Designing a Better Browser for Tor with BLAST

Tao Wang
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
taow@cse.ust.hk

***Abstract*—Tor is an anonymity network that allows clients to browse web pages privately, but loading web pages with Tor is slow. To analyze how the browser loads web pages, we examine their resource trees using our new browser logging and simulation tool, BLAST. We find that the time it takes to load a web page with Tor is almost entirely determined by the number of round trips incurred, not its bandwidth, and Tor Browser incurs unnecessary round trips. Resources sit in the browser queue excessively waiting for the TCP and TLS handshakes, each of which takes a separate round trip. We show that increasing resource loading capacity with larger pipelines and even HTTP/2 do not decrease load time because they do not save round trips.**

We set out to minimize round trips with a number of protocol and browser improvements, including TCP Fast Open, optimistic data and 0-RTT TLS. We also recommend the use of databases to assist the client with redirection, identifying HTTP/2 servers, and prefetching. All of these features are designed to cut down on the number of round trips incurred in loading web pages. To evaluate these proposed improvements, we create a simulation tool and validate that it is highly accurate in predicting mean page load times. We use the simulator to analyze these features and it predicts that they will decrease the mean page load time by 61% over HTTP/2. Our large improvement to user experience comes at trivial cost to the Tor network.

I. Introduction

The Snowden revelations showed us the massive scale and breadth of state-level surveillance against internet activity. To avoid privacy compromise, web-browsing clients may choose to use anonymity networks. Anonymity networks relay user traffic through multiple nodes across the globe, ensuring that a single eavesdropper cannot know both the true origin and destination of any traffic.

Tor [7] is a highly successful anonymity network with millions of daily users. Its success can be partly attributed to the easy-to-use Tor Browser, which is based on Firefox. One of its chief downsides — and a barrier to further adoption — is that web browsing using Tor Browser is notably slow. From our data, we find that it takes a mean of 16 to 19 seconds to load web pages over Tor Browser depending on version.

In this work, we seek to reduce load times on Tor through improving its browser design to improve user experience. Fabian et al. [9] studied browser load times; their results suggest that a mean of 16 to 19 seconds would have corresponded to high loading cancellation rates and low user satisfaction if they were experienced on a normal browser. Faster load times reduce user frustration and downtime. A better user experience would furthermore draw in more users who were previously unwilling to trade off utility for its better privacy. Having more users improves the anonymity of Tor Browser by creating larger anonymity sets, reducing the chance that eavesdroppers could deanonymize a Tor user using side information.

Anonymity network optimization is a well-studied privacy problem with a decade of research, generally focused on Tor [4], [5], [12], [14], [18]. Researchers have proposed various solutions to optimize Tor's performance on the network level so as to improve user experience. On the other hand, the problem of **browser design** for anonymity networks remains academically untouched, with many open problems that are just as significant for user experience as network design problems. Tor Browser should be designed with Tor's high latency in mind, but instead it is largely identical to Firefox with regards to how it loads web pages.

There are two chief barriers to research in browser design. First, any changes to browser design require a lengthy implementation, validation, and data collection process to measure results. Second, random network conditions can significantly alter results, so that one browser design may only be performing better than another because of better network conditions. An example of the inconsistency of browser design evaluation can be found in the history of HTTP Pipelining. Pipelining was implemented and standardized in Firefox and Chrome to reduce page load times, but quickly abandoned and disabled in both browsers as further analysis showed no significant improvement in performance (despite earlier claims).

We overcome both barriers by creating BLAST (Browser Logging, Analysis and Simulation for Tor), a tool capable of logging, analyzing and simulating page loading on Tor in minute detail. BLAST consists of a logger and a simulator. The logger is an extensive instrumentation of Tor Browser to analyze its page loading process; the simulator simulates page loads based on basic information about its structure. With BLAST, we make the following contributions:

1) Analyzing BLAST logs, we show that Tor Browser's high page load times are almost entirely due to incurring round trips with Tor's high latency, not because of limited bandwidth or resource loading capacity. In addition, we find that on Tor Browser, HTTP/2 does *not* load pages more quickly than HTTP/1.1 with pipelining despite its superior connection multiplexing.
2) We create a page loading simulator tool that is highly accurate at predicting mean page load times and can do so for a variety of browser set-ups, including HTTP/1.1, HTTP/1.1 with pipelining, and HTTP/2.
3) We propose a number of browser and protocol improvements to eliminate unnecessary round trips and, using the BLAST simulator, we verify their positive effect on reducing page load times. Our simulator predicts that our proposed features combined reduce mean page load time from 18.0 s to 7.1 s, a 61% decrease.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24199
www.ndss-symposium.org

We proceed as follows. In Section II, we go through the basics of web browsing and the implementation of BLAST. We analyze HTTP/1.1 with pipelining and examine its shortcomings for use in anonymity network browsing in Section III, focusing on the causes of long load times. We then analyze HTTP/2 and compare against HTTP/1.1 with pipelining in Section IV. Conclusions drawn from the comparison guide our design of a better browser for anonymity networks in Section V, where we also evaluate each of our design decisions separately with the help of our simulator. We discuss various browser design issues in Section VI, compare our work with related work on anonymity network design in Section VII, and conclude with Section VIII.

II. Tools and Terminology

A. Basics of Page Loading

To load a web page, the client's web browser dispatches **resource** requests onto TCP **connections**. If no connection is available, the request is instead appended onto a **resource queue** where it waits until a connection is available. We expand on these terms in the following.

Resources. A web page fully comprises of a set of web resources. Each resource is associated with a distinct GET or PUT request. For example, a resource could be an HTML document, a CSS sheet, JS code, PHP code, or an image. A resource load can be triggered by user activity (such as typing in a URL or clicking on a link), by another resource (such as an image that is referred to by an HTML document), or the browser application itself (such as an update for an add-on). We can represent the structure of each web page as a **resource tree** with each resource as a node. The root resource is the original resource requested by the user's activity, and a node's parent is whichever resource triggered its loading.

Connections. To obtain a resource, the browser establishes TCP connections and then *dispatches* resources onto available connections. The connection thus becomes active until the resource is fully loaded. After the resource is fully loaded, the connection becomes idle until the browser chooses to dispatch another resource onto the connection (often immediately). A resource cannot be dispatched onto multiple connections, while a single connection is likely to dispatch many resources. The browser can establish multiple connections to the same server, up to a limit, to dispatch more resources concurrently. Currently, Tor Browser will establish up to six simultaneous connections to the same server.[1]

Resource queue. When the browser determines that a resource is necessary, due to user action or a reference by a previously loaded resource, the browser adds its request to the resource queue, and attempts to find connections to dispatch all resources in the queue. A connection becoming established, idle, or closed also causes the browser to attempt to dispatch resources in the queue. If there are not enough connections to dispatch resources in the queue, the browser may establish a new connection, respecting limits on the total number of connections. Thus the queue acts as a callback system to ensure that resources can be sent as soon as conditions permit.

[1] The maximum number of simultaneous connections to different servers in total is 900, which will not be reached in normal browsing.

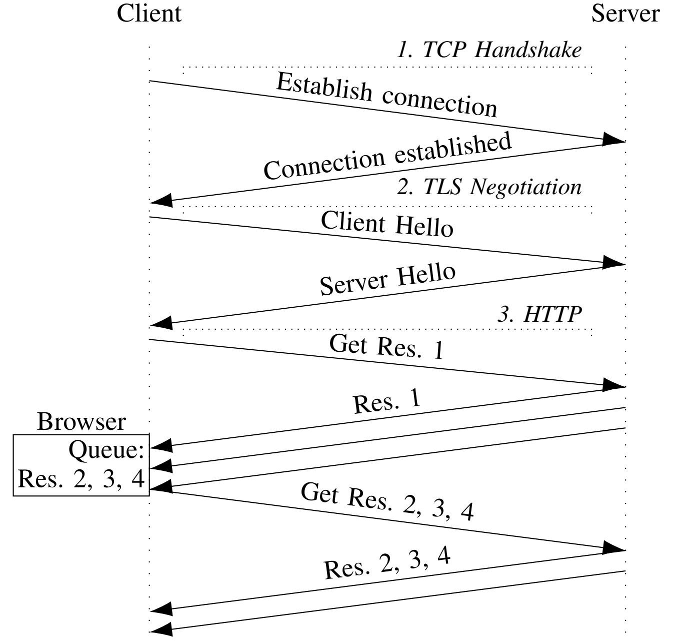

Fig. 1: Client-server diagram of how the browser loads a web page with four resources (Res. 1–4) from a server with HTTPS. The browser only establishes one connection in this case. Loading Res. 2–4 this way requires either pipelining or HTTP/2.

We illustrate these concepts with a client-server diagram in Figure 1 for a web page with four resources, all on the same server. The client types in an HTTPS URL corresponding to the first resource (Res. 1), whose children are Res. 2, 3, and 4. It takes three round trips to load the first resource. If the client parses references in Res. 1 to its children before Res. 1 is fully loaded, the client would establish another connection in an attempt to load the children (since the one shown is still actively loading Res. 1).

It is not possible to load Res. 2, 3 and 4 as in the diagram with original HTTP/1.1 because each connection can only load one resource at once; more connections would be constructed to load them, requiring more round trips. For one connection to dispatch more than one resource simultaneously, the client needs either HTTP/1.1 pipelining or HTTP/2. We highlight the differences between HTTP/1.1 without pipelining, HTTP/1.1 with pipelining, and HTTP/2 in Table I for easier comparison; we describe their mechanisms in detail in their respective sections (Section III and Section IV).

