Mahimahi: Accurate Record-and-Replay for HTTP

Ravi Netravali[*], Anirudh Sivaraman[*], Somak Das[*], Ameesh Goyal[*], Keith Winstein[†], James Mickens[‡], Hari Balakrishnan[*]
[*]MIT CSAIL [†]Stanford University [‡]Harvard University
{ravinet, anirudh, somakrdas, ameesh, hari}@csail.mit.edu, keithw@cs.stanford.edu, mickens@eecs.harvard.edu

Abstract

This paper presents Mahimahi, a framework to record traffic from HTTP-based applications, and later replay it under emulated network conditions. Mahimahi improves upon prior record-and-replay frameworks in three ways. First, it is more accurate because it carefully emulates the *multi-server* nature of Web applications, present in 98% of the Alexa US Top 500 Web pages. Second, it *isolates* its own network traffic, allowing multiple Mahimahi instances emulating different networks to run concurrently without mutual interference. And third, it is designed as a set of *composable shells*, providing ease-of-use and extensibility.

We evaluate Mahimahi by: (1) analyzing the performance of HTTP/1.1, SPDY, and QUIC on a corpus of 500 sites, (2) using Mahimahi to understand the reasons why these protocols are suboptimal, (3) developing Cumulus, a cloud-based browser designed to overcome these problems, using Mahimahi both to implement Cumulus by extending one of its shells, and to evaluate it, (4) using Mahimahi to evaluate HTTP multiplexing protocols on multiple performance metrics (page load time and speed index), and (5) describing how others have used Mahimahi.

1 Introduction

HTTP is the de facto communication protocol for client-server applications today [27]. Beyond its widespread use as an application-layer protocol for loading Web pages, HTTP is now used for mobile apps [22], video streaming [14], and instant messaging [19].

It is useful to evaluate the performance of these applications under controlled experimental conditions. For example, browser developers may wish to evaluate how changes to their document object model (DOM) and JavaScript parsers affect Web page load times, while network-protocol designers might want to understand the application-level impact of new multiplexing protocols like QUIC [30]. Similarly, a mobile app developer may wish to determine the user-perceived latency [28] for user interactions over different wireless networks.

Motivated by such questions, we developed Mahimahi[1], a framework to record traffic from applications that use HTTP, and later replay recorded traffic under emulated network conditions. Mahimahi works with any application that uses HTTP or HTTPS. Application clients (Web browsers, video players, and apps within mobile-phone emulators) can be run unmodified within Mahimahi. Additionally, Mahimahi's replay semantics can be extended to support the server-side logic of many applications, such as YouTube.

Mahimahi has three notable features that distinguish it from other record-and-replay tools such as Google's web-page-replay [11] and Fiddler [34]:

1. **Accuracy:** Mahimahi is careful about emulating the multi-server nature of Web applications. Instead of responding to all requests from a single server, Mahimahi creates a separate server for each distinct server contacted while recording. We find that emulating multiple servers is a key factor in accurately measuring Web page load times (§4.1).
2. **Isolation:** Using Linux's network namespaces [7], Mahimahi isolates its traffic from the rest of the host system, allowing multiple instances of its shells to run in parallel with no mutual interference (§4.2). Because other tools modify the network configuration of the entire host [11, 34], they cannot provide this feature.
3. **Composability and extensibility:** Mahimahi is structured as a set of UNIX shells, allowing the user to run unmodified client binaries within each shell. *RecordShell* allows a user to record all HTTP traffic for any process spawned within it. *ReplayShell* replays recorded content using local servers that emulate the application servers. To emulate network conditions, Mahimahi includes *DelayShell*, which emulates a fixed network propagation delay, and *LinkShell*, which emulates both fixed-capacity and variable-capacity links. These shells can be nested within one another, allowing the user to flexibly experiment with many different network configurations. Mahimahi makes it easy to modify these shells and add new ones; e.g., to record-and-replay YouTube videos, emulate packet losses, implement active queue management algorithms, etc. (§4.3).

We used Mahimahi to evaluate Web multiplexing protocols. We were able to easily extend Mahimahi to support QUIC, a new protocol in active development at Google. We compared HTTP/1.1, SPDY [3], and QUIC to a hypothetical optimal protocol and found that all three are suboptimal. We then used Mahimahi to understand the shortcomings of these multiplexing protocols. We found that each protocol is suboptimal because of the request serialization caused by source-level object dependencies present in today's Web pages. Resolving each dependency requires an RTT between the client and ori-

[1]Mahimahi was previously introduced in a demo [23].

Configuration	HTTP/1.1	SPDY	QUIC-toy	**Cumulus**	Optimal
1 Mbit/s, 120 ms	8.7, 15.0	8.6, 12.6	*7.6, 10.8*	**6.4, 9.8**	5.3, 8.8
14 Mbits/s, 120 ms	4.3, 6.0	3.9, 5.6	*3.8, 5.4*	**2.4, 3.6**	1.8, 2.9
25 Mbits/s, 120 ms	4.3, 6.0	3.9, 5.4	*3.6, 4.9*	**2.0, 3.2**	1.7, 2.7

Table 1: Median, 75%ile page load times, in seconds, for the Alexa US Top 500 sites for different link rates and the same minimum RTT (120 ms). Comparing the median page load times, Cumulus is between 18-33% of the hypothetical optimal, outperforming the *best of the other schemes* (shown in each row in italics) by between 19% to 80% in these configurations. Moreover, we show later that as RTT grows, the gap from optimal for HTTP/1.1, SPDY and QUIC grows quickly, whereas Cumulus is a lot closer to optimal.

gin Web servers; Mahimahi allowed us to pinpoint the problem because we were able to conduct a large number of emulation experiments under different network conditions quickly.

We used these findings to develop *Cumulus*, a new system to improve HTTP application performance, especially on long-delay paths. Cumulus has two components: the "Remote Proxy," a headless browser that the user runs on a well-provisioned cloud server, and the "Local Proxy," a transparent, caching HTTP proxy that runs on the user's computer. These two components cooperate to move the resolution of object dependencies closer to origin Web servers, reducing the effective RTT. Mahimahi's shell structure allowed us to implement Cumulus with ease by adapting RecordShell to implement the Local Proxy.

