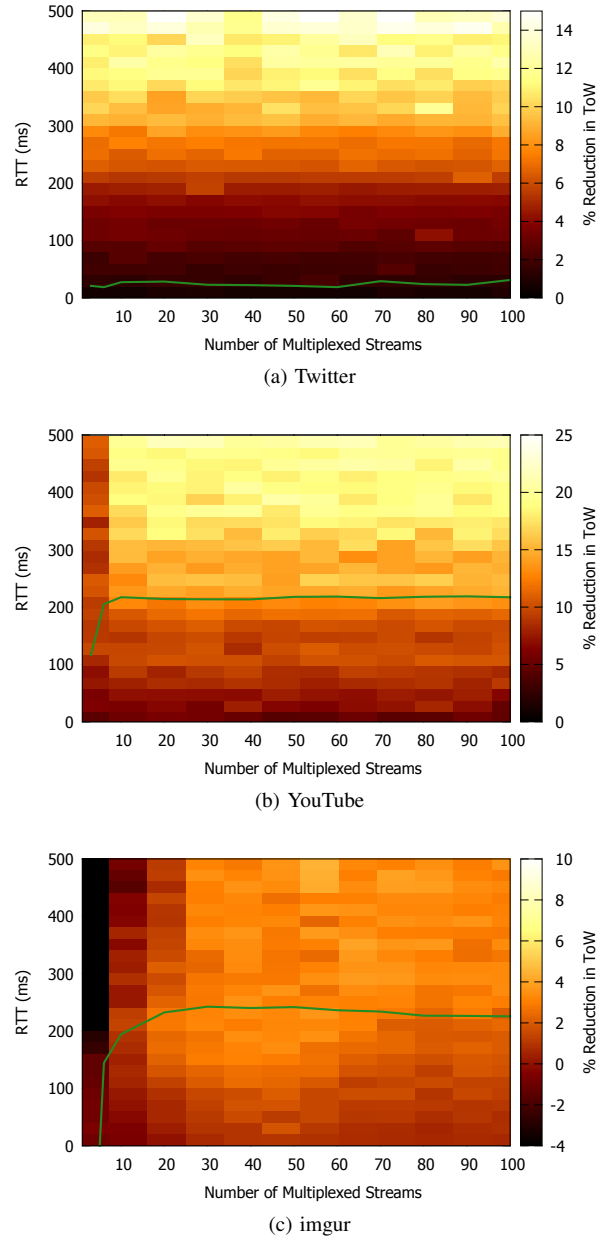

(a) Twitter

(b) YouTube

(c) imgur

Fig. 9: Effect of the Number of Multiplexed Streams per SPDY Connection over Varying RTTs.

the reasons behind this, but it could be attributed to SPDY's multiplexing overheads diagnosed in §V.

VII. CONCLUSIONS & FUTURE WORK

SPDY provides a low-cost upgrade of HTTP, aiming to reduce page load times leading to improved user experience. To do this, it introduces a variety of new features, including stream multiplexing and header compression. Currently, the behaviour and performance of SPDY are quite poorly understood, exacerbated by the often conflicting results reported by various early stage studies. Our own live experiments confirmed these observations, highlighting SPDY's ability to both decrease and increase page load times.

We therefore turned our efforts to identifying the conditions under which SPDY thrives. We found that SPDY offers maximum improvement (over HTTPS) when operating in challenged environments. We concluded that stream multiplexing is at the heart of SPDY's performance, allowing it to deal with low bandwidth and high delay situations far better than HTTPS. This feature minimises the number of round trips required to fetch resources. It also facilitates more disciplined congestion control, allowing SPDY to outshine HTTP on low bandwidth links. This also helps support further network enhancements such as increasing TCP's IW. On the other hand, SPDY's multiplexed connections last much longer than HTTP's, which makes SPDY more susceptible to loss and the subsequent issues with TCP backoff.

We also investigated the impact of infrastructural decisions on SPDY's performance, namely the prevalent practice of domain sharding. We observed that SPDY's benefits are reduced in sharded environments where SPDY is prevented from maximising on multiplexing. We predict this could have palpable implications on website design and deployment strategies, especially considering that ultimate shard consolidation is practically extremely difficult due to third party resources. Finally, we observed throughout our experiments that page type has huge influence on SPDY's performance: SPDY favours pages with more and larger resources, as opposed to pages with a very large number of small resources which induces perceptible multiplexing overheads.

Trying to anatomise all aspects pertaining to a protocol like SPDY is a daunting task that is beyond the scope of any one paper. We have explored only a subset of SPDY's overall parameter space. Our future work intends to expand to experiment with alternate network configurations and different page characteristics, and to further inspect some of the findings (e.g. multiplexing overheads) in further detail. We also plan to study other SPDY features such as Server Push and Hint. Finally, this work should feed into the wider discussion regarding HTTP/2.0, and the future of the web.

