QUIC: Investigating How the Protocol Achieves its Goals

Austin Benoit (V00818146)

Chris Holland (V00876844)

CSC 466: Overlay and Peer-to-Peer Networking

University of Victoria

March - April 2020

Table of Contents

Abstract

There has been plenty of work comparing the efficiency of QUIC and TCP/TLS that largely lead to the conclusion that QUIC successfully outperforms TCP/TLS in nearly all scenarios. In this report we aim to explore how QUIC achieves the original goals of its designers, and map some of its key features, such as 0-RTT connection establishment and stream multiplexing to its performance results and implementation. QUIC has found large success, being used nowadays in Google Chrome and YouTube, but the protocol is also far from perfect. We take aim at a few of QUIC's key features and analyze how the implementation achieves, or falls short of the goals of the protocol. We found that QUIC's implementation at the application layer intuitively fulfills the goal of being easily modifiable and deployable, however this may cause strain on the endpoints in terms of increased CPU usage. QUIC's 0-RTT time has also been shown to greatly reduce latency, however is prone to various attacks that may harm performance. Furthermore, QUIC has achieved its goal of stream multiplexing. QUIC adds additional information to packets to provide stream abstractions to end hosts. The benefits of stream multiplexing can be seen in contrast to TCP in preventing head of line blocking.

1. Introduction

QUIC (Quick UDP Internet Connections) is a transport layer mechanism that was originally created by Google with hopes of reducing the latency time of the current TCP stack. Over the years, QUIC has been standardized by the IETF, with the original protocol created by Google now being called gQUIC, whereas the standardized version of QUIC is not an acronym. The standardized version of QUIC has now deviated significantly from gQUIC in multiple ways. QUIC is implemented on top of UDP, meaning its implementation is manifest at the application layer; this allows changes to QUIC to happen rapidly as modifications to transport layer protocols like TCP are implemented in operating system kernels, which means making updates to them require rigorous testing and analysis. Using UDP as the datagram transport enabled QUIC to be deployed on the current internet. QUIC was developed with a few goals in mind, including:

- Deployable on current internet
- Reducing the latency in communications created by TCP
- Ease of maintainability and deployability
- End-to-end encryption
- Stream multiplexing

Countless papers exist already that aim to measure the differences between TCP and QUIC in terms of latency, speed, upload/download time, etc. We do not wish to replicate such experimentation, but instead bridge the gap in understanding between the design goals of QUIC and the common test findings. In this report we take a look into the implementation and design of QUIC to determine how the protocol attempts to achieve these listed goals. Depending on the goal being analyzed, we used a variety of techniques to gain a deeper understanding of QUIC. These techniques range from the use of "toy" QUIC clients and servers, to analysis of the QUIC source code itself.

2. Related Work

There are a number of papers that we found useful while performing research for this paper. In this section some of the most influential will be overviewed. Please see the Bibliography for all of our sources.

Overview of QUIC: [1] Goes into detail about the original goals and motivation behind the implementation of QUIC. Tests are run comparing QUIC to the TCP/TLS stack in terms of speed and latency. This paper helped us develop our fundamental understanding of the motivations and design of QUIC.
Evolution of QUIC: As QUIC is a rapidly evolving protocol, there have been few papers aiming to analyze QUIC's architecture and implementation. [2] Aims to explore how updates to the protocol have changed QUIC's performance, along with deriving a state machine for QUIC's execution.
QUIC Multiplexed Stream Transport Over UDP: [3] The original design document of QUIC. Outlines all of the goals of the new protocol. Furthermore, the groundwork for QUIC is presented and details about the implementation are given.
A Replicable and Extensible Comparison of QUIC and TCP: [4] Is a project on QUIC done by students in a previous offering of CSC 466. Their research focuses primarily on the measurable differences in latency between TCP and QUIC. During our own research, we referenced their results to help us analyze QUIC's implementation details.

3. Ease of Maintainability and Deployability

TCP and UDP are transport layer protocols that are implemented in the kernels of operating systems. This means that making changes to transport layer protocols would involve rework and redeployment of the operating systems of nearly all computers. The designers of QUIC aimed to have a protocol that was easy to maintain and redeploy, so they implemented the main logic of QUIC in the application layer, or user space, while using UDP at the transport layer. The contrast in network protocol stacks can be seen in Figure 1, with the TCP/TLS stack on the left, and QUIC on the right.