B. What causes long page load times on Tor?

Tor's high latency imposes a challenging condition on browser design. Here, we characterize the types of pages that would be especially difficult to load when the client suffers from high latency. Later, we will investigate whether or not Tor Browser deals with these situations effectively.

Large number of web servers. Loading a resource from a HTTPS server necessitates up to three round trips: TCP connection establishment, TLS negotiation, and HTTP resource request. A resource reusing a previously established

TABLE I: Summary of page loading restrictions under different protocols.

	HTTP/1.1 without pipelining	HTTP/1.1 with pipelining	HTTP/2
Max. number of connections per server	6	6	1
Max. number of simultaneous resources loaded per connection	1	6–14	100
Response must be in order of request	Yes	Yes	No

connection to the same server can avoid the first two round trips. Therefore, a web page that distributes its resources across more servers often requires more round trips to load, counter-intuitively causing longer load times.

Tall resource trees. The browser can only request a resource once its parent's data has been received and parsed to obtain its reference. Since it takes at least one round trip to request and receive a resource, the height of a resource tree is the minimum number of round trips required to load the page. In the worst case, if each resource between the parent and the deepest leaf of the tree is on a separate HTTPS server, we would need three round trips per height of the tree to load the web page (as above).

Excessive number of resources. The total number of resources that can be loaded concurrently is limited by the number of connections and the number of resources that can be sent on each connection; we refer to this as the **resource loading capacity**. On HTTP/1.1 without pipelining, the resource loading capacity is 6. Pipelining multiplies the resource loading capacity with the depth of pipelines (6 to 14). On HTTP/2, the resource loading capacity is 100; few servers have more than 100 resources. If resource loading capacity is too low, resources would have to wait excessively in the queue for connections to free up.

C. Categorizing load time

We use two metrics to capture user experience. First, we use *page load time*, which is how long it takes 95% of all resources to load. We do not require all resources to load when calculating page load time because many web pages have random advertisements or user tracking resources that are loaded long after all other resources have been completed. For example, a script may track user behavior and send it to the server via a resource request every ten seconds. If we counted those resource requests, the page load time would not be as meaningful.

Second, we use *resource load time*, the time difference between the browser requesting a resource and receiving that resource. This metric helps us determine why pages are taking a long time to load and how it can be resolved, as we break down each resource loading into five events:

(1) Resource request created.
(2) Resource dispatched onto an available connection.
(3) First byte sent from the client (request).
(4) First byte received from the server (response).
(5) Final byte received from the server (response).

For a resource r, the time gap between events (1) and (5) would be the *resource load time*. We refer to the gap between events (1) and (2) as the *queue wait time*, (2) and (3) as the *TLS wait time*[2], (3) and (4) as the *server wait time*, and (4) and (5) as the *transfer time*. Ideally, we would want server wait time to be one round-trip time, and other types of load times to approach zero.

We also categorize page load time using the resource tree. We sum up the categorized resource load time of all resources between the final resource (counting 95% of resources) and the root resource, following the path of parenthood in the resource tree. If the final byte of a resource was received after its child resource was created, then its contribution to page transfer time is accordingly cut. Any time gap between the final byte received and the child's resource request being created is classified as *non-categorized time*. Non-categorized time is usually a minuscule time gap due to the browser parsing the previous resource, although it can also be due to timed scripts.

III. HTTP/1.1 PIPELINING

In this section, we analyze HTTP/1.1 with pipelining. We investigate the usefulness of pipelining for Tor Browser and whether or not observed or suspected issues hamper its usefulness. We start with a description of how pipelining works in Section III-A, follow up with some potential issues in Section III-B, describe our data collection methodology in Section III-C, and analyze pipelining performance to determine the impact of these potential issues in Section III-D.

A. Pipelining in Tor Browser

Normally, resources can only be dispatched on newly established or idle connections that have finished receiving resources. HTTP Pipelining (or simply pipelining) allows resources to be dispatched on active connections without waiting for a response for previous resources. The server should respond to each resource request in order. Each pipeline has a maximum depth, the maximum number of resource requests that can be sent simultaneously on a connection. The browser's resource loading capacity for a server is therefore equal to the number of concurrent connections allowed (six) times the depth of each pipeline.

Pipelining has a difficult history with HTTP. Pipelining was briefly implemented and deployed in both Firefox and Chrome for HTTP/1.1, though it was quickly disabled in both due to a lack of noticeable load time improvement. This was not before pipelining became standard in HTTP/1.1, and as a result all HTTP servers should support pipelining (no extra negotiation is required to initiate pipelining). However, many servers still respond incorrectly to pipelining.

[2] This gap is the TLS wait time because resources do not wait for TLS completion before being dispatched onto a connection, but they cannot be written as network bytes before TLS negotiation is complete.

Partly as a response to website fingerprinting attacks, Tor developers enabled randomized pipelining on Tor Browser after it had been disabled on major browsers. This implementation also contains several other features, including randomized pipeline depth and randomized resource loading order. However, after a few years, Tor developers also disabled pipelining, as it had "become a maintenance burden" due to its large amount of difficult-to-understand code. Pipelining code was entirely removed from newer versions of both Firefox and Tor Browser in favor of using HTTP/2.

Despite its problems, pipelining does reduce load times on Tor Browser. Across our data set, the largest server for each web page has a mean of 34.4 resources. Without pipelining, only 6 resources can be loaded in one round trip, round trips being the prohibitive bottleneck of browsing times on an anonymity network. Pipelining is important for expanding the limited resource loading capacity of HTTP/1.1.

B. Issues with Pipelining

We discuss some issues with pipelining as a protocol in the following.

Head-of-line blocking. Head-of-line blocking refers to the restriction that the web server can only respond to each requested resource following the order in which the client requested them. Later resources can suffer delays waiting for the pipelined connection to load earlier resources, with larger resources causing greater delays.

Pipelining errors. Despite the standardization of pipelining support in HTTP/1.1, many servers still respond to pipelining incorrectly, leading to connection errors in the browser. These connections are discarded and the resource requests must be re-sent in another connection, causing delays. This is especially detrimental if a large number of resource requests were pending on that connection. The browser will then cease to use pipelining on that server for the given page load, losing the benefits of pipelining.

Choosing pipeline depth. While a larger pipeline depth increases the total resource loading capacity of the browser, it also exacerbates both head-of-line blocking and the potential delay caused by loading errors. On the other hand, a larger pipeline depth is helpful for loading servers with many small resources in parallel. Another way to increase resource loading capacity is to use more connections in parallel, but this causes issues with some routers and increases memory consumption for servers.

C. Data collection and methodology

To answer our questions about pipelining performance, we collect data on two versions of Tor Browser: Tor Browser 8.5 (TB-8.5), the latest version of Tor Browser at data collection, uses HTTP/2; Tor Browser 6.5 (TB-6.5), an earlier version, instead uses pipelining. To eliminate extraneous factors in our comparison between these protocols, we reintroduced pipelining code to TB-8.5, where it had been removed. The results in this section (to follow) are chiefly based on TB-8.5 with pipelining. Later, in Section IV, we compare TB-8.5 with TB-6.5.

We collected all data using a single computer connected to Tor. Tor consists of multiple nodes around the world and we disabled guard selection so that new entry nodes would be selected for every Tor circuit, allowing for greater coverage of the global Tor network. Tor connections serve as the bandwidth bottleneck so we did not attempt to limit our own bandwidth, and it is responsible for almost all of the round trip times. In experiments where we compared the performance of different browser designs, we visited the same page with each browser on the same circuit, and then we dropped the Tor circuit before moving on to the next browser design. Using the same circuit for the same instance allows us to compare page loads directly (as different circuits may access different versions of the page due to localization of the exit node), and using different circuits between different pages lets us capture a wider range of the Tor network.

We visited three sets of pages for each experiment: (1) Top 10 pages, 50 times each; (2) Top 200 pages, 5 times each; and (3) Top 1000 pages, 1 time each. We visit the home pages of Alexa's top 1000 pages as they were the most popular, to best represent likely user experience of the network. In total, we have 2500 page instances, although some instances did not load properly and were discarded (e.g. pages from Chinese servers were often inaccessible to Tor). These pages were visited from June 2019 to August 2019. We define a properly loaded page as a page where at least 2 resources were fully loaded. In comparative experiments, if an instance of the page was not loaded properly for any of the browser versions we were comparing, we also excluded that instance from the experiment for the other browser versions.

D. Data Analysis

We analyze the features and implementation of pipelining on Tor Browser. We begin with an analysis of the overall performance in loading times, and proceed by analyzing whether or not each of the above issues hampers the usefulness of pipelining.

1) Overall performance: Over our pipelining data set, pages had a mean of 104 resources and the mean page size was 2.1 MB, so the mean resource size was 20 kB, but the median resource size was only 3 kB: the majority of resources were very small. The resource tree had a mean height of 15.4. The top 10 sites were much smaller other sites, with a mean of only 55 resources compared to 116 for the top 200 and 118 for the next top 1000, and a mean page size of 1.3 MB compared to 2.3 MB and 2.2 MB respectively. The mean page load time over our whole pipelining data set was 16.4 s. This was 13.7 s for the top 10 sites, and 19.6 s for both the top 200 sites and the top 1000 sites.

2) Categorization of load time: For every page in our data set, we break down its load time into the four categories described in Section II-C, and show them in Figure 2. Roughly 5.8 s (36%) of page load time was due to resources waiting in queue for connections; the same amount was incurred again waiting for servers to respond to resource requests (as Tor has a high latency). Only around 0.91 s (5.6%) of page load time was due to transfer time.

