Understanding and Improving Web Page Load Times on Modern Networks

by

Ravi Arun Netravali

B.S. Electrical Engineering, Columbia University (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Author . . . . . .
Department of Electrical Engineering and Computer Science
December 19, 2014

Certified by . . . . . .
Hari Balakrishnan
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Understanding and Improving Web Page Load Times on Modern Networks

by

Ravi Arun Netravali

Submitted to the Department of Electrical Engineering and Computer Science
on December 19, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis first presents a measurement toolkit, Mahimahi, that records websites and replays them under emulated network conditions. Mahimahi improves on prior record-and-replay frameworks by emulating the multi-origin nature of Web pages, isolating its network traffic, and enabling evaluations of a larger set of target applications beyond browsers.

Using Mahimahi, we perform a case study comparing current multiplexing protocols, HTTP/1.1 and SPDY, and a protocol in development, QUIC, to a hypothetical optimal protocol. We find that all three protocols are significantly suboptimal and their gaps from the optimal only increase with higher link speeds and RTTs. The reason for these trends is the same for each protocol: inherent source-level dependencies between objects on a Web page and browser limits on the number of parallel flows lead to serialized HTTP requests and prevent links from being fully occupied.

To mitigate the effect of these dependencies, we built Cumulus, a user-deployable combination of a content-distribution network and a cloud browser that improves page load times when the user is at a significant delay from a Web page's servers. Cumulus contains a "Mini-CDN"—a transparent proxy running on the user's machine—and a "Puppet": a headless browser run by the user on a well-connected public cloud. When the user loads a Web page, the Mini-CDN forwards the user's request to the Puppet, which loads the entire page and pushes all of the page's objects to the Mini-CDN, which caches them locally.

Cumulus benefits from the finding that *dependency resolution*, the process of learning which objects make up a Web page, accounts for a considerable amount of user-perceived wait time. By moving this task to the Puppet, Cumulus can accelerate page loads without modifying existing Web browsers or servers. We find that on cellular, in-flight Wi-Fi, and transcontinental networks, Cumulus accelerated the page loads of Google's Chrome browser by 1.13–2.36×. Performance was 1.19–2.13× faster than Opera Turbo, and 0.99–1.66× faster than Chrome with Google's Data Compression Proxy.

Thesis Supervisor: Hari Balakrishnan
Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

I first thank my advisor, Hari Balakrishnan, for giving me unbounded flexibility in defining research problems. I have benefited immensely from his optimism, creativity, vision, and ability to fine-tune preliminary ideas into interesting problems.

I am especially grateful to my two labmates, Keith Winstein and Anirudh Sivaraman, for their willingness and patience in teaching me how to carefully build software systems and how to navigate through research problems from beginning to end.

I thank my labmates and members of the ninth floor—Katrina LaCurts, Raluca Ada Popa, Shuo Deng, Lenin Ravindranath, Peter Iannucci, Tiffany Chen, Amy Ousterhout, Jonathan Perry, Pratiksha Thaker, Ameesh Goyal, Somak Das, Eugene Wu, Alvin Cheung, Yuan Mei, Frank Wang, Guha Balakrishnan, Jen Gong, Amy Zhao, Hariharan Rahul, Cody Cutler, Anant Bhardwaj, and Sheila Marian—for always being available for discussions and for making Stata Center a terrific work environment.

Marcella Lusardi has been a constant source of support and happiness both academically and outside of work. I am truly lucky that I have been able to grow at MIT with my best friend.

Finally, I am forever grateful to my family for their unconditional support and guidance throughout my academic career along with their never-fading belief in me:

My sister, Ilka Netravali, for her ability to talk to me as an academic, sister, and friend, and for truly being an inspiration to me.

My mother, Chitra Netravali, for her unbounded desire to help me focus on my academics and for her unwavering love.

Lastly, my father, Arun Netravali, for being a terrific bouncing board for research ideas and for always teaching me the importance of perseverance in research and in life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the invention of the Web, much attention has been devoted to making access to websites faster for users. Over the past two decades, many methods have been designed and deployed to improve Web performance, including multiple concurrent TCP connections, persistent HTTP multiplexed over a small number of TCP connections, pipelining, prefetching, long polling, framing to multiplex concurrent HTTP streams over a single TCP connection, request prioritization, HTTP header compression, and server push (§2 describes prior work). Web delivery protocols like HTTP/1.1, SPDY, and QUIC incorporate one or more of these techniques.

How well do current techniques that aim to make the Web faster perform with different link speeds and propagation delays? This question is of interest to several stakeholders: network protocol designers who seek to understand the application-level impact of new multiplexing protocols like QUIC [48], website developers wishing to speed up access to their Web properties, and browser developers who need to evaluate how changes to their DOM and JavaScript parsers impact page load times.

Motivated by these diverse interests in reproducibly evaluating Web performance, we built **Mahimahi** (§3), a record-and-replay framework for reproducible Web measurements. Mahimahi is structured as a set of UNIX shells. RecordShell allows a user to record entire Web pages, and ReplayShell subsequently allows a user to replay them under emulated network conditions. For network emulation, Mahimahi includes two more shells: LinkShell emulates both fixed-capacity and variable-capacity links while DelayShell emulates a fixed

propagation delay.

Mahimahi makes several advances over current record-and-replay frameworks such as web-page-replay (§4). In particular, Mahimahi captures the multi-origin nature of Web pages today, where a website's resources are distributed across several physical servers. We show in §4.1 that this is crucial to accurate measurements of page load times on the Web: measurements with multi-origin preservation are 20% closer to measurements collected on the Web. Through the use of network namespaces, Mahimahi isolates its traffic from the rest of the host system allowing multiple instances of Mahimahi's shells to run in parallel with no effect on one another or on collected measurements. Lastly, by structuring its experimental environment as a set of shells, arbitrary processes such as emulators and virtual machines can be run within Mahimahi's shells, in contrast to prior approaches that work only with browsers.

We use Mahimahi to perform a detailed case study (§5) of the Web using several multiplexing protocols: HTTP/1.1, SPDY, and QUIC, a protocol in active development at Google. We evaluate these protocols on two metrics: the page load time that measures how long it takes a page to load after the user initiates the process and the Speed Index [26], an emerging metric that better captures the dynamic nature of the page load process. In both cases, we compare all three multiplexing protocols to a hypothetical optimal protocol and find that all three are significantly suboptimal over canonical network configurations. Further, this gap increases as link speeds increase, implying that more and more capacity is being wasted as link speeds improve (§6): page load times with SPDY are 1.63× worse than optimal over a 1 Mbit/sec link with 120 ms minimum RTT, but are 2.39× worse than optimal when the link speed increases to 25 Mbits/sec. Similarly, over the same two links, the Speed Indices of HTTP/1.1 are 1.52× and 3.63× worse than optimal, respectively. We find that these gaps from optimal are a result of source-level dependencies inherent to Web pages and browser constraints which lead to HTTP request serialization and unoccupied links. Due to the complexity of modern Web pages, the negative effect that HTTP request serialization has on page load times increases as RTTs increase.

An individual Web page can comprise hundreds of objects: embedded HTML frames, scripts, images, stylesheets, etc. For a Web browser, loading a page means first loading

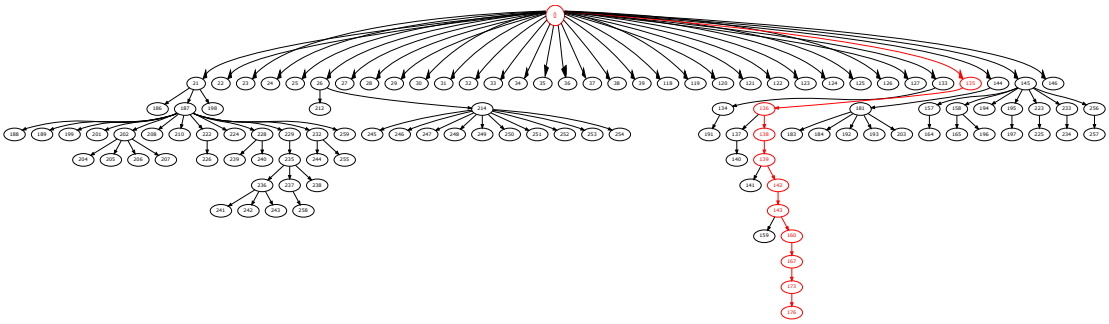


Figure 1-1: Part of the dependency tree for TMZ's homepage. The longest dependency chain of 10 objects is marked in red.

a root object, typically an HTML page whose URL is displayed in the title bar, and then recursively loading all of the objects that it refers to.

Each step of this process—fetching a referenced object, decoding it, and requesting the objects that it references—requires at least one round-trip time between the browser and the origin Web servers. For a popular contemporary Web page with a deep dependency tree (see, e.g., Figure 1-1), it can require 10 RTTs or more to load the page, regardless of the speed of the user's Internet connection.

The industry has expended considerable effort to enable site operators to reduce the duration of an RTT, the required number of RTTs to load a Web page, and the coded size of each object.

- Content-distribution networks place Web servers closer to users' browsers, reducing the duration of an RTT.

- Google's SPDY protocol allows a server-side Web application to speculatively "push" referenced objects alongside the object that refers to them [1], when the reference is made to another object served by the same origin Web server.

- "Cloud" browsers, such as Opera Turbo [2] and Chrome's Data Compression Proxy [3], compress Web objects before sending them to the user.

Many users nonetheless receive poor Web performance. For users on cellular connections or on aircraft Wi-Fi, it is the user's "last mile" access link that has a large RTT. More

than half of the world lives in developing regions where CDN coverage is sparse. Our measurements find that commercially-deployed CDNs have difficulty getting "close enough" to these types of users, and cloud browsers also have considerable room for improvement (§9).

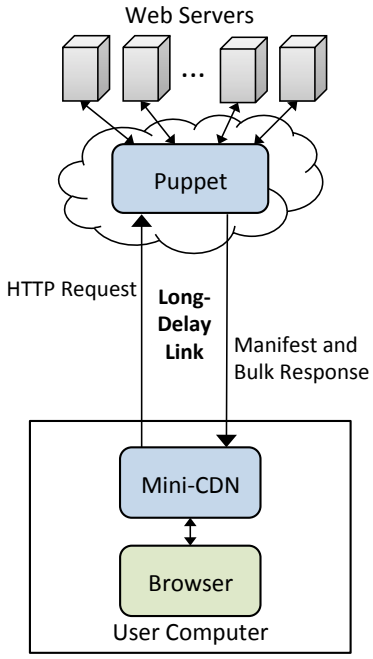


Figure 1-2: Cumulus uses a Puppet running a headless browser and a Mini-CDN to reduce page load times.

To address these issues, we built **Cumulus**, a user-deployable combination of a CDN and a cloud browser. Cumulus reduces the user-experienced page load times of popular Web pages in a variety of real-world settings where the user is at a significant delay from the page's Web servers.

Cumulus has two components: the "Puppet," a headless browser that the user runs on a well-provisioned public cloud server, and the "Mini-CDN"—a transparent, caching HTTP proxy that runs on the user's computer (Figure 1-2). These two components cooperate together to reduce page load times.

When the user's browser requests a URL that is not resident in the Mini-CDN's cache, the Mini-CDN forwards the request to the well-provisioned "Puppet" browser. This browser recursively fetches the root object identified by the URL and prefetches the referenced objects on the same page, and sends them back to the Mini-CDN, which holds them until requested by the browser.

Network	Approx. RTT	Speedup vs. Google Chrome	vs. Chrome Proxy	vs. Opera Turbo
AT&T cellular network	100 ms	1.38×	1.30×	1.35×
In-flight Wi-Fi	1000 ms	2.36×	1.66×	2.13×
São Paulo to U.S.	170 ms	1.77×	1.31×	1.67×
Boston to Australia	230 ms	1.61×	1.32×	1.44×
Boston to Japan	200 ms	1.60×	1.33×	1.54×
Boston to Singapore	210 ms	1.45×	1.26×	1.35×
Boston to Brazil	170 ms	1.46×	1.34×	1.44×
Boston to Ireland	80 ms	1.13×	0.99×	1.19×

Table 1.1: Median speedups achieved by Cumulus (with Google Chrome), compared with Google Chrome by itself, Google Chrome Data Compression Proxy, and Opera Turbo.

Cumulus reduces the user's page load times by moving the resolution of object references closer to the origin Web servers—reducing the effective RTT—without modifying the Web browser or servers. We find (Table 1.1) that Cumulus outperforms Google Chrome, Opera Turbo, and a plugin that allows Google Chrome to interoperate with Google's Data Compression proxy on several real-world experiments including a cellular link (§9.1), in-flight Wi-Fi (§9.2), and wide-area scenarios (§9.3 and §9.4).

Cumulus's greatest improvements are on in-flight Wi-Fi, where RTTs of 1 second are common, and on cellular links, where RTTs routinely exceed 100 ms. Cumulus also performs well in wide-area scenarios when no CDN node is located nearby.

In summary, the second part of this thesis makes three contributions through the design, implementation, and experimental evaluation of Cumulus.

1. It identifies *dependency resolution* over a shorter-delay link as an important factor in improving Web performance for users with a significant delay to Web servers.
2. It exploits the strategy of server-side push, without modifying server-side Web applications, by using the Puppet to identify the objects required to load a page and pushing them to the Mini-CDN in advance of a browser's request.
3. Lastly, it does both with no modifications to existing browsers or servers.

We have open-sourced Mahimahi (`http://mahimahi.mit.edu/`) and Cumulus (`http://web.mit.edu/anirudh/cumulus`) along with all of the data and experimental results presented in this thesis.