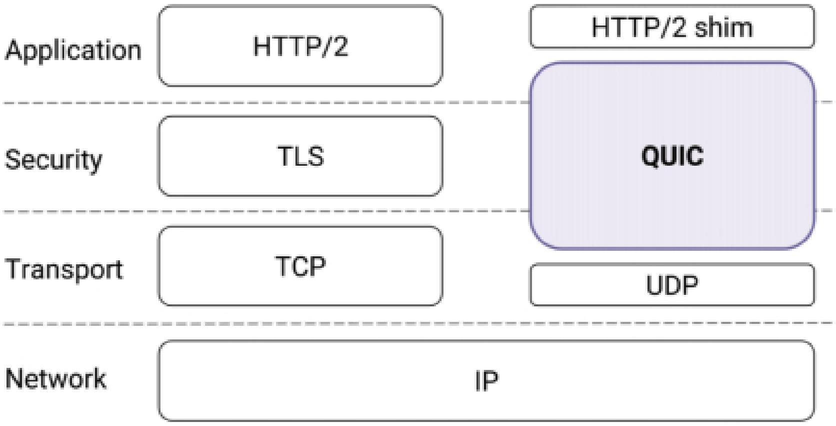
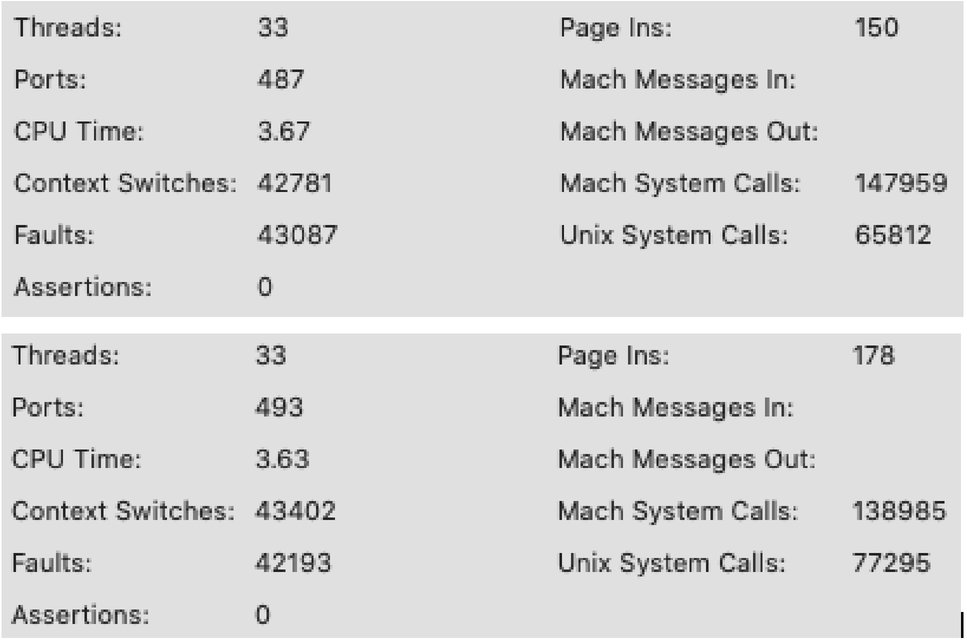


Figure 1: TCP/TLS Protocol Stack verses QUIC [5]

Since QUIC is implemented at the application layer in applications like Google Chrome and YouTube, it is much easier to update and redeploy, as applications can be updated far quicker and more easily than operating system kernels. One potential downside we see with this architecture design is that it may put more of a computational load on the applications (the servers and clients). This would imply and increase in CPU usage at the endpoints. The reasoning for this is simple yet not immediately obvious; TCP runs in the OS kernel, and is extremely optimized to run in the OS. Transferring heavy networking logic to the application layer, or user space, will increase the CPU usage as the application code is less optimized, and more communication with the kernel via system calls will be used.

3.1 Analysis of Running the QUIC Server and Client

We installed the toy QUIC client and server from Google's Chromium Project in order to gain an understanding of the runtime behaviours of QUIC. We were able to view the packets exchanged during the connection establishment and data exchanges (as discussed on our website) and use that to help us understand the connection establishment process as explained in Section 4. However, tests comparing the CPU usage between two very different implementations of client/servers would have not been an honest comparison. Instead we decided to use Google Chrome because you are able to choose between using TCP or QUIC. We opened Chrome and loaded www.example.org using both TCP and QUIC and the results were noticeable; while using TCP the CPU usage topped at 12%, and while using QUIC the CPU usage topped at 16%. We ensured to clear the cache before running the tests for both TCP and QUIC. This was performed on a late 2014 Mac Mini with a 3 GHz Dual-Core Intel Core i7 with 8GB 1600 MHz DDR3 memory running macOS Catalina and Chrome version 80.0.3987.149. Below in Figure 2 some of the system utilization metrics are shown using Apple's Activity Monitor. The TCP case is on top while the QUIC case is on the bottom.



Threads:	33	Page Ins:	150
Ports:	487	Mach Messages In:	
CPU Time:	3.67	Mach Messages Out:	
Context Switches:	42781	Mach System Calls:	147959
Faults:	43087	Unix System Calls:	65812
Assertions:	0		

Threads:	33	Page Ins:	178
Ports:	493	Mach Messages In:	
CPU Time:	3.63	Mach Messages Out:	
Context Switches:	43402	Mach System Calls:	138985
Faults:	42193	Unix System Calls:	77295
Assertions:	0		

Figure 2: Google Chrome QUIC verses TCP Metrics

As we can see, while using QUIC there were 11483 more Unix system calls, and 28 more page load-ins. Although QUIC is faster than TCP at loading web pages, the CPU usage at the end points (in our case the client side) is greatly increased. However, this design may be a necessary evil. CPU speed rapidly increases each year, while the speed of data transfer over a network does not. Increasing the computational load at the endpoints may be a wise decision as it pushes a larger load to a component of the stack that will scale more quickly and cheaply. So although QUIC seems to increase the computational load at the endpoints, literature shows that QUIC almost always outperforms TCP and increases utility, which is far more important.

4. Reduced Latency over TCP

Reducing latency is one of the biggest aims for network protocol engineers, including the creators of QUIC. When considering how to reduce latency compared to TCP, the engineers of QUIC decided that reducing the RTT (round-trip delay time) of connection establishment would be an approach to take. Deciding to reduce RTT within the protocol design itself stems from two observations, (1) Bandwidth is cheap and will continue to grow, (2) Information cannot travel faster than the speed of light. In this section we will be analyzing the inner workings of QUIC's 0-RTT and its benefits, along with any issues it poses.

4.1 QUIC Connection Establishment

To reduce latency, QUIC attempts to connect the client and server with a 0-RTT handshake. Upon initial connection of a client and server QUIC will use a 1-RTT handshake, and then cache the cryptographic protocol parameters to achieve a 0-RTT for future connections. One could think of the 0-RTT as a resumption, however the QUIC crypto does not have the same notion of a resumption like TLS does. In Figure 3 the various scenarios for establishing a QUIC connection are demonstrated.

