Mind the Delay: The Adverse Effects of Delay-Based TCP on HTTP

Neil Agarwal
UCLA
neilagarwal@cs.ucla.edu

Matteo Varvello*
Nokia, Bell Labs
matteo.varvello@nokia.com

Andrius Aucinas
Brave Software
aaucinas@brave.com

Fabián Bustamante
Northwestern University
fabianb@cs.northwestern.edu

Ravi Netravali
UCLA
ravi@cs.ucla.edu

ABSTRACT

The last three decades have seen much evolution in web and network protocols: amongst them, a transition from HTTP/1.1 to HTTP/2 and a shift from loss-based to delay-based TCP congestion control algorithms. This paper argues that these two trends come at odds with one another, ultimately hurting web performance. Using a controlled synthetic study, we show how delay-based congestion control protocols (e.g., BBR and CUBIC + Hybrid Slow Start) result in the underestimation of the available congestion window in mobile networks, and how that dramatically hampers the effectiveness of HTTP/2. To quantify the impact of such finding in the current web, we evolved the web performance toolbox in two ways. First we develop Igor, a client-side TCP congestion control detection tool that can differentiate between loss-based and delay-based algorithms by focusing on their behavior during slow start. Second, we develop a Chromium patch which allows fine-grained control on the HTTP version to be used per domain. Using these new web performance tools, we analyze over 300 real websites and find that 67% of sites relying solely on delay-based congestion control algorithms have better performance with HTTP/1.1.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Application layer protocols**; **Network measurement**.

KEYWORDS

HTTP, TCP, congestion control algorithm, protocol design, web performance

ACM Reference Format:
Neil Agarwal, Matteo Varvello, Andrius Aucinas, Fabián Bustamante, and Ravi Netravali. 2020. Mind the Delay: The Adverse Effects of Delay-Based TCP on HTTP. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3386367.3431299

*Work partially done while at Brave Software.

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12.
https://doi.org/10.1145/3386367.3431299

1 INTRODUCTION

HTTP has underpinned web page loads for the past three decades, defining both message formats as well as transmission patterns for web transfers. Despite early concerns about ossification, the HTTP protocol has seen considerable evolution in recent years [24, 26, 28, 50]. For example, HTTP/2 (H2) [19] debuted in 2014 and introduced header compression, request multiplexing onto a single TCP connection, and the server push feature. More recently, the nascent HTTP/3 (H3) [20] proposal moves to UDP-based QUIC as the transport protocol, tightly integrating with its streamlined handshakes.

In parallel to the HTTP evolution, components lower in the transport stack have undergone an evolution of their own. In particular, there has been a steady shift from loss-based congestion control algorithms to delay-based variants, which rely on packet delay measurements as a signal of congestion. Examples of these algorithms are BBR [23], CUBIC's Hybrid Slow Start [31], and YeAH [17].

When studied in isolation, each new HTTP- or transport-level protocol's features appear to deliver significant promise, with little downside. For instance, Google reports that BBR offers considerable improvement over CUBIC (in both throughput and quality-of-experience metrics) [23]. Similarly, the request multiplexing feature that H2 introduced alleviates head-of-line blocking and connection setup overheads as compared to HTTP/1.1 (H1) [19].

As prior work has shown, there exists a complex interplay between these cross-stack network protocols which ultimately governs the performance of the applications that they support [22, 32, 50]. The focus of this paper is on understanding the fundamental interplay between HTTP variants in use today and delay-based TCP versions in the context of web page loads. This relationship is of critical importance as HTTP and TCP variants continue to evolve separately but operate together in the wild; we analyze it in detail, highlighting aspects that have been glanced over in past studies. In addressing this question, we make three contributions.

First, we perform controlled synthetic experiments to understand the interplay between HTTP and TCP congestion control (CC) across different network conditions and different protocol combinations. We find that delay-based variants (BBR, YeAH, and CUBIC + Hybrid Slow Start) favor H1 for large page sizes and cellular-like network conditions. We attribute the better performance of H1 to the combination of two behaviors: 1) delay-based variants tend to underestimate network capacity in jittery network conditions such as on mobile, and 2) the multiple TCP connections used by H1 allows it to make up for this underestimated capacity. These findings are important because they go against the de facto HTTP protocol mechanism used today which always opts for H2 when

available. Further, the upcoming H3 is expected to suffer from the same issue when coupled with delay-based congestion control.

Second, we extend the web performance toolbox to achieve fine-grained *visibility* and *control* on the HTTP and TCP interplay in the wild (*i.e.,* on real page loads). We built and open sourced Igor [14], a client-side tool which detects whether a domain/server runs a loss-based or delay-based CC algorithm. The main novelty of Igor is that it focuses on *pre-loss* behavior (during TCP slow start), and is thus able to operate with small web objects, which is more than often a necessity in today's web. Next, we developed a Chromium patch which enables fine-grained control on which HTTP versions to use with each domain involved in a webpage load.

