Evaluation of QUIC on Web Page Performance

by

Somak R. Das

S.B., Massachusetts Institute of Technology (2014)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2014

Signature redacted

Certified by..... .....
Hari Balakrishnan
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Evaluation of QUIC on Web Page Performance

by

Somak R. Das

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This work presents the first study of a new protocol, QUIC, on Web page performance. Our experiments test the HTTP/1.1, SPDY, and QUIC multiplexing protocols on the Alexa U.S. Top 500 websites, across 100+ network configurations of bandwidth and round-trip time (both static links and cellular networks). To do so, we design and implement QuicShell, a tool for measuring QUIC's Web page performance accurately and reproducibly.

Using QuicShell, we evaluate the strengths and weaknesses of QUIC. Due to its design of stream multiplexing over UDP, QUIC outperforms its predecessors over low-bandwidth links and high-delay links by 10 – 60%. It also helps Web pages with small objects and HTTPS-enabled Web pages. To improve QUIC's performance on cellular networks, we implement the Sprout-EWMA congestion control protocol and find that it improves QUIC's performance by $> 10\%$ on high-delay links.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

Acknowledgments

First and foremost, I thank Prof. Hari Balakrishnan for his guidance and feedback over the course of my research. It was a cold and windy day in 2010 when I first walked into his office, inquiring about a UROP position. I have learned a tremendous amount since then.

Likewise, I am grateful for the supervision I received from my mentors in his group, who were (in chronological order) Lenin Ravindranath, Tiffany Chen, Anirudh Sivaraman, and Ravi Netravali. I also thank Keith Winstein and Ameesh Goyal for useful discussion, along with the rest of my labmates in the CSAIL Networks and Mobile Systems Group, who made the weekly group meetings enjoyable.

Last but definitely not least, I am forever grateful to my friends and family for their support.

Contents

List of Figures

List of Tables

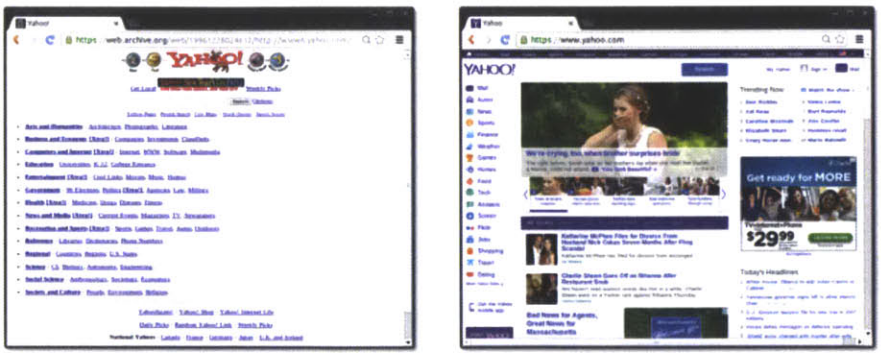
Chapter 1

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 (Section 2.1.1 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, including network protocol designers who seek to understand the application-level impact of new multiplexing protocols like QUIC [13] and website developers wishing to speed up access to their Web properties.

In this study, we compare two established multiplexing protocols, HTTP/1.1 and SPDY, to one in development by Google, QUIC. To the best of our knowledge, we present the first study to do so. This is not an easy task: (i) QUIC is in active development and is not widely available, (ii) websites have a large variety in components (see Figure 1-1 for just one example of how they have changed), and (iii) users are accessing the Web in many ways (cellular networks, wireless and wired links).



(a) Yahoo in 1996 **(b)** Yahoo in 2014

Figure 1-1: Like many other websites, Yahoo's homepage has become considerably more complicated over the years. It has added more objects, such as dynamic scripts, images, and ads. This can degrade Web page performance.

Furthermore, real Web page loads are difficult to model because Web pages have complex dependencies between their objects (scripts, images, ads, etc.) and involve both networking (transferring objects) and computation (processing and displaying objects).

Our approach is to sweep the parameter space and evaluate where QUIC currently helps or hurts. More specifically, our experiments load the Alexa U.S. Top 500 websites in a real Web browser using the three multiplexing protocols and measure the page load time. They test 100+ network configurations of bandwidth and round-trip time (RTT or delay), including both static links and cellular networks. To do so in an accurate and reproducible manner, we design and implement QuicShell, a tool for emulating websites over QUIC.

Due to its design of stream multiplexing over UDP, QUIC outperforms its predecessors over low-bandwidth links and high-RTT links by 10 – 60%. It also helps Web pages with small objects and HTTPS-enabled Web pages. To improve QUIC's performance on cellular networks, we implement the Sprout-EWMA congestion control protocol and find that it improves QUIC's performance by $> 10\%$ on high-RTT links.

Chapter 2

Related Work

As noted before, there exist no other QUIC studies to compare with this work. Therefore, in this chapter, we survey the protocols to study. They are organized in two levels. *Multiplexing protocols* like HTTP/1.1, SPDY, and QUIC are at the higher application level. *Congestion control algorithms* like TCP Cubic, WebRTC Inter-Arrival, and Sprout-EWMA are at the lower transport level.

2.1 Multiplexing Protocols

Starting with persistent HTTP [5], researchers have grappled with the problem of mapping HTTP requests to the underlying transport level. Older work includes the Stream Control Transmission Protocol (SCTP) [15] 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 [7] also allows the application to multiplex different streams onto one network connection, and operates hierarchically, by allowing streams to spawn streams of their own. Deployed work in this space includes Google's SPDY [3] and their recent proposal, QUIC [13], each of which we evaluate in Section 5.1.

2.1.1 HTTP/1.1

HTTP/1.1 (shortened to HTTP) is the *de facto* application-level protocol of the Web. Web browsers (clients) send HTTP requests to the server and receive HTTP responses. The requests and responses are data accompanied by header fields. Since Web pages can be composed of many objects (text, images, etc., each with a different URL) browsers open a new TCP connection for each request and response.

Several proposed modifications to HTTP/1.1 aim to improve page load times. HTTP Server Push uses existing connections to push modified data from the server to the client. This modification targets sites where content changes quickly. Similarly, HTTP pushlets take advantage of HTTP persistent connections to fool the browser into continually loading the page. This allows a Web server to update dynamic content even if the page has already loaded once. HTTP long polling allows the server to respond to requests whenever it receives new data. Finally, prefetching allows a client to fetch resources in advance of when they are needed and might often transmit a lot of unnecessary data [11]. However, we do not individually evaluate these modifications due to the lack of standardized and authoritative implementations for any of these techniques.

2.1.2 SPDY

In 2009, Google proposed a new multiplexing protocol called SPDY (and pronounced "speedy") to be built over HTTP/1.1; today, its design is the basis of the upcoming HTTP/2.0 standard [3]. Instead of opening a new TCP connection for each request and response, Web browsers only use a single TCP connection for SPDY. This saves the round-trip costs establishing new connections, which are especially expensive when the connection is secured (for HTTPS) by Transport Layer Security (TLS), and is good for high-RTT links.

In order to work over a single connection, each request/response pair is abstracted as a stream, and streams are multiplexed (combined) into packets to be sent over the wire. As a result, several requests can load simultaneously. Unlike HTTP pipelining,

where multiple HTTP/1.1 requests are sent on a single TCP connection, the requests do not have to return in first-in-first-out order. Therefore, if a request is particularly slow for the server to respond to, it does not hold up the other requests (avoiding *head-of-line blocking*).

The benefits of stream multiplexing over a TCP connection also include reduced packet count, since many small objects can be combined into a single packet to be sent. It also allows better compression of stream (HTTP request and response) headers, since many headers are likely to include the same lines, such as User-Agent: Mozilla/5.0. This is good for low-bandwidth links. However, the issue is that in the event of a packet loss, the whole TCP connection (and all the streams) is affected; whereas in HTTP/1.1, only one TCP connection is affected by the packet loss, but the others running in parallel continue without seeing that loss.

To evaluate SPDY, we enable the `mod_spdy` extension on Apache servers. SPDY allows two other features, request priorities (where server schedules the order of responses) and server push (where the server pushes responses even before the client requests them), but the SPDY configuration evaluated here does not have them. This is 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 [17] corroborates this intuition; it shows that the benefits of SPDY under such a server push policy are slight.

