SEMESTER PROJECT REPORT

IN

DATA CENTER SYSTEM LABORATORY

OVERVIEW OF THE QUIC PROTOCOL

June 8, 2018

Sutandi, I Made Sanadhi
(i.sutandi@epfl.ch)
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne
Spring Semester 2018

Contents

1 INTRODUCTION

The implementation of transport-layer protocols in the layered networking protocol model plays an important role in the utilization of Internet. Transport-layer protocols provide not only node-to-node communication but also optimal link utilization and maintenance of a receiver's buffer. However, most commonly used transport-layer protocol, TCP and UDP, are commonly implemented in Operating System (OS) kernel. This creates a strong coupling between transport-layer protocol and OS. While the release of a newer OS version tends to be slow, the current Internet-scale communication demands for a new protocol that can enhance throughput and decrease end-to-end latency. To achieve so, Google introduced QUIC, a transport-layer protocol that is implemented in user-space [1].

Although Google initiated QUIC for global-scale HTTP communication, i.e. web applications, it is interesting to evaluate QUIC in data center environment as some of its features may be applicable to the data center communication needs. It is important to deeply understand the structures and natures of QUIC, though, before applying it into certain setups. Therefore, in this project, we study the structure, semantics, and features of the QUIC protocol.

2 BACKGROUND

2.1 Transmission Control Protocol (TCP)

TCP is the most widely used transport-layer protocol because TCP offers reliable data delivery with acknowledgement packets (ACK/SACK) and implements connection-level flow control and congestion control. First, flow control in TCP prevents receiver-buffer overflow by adapting the rate of TCP source to speed of the receiver. Then, the usage of congestion control in TCP ensures every TCP connection to highly utilize the bandwidth of the Internet links against other competing flows in order to avoid congestion collapse. Unlike UDP, TCP maintains fair sharing of Internet links for multiple flows. These features of TCP make every established TCP connection to be able to adjust the source's sending rate to the state of both receiver and network.

On the other hand, aside from its benefits, TCP has several shortcomings:

1. **1-RTT Connection Establishment**: Since TCP is a reliable transport protocol, it always requires acknowledgement for each segment of data it tries to send to indicate that the corresponding segment is lost or not. As this requirement is needed even for connection setup, thus TCP connection needs at least 1 round-trip delay (RTT) for its connection

establishment with three-way handshake mechanism.

2. **Head-of-Line (HOL) Blocking**: HOL is a phenomenon that occurs when the transport-layer protocol offers ordered or partial-ordered service. In TCP, which offers ordered reliable service, if a packet is loss in the Internet, the subsequent messages are suspended until TCP manages to recover or retransmit the lost packet.

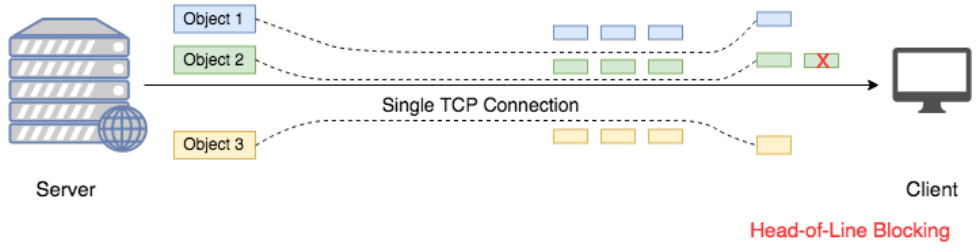

Figure 1: Example of HOL in TCP

3. **Additional 1-RTT for TLS**: TCP does not support packet encryption natively, so there is a need to add another security layer, i.e. Transport Layer Security (TLS), to support secure communication over TCP. Although some extensions, e.g. TCP fast open [2], attempt to reduce additional RTTs for initial cryptographic handshakes, operation of TLS at least brings us another 1-RTT to the overall phase for connection establishment.

2.2 User Datagram Protocol (UDP)

In contrast with TCP, UDP is a message oriented transport-layer protocol that performs *fire-and-forget* principle for sending packets through the network. Since UDP does not guarantee reliable delivery, applications must handle data recovery themselves if some messages are loss during transmission period. Moreover, UDP also does not implement any congestion control thus packet transmission over UDP will ignore network condition and may act as a greedy source. However, UDP imposes less overhead compared to TCP because it does not perform any congestion control, error checking and packet reordering. Consequently, for time-sensitive applications, it might be desirable to make use of UDP since UDP communication does not wait for recovery and retransmission for packets which are loss in the network.