Third, we leverage the above tools to study the HTTP and TCP interplay in the wild, *i.e.,* 300 popular webpages and mobile-like network conditions. With respect to the prevalence of *delay-based* traffic, we show that over 50% of websites have more than 75% bytes served via connections using delay-based CC. With respect to web performance, we show that over 67% of websites relying solely on delay-based CC have better performance with H1. Then, in a series of deep-dive case studies, we deconstruct page loads to uncover how exactly this behavior plays out for real websites. We verify this behavior on several webpages, but also find that, despite the negative interplay between H2 and delay-based TCP variants, there are still cases where H2 outperforms H1 due to the overheads introduced by Head of Line (HOL) blocking and setting up multiple TLS connections.

2 EXPERIMENT SETUP/ METHODOLOGY

In this section, we describe the testbed used in in §3 and §5.

Client Module. We use the (Chromium-based) Brave browser [1] to load webpages because of its built-in third-party tracker and ad blocking. This reduces the non-determinism of page loading by excluding traffic that is highly dependent on a user's ad-matching profile and allows us to focus on the primary content of the tested web pages. We leverage Lighthouse [2] for HTTP data collection and Chrome DevTools [3] to control the browser and log page load information. We pair the data gleaned from Lighthouse with packet traces captured by tcpdump [4] and parsed with tshark [5]. To decrypt HTTPS traffic, we instrument the browser to record used SSL session keys. With respect to DNS, we preface each experiment with a *primer* which, among other things that we will discuss later, caches DNS resolutions in `/etc/hosts` to guarantee consistent DNS resolutions across comparative experiments involving the same website.

Network Module. To provide a configurable bridge between the client module and the Internet, we build a generic module that supports network emulation as well as multiple access networks. We consider 4 network configurations:

`fiber`: low-latency and high-bandwidth fixed access provided by a North American fiber link; average bandwidth of 80 Mbps, in both directions, and latency of 5 ms as per `fast.com`.

`continuous-slowdown`: a synthetic network setting in which we gradually increase network latency (atop the low-latency `fiber` connection) by 15 ms every 100 ms up to a maximum of 300 ms. Note that this is not intended to reflect a realistic network condition, but was instead designed to trigger delay-based congestion control mechanisms for our analysis.

`jitter`: a realistic network setting encountered mostly in mobile networks where the network delay has high variability, e.g., due to poor/variable signal conditions. Jitter is defined by the pair <mean, stdev> and generated using dummynet [44] queues by repeatedly changing the queue's delay to a randomized one with target jitter values (using a Box-Muller transform to generate normally-distributed values from Linux standard `$RANDOM` following uniform distribution). This method of generating jitter avoids packet reordering, differently for example from TC's method of adding per-packet delays [6]; thus it better reflects latency fluctuations in cellular networks.

`tethering`: a real network setting involving a tethered mobile connection between an Android device and the client module (running on a modern Mac). The mobile network is from Mint Mobile [7]), a North American virtual mobile operator running atop of T-Mobile.

Server Module. To gain end-to-end traffic visibility for our synthetic experiments, we host synthetic pages on a server located in a university campus with, on average, 400 Mbps upload bandwidth. This content is served by an Nginx server [8] (v1.19) with H1 and H2 configured with TLS and GZIP compression. We instrument the server with a tool based on `ss` [9] to augment client-side logs with fine-grained server-side TCP information, e.g., congestion control algorithm, congestion window over time (`cwnd`), round trip time (`RTT`), and packet loss data. We start with the default TCP configuration in the Linux kernel 4.15 (i.e., TCP CUBIC with Hybrid Slow Start which we denote as `hystart`) and an initial `cwnd` of 10 packets. We then investigate other common configurations observed in the wild: CUBIC with regular slow start (`CUBIC`), `BBR`, `YeAH`, and `Illinois`.

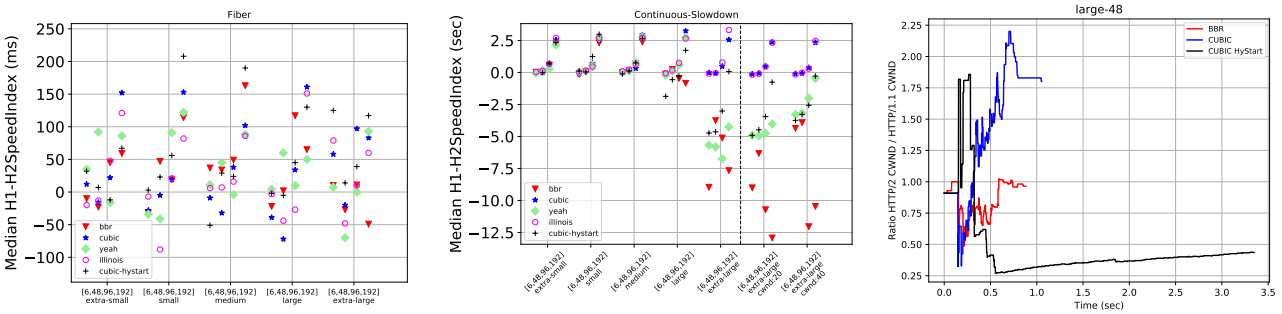
3 HTTP & TCP INTERPLAY

When thinking about web performance, the interaction between HTTP and TCP translates into how *effectively* each protocol combination can move packets through a network. To illustrate this, we look at a hypothetical scenario.