To evaluate Cumulus, we used Mahimahi yet again, this time on the same large number of network configurations used to understand HTTP/1.1, SPDY, and QUIC. Our key result is that page load times with Cumulus do not degrade dramatically with increasing round-trip times (RTTs), unlike the other multiplexing protocols. Some representative results are shown in Table 1. We have also evaluated Cumulus on AT&T's live cellular network in Boston, finding that it outperforms existing Web accelerators such as Opera Turbo [1] and Chrome Data Compression Proxy [15].

Mahimahi has been used in other projects, including an analysis of mobile app traffic patterns to compare single-path and multi-path TCP [13], and an evaluation of intelligent network selection schemes [12]. Mahimahi has also been used in Stanford's graduate networking course [41] and at Mozilla to understand and improve networking within browsers. Mahimahi and our experimental data are available under an open source license at `http://mahimahi.mit.edu`. Mahimahi has been queued for inclusion with the Debian distribution.

2 Related Work

This section describes prior work on Web record-and-replay tools and network emulation frameworks.

2.1 Record-and-replay tools

The most prominent Web page record-and-replay tools are Google's web-page-replay [11] and Telerik's Fiddler [34]. web-page-replay uses DNS indirection to intercept HTTP traffic during both record and replay, while Fiddler adjusts the system-wide proxy settings in the Windows networking stack. With both tools, all HTTP requests from a browser are sent to a proxy server that records the request and forwards it to the corresponding origin server. Responses also pass through the proxy server and are recorded and sent back to the browser.

Both tools suffer from two shortcomings. First, because they serve all HTTP responses from a single server, neither tool preserves the multi-server nature of Web applications. Consolidating HTTP resources onto a single server during replay allows browsers to use a single connection to fetch all resources, which is impossible when resources are on different servers. Mahimahi faithfully emulates the multi-server nature of Web applications, leading to more accurate measurements (§4.1).

Second, these tools do not provide isolation: the network conditions that web-page-replay and Fiddler emulate affect all other processes on the host machine. These include the link rate, link delay, and DNS indirection settings for web-page-replay, and the system proxy address, specified in the Windows networking stack, for Fiddler. During replay, this lack of isolation could lead to inaccurate measurements if cross traffic from other processes reaches the replaying proxy server. The lack of isolation also precludes multiple independent instances of web-page-replay or Fiddler from running concurrently—a useful feature for expediting experiments, or for experimenting with different applications concurrently. Mahimahi overcomes these problems by using Linux's network namespaces [7].

Other record-and-replay tools such as Timelapse/Dolos [8] and WaRR [6] target reproducible application debugging by capturing program executions (including user input and activity) and replaying them, while providing popular debugging abstractions including breakpoints. These systems are complementary to Mahimahi; they can be run within ReplayShell, which ensures that served HTTP content, including dynamic content such as JavaScript, does not vary during replay.

2.2 Emulation Frameworks

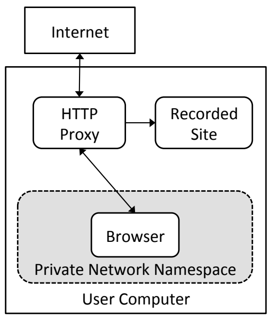
Tools like dummynet [10] and netem [20] emulate network conditions including link rate, one-way delay, and stochastic loss. Mahimahi uses its own network emulation shells, LinkShell and DelayShell. Unlike dummynet and netem, LinkShell can emulate variable-rate cellular links, in addition to static link rates, because it runs over packet-delivery traces. Mahimahi also allows users 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 for dummynet, a more complicated task.

Mahimahi is general enough to record and replay any HTTP client-server application under emulated conditions. It is, however, limited in that it only emulates one physical client connected to an arbitrary number of servers. Mahimahi supports a single shared link from the client to all servers, as well as multi-homed clients (§5.5), allowing the evaluation of multipath-capable transport protocols such as MPTCP [25]. Mahimahi cannot emulate arbitrary network topologies such as transit-stub [9]; for emulating applications over such topologies, tools like Mininet [21] are more suitable.

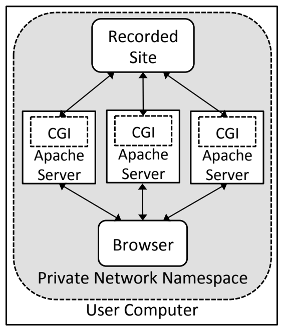
3 Mahimahi

Mahimahi is structured as a set of four UNIX shells, allowing users to run unmodified client binaries within each shell. 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" [7]. 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 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 each other. For example, to replay recorded content over a cellular network with a 10 ms minimum RTT, one would run a client application inside DelayShell inside LinkShell inside ReplayShell.



(a) RecordShell



(b) ReplayShell

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

3.1 RecordShell

RecordShell (Figure 1a) records HTTP data 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 a 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 server's address for the connection, and connects to the server on the application's behalf. An HTTP parser running at the proxy captures 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, associating the request with the response. At the end of a record session, a recorded directory consists of a set of files, one for each HTTP request-response pair seen during that session.

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 server. The proxy can establish a secure connection with the application in two ways. In the first approach, RecordShell's proxy uses a new Root CA, in the same way Fiddler does [35]. Clients must manually trust this CA once and individual certificates are signed by this Root CA.

Another approach is for RecordShell's proxy to use a self-signed certificate. This approach may trigger warnings within applications that only accept certificates signed by any one of a list of trusted Certificate Authorities (CAs). Most modern browsers allow users to disable these warnings. Certain applications, such as mobile phone emulators, do not allow these warnings to be disabled; the first approach handles these applications [31].

3.2 ReplayShell

ReplayShell (Figure 1b) also runs on the test machine and mirrors the server side of Web applications using content recorded by RecordShell. ReplayShell accurately emulates the multi-server nature of most Web applications today by spawning an Apache 2.2.22 Web server for each distinct IP/port pair seen while recording. Each server handles HTTPS traffic using Apache's `mod_ssl` module and may be configured to speak HTTP/1.1 or SPDY (using `mod_spdy`).