The same figure shows that resources took 2.85 s to load on average, but only 0.079 s (2.7%) of resource load time was

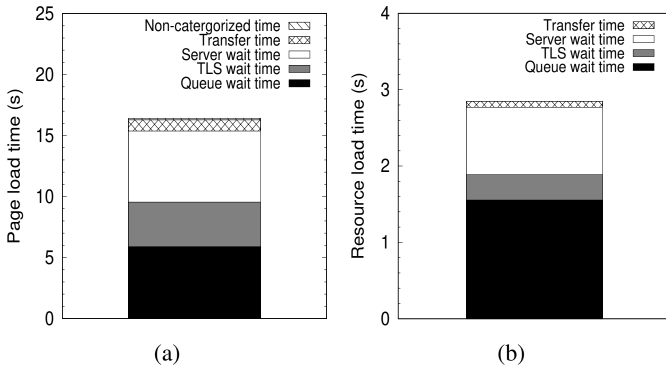

Fig. 2: Categorization of mean page load time and resource load time on the `pipeline` data set. Note the different scales of the y-axes.

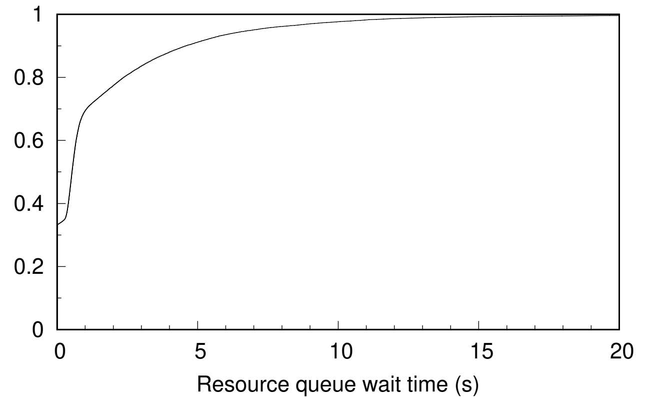

Fig. 3: Cumulative distribution function for queue wait times per resource, with a cutoff of 20 seconds.

due to transfer time. A much greater portion of resource load time, 1.56 s (55%), was spent waiting in queue. This is not sufficiently explained by resources waiting for connections to be established as it far exceeds one round-trip time. We show the distribution of queue wait times in Figure 3. Around a third of resources did not have to wait in queue as an established connection was already available, but more than half of them had to wait for more than half a second. and 30% had to wait for more than a second. These long queue wait times could indicate a lack of resource loading capacity or pipelining errors causing connection closure.

3) Analysis of potential issues: We analyze the possible issues raised in Section III-B.

Head-of-line blocking. Head-of-line blocking can delay pipelined resource loading because a resource cannot begin transferring until the previously pipelined resource has finished. We measure the *block time* of a resource by summing up the transfer times of all resources pipelined before it, only counting resources that were dispatched onto the same pipeline within 0.01 s of each other. We set this restriction to ensure that we are considering resources that were indeed blocked by previously pipelined resources. Among these resources (33% of all resources), the mean block time was 0.25 s. Only 1% of resources suffered a head-of-line blocking time over 0.5 s, and if we removed those, the mean block time drops to 0.05 s, insignificant compared to the mean resource load time.

Therefore, a tiny portion (around 0.2%) of resources suffered the majority of all head-of-line blocking. These large block times suggest that head-of-line blocking was rarely severe and had little impact on load times in the vast majority of pages. Note that block times are not necessarily caused by pipelining: they may be due to connection stalling issues, Tor circuit issues, server unresponsiveness, etc.

Pipelining errors. Pipelining support is required in HTTP/1.1 web servers. Surprisingly, we found that a significant portion of HTTP/1.1 pipelines encountered errors and were closed prematurely. Out of the 118,746 resources we found that attempted to use pipelining, 25,679 (22%) of them were dispatched on connections that were closed before loading them completely, forcing them to dispatch on a new connection. On the other hand, only 68 out of the 73,710 non-pipelined resources were sent on prematurely closed connections.

We found that most servers seemed to support pipelining correctly: 58% of the servers in our database that pipelined at least 2 resources did not prematurely close connections. It is not clear exactly why the other servers do not support pipelining: we observed that they simply aborted the connection without explanation some time after pipelined requests were sent. The large error rate of pipelining hampers its usefulness.

Choosing pipeline depth. As poor resource loading capacity increases queue wait time, and queue wait time is a significant portion of load time, it would seem that increasing resource loading capacity should help ameliorate long load times. In TB-8.5 with pipelining, we used 6 simultaneous connections per server and a pipeline depth of 6 (referred to as 6-6), and we change both these parameters to produce three more setups with higher resource loading capacity: 6 connections and 20 pipeline depth (6-20), 20 connections and 6 pipeline depth (20-6), and 20 connections and 20 pipeline depth (20-20). In comparative experiments, we found that, disappointingly, none of these parameter changes produced any notable difference in load time. The mean load time was 16.4 s for 6-6 and 20-6, 16.3 s for 6-20, and 16.5 s for 20-20.[3]

These results show that further increasing resource loading capacity on pipelining does not reduce page load time even for pages with many resources. In other words, pipelining already gives enough resource loading capacity.

Summary. Head-of-line blocking had little effect on page load times. We also found that adding to the resource loading capacity of pipelining by increasing the number of simultaneous connections and pipeline depth do not affect page load times; this suggests that pipelining had sufficient resource loading capacity. However, pipelines often close prematurely for a large portion of servers.

[3] As a sanity check, we confirmed that our parameters do affect the page load time by testing an intentionally poor setup of 1 connection and 100 pipeline depth, which had a mean load time of 21.2 s.

IV. HTTP/2

In this section, we analyze HTTP/2 and its impact on Tor Browser. This section is structured similarly to Section III: We describe how HTTP/2 works in Section IV-A, list potential issues in Section IV-B, and analyze real data in Section IV-C.

A. HTTP/2 in Tor Browser

Built on Google's experimental SPDY protocol, HTTP/2 changes how web pages are loaded: it uses multiplexed connections instead of multiple connections. The client establishes a single multiplexed connection with each server, able to carry 100 resource requests.[4] The responses can arrive in any order, and requests can be sent at any time without waiting for responses. To manage concurrent resource loading, the web-browsing client enforces flow control on incoming responses.

HTTP/2 replaces HTTP/1.1 pipelining on newer versions of Tor Browser. To our knowledge, its performance on anonymity networks is largely unevaluated. Web servers have increasingly adopted HTTP/2. On all major browsers, only pages using TLS can be loaded with HTTP/2.

B. Issues with HTTP/2

We discuss some potential issues with HTTP/2 here. Later, we analyze whether or not they affect web browsing.

Head-of-line blocking. With HTTP/2, all resources on the same server are loaded over a single TCP connection. Since TCP packets must arrive in order, a stalled connection will delay every packet and thus every resource currently being loaded on that connection. In other words, HTTP/2 still suffers from head-of-line blocking on the transport layer (TCP), though it does not cause head-of-line blocking by itself.

Transfer rate. Since a random delay is more likely to affect many resource loads on HTTP/2, the use of a single TCP connection renders the client more vulnerable to congestion. These issues may be exacerbated by poor network conditions on anonymity networks like Tor, and would be reflected in a poor transfer rate.

Premature connection close. We saw that pipelined connections were often closed prematurely after receiving an erroneous response. If HTTP/2 connections were to be closed prematurely, even more resources in flight would be dropped and have to be requested again, delaying the page load.

Extra round trip for ALPN negotiation. ALPN (Application-Layer Protocol Negotiation) lets the client determine if the server supports HTTP/2, and the protocol requires one round trip. In Tor Browser, we observed that ALPN negotiation happens after TLS negotiation is complete, even though ALPN's specifications claim that it should be negotiated alongside TLS. This implies that HTTP/2 can only begin one round trip after TLS is negotiated. For the duration between TLS completion and ALPN completion, TB-8.5 defaults to using HTTP/1.1 **without** pipelining to load resources.

[4] This parameter can be changed in the browser.

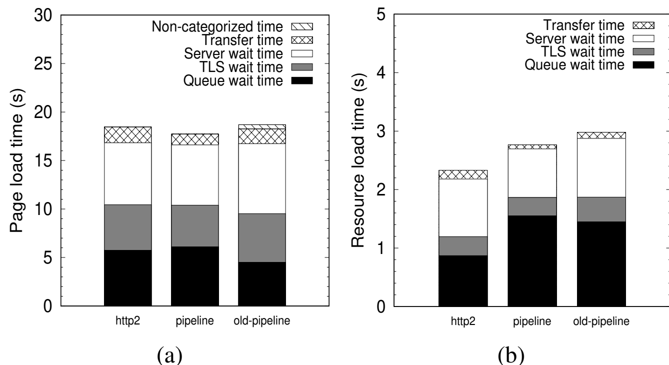

Fig. 4: Categorization of mean resource load time and page load time on our data set, comparing HTTP/2 on TB-8.5 (`http2`), pipelining on TB-8.5 (`pipeline`), and HTTP/1.1 on TB-6.5 (`old-pipeline`). Note the different scales of the y-axes.

The extra round trip time slows down the build-up of resource loading capacity in HTTP/2: it can only load six resources in the first round trip, while pipelining can load six times the pipeline depth. This undermines the advantage of using HTTP/2.

