A Comparison of Network Protocols: TCP vs QUIC

Carl Masri, Kjalen Hansen, Jakob Roberts

Abstract

The rapid advancement of web technologies has given rise to the search for a faster, more improved transport protocol. In 2013 Google released QUIC (Quick UDP Internet Connection), an experimental network protocol which aims to solve some of the current issues surrounding TCP. The goals of this paper are twofold: first, we aim to give a comprehensive overview of the QUIC protocol including the motivations behind its creation and limitations of the existing protocols; second, we study the performance of QUIC by comparing it against HTTP/2 (which runs over TCP) by use of a purpose built server-client architecture in order to gain a deeper understanding of its operation and to collect qualitative metrics upon which QUIC's performance can be evaluated. Our findings concluded that, although the QUIC protocol was able to outperform TCP in a number of cases, network conditions were the largest factor in determining which protocol performs best.

Preprint submitted to CSC 466 *April 11, 2017*

1. Introduction

The dominant form of point-to-point communication via the Internet has been Transmission Control Protocol (TCP) for many years due to its reliability and robustness. Together with User Datagram Protocol (UDP), TCP is one of the most widely used transport protocols for handling data communications in today's current Internet. However, much has changed since TCP was first introduced in 1974–the Internet now operates with over 3 billion users, which has lead to a steep rise in the amount of network traffic over the past decade [1]. More recently, the rapid adoption of smart phones and smart home devices has only seen this number increase even further. According to Gartner Research, there was an estimated 59 percent increase in mobile video traffic in 2015, and this trend is projected to continue [2].

The pressure on modern networking infrastructure's need to expand and diversify has highlighted issues with the current protocols and brought an increased need for improved technologies which are able to meet future demands. In 2013 Google released QUIC (pronounced "quick"), a new protocol which serves as a proof-of-concept for some new ideas intended to mitigate some of the limitations of TCP. The main goal of this paper is to contribute to a better understanding of the performance of QUIC as compared to the more widely used TCP and UDP protocols.

The following organizational scheme will be used: Section II describes the background of QUIC, discusses some limitations of the current protocols, and also presents the related work. Section III outlines the methodology used to evaluate QUIC, including a description of the measurement environment used to test QUIC's performance. Section IV presents the results of the measurements. Section V summarizes the work and discusses potential future areas of research.

2. Background and Related Work

QUIC is a experimental transport layer protocol built by Google that aims to provide the guaranteed transport layer services of TCP with lower overhead and latency. QUIC is built entirely upon UDP which allows flexibility in its implementation, as UDP

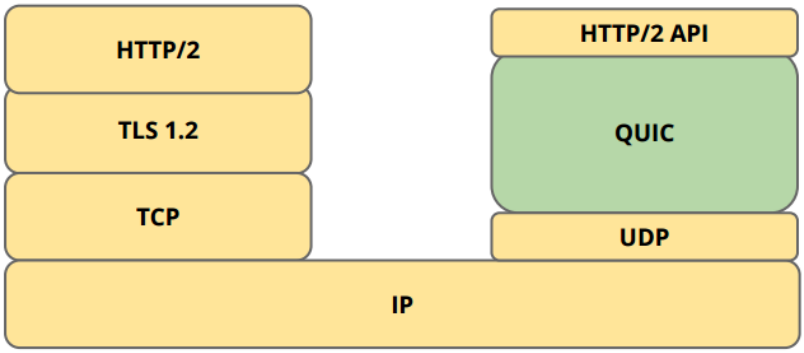


Figure 1: QUIC runs over UDP rather than TCP [4]

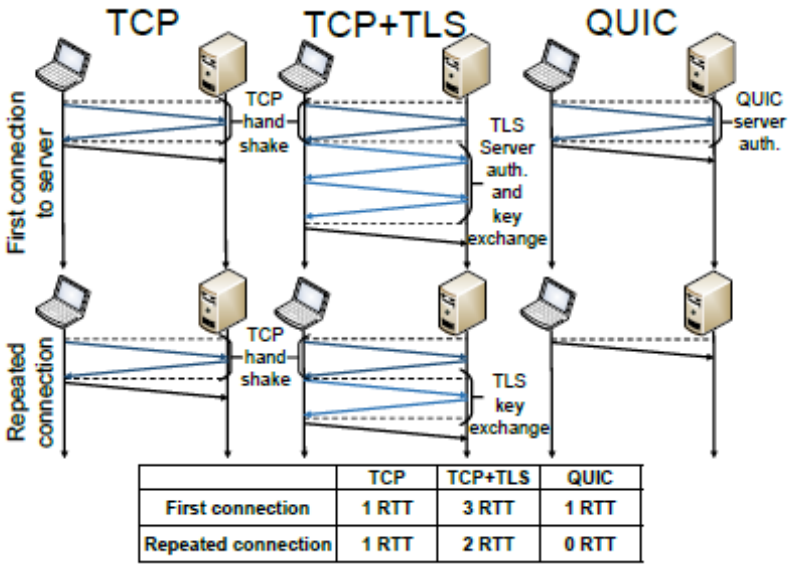
is lightweight and has very low overhead. Google maintains that there are many performance benefits of using UDP instead of TCP including reduced connection time, improved congestion control, and forward error correction. QUIC is currently used in production by Google for many of their back-end services and features in their Chrome browser. Figure 1 shows how QUIC fits into the traditional TCP/IP stack.

The increasing demand for near-real-time responsiveness cannot be met by the current infrastructure mainly due to present limitations in the HTTP and TCP protocols. As the primary motivation behind QUIC, these limitations are enumerated in the following subsections.