Chapter 2

Related Work

2.1 Record-and-Replay Tools

The most prominent Web page record-and-replay tool is Google's web-page-replay [28]. web-page-replay uses DNS indirection for both recording and replaying Web content. To record, web-page-replay responds to all DNS requests with the address of a local HTTP proxy server. All HTTP requests from a browser are sent to this proxy server which records the request and forwards it to the corresponding origin server on the Web. Responses also pass through the proxy server and are recorded and sent back to the browser. Replaying is done with the HTTP proxy server, which has access to the recorded content, responding to each incoming HTTP request without forwarding it to the origin server. However, web-page-replay suffers from three primary shortcomings:

1. All HTTP responses are served from a single HTTP proxy server and thus, web-page-replay does not preserve the multi-origin nature of Web pages. Consolidation of HTTP resources onto a single server during emulation allows browsers to use a single flow to fetch all resources — something that is impossible when resources are on different servers. Additionally, browser constraints, such as the maximum number of connections per server, degrade performance when there is only one server. Mahimahi accurately preserves the multi-origin nature of Web pages, leading to more accurate measurements (§4.1).
2. web-page-replay does not provide isolation: the virtual network conditions that web-

page-replay creates (link speed, one-way delay, and DNS indirection) affect all other processes on the host machine. During replay, lack of isolation leads to inaccurate measurements when cross traffic from other processes reaches web-page-replay's proxy server. This occurs because DNS indirection rules affect all processes on the host machine. The lack of isolation also precludes multiple independent instances of web-page-replay from running concurrently — a useful feature for performance regression tests and for expediting evaluations. Mahimahi remedies these problems by using the network namespace feature of Linux [4] to provide isolation.

3. Finally, most record-and-replay tools, including web-page-replay, only operate with browsers. By providing a shell as the emulation environment, Mahimahi enables the recording and replaying of any HTTP content, from browsers to virtual machines and mobile device emulators.

2.2 Network Emulation

Tools such as dummynet [33] and netem [40] emulate network conditions such as link speed, one-way delay, and stochastic loss. web-page-replay, for instance, uses dummynet for network emulation. Mahimahi uses its own network emulation tools (LinkShell and DelayShell), based on the design of cellsim [54]. In addition to static link speeds, LinkShell can also emulate variable-rate cellular links using packet delivery traces. Additionally, Mahimahi allows a user to evaluate new in-network algorithms (instead of Drop Tail FIFO) by modifying the source code of LinkShell. A similar evaluation using web-page-replay would require developing a new kernel module—a more error-prone task.

2.3 Enhancing HTTP and TCP for the Web

Much progress has been made on modifying the HTTP protocol to improve Web performance. Most notably, persistent connections [44] enable a single TCP connection to carry several HTTP requests and responses, while pipelining [5] allows multiple HTTP requests to be sent without waiting for the corresponding responses. TCP has also been modified

to support better performance on Web-like workloads, including increasing the initial window [34], sending data within the TCP SYN packet [45], and better TCP loss recovery algorithms [37].

We do not propose modifications to HTTP or TCP. We instead highlight the limitations of both HTTP/1.1 and SPDY, run with TCP at the transport layer, in the current HTTP architecture. We do note that each of these modifications to HTTP and TCP can be used within Cumulus.

2.4 Multiplexing Protocols and Server Push

Starting with work on persistent HTTP connections, researchers have grappled with the problem of mapping HTTP requests and responses to the underlying tranport. Older work includes the Stream Control Tranmission Protocol (SCTP) [27] that allows multiple distinct streams (which could correspond to images and text in the Web context) to be multiplexed on a single SCTP association. Structured Stream Transport [38] also allows different streams to be multiplexed onto one network connection and operates heirarchically by allowing streams to spawn streams of their own.

Deployed work in this space includes Google's SPDY [25] [6]. SPDY incorporates multiplexed streams, request prioritization, and header compression in an effort to reduce Web latency. Additionally, both SPDY and the proposed HTTP/2.0 [7] standard allow a server-side Web application to speculatively "push" objects that belong to the same origin server before the user requests them. SPDY leaves the control of whether and when to push an object up to each server-side application.

Wang et al. [52] analyze the performance of Apache's SPDY implementation and present a server-push policy, based on inferred object dependencies, that attempts to push only objects that will truly be required by the user.

Cumulus's Puppet takes dependency-based pushing to its logical extreme by running a headless browser that follows dependencies as a real Web browser would load a page: recursively, until there are no more dangling references. The Puppet sends the Mini-CDN exactly those resources required for a page load.

The Puppet provides a site-agnostic push policy that requires no modification to server-side scripts. Further, Cumulus's Puppet bundles and pushes objects from multiple origin servers because it is trusted by the user to load the entire page on the user's behalf. This is in contrast to SPDY's server push, which allows a server only to push objects belonging to the same server.

More recently, Google's proposal, QUIC [48], is a transport layer protocol which runs over UDP. QUIC attempts to reduce the number of incurred RTTs during page loads by allowing clients to send data without having to wait for the completion of a handshake. QUIC also provides packet-level error correction codes to reduce retransmissions.

2.5 Measurement Studies

Many previous studies compare HTTP/1.1 with SPDY because they are the two most commonly used Web application protocols today. Results on SPDY's benefits, however are inconclusive. In [43], Padhye et al. show that HTTP/1.1 can be configured to have performance similar to SPDY by using pipelining, while a study at Cable Labs [22] shows that SPDY can either improve page load time or drastically increase it. In [35], the authors analyze the performance of HTTP/1.1 and SPDY over cellular networks and find that both perform poorly due to TCP's interactions with cellular-radio idle states. Unlike prior work, we focus on comparing all candidate protocols to a hypothetical optimal protocol.

WProf [51] studies the interaction between browser computation and network transmissions in the context of Web page loads, and finds that browser computation constitutes a significant portion of page load time for average pages. We find that whether computation is significant or not is dependent on the speed of the network over which the page is being served. At lower link speeds, the computation is faster than the network, keeping the links almost always occupied, resulting in the networking time being a higher fraction of the page load time. At much higher link speeds, exactly the opposite is true. Furthermore, the suboptimal performance of HTTP/1.1 and SPDY isn't tied to the finite computation times in browsers. Even assuming zero computation time between a response and a subsequent request, the dependency chains inherent in the page load process prevent both HTTP/1.1

and SPDY from fully utilizing the link because the list of all requests is not known upfront.

The authors of WProf also note that reducing the time spent on the critical path (i.e. the longest dependency chain) is more important than compressing objects using extensions such as mod_pagespeed [8]. Our results with Cumulus corroborate this finding. We find that moving dependency resolution to a link with a shorter delay shrinks the critical path and improves total page load time.

2.6 Cloud Browsers

Closest to Cumulus are cloud browsers, which also divide the page-load process between the user's Web browser and a remote component.

Opera Turbo [2] and Chrome Data Compression Proxy [3] use a proxy server—operated by Opera or Google—to compress objects before sending them to the user.

Cumulus uses the Puppet (the headless proxy browser in the cloud) for dependency resolution, not compression. Because the Puppet is run by the user as a well-connected proxy browser, it has access to all HTTP and HTTPS objects and can perform dependency resolution over an entire Web page. Opera Turbo and Chrome Proxy use centrally-operated proxy servers, and because it would invoke serious privacy concerns to allow either Opera or Google to observe the plaintext of users' encrypted traffic, neither system attempts to compress resources retrieved over HTTPS connections.

The older Opera Mini [9] and MarioNet [10] systems perform remote dependency resolution by using a proxy browser to load and compress a page before sending it to a mobile browser on a user's phone. Amazon Silk [11] is also said to split the page load process between a component in the cloud and one on the user's device [53].

These three systems run on specific mobile platforms and are integrated with a client Web browser. Cumulus, by contrast, works without modifications to existing browsers. A head-to-head comparison of Cumulus with these latter systems would be desirable, but we were not able to achieve timed measurements of these proprietary mobile browsers within the scope of the current work.

Sivakumar et al. [49] analyze the performance of cloud browsers and find that cloud

browsers do not always improve page load times, especially for simple Web pages that contain little JavaScript. We replicate this finding in Figure 10-1.

2.7 Content-Distribution Networks

Content-distribution networks [39, 50, 42, 12] are an effective means of reducing the RTT to clients by placing content servers at locations close to clients. However, it is challenging to place a CDN node close to the user when the user's last-mile access link has a long delay, as is the case with a cellular link or in-flight Wi-Fi.

For a particular website, the ultimate effectiveness of a CDN hinges on every origin server's making its content available via a CDN close to the user. Cumulus is an application-layer prefetching strategy that works across multiple origin servers.

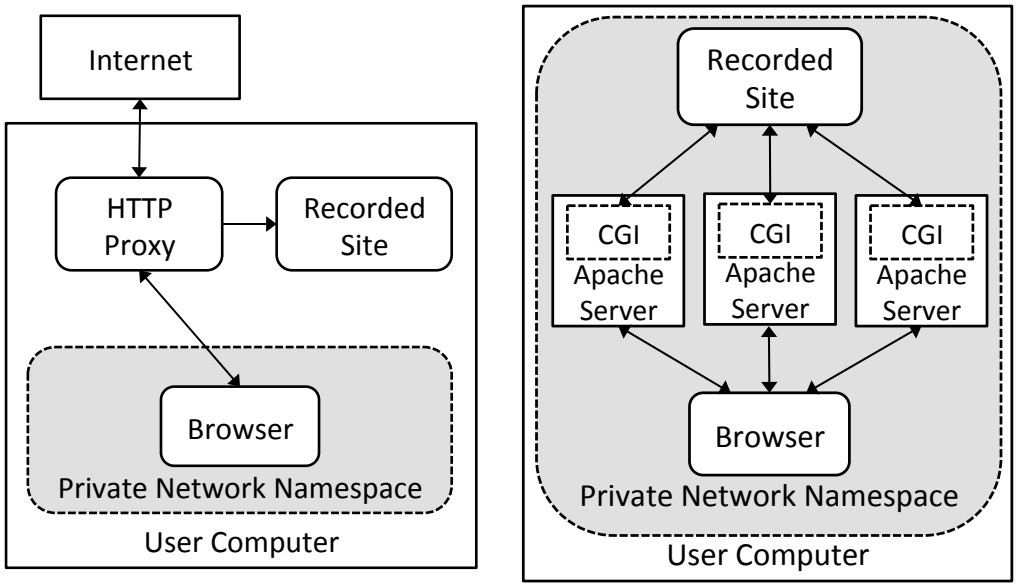
Chapter 3

Design of Mahimahi

The design of Mahimahi was in part informed by the issues plaguing existing record-and-replay frameworks, particularly web-page-replay: accurate emulation of multi-origin Web pages, isolation of the record-and-replay framework from other processes running on the same machine, and flexibility to evaluate applications beyond browsers.

Mahimahi is structured as a set of UNIX shells. Each shell creates a new network namespace for itself prior to launching the shell. Quoting from the man page, "a network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices" [4]. A separate network namespace minimizes disruption to the host machine during recording, prevents accidental download of resources over the Internet during replay, and ensures that the host machine is isolated from all network configuration changes (emulating delays and link speeds and modifying DNS entries) that are required to evaluate an application.

RecordShell (§3.1) records all HTTP traffic for subsequent replay. *ReplayShell* (§3.2) replays previously recorded HTTP content. *DelayShell* (§3.3) delays all packets originating from the shell by a user-specified amount and *LinkShell* (§3.4) emulates a network link by delivering packets according to a user-specified packet-delivery trace. All components of Mahimahi run on a single physical machine (which we call the host machine) and can be arbitrarily composed with one another (§4.2). Mahimahi is lightweight (§3.6) and adds low overhead to the page load times we report. We use Mahimahi to create a corpus of Web page recordings of the Alexa US Top 500 (§3.5) which we use in our case study of the Web.



(a) RecordShell (b) ReplayShell

Figure 3-1: RecordShell has a transparent proxy for HTTP traffic. ReplayShell handles all HTTP traffic inside a private network namespace. Arrows indicate direction of HTTP Request and Response traffic.

3.1 RecordShell

RecordShell (Figure 3-1a) records HTTP data sent during actual page loads and stores it on disk in a structured format for subsequent replay. On startup, RecordShell spawns a man-in-the-middle proxy on the host machine to store and forward all HTTP traffic both to and from an application running within RecordShell. To operate transparently, RecordShell adds an iptable rule that forwards all TCP traffic from within RecordShell to the man-in-the-middle proxy. When an application inside RecordShell attempts to connect to an origin server, it connects to the proxy instead. The proxy then establishes a TCP connection with the application, uses the `SO_ORIGINAL_DST` socket option to determine the origin server's address for the connection, and connects to the origin server on the application's behalf. An HTTP parser running at the proxy parses traffic passing through it to parse HTTP requests and responses from TCP segments. Once an HTTP request and its corresponding response have both been parsed, the proxy writes them to disk in a structured format associating the request with the response.

SSL traffic is handled similarly by splitting the SSL connection and establishing two separate SSL connections: one between the proxy and the application and another between the proxy and the origin server. To establish a connection with the application, the proxy creates a fake self-signed certificate with the common name "Mahimahi." Intercepting secure connections in this manner causes two problems. First, the common name on the certificate presented to the browser ("Mahimahi") differs from the common name expected by the browser (typically the hostname that the browser is attempting to connect to). Second, browsers only accept certificates signed by any one of a list of trusted Certificate Authorities. This triggers a warning within the browser when attempting to use HTTPS, which the user must dismiss to proceed. As discussed in §5.3, these warnings can be disabled with most modern browsers.

At the end of a page load, a recorded folder consists of a set of files: one for each request-response pair seen during that browser session. RecordShell is compatible with any unmodified browser because recording is done transparently, without the browser knowing of the existence of a proxy.