2.1.3 QUIC

QUIC (pronounced "quick") is Google's new (first proposed in 2013) multiplexing protocol that that runs over UDP [13], attempting to further improve Web performance compared with SPDY. Since it runs stream multiplexing, it inherits the same benefits as SPDY. However, because QUIC runs over UDP, it has better resilience to packet loss and a faster start, as explained in Section 3.1.

2.2 Congestion Control Algorithms

Congestion control algorithms schedule when packets are sent on the wire to and from the client, which directly impacts Web page performance. Many have been designed over the years, but we study the ones specifically implemented in QUIC.

2.2.1 TCP Cubic

TCP Cubic is the default TCP implementation in Linux. It maintains a congestion window, which determines how many packets can be sent and outstanding. The window grows with successful packet deliveries and shrinks with failed packet deliveries (i.e., packet losses). Cubic increases its window as a cubic function (instead of, say, linearly) and thus aggressively achieves high throughput.

Google recently proposed several modifications to TCP to enhance its performance on Web page loads [10, 4]. These include increasing the initial TCP congestion window from 4 to 10 (the current default on Ubuntu) and TCP Fast Open, which enables data exchange during TCP's SYN/ACK handshake. Both suggestions are motivated by the short-flow nature of the Web where a low initial congestion window and the latency incurred by having to wait for the initial handshake can significantly increase flow completion times, and by implication, page load times. More recently, Google proposed Gentle Aggression [6], a technique to combat tail losses that adversely affect page load times. We do not evaluate these modifications to TCP in this case study since they are not the current defaults; instead, we run HTTP/1.1 and SPDY using the default TCP implementation in Ubuntu 13.10, Cubic with an initial window of 10.

2.2.2 WebRTC Inter-Arrival

Besides implementing Cubic, QUIC supports Inter-Arrival protocol (formally called Receiver-side Real-Time Congestion Control) from WebRTC (a communication platform separate from QUIC). This takes a very different approach to congestion control

that Cubic's window. Instead of reacting to failed packet deliveries, Inter-Arrival proactively paces out packets at a certain rate. It measures the current sending rate and estimates the available bandwidth from the link; if bandwidth is available, then it increases the sending rate. However, a previous study [8] showed that the Inter-Arrival's implementation in WebRTC was not robust to different network conditions.

2.2.3 Sprout-EWMA

Sprout is an end-to-end transport protocol for interactive applications that require high throughput and low delay [19]. It is targeted toward cellular networks, which differ from traditional links by having highly-variable bandwidths and dropping no packets. Sprout is a window-based protocol like Cubic, but it calculates the receiving rate similar to Inter-Arrival. Specifically, Sprout-EWMA takes packet arrival times at the receiver to estimate the network throughput and applies an exponentially-weighted moving average to smooth the estimate. It is a simpler version of Sprout, which uses flicker-noise modeling and Bayesian inference to make a prediction instead. It multiplies the rate by a target delay to calculate the window.

We chose to implement Sprout-EWMA because it achieved the same throughput/delay ratio as Sprout but better throughput. In Section 6.3, we compare its performance to the status quo, TCP Cubic.

Chapter 3

QUIC

QUIC (**Q**uick **U**DP **I**nternet **C**onnections) is Google's new multiplexing protocol that runs over UDP, attempting to further improve Web performance compared with SPDY [13]. It inherits SPDY's features, such as 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's 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. We

enumerate these benefits in greater detail below, since no study has discussed QUIC before (though a technical design document from Google is available [13]).

3.1 Design

The primary goal of QUIC is to be deployable on today's internet. In order to do that but still improve on TCP, QUIC runs stream multiplexing over UDP. Although this means that QUIC has to re-implement TCP's reliable delivery, it is able to modify the semantics for Web page loads: unlike TCP, it is not bound to in-order byte delivery.

Since UDP is connectionless, QUIC establishes its own application-level secure connection. This is identified by a globally unique ID instead of client/server IP/port pairs. GUIDs are good for mobile clients that roam between networks, like Wi-Fi and cellular, and change their IP addresses. By have a GUID instead of IP addresses, clients using QUIC do not have to re-establish a new connection when roaming.

QUIC sends packets (more accurately, datagrams) over UDP, and these packets contain frames. The most common frame is *Stream Frames*, that contain data for a QUIC stream (a HTTP request/response pair) identified by a unique stream ID. *ACK Frames* are acknowledgments for stream data. They contain the highest contiguous sequence number seen so far and, optionally, a list of missing sequence numbers. *Congestion Feedback Frames* contain receiver-to-sender feedback for the underlying congestion control algorithm like TCP Cubic, WebRTC Inter-Arrival, or Sprout-EWMA. A packet to or from the client can contain any combination of these frames, which is less costly than sending the frames in individual packets.

These are the features of the QUIC design:

Stream multiplexing As in SPDY, this reduces the packet count and allows for better compression of stream headers. The result is consuming less bandwidth.

UDP resilience to loss In SPDY, if a packet was lost, then all following packets cannot be delivered to the application because TCP guarantees in-order byte

delivery. However, it is very likely that the lost packet contains data for one stream and the following packets contain data for other streams. The lost packet is holding up the other streams in a form of head-of-line blocking. QUIC, over UDP, can bypass this and deliver data for other streams.

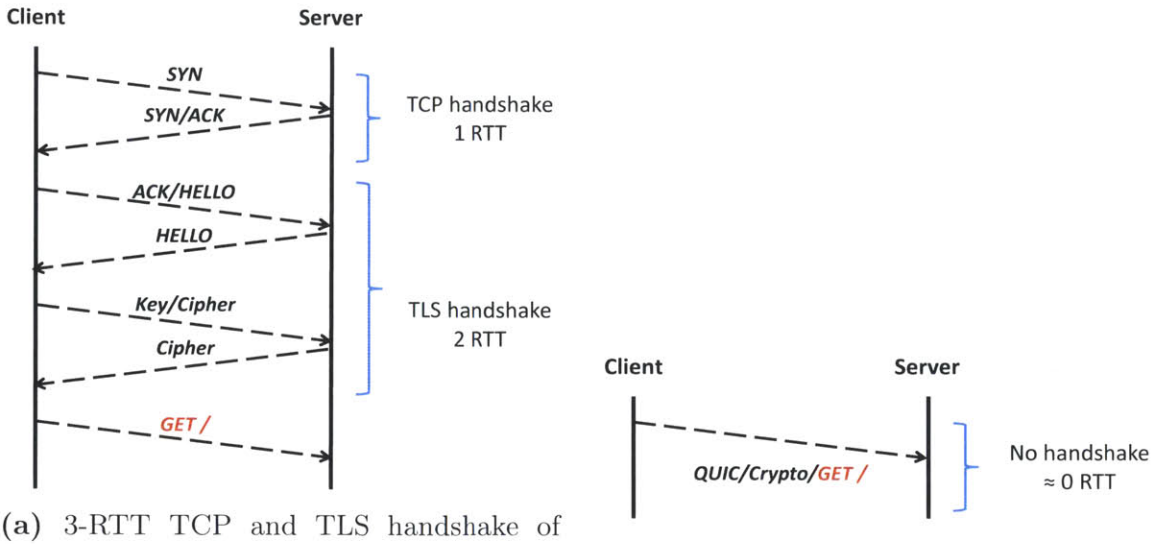
FEC resilience to loss When a packet is lost over TCP, the client has to wait for the sender to detect the loss and resend the lost packet. QUIC proactively sends a redundant packet so that the client can reconstruct a lost packet by itself. It uses the Forward Error Correction (FEC) technique, where it XORs a FEC group of n packets and sends the XORed FEC packet. If a single packet is lost, then the client can recover it using the XORed packet. The speed of this approach offsets the $\frac{n+1}{n}$ bandwidth cost.

TLS-like security QUIC provides as much security as HTTPS. In fact, it also encrypts all packets under HTTP too.