2.1. Limitations of TCP

The limitations of TCP stem from its rich assortment of baked-in features. It is capable of bi-directional communication between hosts, can handle packet loss, network congestion, packet re-ordering, and as such is an optimal base for layering encryption methods. As such, TCP proves excellent for commonly used application layer protocols such as HTTP and SMTP. However, as Dr. Pan often says, "there's no free lunch"; these features come at a high and non-negotiable cost. The TCP handshake requires additional round trip time (RTT), and the designers of TCP did not include security considerations meaning the additional of any security layers will require extra round-trips on top of the initial connection handshake.

Figure 2 displays the additional delay cost of setting up a TLS encrypted connection.



	TCP	TCP+TLS	QUIC
First connection	1 RTT	3 RTT	1 RTT
Repeated connection	1 RTT	2 RTT	0 RTT

Figure 2: Connection RTTs in various network protocols [3]

Given that RTT is a limiting factor which is governed by physics, the only way to reduce round-trip related network latency is to reduce the number of RTTs required. QUIC employs special methods to reduce the number of RTTs needed to establish a connection, thereby reducing the latency. By design, TCP uses automatic re-transmission of lost packets to ensure reliable delivery of packets, however, this is not useful for real time applications such as video streaming or VoIP where dropped packets are better left forgotten.

Another major shortfall of TCP addressed by QUIC is head of line (HOL) blocking. If a packet is lost, TCP's requirement for in-order packet delivery means that the current stream must halt and wait until the correct packet is re-transmitted. It has been shown in several studies that this HOL blocking behaviour has negative impacts in high-loss environments [5] [6]. With the advent of HTTP/2 and its multiplexed connections (largely based on Google's SPDY protocol), the HOL blocking issue becomes an even greater impediment to performance.

One of TCP's greatest strengths is its congestion control mechanisms. However, these processes are highly standardized and therefore notoriously slow to evolve. Deployment also becomes an issue, as transport layer protocols are deeply integrated into the current

Internet infrastructure making them difficult to change.

2.2. Limitations of UDP

At the outset of the internet, the requirements were simply to have a reliable and guaranteed service. As the types of applications on the net expanded, so did the requirements. Low latency and high throughput were becoming more and more important and a new protocol was needed. This is when the TCP stack was separated from the IP stack and the new connectionless protocol UDP was made available. UDP served as a basic building block with low overhead allowing users to tailor the protocol to their specific needs should TCP not be sufficient. As it is only a basic protocol, there are several limitations, the chief among them being no guarantee of delivery and no security (a limitation shared with TCP). The speed and low latency of UDP comes from its very low overhead from having no built-in features. There are no acknowledgement packets to worry about, the sender does not need to be troubled with client buffer sizes, and no attempts are made to correct network congestion. This makes UDP the protocol of choice for delay-sensitive applications such as voice or video streaming, but insufficient for the majority of web traffic.

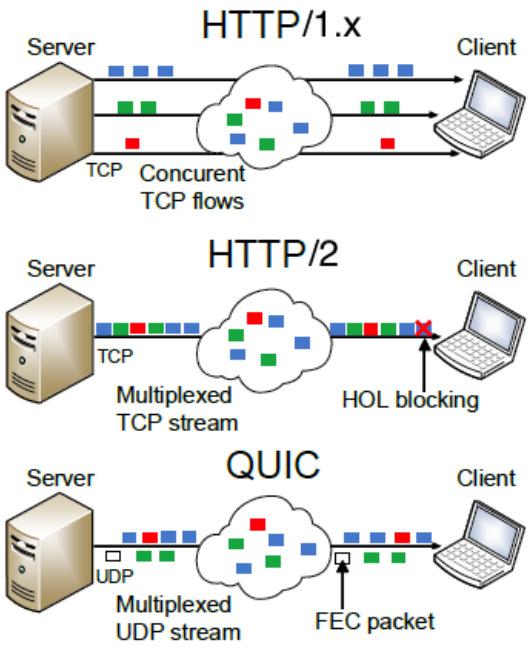


Figure 3: Both QUIC and SPDY use multiplexed streams. Adapted from [3]

2.3. Limitations of HTTP

The foundation of data communication for the World Wide Web, HTTP (which stands for Hypertext Transfer Protocol) is the most widely adopted application protocol in use today. It is a request-response based protocol which specializes in the exchange of

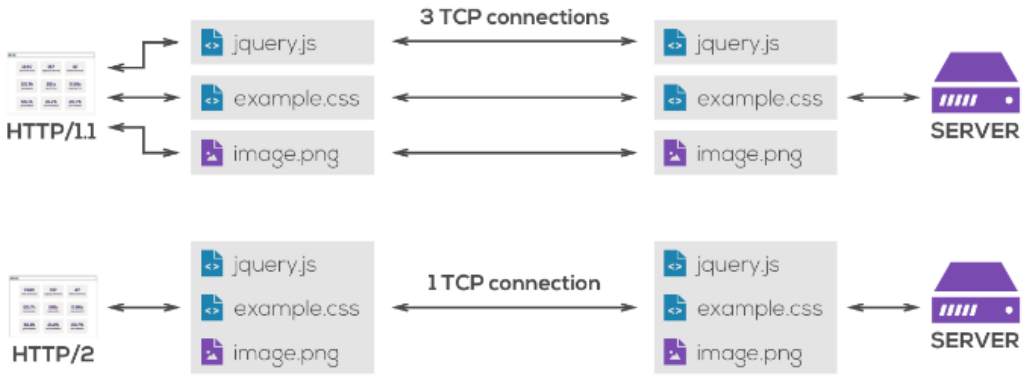


Figure 4: HTTP/1.1 and HTTP/2 connections [7]

hypertext, a process which facilitates the delivery of billions of web pages every day. The first version of HTTP, 0.9, was released in 1991. This was followed up with the officially recognized version 1.0 in RFC 7230 (1996) and 1.1 in RFC 2068 (1997) [8]. However, at this time web pages were much less complex than they are nowadays and the protocol was designed without the considerations that are now faced by modern Internet technologies.

