Kyle O'Connor

108070950

Performance Comparison of HTTP/1.1, HTTP/2, and QUIC

Github Repository: https://github.com/koconnor26/CSE-534-Project

1. Introduction and Background

When hosting content on the web you have the choice of serving your content using HTTP/1.1, HTTP/2, or QUIC. My project explores the differences in these application layer protocols and determines how they perform under different network conditions and types of content served. Before getting into the details of my project it's important to understand the different features and downsides of these protocols.

HTTP/1.1 builds on top of HTTP/1.0 with the addition of persistent connections and parallelization. Persistent connections allows a client to perform consecutive requests over a single connection rather than having to create a new TCP connection for every request. Parallelization adds the ability to create multiple TCP connections to a webserver and simultaneously request resources on each of these connections. Typically the client limits how many simultaneous connections it will open which in the case of this project is a maximum of six connections in Google Chrome. The features introduced in HTTP/1.1 are beneficial, but the protocol still has some drawbacks including increased network congestion due to multiple independent TCP connections being created, the inability to carry more than 1 HTTP request or response in a TCP segment, and the inability to compress HTTP headers.

HTTP/2 addresses the problems of HTTP/1.1 and introduces some new features as well. HTTP/2 relies on multiplexing which allows multiple requests and response to be sent over a single TCP connection. This means HTTP/2 only has to open a single connection with the

webserver which addresses the congestion issue of HTTP/1.1. Additionally, HTTP/2 adds priorities to requests in order to give them more bandwidth within the connection. A downside to the implementation of multiplexing over a single TCP connection is the issue of head of line blocking. TCP relies on in order delivery, so if a TCP segment is lost then the whole TCP connection will be held up waiting for the missing segment. This means all requests on the HTTP/2 connection will be blocked until the lost segment is retransmitted which can add request latency. HTTP/2 also adds the ability for the server to push data to the client and introduces the compression of HTTP headers making messages more compact. The idea with server push is the server can anticipate what the client is going to request and start sending it ahead of time. This isn't possible in HTTP/1.1 as only the client can initiate the transfer of data. The final change HTTP/2 introduces is the connection must be over SSL which is an added security benefit.

QUIC is a new application layer protocol proposed by Google to be used for transferring web pages. QUIC is a novel approach and is starting from a fresh slate unlike HTTP. I will only be giving a brief overview of QUIC and its relevant parts to this project. The biggest difference between QUIC and HTTP is QUIC runs on top of UDP whereas HTTP runs on top of TCP. QUIC moves the TCP reliability, congestion control, and in order delivery to the application layer thus giving it more control and allowing for changes to be applied easier. For example, one of the features of QUIC is its congestion control algorithm is pluggable thus you can choose whichever congestion control algorithm you like without having to change the underlying network stack. QUIC offers the same multiplexing capabilities as HTTP/2 without the issue of head of line blocking. QUIC creates multiple independent streams over its single UDP connection and a loss on one of the streams doesn't cause the other streams to become blocked. QUIC also reduces the connection setup time by introducing a 1 RTT handshake unlike TCP

which relies on a 3 RTT handshake. It also has the ability to cache connection details allowing for a 0 RTT connection in the future. QUIC originally had forward error correction to reconstruct lost packets, but it seems like it has been removed from the latest version so I chose not to use it in this project.

2. Problem Statement and Hypothesis

The problem my project is addressing is determining which application layer protocol, HTTP/1.1, HTTP/2, or QUIC, provides the best web page load time for all network conditions and types of content being served. Additionally, if none of these protocols are ideal for all scenarios then I would like to explore how their performance differs. I will also be looking at some other metrics described in the next section which will provide an understanding of the different behavior of these protocols. I believe this additional analysis will earn me the additional 30% on this project. I hypothesize HTTP/2 and QUIC will outperform HTTP/1.1 when there is minimal loss in the network with QUIC performing slightly better than HTTP/2. I think HTTP/2 and QUIC only having to create a single connection to a webserver will give them better performance than HTTP/1.1. This is because these protocols are able to efficiently multiplex requests over a single connection and don't have the overhead of creating multiple TCP connections like HTTP/1.1. The reason I think QUIC will have better performance than HTTP/2 is because it can setup connections faster and doesn't have to deal with head of line blocking.