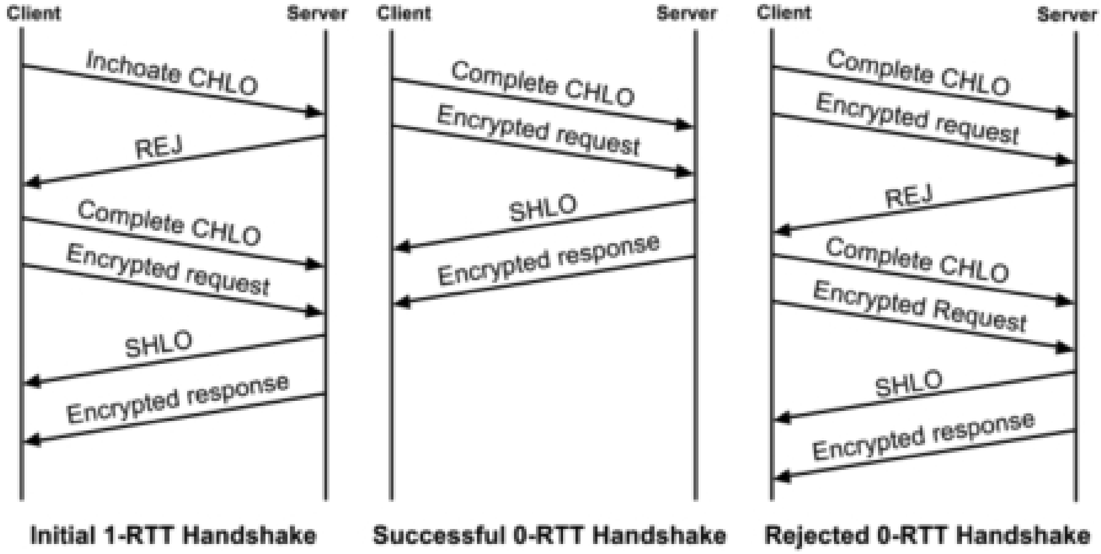


Figure 3: QUIC Handshakes [5]

4.1.1 Initial 1-RTT Handshake Overview

Before a client and server may connect with a 0-RTT Handshake, they must first initially connect with a 1-RTT Handshake. Below are the steps explaining the 1-RTT connection in Figure 3.

Step 1: During the Initial 1-RTT Handshake, the client sends a connection request to the server, which is known as the Inchoate CHLO. In this message the client will send a cid (connection ID) to the server so that the server may help create the connection.

Step 2: Since the client and server are initiating a brand new connection, the server will initially reject (REJ) but send the client a message with the cid, scfg (server configuration), and a stk (source-address token) used to prevent spoofing.

Step 3: Once the client receives the connection parameters from the server, it will generate Diffie-Hellman values and send the public key along with the cid, stk, and scfg to the server to complete the client hello (CHLO).

Step 4: At this point the client may begin to make requests to the server while the server is creating its own keys using the clients public key. It is important to note that the client will make an encrypted request *before* the server has its encryption parameters set, however, this is a part of the design that helps reduce latency.

Step 5: The server then sends the server hello (SHLO) with the cid and its newly generated public Diffie-Hellman key. At this point the connection is established.

Step 6: The server finally sends the encrypted response to the client's request from step 4 and further requests can be made by the client.

4.1.2 0-RTT Handshake Overview

Once a server and client have successfully connected using an initial 1-RTT Handshake, they may continue to connect and make requests using a 0-RTT Handshake; this is one of the big innovations of QUIC. If the stk and scfg have not yet expired, the client only needs to choose a new cid and it can then request a new connection using the pre-existing, cached, stk and scfg values. Below are the steps explaining the 0-RTT connection in Figure 3.
Step 1: The client picks a new cid and sends a client hello (CHLO) to the server with the new cid, the previous stk and scfg values, and a new Diffie-Hellman public key. The server verifies the stk and creates a new Diffie-Hellman public key for the session. The client may also make requests after sending the client hello.
Step 2: Data can now be exchanged. The server will first send a server hello (SHLO) including the cid and the servers new Diffie-Hellman public key, and then the response to the clients request in step 1.
Step 3: The connection is now established via the 0-RTT Handshake and data may be exchanged between the client and server.

4.1.3 QUIC Latency Empirical Results

We can now compare QUIC’s 0-RTT handshake for repeat connections to TCP’s 3-way handshake for all established connections (with the exception of the 4-way handshake on rare occasions). Figure 4 represents findings by Google while comparing metrics of QUIC and TCP and compares the handshake process.