2.3 Hyper Text Transfer Protocol (HTTP) over TCP

HTTP is an application-layer protocol and the underlying protocol for web applications. HTTP follows a client-server computing model and its definitions infer the needs for a basic reliable transport-layer protocol. Hence, TCP is generally used for HTTP communication. Despite all the benefits HTTP obtains from using TCP; TCP has failed to cope with evolution of HTTP protocol. The HTTP's evolution and its issue with regard to integration with TCP is described as follows:

- **HTTP/1.0 & HTTP/1.1**
  In HTTP/1.0, every request for an object in the same server compels both client and server to open a new TCP connection. This implies that, for every HTTP request, the client and server must open a number of distinct TCP connections. Whereas HTTP/1.1 permits connection reuse for different objects to reduce the number of open connections [3], yet both client and server may maintain several TCP connections only for a single HTTP request.

- **HTTP/2**
  In order to achieve reduced latency and higher throughput for HTTP communication, HTTP/2.0 proposes the idea of multiplexing different objects into a single TCP connection between client and server [4]. In this way, there is no need to preserve many unnecessary open connections and thus needless RTTs for several connection establishments are avoided, leading to enhanced throughput and decrease of latency. However, HTTP/2 suffers from TCP's sequential delivery mechanism [5], which produces a "latency tax" because of HOL that occurs in TCP.

- **HTTPS**
  Secure HTTP (HTTPS) is a secure HTTP communication over TCP and TLS stack. As described in section 2.1, this integration for providing secure HTTP communication involves at least 2-RTT: (1) 1-RTT for connection establishment, (2) 1-RTT for cryptographic handshake.

3 QUICK UDP INTERNET CONNECTION (QUIC)

QUIC is delivered by Google as an encrypted, multiplexed, and low-latency application-layer transport protocol over UDP for improved secured HTTP (HTTPS) performance [1]. Although

initially QUIC is aimed to address issues on HTTP/2 over TCP connection, it has already been used for other applications that involve reliable delivery, such as video streaming.

It is important to underline that QUIC is designed for Internet-scale communications via Wide Area Network (WAN). QUIC has positive performance impacts compared to TCP for networks with higher average RTT and retransmission rate [1], which are a typical condition for WAN communication. Whereas on networks with low delay and low loss rate —which usually happens on short distance communication, e.g. inside a LAN—, QUIC is expected to gain small positive performance feedback to negative performance impact [1].

3.1 Motivation

In general, QUIC focuses on: (1) reducing handshake delay (RTT) for connection establishment, (2) avoiding HOL issue to arise, and (3) providing secure (encrypted) communication. As described in Section 2, these challenges are encountered in traditional HTTP/2 on TCP and TLS stack. While it is possible to upgrade TCP in order to overcome the concerns, it is more feasible to implement the new transport protocol atop of UDP, considering the following points:

- **OS Coupling**
  TCP is regularly implemented in the OS kernel. Therefore, even a slight change of TCP's features can require the clients to upgrade their OS version. Since the OS upgrades tend to be relatively slow [6], the coupling between TCP implementation and OS has created a bottleneck for deploying TCP changes.

- **Support from Middleboxes**
  Middleboxes, e.g. firewall and network address translation (NAT), are main control points in the Internet and they ought to inspect, filter, and rewrite packets for packet forwarding. Aside from TCP and UDP, middleboxes may block or even drop packets that are sent using uncommon or unsupported transport-layer protocol. Adding support for a new protocol or upgrading OS with a newer version of TCP may necessitate global-scale upgrades of Internet's middleboxes. Consequently, it takes an uncertain period [7] to see the complete effects of either state-of-the-art transport-layer protocol or newer TCP deployment.

Based on these considerations, QUIC is implemented in the application-layer (user-space) and on top of UDP in order to not only avoid coupling with OS but also to make faster deployment, testing, and iterations.

3.2 Design

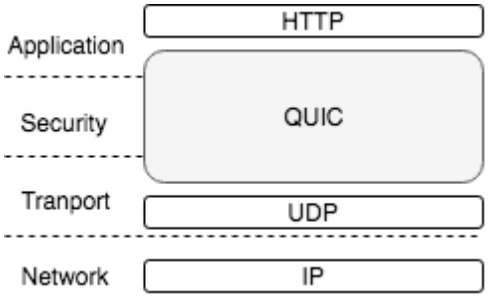

Figure 2: QUIC Stack

QUIC implements not only transport-layer but also security-layer and application-layer functionalities. In general QUIC has two different levels of abstraction, which incorporate:

- **QUIC Connection**
  UDP connection underlies a QUIC connection and the user-space operation of QUIC allows the connection to provide reliable delivery, relevant to TCP. To avoid congestion on the Internet links, QUIC does not rely on a specific congestion control algorithm but instead has "pluggable" congestion control, which supports widely-used algorithm such as Cubic and BBR. In addition, QUIC enforces flow control to handle buffer overflow on receiver's side.

- **QUIC Streams**
  QUIC streams are a lightweight abstraction that establish bidirectional reliable delivery for application data. Streams can be used to frame application data with an arbitrary size and every stream owns a stream ID as an unique identifier. The stream ID is important since it identifies the initiator of the stream. QUIC maps objects into streams and multiplexes these streams over a single connection. A single stream may emulate one single TCP connection and a single QUIC connection may then represent N TCP connection(s). The peer may receive streams out of order but QUIC guarantees in-order delivery within a stream. Furthermore, QUIC prohibits a potential HOL —a stream with data that is slowly delivered to application— by imposing stream-level flow control, which limits the buffer that a single stream can consume.