Perhaps the largest performance limitation of the HTTP/1.x protocol was the need for each request to be sent over a single TCP connection. To increase performance, clients opened dozens of distinct connections (as shown in Figure 3). However, TCP connections were not designed to handle these types of short-lived, bursty transfers. If the number of connections grows too large, TCP's congestion control mechanisms will kick into play resulting in high packet loss.

Other difficulties with HTTP/1.x include the inability for TCP segments to carry more than a single HTTP request/response, meaning that clients send a significant amount of redundant data in the headers. Clients are also the only ones allowed to initiate a transfer, meaning servers must wait for the client to process the entire web page before it is able to send any embedded objects [3]. As it runs over TCP, HTTP/1.x also suffers from HOL blocking. An attempt was made to fix this with pipelining (Figure 5) but was unsuccessful due to difficulties with deployment [10]. By nature, HTTP/1.x is a text-based protocol that does not implement any security and is therefore vulnerable to attacks such as Man in the Middle. SSL/TLS encryption can be added to form the HTTPS protocol, but as mentioned above this adds additional round-trip time.

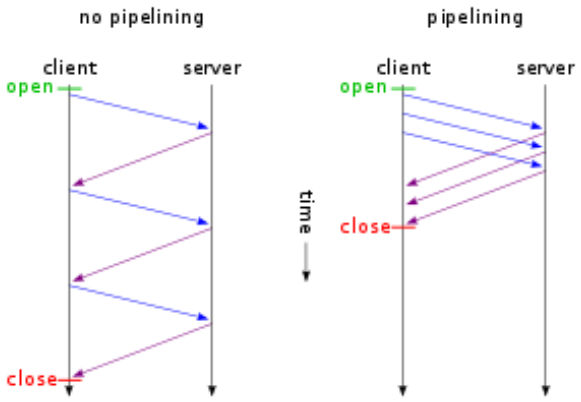


Figure 5: Non-pipelined vs pipelined connection [9]

These limitations of HTTP/1.1 led to the development of a new and improved version of the protocol. HTTP/2 was released in May 2015 and incorporates much of the design philosophy of the SPDY protocol (the first draft of HTTP/2 was based on SPDY/2 [10]). Its primary goals were to decrease latency by considering the aforementioned problems with the HTTP/1.1 protocol.

HTTP/2 brought numerous improvements, most notably the multiplexing of requests. This had two benefits: 1) mitigate HOL blocking by allowing multiple request and response messages to be in flight at the same time, and 2) allow web pages to be loaded over a single TCP connection, limiting connection-related congestion. It also introduced binary-encoded header compression to cut down on overhead and added the ability for servers to send responses to the client before an explicit request is made, thereby reducing round trip delay. This mechanism is known as Server Push [10].

2.4. QUIC

The primary design goals of QUIC were to bring a reliable, multiplexed, and faster transport system which runs over UDP. The decision to build the protocol on UDP rather than TCP has two main advantages: current TCP mechanisms are highly ingrained into current web technologies and have proven exceedingly difficult to change, and UDP allows for accelerated development of new transport protocols which can be deployed rapidly while leveraging best practices learned from the past two decades of TCP experimentation.

As general internet access has gotten faster and faster, latency has become the most important metric of performance. As QUICs design sits on top of UDP and not on top TCP, the latency issue is attempting to be addressed by Google. The long-term goal for QUICs designer Google was to make a QUIC connection to be nearly equivalent but to improve latency and have better stream-multiplexing support like in SPDY. QUIC is combining the best parts of multiplexing from SPDY and HTTP/2, and having this on top of a protocol that does not have any blocking, the speed increase is noticed.

QUIC improves on TCPs default Head of Line blocking deficiencies by using a Forward Error Correction (FEC) style similar to how RAID 5 is implemented in disk storage. The QUIC protocol can recover from transmission errors without retransmitting lost packets. An arbitrary number of packets can be grouped together and transmitted along with one extra packet for error recovery. If a single packet of that group is lost, it can be recovered at the destination by using the extra error packet that is an XOR of all information packets sent in the grouping. The smaller the group, the more robust against transmission errors. However, the advantages in loss recovery brought on by smaller FEC groups incur consequences of increased bandwidth overhead.

Accepted encryption standards at this time generally use a form of TLS on top of a TCP connection; QUIC has implemented its own form of encryption (QUIC Crypto) that is accepted at about the same level as most forms of TLS and thus everything send with QUIC is encrypted. Unfortunately QUIC does not stack up against the TLS-DHE mode as the forward secrecy is lost. QUIC trades off some security weaknesses for reduced connection delay/latency. Although some security is lost, this is only lost in the Integrity section of the CIA security triad; QUIC can maintain packet confidentiality, but an attacker has the ability to modify information in the packet without knowing what they are modifying.