3. Metrics and Test Scenarios

Along with web page load time I am interested in collecting the following metrics:

- Time to first byte – This determines which protocol has the fastest connection setup and can start receiving data as soon as possible.
- Time for first image to download – This determines when content can start being displayed on the web page.

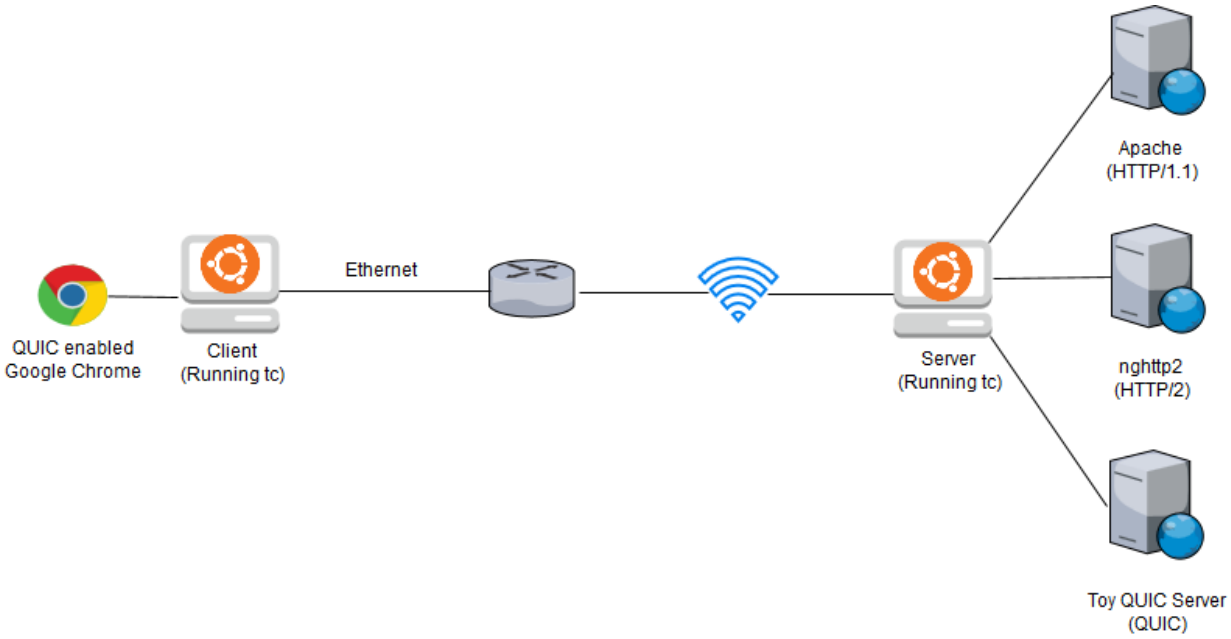

Figure 1

Figure 1 contains the test system I used for my evaluation. This test system is based off of the testbed used in HTTP over UDP: an Experimental Investigation of QUIC, a paper that inspired my project. The server is running Ubuntu Desktop 16.04.2 and contains three web servers, Apache, nghttp2, and the toy QUIC server provided in the chromium repository. The server uses the program tc to limit the bandwidth and set the loss rate of its interface. The server is connected to a router via WiFi. In hindsight this should be an Ethernet connection, but unfortunately I didn't realize this until it was too late. I believe the WiFi connection may have caused some anomalies in my end results. The client is connected to the router via Ethernet and

is running Google Chrome with QUIC enabled. The client also uses tc to limit the bandwidth and set the loss rate of its interface.