3.3 Features

QUIC is a transport protocol that is delivered to deal with problems in TCP, TLS and HTTP/2 designs. Thus, QUIC adopts and optimizes implementation of secure HTTP/2 over TCP and TLS, which includes:

- **0-RTT Connection Establishment**
  On a successful handshake to a server, a client caches the information of the origin and subsequently use this information to establish encrypted communication without additional RTT, achieving 0-RTT connection establishment. In the worst case —during every first communication to a server and when the cached information is expired— QUIC connection invokes 1-RTT at maximum. In both 0-RTT and 1-RTT cases, clients are allowed to send the application data in parallel with client hello (CHLO) message.

- **Diminishing of Head-of-Line Blocking**
  Unlike TCP that enforces sequential delivery, QUIC introduces streams abstraction to multiplex several objects into multiple streams within a single UDP connection without a strict in-order delivery. A packet loss only affects the operation of streams that it carries. As a result, overall stream deliveries over QUIC will never be blocked due to a packet loss.

- **Full Encryption**
  QUIC packets are mainly authenticated and mostly encrypted. QUIC gives a native support for secure traffic by integrating the cryptographic functions into its core implementation. Therefore, QUIC is not designed to allow the user to disable packets authentication. During connection establishment, every QUIC connection owns two kind of keys:

  – Initial keys: Used in both client and server after a client send a complete CHLO and before the handshake is successful.

  – Forward-secure keys: Used in both client and server after the handshake is successful, i.e. the client receives a server hello (SHLO).

4 QUIC IMPLEMENTATION

Although QUIC has already been used for production in Google, the available source code is not designed for production [8, 9]. The protocol itself is still on its way of being standardized.

Currently, there are two different standardizations of QUIC, namely Google QUIC and IETF QUIC.

4.1 Google QUIC (GQUIC)

GQUIC is bundled up with the Chromium project and it basically defines the fundamentals of QUIC protocol. The Chromium project includes standalone QUIC client and server, which are being called as **GQUIC "Toy" Client and Server**. However, there is no clear documentation or explanation about what Google is currently using in the production and it is stated that the GQUIC Toy Client and Server are meant for integration testing only [8]. There are certain GQUIC semantics that differ from the equivalent IETF implementation of QUIC:

- **Stream ID Numbering**: In order to avoid streams having the same IDs, especially when they can be created in both client and server, servers initiate streams with even stream IDs and the clients initiate streams with odd streams IDs.
- **Dedicated streams**: In GQUIC implementation, stream 1 is assigned only for cryptographic function. In addition, stream 3 is dedicated for transmitting compressed HTTP/2 headers for all other streams [10]. This is necessary to perform reliable in-order delivery and processing of headers. GQUIC currently compresses HTTP headers via HTTP/2 HPACK header compression (RFC7541) [11] on a stream 3, which imposes head-of-line blocking for header frames only since HTTP/2 header blocks must be decompressed in the order they were compressed.
- **Cryptographic Handshake**: GQUIC's cryptographic handshake adopts home-grown cryptographic function for TLS, namely QUIC's cryptographic protocol.
- **Congestion control**: By default, GQUIC's congestion control algorithm is CUBIC.

4.2 IETF QUIC

The proposal for QUIC has been submitted to IETF for being officially standardized by IETF QUIC Working Group (QUICWG) [12] and it is expected to bring more performance for QUIC. There are several work in progress implementations of IETF QUIC but none of them is ready for usage in production [9]. Although the core design of QUIC remain unchanged, there are several adjustments [13, 14, 15] in IETF QUIC that are worth to mention:

- **HelloRetry instead of Complete CHLO**: If the CHLO fails, the server will request a valid cookie extension and an address validation token via HelloRetryRequest. Thus, the second CHLO is much smaller in size because it does not resend the data in first CHLO.
- **Stream ID Numbering**: The IETF QUIC has the opposite rule for streams identifiers from GQUIC. Clients initiate even-numbered streams and servers initiate odd-numbered streams.
- **Dedicated Streams**: IETF QUIC dedicates stream 0 only as a dedicated stream for cryptographic exchange. IETF QUIC exploits QPACK [16], which is a variation of HPACK being used in GQUIC. QPACK is an efficient compression format for HTTP headers fields and it is more robust to HOL compared to HPACK. Thus, HTTP over IETF QUIC has no need to use any dedicated streams for HTTP/2 headers mapping.
- **Cryptographic Handshake**: TLS 1.3, which is also inspired by GQUIC's cryptographic protocol, will be wielded for QUIC's cryptographic handshake.
- **Congestion control**: IETF will utilize TCP NewReno as a basis for QUIC's congestion control algorithm.