To operate transparently, ReplayShell binds each Apache server to the same IP address and port number as its recorded counterpart. To do so, ReplayShell creates a separate dummy (virtual) interface for each distinct server IP. These interfaces can have arbitrary IPs because they are in a separate network namespace.

All client requests are handled by one of ReplayShell's servers, each of which can read all of the previously recorded content. Each server redirects all incoming requests to a CGI script using Apache's `mod_rewrite` module. The CGI script on each server compares each incoming HTTP request to the set of all recorded request-response pairs to locate a matching request and return the corresponding response. Incoming requests may be influenced by local state present in the client application (e.g. time-sensitive query string parameters) and may not exactly match any recorded request. We handle such requests using a matching heuristic that enforces that some parts of the request must match exactly, while tolerating some degree of imperfection in other parts.

We expect the Host and User-Agent header fields, along with the requested resource (without the query string), to exactly match the corresponding values in some stored request. If multiple stored requests match on these properties, the algorithm selects the request whose query string has the maximal common substring to the incoming query string.

3.3 DelayShell

DelayShell emulates a link with a fixed minimum one-way delay. All packets sent 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.

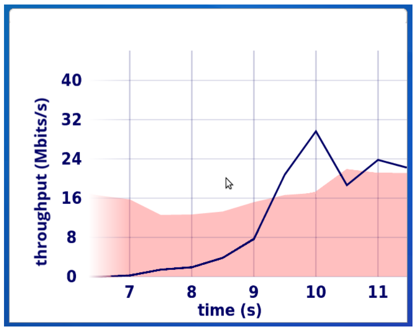


Figure 2: LinkShell supports live graphing of network usage, comparing the link capacity of the input traces (red shading) to the amount of data a client application attempts to transmit (blue line).

3.4 LinkShell

LinkShell emulates a link using packet-delivery traces. It emulates both time-varying links such as cellular links and links with a fixed link rate. 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.[2] 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.

LinkShell supports live graphing of network usage and per-packet queuing delay, giving near-instantaneous feedback on the performance of applications and network protocols. Uplink and downlink capacity are calculated using the input packet-delivery traces, while network usage, in each direction, is based on the amount of data that a client application attempts to transmit or receive. Per-packet queuing delay is computed as the time each packet remains in LinkShell's uplink or downlink queues.

Figure 2 illustrates the downlink network usage of a single Web page load of `http://www.cnn.com`, using Google Chrome over an emulated Verizon LTE cellular network with a minimum RTT of 100 ms. As shown, Web servers try to exceed the link capacity at around 9.3 seconds into the trace.

[2]For example, a link that can pass one MTU-sized packet per millisecond (12 Mbits/s) can be represented by a file that contains just "1" (LinkShell repeats the trace file when it reaches the end).

4 Novelty

Mahimahi introduces three new features in comparison to existing record-and-replay tools. We describe each of these in greater detail below.

4.1 Multi-server emulation for greater accuracy

A key component of ReplayShell is that it emulates the multi-server nature of Web applications. As discussed in §3, ReplayShell creates a network namespace containing an Apache server for each distinct server encountered in a recorded directory. We show through three experiments that emulating this multi-server nature is critical to the accurate measurement of Web page load times.

A large number of websites today are multi-server. We measure the number of physical servers used by each site in the the Alexa US Top 500 [5]. We find that the median number of servers is 20, the 95%ile is 51, and the 99%ile is 58. Only 9 of the 500 Web pages (1.8%) we consider use a single server.

Next, we illustrate the importance of preserving the multi-server nature of Web applications by comparing measurements collected using ReplayShell and web-page-replay to real page load times on the Internet. To obtain measurements on the Internet, we use Selenium to automate Google Chrome loading 20 Web pages from the Alexa US Top 500, 25 times each, inside a LinkShell of 5 Mbits/s and a DelayShell with a minimum RTT of 100 ms. We chose a minimum RTT of 100 ms to equalize delays to Web servers contacted while loading each Web page.[3] For a fair comparison, we record copies of each Web page with RecordShell and web-page-replay immediately following the completion of these Internet measurements; Web content can change frequently, which can significantly affect page load time. We then replay each recorded Web page 25 times using ReplayShell, a modified version of ReplayShell that serves all resources from a single server, and web-page-replay. With ReplayShell, we perform each page load inside LinkShell with a 5 Mbits/s trace and DelayShell with a minimum RTT of 100 ms, as described above. We emulate these same network conditions with web-page-replay.

We define the error, per site, as the absolute value of the percent difference between mean page load times (over 25 runs) within an emulation environment and on the Internet. As shown in Figure 3, ReplayShell with multi-server emulation yields page load times that most accurately resemble page load times collected on the Internet. The median error is 12.4%, compared to 36.7% and 20.5% with web-page-replay and single-server ReplayShell, respectively.[4]

[3]The 20 sites used here are all hosted by CDNs in close proximity with ping times of less than 5 ms.

[4]We are not certain why single-server ReplayShell is so much more accurate than web-page-replay.

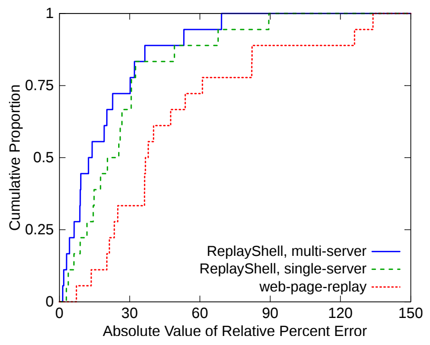


Figure 3: Preserving a Web page's multi-server nature yields measurements that more closely resembles measurements on the Internet.

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

Table 2: Median and 95%ile difference in page load time without multi-server emulation.