3.2 ReplayShell

ReplayShell (Figure 3-1b) also runs on the host machine and mirrors a website using solely content recorded by RecordShell. ReplayShell accurately emulates the multi-origin nature of most websites today by spawning an Apache 2.2.22 Web server for each distinct IP/port pair seen while recording.

To operate transparently, ReplayShell binds its Apache Web servers to the same IP address and port number as their recorded counterparts. To do so, ReplayShell creates a separate dummy interface[1] for each distinct server IP. These interfaces can have arbitrary IPs because they are in a separate network namespace. All browser requests are handled by one of ReplayShell's servers, each of which can access the entire recorded content for the site. The Apache configuration redirects all incoming requests to a CGI script on the appropriate server using Apache's `mod_rewrite` module. The CGI script running on

[1] A dummy interface is the term for a virtual interface in Linux.

each Web server compares the incoming HTTP request to the set of all recorded request-response pairs to locate a matching HTTP response. This comparison is complicated by the presence of time-sensitive header fields (e.g., If-Modified-Since, If-Unmodified-Since) and time-sensitive query string parameters. To handle both cases, we execute the following algorithm on every incoming request:

1. Parse the Request-URI on the incoming request to determine if it contains a query string.
2. If it does not have a query string:
   (a) Find a request-response pair whose resource name exactly matches that of the incoming request.
   (b) Check whether the request in this request-response pair matches the incoming request for the Host, Accept-Language, Accept-Encoding, Connection, and Referer HTTP header fields. If so, return the corresponding response. If not, continue searching through stored request-response pairs.
3. If it does have a query string:
   (a) Find all request-response pairs with the same requested resource name as the incoming request.
   (b) For each, check whether the request-response pair matches the incoming request for the Host, Accept-Language, Accept-Encoding, Connection, and Referer HTTP header fields. If so, add the request-response pair to the list of potential matches.
   (c) From the list of potential matches, find the pair whose request query string shares the longest common substring with the incoming query string and return the corresponding response.

The Apache Web servers are configured to speak either HTTP/1.1 or SPDY (using `mod_spdy` 0.9.3.3-386) allowing us to compare both with the same setup. We use Apache's `mod_ssl` module to handle HTTPS traffic.

3.3 DelayShell

DelayShell emulates a link with a fixed minimum one-way delay. All packets to and from an application running inside DelayShell are stored in a packet queue. A separate queue is maintained for packets traversing the link in each direction. When a packet arrives, it is assigned a delivery time which is the sum of its arrival time and the user-specified one-way delay. Packets are released from the queue at their delivery time. This technique enforces a fixed delay on a per-packet basis.

3.4 LinkShell

LinkShell is used to emulate a link using packet-delivery traces for that link. It is flexible enough to emulate both time-varying links such as cellular links and links with a fixed link speed. When a packet arrives into the link, it is directly placed into either the uplink or downlink packet queue. LinkShell is trace-driven and releases packets from each queue based on the corresponding packet-delivery trace. Each line in the trace is a packet-delivery opportunity: the time at which an MTU-sized packet will be delivered in the emulation. Accounting is done at the byte-level, and each delivery opportunity represents the ability to deliver 1500 bytes. Thus, a single line in the trace file can correspond to the delivery of several packets whose sizes sum to 1500 bytes. Delivery opportunities are wasted if bytes are unavailable at the instant of the opportunity.

3.5 Benchmark Corpus

Using RecordShell, we created a corpus (`https://github.com/ravinet/sites`) of the 500 most popular Web pages in the United States [13]. To record each Web page, we navigate to each website's homepage using Google Chrome Version 31.0.1650.63 with the --incognito flag to avoid loading resources from the cache. To ensure experimental flexibility, we verified that each recorded Web page can be completely loaded when using either Google Chrome, Mozilla Firefox, or Internet Explorer, inside ReplayShell. In addi-

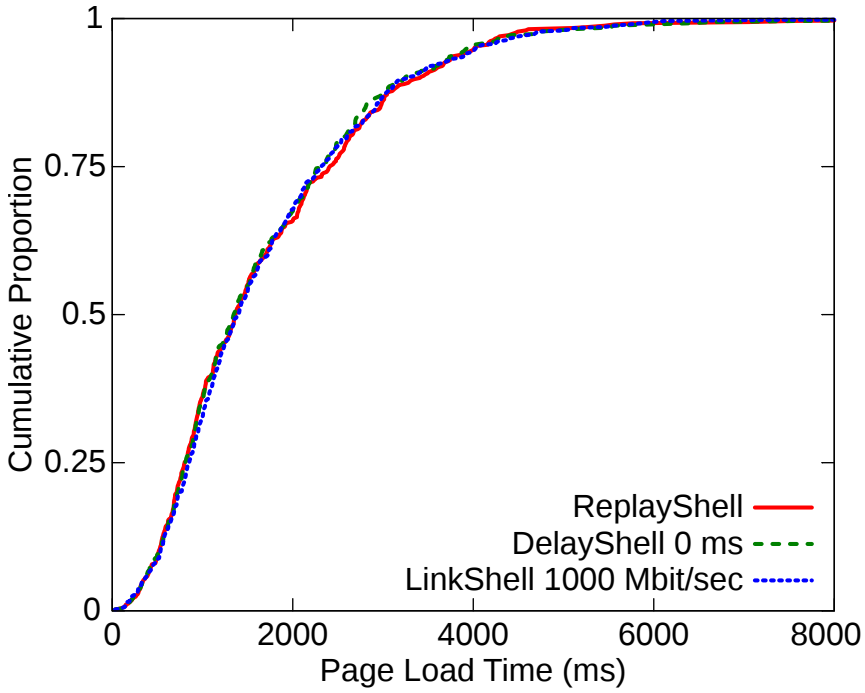


Figure 3-2: DelayShell and LinkShell minimally affect page load times in ReplayShell

tion to the recorded Web pages, the corpus also provides results discussed in Section §5 as benchmark measurements for future development. These measurements include page load times, recorded over more than one hundred network configurations, for each recorded site when using different multiplexing protocols. Lastly, the corpus provides 3G and 4G cellular network traces for Verizon, AT&T, T-Mobile, and Sprint. These traces were originally collected in [54] and are modified to be compatible with LinkShell.

3.6 Low Overhead

Mahimahi imposes low overhead on page load time measurements (§5.3.2). We illustrate this in Figure 3-2 which shows the overhead DelayShell and LinkShell impose on page load time measurements. We first run ReplayShell on our corpus of recorded sites without LinkShell or DelayShell. For comparison, we then run DelayShell, with 0 ms fixed per-packet delay, inside ReplayShell. Separately, we run LinkShell, with 1000 Mbits/sec uplink and downlink traces, inside ReplayShell. We chose 1000 Mbits/sec to ensure that link capacity was not a limiting factor in page load time and to mimic the link capacity seen

	Machine 1	Machine 2
CNBC	7584 ms, 120 ms	7612 ms, 111 ms
wikiHow	4804 ms, 37 ms	4800 ms, 37 ms

Table 3.1: Mean, standard deviation for page load times on similarly configured machines

when using ReplayShell without LinkShell. Each of these experiments was performed on the same Amazon EC2 m3.large instance configured with Ubuntu 13.10 and located in the US-east-1a region. We find that the difference in median page load time with vs. without DelayShell is only 0.15%. Similarly, we find that the difference with vs. without LinkShell is only 1.5%.

3.7 Reproducibility

We investigate whether Mahimahi produces reproducible results for repeated experiments performed on the same host machine and across different host machines with similar hardware specifications. We choose two sites from our corpus for this experiment, `http://www.cnbc.com/` and `http://www.wikihow.com/`, as they are are at the median and 95th percentile of site sizes (1.2 MB and 5.5 MB, respectively).

We use two different Amazon EC2 m3.large instances, each in the US-east-1a region and running Ubuntu 13.10. On each machine, we load the CNBC and wikiHow Web pages 100 times each inside ReplayShell, over a 14 Mbits/sec link with minimum RTT of 120 ms. Table 3.1 shows a summary of the distribution of page load times from these experiments. To evaluate Mahimahi's reproducibility across machines, we consider the mean page load times for each site. The mean page load times for each site are less than 0.5% apart across the two machines suggesting that Mahimahi produces comparable results across different host machines. Similarly, to evaluate Mahimahi's reproducibility on a single machine, we consider the standard deviation in page load times for each site. The standard deviations are each within 1.6% of their corresponding means. This suggests that Mahimahi produces consistent results for the same experiment on a single host machine or similarly configured host machines.

Chapter 4

Novelty of Mahimahi

4.1 Multi-origin Web pages

A key component of ReplayShell is its preservation of the multi-origin nature of Web pages. As discussed in §3, to preserve a page's multi-origin nature, ReplayShell creates a network namespace containing an Apache server for each distinct Web server encountered in a recorded folder. We show through three experiments (Figure 4-1, Table 4.1, and Figure 4-2) that preserving this multi-origin nature is critical to the accurate measurement of page load times.

We first show that a non-trivial amount of websites today are multi-origin. Specifically, we evaluate the number of physical servers used by Alexa top 500 Web pages. We collect this data by recording, per Web page, the number of distinct IP addresses for servers listed in our recorded folders. Figure 4-1 shows the distribution of the number of physical servers per Web page. The median number of servers is 20 while the 95th percentile is 51 and the 99th percentile is 58. Only 9 of the 500 Web pages we consider use a single server; web-page-replay can thus only accurately represent 1.8% of the top 500 Web pages in the US.

Next, we show the effect that the multi-origin nature has on page load times across different network conditions and Web pages. Using an Amazon EC2 m3.large instance located in the US-east-1a region and running Ubuntu 13.10, we record page load times for each page in our recorded corpus when loaded with Google Chrome. We consider 9

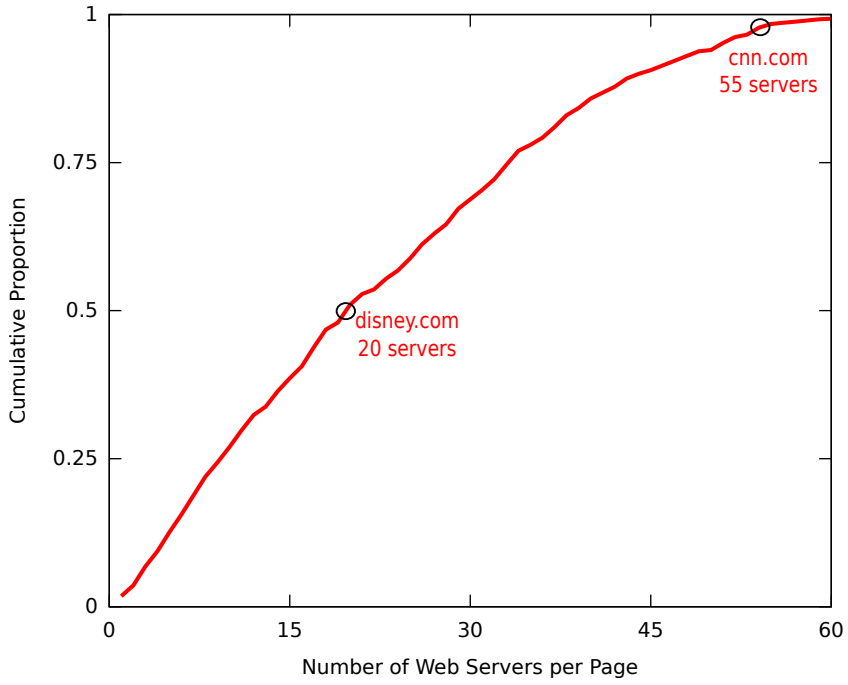


Figure 4-1: Approximately 98% of the sites we recorded use more than a single server which increases the importance of preserving a Web page's multi-origin nature

	30 ms	120 ms	300 ms
1 Mbit/sec	1.6%, 27.6%	1.7%, 10.8%	2.1%, 9.7%
14 Mbits/sec	19.3%, 127.3%	6.2%, 42.4%	3.3%, 20.3%
25 Mbits/sec	21.4%, 111.6%	6.3%, 51.8%	2.6%, 15.0%

Table 4.1: Median and 95th percentile error without multi-origin preservation

different configurations spanning a link speed range of 1 Mbit/sec to 25 Mbits/sec and a minimum RTT range of 30 ms to 300 ms. We load each page over each configuration using both ReplayShell and a modified version of ReplayShell that eliminates the multi-origin nature altogether by setting up one Apache server to respond to all requests and resolving all DNS addresses to that server alone. Table 4.1 shows the median and 95th percentile error in page load time when the multi-origin nature is not preserved. While the page load times are comparable over a 1 Mbit/sec link, lack of preservation often yields significantly worse performance at higher link speeds.

Finally, we illustrate the importance of preserving the multi-origin nature of websites by comparing measurements collected using ReplayShell to real page load times on the Web.

To obtain measurements on the Web, we use Selenium to automate Google Chrome loading the New York Times homepage, `http://www.nytimes.com/`, 100 times. Before each page load, we record the round trip time to the origin New York Times Web server. For a fair comparison, we recorded a copy of the New York Times homepage immediately following the completion of these Web measurements; Web content can change frequently, which can significantly affect page load time. We use this recorded folder to measure page load times with Mahimahi. Using ReplayShell, we measure 100 page load times for the New York Times homepage. We perform each page load inside DelayShell with a minimum RTT set to the corresponding recorded round trip time from the real Web measurements. We do not emulate a finite-capacity link for these experiments because the corresponding measurements on the real Web are run from a desktop at MIT that has a 100+ Mbits/sec connection to the Internet. We assume that the minimum round trip time to most Web servers contacted while loading the New York Times homepage is similar to the recorded minimum round trip time to nytimes.com, as they are likely in close proximity to the same Web property. We then load the New York Times homepage over both an unmodified ReplayShell and the modified ReplayShell described earlier that serves all resources from a single server.