The early results from Google are promising [7] with 93% successful connection establishments, a 5% page load time reduction, and a 30% reduction in re-buffering events on YouTube. Recently chartered as an IETF working group, QUIC is on its way to being specked. The ACCE security tests used to test a TLS connection are being re-evaluated for the new Quick Protocols like QUIC and the QACCE is in development.

2.5. Related Work

Although QUIC is a relatively young protocol, it has garnered a fair amount of attention in the research community. Several performance studies have been done on QUIC. Megyesi et. al. [3] have studied the bottle-necking that is caused by modern internet traffic and the failure of TCP to handle it properly. They elaborate on the limitations of TCP, HTTP/1.1 and SPDY, and show how QUIC tries to address these limitations. The authors test QUIC against HTTP/1.1 and SPDY in a number of live network scenarios. They find that both QUIC and SPDY have an advantage when loading pages with a high number of small objects due to their multiplexing abilities, but SPDY suffers from HOL blocking in scenarios with high packet loss. They do not study the performance of HTTP/2.

Additional research by R. Lychev et. al. delves into the security standards implemented by the QUIC protocol [11]. The research shows that it is not quite as good as existing TLS implementations as there is no forward secrecy, it is vulnerable to replay attacks, and security downgrade attacks. They show how these vulnerabilities are caused by low latency mechanisms and give us an idea of how these limitations could be overcome in future versions of QUIC. The bulk of comparisons are limited to TLS and QUIC Crypto.

Carlucci et. at. [12] provides an overview of the basic functionality of QUIC, specifically, how it performs over lossy channels compared to existing protocols and how its Forward Error Connection (FEC) module impacts performance. They study the impact of QUIC on goodput, link utilization and compared page load time for QUIC, SPDY and HTTP/1.1. The benefits of multiplexing are also explored, and Carlucci shows how it solves TCP's head of line blocking problem by providing many independent streams of information through a single connection. The authors find that QUIC has better link utilization than HTTP/1.1, especially under high loss. Further, QUIC loads pages faster than HTTP and outperforms SPDY in high loss. However, when FEC is enabled the goodput is reduced (FEC packets use 33% of available bandwidth). In our work, we account for a larger parameter space and use a more recent version of the QUIC protocol with an increased number of parallel streams. We also test the protocol against HTTP/2

rather than HTTP/1.1.

Alshammari & A. Al-Mogren [13] survey the current status of the HTTP protocol, comparing HTTP/1.1/2 against QUIC and SPDY. The focus of this paper is on the specific performance issues experienced by HTTP and shows how some of the ideas implemented in QUIC could help to mitigate those issues. They examine HTTP/2 in both modern Web and mobile sensing-based applications, performing benchmark studies in a variety of areas such as packet loss, energy consumption, and number of requests per second. However, their tests are very limited in scope and mostly focus on HTTP/1.1 vs HTTP/2.

Unlike most studies previous studies, we compare QUIC against HTTP/2 rather than the SPDY protocol, as it is the new Internet web standard and currently accounts for more than 60% of all HTTPS web traffic [14]. We also include results for an uncontrolled scenario over a home WiFi network, as this is the most typical setting for the average user.

3. Methodology

Page Load Time (PLT) was used as the primary metric of performance. Two experiments were set up to test this measure: a controlled scenario, which consisted of a local client/server and specified network parameters; and an uncontrolled scenario, which utilized a server running in the cloud and accessed with a local client via the Internet. Both scenarios provided us with an empirical body of evidence upon which to base conclusions of QUICs performance. Several steps were taken to ensure testing was fair, repeatable and reliable. For each of the test scenarios, the same machine was used to ensure consistent results.

3.1. Controlled Experiment

For the controlled scenario, two servers were set up to test the PLT downloading various html pages using QUIC and HTTP/2 (TCP), respectively. In both cases, the Chrome browser was used as the client with caching disabled.

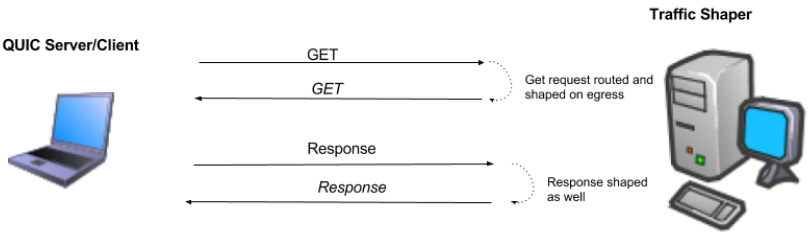


Figure 6: The controlled scenario testing environment

To emulate the various network scenarios, the traffic conditioning function of the native Linux network emulator (Netem) was used. In order to achieve the most unbiased performance metrics possible, traffic was routed through a secondary machine which acted as a traffic shaper and applied the network conditioning. This set-up is depicted in Figure 6.

The connection from client/server to the shaper server was made over ethernet where we manually assigned IP addresses to the interfaces and set up back-pipes on the shaper server to control traffic conditions. The transmission control utility shapes traffic on egress only, in our setup we had 2 moments of egress for every connection, so we halved the assigned shaping numbers to account for the double exposure to the shaper server each request. Lag and packet loss were confirmed with the ping utility.

3.1.1. QUIC Server

The QUIC source code is open-sourced in the Chromium browser. As Chromium is designed for serious developers and includes many unnecessary features (for our purposes), we opted to use the proto-quic library instead, which incorporates the bare minimum of the QUIC codebase needed to run the basic QUIC server provided by Google (see [15]). In order to run the server, a self signed certificate was generated and added to the OSs trusted certificate store. This is because the security encryption QUIC runs on is built into the protocol and certificates are required for the initial QUIC handshake.