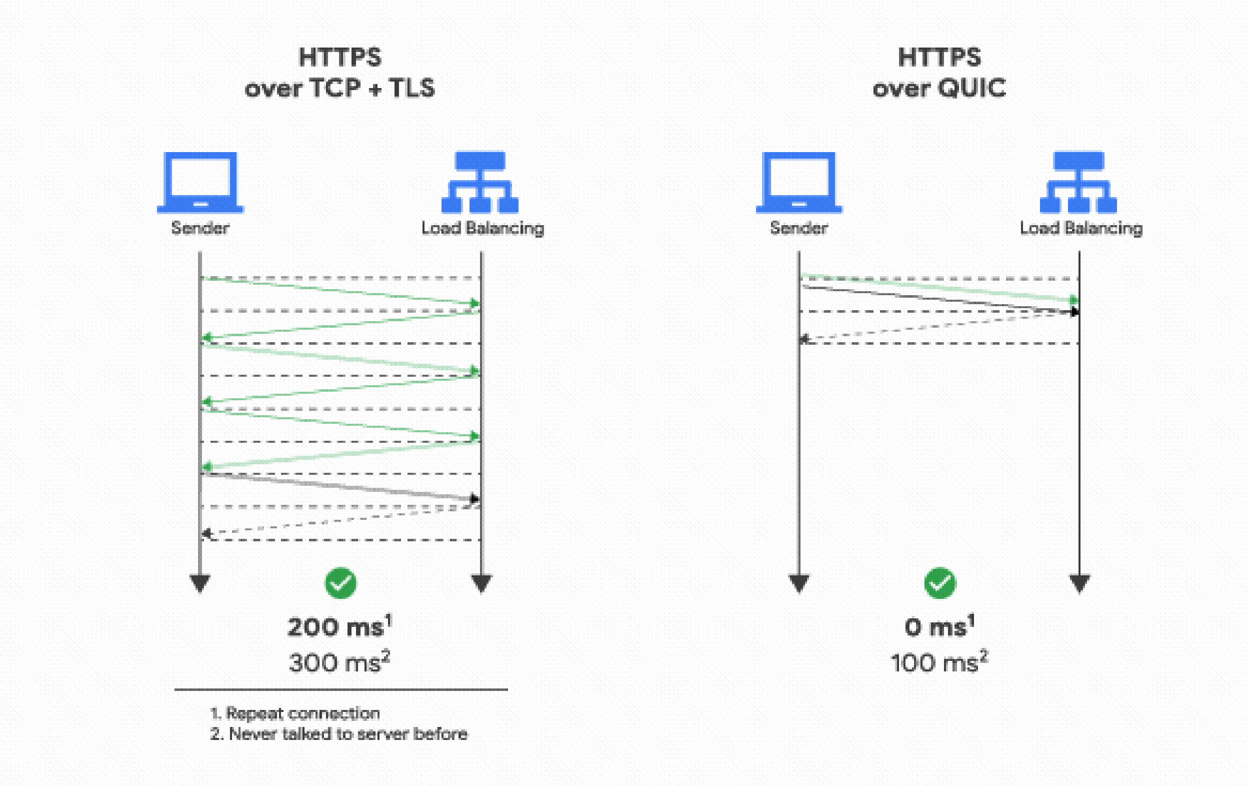


Figure 4: TCP/TLS vs QUIC Connection Establishment Latency [6]

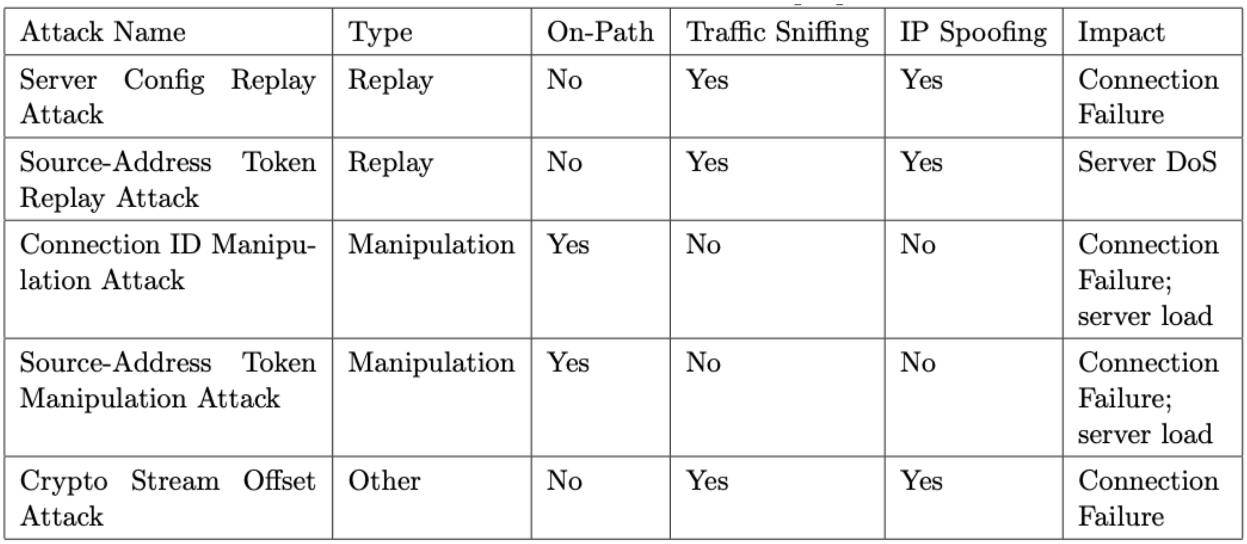
In theory it is evident that QUIC has faster connection establishment compared to the commonly used TCP+TLS protocol stack. Tests comparing the performance of QUIC vs TCP generally prove this sentiment. In a few cases throughout various studies, it is believed that QUIC has no performance increase (or matches) the performance of TCP. For example, in a previous analysis of QUIC [4] it was discovered that QUIC outperforms TCP in all tested cases except for the case where the client requests a large amount of objects from the server. Another study determined that QUIC outperformed TCP in all of their tested scenarios except while streaming non-high definition video from YouTube [2].

In the case of TCP outperforming QUIC when the client requests a large amount of objects, [4] believes that is due to non-optimal multithreading in the QUIC client/server implementation they were basing their experimentations off of. This is likely, as efficiency in requesting objects from a web server and page load times are generally determined by the RTT and multiplexing (discussed in Section 4) [2]. On the other hand, video streaming relies heavily on the efficiency of the transport layer to load quickly and maintain smooth playback [2]. YouTube supports both TCP and QUIC, and Google tests have reported that a video streamed using QUIC has on average 18% less rebuffers compared to the same video streamed over TCP [5]. YouTube uses a video buffer of a few seconds or so in order to ensure smooth playback; for this reason, it may be possible that the reason QUIC doesn't out-perform TCP when the video is not high-resolution is because the buffer and TCP streaming mechanisms for YouTube are already sufficient enough for basic use.