Figure 4-2 shows the distribution of page load times for each of these experiments. As shown, ReplayShell with multi-origin preservation yields page load times which more accurately resemble page load times collected on the Internet. The median page load time is 7.9% larger than the Internet measurements, as compared to the 29.6% discrepancy when the multi-origin nature is not preserved.

The inaccuracy in page load times without multi-origin preservation is likely because browsers impose a limitation on the maximum number of connections that can be concurrently open to a single origin server. When eliminating the multi-origin nature of websites, all requests must be made to a single origin server, limiting the total number of TCP connections used to fetch objects and degrading performance in the process.

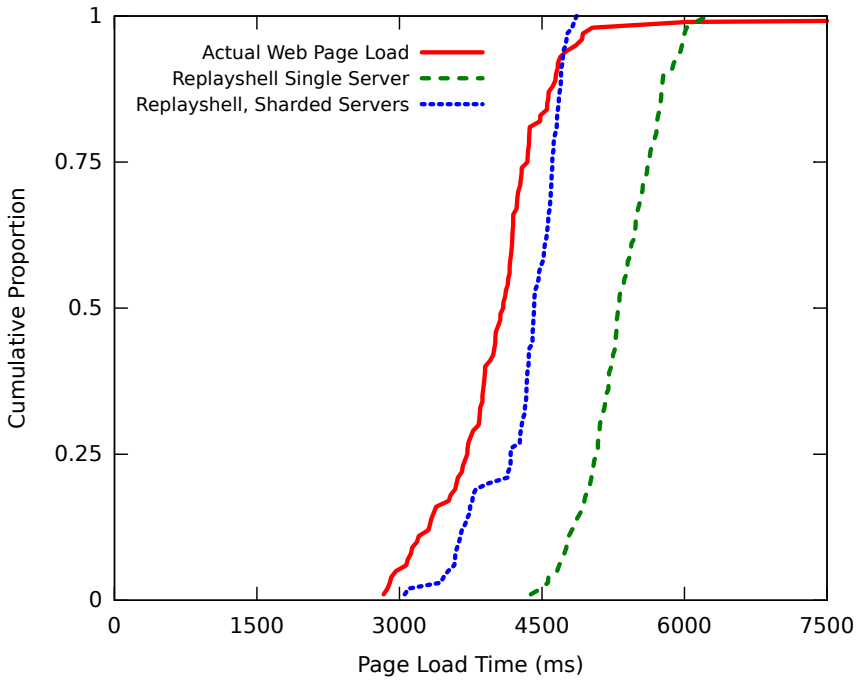


Figure 4-2: Preserving a Web page's multi-origin nature yields measurements which more closely resembles measurements on the Internet

4.2 Composability

ReplayShell, DelayShell, and LinkShell are designed such that they can be nested multiple times in any order. For example, to emulate a page load over a 4G cellular network with 10ms minimum RTT, one would run ReplayShell followed by LinkShell inside ReplayShell, followed by DelayShell inside LinkShell. This composability is enabled by two features of Mahimahi: network isolation, using network namespaces, that isolates each shell from the host machine and the use of the shell interface that naturally nests with other shells.

Composability is useful for future extensions to Mahimahi. For instance, consider a scenario where the program to be evaluated is only available on a different physical machine not running Linux. Let's call this machine *M*. We assume a separate Linux machine *A* is available. An *EthShell* can now ferry packets from an Ethernet interface between *M* and *A* to a virtual network interface on *A*. Analogously, a different shell, *UsbShell* can ferry packets between an Ethernet-over-USB interface connected to a phone and a virtual interface on *A*. UsbShell could be used to run performance regression tests on actual phones

without resorting to emulators. Neither of these has been developed yet, but Mahimahi's composability allows these shells to be nested inside any of Mahimahi's existing shells. For instance, to test Android's browser over a 4G link with a 10 ms RTT, we would nest UsbShell inside DelayShell inside LinkShell inside ReplayShell.

4.3 Isolation

By creating a new network namespace for each shell, Mahimahi eliminates much experimental variability that results from interfering cross traffic during an experiment. Each namespace is separate from the host machine's default namespace and every other namespace and thus, processes run inside the namespace of a Mahimahi tool are completely isolated from those run outside. As a result, host machine traffic does not affect the measurements reported by Mahimahi. Similarly, network emulation done by Mahimahi's tools does not affect traffic outside of Mahimahi's network namespaces. This property of Mahimahi, along with its composability, enables many different configurations to be tested, simultaneously, on the same host machine, and in complete isolation from one another.

4.4 Beyond Browsers

While most existing record-and-replay frameworks only replay browser page loads, Mahimahi's design allows it replay any application that uses HTTP. For instance, a mobile device emulator, such as the Android Emulator [23], can be used to analyze and measure mobile application performance or for automated fault testing [46]. Similarly, as described in §5, to measure Speed Index, virtual machines can be run within Mahimahi's shells.

4.5 Scope and Limitations of Mahimahi

Mahimahi's record-and-replay framework is general enough to reproducibly record and replay almost any client-server application, provided that its matching algorithm is modified to mimic the logic that the server executes to respond to client requests. The algorithm in

§3.2 is just one example of this. Our extension to handle Facebook (`https://github.com/ravinet/mahimahi/releases/tag/old%2Ffacebook`) provides probative evidence of this generality. Here, we modify the matching algorithm for the HTTP header Cookie field, which is used to load personalized content for a user. Previously, we disregarded this header field as we only considered landing pages without any personalization.

Mahimahi is, however, limited in that it only emulates one physical client (although a client can run an arbitrary number of processes) connected to an arbitrary number of servers. Specifically, there is only a single shared link from the client to all servers. We have extended Mahimahi to support multihomed clients (`https://github.com/ravinet/mahimahi/releases/tag/old%2Fmpshell_scripted`) to evaluate multipath-capable transport protocols such as MPTCP [24], but Mahimahi cannot emulate arbitrary network topologies such as transit-stub [32]. For client-server applications over those network topologies or for peer-to-peer applications, a framework like Mininet [41] is more suitable.

Chapter 5

Evaluation of Multiplexing Protocols for the Web

We use Mahimahi to evaluate Web page loads under common multiplexing protocols, HTTP/1.1 and SPDY, and an emerging protocol, QUIC, currently in development at Google. We describe our experimental setup to emulate network conditions in (§5.1), followed by a description of each protocol's specific setup (§5.2). We then evaluate the multiplexing protocols on two metrics: page load time (§5.3) and Speed Index (§5.4). In both cases, we define the metric, describe a procedure to measure it, calculate an optimal value of the metric given a network configuration, and compare all evaluated multiplexing protocols with this optimal value. Our goal is not to compare different protocols with each other and explain why one is better or worse. Rather, we wish to compare all protocols with an optimal value to understand shortcomings that are universal.

In all evaluations, traffic originates from the Web browser alone. We specify link speeds and minimum RTTs for each network configuration but do not emulate competing cross traffic. This evaluates the best case performance of all multiplexing protocols. Any slowdown in a metric relative to the optimal value will only worsen when competing traffic shrinks available link capacity.

5.1 Emulating Network Conditions

To emulate page loads over specific network conditions, we use ReplayShell in combination with LinkShell and DelayShell. For instance, to emulate a page load over a 1 Mbit/sec link with 150 ms minimum RTT:

1. We first run ReplayShell to setup a recorded website for replay.
2. Within ReplayShell, we run DelayShell with a one-way delay of 75 ms.
3. Within DelayShell, we run LinkShell with a 1 Mbit/sec packet-delivery trace.
4. Within LinkShell, we run an unmodified browser as we would within any other shell.

For each network configuration, we emulate a finite buffer size of 1 Bandwidth-Delay product and evaluate all Web pages stored in our corpus.

5.2 Experimental Setup for Each Multiplexing Protocol

5.2.1 HTTP/1.1

We evaluate HTTP/1.1 using ReplayShell running Apache 2.2.22. No further modifications are required.

5.2.2 SPDY

To evaluate SPDY, we enable the `mod_spdy` extension on all Apache servers within ReplayShell. The SPDY configuration evaluated here does not have any server push because there is no canonical implementation in Apache. We considered pushing objects up to a configured "recursion level" as proposed earlier, but decided against it because it may send more resources than is actually required by the browser. A recent paper [52] corroborates this intuition; it shows that the benefits of SPDY under such a server push policy are slight.

5.2.3 QUIC

QUIC (Quick UDP Internet Connections) is Google's new multiplexing protocol that runs over UDP [48], attempting to further improve Web performance compared with SPDY. It

inherits SPDY's features, including multiplexing streams onto a single transport-protocol connection with priorities between streams.

QUIC solves two drawbacks SPDY has due to its reliance on TCP. First, since SPDY's streams run over the same TCP connection, they are subject to head-of-line blocking—one stream's delayed or lost packet holds up all other streams due to TCP's in-order delivery requirement. QUIC avoids this issue by using UDP as its transport, allowing it to make progress on other streams when a packet is lost on one stream. Second, TCP today starts with a three-way handshake. Paired with HTTPS' Transport Layer Security (TLS) protocol handshake, several RTTs are wasted in connection setup before any useful data is sent. QUIC aims to be 0-RTT by replacing TCP and TLS with UDP and QUIC's own security. The initial packet from client to server establishes both connection and security context, instead of doing one after another. It also allows the client to optimistically send data to the server before the connection is fully established and supports session resumption. QUIC uses TCP Cubic as its default congestion control today, although this can be changed because QUIC supports a pluggable congestion-control architecture. QUIC also includes packet-level FEC to recover quickly from losses.

Chrome is the only browser to have experimental support for QUIC and the only publicly available websites that support QUIC are Google's properties which include Alternate-Protocol: 80:quic in their response headers. Our evaluation extends QUIC to a much wider corpus of websites (§3.5).

In evaluating QUIC, we faced a few challenges. Unlike SPDY, Apache has no extension for QUIC. So, we adapted a recent version of the QUIC toy server [47] from the Chromium project (commit `a8f23c`) as ReplayShell's Web server instead of Apache. However, the toy server simply reads a folder of objects from disk into memory as a key-value store with the requests as keys and responses as values. It then finds the stored request which matches the incoming request URL exactly. This is insufficient for replaying Web pages.

Since we needed more complicated matching of HTTP/1.1 request header fields, we rewrote the server to support the matching semantics in ReplayShell's CGI script (§3.2). On an incoming request, our modified QUIC server sets up the request header fields as environment variables, executes the CGI script, reads the response generated by the CGI

script, and sends it back to the browser. We also changed the QUIC server to run on any IP address, instead of its previous default of 127.0.0.1, to mimic the multi-origin nature of websites. Lastly, we modified ReplayShell to spawn a QUIC server for each IP/port pair, replacing Apache.

Our evaluation of QUIC does not enable FEC because it is not implemented in QUIC's toy server. Additionally, we built the QUIC server code in debug mode because all QUIC transfers were unsuccessful in release mode. While this incurs a slight performance penalty due to debug-only code (e.g., extra checks), we minimized the impact by disabling logging.

5.3 Page Load Time Measurements

5.3.1 Definition

Page load time is defined as the time elapsed between two timing events, navigationStart and loadEventEnd, in the W3C Navigation Timing API [14]. navigationStart is the time at which a browser initiates a page load as perceived by the user. loadEventEnd is the time immediately after the browser process's load event is fired which corresponds to the end of a page load. This difference represents the time between when the page was initially requested by the user and when the page is fully rendered. It is important to note that bytes can continue to be transferred for a page load even after loadEventEnd [36].

5.3.2 Measuring Page Load Time

To automate the page load process and measure page load times, we use Selenium version 2.39.0 and Chrome Driver version 2.8, a widely used browser-automation tool. To ensure Chrome does not load any objects from its local cache, we pass the --incognito flag instructing it to open a private instance. We also pass the --ignore-certificate-errors flag to override certificate warnings for HTTPS sites. We use the Web Driver API to obtain timing information from Selenium and define page load time to be the time elapsed between the navigationStart and loadEventEnd events.

5.3.3 Optimal Page Load Time

We define the optimal page load time for each website as:

$$minimumRTT + (siteSize/linkRate) + browserTime.$$

The first term represents the minimum time between when the first HTTP/1.1 request is made at the client and the first byte of the HTTP/1.1 response is received by the client. Ignoring server processing time, this term is the minimum RTT of the emulated network.

The second term represents the time taken to transfer all bytes belonging to the Web page over a fixed capacity link. We calculate the site size by counting the total number of bytes delivered over the emulated link from the Web servers to the browser between the navigationStart and loadEventEnd events.[1] This term represents the minimum amount of time required to deliver a Web page worth of bytes regardless of the transport protocol in use.

The third term represents the time it takes for the browser to process all the HTTP/1.1 responses and render the Web page (using the definition of "loaded" above). We equate this time to page load times encountered in ReplayShell alone without network emulation. This mimics an infinite-capacity, zero-delay link and all delays incurred in this scenario are solely due to processing in the browser.

This optimal time represents a lower bound for achievable page load times. We calculate this value for each site and plot the distribution of optimal page load times alongside the page load time distributions obtained when using HTTP/1.1, SPDY, and QUIC.

5.3.4 Static Link Results

We evaluate each of the above multiplexing protocols over 100+ configurations: the Cartesian product of a logarithmic range of link speeds from 0.2 Mbits/sec to 25 Mbits/sec and a linear range of minimum RTTs from 30 ms to 300 ms. We selected this range of link speeds to incorporate the range in global average link speeds reported by Akamai [30]. Similarly,

[1] Recall that bytes may be transferred even after loadEventEnd.

we selected the RTT range to incorporate a variety of different network conditions, from local area networks to wide area networks and satellite links. For each network configuration, we compare HTTP/1.1, SPDY, and QUIC with the optimal page load times defined above.