5 EXPERIMENTAL SETUP

5.1 Software Setup

In this project, we try and observe both GQUIC and IETF QUIC implementations since both are QUIC but with some different semantics. We conduct the experiments on a single-machine setup. The machine runs OSX Sierra 10.12.6 with kernel version 16.7.0. We also repeat and verify the experiments on Ubuntu Xenial 16.04 VM with kernel version 4.4.0-87-generic. The machine has 1.6 GHz Intel Core i5-5250U dual-core processor and 8GB of DDR3 memory, with git and Python 2.7.13 already installed. We make use the following open-source codes [8, 17, 18, 19]:

1. **depot_tools**: A package of scripts to manage Chromium's code from Google repository. This package contain all necessary scripts to modify Chromium's code, such as Ninja (build tool), gn (meta-build system), and gclient (git client for Google repository).
2. **Chromium project**: The open-source project that consists of two big projects: (1) Google Chrome browser and (2) Google Chrome OS. Since there is no official and

up-to-date project for standalone QUIC implementation, we fetch this project to test and observe GQUIC implementation.

3. **ngtcp2**: ngtcp2 is an ongoing and an early implementation of IETF QUIC protocol that is focused mainly on QUIC Transport. We use this implementation since it has already provided some basic documentations and an application layer mapping for HTTP communication. This project's QUIC client and QUIC server have a dependency for OpenSSL's TLS 1.3 for its cryptographic back end.

4. **OpenSSL**: OpenSSL aims to deliver Transport Layer Security (TLS) protocols and general purpose cryptographic libraries. We use TLS 1.3 from OpenSSL since it is a requirement of ngtcp2's implementation.

5.2 Code Modification

To better understand QUIC's nature. We make some changes to the GQUIC Toy Server's code and Toy Client's code [20] as follows:

1. **Increase Logging Verbosity**: As GQUIC has not intended to support decryption, it is hard to analyze what actually happens on wire, i.e. packets exchange. Hence, since we conduct all experiments on a single-machine setup, we enhance the level of logging verbosity in both the client and the server code, e.g. by printing hexadecimal dump for streams data. Then, we observe the captured QUIC packets and match the contents with what written on the logs files.

2. **Leverage QUIC for Persistent Connection**: In contrast with ngtcp2, GQUIC Toy Client implements no functionality for persistent QUIC connection, which is an idea to reuse a single QUIC connection for multiple HTTP request-response pairs. Accordingly, we modify the GQUIC Toy Client so that we can have an optional "switch" for persistent connection. We add a simple *while* statement to loop over and to repeat the request until the desired number of requests is reached.

6 RESULTS

Throughout our experiment, we test both GQUIC and ngtcp's QUIC protocol for HTTPS communication only. The client sends a single HTTP request packet and the server returns HTTP response packets containing a web page. We execute the same HTTP GET request in all

our scenarios. The client always requests for the identical HTML page of www.example.org and the server always replies with the exact same page. The size of the page is 1593 bytes (1270 bytes excluding the headers) and it is not fit in a single packet. Therefore, both implementations split the response page into two packets. GQUIC Toy server splits the headers and the actual HTML body into two different messages meanwhile ngtcp2 QUIC server treats the headers as a part of the HTML body and creates the second packet after the first segment's size reach the MTU. We then observe the packet exchanges for overall connection establishment and data transfer.

6.1 GQUIC Toy Server and Client

In this section, we want to identify the behaviors of GQUIC and match our findings into the given fundamentals of QUIC. We use Wireshark [21] to capture the QUIC traffic and we make our analysis by examining the logs files of both client and server. We also match the details with the captured packets in Wireshark.

6.1.1 Default Packet Exchange

To begin with, as we need to understand the characteristics of QUIC Toy Server and Client code, we attempt to see the default packets that are exchanged and how they are sequenced by default. We compile the QUIC Toy Server and Client with Ninja [22] without making any modification to the main programs (except the verbose logging as we explained in section 5.2). We spot one UDP connection (one connection ID) and three streams with IDs: 1 (cryptographic), 3 (HTTP/2 headers), 5 (request/response body). To our surprise, as presented in Figure 3, we see different behaviors than we expected from the theory:

1. 3-RTT connection establishment instead of 1-RTT only, for the first connection.
2. Acknowledgement (ACK) packets during the cryptographic handshake.

We then categorize two types of packet for the rest of our discussion onwards, including discussion for ngtcp2 QUIC in section 6.2. First, the first phase packets, which are necessary to be exchanged until final keys (forward-secure keys) are employed in both client and server, are the **connection establishment** packets. Second, we consider the packets after the cryptographic handshake is finished as **data transfer** packets. However, the first phase packets that carry actual requests or responses belong to the second type as well, despite they are encrypted with initial keys.