Assuming the default configuration of the Linux kernel (TCP CUBIC with an initial `cwnd` of 10 packets) and a maximum transmission unit (MTU) of 1.5 KB, we can derive that 15 KB of data could be sent in the first round per connection. Therefore, during the first round, H1 can transmit up to 90 KB of data per domain (spread across 6 connections) while H2 can send only 15 KB of data per domain (using 1 connection). With this in mind, we can consider the effect of different object sizes.

Consider a small web object, say 1 KB. H1 is subject to head-of-line blocking (HOL) and can send at most one object per connection at a time. Thus, in the first round, H1 can send at most six 1 KB objects. H2 is not subject to this and can send as many objects to fill up the current congestion window (via multiplexing)—in this case, 15 1 KB objects. Therefore, H2 improves startup throughput by 2.5×.

Now consider a slightly larger object, say 15 KB. In this situation, H1 is able to send six 15 KB objects while H2 can send only one 15 KB object. This results in H1 improving startup throughput by 6×. Here, the benefits of H1 are most pronounced during the *slow start* phase of congestion control, where the TCP CC algorithm probes the network to estimate the available congestion window.



(a) `fiber` network connection (b) `continuous-slowdown` network condition (c) `cwnd` ratio between H2 and H1 for `large-48` in `continuous-slowdown`

Figure 1: Difference in median SpeedIndex across 10 runs. Values > 0 mean H2 was faster than H1, and vice-versa.

The longer the available congestion window is underestimated, the more benefit H1 can provide. Moreover, a majority of web flows are often short-lived (page loads) and therefore behavior during slow start dramatically affects page load performance.

This observation becomes even more important when we consider recent studies that show that delay-based congestion control algorithms fail to accurately estimate the actual congestion window in mobile, jittery network conditions [15, 16]. We speculate that when delay-based congestion control algorithms are combined with mobile networks, H1 can outperform H2. We recognize that this does not *always* result in better H1 performance. The overhead introduced by establishing TLS for each of H1's multiple connections along with any HOL blocking that occurs can still exceed the benefits H1 achieves for delay-based variants.

3.1 Actualizing the Interplay

We perform controlled experiments to measure web performance across a test matrix with different synthetic pages and protocol configurations: HTTP version (H1, H2), TCP CC algorithm (`CUBIC`, `CUBIC-hystart`, `BBR`, `Illinois`, `YeAH`), initial congestion window (`cwnd`: default 10, up to 40 in some tests), number of objects (6, 48, 96, 192), and page size (40 KB, 140 KB, 550 KB, 2.3 MB, 9 MB). The synthetic pages we generate extend on the H2 demos (e.g., Akamai's [10] and Cloudflare/gophertiles's [11]) enabling fine-grained control on the number of objects and their sizes. We report the difference in median SpeedIndex [30], *i.e.,* the average time at which visible parts of the page are displayed, across 10 runs. We observe that variation across runs was a result of random losses or aberrant Lighthouse behavior and that using the median filtered out this noise.

We first run this experiment on the `fiber` network setting (Fig. 1(a)). We make few observations: 1) for pages smaller than 1 MB, as the number of objects grows, H2 does better, 2) H1 is often slightly faster with fewer and bigger objects, regardless of the TCP CC algorithm — however, the relative speedup of either HTTP version is barely noticeable (note the y-axis is in ms).

With a baseline established, we now focus on the `continuous-slowdown` network setting (Fig. 1(b)). Here, the goal is to expose the unique behavior of delay-based CC algorithms. First, we see that H2 tends to outperform H1 as the number of objects increases. This is well understood [36] and can be attributed to H2 multiplexing reducing HOL blocking. Furthermore, this is the reason for both protocols performing similarly with just 6 objects before they get too large: HOL blocking has little impact on H1 because the browser opens 6 connections simultaneously.

As webpages get bigger (large: 2.3 MB and extra-large: 9 MB), H1 tends to outperform H2 in presence of delay-based CC algorithms. This is because more time is needed to download larger pages, thus increasing the chance of the delay-based portion of the CC algorithm to be triggered and rate limit the transmission. While this impacts both H1 and H2, the latter ends up suffering more due to the single TCP connection. Furthermore, the right side of the dashed line in the figure shows the impact of a larger initial *cwnd* (20 and 40) at the sender. The rationale of these experiments is that by increasing such window, it decreases the amount of time spent in the delay-based portion of the CC algorithm. The figure indeed shows improved SpeedIndex (lower negative delta) for H2 when focusing on both `YeAH` and `CUBIC-hystart`, but H1's performance edge is confirmed. No improvement or specific trend is observed for `BBR`. Note that SpeedIndex gets quite noisy at high values (e.g., tens of seconds for H2 in this scenario) which is the main reason of the differences between SpeedIndex values measured at different `cwnd` and number of objects values. Although not shown due to space limitations, more stable results across these parameters are observed when focusing on *OnLoad*, *i.e.,* the time when the browser considers the page as fully loaded.