Finally, we run more exhaustive experiments to show the effect that multi-server emulation has on Web page load times across different network conditions. Using an Amazon EC2 m3.large instance located in the US-east-1a region and running Ubuntu 13.10, we measure page load times for each recorded page in the Alexa US Top 500 when loaded with Google Chrome. We consider 9 different configurations: link rates in $\{1, 14, 25\}$ Mbits/s and RTTs in $\{30, 120, 300\}$ ms. We load each page over each configuration using both ReplayShell and the modified version of ReplayShell used above that eliminates the multi-server nature altogether by setting up one Apache server to respond to all HTTP requests and resolving all DNS queries to that server alone.

Table 2 shows the median and 95%ile difference in page load time when multi-server nature is not preserved, compared to when multi-server nature is preserved. Although the page load times are comparable over a 1 Mbit/s link, the lack of multi-server emulation yields significantly worse performance at higher link rates.

4.2 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 running directly on the host or in other namespaces. 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 prop-

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

Table 3: Mean and standard deviation for page load times across two similarly configured machines.

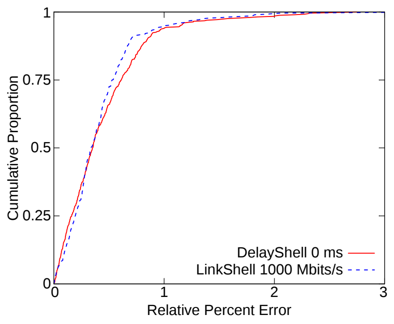


Figure 4: DelayShell and LinkShell have a negligible effect on page load times in ReplayShell.

erty of Mahimahi, along with the fact that its shells can be arbitrarily nested, enables many different configurations to be simultaneously tested on a host machine, in complete isolation from one another.

Using distinct network namespaces for each shell also enables Mahimahi to produce reproducible results while imposing low overhead on collected measurements.

Reproducibility: To evaluate the reproducibility of Mahimahi's measurements, we perform repeated experiments on the same host machines and across different host machines with similar hardware specifications. We choose two sites from the Alexa US Top 500 for this experiment, `http://www.cnbc.com/` and `http://www.wikihow.com/`, as they are are at the median and 95%ile 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/s link with a minimum RTT of 120 ms. Table 3 shows a summary of the distribution of page load times from these experiments. 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, standard deviations are all within 1.6% of their corresponding means, implying that Mahimahi produces consistent results on a single host machine.

Fidelity: Mahimahi's shells impose low overhead on collected measurements, even when they are nested within one another, leading to high fidelity in the results. We illustrate this property in Figure 4, which shows the overhead DelayShell and LinkShell impose on page load time measurements. We first load our recorded copies of the Alexa US Top 500 sites inside ReplayShell, without LinkShell or DelayShell. For comparison, we then load the 500 sites inside DelayShell, with 0 ms fixed per-packet delay, inside ReplayShell. Separately, we load the 500 sites inside LinkShell, with 1000 Mbits/s uplink and downlink traces, inside ReplayShell.[5] 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 median per-site errors with DelayShell and LinkShell, relative to ReplayShell alone, are 0.33% and 0.31%, respectively.

4.3 Composability and extensibility

Unmodified application clients can be run within any of Mahimahi's shells. For instance, as described in §5.5, a mobile device emulator can be run within Mahimahi to measure mobile app performance. Similarly, to measure new performance metrics such as the speed index, virtual machines can be run within Mahimahi's shells (§5.4).

The default replay algorithm is but one instance of a server-side HTTP matching algorithm. Mahimahi's replay semantics can be easily extended to support the server-side logic of many other applications and multiplexing protocols; for example, in §5.1.1, we extend ReplayShell to use QUIC Web servers rather than default Apache Web servers. It has also been extended to handle record-and-replay for YouTube videos (§5.5).

In addition to DelayShell and LinkShell, which emulate different minimum RTTs and link rates, Mahimahi can be extended to support other network characteristics. For example, to emulate different levels of stochastic packet loss, we created *LossShell* [24], which probabilistically drops packets stored in LinkShell's upstream and downstream queues. Similarly, Mahimahi can be modified to evaluate in-network algorithms such as queuing disciplines. By default, LinkShell implements a Drop Tail FIFO queue, but we have extended it to implement CoDel, an active queue management scheme [32].

Mahimahi could also be used to replay recorded content to a different physical machine. Consider a scenario where the application to be evaluated is only available on Machine M, and a separate Linux Machine, A, is available. An *EthShell* could ferry packets from an Ethernet interface between M and A to a virtual network interface on A. Analogously, a *UsbShell* could 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 rather than emulators. Neither of these has been developed yet, but Mahimahi's design allows these shells to be nested inside any of Mahimahi's existing shells. For instance, to test a mobile phone's browser over an LTE link with a 100 ms RTT, we would nest UsbShell inside DelayShell inside LinkShell inside ReplayShell.

[5]We chose 1000 Mbits/s to ensure that link capacity was not a limiting factor in page load time.

5 Case Studies

5.1 Understanding Web Performance

We use Mahimahi to evaluate Web page load times under three multiplexing protocols: HTTP/1.1, SPDY [3], and QUIC [30], a protocol currently in development at Google. To put these measurements in context, we compare each protocol with an optimal protocol for each network configuration.

To automate the page load process and measure page load times, we use Selenium, a widely used browser-automation tool, along with Chrome Driver version 2.8 and the Web Driver API [38]. We measure page load time by calculating the time elapsed between the navigationStart and loadEventEnd events [38].

In all evaluations, traffic originates from the Web browser alone. We emulate link rates and minimum RTTs (§3), but do not emulate competing cross traffic. For each network configuration, we emulate a buffer size of 1 bandwidth-delay product and evaluate all sites in the Alexa US Top 500.

5.1.1 Setup

HTTP/1.1: We evaluate HTTP/1.1 using ReplayShell running unmodified Apache 2.2.22.

SPDY: To evaluate SPDY, we create *SPDYShell*, which enables the `mod_spdy` extension on all Apache servers within ReplayShell. The SPDY configuration evaluated here does not include server push because the push policy is specific to each website and is hard to infer automatically. If push policies were known, however, the CGI script within ReplayShell's servers could be modified to reflect them.