Low-cost, 0-RTT start As shown in Figure 3-1, QUIC aims to establish a secure connection with 0-RTT overhead. Instead of 1-RTT TCP handshake followed by a 2-RTT TLS handshake, it initializes the security and the connection at the same time, and also allows the client to optimistically send data in the first packet.

Pluggable congestion control Because QUIC implements its own reliable delivery instead of relying on TCP implementations in the kernel, it allows switching between congestion control algorithms. The current default is TCP Cubic, but it allows both window-based and packet pacing-based algorithms like WebRTC Inter-Arrival and Sprout-EWMA.

To summarize, QUIC has stream multiplexing and compression for low-bandwidth links, reduced head-of-line blocking and FEC for lossy links, low-cost starts for high-RTT links, and roaming support and low overhead for mobile clients. We evaluate



(a) 3-RTT TCP and TLS handshake of HTTP(S) and SPDY

(b) 0-RTT start of QUIC

Figure 3-1: Overhead of HTTP, SPDY, and QUIC connection establishment. Compared to HTTP and SPDY, QUIC provides a low-cost, 0-RTT start, so clients can (securely) request Web pages sooner.

these features in Section 5.1. We also experiment with our own congestion control algorithm in Section 6.3.

3.2 Implementation

QUIC is implemented by adding a framing layer to the network stack between HTTP and the underlying UDP transport. As intended, it works with today's Internet and is enabled in the Chrome browser (Dev and Canary builds) using the `--enable-quic` and `--enable-quic-https` flags.

It is developed as a library inside the Chromium project's repository, not as a separate project. This enables quick deployment tied to the Chrome browser. The QUIC library in `net/quic/` is currently 38,784 lines of C++ code and includes `congestion_control` (for pluggable congestion control) and `crypto` (for built-in TLS-like security) modules. It is accompanied by a QUIC "toy server" (also known as the test server or prototype server) in `net/tools/quic/`, taking 4,332 lines of code. The toy server is a simple in-memory server that handles requests whose URLs

exactly match one of its stored objects.

Since QUIC is in active development, two of its features are not yet fully implemented, so we do not test them in this study:

1. FEC is not available since there is no a canonical heuristic [12]. Some proposals include calculating the XORed FEC packet near the start of a stream, to reduce processing delay of stream headers; near the end of a stream, to reduce stream completion delay; or near the end of a burst, to deal with tail (end-of-burst) loss. But the current FEC implementation in QUIC only allows enabling FEC groups strictly of size n for redundancy or code rate of $\frac{n}{n+1}$, which does not satisfy any of these strategies. In addition, n is currently set to 0 to disable FEC.

2. There is no working implementation of WebRTC Inter-Arrival's packet-pacing algorithm.

Chapter 4

Experimental Design

Our goal is to measure Web page performance of QUIC versus its predecessors HTTP and SPDY over a wide variety of network conditions and Web pages. We want to answer the question: where does QUIC currently help or hurt? To do so, these are the factors we consider:

- **Network conditions**
  - *Bandwidth*: Does QUIC's header compression and low-overhead start help low-bandwidth connections?
  - *Round-trip time*: Does QUIC's 0-RTT start help high-RTT connections?
- **Web pages**
  - *Number of objects*: Does QUIC's stream multiplexing help large numbers of objects?
  - *Object sizes*: Does QUIC's stream multiplexing help small objects?

However, there are two main technical challenges when setting up these experiments. The first is *availability*. The only servers on the Web that understand QUIC are the Google's properties that include Alternate-Protocol: 80:quic in their HTTP responses, such as `https://www.google.com/` and `https://www.youtube.com/`. But Google only represents 4 of the Alexa U.S. Top 500 websites ($< 1\%$) [2] and this

does not provide variety in Web pages. Moreover, we found that many of Google's websites do not provide a page load consistently over QUIC because they fall back to HTTP/1.1 and SPDY. The second is reproducibility. Measurements can have high variability on the Web, which makes it difficult to compare these protocols when the variance is greater than difference in their performance.

So, instead of conducting experiments on the real, we conduct experiments using Web page emulation. In this process, we record a Web page (save the content during a real page load) and replay it (serve that content during the emulated page load) over emulated network conditions (a chosen bandwidth and minimum RTT). This addresses availability because we extend QUIC to a much wider corpus of websites (all of the Alexa U.S. Top 500 websites, as explained in Section 4.3) and compare it to HTTP/1.1 and SPDY. Emulation also addresses reproducibility, because we select a record-and-replay framework that has been verified to produce reproducible Web measurements.

The initial difficulty was that while some frameworks support HTTP and SPDY, no framework supports QUIC. So, the following sections describe how we modify an existing framework, Mahimahi [9], to measure the page load time of Web pages served over QUIC.

4.1 Mahimahi

Mahimahi is a recently proposed Web measurement toolkit that records websites and replays them under emulated network conditions [9]. It is structured as a set of four UNIX shells:

1. **RecordShell** records Web content (from a website, mobile application, etc.), writing a folder of objects.

2. **ReplayShell** replays Web content, reading a folder of objects.

3. **DelayShell** emulates a link that adds a fixed one-way delay d to both directions of the link. This increases the minimum RTT by $2 \cdot d$.

4. **LinkShell** emulates a link using packet-delivery traces, which either specify static links of bandwidth r or dynamic links that vary bandwidth over time. We use this feature to test cellular networks in Section 6.3.

It has several benefits over previous work such as Google's web-page-replay: has low overhead, produces reproducible and accurate (preserving the multi-origin nature of Web pages by mirroring all unique server IP/port pairs) measurements, has composable shells, and is isolated from other test environments on the same machine.

However, the most important reason we chose Mahimahi is its extensibility. It replays by starting Web servers which talk to a CGI script which finds a matching response from the folder of recorded content for an incoming request. As explained in Section 4.2, we replace the existing Web server (Apache 2.4.6) with the QUIC server and added support for CGI scripting in the QUIC server.

4.2 QuicShell

Chrome browser has experimental support for QUIC on the client side. On the server side, ReplayShell currently uses Apache to replay content over HTTP/1.1 and Apache with the `mod_spdy` module to replay content over SPDY. However, there is currently no `mod_quic` module available for Apache, and we do not have access to the front-end QUIC servers for Google's properties. Therefore we invent our own solution, modifying the QUIC toy server from the Chromium project (commit `a8f23c`). The major obstacle was that the toy server cannot replay the folder of recorded content because it cannot perform the complicated matching needed to match a request to a response. We modify the QUIC toy server to support CGI scripting, so that it can use the same CGI script that the Apache server uses. Put together, we create **QuicShell**, which replays Web content over QUIC.

Modifying QUIC toy server 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

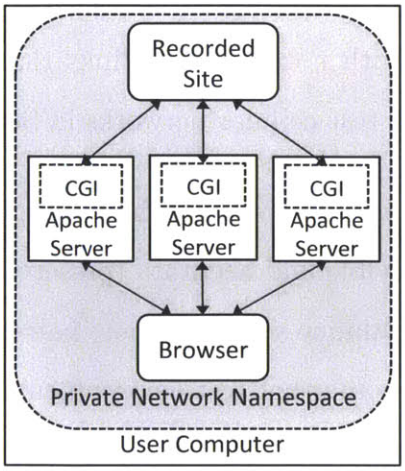


Figure 4-1: Mahimahi's ReplayShell handles all HTTP traffic inside a private network namespace. Arrows indicate direction of HTTP request and response traffic. To create QuicShell, we replace the Apache server (which does not support QUIC) with a modified QUIC server.

as values. It then finds the stored request which matches the incoming request URL exactly. This is insufficient for replaying Web pages.

Since we need more complicated matching of HTTP/1.1 request header fields, we rewrote the server to support the matching semantics in ReplayShell's CGI script. 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.