4.2 Security Concerns with QUIC's Connection Establishment

In this section we will overview some of the security concerns associated with QUIC's connection establishment. As of now, we are only aware of attacks that compromise the availability of a server or connection that uses QUIC; we are not aware of any exploits that may reduce the authenticity or integrity of the endpoints in communication or the data being transmitted. Ironically, [6] found that the mechanisms QUIC uses to lower latency can be hijacked in order to create denial of service (DoS) attacks. They were able to create types of attacks that (1) exploit the cacheable connection parameters, and (2) exploit unprotected fields in the packets. The following Table 1 is their illustration of the types of attacks along with their impacts. We recommend reading [6] further if you are interested.

Table 1: QUIC Connection Establishment Exploit Classifications [6]



Attack Name	Type	On-Path	Traffic Sniffing	IP Spoofing	Impact
Server Config Replay Attack	Replay	No	Yes	Yes	Connection Failure
Source-Address Token Replay Attack	Replay	No	Yes	Yes	Server DoS
Connection ID Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Source-Address Token Manipulation Attack	Manipulation	Yes	No	No	Connection Failure; server load
Crypto Stream Offset Attack	Other	No	Yes	Yes	Connection Failure

These security concerns and their potential solutions create somewhat of a paradox. On one hand, reducing latency compared to TCP is one of QUIC's highest priorities, yet these relatively straightforward attacks compromise the latency of QUIC connections. Furthermore, implementing security features to combat these security concerns would likely increase the latency as well. However, security is also a high priority for QUIC, so if these security issues were to be addressed it would probably include a rework of the mechanisms of the protocol. In [6] they note that the byte stream QUIC uses may not be the best way to exchange handshake data, and recommend a message stream instead.

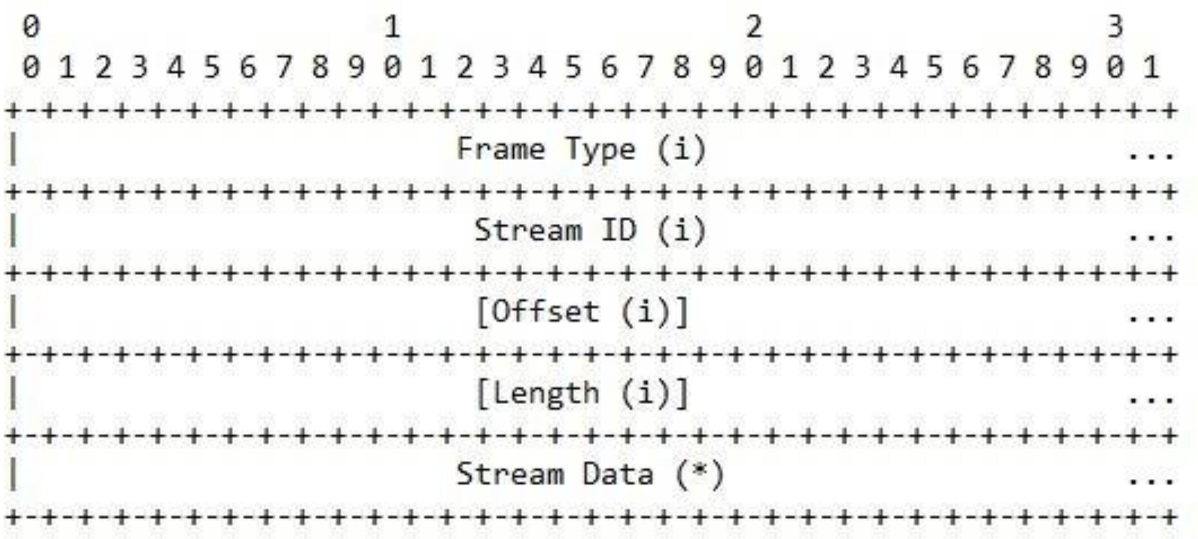
5. Data Communication

The basis for communication in QUIC is a stream. Streams provide the abstraction for data communication that an application will see. Currently QUIC supports byte order bidirectional or unidirectional streams [7]. Much of the following sections have been created using [7] as a reference. An application can open an arbitrary number of streams, to an agreed upon limit. During the connection setup each end point will advertise the maximum number of supported streams and the lowest number of streams will be the maximum supported streams. Streams can be used concurrently and can be created by both endpoints. QUIC therefore supports stream multiplexing in a connection. During communication a stream can be opened, closed or reset. A stream may persist throughout a communication session or it may only be used to send a small piece of data. To achieve stream multiplexing in a communication session, each stream is given a stream ID, which is unique to each stream in a communication session. Stream IDs are created in numerically increasing order. The initial stream ID will be zero. Once at least one stream has been created the two end hosts can begin communicating.

5.1 Sending and Receiving Data

To send data across the wire, several layers of encapsulation occur. First streams are broken into stream frames. Then the stream frames are put inside of a QUIC packet. Finally QUIC Packets are put into UDP packets that are then sent from host to host. The specifics of each process is described in greater detail in QUIC Transport Protocol RFC [7]. Streams in a connection are seen as independent from one another.

As stated stream frames encapsulate the data sent in a stream. Streams are the abstraction available to the application, while stream frames are the data that QUIC is going to send from one host to another. A stream frame consists of the following information: frame type, stream ID, offset (optional), length (optional) and stream data. The frame type contains flags that indicate whether or not the offset or length has been included. It also has a flag to indicate if this is the last frame in a stream. Offset informs the recipient of the byte offset of the data, while the length is the length of the data. The offset is the offset of the data in the frame and the first byte of any stream has an offset of 0. A stream data frame is shown in figure 5.