VIII. ACKNOWLEDGMENTS

REFERENCES

[1] HTTP Archive. http://httparchive.org/.

[2] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[3] M. Belshe and R. Peon. SPDY Protocol. http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00, Feb 2012. IETF Network WG.

[4] D. Belson. Akamai state of the internet report. 5(4). http://www.akamai.com/stateoftheinternet, 2012.

[5] A. Cardaci. chrome-har-capturer. https://github.com/cyrus-and/chrome-har-capturer.

[6] A. Cardaci, L. Caviglione, A. Gotta, and N. Tonellotto. Performance evaluation of SPDY over high latency satellite channels. In R. Dhaou, A.-L. Beylot, M.-J. Montpetit, D. Lucani, and L. Mucchi, editors, *Personal Satellite Services*, volume 123 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 123–134. Springer, 2013.

[7] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. RFC 6928 (Experimental), Apr 2013.

[8] M. Devera. Hierachical token bucket theory. http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm, May 2002.

[9] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'ier mobile web? In *Proc. of CoNEXT*, pages 303–314. ACM, 2013.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Jun 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[11] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. SIGCOMM*. ACM, 2013.

[12] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *ACM Queue*, 9(11):40–54, Nov 2011.

[13] I. Grigorik. Making the web faster with HTTP 2.0. *ACM Queue*, 11(10):40–53, Oct 2013.

[14] M. V. Heikkinen and A. W. Berger. Comparison of user traffic characteristics on mobile-access versus fixed-access networks. In *Proc. PAM*, pages 32–41. Springer, 2012.

[15] S. Hemminger. Network emulation with NetEm. In *Proc. of Linux Conf. Australia*, pages 18–23. Citeseer, 2005.

[16] IETF HTTPbis WG. Ticket #131: increase connection limit. http://trac.tools.ietf.org/wg/httpbis/trac/ticket/131, Sep 2008.

[17] S. Kaune, K. Pussep, C. Leng, A. Kovacevic, G. Tyson, and R. Steinmetz. Modelling the internet delay space based on geographical locations. In *Proc. Conf. Parallel, Distributed and Network-based Processing*, pages 301–310. IEEE, Feb 2009.

[18] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *IEEE Computer Magazine*, 40(9):103–105, 2007.

[19] F. Li, M. Li, R. Lu, H. Wu, M. Claypool, and R. Kinicki. Measuring queue capacities of IEEE 802.11 wireless access points. In *Proc. BroadNets*, pages 846–853. IEEE, Sep 2007.

[20] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Info. Tech.*, 23(3):153–163, 2004.

[21] J. Padhye and H. F. Nielsen. A comparison of SPDY and HTTP performance. Microsoft Technical Report MSR-TR-2012-102, 2012.

[22] G. Podjarny. Not as SPDY as you thought. http://www.guypo.com/technical/not-as-spdy-as-you-thought/, 2012.

[23] A. Sheth, S. Nedevschi, R. Patra, S. Surana, E. Brewer, and L. Subramanian. Packet loss characterization in WiFi-based long distance networks. In *Proc. INFOCOM*, pages 312–320. IEEE, 2007.

[24] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *SIGCOMM workshop on Measurements Up the Stack*, pages 49–54. ACM, 2011.

[25] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband internet performance: a view from the gateway. In *Proc. SIGCOMM*, pages 134–145. ACM, 2011.

[26] B. Thomas, R. Jurdak, and I. Atkinson. SPDYing up the web. *Commun. of ACM*, 55(12):64–73, Dec 2012.

[27] R. Trace, A. Foresti, S. Singhal, O. Mazahir, H. F. Nielsen, B. Raymor, R. Rao, and G. Montenegro. HTTP Speed+Mobility. http://tools.ietf.org/html/draft-montenegro-httpbis-speed-mobility-02, Jun 2012. IETF Network WG.

[28] N. Vallina-Rodriguez, V. Erramilli, Y. Grunenberger, L. Gyarmati, N. Laoutaris, R. Stanojevic, and K. Papagiannaki. When David Helps Goliath: The Case for 3G Onloading. In *Proc. HotNets*, pages 85–90. ACM, 2012.

[29] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proc. NSDI*, pages 473–486. USENIX Association, 2013.

[30] M. Welsh, B. Greenstein, and M. Piatek. SPDY Performance on Mobile Networks. https://developers.google.com/speed/articles/spdy-for-mobile, Apr 2012. Google 'Make the Web Faster' project.

[31] G. White, J. Mule, and D. Rice. Analysis of SPDY and TCP Initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00, Jul 2012. IETF Network WG.