The CGI script uses environment variables as input. For example, if the server reads the request header field Accept-Language: en-US, then it sets `HTTP_ACCEPT_LANGUAGE=en-US`. Our goal is to match Apache's support for CGI scripting (via the `mod_rewrite` module) as closely as possible. To do so, we implement a URL parser to separate the incoming request URL into the correct parts, and we pass the remaining request header fields as environment variables. For example, say the incoming request is:

```
GET https://www.google.com/search?q=mit HTTP/1.1
```

It also has header fields for cookies, compression encodings, languages, etc., which we pass directly into the environment variables `HTTP_COOKIE`, `HTTP_ACCEPT_ENCODING`, `HTTP_ACCEPT_LANGUAGE`, etc. Our parser separates the request into:

1. `REQUEST_URI=www.google.com/search?q=mit`
2. `REQUEST_METHOD=GET`
3. `SERVER_PROTOCOL=HTTP/1.1` (since QUIC does not change the HTTP/1.1 content, only how it is sent over the wire)
4. `HTTP_HOST=www.google.com`
5. `SCRIPT_NAME=/search`

The host is important for accurate matching, since the recorded content may contain multiple hosts (as the Web is multi-origin). On the other hand, the script name (the URL path without the parameters like `?q=mit`) is important for approximate matching. We want as many requests to be accurately matched to responses as possible, and sometimes a dynamically generated URL contains the current timestamp, which is not present in the recorded content. The CGI script uses the script name to perform a longest-substring match across URLs.

Finally, we change 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.

Modifying ReplayShell For the last step of creating QuicShell, we modify ReplayShell to spawn a child process running a QUIC server for each IP/port pair (replacing the previous Apache server). As a result, a user simply has to run:

```
$ quicshell mit_recorded_objects/
QUIC server started on 18.181.0.46:80
[quicshell] $ chrome --enable-quic http://mit.edu/
```

Internally, QuicShell converts that to a child process executing:

```
quic_server --ip=18.181.0.46 --port=80
            --record_folder=mit_recorded_objects/
            --replay_server=replayserver.cgi
```

4.2.1 Benefits

QuicShell provides a QUIC emulation platform that meets both of our goals. For availability, we can record any website using RecordShell and replay it using QuicShell. Then, for reproducibility, we run an experiment to verify that QuicShell measurements are reproducible. Since `https://www.youtube.com/` is served over QUIC on the real Web, we measured its page load time (Section 4.4) over 100 runs. We also measured the emulated page load time using QuicShell. As shown in Figure 4-2, real measurements have much higher variance and are fat-tailed. The means and standard deviations are 718 ± 95 ms (over 13% deviation) for the real Web, compared to 743 ± 18 ms (only 2.5%) for the emulated version.

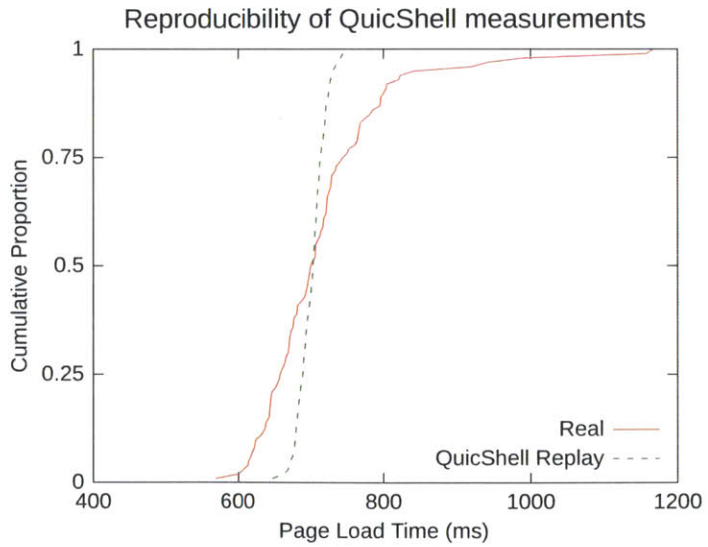


Figure 4-2: CDF of 100 runs loading YouTube over QUIC on the real Web vs. loading a recorded YouTube over QUIC using QuicShell.

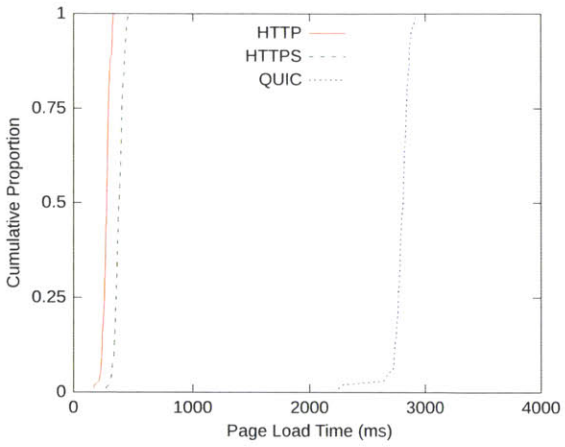
4.2.2 Limitations

As mentioned in Section 3.2, QUIC does not currently implement Forward Error Correction or WebRTC Inter-Arrival. Therefore, we test on TCP Cubic without FEC. Additionally, we build 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 minimize the impact by disabling logging.

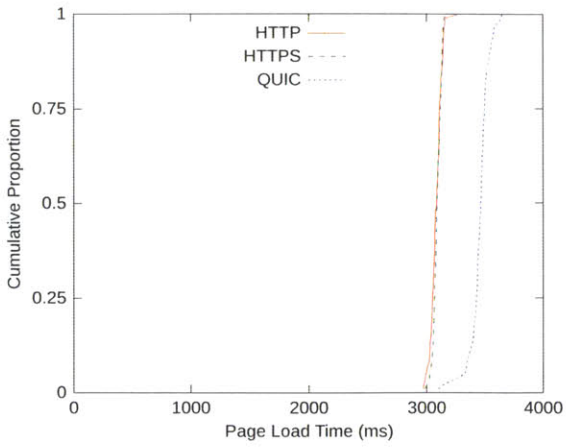
Still, there are a couple limitations that we identify with QUIC. First, because it is in active development, it may not be as fully optimized as the HTTP/1.1 implementation in Apache. The ideal case is an Apache `mod_quic` module, instead of the toy server, but that is not currently available. Second, it provides TLS-like privacy on HTTP (not just HTTPS). While we can enable no-TLS SPDY, QUIC always encrypts the packets to and from the client. Therefore there is overhead there, especially if the toy server is unoptimized. Only 8.6% of the Alexa U.S. Top 500 websites have HTTPS; for the other 91.4%, QUIC incurs extra encryption.

To gauge these two overheads, we measure the page load times of a simple single-object (8 MB) Web page over HTTP, HTTPS and QUIC over infinite and 14 Mbits/sec bandwidth links with zero minimum RTT. Thus, comparing HTTP and HTTPS shows the overhead of HTTPS's security. Comparing HTTPS and QUIC shows the overhead of QUIC's unoptimized implementation: in theory, the protocols should be comparable because a single object has no multiplexing, RTT is zero so QUIC's 0-RTT start does not give it an advantage, and both are using TCP Cubic for congestion control. However, in practice, the implementation in Apache vs. QUIC toy server differ a lot.

As shown in Figure 4-3a, security adds 1.4× overhead. QUIC is 7.3× worse, which we know is due to QUIC's TCP Cubic implementation having capped throughput (at roughly 25 Mbits/sec [16]). In Figure 4-3b, security much adds less ($< 0.5\%$) over a 14 Mbits/sec link because encryption and packet transmission can happen in parallel. However, even though the bandwidth is less than the measured 25 Mbits/sec cap,



(a) Infinite-bandwidth link with minimum RTT of 0 ms



(b) 14 Mbits/sec link with minimum RTT of 0 ms

Figure 4-3: HTTP, HTTPS, and QUIC page load times of a simple single-object (8 MB) Web page.

QUIC is still 12% slower.

4.3 Benchmark Corpus

We use Mahimahi's corpus (`https://github.com/ravinet/sites`) of the 500 most popular Web pages in the United States [2]. We add the results discussed in Section 5.1 to the corpus, so that we compare future iterations of QUIC to these benchmark measurements and track performance improvements or regressions. These measurements include page load times, recorded over more than one hundred network configurations, for each recorded site when HTTP/1.1, SPDY, and QUIC. Lastly, the corpus provides 3G and 4G cellular network traces for Verizon, AT&T, T-Mobile, and Sprint. These traces were originally collected in [19] and are modified to be compatible with LinkShell. We use them to test cellular networks in Section 6.3.