3.1.2. TCP Server

Unfortunately, the experimental nature and relative youth of QUIC meant that our QUIC server was only able to serve pages requested over the QUIC protocol. In order to

achieve comparative TCP metrics, a secondary server was constructed that could handle these types of requests.

To get comparative TCP performance metrics, the OpenLiteSpeed server was used. OpenLiteSpeed was chosen because of its similar architecture to the QUIC server (both using C languages or C language derivatives), ability to run using the latest HTTP/2 specifications, and its ease of configuration and lightweight architecture. The TCP server was configured to use HTTP/2 with SSL enabled. A certificate was generated and introduced to the browser's certificate store which enabled all requests to be made under TLS. This was necessary as QUIC was using encryption as well, and allowed us to test the differences in RTT for the initial handshake between the two protocols when a security layer is involved.

3.1.3. Test Pages

Both servers were responsible for serving up a number of different html files. These files were identical for each protocol, but varied in the number and size of embedded objects present. Objects were defined as being very small (100B), small (1Kb), medium (10Kb), and very large (300Kb). The very large size was added as it showcased some interesting differences between QUIC and TCP in the areas of how loss was handled, as well as a good showcase of the benefits of multiplexing channels. For each object size, pages were created with 2, 64, or 128 of that object, for a total of 12 distinct test pages. Figure 7 provides the complete parameter space for the controlled experiments.

Objects (size)	100b, 1Kb, 10Kb, 300Kb
Objects (#)	2, 64, 128
Loss (%)	0, 2
RTT (ms)	5, 100, 200

Figure 7: Parameter space for controlled experiment

3.1.4. Capture

The actual performance testing was done using a HAR (HTTP Archive format) capturer utility authored by github user Andrea Cardaci. This utility allowed us to automate much of our testing, as PLT is able to be extracted from HAR files. A specialized script was written which would hit each of the test pages and record the resulting PLT.

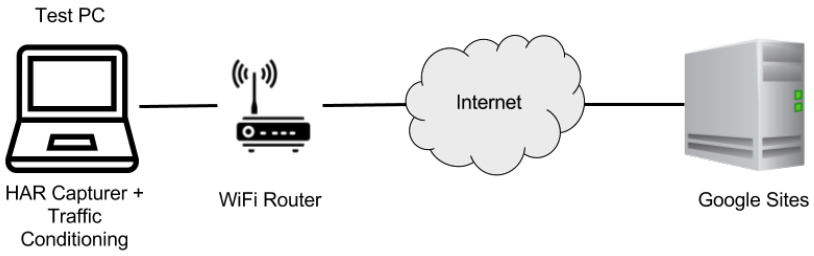


Figure 8: The uncontrolled scenario testing environment

For each of the predefined network scenarios, each page request was made 30 times and the average PLT was calculated which we used for our results. During testing, it was ensured the computer running the client/server had minimal background processes running to minimize interference. The tests were rerun multiple times on multiple dates to ensure consistency.

3.2. Uncontrolled Experiment

For a better look at how production QUIC compares to TCP, several test pages were constructed on Google Sites in a similar fashion to the test cases outlined above. Google Sites is a content management system. Because it is hosted by Google, QUIC is the default protocol used making live data capture possible. Six pages were constructed containing a varying number of objects and object sizes ranging from 10 to 196 objects. The objects consisted of sizes small (100-500B), medium (500B-30Kb), or large (30Kb-300Kb). The automated HAR capture script from the controlled test was modified and used to automate the page requests, making each request 20 times over a regular home WiFi network (approx. bandwidth 150Mbit/s) and reporting the average page load time. Tests were repeated for every network load scenario.

Objects (size)	100B-500B, 500B-30KB, 30KB-300KB
Objects (#)	10, 196
Loss (%)	0, 2
RTT (ms)	25, 75, 125, 225

Figure 9: Parameter space for uncontrolled experiment

This data was meant to supplement the controlled experiment data and show how QUIC currently performs at the production level, rather than a controlled data set to be contrasted with the controlled data.

4. Results

Our research tested QUIC versus TCP and HTTP/2 in controlled and uncontrolled environments. The main results for these tests can be found in figures 10 and 14. In general it was found that QUIC outperformed TCP and HTTP/2 in cases where a % packet loss was introduced. When loss was introduced, QUIC's FEC was able to work and really showed how it was able to help maintain a low latency for the protocol.

4.1. Controlled

TCP /w SSL	128 @ 300Kb	64 @ 300Kb	2 @ 300Kb	128 @ 10Kb	64 @ 10Kb	2 @ 10Kb	128 @ 1Kb	64 @ 1Kb	2 @ 1Kb	128 @ 100b	64 @ 100b	2 @ 100b
0% (5ms)	320.00	200.00	120.00	351.52	192.54	20.76	105.00	115.00	65.00	68.00	43.00	38.00
2% (100ms)	6537.42	4531.96	1550.00	2350.00	1520.85	1275.34	1827.38	1234.75	813.75	642.54	592.37	538.43
2% (200ms)	15875.84	11243.37	3175.85	4195.46	3376.52	1834.75	3957.52	2575.86	1386.45	821.37	759.38	1754.39