Fig. 1(c) visualizes the impact of delay-based CC algorithms on the `cwnd` using the server-side view for the `large-48` example. With `hystart`, H2 `cwnd` quickly drops to 0.25 of the total `cwnd` across H1 connections, resulting in slower transfer. On the other hand, the single H2 `CUBIC` connection starts slow, but quickly exceeds the total H1 congestion window. We note that SI indicates an earlier time in the page load as compared to when network traffic stops, e.g., with `CUBIC` median SI was 655 ms and 719 ms for H2 and H1 respectively. Taken together, these results illustrate the tension between HOL blocking affecting H1 and lower bandwidth utilization by delay-based TCP versions affecting H2.

Last but not least, we verified the existence of this effect via `tethering` over 12 different physical locations. We only focus on the `large-48` example, and `hystart` to bound the experiment duration. For each location, we load the synthetic website 3 times using both H1 and H2, for a total of 72 runs. In addition to *SpeedIndex*, we also report *OnLoad*, and *FirstMeaningfulPaint*, or the time it takes for a page's primary content to appear on the screen.

Fig. 2(b) shows, for each web performance metric, the CDF of the delta between H1 and H2. The figure shows that H1 outperforms

H2 for about $75-80\%$ of the runs if we consider *SpeedIndex* and *OnLoad*, respectively. This implies that, most of the time, the network conditions were such to trigger `hystart` and leave some bandwidth unused for H2. When focusing on *FirstMeaningfulPaint*, we observe a more split result. This happens for two reasons: 1) `hystart` requires several RTTs to manifest its behavior, and 2) H1 pays an extra cost to set up multiple TLS connections.

3.2 Wait, What About HTTP/3?

While this paper considers H1 and H2 (as they represent 92.5% of the current web traffic [13]), HTTP/3 (H3) is actively under development. Because of its infancy, both protocol and implementation wise, we have not conducted experiments with H3. However, we hypothesize that the adverse behavior we have observed with H2 persists with H3. H3 uses the UDP-based QUIC as its transport protocol, offering benefits of stream-multiplexing, low-latency connection establishment, and improved security, amongst others [33]. Although H3 is composed of multiple QUIC streams per domain instead of multiple TCP connections per domain, there remains a *single logical connection* per domain. Thus, faced with an underestimated congestion window, H3 will likely suffer similarly to H2 if coupled with delay-based CC algorithms.

4 EVOLVING THE WEB PERF. TOOLBOX

The previous section highlights that server-side network stack configurations play an important role on HTTP performance. A more subtle outcome is that the multi-domain nature of real webpages implies the potential for diverse TCP CC algorithms to be used concurrently during a web page load, complicating HTTP performance analysis. Existing measurement tools for web performance fall short in offering the fine-grained *view* and *control* of such information. We address this limitation in this section.

4.1 Extra Visibility with Igor

Relying on server access to perform web measurements and protocol development is impractical. Gordon [38] is, to the best of our knowledge, the only modern tool for fingerprinting TCP CC in the wild which is also open source. We have extensively tested Gordon, but we had to dismiss it for two reasons: 1) low accuracy in presence of small objects (Gordon requires objects > 160KB), and 2) it does not identify `hystart` since it leverages post-loss TCP behavior for detection (`hystart`'s behavior only manifests pre-loss).

To fill this gap we built Igor, a tool which extends Gordon, focusing on *pre-loss* TCP behavior. Rather than detecting *which* TCP version is running at the server, it differentiates between *delay-based* and *loss-based* algorithms, since our synthetic results indicate that this property has the largest impact on HTTP performance. This question can be answered by focusing on slow-start only, and solves the limitations of Gordon since slow-start captures `hystart` and can operate on smaller objects.

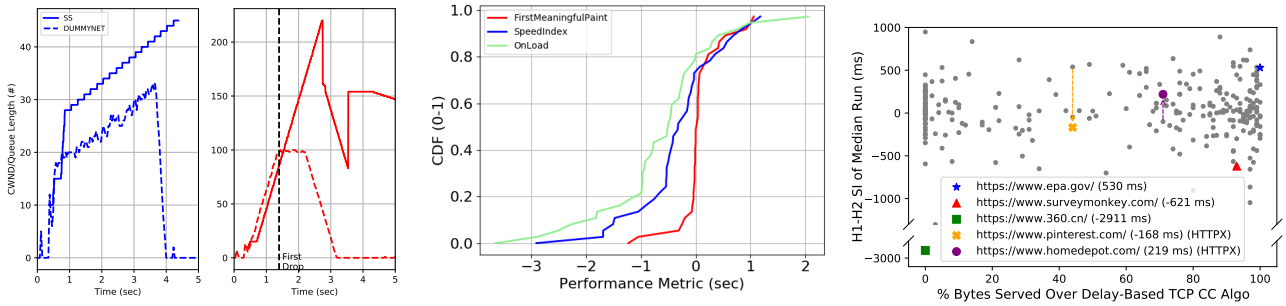
At a high level, Igor works as follows. Given a target website W, it first loads W using the client module from our testbed (§2), and derives the N contacted domains along with their IP addresses, percentage of traffic contributed to the webpage, and *biggest* object served. Next, it proceeds with testing slow start for each of the N biggest objects identified. This entails retrieving each object via an emulated bottleneck associated with a *large queue*, causing ACKs to be progressively delayed, thereby triggering *delay-based* algorithms. We use dummynet [44] to introduce a pipe between a *curl* client and server. The pipe uses (< bandwidth, latency, queue-length >) to force the queue to start filling up quickly, *i.e.,* within the first RTT. At this point, if the server runs a *delay-based* version of TCP, it will quickly slow down, e.g., within N RTTs depending on each algorithm, avoiding queue buildups that would result in packet drops. In contrast, loss-based TCP algorithms would result in the opposite behavior, enabling straightforward differentiation between the two.