Figure 5-1 shows the distributions of page load times with HTTP/1.1, SPDY, and QUIC and the optimal page load times for 9 different representative configurations out of the 100. These network configurations represent every combination of three link speeds (low, medium, high) and three minimum RTTs (low, medium, high). We find that HTTP/1.1, SPDY, and QUIC all achieve page load times far from optimal on most network configurations. This suboptimality widens as link speed increases: on a slow 1 Mbit/sec link with a minimum RTT of 300 ms, `mod_spdy` is 2.02X worse than optimal, whereas on a faster link of 25 Mbits/sec with 300 ms minimum RTT, `mod_spdy` is 4.93X worse than optimal at the median. Similarly, we find that the gap between these protocols and the optimal widens as minimum RTT increases. Over a link of 14 Mbits/sec with 30 ms minimum RTT, HTTP/1.1 is 1.16X worse than optimal at the median, but an increase of minimum RTT to 300 ms makes HTTP/1.1's median page load time 3.77X worse than optimal. We noticed similar patterns over the remaining configurations out of the 100+ configurations that we evaluated and we explain this suboptimality further in §6.

5.3.5 Cellular Network Results

To evaluate performance on cellular networks. we use the cellular network traces provided in our corpus (§3.5). We consider HTTP/1.1, SPDY, and QUIC over a Verizon LTE trace with a minimum RTT of 120 ms. Figure 5-2 shows the distribution of page load times recorded when loading all 500 Web pages in our corpus over a Verizon LTE link with 120 ms minimum RTT. As illustrated in the figure, HTTP/1.1, SPDY, and QUIC perform far worse than optimal. At the 95th percentile, page load times with HTTP/1.1 are 8.5X worse than optimal page load times, while SPDY and QUIC are 7.6X and 12X worse, respectively. We noticed similar patterns across different configurations including traces from AT&T's network and across several other minimum RTTs. Data and figures for these

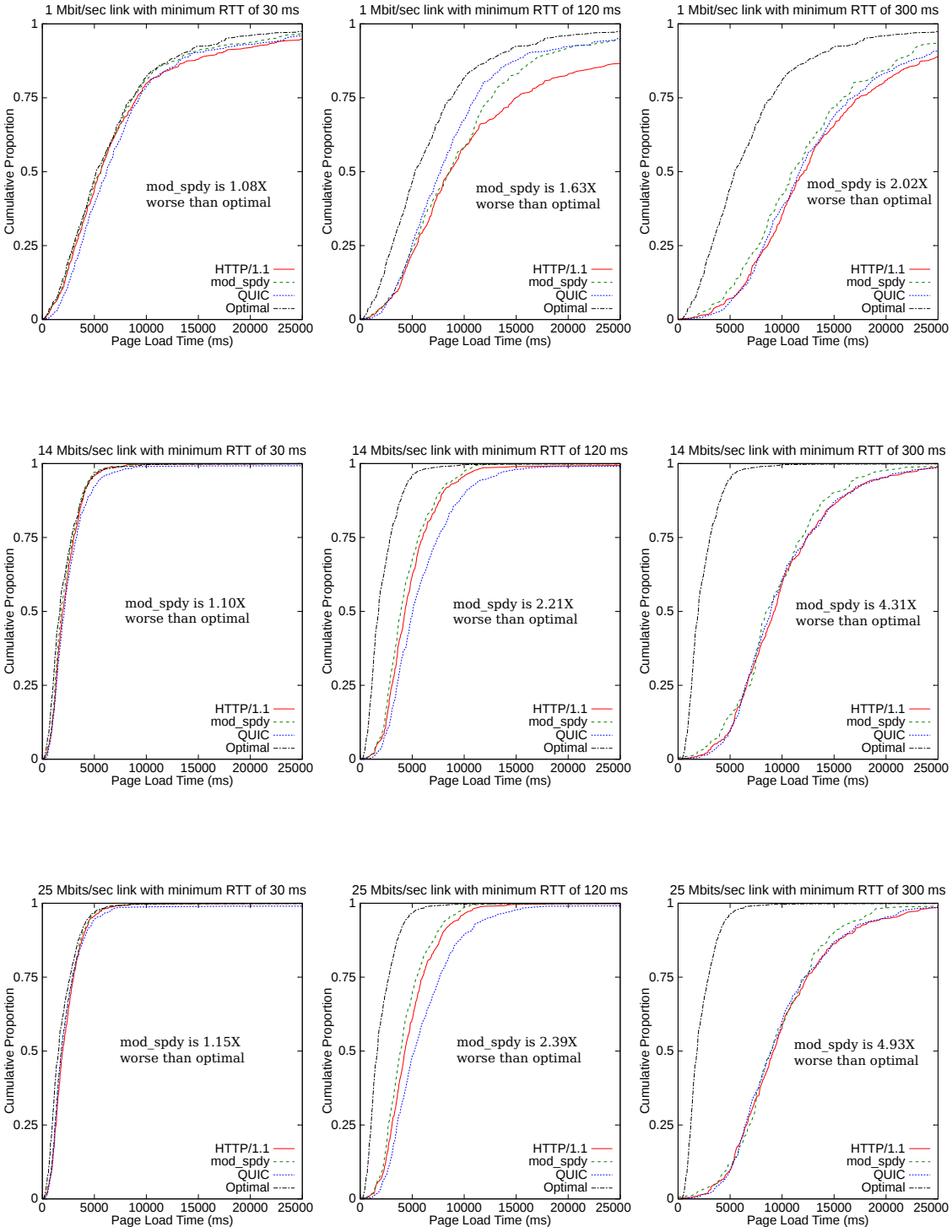


Figure 5-1: The gap between page load times with HTTP/SPDY/QUIC and Optimal grows as link speed or minimum RTT increases

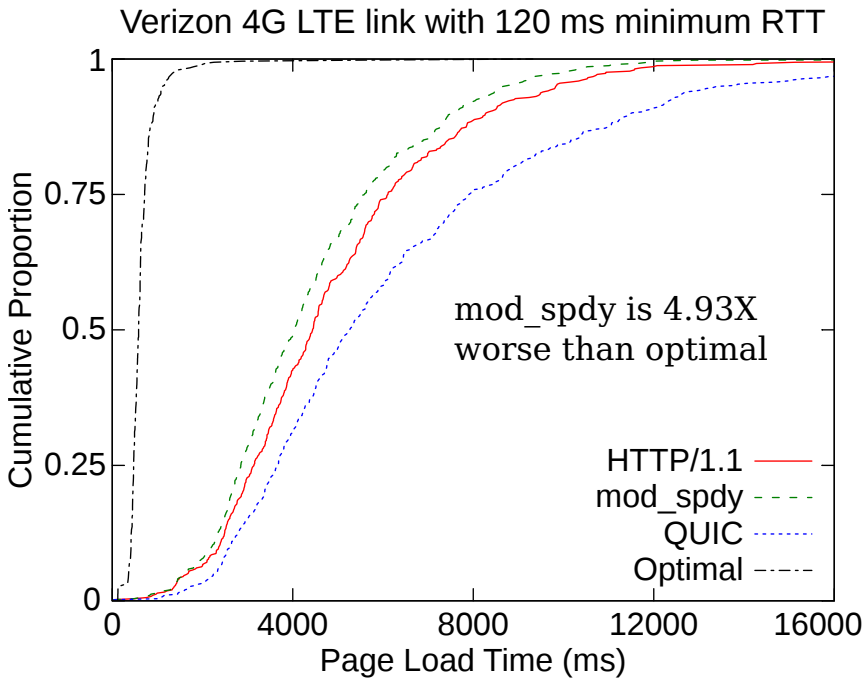


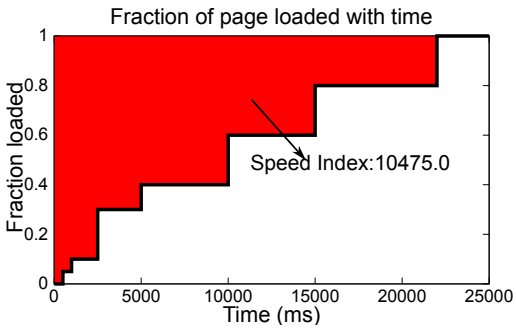
Figure 5-2: Page load times over 4G cellular networks when using HTTP/1.1, SPDY, or QUIC are more than 7X worse than the optimal

other configurations are available at `https://www.github.com/ravinet/sites`.

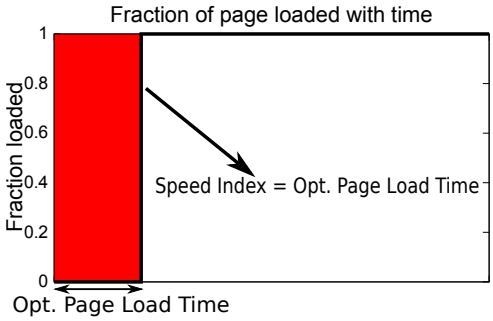
5.4 Speed Index

5.4.1 Definition

Although the page load time is straightforward to measure, it may not indicate when a Web page is actually ready for the client. In particular, for long Web pages, content "below-the-fold" of the screen may be downloaded long after the page is already usable. This problem persists across many modern Web pages which support infinite scrolling. For example, Facebook may load wall postings below the user's current location on the page for a better user experience if the user scrolls down. Similarly, CNN has a homepage that is much larger than standard screens, resulting in the original page load consisting of downloading content below the visible area that cannot be seen immediately. In such cases, the onload event used to measure page load time fires much after the page is ready for user interaction. To overcome this discrepancy between page load time and the user-perceived experience,



(a) Speed Index is the area above the curve of the completeness of a page load as a function of time



(b) We define an upper bound on optimal Speed Index by assuming that a page instantaneously jumps from 0% to 100% completeness at the optimal page load time

Figure 5-3: Speed Index calculation

Google has proposed a new metric: Speed Index [26].

Speed Index tracks the visual progress of a Web page in the visible display area. A lower Speed Index is synonymous with content that is rendered more quickly. For example, a page that immediately paints 90% of its visual content will receive a lower Speed Index than a page that progressively paints 90% of its content, even if both pages fire their onload event at the same time.

To compute Speed Index, the completeness of a page's display area needs to be measured over time. Speed Index can be computed by doing a pixel-by-pixel comparison with the final loaded Web page to determine a percentage complete. Once the entire page has loaded, we plot the completeness percentage of the page rendering over time. The Speed Index is computed by measuring the area "above-the-curve" of this relationship (Figure 5-3a). Thus, the Speed Index stops increasing once the page is completely rendered. Pages which render quicker result in lower Speed Indexes as the percentage completeness reaches 100% quicker and thus the Speed Index stops increasing earlier.

5.4.2 Measuring Speed Index

We calculate Speed Index using WebPagetest [26]. WebPagetest computes Speed Index by recording a video of the page load at 10 frames per second. After recording, it compares each frame to the final captured frame to plot the percentage completeness across time. WebPagetest operates in a client-server model, where requests to measure a page's Speed Index are sent to a server which in turn coordinates sending requests to its clients. The clients execute the requested page load and compute the Speed Index. Upon completion, the clients send the computed Speed Index to the server, which then forwards the results back to the original requester. In a private instance of WebPagetest, the client and server are run on the same machine. To automate this testing, we use WebPagetest's `wpt_batch.py` API [29] that communicates with the WebPagetest server and returns the test results as an XML document.

To measure Speed Index using Mahimahi, we run a private instance of WebPagetest inside ReplayShell. Because WebPagetest runs exclusively on Windows, we run WebPagetest within a VirtualBox Windows virtual machine, inside ReplayShell. The virtual machine is configured to be in NAT mode, and works as any other Linux process running inside ReplayShell would. Since WebPagetest runs only on Windows, we only evaluate HTTP/1.1 on the Speed Index metric.

5.4.3 Optimal Speed Index

Calculating an optimal Speed Index is difficult. Page loads are characterized by dependencies between resources where certain resources cannot be downloaded before others (for instance, due to DOM or JavaScript parsing), and the visual location and size of resources on the display area shares no simple relationship with the dependency graph. Instead, we define an upper bound[2] on the optimal Speed Index below.

We assume that a site renders in one shot at the optimal page load time; Figure 5-3b illustrates its implications for the "optimal" Speed Index. As shown, the percentage completeness of a given Web page is 0% until the optimal page load time where the percentage

[2]Recall that a lower Speed Index is better.

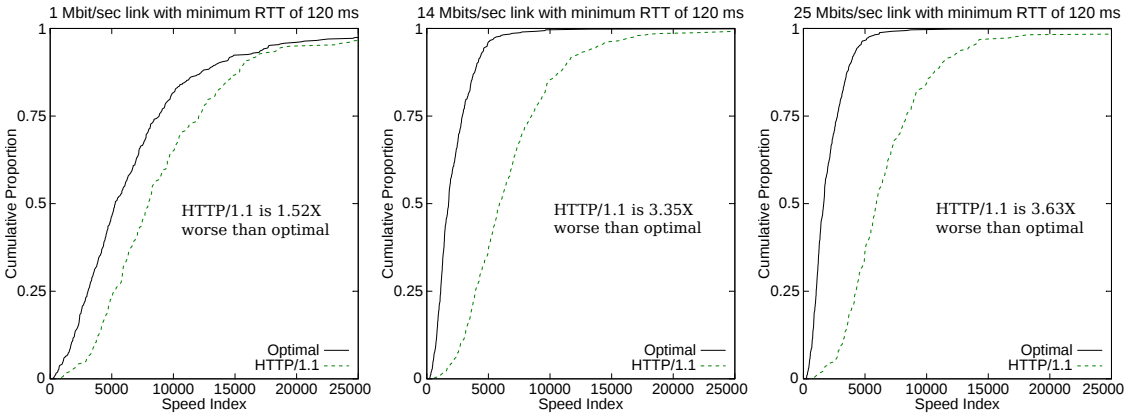


Figure 5-4: The gap between Speed Index with HTTP/1.1 and Optimal grows as link speed increases (at a fixed minimum RTT)

completeness jumps to 100%. As a result, the "area above the curve," or optimal Speed Index, is the optimal page load time. There could very well be better rendering strategies that more gradually render the page between 0 and the optimal page load time, but such improved characterizations of the optimal Speed Index will only further increase the already large slowdowns from the optimal Speed Index. From this point on, we call this approximation the optimal Speed Index.