C. Data Analysis

We analyze HTTP/2 by comparing it with HTTP/1.1 with pipelining. To do so, we compare three Tor Browser setups: TB-8.5 with HTTP/2 (`http2`), TB-8.5 with pipelining and no HTTP/2 (`pipeline`), and TB-6.5 with pipelining and no HTTP/2 (`old-pipeline`). Note that we use HTTP/2 to refer to the protocol and `http2` to refer to the data set where we attempt to load pages on Tor Browser using HTTP/2; in particular, some servers in `http2` do not support HTTP/2.

1) Overall performance: We show the mean page load time over our full data set in Figure 4. Our results show that HTTP/2 performs a little worse than HTTP/1.1 pipelining on TB-8.5, and slightly better than pipelining on TB-6.5.

Focusing on the latter result first, the poor performance of TB-6.5 may be because of several flaws later fixed in TB-8.5. First, TB-6.5 can only establish connections to a server one at a time, starting the next connection establishment after the previous has been established. This slows down the ramp up for resource loading capacity. Second, there are several issues in its pipelining implementation described in Appendix A, and we fixed them when re-implementing pipelining in TB-8.5. TB-6.5 has one significant improvement for page loading time that is not found in TB-8.5, optimistic data (described later in Section V-C).

On the other hand, the poor performance of HTTP/2 is surprising. We explore preliminary explanations for this result in the following.

Did servers support HTTP/2?

We found that in `http2`, 95% of web pages had at least one resource sent using HTTP/2.[5] Overall, 49% of resources

[5] This statistic should not be interpreted as an accurate measurement of HTTP/2 adoption as we only analyze Alexa's top 1000 pages.

were sent on HTTP/2. (Note that servers with few resources would never get to use HTTP/2 even if they did support it, because of ALPN negotiation.) HTTP/2 support and usage is therefore widespread and well-captured in our data set.

Did a small portion of pages behave poorly?

To determine if a small amount of faulty data biased the mean result, we examine the page load time difference between `http2` and `pipeline` by directly comparing between instances on the same circuit (using the methodology in Section III-C). The page load time difference was $-0.74\ s \pm 6.54\ s$: the standard deviation was far higher than the mean, suggesting `http2` was not consistently inferior. If we discarded the top and bottom 20% of page load time differences, the result would be $-0.11\ s \pm 1.30\ s$. The median group of results performed relatively similarly.

Did certain types of pages behave poorly?

We extracted a number of features from each page and used Pearson's r to determine if they were correlated to the loading time difference between `http2` and `pipeline`. The maximum r is 1 and smaller values indicate less correlation. We tested the following features: number of resources, size of page, height of resource tree, and percentage of HTTP/2 resources. Our results show that none of these features were notably correlated with the loading time difference, the highest r being 0.22 between the number of resources and the loading time difference while other r values were under 0.15. This means that poor (or good) performance of HTTP/2 compared to pipelining was not localized to specific types of pages.

Therefore, these preliminary questions do not explain the lack of performance improvement of HTTP/2. We explore the issue further in the following.

2) Categorization of load time: We compare page load times and resource load times on `http2`, `pipeline` and `old-pipeline` in Figure 4, again using the methodology in Section II-C. We see that `http2` had a *shorter* resource load time, with an especially notable gain on queue wait time: this is because resources do not have to wait for HTTP/2 connections to finish loading previous resources. However, for overall page load time, the queue wait time advantage on `http2` vanishes. While on average resources have to wait about 40% less in `http2`, the resources that contribute to page load time do not experience such a beneficial decrease in queue wait time. Note that a resource loaded on a new, separate server waits for an equal amount of time in HTTP/2 and in HTTP/1.1 with pipelining. If these resources are the ones determining page load time, this could explain the lack of performance improvement of HTTP/2.

3) Analysis of potential issues: We turn to the aforementioned issues in HTTP/2 to determine if they impeded performance.

Head-of-line blocking. We first determine if the use of a single connection causes head-of-line blocking. To do so, we measured resource load time specifically for resources sent on HTTP/2 connections that were already loading other resources. Those 5,651 resources never had to wait in queue or connection establishment, but we did find that they waited $1.38\,\mathrm{s}$ on average for server response and took $0.40\,\mathrm{s}$ to transfer, compared to $0.99\,\mathrm{s}$ and $0.15\,\mathrm{s}$ respectively overall on `http2`. This does suggest that HTTP/2 head-of-line blocking is sometimes a problem on Tor Browser: a server that has already established an HTTP/2 connection responds and transfers a little more slowly.

Transfer rate. To find out if the use of a single connection in HTTP/2 affected transfer rates, we summed up the resource sizes and transfer times of all resources over $500\,\mathrm{kB}$ in size. Dividing the two, we found the data transfer rate on `http2` to be $164\,\mathrm{kB/s}$ and `pipeline` to be $347\,\mathrm{kB/s}$. This does not necessarily mean that HTTP/2 is slower, however: the data transfer rate is calculated on a per resource basis, and HTTP/2 allows multiplexing while pipelining does not. On the top 10 sites, transfer rate on `http2` and `pipeline` were respectively $343\,\mathrm{kB/s}$ and $420\,\mathrm{kB/s}$, a small difference.

Premature connection close. Out of the 99,947 resources we dispatched on HTTP/2 connections, only 7 of them could not be loaded before the connection closed prematurely: HTTP/2 connections rarely failed unlike pipelining connections.

Extra round trip for ALPN negotiation. Since we cannot eliminate this round trip in practice, we cannot analyze its effect on page loading yet. To do so, we need the second component of BLAST, the simulator, described in the next section; the simulator will reveal that the ALPN negotiation round trip has a minor effect on page load time.

Summary. Despite the fact that there appear to be no significant issues with HTTP/2 that affected page loading on Tor, pages are loaded slightly more slowly in `http2` than `pipeline`. The main advantages of HTTP/2, superior connection multiplexing and higher resource loading capacity, do not matter for page loading on Tor Browser. In addition, TB-8.5 uses a poor fall back for servers that do not support HTTP/2 — HTTP/1.1 **without** pipelining.

V. DESIGNING A BETTER BROWSER

A. What causes long load times?

Our analysis of HTTP/1.1 with pipelining showed that increasing the number and depth of pipelines did not improve page load time. Comparing HTTP/1.1 with pipelining and HTTP/2, we found that HTTP/2 connection multiplexing did not improve page load time either. Furthermore, whenever their performance differed, such a difference was not due to page structure or size. The lack of improvement in HTTP/2 was not due to any issues with using a single TCP connection, either.

All of these observations suggest that increasing resource loading capacity (whether with more/deeper pipelines, or with HTTP/2 multiplexing) does not solve the issue that loading pages on Tor Browser is slow. Furthermore, our categorization of page loading time also suggests that increasing bandwidth would not significantly speed up page loading either. This would imply that loading time on Tor Browser is chiefly determined by the *minimum number of round trips required to load a page*, which we refer to as $minRTT$. In our data, we calculated the r correlation coefficient between $minRTT$ and

page load time to be 0.57; detailed results are presented in the Appendix. In addition, increasing resource loading capacity and increasing bandwidth would not change $minRTT$: this would explain why they do not speed up page loading.

Therefore, the best way to speed up page loading on Tor Browser should be to reduce the number of round trips required to load a page. We cannot do so by simply changing browser parameters; extensive work with new browser code and network-wide infrastructure is required to achieve this. Since the implementation cost of these proposed features is high, we use simulation to determine their usefulness first so as to motivate implementation.

B. Simulation

The BLAST page loading simulation reads basic information about the structure of the page and generates network events corresponding to its loading. The simulator takes exactly the following information as input:

- The resource tree: The list of resources, their sizes, and their parents;
- The list of servers, which resources each server hosts, and whether it supports TLS, pipelining, and HTTP/2;
- The mean bandwidth and round-trip time.

Note that the simulator does not use any resource timing information; it only needs to know static information about the web page to be simulated. It simulates TCP connection establishment, TLS, and ALPN, and the dispatching and loading of every resource. It outputs when and how each resource was loaded, allowing us to categorize load times as before. It simplistically uses a constant bandwidth rate and round-trip time. While we could randomize those, as we are using the simulator to guide browser design, removing an unnecessary element of randomness ensures that faster load times are a consequence of better design.

Our simulator does not simulate random errors and delays, congestion control on Tor, HTTP/2, or TCP, or random collapse of connections during loading. Despite these limitations, we demonstrate that our simulator can accurately predict load times. We set the round-trip time to $0.8\,\mathrm{s}$ and the bandwidth to $164\,\mathrm{kB/s}$ for `http2` and $347\,\mathrm{kB/s}$ for `pipeline`; these were the same numbers we measured from real data in Section IV-C. The simulator finds a mean page loading time of $18.0\,\mathrm{s}$ for `http2` and $17.1\,\mathrm{s}$ for `pipeline`, compared to $18.4\,\mathrm{s}$ and $17.6\,\mathrm{s}$ in our real data sets respectively. The simulated numbers are not only respectively accurate, the page loading time difference between `http2` and `pipeline` is nearly identical between simulation and reality. This shows the value of our simulator as one of our main findings, "HTTP/2 is slightly slower than HTTP/1.1 with pipelining on Tor Browser", could have been obtained through the BLAST simulator.

We show a scatter plot of real versus simulated page load times in Figure 5 for `http2`. The correlation is strong at $r = 0.63$, and the simulator produces an accurate mean page loading time; however, the simulator is not necessarily accurate at the page instance level, in particular because it is not possible to simulate random round-trip times, delays, and congestion.