For my test scenarios I used the ones described in HTTP over UDP: an Experimental Investigation of QUIC, which consisted of a small, medium, and large web page being request over varying network conditions. The small web page contains 18 small (14KB) images and 3 medium (80KB) images. The medium web pages contains 40 small (14KB) images and 7 medium (80KB) images. The large web page contains 200 small (14KB) images and 17 (80KB) images. The network bandwidth was set to 3 and 10 mbps and the network loss was set to 0% and 1%. I also included my own web pages, one containing 200 small (14KB) images and one containing 25 large (1900KB) images. These web pages were requested over normal network conditions. The reason the web pages only contain images is to reduce the variable load time of the browser when having to parse other types of content such as Javascript and CSS. The web page just has to download the images and display them with no processing and parsing in between. For each of these scenarios I made 25 requests to each server and exported the resulting requests and responses as a HAR file. I then wrote a python script to parse these HAR files and compute the average of the metrics I described earlier over the 25 requests. The next section contains the graphs my script created for these metrics.

4. Results

4.1 Web Page Load Time

The following graphs contain the percentage of web page load time improvement over HTTP/1.1 for HTTP/2 and QUIC.

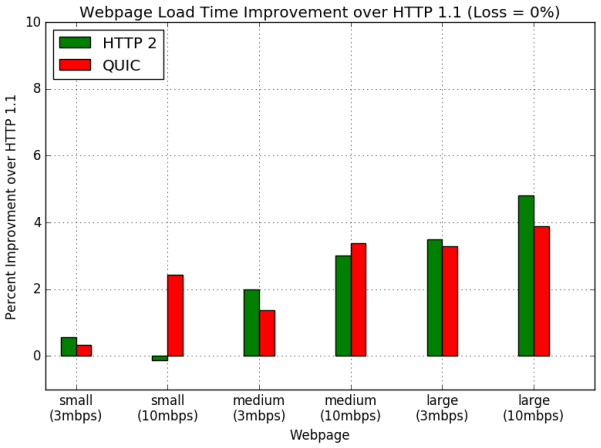

Figure 2

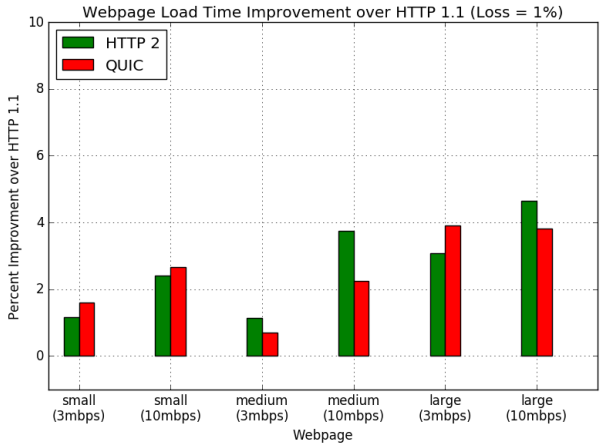

Figure 3

Figure 2 and 3 show the web page load time improvement of QUIC and HTTP/2 for the small, medium, and large web page under the different network conditions. As you can see, HTTP/2 and QUIC generally provide the same performance increases over HTTP/1.1 with HTTP/2 having a slight edge. The load time improvements for both protocols aren't very significant with the maximum being around a 4% increase for the large web page over a 10mbps connection. Much more significant improvements can be seen when requesting the web pages with many small and many large objects.