QUIC	128 @ 300Kb	64 @ 300Kb	2 @ 300Kb	128 @ 10Kb	64 @ 10Kb	2 @ 10Kb	128 @ 1Kb	64 @ 1Kb	2 @ 1Kb	128 @ 100b	64 @ 100b	2 @ 100b
0% (5ms)	605.00	323.14	65.84	199.03	129.51	49.28	158.79	111.79	50.58	184.03	103.12	50.64
2% (100ms)	3275.24	2431.26	817.54	1199.27	937.58	621.54	1029.49	826.91	625.82	1169.69	808.51	435.27
2% (200ms)	8672.42	5237.49	1642.48	2096.03	1836.93	1309.05	2004.88	1463.51	1228.80	2046.25	1544.36	1143.90

Figure 10: Controlled Testing Results

As seen in Figure 10, the large quantity of green (indicating the better time of the two) shows plainly that in most cases QUIC outperformed TCP with SSL encryption. As loss was introduced, the delay for both groups increased, but beacuse of QUICs built in Forward Error Correction, it began to perform better than TCP immediately. Even with increasing the delay QUIC became significantly faster. In the following figures 11 12 13 TCP will be shown in blue and QUIC will be shown in green.

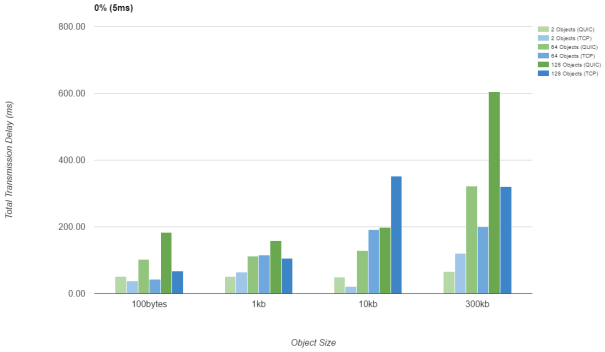


Figure 11: Controlled Test with 0% loss and 5ms added delay

In Figure 11, we are showing a comparison of the data where there was 0% loss and a 5ms induced delay. In the 100bytes category QUIC fails to be faster than all of the TCP attempts, but starts to catch up in the 1kb testing. Unfortunately for this testing category, QUIC does not manage to pull ahead of TCP in timing except in the 128 10kb objects. The reason for QUIC not pulling ahead is that once QUICs forward error correction begins to happen, it starts to outperform TCP as TCPs head of line blocking occurs.

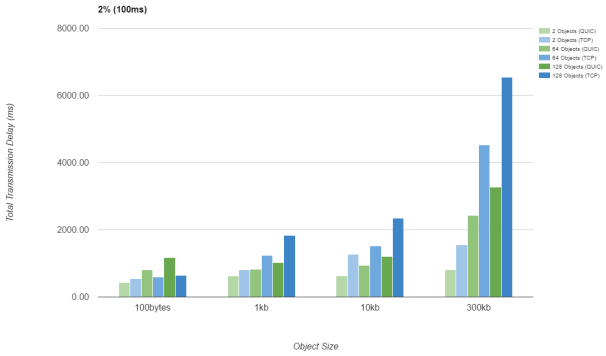


Figure 12: Controlled Test with 2% loss and 100ms added delay

Now that 2% loss has been introduced to the testing, QUIC should be improving its results over the TCP connections and it does as shown in Figure 12. Most of the tests in the 100bytes category the TCP connection was faster, but in all of the other connections, QUIC performed much better than all of the TCP tests. In all of the 300kb tests, QUIC performed almost 2 times better than TCP.

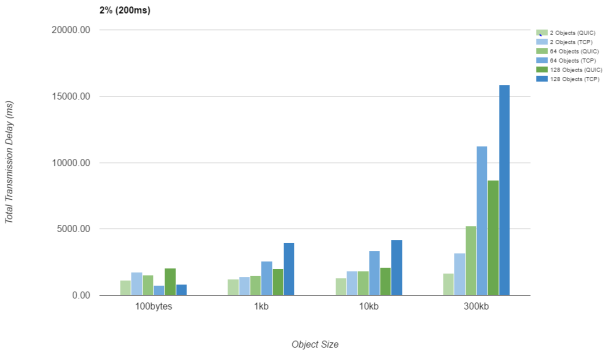


Figure 13: Controlled Test with 2% loss and 200ms added delay

Yet again, with the 2% loss but an increased delay to 200ms, QUIC has massively outperformed the TCP connection as seen in Figure 13. Like in Figure 12, QUIC outperformed TCP by almost double in all of the 300kb tests.

HTTP/2	high small	high-medium	high large	low small	low medium	low large
base (25 RTT)	2245.57	2806.25	3893.67	1562.48	1469.96	1600.41
L0 (+50 RTT)	5053.31	6134.58	6394.57	2592.05	2654.14	2732.70
L0 (+100 RTT)	7847.94	9021.47	9459.26	3722.86	3853.59	4089.17
L0 (+200 RTT)	5918.65	5980.61	6145.37	5963.7	5954.01	5924.07
L2 (25 RTT)	2648.72	3343.60	3877.77	1647.18	1793.56	2019.36
L2 (+50 RTT)	5796.23	6677.43	7128.77	2683.16	3015.96	2898.59
L2 (+100 RTT)	9326.96	10713.48	10821.75	3662.97	3997.66	4324.80
L2 (+200 RTT)	15516.48	18011.45	18726.92	6494.386	7172.89	7173.98