4.4 Page Load Time Metric

Page load time is defined as the time elapsed between two timing events, `navigationStart` and `loadEventEnd`, in the W3C Navigation Timing API [18].

`navigationStart` is the time at which a browser initiates a page load as perceived by the user, and `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. Note that bytes can continue to be transferred for the page load even after `loadEventEnd` [14].

To automate the page load process and measure page load times, we use Selenium 2.39.0 and ChromeDriver 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.

To ensure Chrome uses QUIC, we pass the `--enable-quic` and `--enable-quic-https` flags and modify Chrome to force QUIC on all origins. The previous behavior of the Chrome browser was to either force QUIC on a single origin, passed by the `--origin-to-force-quic-on` flag, or to first contact the server using HTTP and then, if the response contains Alternate-Protocol: *:quic, try QUIC which is too much overhead. Finally, we download the latest Flash Player plug-ins and configure Chrome to use them to support websites like `http://www.pandora.com/`, because the Chromium project does not contain any such plug-ins.

We use the WebDriver API to obtain timing information from Selenium and define page load time to be the time elapsed between the `navigationStart` and `loadEventEnd` events.

4.5 Putting It All Together

To emulate page loads over specific network conditions, we use QuicShell 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 QuicShell to setup a recorded website for replay over QUIC.

2. Within QuicShell, 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 the Chrome browser.

For each network configuration, we emulate a finite buffer size of 1 bandwidth-delay product and evaluate all sites stored in our corpus. Each of these experiments was performed on an Amazon EC2 m3.large instance, configured with Ubuntu 13.10 and located in the US-east-1a region.

The experiments sweep $10 \cdot 10 = 100$ configurations: 0.2 Mbps (3G link) to 25 Mbps (broadband link) bandwidth $\times$ 30 ms (intra-coast) to 300 ms (cross-continent) minimum RTT. For each network configuration, they cover all 500 recorded websites, providing a wide variety of Web page attributes. These are shown in Table 4.1.

Category	Factor	Range	Median
Network	Minimum RTT	30 – 300 ms	150 ms
	Bandwidth	0.2 – 25 Mbits/sec	3 Mbits/sec
Web page	Number of server addresses	1 – 66	22
	Number of objects	2 – 680	100
	Object size	53 bytes – 43 MB	3.0 kB
	Total size	15 kB – 51 MB	1.2 MB

Table 4.1: Full range of network conditions and Web pages studied in this work.

This setup achieves our objective of testing a variety of network conditions and Web pages. More importantly, this accurately forecasts QUIC on the real Web, where users are navigating to websites on their browsers (which load them, concurrently doing networking and computation) over many different links.

Chapter 5

Performance Evaluation

We evaluate HTTP/1.1, SPDY, and QUIC on the 500 websites over 100 network 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 [1]. Similarly, we selected the RTT range to incorporate a variety of different network conditions, from websites located within the same city to satellite links.

5.1 Results and Discussion

For a first analysis of the 150,000 data points, Figure 5-1 shows the 500-website distributions of page load times with HTTP/1.1, SPDY, and QUIC and the optimal page load times for 9 different 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}. On these 9 configurations, QUIC consistently performs 1 – 2× worse than SPDY. However, when compared to the most popular protocol used today, QUIC outperforms HTTP/1.1 on high-RTT links (all links with minimum RTT of 300 ms) as well as low-bandwidth links (1 Mbit/sec link with minimum RTT of 120 ms).

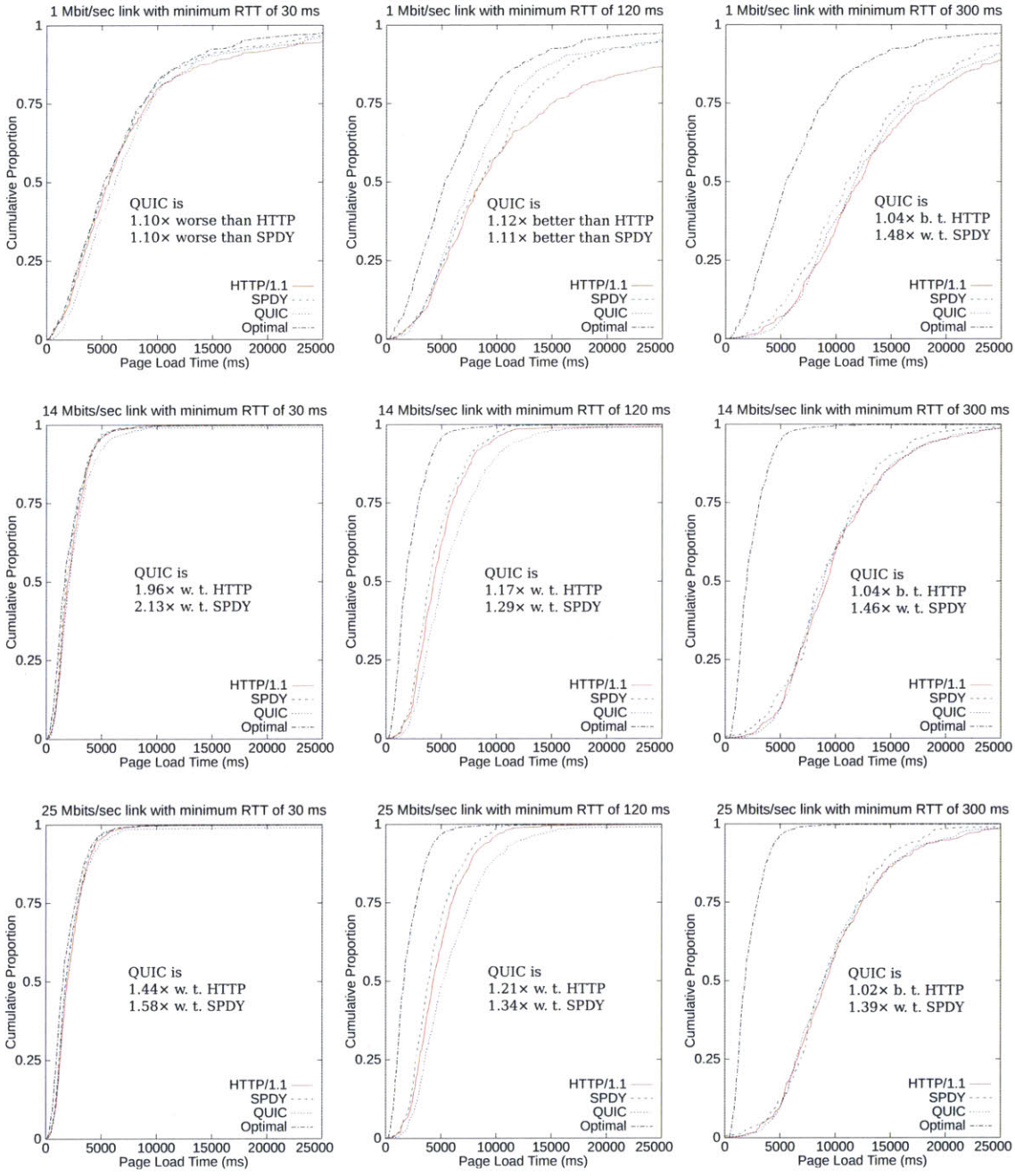


Figure 5-1: 500-website CDFs of HTTP/1.1, SPDY, and QUIC page load times across 9 bandwidth/RTT configurations: {1, 14, 24} Mbits/sec × {30, 120, 300} ms.

For a second analysis, our goal is to observe the overall trends across all 100 configurations, so that we can investigate high-RTT and low-bandwidth links further. In Figure 5-2, we plot heat maps showing the protocol improvement of 500-website median page load times across the 100 configurations. For instance, the bottom heat map shows QUIC's percent improvement in median page load time over SPDY. The color scale ranges from red (negative improvement: QUIC is worse than SPDY) to green (positive improvement: QUIC is better than SPDY).

This uncovered some interesting trends in the data. For each one, we explain why, based on our analysis.