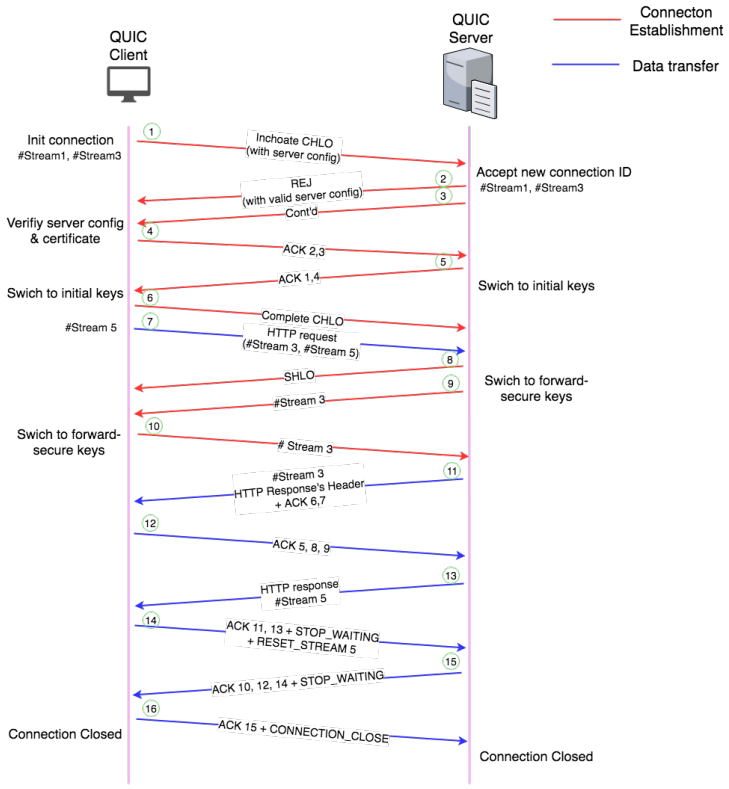

Figure 3: Default Packet Exchange of GQUIC Toy Server and Client

After several trials, we observe that some random delays in both client and server, which is mostly caused by the intensive logging, produce indeterministic order of packets after the connection establishment. The sequence is not necessarily strict to follow the order given in Figure 3. In fact, the server is not waiting for 10th packet before sending the 11th packet, implying the acquired result as 2-RTT connection establishment.

Then, when we try to disable the logging, the sequence of packets that are exchanged aligns more with theory, as portrayed in Figure 4. However, we still have more packets (ACK packets 4,5 in Figure 3) than it is supposed to be, achieving 2-RTT instead of 1-RTT. The client does not immediately send a complete CHLO message after receiving a rejection (REJ) message from the server and it sends an acknowledgement (ACK) packet instead.

6.1.2 Packet Exchange with Configuration Tuning

Next, we prove that QUIC Toy Server and Client are capable of doing 1-RTT connection establishment and that we can produce the sequence which equal to QUIC's standard. We discover that by default the client performs a verification for server certificate in order to

ensure the credibility of the server, making the client sends an extra ACK packet. Fortunately, this is an optional operation. Thus, we disable this in the client side, and we increase the initial maximum transmission unit (MTU) so that the server does not split its configuration to two REJ packets (packet 2,3 in Figure 3). Last but not least, we disable the logging in both sides. We successfully reproduce 1-RTT connection establishment as illustrated in Figure 4.

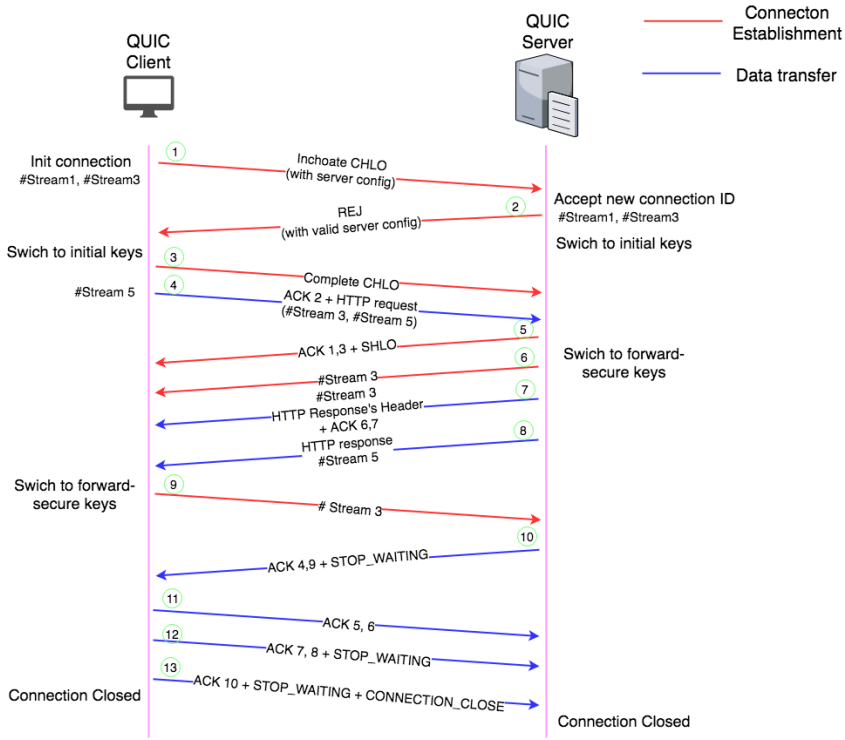

Figure 4: Packet Exchange of GQUIC Toy Server and Client after Configuration Tuning

Aside from the reduced number of packets and the different order of messages, we still have an equal number of connections and streams (with the same streams' IDs) as previous experiment in section 6.1.2.