To demonstrate the merit of Igor's approach, Fig. 2(a) shows an example when fetching a 160KB file using a 500B Maximum Transmission Unit (MTU) from a server we control. At the server, we run `CUBIC` with `hystart` on (left) and off (right). The figure shows both the output of the dummynet queue sampling (50 ms frequency, max queue length of 100 packets, dashed line) along with the ground truth `cwnd` collected at the server (4 ms frequency, via `ss` [9], solid line). Between 0 and 0.5 seconds, the two algorithms behave the same, *i.e.,* regular slow start quickly building up a `cwnd` of about 20-30 packets. At this point, packets are piling up in the queue (see dashed lines, 20% of the 100-packets queue is already occupied) causing ACKs to be delayed. With `hystart` (left plot), the TCP congestion control *slows down* while regular slow start (right plot) keeps increasing its window (doubling across an increasing RTT, peaking at 1 second) until the queue is full (100 packets) and a packet is dropped; the delay between when the first packet is dropped from the queue (t=1.5sec) and when the first loss is recorded (t=2.5sec) is due to the time needed for three consecutive duplicated ACKs.

Next, we benchmark Igor with respect to the most utilized TCP versions in the wild (according to [38]): `BBR`, `CUBIC`, `YeAH`, and `Illinois`, treating CUBIC with HyStart as a separate variant. Focusing on multiple MTU settings and object sizes, we analyze how the different mechanisms fill up the queue and make key observations (Table 1). First, delay-based algorithms (with the exception of `YeAH` which we will discuss below) rarely occupy more than 30% of the dummynet queue, *i.e.,* only with 320KB object and `hystart` Conversely, loss based algorithms tend to fill the queue quickly (on average in about

Table 1: Igor's queue occupation over time for variable object sizes [40,80,160,320]KB, MTU values [500,1500]B, and TCP versions (BBR, CUBIC, YeAH, Illinois, CUBIC+HyStart).

Protocol	Object Size							
	40KB		80KB		160KB		320KB	
	Max Queue	Time To Max	Max Queue	Time To Max	Max Queue	Time To Max	Max Queue	Time To Max
500B MTU								
BBR	20	0.3	20	0.7	20	0.7	30	0.8
CUBIC	40	0.8	90	1.3	100	1.4	100	1.5
YeAH	40	0.8	80	1.2	90	1.3	90	1.3
Illinois	40	0.8	80	1.2	100	1.5	100	1.4
HyStart	20	0.8	20	0.8	30	3	80	6.4
1,500B MTU								
BBR	10	0.6	20	0.6	20	0.6	20	0.6
CUBIC			20	0.6	50	1	80	1.1
YeAH	10	0.5	20	0.6	50	0.8	80	1.2
Illinois			20	0.6	50	0.9	80	1.2
HyStart	10	0.5	20	0.6	40	0.8	40	0.8



(a) (b) (c)

Figure 2: (a) Ground truth cwnd (SS, solid) vs Dummynet queue sampling (Dummynet, dashed). Hystart (Left) vs Slow-start (Right). MTU=500B, Object=160KB; (b) CDF of FMP/SI/PLT for (large, 48, cubic-hystart) measured from 12 diff. physical locations on mobile; (c) Bytes Served Over Delay-Based TCP CC Algo for Tranco Top 300 Websites

1.5 seconds) before queue overflow causes packet losses leading to slowdown. Second, a default MTU (1,500B) makes the protocols hard to distinguish and should be avoided for this purpose. Small MTU, on the other hand, increase the duration of a test. Finally, `hystart` ends up with many packets in the queue given sufficiently large objects and enough time to grow cwnd after the slow-start phase.

Based on the observations, Igor uses a simple heuristic to distinguish between loss-based and delay-based algorithms. If a loss is detected within the first two seconds, the algorithm is labeled as *loss-based*; the opposite behavior implies that it is *delay-based*. We further introduce the label *too-small-to-judge* in presence of objects smaller than 50KB, to be conservative. By default, Igor uses a 500 MTU which it lowers to 100 for objects smaller than 100KB and it increases to 1,500 for objects bigger than 1MB. Note that in this test, the *physical* RTT was about 30 ms (padded to 100ms as explained below) and results can change in the presence of different RTTs. However, given that queue-based delays dominate, e.g., reaching up to 1 second at full queue capacity, we observe minimal RTT-induced impact. Still, we normalize the RTT to the next 50 with a minimum of 100ms, e.g., padding a measured 113ms to 150ms.