1. *SPDY is almost always better than HTTP/1.1.* There is almost no performance regression. SPDY improves high-RTT links because it uses a single TCP connection for each server, so it avoids the overhead of establishing a TCP (and TLS) connection than HTTP/1.1 incurs.

2. *HTTP outperforms QUIC, which outperforms SPDY on very low bandwidth links.* For these (0.2 Mbits/sec), the finite buffer (whose size is set to the bandwidth/delay product) is very small, and it drops many packets. SPDY's single TCP connection ramps down due to the packet loss, but only one of HTTP/1.1's multiple connections is affected. QUIC is affected similarly, and so both QUIC and SPDY are worse than HTTP/1.1. However, QUIC outperforms SPDY because its UDP-based protocol avoids head-of-line blocking.

3. *QUIC is better than HTTP/1.1 on low-bandwidth links.* For these (from 0.3 to 1.0 Mbits/sec), QUIC's stream multiplexing and compression leads to less packets and smaller packets.

4. *QUIC improves as RTT increases.* As explained in Section 3.1, QUIC has a lower cost of establishing a connection by incurring less round trips: 0-RTT vs. 3-RTT start. For minimum RTTs > 210 ms, it is better than HTTP/1.1. The trend also holds against SPDY; QUIC comes closer to SPDY as RTT increases.

5. *QUIC needs to improve on high-bandwidth, low-RTT links, for which HTTP/1.1 is better.* In theory, QUIC improves on SPDY and thus should be better than both predecessors. However, for reasons discussed in Section 4.2.2, there are throughput caps and other inefficiencies in the current QUIC implementation. As expected, Apache's HTTP/1.1 and SPDY implementations are more performant.

5.2 Case Studies

In Section 5.1, we establish the network conditions QUIC performs well: primarily low-bandwidth, high-RTT links. Next, we study the specific Web pages for which QUIC helps or hurts. This is enabled by the wide variety of the 500 websites in our benchmark corpus, in terms of number of objects, object sizes, etc.

This final analysis is summarized in Table 5.1. For each of the 500 websites, we compute a score of how QUIC improves its page load time relative to HTTP/1.1, averaged across all 100 configurations. Then, we find the bottom 10 (most hurt by QUIC) and top 10 (most helped by QUIC) and list them with their Web page attributes. Finally, we compute the change in these attributes from the bottom 10 to the top 10.

For instance, the websites QUIC helps the most have a 12× smaller total size, on average, than the websites QUIC hurts the most. In fact, we find that the website's total size is the best predictor of whether QUIC helps or hurts it. This is followed by mean object size: QUIC's best have 4× smaller objects, on average, than QUIC's worst. We explain this by noting QUIC's unoptimized implementation compared to HTTP/1.1 and SPDY (Section 4.2.2). In particular, when the object is large, QUIC's built-in techniques like header compression and low-cost start matter less and raw throughput matters more, because a larger fraction of the load time is spent transferring the object (e.g., not establishing the connection).

Lastly, in Section 4.2.2, we noted the overhead of security. QUIC encrypts 100% of

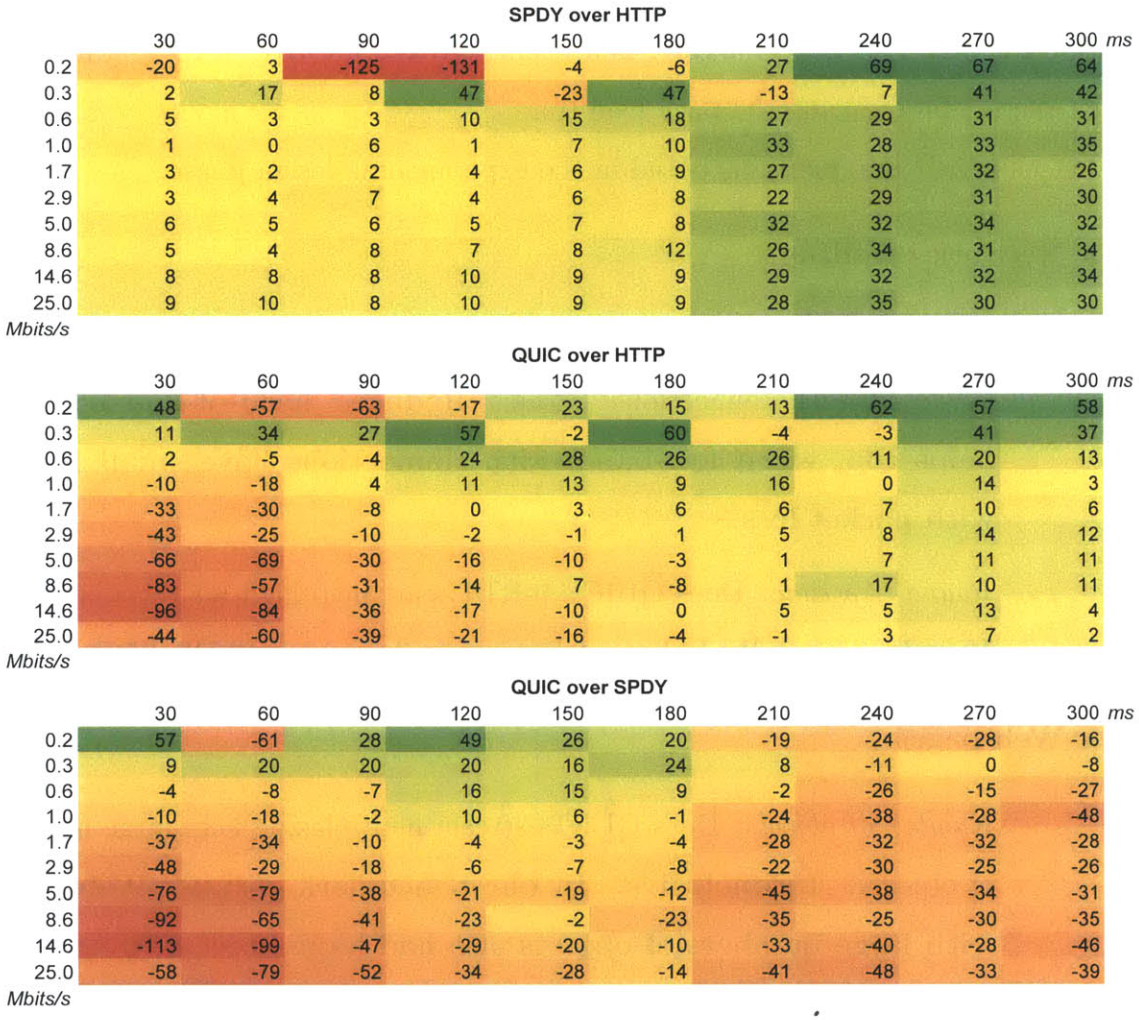


Figure 5-2: Heat maps showing % protocol improvement of 500-website median page load times across 100 bandwidth/RTT configurations: {0.2, 0.3, 0.6, 1, 1.7, 3, 5, 8, 14, 25} Mbits/sec × {30, 60, 90, 120, 150, 180, 210, 240, 270, 300} ms.

the websites, while only 43 (8.6%) of the 500 websites are HTTPS and thus encrypted over HTTP/1.1. Six of them are in QUIC's best, and none of them are in QUIC's worst. Moreover, we find that *QUIC improves all 43 HTTPS websites.* The first reason is that both HTTPS and QUIC encrypt all packets. The second is that while HTTP has 1-RTT start (just TCP handshake), HTTPS has the full 3-RTT start (both TCP and TLS handshakes). As a result, QUIC's low-cost, 0-RTT start has a larger impact on reducing the page load time.