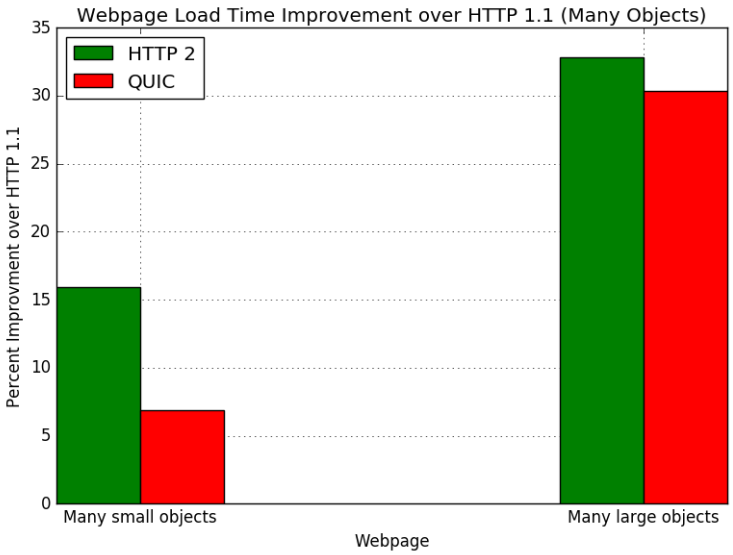

Figure 4

Figure 4 contains the resulting web page load time improvements under normal network conditions when requesting many small and many large objects. HTTP/2 shows the largest improvements with over 15% for many small objects and about 33% for many large objects. QUIC doesn't perform as well as HTTP/2 in both scenarios, but still shows substantial load time improvements over HTTP/1.1 when requesting many large objects with about a 30% improvement.

4.2 Time to First Byte

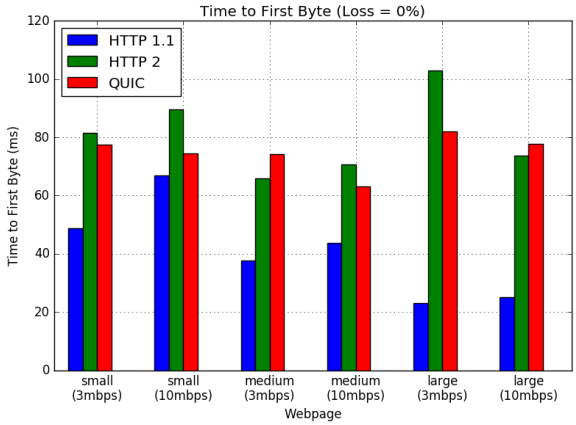

Figure 5

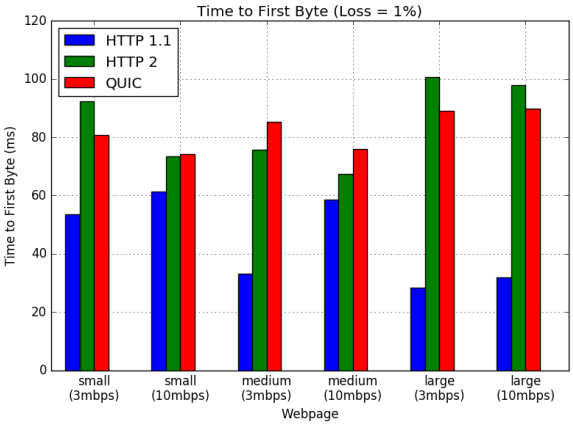

Figure 6

Figure 5 and 6 show the time to first byte for the small, medium, and large web pages with the varying network conditions. Surprisingly HTTP/1.1 shows the quickest time to first byte. The HAR format breaks down the time each request spent in different phases of the connection. I found HTTP/2 and QUIC were spending more time setting up their SSL connections than the HTTP/1.1 connection. I'm not sure what would cause this increase in connection setup, but I think it may have to do with the implementation of each web server.