5.4.4 Static Link Results

We measure the Speed Index for each site in our corpus over several different network configurations. We consider networks with link speeds of 1 Mbit/sec to 25 Mbits/sec and minimum RTT of 120 ms. We notice similar patterns to those discussed with page load times: the gap between Speed Index with HTTP/1.1 and optimal Speed Index grows as link speeds increase; over a 1 Mbit/sec link with 120 ms minimum RTT, Speed Index with HTTP/1.1 is 1.52X worse than optimal, while over a 25 Mbits/sec link with 120 ms minimum RTT, Speed Index with HTTP/1.1 is 3.63X worse than optimal. This trend is illustrated in Figure 5-4.

Chapter 6

Suboptimality of Multiplexing Protocols

In §5, we observed that multiplexing protocols for the Web are suboptimal in typical cases, and their slowdown relative to optimal is only increasing with increasing link speeds and RTTs. We discuss this further by first showing that page load times plateau with increasing link speeds (§6.1) and then understanding why this occurs (§6.2) and how it relates to RTT.

6.1 Trends with Link Speed

We observe that page load times do not continually decrease as link speeds increase. We illustrate this trend in Figure 6-1 which plots page load time as a function of link speed for several fixed minimum RTT values. As shown by the graphs, page load times observed using HTTP/1.1, SPDY, and QUIC plateau as link speeds increase beyond 1 Mbit/sec. Since page load times flatten out despite increasing link speeds, the gap between all three protocols and the optimal increases as the link speed increases, leading to more wasted capacity on faster links.

6.2 Understanding Suboptimality

The results from Figure 6-1 show that HTTP/1.1, SPDY, and QUIC are far from optimal on faster links. We test an extreme scenario where we remove LinkShell altogether (in effect, emulating an infinite-capacity link), and simply add a 75 ms delay to each packet in either

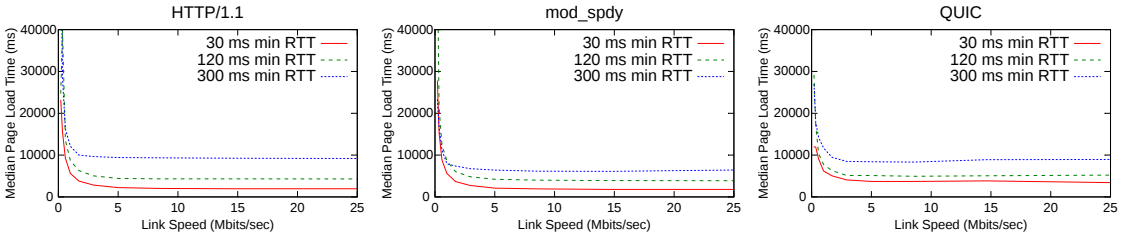


Figure 6-1: Page load times with HTTP/1.1, SPDY, and QUIC stop increasing as link speeds increase beyond 1 Mbit/sec

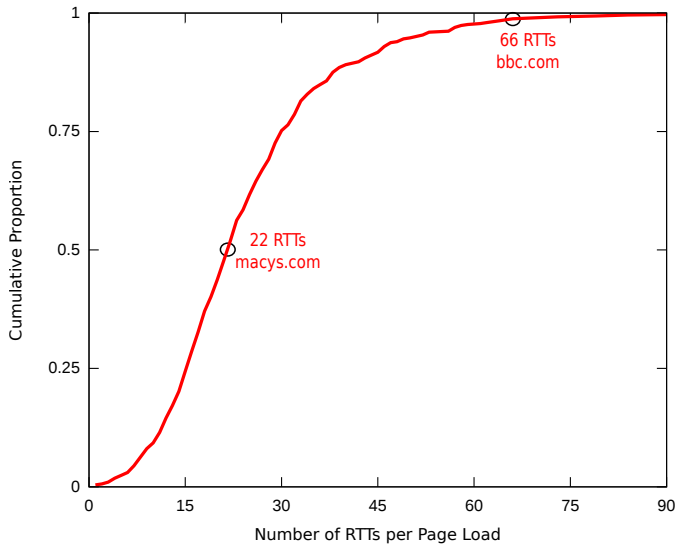


Figure 6-2: Object dependencies lead to many RTTs above baseline page load times

direction (running DelayShell alone).

For each Web page, we compute the page load time on this infinite-capacity, finite-delay link, and compare it to the page load time on an infinite-capacity zero-delay link (this is *browserTime* from §5.3.3). We plot the distribution of the difference between these two times divided by the finite minimum RTT of 150 ms in Figure 6-2. This tells us the number of RTTs that are incurred in the page load process due to the finite-delay link. This difference is 22 RTTs at the median and as high as 90 RTTs in the tail, while a hypothetical optimal protocol would incur only 1 RTT over this infinite-capacity link!

The additional RTTs over an optimal protocol are primarily a result of two factors: inherent source-level dependencies such as Web pages referring to other pages and browser constraints on the number of parallel flows. At the beginning of a page load, a browser is unaware of all the requests that must be made to complete the page load. Parsing HTTP re-

sponses leads to the expansion of the DOM tree and results in more requests when resources refer to other resources. Modern browsers also limit the number of parallel connections, both per origin server and per browser process. As a result, even when a browser has several requests it can make in parallel, it may have to wait for an existing TCP connection to complete. Combined, these properties result in the serialization of HTTP requests (some required due to source-level dependencies and some enforced by the browser), preventing the browser from utilizing the link more completely.

Figure 1-1 shows part of the object dependency graph for `http://www.tmz.com/` extracted from Chrome's Developer Tools. After processing the initial HTTP response, the browser must request over 100 objects. Depending on how these objects are stored across servers, browser constraints could serialize these requests. Additionally, many source-level dependencies are shown. The deepest part of the dependency graph for TMZ shows a dependency chain of 10 objects. Thus, the breadth of this graph causes serialization due to browser constraints and the depth causes serialization due to source-level dependencies. The serialization accounts for the additional RTTs seen in Figure 6-2 and a large part of the gap between HTTP/SPDY/QUIC and optimal.

The disparity between HTTP/SPDY/QUIC and optimal is exacerbated on links with very high bandwidth. Since links between browsers and Web servers may not be fully utilized during a page load due to serialization, higher link speeds lead to larger amounts of wasted link capacity. On links with low capacity, browsers can often saturate the link as network capacity becomes the primary bottleneck over computation. Even though the browser still processes responses before it can make subsequent requests, and adheres to browser constraints, the link remains close to fully utilized as each request and response takes longer to traverse the link. This is evident in Figure 5-1, which shows page load time distributions over an emulated 1 Mbit/sec link with a minimum RTT of 30 ms. There is little difference between the optimal distribution and that of SPDY, HTTP/1.1, and QUIC.

Even at low link speeds, the suboptimality of HTTP/SPDY/QUIC grows with increasing minimum RTT because of the serialization discussed earlier. As discussed in §5.3.3, the optimal page load time eliminates page dependencies and thus only includes a single minimum RTT. Consequently, an increase in minimum RTT yields little change to optimal

page load times. Page load times with HTTP/SPDY/QUIC, on the other hand, are plagued by several additional RTTs (Figure 6-2), each of which is increased with an increase in minimum RTT. As a result, a change to minimum RTT can drastically increase page load time for these multiplexing protocols, whereas, the optimal values stay relatively unchanged.

In §7, we present a system, Cumulus, which attempts to mitigate the effects that the aforementioned source-level dependencies have on page load times when RTTs are high. Specifically, Cumulus combines CDNs with cloud browsers to reduce page load times when the user is at a significant delay from the a Web page's servers.

Chapter 7

Design of Cumulus

Cumulus uses two components that cooperate to speed up page loads: the Puppet (§7.1), a headless browser that is run by the user on a cloud server that is closer to the website than the user, and the Mini-CDN (§7.2), a transparent caching proxy that runs on the user's machine.

When the user's browser requests a URL that is not in the Mini-CDN's cache, the Mini-CDN forwards the request over the long-delay path to the Puppet browser. The Puppet browser recursively fetches the root object identified by the URL and prefetches the referenced objects on the same page, over the shorter delay path between the Puppet and the Web servers. The Puppet then packages these objects and sends them back to the Mini-CDN, which holds them until requested by the user's browser. Figure 7-1 illustrates the interaction between the Puppet and the Mini-CDN on a single page load.

We now describe each of the two components in greater detail.

7.1 Puppet

The Puppet continuously listens for new requests from the Mini-CDN and loads the URL specified in the request in a headless browser. Since the HTTP request by itself doesn't indicate the scheme (HTTP or HTTPS), the Mini-CDN also sends the scheme along with the request.

To load the URL, the Puppet uses a headless browser, PhantomJS [15], to recursively

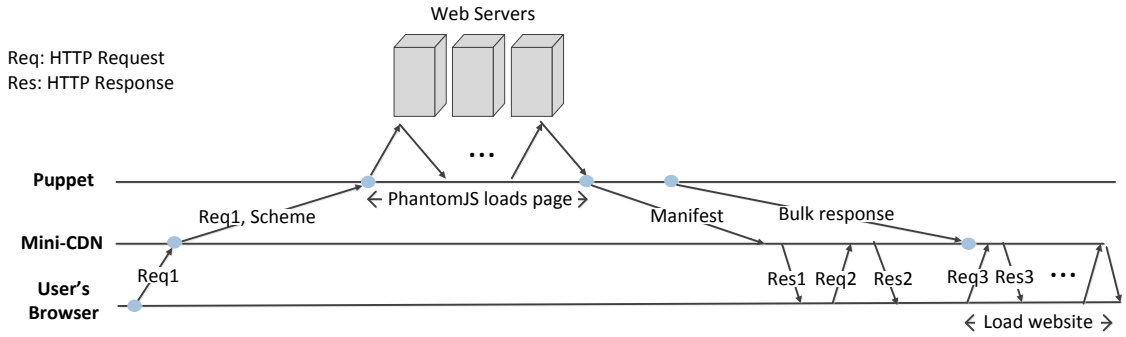


Figure 7-1: A single page load using Cumulus

fetch all the Web resources originating from the URL. Using a headless browser at the Puppet allows Cumulus to mimic the user's browser over a shorter-delay path to the Web servers.

To allow response customization based on header fields (such as the Cookie and User Agent) and data (for POST requests) in the user browser's original request, PhantomJS loads the URL using exactly the same HTTP request that it receives from the Mini-CDN.

The Puppet records all HTTP(S) traffic sent to and from PhantomJS during the page load by running PhantomJS within a Linux network namespace [4], and using Linux's Destination NAT [16] to intercept TCP connections leaving the namespace. The Puppet records all requests and responses in flight.

Once PhantomJS completes the page load, the Puppet sends a *Manifest* (Figure 7-2) of all the requests seen during the page load to the Mini-CDN so that the Mini-CDN can add those requests to its cache and mark their responses as "pending." The Puppet then pushes down the responses en masse (Figure 7-2).[1]

7.2 Mini-CDN

Similar to the Puppet, the Mini-CDN is structured as a Unix shell that runs its children inside a new network namespace and uses Linux's Destination NAT to transparently intercept all TCP connections leaving that namespace. Users wishing to benefit from Cumulus will

[1] As these responses arrive, the Mini-CDN fills in the pending responses. If the Mini-CDN receives a request from the browser for a pending object, it blocks in handling the request until the response is filled in.

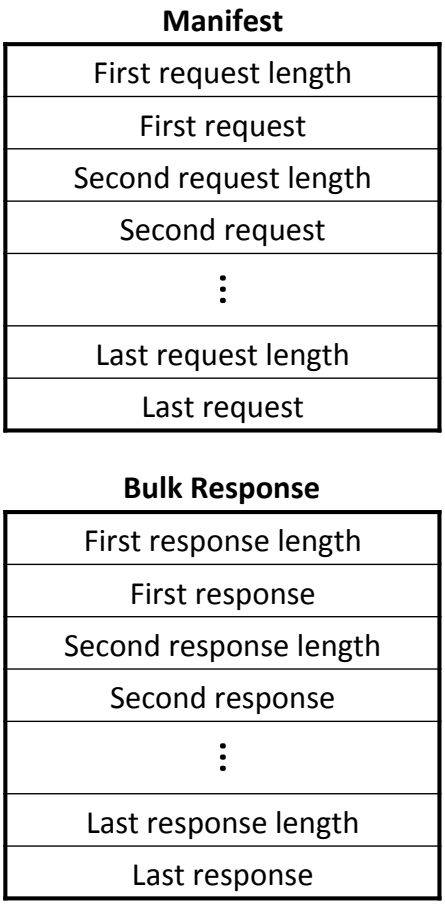


Figure 7-2: Manifest and bulk response formats

run a Web browser "inside" the Mini-CDN.

This design was chosen for ease of use. Using the shell abstraction makes Cumulus compatible with any browser or HTTP(S) network program. The Mini-CDN could also be run as a non-transparent proxy (with the user manually configuring the Web browser to use a local proxy server).