6.1.3 0-RTT Packet Exchange

We test the 0-RTT connection establishment to study how one of QUIC's main benefits is actually realized in the wire. Since GQUIC Toy client does not cache any information about the origin, we need to either modify the QUIC Toy Client to store the necessary server information for the cryptographic handshake or use another QUIC client that can save information about the origin. We decide to take the latter option, and we use Google Chrome browser as a QUIC client to validate the 0-RTT capability of QUIC.

We use the Chrome browser to make the HTTP request without changing its default configuration for QUIC communication. Also, we enable again the intensive logging in the server for analyzing the streams' IDs and contents. Finally, we are able to prove the 0-RTT connection establishment of QUIC connection to the same origin. Additionally, for the same reason as we have explained in section 6.1.1, we want to underline that packets 5, 6, 7, 8 are not necessarily exchanged with the given order as shown in the Figure 5.

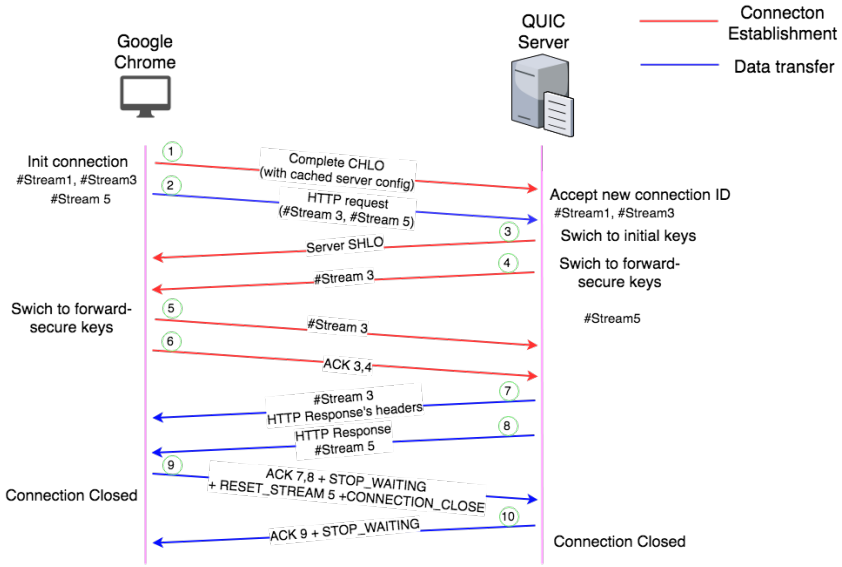

Figure 5: GQUIC Toy Server 0-RTT Packet Exchange with Chrome Browser

6.1.4 Persistent Connection Packet Exchange

We conduct an experiment for QUIC persistent connection in order to comprehensibly comprehend the HTTP/2 requests/responses to stream mapping. We want to know when the streams are created and how long they live in one QUIC connection. After modification as mentioned in section 5.2, we carry out multiple requests over one QUIC connection before closing it.

We spot that stream 3 is always present alongside the odd-numbered data stream frame on every request that client initiates. Likewise, the server always replies with stream 3 containing the response headers and stream with the corresponding stream ID containing the response body. We also spot that a stream is immediately closed (**with an exception for stream 3**) after the client has successfully received the response. This implies that a specific stream ID cannot be reused even if the QUIC connection itself is persistent. As we increase the

number of requests in one connection, we observe the presence of more and more streams with increasing stream IDs: 5, 7, 9, 13, and so on.

6.2 NGTCP2 (IETF QUIC)

In this section, we use a patched Wireshark 2.5.2 (v2.5.2rc0-171-g9dde6d4b) so we can directly decrypt and observe the ngtcp2's QUIC traffic in Wireshark. The latest version of OpenSSL library has also supported *keylog* callbacks in order to retrieve the secrets key of QUIC connection. The keys are then exported to a file, and they are passed to Wireshark to enable payload decryption. The observation that we perform in this part is pretty straightforward since we are able to clearly monitor the contents of every QUIC packet in Wireshark.