Compared to Gordon, Igor allows to detect `hystart` and to distinguish loss-based versus delay-based CC algorithms in the presence of much smaller objects, down to 50KB from 160KB for Gordon. Further, while running Gordon on real webpages, we measured a high fraction of *unknown*, equivalent of Igor's *too-small-to-judge*: 82% for objects smaller than the recommended 160KB, but still 28% for objects between 160 and 300KB, as well as bigger than 300KB.

Last but not least, we confirm that `YeAH` behaves *mostly* as a loss-based algorithm and it thus would be mostly mislabeled by our heuristic. This is because `YeAH` uses changes in packet delay to estimate packet queue size, however until the estimated queue size reaches 80 it uses STCP [34] rule to aggressively increase congestion window size (fast phase). Although the queue size is a configurable parameter of the algorithm, the current kernel implementationdoes not allow for its tuning. Due to dummynet's maximum queue size of 100, Igor is not able to robustly distinguish between `YeAH` and the loss-based STCP its fast phase is based on. This can be improved by recompiling dummynet, and the kernel module for the kernel-level packet handling, but a bigger queue would impact our overall detection heuristic and also increase the minimum packet size supported by Igor. We thus opted to accept potential mislabeling for `YeAH` given its limited usage in the wild (about 5.5% according to [38]).

4.2 Fine-Grained HTTP Control

Chromium-based browsers (like Brave) can be run with H2 disabled using the `-disable-http2` flag. This flag disables H2 *completely*, *i.e.,* for both direct traffic to a website and traffic to external websites embedded on it. This is achieved by not announcing H2 support via the Application Layer Protocol Negotiation (ALPN) [27] TLS extension, an externally visible marker for the application-layer protocol associated with the TLS connection.

The above flag only allows a coarse comparison between HTTP protocols. We extended the `net` Chromium library [29] – which Brave relies upon – to only disable H2 support for a specific set of domains. We implemented this functionality as patch of the Chromium's code, thus applicable to all Chromium-based browsers. The modified browser takes a configuration flag `-disable-http2-per-url` which accepts as input a list of comma separated domains for which to disable H2.

5 EMPLOYING THE TOOL BOX

In this section, we experiment with the new features we have built to evolve the web performance toolbox. We integrate Igor in our client module (see §3), and specifically with the (DNS) *primer*. For each discovered domain during a webpage load, the primer identifies the biggest object and uses Igor to identify whether its server uses a loss-based or delay-based TCP CC. Next, we use this information to control which protocol (H1 or H2) to be used per domain. We test the top 300 websites in the Tranco list [35] focusing on the `jitter` scenario, which is representative of realistic mobile networks but may challenge algorithms sensitive to delay variations. For the jitter, we use 50ms as mean, and a stdev of 70ms which is common in North American cellular networks [53]. Bandwidth is capped to 40Mbps in both directions, with no additional packet loss.

Fig. 2(c) shows the percentage of bytes served over a delay-based TCP CC versus the Speed Index delta of the median of 3 runs, for Tranco top 300 sites [12]. Each dot refers to (H1 – H2), but we also show two examples of *HTTPX*, *i.e.,* a mix where we only turn off H2 for delay-based domains. Regarding the prevalence of delay-based traffic, the figure shows two interesting clusters at 0% (fully

loss-based) and around 90−100% (almost full delay-based). These two clusters cover 20% and 34% of the 300 sites, with the remainder having a more complex TCP mix, uniformly split from 10−90%.

With respect to performance, the analysis is more complex. Overall, for loss-based algorithms H2 outperforms H1 65% of the time. This result finds confirmation in previous works [36], which likely mostly focus on loss-based TCP CC. However, there are cases where H1 outperforms H2 significantly (■) which require a more careful investigation. As the prevalence of delay-based traffic increases, the result is more mixed. While no crystal clear trend arises, we can see that H1 becomes more competitive as the fraction of traffic served via delay-based CC increases. The plot also highlights two scenarios where the usage of HTTPX can both *improve* and *reduce* performance. Finally, the last cluster is where we expect H1 to shine the most, and indeed it does. In fact, when 100% of bytes are served by a delay-based variant, H1 performs better 67% of the time. However, the web is complex and we can still find many examples where H2 is significantly faster than H1 despite these adversarial conditions. We discuss one very interesting example (★) in the following subsection.

5.1 Case Studies

In addition to the extra features we have built, we also argue that web/http measurements largely benefit from client-side TCP traces. Our client module exports a browser SSL session keys to decrypt pcap traces, and then builds a mapping between devtools and TCP data, e.g., bytes received, RTT, and losses. The tool further visualizes the time evolution of the dependency graph, and it has proven quite useful in isolating specific behaviors, some of which we report below.

https://www.epa.gov/ For this site, SI is triggered at 1,905 ms on H1 and 1,375 ms on H2 (★ in Fig. 2(c)). All 75 resources loaded are served by *www.epa.gov* which Igor has classified as delay-based. This example demonstrates that although every byte was served through a delay-based TCP CC algorithm, H1 is not always favorable (in fact, here H2 outperforms H1 by 530 ms). Through a closer look at how the resources were made known, requested, and loaded over time, we can see that H1 was heavily afflicted by HOL blocking. Therefore, any benefits provided by H1's multiple connections were surpassed by the delay induced by HOL blocking.