Let us revisit the questions posed in the experimental design phase:

- **Network conditions**
  - *Bandwidth*: Does QUIC's header compression and low-overhead start help low-bandwidth connections? **Yes. Reduced head-of-line blocking helps too, when low-bandwidth connections have small buffers with packet loss.**
  - *Round-trip time*: Does QUIC's 0-RTT start help high-RTT connections? **Yes. It especially helps when comparing against HTTPS.**
- **Web pages**
  - *Number of objects*: Does QUIC's stream multiplexing help large numbers of objects? **Inconclusive. In our benchmark corpus, Web pages with large numbers of objects also had large objects.**
  - *Object sizes*: Does QUIC's stream multiplexing help small objects? **Yes. QUIC helped Web pages with small objects.**

Website	% Improvement	# Server Addresses	# Objects	Mean Object Size (kB)	Total Size (kB)	HTTPS
Websites QUIC hurts the most						
Answers	−8028	31	156	18	2830	
BuzzFeed	−1979	31	680	17	11304	
CVS	−666	31	203	12	2524	
People	−345	51	392	27	10417	
Credit 6	−127	1	2	1	2	
Twitch	−113	28	181	91	16599	
Pogo	−88	55	262	14	3530	
TMZ	−81	48	301	176	53276	
Fast Daily Find	−74	1	3	6	22	
Zcobit	−72	1	6	23	157	
Websites QUIC helps the most						
HootSuite	+43	11	33	22	731	✓
USPS	+45	3	83	15	1261	✓
ADP	+45	11	79	12	943	✓
SiteScout	+45	39	108	6	697	
LinkedIn	+46	7	35	7	260	✓
Scars	+54	44	202	8	1463	
SunTrust	+58	10	106	9	917	✓
Wikimedia	+58	5	20	10	214	
U.S. Bank	+60	3	59	8	455	✓
Washington Post	+88	52	223	8	1695	
Mean $\Delta \rightarrow$		1.5× less	2.3× less	3.6× less	12× less	60% more

Table 5.1: The websites QUIC helps and hurts most: bottom 10 and top 10 websites ased on mean improvement in page load time from HTTP to QUIC across all 100 configurations. They are accompanied by Web page attributes, such as objects per page and mean object size, and the mean change from QUIC's worst to QUIC's best.

Chapter 6

Sprout-EWMA for Cellular Networks

Mobile traffic is rapidly accounting for larger fraction of overall Web traffic; its volume increased 70% from 2012 to 2013 [1]. To evaluate the state of the mobile Web, we use the cellular network traces provided in our corpus (Section 4.3). We consider 4G LTE networks from both Verizon and AT&T. Our objective is to implement a congestion control algorithm in the QUIC `congestion_control` module and compare it to the default algorithm, TCP Cubic.

6.1 Technical Challenges

To evaluate performance on cellular networks, we use the cellular network traces provided in our corpus (Section 4.3). We consider HTTP/1.1, SPDY, and QUIC over a Verizon 4G trace and a minimum RTT of 120 ms. Figure 6-1 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. The 4G bandwidth is high (9.1 Mbits/sec uplink and 19.1 Mbits/sec downlink) so, applying the results from Section 5.1, both HTTP/1.1 and SPDY outperform QUIC.

Our goal is to improve mobile Web performance over QUIC. QUIC's default congestion control algorithm is TCP Cubic, but there are two technical challenges for QUIC's TCP Cubic to adapt to a cellular network. First, cellular networks almost

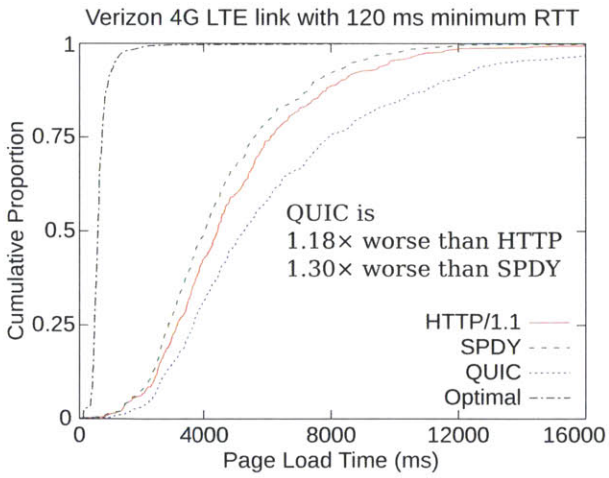


Figure 6-1: HTTP/1.1, SPDY, and QUIC page load times over a Verizon 4G cellular network, 120 ms RTT.

never drop packets, so packet loss is a poor congestion signal. Therefore, during a network outage, Cubic does not adapt until its retransmission timer fires — a slow reaction. Second, these networks have high variability of bandwidth over time, as illustrated in Figure 6-2. Once Cubic reduces its window due to retransmission, it may not adapt to the changing bandwidth quickly enough. We take advantage of QUIC's pluggable congestion control by introducing a congestion control algorithm which is specifically purposed for cellular networks: Sprout-EWMA, as described in [19].

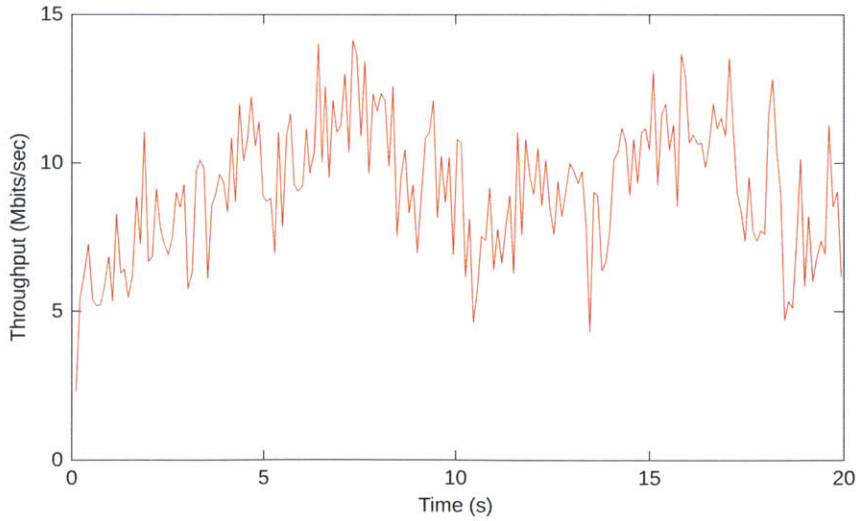


Figure 6-2: The high variability of bandwidth over time, measured on the uplink of a Verizon 4G cellular network.

6.2 Algorithm and Implementation

We implement Sprout-EWMA based on the description in [19]. Sprout-EWMA takes packet arrival times at the receiver to estimate the network throughput and applies an exponentially-weighted moving average to smooth the estimate. It multiplies the rate by a target delay to calculate the congestion window and, from there, acts like a window-based sender similar to TCP Cubic. We implemented this algorithm in QUIC. It required us to place the receiver's data about packet arrival times in a Congestion Control Frame specifically for Sprout-EWMA. We also froze the algorithm's parameters to avoid tuning to the Verizon and AT&T packet-delivery traces, leaving the EWMA gain $\alpha = \frac{1}{8}$ and the target delay $\delta = \min\{100 \text{ ms}, \text{minimum RTT}\}$.

Sprout-EWMA does not optimize for page load time; it is intended to achieve high throughput at a low per-packet delay (e.g., for long-running flows that need interactivity; not the many short, concurrent flows to transfer small Web objects). However, we believe that it may still outperform Cubic for two reasons. As mentioned, cellular networks are highly variable, and Cubic may take too long to adapt to the current network capacity. These networks are also very reliable, so there is very little packet loss, but that is Cubic's congestion signal. Sprout-EWMA's approach of proactively measuring the network throughput (i.e., receiving rate) and adjusting its window accordingly addresses these issues.

6.3 Performance Evaluation

Figures 6-3 and 6-4 show the benefits Sprout-EWMA provides over Cubic on these cellular networks when the minimum RTT is high. For 300 ms minimum RTT, Sprout-EWMA's median page load time is 10% lower than Cubic's over the Verizon 4G LTE link and 13% lower over AT&T. However, we find that Cubic outperforms Sprout-EWMA when we impose a much smaller RTT of 30 ms; Cubic's 95th percentile page load time is 13% lower than Sprout-EWMA's over Verizon and 16% lower over AT&T. The two providers' links differ for the intermediate 120 ms; Sprout-EWMA's median

page load time is 13% higher than Cubic's over AT&T but 10% lower over Verizon. These results suggest than Sprout-EWMA improves page load times over high-RTT links because the receiver's feedback (in a single Congestion Control Frame) allows the sender to more quickly adapt to changing bandwidths, compared to TCP's reliance on packet loss and ACKs. But Cubic is more aggressive in throughput and that helps it for the low-RTT links.