QUIC	high small	high medium	high large	low small	low medium	low large
base (25 RTT)	1580.50	2327.54	3237.99	1581.93	1468.94	1493.72
L0 (+50 RTT)	2686.37	3455.32	4032.86	2685.00	2628.32	2864.02
L0 (+100 RTT)	3932.44	4368.92	5067.29	3879.88	4146.05	4052.40
L0 (+200 RTT)	6476.77	7146.86	8034.79	6330.42	6570.78	6429.07
L2 (25 RTT)	1704.97	2741.44	3025.71	1575.11	1647.19	1835.15
L2 (+50 RTT)	2984.44	3394.49	4260.49	2629.70	2861.70	2795.80
L2 (+100 RTT)	4071.21	5380.59	5153.69	3937.77	4051.51	4127.04
L2 (+200 RTT)	8133.15	7538.29	9122.05	6391.33	6735.88	6897.86

Figure 14: Uncontrolled Testing Results

Figure 14 shows the results from testing in an uncontrolled environment. Yet again, QUIC outperformed its contender HTTP/2. Without a loss introduces to the scenarios, QUIC did not perform as well on the lower bandwidth tests; this is due to QUIC's FEC in action on the tests with an induced loss. In the following figures HTTP/2 versus QUIC's results will be analyzed in a bit more detail.

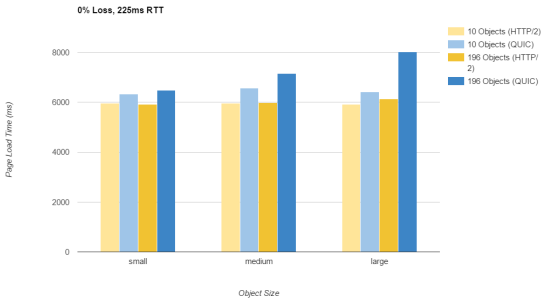


Figure 15: 0% loss, 225ms delay

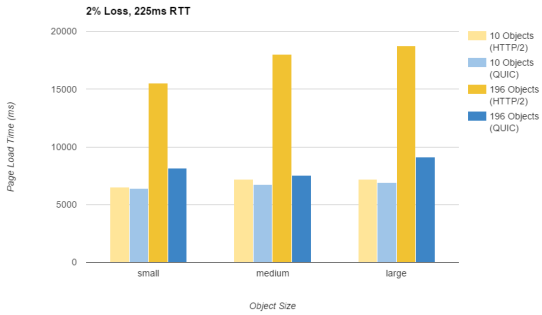


Figure 16: 2% loss, 225ms delay

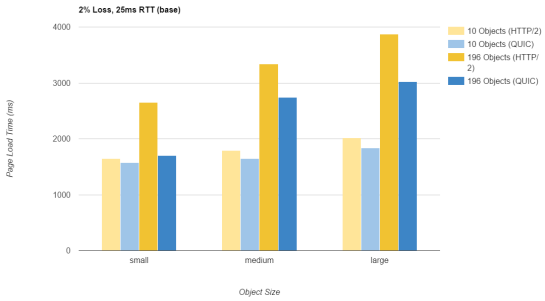


Figure 17: 2% loss, 25ms delay

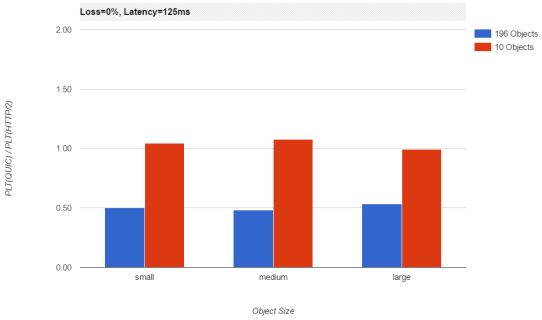


Figure 18: 0% loss, 125ms delay

As shown in Figure 15, both HTTP and QUIC have very similar results, but when a loss is added as shown in Figure 16, we can see that QUIC does more than 2 times better. This is more proof that QUIC's FEC is highly effective at maintaining a low latency even with some transmission failures. HTTP/2 really suffers in high loss and high latency scenarios when downloading a large number of objects as shown in Figure 16. Now if we remove the latency but keep the same loss rate as in Figure 17, we can see that QUIC still manages to outperform HTTP. In Figure 18, we can see that with a high quantity of objects, QUIC outperforms HTTP even with no induced loss, but when there are a low quantity of objects QUIC seems to lack its advantages.

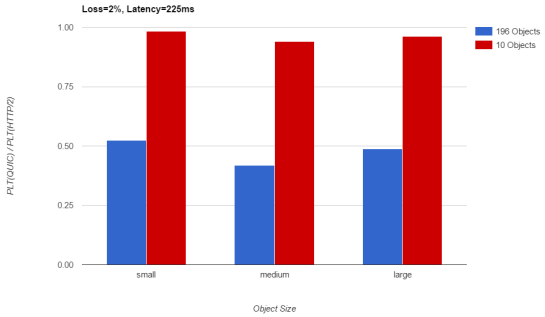


Figure 19: 2% loss, 225ms delay

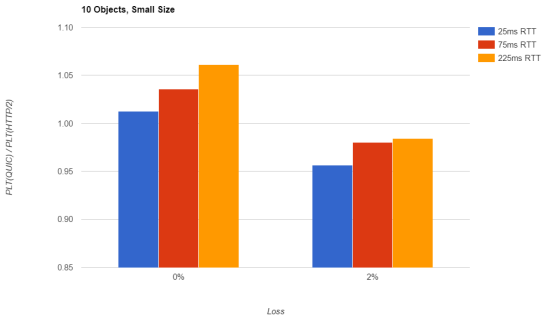


Figure 20: Ratio QUIC/HTTP @ 0% and 2% loss with 10 objects at a small size

To expand further on QUIC's problems with low objects, in Figure 19 we show that even with induced loss where QUIC should take a substantial lead, HTTP still manages to produce similar results when there are a low number of objects to be loaded. Although, as soon as the object count jumps up, QUIC is about twice as fast as HTTP.