The Mini-CDN maintains a cache of HTTP requests and corresponding responses. This cache is filled in by the Puppet and is used to answer requests from the browser when possible. In practice, there are occasional mismatches between requests issued by the headless browser in the Puppet, compared with the user's own Web browser. For example, some sites use JavaScript to issue requests that depend on the precise local time or on the output of a pseudorandom number generator. To handle these cases, the Mini-CDN uses a matching heuristic to retrieve requests from the cache.

At the Mini-CDN, requests from the browser are handled as follows. We use the terminology of Figure 7-1:

1. If the request has a side-effecting verb (POST, PUT, DELETE or PATCH), it is forwarded to the Puppet. The Mini-CDN waits for the response.
2. If the request does not match any request in the cache (Req1), it is forwarded to the Puppet. The Mini-CDN waits for the response (Res1).
3. If a matching request is found in the cache (Req2), but the corresponding response is pending, the Mini-CDN waits until the response arrives (Res2).
4. If a matching request is found (Req3) and the response (Res3) is available in the cache, the Mini-CDN replies with the pre-fetched response.

7.3 Fuzzy Matching Heuristic

The Mini-CDN needs to match requests from the user's browser against the requests stored in its cache. These requests invariably are influenced by local state present in the browser. For instance, certain query string parameters could be derived from the current time. Because this local state differs between the user's browser and the Puppet, the incoming request may not exactly match any request stored in the cache.

We handle this using a matching heuristic that enforces that some parts of the request have to match exactly, while tolerating some degree of imperfection in other parts. Specifically, we expect the Host header field and the requested resource—until the last "/" or "?" character—to exactly match the corresponding values in some stored request. If this isn't the case, the Mini-CDN forwards the request to the Puppet.

The matching heuristic tolerates mismatches beyond the last "/" or "?", which denotes a query string. In these cases, the algorithm retrieves the cached request whose URL has the maximal common substring to the requested URL.

7.4 Browser Behavior Mismatches

When we measure page load times for the experiments in Chapters 9 and 10, we allow the browser to load only one root object and its associated dependencies. The Mini-CDN ignores a browser's subsequent requests that are not found—as judged by the matching heuristic—in the page's dependency tree calculated by PhantomJS. In our measurements, we have found that roughly 10% of popular Web sites receive different execution in PhantomJS compared with Google Chrome, meaning that Chrome will request an object that is not found in the PhantomJS dependency tree (even with fuzzy matching). This typically occurs on advertising-heavy websites that we suspect perform user-agent sniffing based on differences in JavaScript execution engines. Although this issue could lead to an overestimate of Cumulus's gains, we have not found such an effect. We will address this issue in more detail in our near-term work on Cumulus.

Chapter 8

Experimental Setup for Cumulus

8.1 Corpus

To evaluate Cumulus, we created a corpus of 500 pages. We start with the landing pages in the Alexa U.S. top 100 list [13]. For each landing page, we obtain 3–4 other URLs by clicking links on the landing page and performing image and text searches. These additional pages are often far more complex than landing pages. For example, the Google homepage has a total of 14 Web objects, while a Google image search for "MIT" produces a page with 57 Web objects. We also include the mobile version of 20 Web pages. Unlike the corpus of recorded pages we used in our case study of the Web (3.5), this corpus is a list of Web page URLs.

8.2 Page-Load Measurement

To automate page loads and measure page load times, we use Selenium (version 2.39.0), a widely used browser-automation tool, with Chrome Driver version 2.8 and Opera Driver version 1.5. With each browser, we use the Web Driver API to obtain timing information from Selenium and define "page load time" as the time elapsed between the navigationStart and loadEventEnd events [14]. We evaluate the four systems shown in Table 8.1 running on Ubuntu 13.10 GNU/Linux. All our measurements are for single page loads with a cold cache.

System	User's browser	Browser Version	Cloud Proxy
Opera Turbo	Opera Turbo	12.16	Opera's proxy servers
Chrome Proxy	Chrome with publicly available proxy plugin [17]	37.0.2062.120	Google's Data Compression Proxy [3] servers
Chrome	Chrome	37.0.2062.120	No proxies
Cumulus	Chrome within the Mini-CDN	37.0.2062.120	Cumulus's Puppet

Table 8.1: Systems evaluated

When using Cumulus, all TCP connections are intercepted by the Mini-CDN. For HTTPS connections, the Puppet performs certificate checking of the origin server on the user's behalf, and the Mini-CDN presents the browser with a self-signed certificate instead of the certificate of the origin server. We pass the --ignore-certificate-errors flag on the Chrome command line to permit this.

Chapter 9

Results with Cumulus

We test each system listed in Table 8.1 by loading a set of real Web pages in three different network settings: the AT&T cellular network in Boston (§9.1), in-flight Wi-Fi (§9.2), and several different wired transcontinental links (§9.3 and §9.4).

We repeat each measurement five times, rotating among the systems under test in order to mitigate the effects of network variability. We define Cumulus's "speedup" relative to a system as the ratio of the page load time when using that system to the page load time when using Cumulus, under identical network conditions. Thus, a speedup greater than 1 implies lower page load times with Cumulus, and vice versa. To visualize results consistently, we plot the distribution of speedups of Cumulus relative to different systems.

9.1 Cellular Networks

We ran experiments over the AT&T cellular network in Boston by using a PC laptop tethered to a Samsung Galaxy Note running Android OS version 4.2.2 on AT&T's LTE/GSM/WCDMA cellular network. We used our 500-website corpus described in §8.1. Page loads with Cumulus used a Puppet running on an Amazon EC2 instance in Virginia.

Results are shown in Figure 9-1, and median speedups for Cumulus over other systems are listed in Table 1.1.

We also ran this experiment on 20 mobile-optimized versions of sites in our corpus. We find that even on these sites, Cumulus achieves a median speedup of 1.28× over Google

Chrome Data Compression Proxy.

While the number of objects decreases on mobile-optimized sites, a significant number of requests still need to be made and RTTs continue to adversely affect page load times. For instance, in our corpus, the median number of resources on mobile sites is 76. RTTs in cellular networks can easily reach 100 ms. As a result, despite a reduction in data transmission, mobile versions of sites are still plagued by the long RTTs of cellular networks.

9.2 In-Flight Wi-Fi

We ran experiments over in-flight Wi-Fi networks, where RTTs are on the order of 1 second [18]. Because of the poor performance of in-flight Wi-Fi (which leads to lengthy test durations), we used a smaller set of 12 sites from our 500-site corpus. The set included landing pages as well as interior pages.

Experiments comparing Google Chrome and Cumulus were run on a United Airlines flight from Boston to Newark, N.J., using United Wi-Fi [19]. Experiments comparing Chrome Proxy, Opera Turbo, and Cumulus were run on four Delta Airlines flights between Boston, Atlanta, and Charlotte, N.C., using Gogo Wi-Fi [20]. Each experiment was performed using a MacBook Pro laptop running a GNU/Linux virtual machine. Cumulus used a Puppet running on a desktop at MIT.

Our results are shown in Figure 9-2, and median speedups for Cumulus over each other system are listed in Table 1.1.

9.3 Wired Transcontinental Links

We also ran experiments on a varied set of wired transcontinental links. For the first five of these experiments, we used a desktop at MIT to load websites hosted in five countries: Australia, Japan, Singapore, Brazil, and Ireland.

For each country, we selected 10 websites hosted in that country from its Alexa Top 100 list. Page loads with Cumulus used a Puppet running on an Amazon EC2 instance located in the country being considered.

Client Location	Speedup
Bangalore, India #1	1.38×
Bangalore, India #2	1.18×
Haifa, Israel	1.16×
Bogota, Colombia	1.60×
Pretoria, South Africa	1.51×

Table 9.1: Median speedups with Cumulus over Google Chrome for U.S. Web pages loaded from around the World.

For the last experiment, we loaded our 500-site corpus from an Amazon EC2 instance in São Paulo, Brazil. Here, page loads with Cumulus used a Puppet running on an Amazon EC2 instance in Virginia.

Our results are shown in Figures 9-4–9-8, and median speedups for Cumulus over each other system are listed in Table 1.1.

9.4 Volunteer-Run Experiments

We ran more experiments over transcontinental wired links by having volunteers in several different regions across the world load our 500-site corpus with Google Chrome and Cumulus. Pages loaded with Cumulus used a Puppet running on an MIT desktop. (Again, Cumulus refers to the Google Chrome Web browser running inside Cumulus.)

Table 9.1 lists the median speedups for Cumulus, as compared with Google Chrome for each client location.

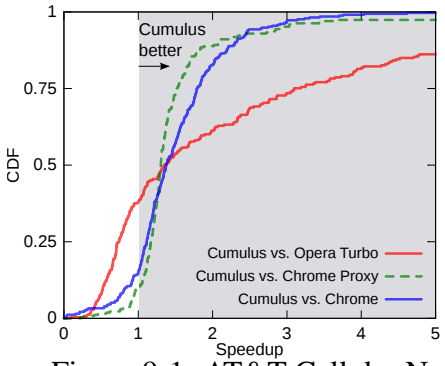


Figure 9-1: AT&T Cellular Network

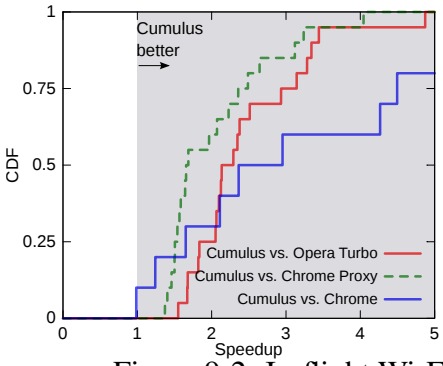


Figure 9-2: In-flight Wi-Fi

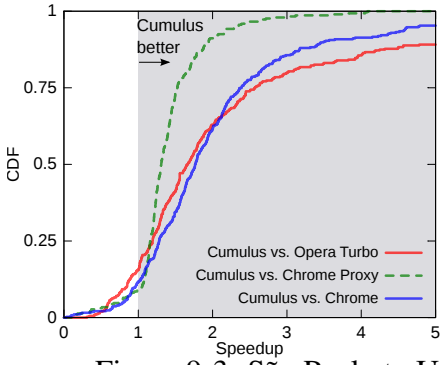


Figure 9-3: São Paulo to U.S.

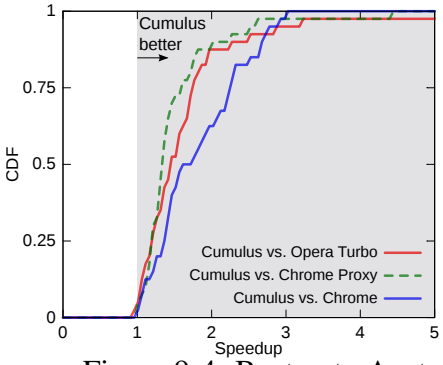


Figure 9-4: Boston to Australia

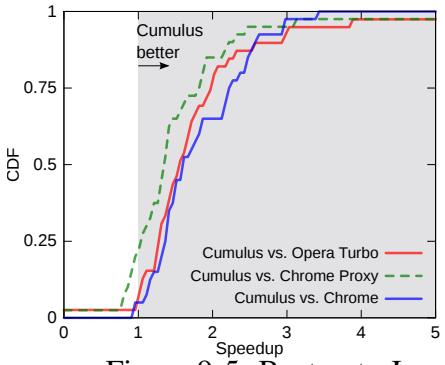


Figure 9-5: Boston to Japan

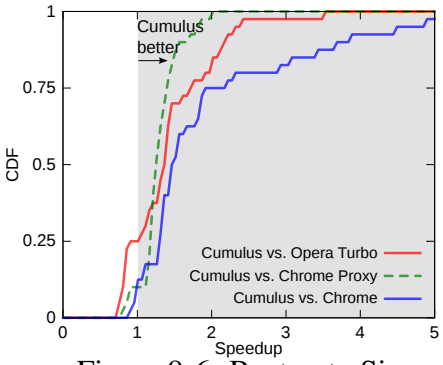


Figure 9-6: Boston to Singapore

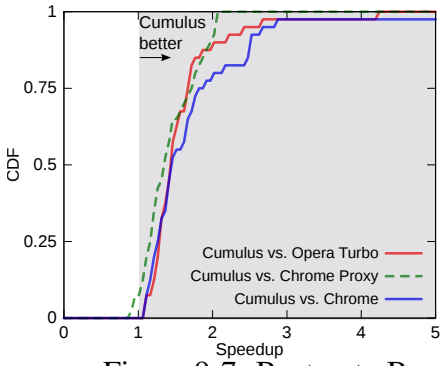


Figure 9-7: Boston to Brazil

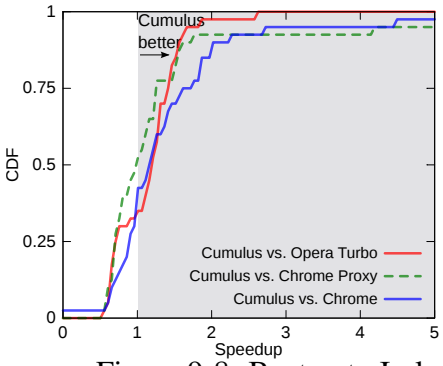


Figure 9-8: Boston to Ireland

Chapter 10

Understanding Cumulus

Cumulus moves dependency resolution to the Puppet where RTTs to Web servers are lower than they are from the client.