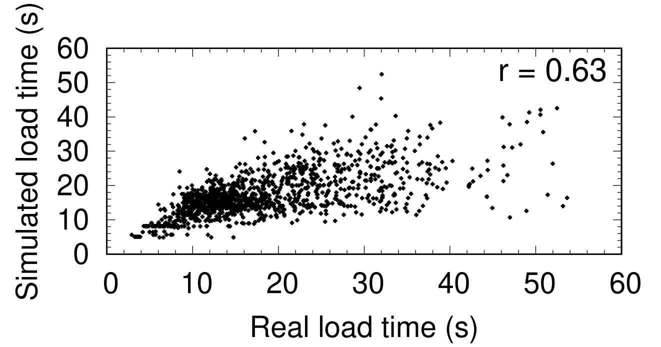

Fig. 5: Simulated versus real page load times on our data set for `http2`, showing Pearson's correlation $r = 0.63$.

C. Proposed features

We propose six improvements to Tor Browser to speed up page loading here, with varying barriers to implementation:

1) Requires changing browser code/protocol implementation: TCP Fast Open, optimistic data, 0-RTT TLS.
2) Requires new infrastructure: Redirection database, HTTP/2 server database, prefetching database.

We decided that we would base our changes on HTTP/2 rather than pipelining. Due to the significant error rate under pipelining and the fact that Firefox and Tor are no longer willing to support pipelining code, improving HTTP/2 performance is a more realistic priority.

Our proposed new infrastructure (additional databases for client use) would not require cooperation from web servers or any third-party entity. We propose that they should be distributed by Tor directory servers, which already distribute Tor relay information, to Tor clients on startup. We describe each of these proposals in the following.

TCP Fast Open. TCP Fast Open is a feature of TCP that eliminates a round trip in the establishment of TCP connections if the client has previously established another TCP connection to the server. By sending a cryptographic cookie in the initial SYN packet, the client validates her identity to the server. The client would send application data — specifically, a GET/PUT request — alongside the initial SYN packet and the server can immediately respond to that request. TCP Fast Open has been experimentally implemented in the latest version of Firefox, though it is not currently enabled in Tor Browser.

Optimistic data. In 2010, Goldberg proposed the use of optimistic data to reduce round-trip times on Tor [10]. We saw in Figure 1 that it takes two or three round trips to load a resource from a server without a prior connection. With optimistic data on an unencrypted connection, the client would send the resource request along with the connection establishment request; the Tor exit node holds the resource request until the connection is established, and then sends the resource request. This reduces the two round-trip times between client and server to one round-trip time between client and server, plus one (much smaller) round-trip time between the Tor exit node and the server. On an encrypted connection, the client would instead send the first TLS negotiation packet

(Client Hello) along with the connection establishment request, which saves about one round trip as well.

Optimistic data was implemented in 2013 with changes to Tor and by hacking a shortcut into the browser SOCKS state machine, but this hack was removed recently because it was not compatible with newer browser code. We could not re-implement optimistic data using the previous hack, but we can test whether a full implementation is useful with simulation.

0-RTT TLS. Normally, TLS negotiation takes one extra round trip after connection establishment, during which the connection cannot be used to send resource requests. TLS 1.3 introduces 0-RTT session resumption: it allows clients to remember negotiated keys with servers and send them back with session resumption tickets. While there are security concerns with 0-RTT regarding forward secrecy and replay attacks, researchers have proposed fixes to resolve these issues and some browsers and servers have enabled it [17]. We do not implement 0-RTT TLS, but we want to evaluate its effect on resource load times and page load times to understand how much it might help Tor Browser. HTTP/3 (HTTP over QUIC) also promises a reduction in round-trip times by allowing a single QUIC handshake to establish connection and negotiate encryption.

Combining the above, we can reduce the number of round trips incurred to load a resource to a single round trip over Tor, plus a (much smaller) round trip between the Tor exit node and the server. Theoretically, we thus approach the minimum number of round trips without changing the resource tree of the web page.

Redirection database. Many pages in our data set redirect the client upon initial page navigation. For example, *youtube.com* redirects to *https://youtube.com*, which then redirects to *https://www.youtube.com*. Each redirection incurs multiple unnecessary round trips. Due to our high latency, redirects are quite detrimental to page load times.

HTTPS Everywhere, included in Tor Browser, does partly reduce the number of redirects: it relies on a user-maintained database to replace clicked or user-typed URLs with their encrypted versions, browser-side. In the above, HTTPS Everywhere would eliminate the first redirect, but not the second one. Further, HTTPS Everywhere does not deal with localization.

We propose that Tor Browser should reduce redirects with an extensive redirection database that also takes care of localization redirects.[6] We generate this database automatically by parsing redirection responses from web servers. The browser UI should inform the client when it skips redirects using the database, so that the client can revert or disable rules that are inconvenient.

HTTP/2 database. We investigate the use of a list of HTTP/2 servers for Tor clients to eliminate the extra round trip to negotiate ALPN (discussed in Section IV-B). Even if there are erroneous entries in the database, which should be rare as servers that start supporting HTTP/2 would not abandon it, the browser would simply default to using HTTP/1.1 after failing to establish a HTTP/2 connection.

Prefetching database. If the client knew which resources were associated with a page before parsing resource responses, she could request these resources as soon as page loading begins. This flattens the resource tree structure and eliminates the otherwise necessary round trips between resources and their children resources. Some resources cannot be effectively prefetched because they are based on randomized or dynamic activity (e.g. advertisements). We automatically generate a resource prefetching database from BLAST logs and attempt to prefetch any resource that occurs in almost all instances of a web page.[7]

D. Evaluation of proposed changes

We evaluate all the browser improvements we proposed above using our simulator here. We evaluate each change both cumulatively and separately. In the cumulative evaluation, a latter improvement in the following list contains all the improvements before it:

1) Original Tor Browser with HTTP/2
2) TCP Fast Open
3) Optimistic data
4) 0-RTT TLS
5) Redirection database
6) HTTP/2 database
7) Prefetching database

In Figure 6, we show the mean page load time and resource load time according to the BLAST simulator. With the exception of the HTTP/2 database, each added feature clearly reduces both load times. Resource prefetching using a database produces the largest improvement, alone able to reduce mean page load time by 35%; all features combined produce a 61% decrease in mean page load time, from 18 s to 7.1 s. We give details on each of the improvements in the following.

TCP Fast Open. TCP Fast Open provides an advantage only if the client has already established a connection to the server and needs to establish another. We found that 18% of our established connections benefited from TCP Fast Open, cutting down one round trip with each connection.

Optimistic data. Optimistic data effectively shortens the round trip for HTTP request right after connection establishment to the much shorter round trip between the exit node and the web server. To simulate this, we reduce the round-trip time in that case to 0.1 s. Optimistic data speeds up page loading greatly while requiring no extra infrastructure.

0-RTT TLS. 0-RTT TLS has been prototyped for session resumption and there is currently a proposal to implement it in HTTP/3 without session resumption. If we discard HTTP/3 (true) 0-RTT TLS because its implementation is not expected to come soon, mean page load time with all other improvements increases from 7.1 s to 9.2 s.

[6] As localization is based on the Tor exit relay's location, not the user's preference, it usually redirects Tor users to a wrong page.

[7] For clarity, our technique is unrelated to HTML5 link prefetching, a technique for websites to use an HTML prompt to instruct clients to load resources.

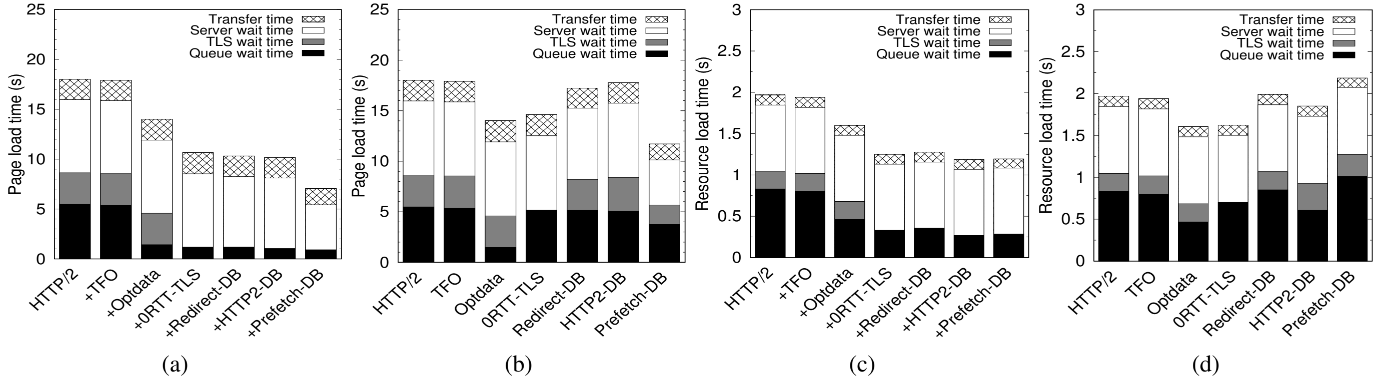

Fig. 6: Mean page load time (a, b) and mean resource load time (c, d) for our suggested improvements as predicted by the BLAST simulator on resource trees parsed from `http2`. (a) and (c) show cumulative improvement (each improvement is applied on top of previous ones), while (b) and (d) show individual improvement.