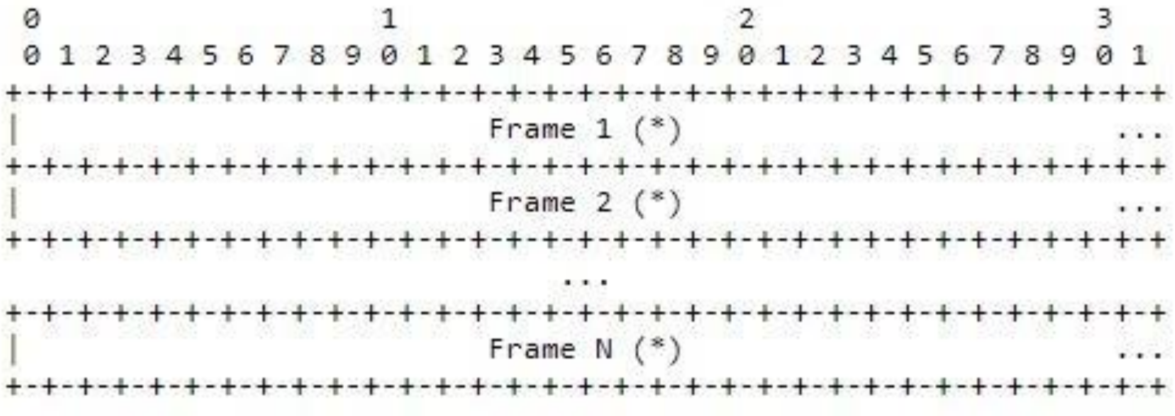


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Frame Type (i)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Stream ID (i)                       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         [Offset (i)]                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         [Length (i)]                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Stream Data (*)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: Stream Frame [1]

There are frame types other than the stream frame. The other frame types are used to facilitate certain aspects of the connection. The current QUIC implementation has 20 frame types whose general purposes are to facilitate data acknowledgment, cryptographic exchange, stream modification and maximum data values. The details of the other frame types are available in [7].

Stream frames are ultimately bundled in a QUIC packet. Each QUIC packet must have at least one frame and may have any number of frames from any number of streams. Figure 6 has an illustration of frames that are in a QUIC packet. It should be noted that a QUIC packet can only have frames from one connection. If a host has multiple QUIC connections, then each connection is independent from one another. More specifically, no streams from a particular connection may be bundled with streams from another connection. The QUIC packet is then placed inside of a UDP packet which is then sent across the wire. Again, if a host has multiple QUIC connections each UDP packet must only have QUIC packets from one connection. When received QUIC sends the QUIC packets to the appropriate connection. From there it is demultiplexed and the stream frames from inside of the packet are delivered to the appropriate stream. The application can then read the stream data and process as appropriate.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Frame 1 (*)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Frame 2 (*)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Frame N (*)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: QUIC Payload [1]

5.2 Stream Benefits and Drawbacks

The QUIC method of creating streams enables the end client to avoid the drawbacks of other protocols. QUIC is able to have per stream flow and error control. If a packet that contains a specific stream frame is lost then the error recovery is applied to the specific stream of the lost frame and not the QUIC packet. Such stream based control allows for all of the other streams to make progress and not be held up by error recovery for data that does not concern them. This is a benefit over TCP as if a TCP packet is lost all data must wait for the lost packet to be recovered first.

One specific area that QUIC should outperform TCP is in HTTP transfer of web pages, specifically HTTP/2. If not explicitly mentioned otherwise then HTTP shall refer to HTTP/2. In transferring web pages TCP may experience head of line blocking if a packet is lost. Take a simple web page which contains some text and 2 images. The general flow of HTTP communication would be a request for the webpage and then the requests for the images. In response the webpage server will return the request for the web page and then the requests for the images. If during that communication the response for the first image is lost, then TCP may induce head of line blocking for the other image. If the data for the second image has arrived, TCP will make the application wait for that data while it proceeds with error recovery for the lost packet. The effects of the head of line blocking has been shown in [9]. If the same scenario were to occur while the client and server are communicating with QUIC, the head of line blocking would potentially be avoided. In the QUIC case each of the HTTP requests may be associated with an individual QUIC stream. In the case that each request has its own stream, the loss of the packet containing the first image will not affect the second. QUIC will read the stream ID of the packet that it has received and send that data to the application. QUIC will then use error recover to detect the missing data from its first image request and begin error recover for the data lost. Thus each stream can continue independent of any other stream. This shows the big difference; QUIC has some knowledge about the data in each packet while TCP has no knowledge of what is in each packet.

5.2.1 Testing QUIC Head of Line Blocking

To verify if the QUIC prevents head of line blocking a simple test was run. A web page with multiple in page objects was created. The webpage will then be requested over a channel that has various degrees of packet loss to see if QUIC is able to prevent head of line blocking. TCP will be used as the control test and will highlight head of line blocking. To investigate head of line blocking the time it takes to download a page will be cross referenced with the waterfall diagram of the download time of the pages elements.

5.2.2 Experiment Set Up

The webpage is served through OpenLiteSpeed version 6.0.9 and accessed with Google Chrome version 80.0.3987.149. Both the client and server are running in Virtualbox instances of Ubuntu 18.04.4. According to http archives [10] the average web page has 76 total requests and 2080 KB of data transferred. To simulate this a webpage was created with 70 images each of 29.0 KB. The total requests are then 72 with a total data transfer of 2077 KB. To emulate a lossy channel GNS-3 is used. The channel loss rate starts with no loss, then 1%, 5%, 10% chance of loss are run. The channel is used with no latency. The server will use QUIC version 46 and TCP with HTTP/2. The QUIC connection shall use the 0-RTT method of communication while TCP will use TLS 1.2. To record the time it takes to download the page and the waterfall of requests downloaded, chrome's developer features will be used.