6.2.1 Default Packet Exchange

First, we verify that the ngtcp2 implementation, which is based on IETF standard of QUIC, implements the 1-RTT connection establishment for the first connection to every origin. In ngtcp2 code, the client program can accomplish the request in two ways: (1) an interactive mode that requires the user to type the request path in the terminal before timeout, (2) a non-interactive mode that allows the client program to obtain the request path on a specified file and immediately send the HTTP request.

We then straightforwardly compile the ngtcp2' client and server code and perform non-interactive HTTP request. As depicted in Figure 6, the QUIC client and server program exhibit 1-RTT capacity during connection establishment. We identify no second CHLO since the client is sending HelloRetry (packet 5) alongside the signal that the handshake is finished. In addition to the actual response, the server program is hard-coded to send a "Hello World!" message (packet 8) via stream 3 (low bits 0x3), which is server-initiated and unidirectional stream according to IETF definition [13]. In case of the client using an interactive HTTP request, the hard-coded message can be used to indicate that 1-RTT connection establishment is successful. The client then proceeds with the HTTP request on an even-numbered bidirectional stream whose stream ID is four (low bits 0x0), following the IETF definition [13]. The server then gives responses in stream with equivalent stream ID with the stream that client sent to carry the request.

Identical to GQUIC Toy Server and Client, some random delays may reorder the packets sequence. For example, it may occur that the 7th and the 8th packets are sent by the server before the client send the HTTP request in the 6th packet. If we do not count the hard-coded stream 3, overall we will see two streams are exchanged inside packets over a single

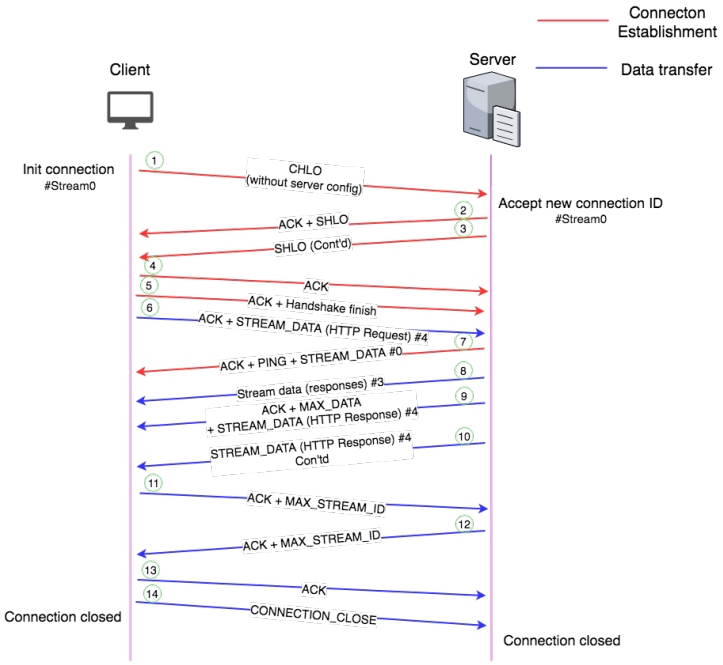

Figure 6: Default Packet Exchange of NGTCP2 QUIC

UDP connection (one connection ID). The stream IDs are: 0 (cryptographic) and 4 (HTTP request/response).

6.2.2 Resumption and 0-RTT Packet Exchange

Subsequently, we evaluate the 0-RTT connection establishment to the same origin of ngtcp2 QUIC code. Despite it is an early implementation of IETF QUIC, ngtcp2 code has managed to uphold the 0-RTT features correctly. We confirm that the ngtcp's QUIC client program supports resumption and 0-RTT by caching a session ticket and the transport parameters from the QUIC server. To achieve so, we need to tell the client which file accommodates the cached information so that the client can load them. Next, the client sends a CHLO with cached server configuration and session ticket in parallel with the request. The server then instantaneously replies with SHLO and the response for the corresponding request. While in previous case we have two SHLO packets since the data in SHLO is relatively big (exceeding 1500 bytes), in 0-RTT scenario the server does not send its certificate again so that we have a single and smaller SHLO packet (around 400 bytes). The numbers of connection and streams remain unchanged with previous experiment in section 6.2.1.

6.2.3 Persistent Connection Packet Exchange

Finally, we clarify the persistent connection of ngtcp2's QUIC in order to scrutinize the application to streams mapping. This is necessary in order to distinguish how streams mapping differs between GQUIC code and IETF QUIC implementation. Fortunately, ngtcp2's QUIC client has already owned feature of persistent connection by sending the same request repeatedly over different streams within a single connection.

We observe that again no stream is allowed to be reused. The lifetime of streams is short in which it is no longer needed once it completes sending the application data. This is well founded for both implementation of GQUIC and IETF QUIC.

We spot the number of streams with low bits 0x0 and this number is indeed equal to the total request that we specify. The sequence of streams ID that we notice is: 4, 8, 12, 16, and so on. The obvious difference that we mark is the streams in IETF QUIC are fully independent between each other, meaning that there is no dependency with other streams. Unlike the GQUIC Toy and Server, which dedicate stream 3 for HTTP/2 headers mapping thus impose dependency for other streams, a single stream of ngtcp2 QUIC represents exactly one independent request-response pair for an object.