QUIC: QUIC inherits several SPDY features, such as multiplexing streams onto a single transport-protocol connection and stream priorities. By using UDP and its own security instead of TCP and TLS, QUIC overcomes two drawbacks of SPDY: head-of-line blocking between streams due to lost packets and the three-way handshake required to establish a secure connection.

Unlike SPDY, Apache currently has no extensions for QUIC. We create *QUICShell* by replacing Apache within ReplayShell with an adapted version of the QUIC toy server [29] from the Chromium project (commit *5bb5b95* from May, 2015, available at `https://goo.gl/Jdr8hi`). We modify the toy server, which originally searched for exact URL matches, to use the matching semantics in ReplayShell's CGI script.

5.1.2 Optimal page load time

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

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

The first term represents the minimum time between when the first HTTP request is made at the client and the first byte of the first HTTP response is received by the client, ignoring processing time at the server.

The second term represents the minimum time 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.

The third term represents the time for the browser to process all the HTTP responses and render the Web page (using the definition of "loaded" above). We measure this as the page load time in ReplayShell alone without network emulation, emulating an infinite-capacity, zero-delay link.

5.1.3 Canonical network results

We evaluate each protocol on 110 configurations: link rates in $\{0.2, 0.3, 0.6, 1, 1.7, 2.9, 5, 8.5, 14, 25\}$ Mbits/s and RTTs between 0 ms and 300 ms in steps of 30 ms. These link rates and RTTs cover the majority of global network conditions reported by Akamai [4]. We also perform evaluations over cellular networks using modified versions of the Verizon and AT&T traces collected in [40]. For each network configuration, we compare HTTP/1.1, SPDY, and QUIC (and in the next subsection, Cumulus) with the optimal page load times defined above.

Figure 5 shows the distributions of page load times with each protocol for six of these configurations: 1 Mbit/s and 25 Mbits/s, with RTTs of 30 ms, 120 ms, and 300 ms. We find that the gap from optimal for HTTP/1.1, SPDY, and QUIC grows quickly with the RTT, and grows with the link rate (although not as quickly). For example, on a 1 Mbit/s link with a minimum RTT of 30 ms, the median page load time for SPDY is 1.08$\times$ worse than optimal. When the minimum RTT increases to 120 ms, the median SPDY page load time is 1.63$\times$ worse than optimal, worsening to 2.02$\times$ worse than optimal when the minimum RTT rises to 300 ms. For this RTT, increasing the link rate from 1 Mbit/s to 25 Mbits/s degrades median SPDY performance to 4.93$\times$ worse than optimal.

5.1.4 Understanding suboptimality

In addition to quantifying the extent of suboptimality of multiplexing protocols for the Web, the results presented in this case study corroborate the qualitative findings of many previous measurement studies [26, 37, 39]. We used Mahimahi in conjunction with browser developer tools to identify the root cause of this suboptimality. We found that the suboptimal performance of each multiplexing protocol is a result of request serialization caused by source-level dependencies between objects on a Web page; this problem is exacerbated by small limits on the number of concurrent connections from the browser, but persists even if those browser limits are removed.

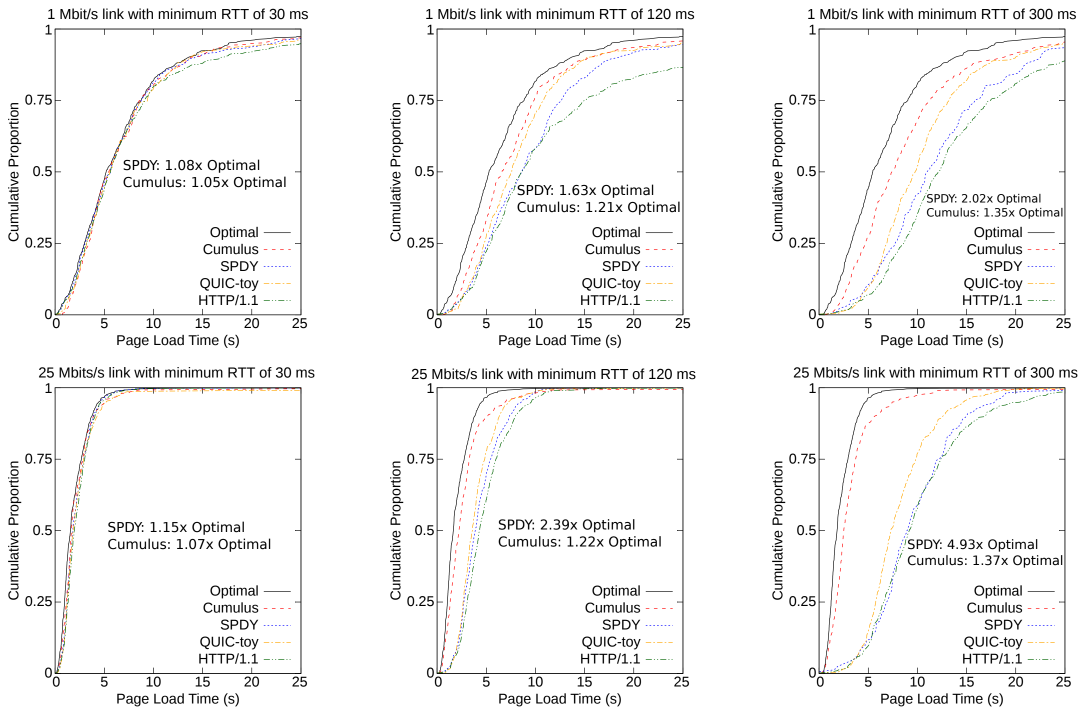


Figure 5: The gap between page load times with HTTP/1.1, SPDY, or QUIC and Optimal grows as link rate or minimum RTT increases. Cumulus is introduced in §5.2.