Even with a low object count, there is still a visible difference in performance improvements of QUIC if there is an induced loss. In Figure 20 we compare the ratio of QUIC/HTTP when there are a low quantity of objects with different delays and under two different loss situations. At 0% loss, HTTP outperforms QUIC in all delay scenarios, but with a 2% loss QUIC pulls ahead in speed. All of the uncontrolled tests have shown that QUIC is very good at handling packet losses during transmission.

5. Conclusion and Future Work

QUIC ends up being faster in all scenarios where there is an induced packet loss during transmission. It manages to perform reasonably well when the pages to load are small and consist of a few small objects, but fails when there are a large quantity of small objects. QUIC does perform exceptionally well when there are large sized objects to load on a page. QUIC achieves substantially better page load times when there are poor network conditions consisting of either low bandwidth, excess delay, or packet losses.

5.1. Limitations

As QUIC is still in its prototyping stages and not production-ready, the only publicly available QUIC server is not as refined as existing HTTP servers. Due to difficulties hosting a QUIC server, it is not possible to test the performance of loading embedded objects at multiple domains. In order for QUIC to function properly and achieve its best performance, three requirements need to be met: UDP's port 443 needs to be open to allow for an encrypted UDP connection, no rate limits can be set on any incoming/outgoing UDP connection, and UDP can't have any negative/worsened quality of service treatment compared to the rest of the network.

6. Member Contributions

The contributions of each team member are enumerated below:

- Carl: Wrote Abstract, Introduction(1.0), Background(2.0), Limitations of TCP and HTTP(2.1 & 2.3), and Related Work(2.5). Edited Limitations of UDP(2.2) and Methodology(3.x). Assisted in the setup of the controlled testing environment. Wrote HAR capture script. Built test pages on Google Sites. Performed results gathering and data analysis for the uncontrolled scenario.

- Kjalen: Wrote limitations of TCP, limitations of UDP, methodology section. Testing: installed and configured QUIC server/client (controlled). Designed test pages for controlled tests. Installed and configured Litespeed server for TCP tests. Designed and created TCP test pages. Modified Carl's HAR capture script for Controlled tests.

- Jakob: QUIC(2.4),Results(4.x), Conclusion(5.x). Assisted with testing. Created graphs for data analysis. Created slides for presentation. Created website.

7. References

[1] InternetLiveStats.com. (2016). *Number of Internet Users (2016) - Internet Live Stats* [Online]. Available: http://www.internetlivestats.com/internet-users/

[2] J. Rivera. (2015, Feb 5). *What's Driving Mobile Data Growth?* [Online]. Available: http://www.gartner.com/newsroom/id/2977917

[3] P. Megyesi et al. (2016, May 10). *How Quick is QUIC?* [Online]. Available: https://www.researchgate.net/publication/302585207_How_quick_is_QUIC

[4] Ian Swett. (2015, Jun 3). *Next generation multiplexed transport over UDP* [Online]. Available: https://www.nanog.org/sites/default/files//meetings/NANOG64/1051/20150603_Rogan_Quic_Next_Generation_v1.pdf

[5] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. *How Speedy is SPDY* in Proc. NSDI14, 2014, pp. 387399.

[6] Y. Elkhatib, G. Tyson, and M. Welzl. *Can SPDY Really Make the Web Faster?* in IFIP Networking Conference, 2014, pp. 19.

[7] L. Lumiaho, B. Chemmagate, S. Hagglund, J. Ott. (2017, Feb 3). *Evolution of Web Protocols: HTTP/2 and QUIC* [Online]. Available: https://www.callstats.io/2017/02/03/web-protocols-http2-quic/

[8] Wikipedia. (2017). *Hypertext Transfer Protocol — Wikipedia, The Free Encyclopedia* [Online]. Available: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[9] Wikipedia. (2016). *HTTP Pipelining — Wikipedia, The Free Encyclopedia* [Online]. Available: https://en.wikipedia.org/wiki/HTTP_pipelining

[10] HTTP/2 Specifications. (2017). *Github - HTTP/2 Frequently Asked Questions* [Online]. Available: https://http2.github.io/faq/

[11] R. Lychev et al. (2015, Sep 16). *How Secure and Quick is QUIC? Provable Security and Performance Analyses* [Online]. Available: http://www.cc.gatech.edu/~aboldyre/papers/quic.pdf

[12] G. Carlucci et al. (2015, Apr 13). *HTTP over UDP: an Experimental Investigation of QUIC* [Online]. Available: http://c3lab.poliba.it/images/3/3b/QUIC_SAC15.pdf

[13] A. Alshammari & A. Al-Mogren. (2016, Aug 29). *HTTP/2 in Modern Web and Mobile Sensing-based Applications Analysis,Benchmarks and Current Issues* [Online]. Available: https://www.omicsgroup.org/journals/seqsplithttp2-in-modern-web-and-mobile-sensingbased-applications-analysisbenchmarks-and-current-issues-2332-0796-1000193.pdf

[14] HTTP/2 Statistics. (2016). *HTTP/2 Statistics KeyCDN Report on HTTP/2 Distribution* [Online]. Available: https://www.keycdn.com/blog/http2-statistics/

[15] Google developers. (2017). *proto-quic* [Online]. Available: https://github.com/google/proto-quic