https://www.surveymonkey.com/ SI triggered at 1,866 ms on H1 and 2,487 ms on H2 (▲ in Fig. 2(c)). Here, over 90% of bytes from *https://www.surveymonkey.com/* were classified by Igor as a delay-based variant. After the index page is loaded, 4 resources are first requested (100K, 115KB, 6KB, 322KB). In this example, those 4 (mostly large) resources are done loading by 769 ms in H1 and 1,043 ms in H2. This is a classic example of H2 underutilizing available bandwidth as a result of an underestimation of network congestion by a delay-based TCP CC algorithm.

https://www.360.cn/ Here, SI triggered at 6,167 ms on H1 and 9,078 ms on H2 (■ in Fig. 2(c)). For this website, 0% percent of bytes are served by a delay-based variant. Despite this, H1 surpasses H2 by over 300 ms. We took a closer look and using the RTT measurements and the number of losses recorded during our experiments, we discovered that the average RTT to this website was 124 ms and on average, 7 losses were detected. Considering that this website was based in a geographically far location (the *.cn* TLD corresponds to China), it is very reasonable to expect high RTTs and a lossy environment. As a result of these network conditions, H1's multiple connections improve robustness against losses and load resources much faster.

6 RELATED WORK

Web Performance Studies The development of HTTP and TCP over the last 3 decades has given way to a number of performance studies on HTTP [21, 24, 28, 41, 50–52] and TCP [18, 25, 26, 37, 43, 46, 48, 49]. For example, Wang et al. study the impact of SPDY on web performance, concluding that SPDY does not definitively outperform H1 [50]. Butkiewicz et al. demonstrate how one can dynamically reprioritize web content to improve overall user experience [21]. In doing so, they develop a model to estimate the load time of a web page. Flach et al. analyze TCP connections from clients to Google services, investigating the impact of losses and then propose and evaluate faster loss recover methods [26]. Prior work has also demonstrated the strong interplay between HTTP and TCP [22, 32, 40]. For example, Cao et al. build an analytical model based on the TCP congestion control algorithm to estimate TCP throughput [22]. They also explore prediction on H1 vs. H2 in a limited synthetic setting. Naseer et al. introduce Configtron, a tool to optimize network stack configurations on a server [40]. While we build on these prior findings, the primary focus of our paper is on understanding the performance consequences of the parallel (and disconnected) evolution of TCP and HTTP protocols, with a focus on recent delay-based CC algorithms. This interplay has not been studied in detail by the aforementioned prior studies, and is of increasing importance given the prevalence of scenarios in which these protocols operate concurrently.

Inferring Server-Side TCP Settings A number of past studies developed various methods to infer congestion control mechanisms from just the client [38, 39, 42, 45, 47]. However, these mechanisms either did not realize into usable tools or failed to differentiate between delay-based variants that arise during TCP slow start (§4.1). We found the latter paramount in today's web where many domains only serve quite small objects, for which TCP slow start is the only behavior we can study. These observations have motivated our development of Igor which we have also open sourced [14].

7 CONCLUSION

This paper highlights how the divergence in network protocol evolution across the stack—the transition to a single multiplexed connection in HTTP and a shift from loss-based to delay-based congestion control algorithms in TCP—has adverse effects on web page load times. We demonstrate this behavior with in-lab experiments, which motivate the deployment of a new web performance toolbox characterized by fine-grained visibility and control on the HTTP/TCP interplay. By testing 300 popular webpages, we show that 67% of websites relying solely on delay-based congestion control have better performance with HTTP/1.1 than HTTP/2, nowadays the de facto standard protocol adopted in the web. These results highlight the importance of co-design between cross-stack network protocols.

Acknowledgements. We thank Pradeep Dogga and the anonymous CoNEXT reviewers for their valuable feedback. This work was partially supported by NSF grant CNS-1943621.