5.2.3 Experiment Results

Downloading the web page resulted in the download times seen in figure 7. In the case of no loss TCP is able to transmit the webpage faster and to a lesser margin in a one percent loss. From there QUIC is able to transmit the data faster. In the last case of 10% loss QUIC is able to transmit the page significantly faster. This shows that the QUIC protocol is not simply just faster than TCP but that there are some protocol differences that enable QUIC to perform differently from TCP. The protocol differences are more readily seen in the waterfall diagram figure 8.

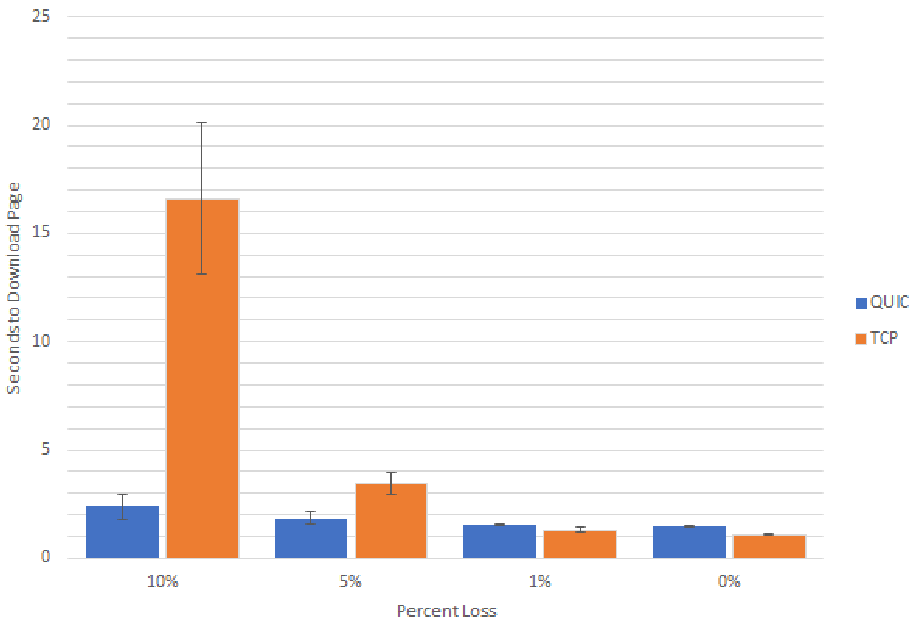


Figure 7: QUIC verses TCP Download Times

The waterfall figure shows the QUIC request handling versus the TCP handling. The time scales are not equivalent for both waterfalls. Only a snapshot of the waterfall is presented here, but the trends presented are consistent with the rest of the graph. As the packet loss is random a get request may be dropped or the data requested from the get may be dropped. In the case that the get request is dropped the request then has to wait for retransmission which is a separate issue, but an issue that should affect both protocols equally. The green portion of the waterfall is the time from when the request is issued to the time when the first byte of data is received. The blue portion is the time it takes for the data to be downloaded. The data is ordered by the time the first data byte has been revived.