Redirection database. We automatically built a redirection data set using BLAST logs for our pages and modelled the use of such a data set. The top 200 pages had a mean of 1.17 redirects, allowing us to save two or three round trips for each redirect. This approach is however too optimistic since we know all the pages we want to load. We could have created a data set with more pages — top 10,000 pages, for example — but users will still visit pages outside of this set. On the other hand, it is usually only the top sites that use one or more initial redirects, as less popular websites are less likely to have multiple servers, localization, or URL canonicalization.

To simulate a failure rate, we only used redirection for the top 200 pages, simulating a 60% hit rate where only hits produce a benefit and misses do not affect page loading.

HTTP/2 database. The use of an HTTP/2 database — to allow clients to benefit from HTTP/2 without waiting for an extra round trip during which she has limited resource loading capacity — produced almost no discernible benefit for page load times. This confirms that further increasing resource loading capacity is not a concern for page loading on anonymity networks.

Prefetching database. We only attempted to prefetch a resource if it occurred in 90% or more of the page's instances. For the top 10th to 200th pages, since we have no more than 5 instances each, this means that all instances of the page must contain that resource. Since we loaded instances of every page over two months, the resources we chose to prefetch certainly have a long lifespan. Similar to the redirection database, we only apply prefetching to the top 200 pages so that the hit rate is 60%.

Despite our conservative strategy, we prefetched a mean of 55.7 resources per page among our top 200 pages, and 58% of resources were prefetched. Only 1% of the resources prefetched using our database were false positives, i.e. not needed for that page, so the additional bandwidth cost is quite low. As a database to contain top 10,000 pages would still be less than a megabyte in size, there is almost no storage or communication cost to regularly update such a database, which we propose Tor directory servers should distribute. A more aggressive database could cause more false positives to further reduce load times; on the other hand, the benefit could be reduced if resource loads were forced by scripts and cache policies regardless, which we could not simulate.

E. Extensibility of improvement

We checked to see if load time decreases were spread evenly across most pages or restricted to select pages. Over our data set, we observed at least a 25% decrease in load time in 89% of page instances, and at least a 50% decrease in load time in 68% of page instances. This suggests that most web pages benefited significantly to varying degrees. This is despite the fact that we intentionally constructed our data set so that a third of page instances were not in the databases and thus did not benefit from those methods.

We also wanted to evaluate whether or not load times decreased disproportionately in certain types of pages, such as smaller pages, or pages with fewer resources. To do so, we measured the performance improvement (as a percentage) against three features: total page size, height of resource tree, and number of resources. The respective r correlation values were 0.20, 0.03, and 0.08; these low correlation values are a positive result suggesting that our improvements were equally felt by all types of pages. Further examining total page size, we divided the data set into two equal halves (above or below total page size of $1.36\,\text{MB}$), and observed a 62% $\pm$ 24% improvement on the lower half and a 51% $\pm$ 31% improvement on the upper half. The small difference between these two sets suggests that even the largest pages benefit significantly from our features.

It makes sense that our improvements would be generally felt across all pages: they reduce round trip times and flatten resource tree structures, which affect all pages. We can therefore anticipate that these improvements would extend beyond Alexa's top 1000 pages.

F. Evaluation of implementation

We implemented a prototype of two features as Firefox add-ons for the top 200 pages: (1) redirection database, where

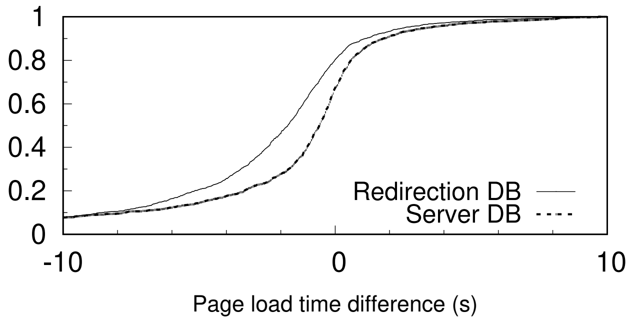

Fig. 7: CDFs of differences in page load times for implementations of Redirection database and Server database compared to original Tor Browser, tested on ten instances each of the top 200 pages collected with Tor Browser. These features were implemented as Firefox add-ons for Tor Browser. Negative time values indicate an improvement.

we used BLAST to automatically obtain redirects and substitute the user's action to visit the top 200 pages without requiring server round-trips; (2) server database, where we used BLAST to automatically obtain the servers each web page connects to and immediately create a connection to them when the user accesses the top 200 pages.

The server database is a weaker version of the prefetch database; instead of prefetching known resources on the page, we pre-establish connections to known servers used by the page so resources can use these connections without waiting. There are significant technical challenges in implementing the prefetch database for Tor Browser through an add-on.[8] Both implementations are evaluated on 10 instances each of the top 200 pages through Tor Browser. Though compatible, the two implementations were evaluated independently. We present the results separately in the following, based on real data (not simulation).

Redirection database. The redirection database allows the user to skip a few unnecessary round-trips when loading a page. On the top 200 pages, we obtained a 14.7% decrease in page load times from 18.4 s to 15.7 s. Per-instance, the decrease was 2.7 s $\pm$ 5.7 s, a large variation due to the randomness of network conditions and Tor circuits. 80% of page instances saw a decrease in load times; 62% saw a decrease of more than 1 second, while 12% saw an increase of more than one second. The simulator predicted a decrease from 17.3 s to 15.5 s, a similar but smaller 10.4% decrease.

Server database. The server database allows the user to pre-establish connections with required servers at the moment page load begins, rather than at the moment resources are required. On the top 200 pages, we obtained a 9.4% decrease in page load times from 18.5 s to 16.8 s. Per-instance, the decrease was 1.7 s $\pm$ 6.0 s, a somewhat larger variation than the redirection database. 66% of page instances saw a decrease in load times; 66% saw a decrease of more than 1 second, while 16% saw an increase of more than one second.

We show a CDF of reduction in load times per instance for both features in Figure 7. The success of these prototype implementations suggest that the large load time decreases predicted by the simulator are indeed achievable, and in fact we see a slightly larger improvement due to the redirection database than predicted.

[8] We observed that Tor Browser refused to use cached resources during the same page load even though memory caching was on, although it used cached resources in a subsequent page load.

VI. Discussion

A. Implementation and maintenance

Our proposed features require changes to Tor Browser to support. We anticipate that those changes would largely be focused on the browser's HTTP connection manager, which governs how connections are created and how resources are dispatched onto them:

- TCP Fast Open is already available on Firefox, although currently disabled; TLS session resumption is also available but used by limited servers due to potential risks. True 0-RTT TLS (without resumption) is anticipated in HTTP-over-QUIC.
- Since the optimistic data hack no longer works (as earlier described), implementing it would likely require a re-writing of the connection manager's state machine so that it would recognize that resource requests should be sent earlier.
- We can implement the redirection and prefetching databases as Firefox add-ons, though to force the browser to use prefetched resources, we would need to change the connection manager. We would also need to make changes to Tor itself to load those databases for the client.
- We do not propose to implement the HTTP/2 database due to its small performance gain shown and the possibility that future changes (such as a better ALPN implementation) would eliminate the need.

Patches to the connection manager would not be alien to Tor Browser as it has had multiple patches to implement its own features such as randomized pipelining. However we saw from the failure of optimistic data in newer browser versions that Firefox version changes can demand additional maintenance for such features, but Tor developers may not necessarily prioritize such maintenance. Our proposed features require expert maintenance to ensure they remain functional with every browser version update, especially since Firefox is unlikely to support our features (as they are not designed for normal browsing). The proposed databases also need to be constantly updated, though this is easily automated.

The volatility of browser enhancements is why we believe simulation has great value for experimental validation. Simulation allows our results to be obtained on a reproducible and verifiable medium. The predictive power of the BLAST simulator allows us to prove the usefulness of our enhancements in order to adopt motivation and constant maintenance.

B. Network design and browser design

Previous work on improving anonymity network user experience focused on improving network design rather than browser design (detailed in Section VII). Better network design works hand-in-hand with better browser design in improving user experience, but we have not seen an evaluation of how network design would affect page load times. We perform such an evaluation here using the BLAST simulator.

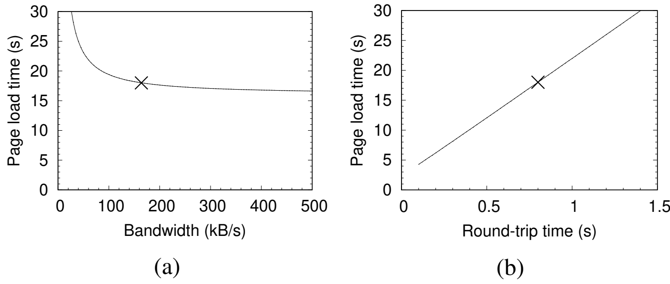

Fig. 8: Mean page load time based on round-trip time and per-resource bandwidth, simulating `http2`. In (a) we vary the bandwidth used while fixing the round-trip time to $0.8\,\mathrm{s}$, marking the original bandwidth setting of $164\,\mathrm{kB/s}$ with a cross. In (b) we vary the round-trip time while fixing the bandwidth used to $164\,\mathrm{kB/s}$, marking the original round-trip time of $0.8\,\mathrm{s}$ with a cross. Each graph contains 100 data points.

We change the simulator parameters of mean round-trip time (originally $0.8\,\mathrm{s}$) and per-resource bandwidth (originally $164\,\mathrm{kB/s}$), and evaluate their effects on HTTP/2 using the simulator. We show how page load time changes based on round-trip time and bandwidth in Figure 8.