REFERENCES

[1] https://brave.com/.
[2] https://developers.google.com/web/tools/lighthouse.
[3] https://developers.google.com/web/tools/chrome-devtools/.
[4] https://www.tcpdump.org/2.
[5] https://www.wireshark.org/docs/man-pages/tshark.html.
[6] https://man7.org/linux/man-pages/man8/tc.8.html.
[7] https://www.mintmobile.com/.
[8] https://www.nginx.com/.
[9] https://linux.die.net/man/8/ss.
[10] https://http2.akamai.com/demo.
[11] https://http2.golang.org/gophertiles.
[12] https://tranco-list.eu/list/6W9X.
[13] Usage statistics of HTTP/3 for websites. https://w3techs.com/technologies/details/ce-http3.
[14] Web performance toolbox. https://github.com/svarvel/web-perf-toolbox, 2020. Accessed on 10.23.2020.
[15] E. Atxutegi, Å. Arvidsson, F. Liberal, K. J. Grinnemo, and A. Brunstrom. TCP performance over current cellular access: A comprehensive analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, LNCS 10768:371–400, 2018.
[16] E. Atxutegi, F. Liberal, K.-J. Grinnemo, A. Brunstrom, Å. Arvidsson, and R. Robert. Tcp behaviour in lte: impact of flow start-up and mobility. In *2016 9th IFIP Wireless and Mobile Networking Conference (WMNC)*, pages 73–80. IEEE, 2016.
[17] A. Baiocchi, A. P. Castellani, and F. Vacirca. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.
[18] W. Bao, V. W. Wong, and V. C. Leung. A model for steady state throughput of tcp cubic. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–6. IEEE, 2010.
[19] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, RFC Editor, May 2015.
[20] M. Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-29, Internet Engineering Task Force, June 2020. Work in Progress.
[21] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328, 2011.
[22] Y. Cao, J. Nejati, A. Balasubramanian, and A. Gandhi. Econ: Modeling the network to improve application performance. In *Proceedings of the Internet Measurement Conference*, pages 365–378, 2019.
[23] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.
[24] H. De Saxce, I. Oprescu, and Y. Chen. Is HTTP/2 really faster than HTTP/1.1? *Proceedings - IEEE INFOCOM*, 2015-Augus:293–299, 2015.
[25] R. Dunaytsev, Y. Koucheryavy, and J. Harju. Tcp newreno throughput in the presence of correlated losses: The slow-but-steady variant. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–6. IEEE, 2006.
[26] 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 *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 159–170, 2013.
[27] S. Friedl, A. Popov, A. Langley, and E. Stephan. Transport layer security (tls), application-layer protocol negotiation extension, 2014.
[28] U. Goel, M. Steiner, M. P. Wittie, S. Ludin, and M. Flack. Domain-Sharding for Faster HTTP/2 in Lossy Cellular Networks. 2017.
[29] Google. net - chromium/src.git. Accessed on 01.23.2020.
[30] Google. Speed index. https://developers.google.com/web/tools/lighthouse/audits/speed-index, 2020. Accessed on 03.22.2020.
[31] S. Ha and I. Rhee. Taming the elephants: New tcp slow start. *Computer Networks*, 55(9):2092–2110, 2011.
[32] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of http over several transport protocols. *IEEE/ACM transactions on networking*, 5(5):616–630, 1997.
[33] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-31, Internet Engineering Task Force, Sept. 2020. Work in Progress.
[34] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.
[35] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Feb. 2019.
[36] Y. Liu, Y. Ma, X. Liu, and G. Huang. Can HTTP/2 really help web performance on smartphones? *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, pages 219–226, 2016.
[37] S. H. Low, L. L. Peterson, and L. Wang. Understanding tcp vegas: a duality model. *Journal of the ACM (JACM)*, 49(2):207–235, 2002.
[38] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong. The Great Internet TCP Congestion Control Census. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–24, 2019.
[39] U. Naseer and T. Benson. Inspectorgadget: Inferring network protocol configuration for web services. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1624–1629. IEEE, 2018.
[40] U. Naseer and T. Benson. Configtron: Tackling Network Diversity with Heterogeneous Configurations. *arXiv preprint*, pages 1–26, 2019.
[41] J. Padhye and H. F. Nielsen. A comparison of spdy and http performance. 2012.
[42] J. Pahdye and S. Floyd. On inferring TCP behavior. *Computer Communication Review*, 31(4):287–298, 2001.
[43] N. Parvez, A. Mahanti, and C. Williamson. An analytic throughput model for tcp newreno. *IEEE/ACM Transactions on Networking*, 18(2):448–461, 2009.
[44] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, Jan. 1997.
[45] J. Rüth, C. Bormann, and O. Hohlfeld. Large-scale scanning of TCP's initial window. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, Part F1319:304–310, 2017.
[46] C. Samios and M. K. Vernon. Modeling the throughput of tcp vegas. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):71–81, 2003.
[47] C. Sander, J. Rüth, O. Hohlfeld, and K. Wehrle. Deepcci: Deep learning-based passive congestion control identification. *NetAI 2019 - Proceedings of the 2019 ACM SIGCOMM Workshop on Network Meets AI and ML, Part of SIGCOMM 2019*, pages 37–43, 2019.
[48] B. Sikdar, S. Kalyanaraman, and K. S. Vastola. Analytic models for the latency and steady-state throughput of tcp tahoe, reno, and sack. *IEEE/ACM Transactions On Networking*, 11(6):959–971, 2003.
[49] J. Wang, D. X. Wei, and S. H. Low. Modelling and stability of fast tcp. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 938–948. IEEE, 2005.
[50] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, pages 387–399, 2014.
[51] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, page 91–96, New York, NY, USA, 2011. Association for Computing Machinery.
[52] K. Zarifis, M. Holland, M. Jain, E. Katz-Bassett, and R. Govindan. Modeling HTTP/2 speed from HTTP/1 traces. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9631:233–247, 2016.
[53] S. Zhang, W. Li, D. Wu, B. Jin, D. Gao, Y. Wang, R. K. Chang, and R. K. Mok. An empirical study of mobile network behavior and application performance in the wild. *Proceedings of the International Symposium on Quality of Service, IWQoS 2019*, 2019.