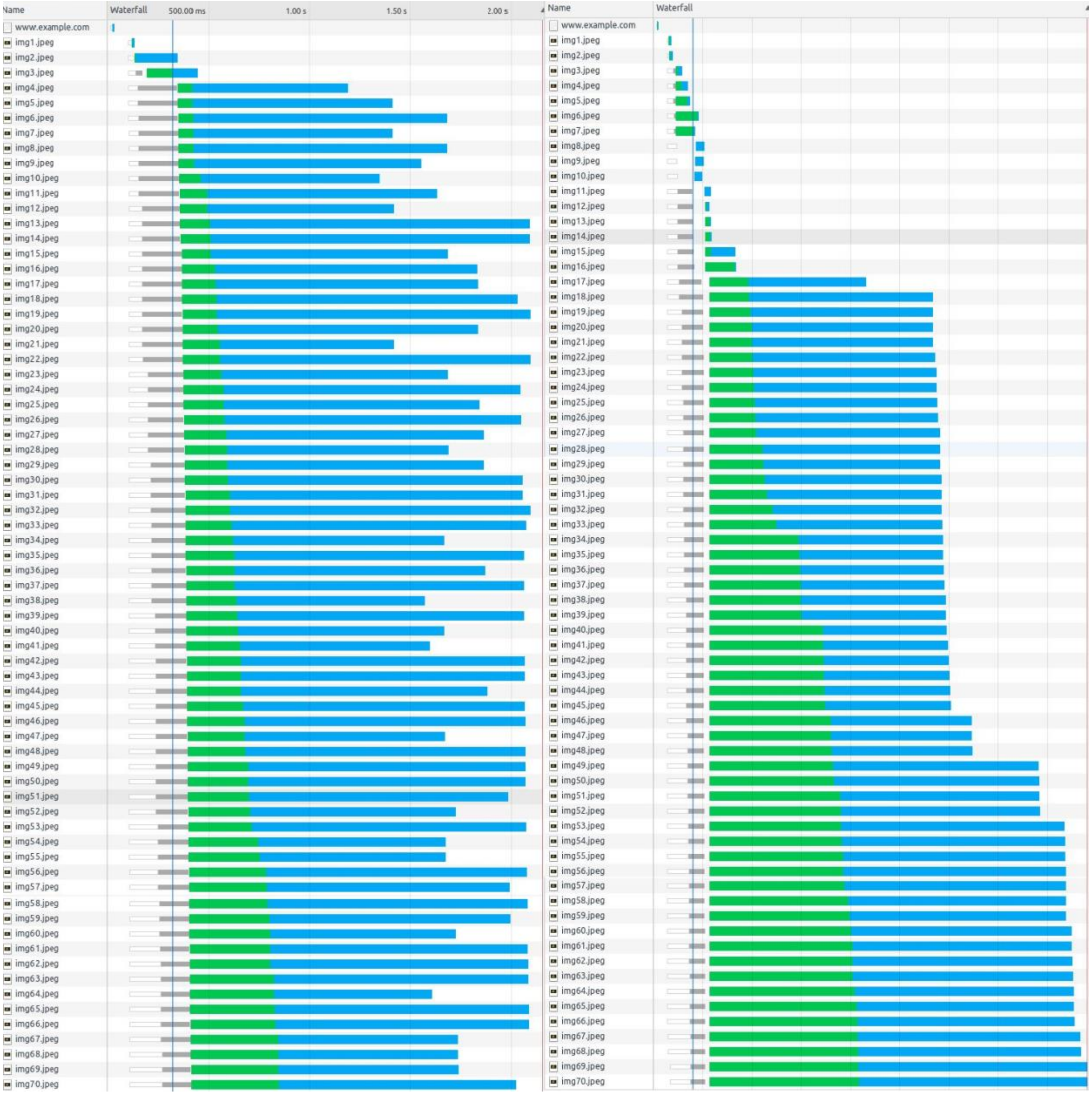


Figure 8: QUIC (Left) verses TCP (Right) Waterfall

5.2.4 Discussion

Examining the QUIC data it can be seen that there is no head of line blocking. This is seen by the fact that the time the first data byte has been received does not impact the time the last data byte is received. If the protocol did not have per stream error control, then all streams would have to wait for the preceding packet to be recovered which would add to delay. In the case of TCP one can see the delay caused by packets being dropped. The time it takes for a packet to be downloaded is dependent on the time the packet before takes to download. This can be seen in most of the packets however some of the starting packets do not follow this phenomenon, most likely due to the requesting being lost.

By enabling the applications to have control over the data streams practical improvements can be made to latency. HTTP proves to be a great example of an application that benefits from QUIC stream multiplexing. HTTP is able to have each of its get requests treated independently thus QUIC can prevent head of line blocking for each request. This does not come without drawbacks which are seen in the 1% and no loss. QUIC takes longer to download and as mentioned in previous sections QUIC uses more CPU power. Thus if the channel is stable and has a small amount of loss TCP may still be the more effective protocol. To become a faster protocol QUIC may need to trade off its rapid development for tighter integration with the OS. Once the protocol has reached maturity it would also benefit from the same OS/hardware level support TCP has.

6. Conclusions

Countless studies have proven QUIC to have a higher utility than the more traditional TCP communication stack. By being implemented largely at the application layer, QUIC pushes a lot of the protocol logic to the endpoints, which increases CPU usage compared to TCP. However, higher utility overall means that this reorganizing of computational load is not a big cause for concern. QUIC’s 0-RTT connection establishment is innovative and efficient in creating lower-latency connections between a client and server, however, is not perfectly immune to various types of DoS attacks that may in fact harm the lower-latency goal that the 0-RTT handshake aims to achieve. QUIC’s logic for stream multiplexing is efficient and the benefits are noticeable in preventing head of line blocking in comparison to TCP. In conclusion, we believe that QUIC’s implementation is innovative and thoughtful. QUIC is successful in improving some of the drawbacks of TCP and we believe we will see its use more widespread in the near future. Concerns with QUIC such as increased CPU usage at the endpoints and security vulnerabilities associated with the 0-RTT time should be addressed as we believe removing those small doubts will help make QUIC’s use more widespread, therefore making the internet faster.

Group Contributions

Chris Holland	**Austin Benoit**
• Website maintenance • Logbook entries • Report section 1-4 • Report Editing • Video recording and editing • QUIC CPU usage experiment • QUIC server/client packet analysis • Literature review	• Report section 5 • Report editing • Video recording and editing • QUIC toy server/client setup • OpenLitespeed setup and tests • Literature review • Head of line blocking tests • Multiplexing tests

Bibliography

[1] A. L. e. al, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," *SIGCOMM '17,* pp. 183-196, 2017.

[2] S. J. D. C. C. N.-R. a. A. M. Arash Molavi Kakhki, "Taking a Long Look at QUIC," London, 2017.

[3] J. Roskind, "Multiplexed Stream Transport Over UDP," 2 December 2013. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit.

[4] L. S. Henry Baxter, "A Replicable and Extensible Comparison of QUIC and TCP," 2018. [Online]. Available: https://csc466quic.wordpress.com.

[5] A. R. A. W. A. V. C. K. D. Z. F. Y. F. K. I. S. J. I. J. B. J. D. J. R. J. K. P. W. R. T. R. S. R. H. Adam Langley, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," Los Angeles, 2017.

[6] S. J. A. B. C. N.-R. Robert Lychev, "How Secure and Quick is QUIC? Provable Security and Performance Analyses," IEEE Symposium on Security and Privacy, 2015.

[7] M. T. J. Iyengar, "QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-27," 21 February 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-27#section-2. [Accessed 5 April 2020].

[8] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-27," 21 Febuary 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-27#section-2. [Accessed 5 April 2020].

[9] I. O. a. Y. C. H. de Saxcé, "Is HTTP/2 really faster than HTTP/1.1?," 2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Hong Kong, 2015.

[10] "http archive," [Online]. Available: https://httparchive.org/reports/page-weight.