We see that the round-trip time has an almost entirely linear effect on overall page load times. Our choice of $0.8\,\mathrm{s}$ probably does not represent Tor's minimal round-trip time and includes some degree of congestion. If we could reduce Tor's mean round-trip time by half, page load time would also be reduced by half. In contrast, increasing bandwidth has a much reduced effect on page load times. If we doubled the bandwidth, page load time would decrease from $18.0\,\mathrm{s}$ to $17.0\,\mathrm{s}$ (5.6% decrease). Tor's bandwidth is sufficiently high that increasing it further has only a minor effect on page load time.

C. Privacy implications

In this section, we analyze if and how our proposed browser features would impact Tor's anonymity guarantees. In doing so, we also make recommendations on their implementation to avoid potential compromises.

First, we note that an eavesdropping attacker could distinguish between someone who is using our proposed enhancements and someone who is not; we do not consider this a threat to anonymity because such features, if proven beneficial, should be adopted universally and not by a small portion of users. In that case, the client's identity would not be linked to whether or not they use these enhancements. Instead, we focus on whether or not an eavesdropping attacker can distinguish between someone who has visited a webpage and someone who has not in the following.

We analyze how our proposed enhancements impact Tor's anonymity by first dividing them into three categories:

1) Requires *tokens*. TCP Fast Open and 0-RTT TLS uses tokens stored in the browser to save round trips when a client re-connects to a previously visited website.[9]
2) Requires *databases*. These include the Redirection, HTTP/2, and Prefetching databases; we propose that the client loads these databases from the Tor directory servers to aid in their web browsing alongside server descriptors, which does not increase the client's vulnerability since server descriptors are already loaded.
3) Optimistic data does not require tokens or databases, so it belongs in its own category.

[9] Note that if 0-RTT TLS is implemented using a single QUIC handshake as proposed in HTTP/3, it would not require tokens.

We determine if these tokens and databases cause privacy issues by considering three types of relevant attacks in this work.

Storage-seizing attackers. These attackers gain control over the client's hard disk and seek to determine what they did on Tor Browser. To defend against these attackers, *tokens* should be stored in memory only and should not leave a hard disk trace. *Databases* should not be modified even if the client would obtain a performance improvement; for example, if the client knows that a website allows HTTP/2 and it was not recorded in the database, she should not add that record as it would serve as evidence that she visited the site.

Browser Fingerprinting attackers. These attackers control a web server and seek to determine the client's identity with prompts to which different clients answer differently [8]. To defend against browser fingerprinting, *tokens* should be implemented in such a way that a web server cannot check whether or not they exist unilaterally. For example, a web server should not be able to ask if the client has a 0-RTT TLS session resumption ticket for another domain. *Databases* are not open to browser fingerprinting attacks; even if a leak would allow the server to query the client's databases, its contents are determined by the Tor directory servers and do not reflect the client's activity.

Website Fingerprinting attackers. These attackers are local, passive eavesdroppers who observe traffic patterns in order to determine what the client is doing (which websites she is visiting). We observe that our faster re-connection *tokens* alter traffic patterns, allowing such an attacker to determine whether or not the client has visited a web page previously within the same session. However, this is already true for Tor Browser, which uses memory-based caching, allowing web resources to be stored in memory for use in a future web visit. Our proposals therefore do not compromise the client more than memory-based caching already does, as long as the necessary tokens are deleted along with memory caches after the termination of a browsing session.

There remains the harder question of whether or not enhancing the browser would in and of itself make the website fingerprinting classification task easier. We cannot definitively answer this question without a full implementation of all proposed enhancements and subsequent testing with state-of-the-art website fingerprinting attacks; even then, better attacks in the future or alternative implementations may change the answer. There is some evidence to suggest that our enhancements, which generally reduce the number of round trips required to load a page, would make website fingerprinting harder. Researchers have noted that website fingerprinting relies on identifying bursts of packets in the same direction [11], [19], [21], [22], [26], and that web pages with more bursts are harder to obscure [27]. This would suggest that reducing the number of round trips is more likely to impede website fingerprinting than to aid it.

VII. Related Work

We are not aware of any related academic work on solving the problem of *browser design* for anonymity networks. On the other hand, much work has been done on *network design* for anonymity networks (focusing on Tor), with the same motivation as our work: to improve users' experience, especially latency-sensitive users who need to be convinced to sacrifice utility for privacy. We survey these works here as we share its objectives and some of its methodology.

One way to improve anonymity network design is to tackle the *relay selection* problem. To deliver user traffic, Tor chooses volunteer relays at random to form a circuit lasting for about 10 minutes. Snader and Borisov [23] show a way to improve relay bandwidth reporting and for users to choose relays based on a trade-off between bandwidth and anonymity. Wang et al. [25] recommends congestion-aware path selection: the user can measure relay congestion based on timing packet round trips, and elect to drop congested circuits to improve performance. Akhmoondi et al. [1] suggests that Tor clients should choose relays based on their autonomous systems to reduce latency.

Other works have attempted to enhance Tor's performance by changing its behavior at the network-stack level. These proposals include DefenestraTor [4] to improve Tor's congestion control behavior; DiffTor to classify Tor traffic in real time to offer distinct classes of service [3]; and traffic splitting with circuit multiplexing [2], by AlSabah et al. DefenestraTor showed a 10–20% decrease in time to download 5 MB files. A proposal by Jansen et al. to improve Tor's socket interactions with real-time dynamic computation of socket congestion, KIST, is currently used by Tor. [12] It showed a reduction of latency from 0.838 s to 0.686 s, a 18% decrease; we saw in this work that latency reduction is highly significant for speeding up page loading.

To address the challenge of determining how much these proposals would improve Tor, Bauer et al. created ExperimenTor [5] to emulate Tor nodes so as to test various proposals on a toy network. Jansen et al. created Shadow [13] to simulate Tor nodes for the same goal. Using an improved version of ExperimenTor, Wacek et al. [24] evaluated relay selection proposals.

Improving anonymity network performance serves two important privacy goals, besides the overarching goal of improving user experience. First, the more concurrent users there are, the larger their anonymity set. Therefore, the greater the number of people using Tor, the less likely an eavesdropper is to identify a Tor user, which is easier if they have prior information (such as timing or locale). Convincing otherwise reluctant users to use Tor therefore benefits all current Tor users. For this reason, there has been much work on enhancing the scalability of Tor [14], [18].

Second, Tor may be vulnerable to website fingerprinting, which have recently started to show success in open-world experiments against large Tor data sets [11], [15], [19], [21], [22], [26]. Currently, the best proposed defenses against website fingerprinting all have large overhead values [6], [16], [27] and will inevitably slow down the network, hurting user experience. Combining a defense with our browser improvements could still result in reduced page load times while offering better resilience against website fingerprinting.

VIII. Conclusion

We investigated how Tor Browser loads web pages using BLAST, our new logging, analysis and simulation tool. Analyzing BLAST logs, we found inordinately large queue wait times and unnecessary round trips. We leveraged resource trees to effectively represent web pages; with resource trees, the BLAST simulator is able to predict how long it will take page to load and how they will be loaded. This lets us easily observe how much different browser improvements would speed up page loading.

The analytical and predictive power of BLAST allows us to make several important observations. We found that browser performance on Tor, a high-latency environment, is almost entirely dependent on round trips instead of bandwidth. We observed that increasing resource loading capacity does not improve page load time on anonymity networks, and therefore HTTP/2 did not help. To speed up browsing, we need to reduce the number of round trips. We also found that several theoretical issues behind pipelining and HTTP/2 had no significant effect on Tor, including head-of-line blocking, transfer rate, and potential errors in HTTP/2 connections.

We proposed a series of improvements to speed up browsing focusing on reducing round-trips in two directions: reducing the number of round trips required to load each resource (TFO, optimistic data, 0-RTT TLS) and reduce the number of round trips required to load the web page as a whole (databases for redirection, HTTP/2, and prefetching). Our simulator predicts that page load times on Tor Browser would be reduced by 61%, prefetching contributing to roughly half of the improvement. There are only two trivial sources of extra bandwidth for Tor in our proposed features: the cost to distribute and update page databases for clients (less than a megabyte with 10,000 pages), and the minor chance of a prefetching false positive causing unnecessary loading. All of our proposed changes are client-side, and adoption is instantaneous and invisible once deployed.

Reproducibility. We publish our code and data at:

`github.com/blastpipeline/blastpipeline`

The repository includes the following:

- The BLAST logger: a patch to Tor Browser to instrument it, and Python code to parse those logs into resource trees and other useful formats.
- The BLAST simulator: Python code to simulate HTTP/1.1 with and without pipelining, HTTP/2, and all six proposed features.
- Data sets to validate the logger and simulator for HTTP/1.1 with pipelining on TB-8.5, HTTP/2 on TB-8.5, and HTTP/1.1 with pipelining on TB-6.5.
- Prototype implementation of redirection and server databases.

References

[1] Masoud Akhoondi, Curtis Yu, and Harsha V Madhyastha. Lastor: A low-latency as-aware tor client. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 476–490, 2012.

[2] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. The path less travelled: Overcoming tors bottlenecks with traffic splitting. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 143–163. Springer, 2013.

[3] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. Enhancing Tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 73–84, 2012.

[4] Mashael AlSabah, Kevin Bauer, Ian Goldberg, Dirk Grunwald, Damon McCoy, Stefan Savage, and Geoffrey M Voelker. Defenestrator: Throwing out windows in tor. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 134–154, 2011.

[5] Kevin S Bauer, Micah Sherr, and Dirk Grunwald. ExperimenTor: A Testbed for Safe and Realistic Tor Experimentation. In *CSET*, 2011.