7 CONCLUSION

In this project, we study the underlying concept, fundamental structure and features of QUIC. We have seen how QUIC can improve and address the shortcomings of secure HTTP/2 communications over TCP and TLS by utilizing 0-RTT connection establishment and streams multiplexing atop of UDP. Hence, QUIC will be able to replace TCP and TLS stack for secure HTTP communications. However, it is important to know that the available QUIC implementations are currently immature and they are rapidly changing. Despite of Google implementation of QUIC (GQUIC) having different semantics with IETF QUIC, IETF implementation of QUIC will become the de facto standard for the QUIC protocol.

8 FUTURE WORK

Due to a limited period of this semester project, many developments, tests, and experiments have been left. Therefore, the following ideas for the future work are interesting to explore:

1. Develop a sophisticated and easy-to-use QUIC application programming interface (API) for another programming languages, such as Java, Python, Scala, and so on.

2. Test and evaluate QUIC implementation in custom set-ups/environments, e.g. testing QUIC in data center environment, and measure QUIC performance by taking into account: (1) network latency, (2) throughput, and (3) node utilization (CPU, memory).

ACKNOWLEDGEMENTS

This project was conducted in fulfillment for the completion of the author's master study at EPFL. The author thanks Marios Kogias and Edouard Bugnion, who wholeheartedly supervise this project.

9 REFERENCES

[1] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic tranport protocol: Design and internet-scale deployment," *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, New York, NY, USA, pp. 183–196*, 2017.

[2] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "Rfc 7413: Tcp fast open. internet engineering task force (ietf)," *Internet Engineering Task Force (IETF)*, 2016.

[3] R. Fielding and J. Reschke, "Rfc 7230: Hypertext transfer protocol (http/1.1): Message syntax and routing," *Internet Engineering Task Force (IETF)*, 2014.

[4] M. Belshe, R. Peon, and M. Thomson, "Rfc 7540: Hypertext transfer protocol version 2 (http/2)," *Internet Engineering Task Force (IETF)*, 2015.

[5] D. Clark and D. Tennenhouse, "Architectural considerations for a new generation of protocols," *ACM SIGCOMM*, 1990.

[6] M. Honda, F. Huici, C. Raiciu, J. Araújo, and L. Rizzo, "Rekindling network protocol innovation with user-level stacks," *ACM Computer Communication Review*, 2014.

[7] J. Rosenberg, "Udp and tcp as the new waist of the internet hourglass. draft-rosenberg-internet-waist-hourglass-00," *Internet Engineering Task Force (IETF)*, 2008.

[8] Playing with quic. Accessed: 04- Jun- 2018. [Online]. Available: https://www.chromium.org/quic/playing-with-quic

[9] Quic implementations. Accessed: 04- Jun- 2018. [Online]. Available: https://github.com/quicwg/base-drafts/wiki/Implementations

[10] R. Shade and M. Warres, "Http/2 semantics using the quic transport protocol. draft-shade-quic-http2-mapping-00," *Internet Engineering Task Force (IETF)*, 2016.

[11] R. Peon and H.Ruellan, "Hpack: Header compression for http/2," *Internet Engineering Task Force (IETF)*, 2015.

[12] Quic ietf working group. Accessed: 04- Jun- 2018. [Online]. Available: https://quicwg.github.io/

[13] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport. draft-ietf-quic-transport-10," *Internet Engineering Task Force (IETF)*, 2018.

[14] J. Iyengar and I. Swett, "Quic loss detection and congestion control. draft-ietf-quic-recovery-10," *Internet Engineering Task Force (IETF)*, 2018.

[15] M. Thomson and S. Turner, "Using transport layer security (tls) to secure quic. draft-ietf-quic-tls-11," *Internet Engineering Task Force (IETF)*, 2018.

[16] C. Krasic, M. Bishop, and A. Frindell, "Qpack: Header compression for http over quic," *QUIC IETF Working Group (QUICWG)*, 2018.

[17] Checking out and building chromium on linux. Accessed: 04- Jun- 2018. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/docs/linux_build_instructions.md

[18] ngtcp2. Accessed: 04- Jun- 2018. [Online]. Available: https://github.com/ngtcp2/ngtcp2

[19] Openssl: Tls/ssl and crypto library. Accessed: 04- Jun- 2018. [Online]. Available: https://github.com/openssl/openssl

[20] Quic overview. Accessed: 01- Jun- 2018. [Online]. Available: https://github.com/sanadhis/quic-overview

[21] Wireshark. Accessed: 07- Jun- 2018. [Online]. Available: https://www.wireshark.org/

[22] The ninja build system. Accessed: 07- Jun- 2018. [Online]. Available: https://ninja-build.org/manual.html