The benefits of this technique depend on two factors:

1. The RTT between the user and the Web servers.
2. The complexity of the Web page.

To further understand the importance of each, we performed Web page loads in emulation using two pages: a TMZ article with 458 objects and the Google homepage, which has 15 objects. These pages represent two extremes of the complexity spectrum. We use DelayShell to emulate fixed minimum RTTs from 0 ms to 400 ms. For each RTT, we load each page five times with Google Chrome Data Compression Proxy—which compresses objects in-flight, but does not perform dependency resolution on the user's behalf. We compare these results with Cumulus, which performs dependency resolution but does not compress objects in-flight.[1]

Page loads with Cumulus used a Puppet running on the other side of the emulated long-delay link. Speedups for Cumulus relative to Chrome Data Compression Proxy are shown in Figure 10-1.

We observe two trends:

1. For a fixed Web page, speedups with Cumulus increase as RTT increases.
2. For a fixed RTT, speedups with Cumulus are larger for more complex Web pages.

[1] In practice, many Web objects are already compressed in transit by the origin Web server.

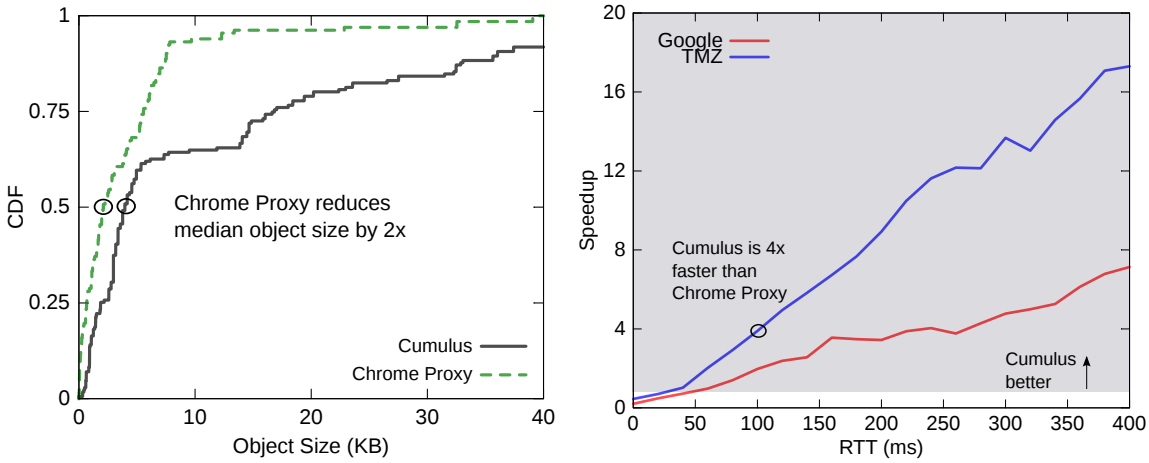


Figure 10-1: Chrome's Compression Proxy reduces the median object size for TMZ by 2× and reduces the total amount of data transmitted by 4×. Yet, Cumulus still loads TMZ 4× faster over a 100 ms RTT link.

Our results show as much as a 4× speedup relative to Google Chrome Data Compression Proxy at an RTT of 100 ms, an approximate figure for a transcontinental link. This is surprising given claims that Google's Data Compression Proxy reduces Web pages by up to 50% in size [3].

To understand this apparent discrepancy, we plot the distribution of object sizes seen when loading the same TMZ article with Chrome Data Compression Proxy and Cumulus (Figure 10-1). With Chrome Data Compression Proxy, median object sizes are indeed 2× smaller than with Cumulus, and the overall data transfer is 4× less.

Nonetheless, Cumulus outperforms Chrome Data Compression Proxy beyond RTTs of 30 ms. These results suggest that remote dependency resolution is an effective use of a cloud browser, especially when the client has a long-delay path to Web servers.

Chapter 11

Limitations and Future Work with Cumulus

Cumulus achieves considerable gains by combining the notion of a CDN with a cloud browser, allowing the Puppet and Mini-CDN to cooperate to pre-fetch an entire page's objects in one RTT. In doing so, Cumulus goes where other cloud browsers have not: the Puppet, deployed by the user on a public cloud server, gets to see the plaintext of the entire Web page, including each origin server's respective components. The user's own Web browser must delegate the checking of certificates to the Puppet on its behalf.

This raises security concerns. The traditional use of HTTPS is to achieve end-to-end confidentiality to the user's own browser, and for this reason, Opera Turbo and Chrome Data Compression Proxy do not touch HTTPS requests. With Cumulus, the user achieves confidentiality to a cloud-deployed intermediary browser, and again between that browser and the user's own computer. Whether users are comfortable with the possibility that a rogue cloud operator may spy into the plaintext of their encrypted communications is not clear.

We propose an alternative deployment scheme where a user does not trust a cloud operator with the plaintext of their HTTPS connections. In this model, each origin Web server would run a Puppet locally. Requests from a client's mini-CDN would be delivered directly to an origin Web server, as they are on today's Web, ensuring that a client has a direct HTTPS connection with the correct server and personalized content (eg. Cookies) is

encrypted.

Puppets at origin Web servers will load the transitive closure of the incoming request, but will omit content which is to be served via HTTPS by origin Web servers belonging to different commercial entities. As a result, client browsers will make HTTPS requests to these different Web servers and these requests will not be served by the mini-CDN. Instead, these requests will be sent via HTTPS directly to the appropriate origin Web servers whose Puppets will perform dependency resolution as above, and reply with new Bulk Responses.

To preserve a client's ability to verify the integrity and authenticity of objects and origin servers, we propose having all objects requested by an origin Web server from a different origin Web server (via HTTPS) to be signed with the incoming request, corresponding response, and expiration time. Using these signatures, modified client browsers can confirm that objects loaded by an origin Web server on the client's behalf did indeed originate from the correct origin server. These content signatures are required only across origins belonging to the same commercial entity that already trust each other, making deployment more practical.

Our measurements evaluated only fresh page loads with a cold cache. Although this generally represents a worst-case scenario, the presence of local caching may diminish Cumulus's relative gains—both because the raw weight of Web pages is effectively less, and because of the risk that the Puppet may push an object to the Mini-CDN that the browser already has (and will not need to request) [49]. We plan to address this by observing and respecting the browser's If-Modified-Since header in the Mini-CDN and Puppet, but to evaluate the actual benefit of doing so quantitatively, we will need to come up with an automated test of non-cold-cache Web browsing behavior (e.g., loading an entry Web page and then following a certain number of hyperlinks).

Our transcontinental measurements in Figures 9-3–9-8 each use a Puppet that is well-positioned for the hosting country of the target websites. In practice, our system will not know in advance the best Puppet to use for a particular Web request. We have not evaluated the consequences of imperfect routing of requests to geolocated Puppets. CDNs have expended considerable effort on this problem [31], and we believe the same techniques are applicable.

The matching heuristic (§7.3) is imperfect. In the near term, we plan a more rigorous evaluation of the factors that lead to a mismatch in local state (and issued requests) between the Puppet's headless browser and the user's own Web browser, and whether it is possible to reduce such mismatches and quantify their performance impact.

We do not address the economic question of whether typical users will choose to run a Puppet on a public cloud server, relative to the improvement in Web performance. In most settings, there are benefits from economies of scale (e.g., a single service that ran Puppets for many users), but the security posture of Cumulus complicates this matter. It would be beneficial to research other ways by which different users' private data might be isolated within a cloud Puppet service.

All of our present measurements are of Cumulus's reduction in the experienced "page load time," which we define as the difference between the time of the browser's JavaScript events "navigationStart" and "loadEventEnd." Some contemporary work on browser performance has emphasized that waiting until the "bitter end" for a page to load completely is not the best metric of a user's Web experience. Google has proposed and works to optimize Chrome's performance on the "Speed Index" metric, which credits a browser for partial loads of visible content within a Web page (§5.4). We intend to examine the effect of Cumulus on this metric in the future.

Chapter 12

Conclusion

This thesis first presented the design, implementation, and applications of a new record-and-replay framework, Mahimahi, that enables reproducible measurements of the Web under emulated network conditions. Mahimahi improves upon existing record-and-replay systems in several ways: it accurately preserves the multi-origin nature of websites today, and, by isolating its own traffic, it allows several instances to run in parallel. Further, Mahimahi's flexibility permits a wide-range of client-server interactions and the use of UNIX shells in Mahimahi allows evaluation of arbitrary applications, beyond browsers.

We used Mahimahi to present a detailed case study of the Web today by evaluating three different multiplexing protocols: HTTP/1.1, SPDY, and QUIC, on two different metrics: page load time and Speed Index. We found that on typical network configurations all three protocols are substantially suboptimal on both metrics due to source-level dependencies and browser constraints which lead to HTTP request serialization and unoccupied links. Moreover, we found that this gap from optimal only widens as link speeds and RTTs increase.

To mitigate the effect of this HTTP request serialization when RTTs are high, we have implemented and tested Cumulus, a user-deployable combination of a content-distribution network and a cloud browser that improves page load times without modifying the Web browser or servers.

On several cellular networks, in-flight Wi-Fi connections, and transcontinental links, Cumulus accelerated the page loads of Google's Chrome browser by 1.13–2.36×. Cumulus

outperformed Opera Turbo by 1.19–2.13× and the Chrome Data Compression Proxy by 0.99–1.66×.

The most marked benefits—a 66% improvement over Chrome Proxy, and a more than doubling of performance relative to Opera Turbo and Chrome by itself—were seen on in-flight Wi-Fi, when a user's RTT to a page's Web servers is the greatest.

These results demonstrate the importance of *dependency resolution* "as a service"—that is, as close to the origin Web servers as possible—and the benefits that can be achieved by intelligent server-side push over current Web protocols and applications, without modifying applications.

The source code to Mahimahi and results from our case study on the Web are available at `http://mahimahi.mit.edu/`. Additionally, we have open-sourced Cumulus and the corresponding measurements at `http://web.mit.edu/anirudh/cumulus`.

Bibliography

[1] http://www.chromium.org/spdy/link-headers-and-server-hint.

[2] http://www.opera.com/turbo.

[3] https://developer.chrome.com/multidevice/data-compression.

[4] http://man7.org/linux/man-pages/man8/ip-netns.8.html.

[5] https://tools.ietf.org/html/rfc2616.

[6] https://developers.google.com/speed/spdy/.

[7] http://http2.github.io/.

[8] https://developers.google.com/speed/pagespeed/module.

[9] http://www.opera.com/mobile/mini.

[10] http://en.wikipedia.org/wiki/MarioNet_split_web_browser.

[11] http://amazonsilk.wordpress.com/.

[12] http://www.level3.com/.

[13] http://www.alexa.com/topsites.

[14] https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.

[15] http://phantomjs.org/.

[16] http://linux-ip.net/html/nat-dnat.html.

[17] https://chrome.google.com/webstore/detail/data-compression-proxy/ajfiodhbiellfpcjjedhmmmpeeaebmep?hl=en.

[18] http://www.appneta.com/blog/gogo-slow-airplane-wifi/.

[19] http://www.united.com/web/en-US/content/travel/inflight/wifi/default.aspx.

[20] http://www.gogoair.com/gogo/splash.do.

[21] http://mahimahi.mit.edu/.

[22] Analysis of SPDY and TCP initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.

[23] Android emulator. http://developer.android.com/tools/devices/emulator.html.

[24] Multipath TCP. http://multipath-tcp.org/.

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

[26] Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[27] Stream Control Transmission Protocol. http://www.ietf.org/rfc/rfc2960.txt.

[28] web-page-replay. http://code.google.com/p/web-page-replay.

[29] WebPagetest batch processing APIs. https://sites.google.com/a/webpagetest.org/docs/advanced-features/webpagetest-batch-processing-apis.

[30] Akamai. State of the Internet. http://www.akamai.com/stateoftheinternet/, 2013.

[31] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known Content Network (CN) Request-Routing Mechanisms, 2003. IETF RFC 3568.

[32] Kenneth L Calvert et al. Modeling Internet topology. *IEEE C.M.*, 35(6):160–163, 1997.

[33] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM CCR*, 40(2):12–20, 2010.

[34] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing tcp's initial congestion window. *SIGCOMM Comput. Commun. Rev.*, 40(3):26–33, June 2010.

[35] J. Erman et al. Towards a SPDY'ier mobile web? In *CoNEXT*, 2013.

[36] A. Sivakumar et al. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In *HotMobile*, 2014.

[37] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: The virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 159–170, New York, NY, USA, 2013. ACM.

[38] Bryan Ford. Structured streams: A new transport abstraction. In *SIGCOMM*, 2007.

[39] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[40] A. Jurgelionis et al. An empirical study of netem network emulation functionalities. In *ICCCN*, 2011.

[41] Bob Lantz et al. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*, 2010.

[42] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010.

[43] J. Padhye and H. F. Nielsen. A comparison of SPDY and HTTP performance. Technical report, Microsoft, 2012.

[44] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving http latency. *Comput. Netw. ISDN Syst.*, 28(1-2):25–35, December 1995.

[45] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. Tcp fast open. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 21:1–21:12, New York, NY, USA, 2011. ACM.

[46] Lenin Ravindranath et al. Automatic and scalable fault detection for mobile applications. In *MobiSys*, 2014.

[47] Jim Roskind. Experimenting with QUIC. `http://blog.chromium.org/2013/06/experimenting-with-quic.html`.

[48] Jim Roskind. QUIC: Multiplexed stream transport over UDP. `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit` .

[49] Ashiwan Sivakumar, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, Subhabrata Sen, and Oliver Spatscheck. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, HotMobile '14, pages 21:1–21:6, New York, NY, USA, 2014. ACM.

[50] Limin Wang, Kyoung Soo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and security in the codeen content distribution network. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[51] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 473–486, Berkeley, CA, USA, 2013. USENIX Association.

[52] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is spdy? In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[53] Xiao Sophia Wang, Haichen Shen, and David Wetherall. Accelerating the mobile web with selective offloading. In *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, MCC '13, pages 45–50, New York, NY, USA, 2013. ACM.

[54] Keith Winstein et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.