[6] Xiang Cai, Rishab Nithyanand, and Rob Johnson. CS-BuFLO: A Congestion Sensitive Website Fingerprinting Defense. In *Proceedings of the 13th ACM Workshop on Privacy in the Electronic Society*, 2014.

[7] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[8] Peter Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18, 2010.

[9] Benjamin Fabian, Florian Goertz, Steffen Kunz, Sebastian Müller, and Mathias Nitzsche. Privately waiting–a usability analysis of the tor anonymity network. In *SIGeBIZ track of the Americas Conference on Information Systems*, pages 63–75. Springer, 2010.

[10] Ian Goldberg. Optimistic data for Tor (PETS rump session talk). https://thunk.cs.uwaterloo.ca/optimistic-data-pets2010-rump.pdf. Accessed Jan. 2019.

[11] Jamie Hayes and George Danezis. k-Fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[12] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul F Syverson. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *Proceedings of the 23rd USENIX Security Symposium*, pages 127–142, 2014.

[13] Rob Jansen and Nicholas Hooper. Shadow: Running Tor in a box for accurate and efficient experimentation. In *Proceedings of the 18th Network and Distributed System Security Symposium*, 2011.

[14] Rob Jansen, Nicholas Hopper, and Yongdae Kim. Recruiting New Tor relays with BRAIDS. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 319–328, 2010.

[15] Rob Jansen, Marc Juarez, Rafael Galvez, Tariq Elahi, and Claudia Diaz. Inside job: Applying traffic analysis to measure Tor from within. In *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

[16] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. Toward an Efficient Website Fingerprinting Defense. In *Computer Security–ESORICS 2016*, pages 27–46. Springer, 2016.

[17] Colm MacCárthaigh. Security Review of TLS1.3 0-RTT. https://github.com/tlswg/tls13-spec/issues/1001. Accessed Jan. 2019.

[18] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable Onion Routing with Torsk. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 590–599, 2009.

[19] Andriy Panchenko, Fabian Lanze, Andreas Zinnen, Martin Henze, Jan Pennekamp, Klaus Wehrle, and Thomas Engel. Website Fingerprinting at Internet Scale. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[20] Mike Perry. Experimental Defense for Website Traffic Fingerprinting. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting, September 2011. Accessed Feb. 2015.

[21] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. 2018.

[22] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 25th ACM Conference on Computer and Communications Security*, pages 1928–1943. ACM, 2018.

[23] Robin Snader and Nikita Borisov. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *Proceedings of the 15th Network and Distributed System Security Symposium*, volume 8, page 127, 2008.

[24] Chris Wacek, Henry Tan, Kevin S Bauer, and Micah Sherr. An Empirical Evaluation of Relay Selection in Tor. In *Proceedings of the 20th Network and Distributed System Security Symposium*, volume 13, pages 24–27, 2013.

[25] Tao Wang, Kevin Bauer, Clara Forero, and Ian Goldberg. Congestion-aware path selection for tor. In *International Conference on Financial Cryptography and Data Security*, pages 98–113, 2012.

[26] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[27] Tao Wang and Ian Goldberg. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

Appendix

A. Pipelining implementation issues

We note a number of issues in the implementation of pipelining in TB-6.5 that delayed page loading:

- **Minimum depth.** TB-6.5 enforces a minimal length of three on all pipelines. If there are fewer than three requests in the pending queue, and at least one active pipeline, the browser will never send out any resource requests, even if there are idle connections waiting to send out requests. This causes unnecessary queue wait times.
- **Randomization of resources.** TB-6.5 randomly shuffles resources before dispatching. As more important resources are often parsed first, this is disadvantageous to page loading. This may have been meant to defeat website fingerprinting attacks [20], but previous work suggests that this has no effect against any website fingerprinting attack [26].
- **Randomization of pipelines.** TB-6.5 randomly chooses pipelines to dispatch from the set of all valid pipelines. This causes TB-6.5 to lose possible optimization options for pipeline selection. For example, pipelines that have completed TLS negotiation should be prioritized. Among those, pipelines with fewer dispatched resource requests should be prioritized to avoid head-of-line blocking.
- **Blocking resources.** The HTTP server can mark any resource as a *blocking resource*. Before a blocking resource is fully loaded, no new resource can be dispatched, even onto pipelines, and no new connection can be created. This was intended to ensure that certain resources would be loaded as soon as possible. However, it is not worth delaying all other resources by round trips to accommodate a single resource.

B. Detailed results on correlation of page load time

We calculated the r correlation coefficient with four features as described in Section V-A on `http2`. These four features were $minRTT$, size of page, HTTP/2 usage percentage, and number of resources. $minRTT$ is calculated by taking

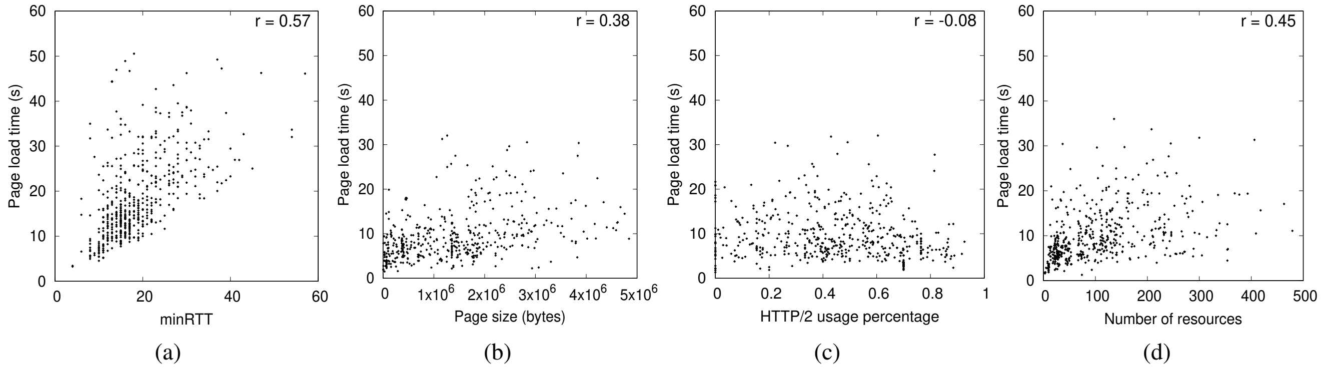

Fig. 9: Scatter plots for page load times on `http2` versus (a) minRTT, (b) Size of page, (c) HTTP/2 usage percentage, and (d) Number of resources. In each graph, r is the Pearson correlation coefficient between the plotted serieses.

each resource on the path between the last resource and the root of the resource tree, and adding:

- 3 if the resource is on a new HTTPS connection;
- 2 if the resource is on a new HTTP connection;
- 1 otherwise.

We present the results in Figure 9. $minRTT$ indeed has the best correlation with page load time, with $r = 0.57$ compared to $r = 0.45$ for number of resources and $r = 0.38$ for page size. In addition, from the $minRTT$ plot, we can see that the page load time was always greater than and often close to $minRTT$ times $0.38\,\text{s}$, reinforcing the notion that page load time was often directly caused by round trips.

C. BLAST implementation

1) How BLAST simulates web page loading: BLAST simulates web page loading by mimicking while simplifying the logic of Firefox's connection manager. It maintains two types of objects, connections and resources. The simulation is event-driven; the simulation begins with a single event corresponding to loading the first resource, which generates more events, and the simulation ends when all events have been treated. Each event has a time, a type, and an attached connection or resource. We describe the simulator's logic by explaining how it deals with each event.

Several events trigger an attempt to "dispatch all resources", which checks all connections to see if any is available to dispatch any resources in the waiting queue, and creates new connections if allowed to (due to the rules of HTTP/2, pipelining, or connection limits). We mimic the rules in the browser regarding which connections to choose to dispatch on.

Resource events

- Resource created: Add the resource to the waiting queue for the relevant server, and dispatch all resources.
- Resource dispatched: This happens when a resource is successfully requested over a connection. Calculate how long it would take to load such a resource based on round trips required and bandwidth, declare the connection occupied, and create a resource completed event after that time. (Pipelining would alter this calculation.) Create new resource created events if this resource has any children in the resource tree after an appropriate time.
- Resource completed: Declare the relevant connection to be available for further dispatch, and mark the resource as complete with relevant time statistics. Dispatch all resources.

Connection events

- Connection created: Simulate TLS and ALPN handshakes if necessary, then declare TLS finished and ALPN finished after appropriate times.
- TLS finished: Dispatch all resources.
- ALPN finished: Mark the connection as allowing HTTP/2 from now on (instead of just HTTP/1.1).

2) How BLAST determines resource parenthood: It is necessary to know the parent of each resource to construct the resource tree, a crucial data structure used to analyze and simulate the page loading process. However, the web browser does not record or output the parent of each resource.

To determine the parent of a resource, BLAST uses browser logs to examine the context under which it was created as follows. We start by determining the parent candidates of every resource: the last resource that was written to before the examined resource was created, as well as any other resource written within $0.05\,\text{s}$ of that, is a parent candidate. We chose $0.05\,\text{s}$ heuristically because we observed that parsing time usually did not exceed this amount. Then, we set parents in three loops of all resources:

1) We mark all resources with only one parent candidate as having such a parent.
2) For the remaining resources, if only one of their parent candidates was chosen as a parent for some other resource in step 1, we mark that candidate as the parent.
3) For the remaining resources, we mark the last resource written to as the parent.

The final step ensures that all resources (except the first resource for each page, representing user action to load the page) will have a parent.