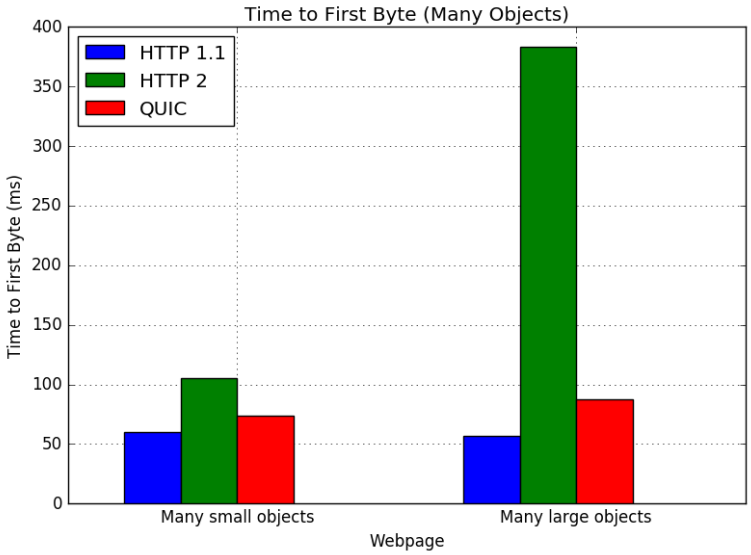

Figure 7

Similar results were collected for the many small and many large objects web pages. HTTP/2 shows an extremely high latency to receive the first byte when requesting the many large objects web page. I think this is an anomaly perhaps with the WiFi connection I was using although I performed all of these requests consecutively so if there was some sort of spike in latency then it was only when I was requesting the web page over HTTP/2. I don't see any reason HTTP/2 should take longer to receive its first byte when requesting that particular web page. I examined the HAR files and found the latency was consistently high, so the average wasn't being skewed by an outlier.

4.3 Time For First Image To Download

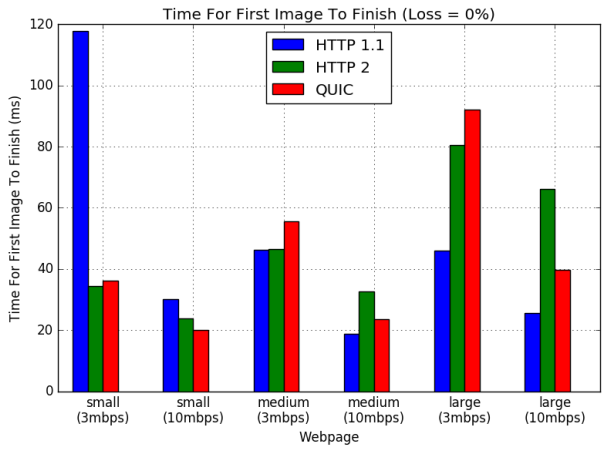

Figure 8

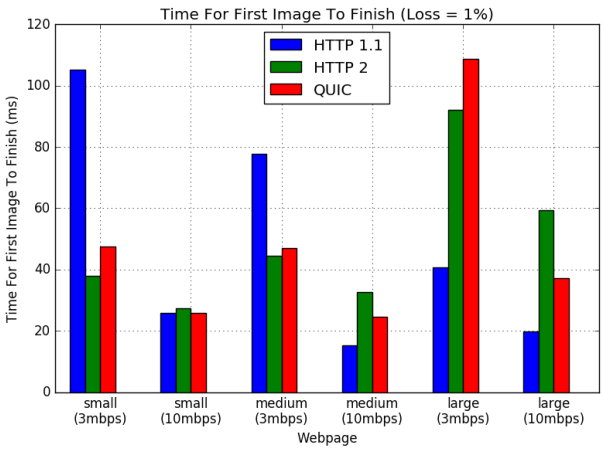

Figure 9

Figure 8 and 9 show the time for the first image to download for the small, medium, and large web pages under the varying network conditions. For the most part HTTP/1.1 loads the first image the fastest or does just as well as QUIC and HTTP/2. The main exception being when HTTP/1.1 requests the small web page with a 3mbps connection. I think HTTP/1.1 is able to finish downloading an image faster because it has a dedicated connection per each image it is downloading. QUIC and HTTP/2 are multiplexing many different image downloads over a single connection, so they are slowly downloading many images thus making it slower for the first image to finish downloading.