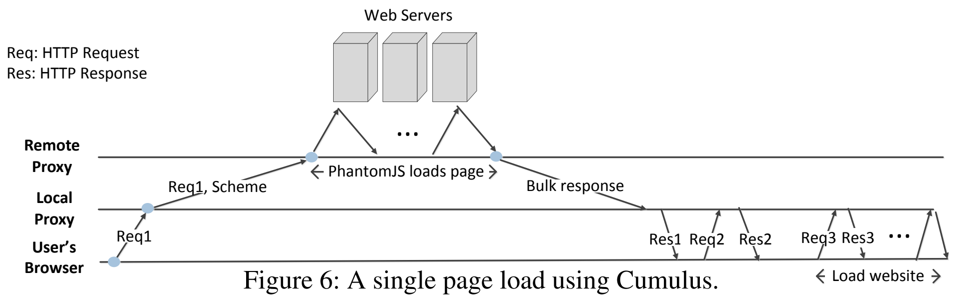


Figure 6: A single page load using Cumulus.

The fundamental issue is that resolving each dependency requires a round-trip communication between the client and origin Web servers. As a result, the negative effect of request serialization is more pronounced at high RTTs (Figure 5). This finding motivated us to develop Cumulus, a system that uses Mahimahi to improve page load times on long-delay paths.

5.2 Improving Web performance with Cumulus

Cumulus has two components: the "Remote Proxy," a headless browser that the user runs on a well-provisioned cloud server, and the "Local Proxy," a transparent, caching HTTP proxy that runs on the user's computer. These two components cooperate to move the resolution of object dependencies closer to origin Web servers—reducing the effective RTT—without modifying Web browsers or servers.

The Remote Proxy listens for new requests from the Local Proxy. For each incoming request, the Remote Proxy launches an unmodified RecordShell and runs a headless browser, PhantomJS [2], to load the specified URL using the original HTTP headers. Once the page is loaded, the Remote Proxy packages and compresses the recorded HTTP request/response pairs into a *bulk response*, which it sends to the Local Proxy.

The Local Proxy is a modified version of RecordShell that caches HTTP objects rather than storing them in files. When the user's browser requests a URL not resident in the Local Proxy's cache, the Local Proxy forwards the request to the Remote Proxy. Upon receiving a bulk response from the Remote Proxy, the Local Proxy responds to the user's browser with the appropriate response and caches the remaining objects to handle subsequent browser requests. Figure 6 illustrates how Cumulus loads a single Web page.

5.3 Evaluating Cumulus with Mahimahi

We first evaluate Cumulus over each emulated network configuration listed in §5.1.3. Page loads with Cumulus used Google Chrome and a Remote Proxy running on the

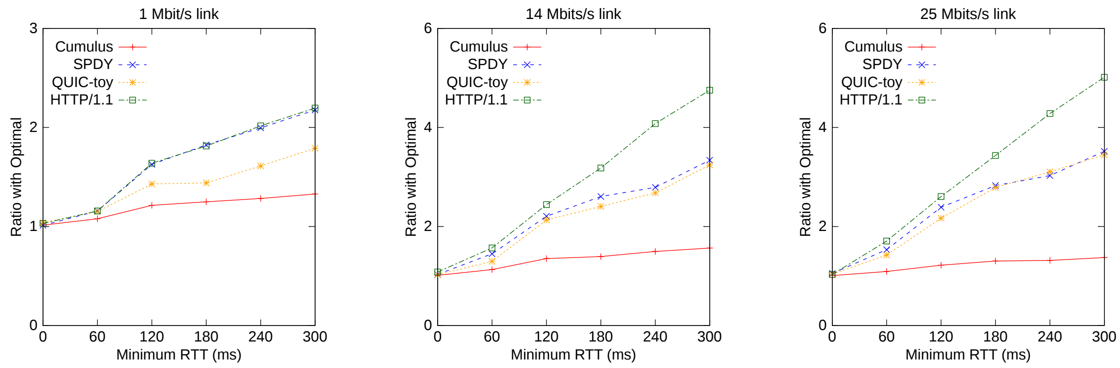


Figure 7: Cumulus's performance does not degrade dramatically as RTTs increase (at fixed link rates), unlike HTTP/1.1, SPDY, and QUIC. Each point plots the ratio of median protocol performance to median performance of the optimal scheme (lower is better).

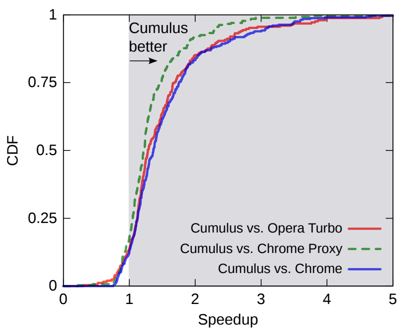


Figure 8: Evaluating Cumulus on the live AT&T Cellular Network in Boston.

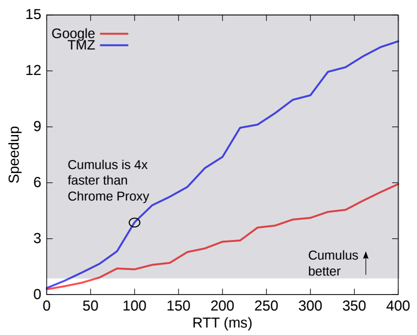


Figure 9: Benefits with Cumulus increase as RTT or Web page complexity increase.

other side of each emulated link. We find that Cumulus outperforms SPDY by 1.03–3.60× over these configurations (Figure 5). Figure 7 shows how the ratio between median page load times with each protocol and the optimal varies as RTTs increase at fixed link rates. We find that Cumulus is less affected by increases in RTT compared to today's multiplexing protocols. For example, at a link rate of 14 Mbits/s and an RTT of 60 ms, Cumulus is 1.13× worse than optimal while SPDY is 1.44× worse than optimal. When RTT increases to 180 ms, Cumulus is 1.39× worse, whereas SPDY is 2.61× worse than optimal.

5.3.1 *Some live experiments*

We also compare the performance of Google Chrome run inside Cumulus with Chrome, and with Chrome Data Compression Proxy [15, 16] and Opera Turbo [1], which are cloud browsers that use proxy servers for compression. We load each page in the Alexa US Top 500 five times with each system, rotating among the systems under test to mitigate the effects of network variability. We define Cumulus's "speedup" relative to a system as the ratio of the page load time using that system to the page load time using Cumulus.