The exact crossover RTT depends on the link in question. While Verizon's bandwidth is high (9.1 Mbits/sec uplink and 19.1 Mbits/sec downlink), AT&T's bandwidth is 3 – 8× lower (1.1 Mbits/sec uplink and 6.7 Mbits/sec downlink). Based on our analysis, we suggest the benefits of Sprout-EWMA (e.g., ramping up to a sudden bandwidth increase) are lower when the network's bandwidth is lower. Ultimately, we note that developing a congestion control algorithm for page loads over cellular networks, across all RTTs, is an area for further work.

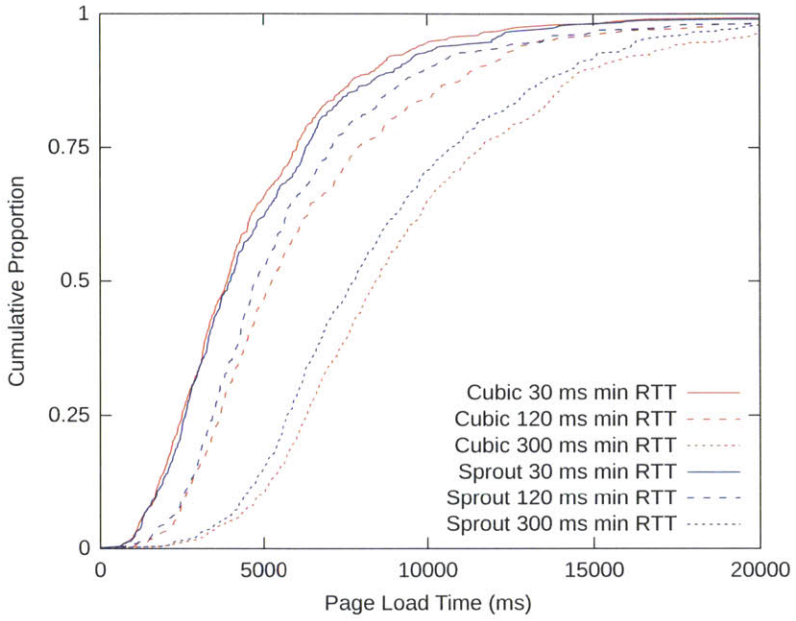


Figure 6-3: QUIC/Cubic vs. QUIC/Sprout-EWMA page load times over a Verizon 4G cellular network, $\{30, 120, 300\}$ ms RTT.

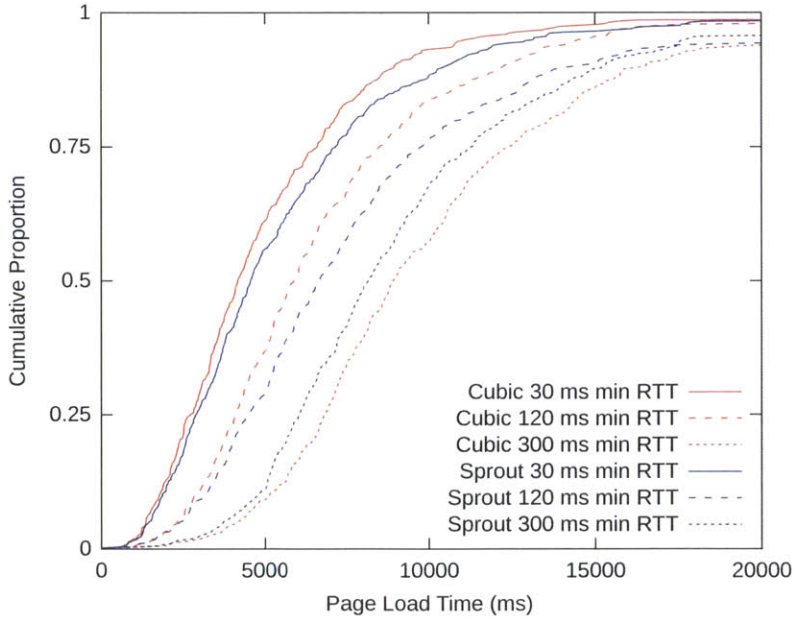


Figure 6-4: QUIC/Cubic vs. QUIC/Sprout-EWMA page load times over a AT&T 4G cellular network, $\{30, 120, 300\}$ ms RTT.

Chapter 7

Conclusion

7.1 Summary of Contributions

This work present the first study of QUIC on Web page performance. Our experiments span 500 Web pages across at least 100 network configurations of bandwidth and minimum RTT and test the HTTP/1.1, SPDY, and QUIC multiplexing protocols. We summarize our other contributions below.

- Providing QuicShell, a tool for measuring QUIC's Web page performance in an accurate and reproducible way. It can be easily used by future researchers to check for performance improvements or regressions across QUIC versions.
- Studying QUIC across a large benchmark corpus and many network conditions and finding that:
  - QUIC outperforms (has a lower page load time than) HTTP/1.1 and SPDY over low-bandwidth as well as high-RTT links.
  - QUIC outperforms HTTP/1.1 for Web pages with small objects as well as HTTPS-enabled Web pages.
  - QUIC/Sprout-EWMA outperforms QUIC/Cubic over high-RTT cellular networks.

- Making all implementations and results open-source (see Table 7.1)

Implementation	URL	Code Δ (lines)
Benchmark corpus, results, Python scripts measuring page load time	`https://github.com/ravinet/sites/tree/scripts`	22
QuicShell *(within Mahimahi)*	`https://github.com/ravinet/mahimahi/tree/quicshell`	88
Modified QUIC server for QuicShell *(within QUIC)*	`https://github.com/anirudhSK/chromium/tree/cgi_script`	207
Sprout-EWMA *(within QUIC)*	`https://github.com/anirudhSK/chromium/tree/mycubic`	646

Table 7.1: Open-source repositories for the code written for this work.

7.2 Directions for Future Work

In the future, we are looking to investigate the features that are missing from the current version of QUIC.

- Evaluate Forward Error Correction (FEC) strategies for lossy networks
- Evaluate packet pacing (e.g., in WebRTC Inter-Arrival) on Web page performance
- Evaluate the dependencies between computation and networking (similar to like Eload and WProf [17] which were used to study SPDY in an 2012 version of Chrome, except for QUIC in a modern version of Chrome instead)

Bibliography

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

[2] Alexa. Top sites in the United States. http://www.alexa.com/topsites/countries/US, 2014.

[3] Chromium. SPDY: An experimental protocol for a faster Web. http://www.chromium.org/spdy/spdy-whitepaper, 2009.

[4] 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 CCR*, 40(3):27 33, 2010.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Persistent connections. http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html, 1999.

[6] 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 *SIGCOMM*, 2013.

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

[8] Albert Abello Lozano. Performance analysis of topologies for Web-based real-time communication. https://aaltodoc.aalto.fi/bitstream/handle/123456789/11093/master_AbellÃş_Lozano_Albert_2013.pdf, 2013.

[9] Ravi Netravali, Anirudh Sivaraman, and Keith Winstein. Mahimahi: A lightweight toolkit for reproducible Web measurement. http://mahimahi.mit.edu/, 2014.

[10] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. TCP Fast Open. In *CoNEXT*, 2011.

[11] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Christopher Riederer. Give in to procrastination and stop prefetching. In *HotNets*, 2013.

[12] Jim Roskind. Is FEC enabled by default? https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/ol0lwEjCucI, 2013.

[13] Jim Roskind. QUIC: Multiplexed stream transport over UDP. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit, 2013.

[14] 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 *HotMobile*, 2014.

[15] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. http://www.ietf.org/rfc/rfc2960.txt, 2000.

[16] Ian Swett. A performance problem of QUIC: Low throughput. https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/-Tb0oODDSZU, 2014.

[17] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[18] Zhiheng Wang and Arvind Jain. Navigation timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html, 2013.

[19] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.