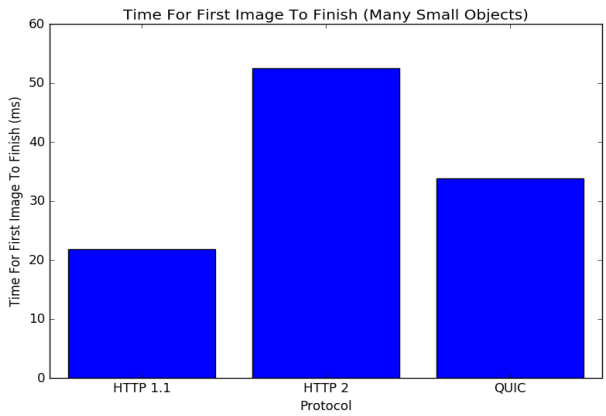

Figure 10

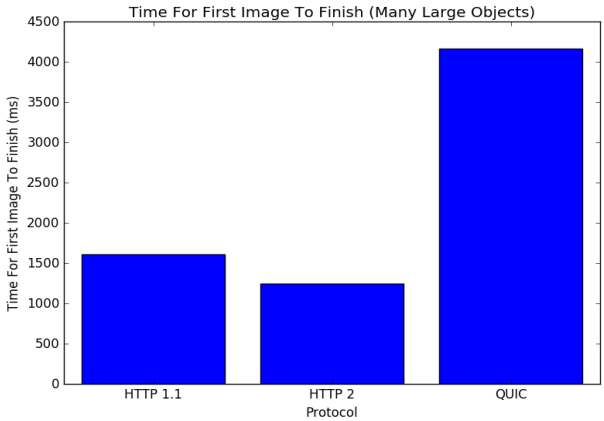

Figure 11

Figure 10 and 11 show the time for the first image to finish downloading when requesting the many small and many large objects web pages. The many small objects web pages shows similar results to the previous graphs. The many large objects web pages shows some interesting results because QUIC takes significantly longer to finish its first image compared to HTTP/1.1 and HTTP/2. QUIC multiplexes up to 100 requests over a single connection which in this case causes it to simultaneously download 25 large images. This means QUIC is slowly building all 25 images which causes a single image to take a while to finish. I couldn't find any specification stating how many objects HTTP/2 multiplexes over its connection, but I assume the amount of objects is either capped or HTTP/2's use of priorities allows for its first image to finish faster.

5. Conclusion and Future Work

Overall we can conclude that HTTP/2 is the best choice for the fastest web page load time. For the most part HTTP/2 was able to outperform QUIC and HTTP/1.1 in the web page load time test scenarios. It was surprisingly to see HTTP/1.1 had the fastest time to first byte. I assumed QUIC would be the fastest since it has the advantage of a 1 RTT connection setup, but it appears it and HTTP/2 spent a lot of time doing SSL connection setup. I'm not exactly sure what would cause this latency, but I believe it has to do with the implementation of the web servers. Finally, HTTP/1.1 was able to load the first image of a web page the fastest which is an important metric because if you are serving web content you want something to display on your web page as soon as possible. QUIC appears to be a poor choice when a web page contains many large resource because QUIC multiplexes the requests for all of these resources which leads to a large latency for one of those resources to finish.

For future work I think it would be beneficial to have a test system without a WiFi connection as I feel it caused some of the anomalies in my results. I would also like to try these

experiments with a higher packet loss percentage. The paper that inspired my project showed a 1% network loss to have a huge impact on the performance of these protocols, but I barely saw any changes in their performance. Additionally, I would like to use one web server to host all three of these protocols which I don't believe is currently possible because QUIC is only available via the toy QUIC server found in the Chromium repository.