We ran experiments over the live AT&T LTE/GSM/WCDMA cellular network in Boston using a PC laptop tethered to a Samsung Galaxy Note running Android OS version 4.2.2. Cumulus used a Remote Proxy running on an Amazon EC2 instance in Virginia. Cumulus had median speedups of 1.36×, 1.23×, and 1.28× over Chrome, Chrome Data Compression Proxy, and Opera Turbo, respectively. Figure 8 shows the CDF of speedups.

5.3.2 *Understanding Cumulus' gains*

Cumulus moves dependency resolution to the Remote Proxy where RTTs to Web servers are lower than from the client. The benefit of this technique depends on:

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

To understand the importance of each factor, we use Mahimahi's shell abstraction to load two Web pages in emulation: TMZ's homepage with 508 objects and the Google homepage with only 15 objects. We use DelayShell to emulate fixed minimum RTTs from 0 ms to 400 ms. For each RTT, we load each page five times with Chrome Data Compression Proxy—which compresses objects in-flight, but does not perform dependency resolution on the user's behalf—and Cumulus, which performs dependency resolution *and* compresses objects in-flight.

Page loads with Cumulus used a Remote Proxy running on the other side of the emulated long-delay link. Speedups for Cumulus relative to Chrome Data Compression Proxy are shown in Figure 9.

We observe two trends:

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

Our results show a 4× speedup relative to Chrome Data Compression Proxy at an RTT of 100 ms, a typical RTT for cellular and transcontinental links. This corroborates the well-known intuition that Web page load times are dominated by network latencies rather than link rates, and suggests that the combination of remote dependency resolution and object compression helps Cumulus achieve performance not far from optimal.

5.4 Speed index

All of our measurements thus far have been of page load time. We now show that it is straightforward to use a different performance metric. We use Google's proposed speed index [17] as an example.

5.4.1 Definition

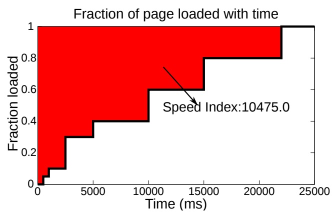
Page load time may not accurately measure when a page is usable by the client. For long Web pages, content "above-the-fold" of the screen is important to retrieve quickly, but other content may not be. Taking this point into consideration for measurement is especially relevant for pages that support infinite scrolling. For example, Facebook "preloads" wall postings below the user's current location on its page in anticipation of a user scroll. In such cases, the "onload" event used to measure page load time would fire long after the page is ready for user interaction. Speed index is an attempt to address this issue.

Speed index tracks the visual progress of a Web page in the visible display area. A lower speed index signifies that the content 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.

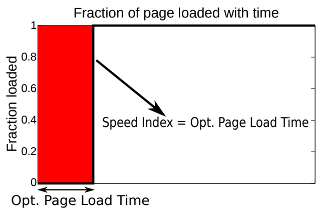
Speed index is calculated by measuring the completeness of a page's display area over time. Completeness is defined as the pixel-by-pixel difference of a page snapshot with the final loaded Web page. Once the entire page has loaded, the completeness percentage of the page rendering over time is plotted. Speed index is defined as the area "above-the-curve" (Figure 10a).

5.4.2 Measuring speed index

We calculate speed index using WebPagetest [17], which records videos of page loads at 10 frames per second



(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 10: Speed index calculation.

and plots the percentage completeness over time by comparing each frame with the final captured frame. To measure speed index, we create *SpeedIndexShell* where we run a private instance of WebPagetest inside ReplayShell. To automate testing, we use WebPagetest's `wpt_batch.py` API [18]. Because WebPagetest runs only on Windows, we run WebPagetest within a VirtualBox Windows virtual machine, inside ReplayShell.

5.4.3 Optimal speed index

Calculating an optimal speed index is difficult. Instead, we define an upper bound[6] on the optimal speed index. We assume that a site renders in one shot at the optimal page load time; Figure 10b illustrates its implications on 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 completeness jumps to 100%. As a result, the "area above the curve," or optimal speed index, equals the optimal page load time. There could 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 (Figure 11) from the optimal speed index.

5.4.4 Static link results

We measure the speed index for each site in the Alexa US Top 500 over networks with link rates between 1 Mbit/s and 25 Mbits/s and a fixed minimum RTT of 120 ms (Figure 11). 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 rates

[6]Recall that a lower speed index is better.

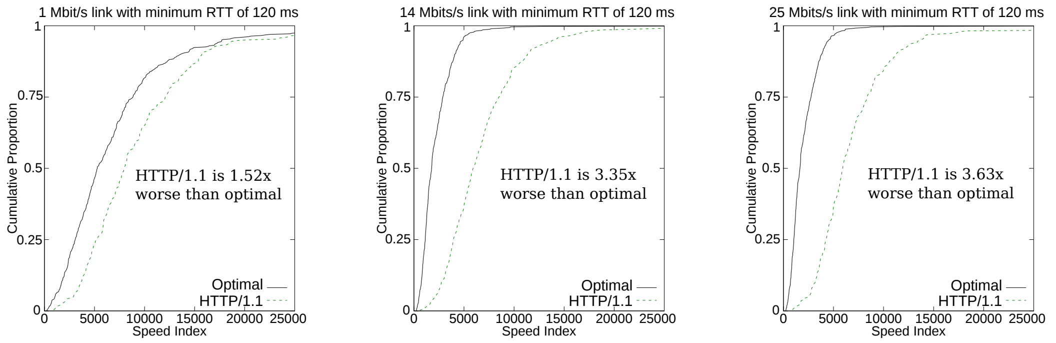


Figure 11: Gap between speed index with HTTP/1.1 and Optimal grows as link rate increases (fixed minimum RTT).

increase; over a 1 Mbit/s link with a 120 ms minimum RTT, speed index with HTTP/1.1 is 1.52× worse than optimal at the median, while over a 25 Mbits/s link with a 120 ms minimum RTT, the median speed index with HTTP/1.1 is 3.63× worse than optimal.

5.5 External case studies

This section describes external use cases of Mahimahi in research, educational, and industrial settings.

Mobile app record-and-replay: RecordShell has been used to characterize mobile app traffic by recording all HTTP traffic to and from mobile apps running inside an Android emulator [13]. Using this recorded traffic, they evaluated the performance of mobile apps over Wi-Fi and LTE networks by running an Android emulator inside ReplayShell to measure the duration of data transfers for mobile apps over these wireless networks. The results showed that LTE outperforms Wi-Fi 40% of the time on flow completion time.

Mobile multi-homing: To emulate mobile multi-homing with Wi-Fi and LTE, the authors in [12] extended LinkShell to create *MpShell* [33]. They then compared single-path TCP and MPTCP by replaying mobile app traffic over 20 different emulated network conditions.

Record-and-replay for video streaming: Mahimahi has been extended to handle record and replay for YouTube videos [36]. Compared to Web pages, video replay requires more involved matching logic on the server side. HTTP requests encode the location (start and end time) and quality of video chunks requested by the client's video player. Both the location and quality attributes can change significantly from run to run, and between record and replay.

Educational uses: Mahimahi is being used by students in Stanford's graduate networking course [41] to understand the performance of their networked applications under controlled conditions. As part of a protocol design contest conducted in the same course, students used LinkShell's live graphing of network usage and per-packet queuing delay to obtain real-time feedback on the performance of their congestion-control protocols.

Browser networking: Engineers at Mozilla are using Mahimahi to improve the speed of Firefox's networking. Here, Mahimahi is helpful in understanding how improvements to link utilization and pipelining of HTTP requests affect Web performance over various networks.

6 Conclusion

Mahimahi is an accurate and flexible record-and-replay framework for HTTP applications. Mahimahi's shell-based design makes it composable and extensible, allowing the evaluation of arbitrary applications and network protocols. It accurately emulates the multi-server nature of Web applications during replay, and by isolating its own traffic, allows several instances to run in parallel without affecting collected measurements.

We presented several case studies to evaluate Mahimahi and demonstrate its benefits. These include a study of HTTP/1.1, SPDY, and QUIC under various emulated network conditions. We used Mahimahi both to conduct the experiments and to understand the reasons for the suboptimality of these protocols. We then used our key finding—that these protocols are suboptimal due to source-level dependencies in Web pages—to design Cumulus. Mahimahi was useful in our implementation of Cumulus, as well as in our experiments to measure its performance. As round-trip times and link rates increase, the performance of Cumulus degrades much slower than previous HTTP multiplexing protocols.

We have released Mahimahi under an open source license at `http://mahimahi.mit.edu`.

7 Acknowledgements

We thank Amy Ousterhout, Pratiksha Thaker, the ATC reviewers, and our shepherd, Liuba Shrira, for their helpful comments and suggestions. This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-1407470. We thank the members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

References

[1] Opera Turbo. http://www.opera.com/turbo.

[2] PhantomJS. http://phantomjs.org/.

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

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

[5] Alexa. Top sites in the United States. http://www.alexa.com/topsites/countries/US.

[6] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, 2011.

[7] E. W. Biederman. ip-netns. http://man7.org/linux/man-pages/man8/ip-netns.8.html.

[8] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, 2013.

[9] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, 1997.

[10] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM CCR*, 40(2):12–20, 2010.

[11] Chromium. web-page-replay. https://github.com/chromium/web-page-replay.

[12] S. Deng. Intelligent Network Selection and Energy Reduction for Mobile Devices. http://people.csail.mit.edu/shuodeng/papers/thesis.pdf.

[13] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or both? Measuring multi-homed wireless Internet performance. In *IMC*, 2014.

[14] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube traffic characterization: A view from the edge. In *IMC*, 2007.

[15] J. Glowacki. Data compression proxy. https://chrome.google.com/webstore/detail/data-compression-proxy/ajfiodhbiellfpcjjedhmmmpeeaebmep.

[16] Google. Data compression proxy. https://developer.chrome.com/multidevice/data-compression.

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

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

[19] R. Jennings, E. Nahum, D. Olshefski, D. Saha, Z.-Y. Shae, and C. Waters. A study of Internet instant messaging and chat protocols. *Network, IEEE*, 20(4):16–21, 2006.

[20] A. Jurgelionis, J. Laulajainen, M. Hirvonen, and A. Wang. An empirical study of netem network emulation functionalities. In *ICCCN*, 2011.

[21] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*, 2010.

[22] K. Ma, R. Bartos, S. Bhatia, and R. Nair. Mobile video delivery with HTTP. *Communications Magazine, IEEE*, 49(4):166–175, 2011.

[23] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement (demo). In *SIGCOMM*, 2014.

[24] R. Netravali, K. J. Winstein, and A. Sivaraman. LossShell. https://github.com/ravinet/mahimahi/tree/lossshell.

[25] C. Paasch, S. Barre, et al. Multipath TCP in the Linux kernel. http://multipath-tcp.org/.

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

[27] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *Hotnets*, 2010.

[28] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.

[29] J. Roskind. Experimenting with QUIC. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

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

[31] N. Rudrappa. Defeat SSL Certificate Validation for Google Android Applications. Technical report, McAfee, 2013.

[32] A. Sivaraman, R. Netravali, and K. J. Winstein. CodelShell. https://github.com/ravinet/mahimahi/releases/tag/old/codel.

[33] A. Sivaraman, R. Netravali, and K. J. Winstein. MPShell. https://github.com/ravinet/mahimahi/

tree/old/mpshell_scripted.

[34] Telerik. Fiddler. http://www.telerik.com/fiddler.

[35] Telerik. Fiddler documentation. http://docs.telerik.com/fiddler/Configure-Fiddler/Tasks/TrustFiddlerRootCert.

[36] V. Vasiliev, R. Netravali, K. J. Winstein, and A. Sivaraman. YoutubeShell. https://github.com/vasilvv/mahimahi.

[37] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[38] Z. Wang and A. Jain. Navigation timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.

[39] G. White, J.-F. Mule, and D. Rice. Analysis of SPDY and TCP initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.

[40] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.

[41] K. J. Winstein. (Your) great ideas for networked applications. https://web.stanford.edu/class/cs344g/.