DEPARTMENT OF INFORMATICS

Technische Universität München

MASTER’S THESIS IN INFORMATICS

AUTOMATED IETF QUIC INTEROPERABILITY MATRIX

Angelin Rashmi Antony Rajan

DEPARTMENT OF INFORMATICS

Technische Universität München

MASTER'S THESIS IN INFORMATICS

AUTOMATED IETF QUIC INTEROPERABILITY MATRIX

Author:	Angelin Rashmi Antony Rajan
Supervisor:	Prof. Dr. -Ing Jörg Ott
Advisor:	M.Sc. Teemu Kärkkäinen
Submission Date:	August 13, 2018

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

August 13, 2018 Angelin Rashmi Antony Rajan

Abstract

With the rapid growth of web based application and mobile revolution in recent years the main deciding factor for user experience is low latency. Quick UDP Internet Connections(QUIC) which runs on top of User Datagram Protocol(UDP) is a new transport protocol developed to overcome delay and also provide security to the data traffic. It is currently under development by IETF working group and more than 16 groups are working on the implementation of QUIC on different languages. Since QUIC runs on user space it is easy to deploy and each group can develop their own implementation according to the QUIC's specification.

Often, these implemenatations need to interoperate with each other in the real world scenario. So, the groups meet periodically to carry interoprability testing. This process is time consuming because there are lot of implementations and to conclude that two implementations interoperate with each other the entire functionality of the protocol must be tested for interoperability. This thesis deals with analyzing the various implementation of IETF QUIC and designing a test framework to automate the interoperability tests between them. Also, the specification is continuously evoloving which makes the design of an automated framework a challenge.

The framework aims to reduce the testing time by automating the process of building the project, running the tests, analyzing the logs, validating the test results and finally displays the result in an interoperability test matrix. The benefits the developers of the protocol to see the results of the interoperability test at a much early time and the framework will also developer to understand the failure of a test scenario by providing them with the necessary log files for analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the aim to reduce latency in connection establishment and improve perfomance of HTTPS traffic, Google developed the experimental transport protocol QUIC in 2013. QUIC enables rapid deployment as it runs on user space on top of UDP instead of a traditional kernel space transport protocol. It was deployed on thousands of google servers which handles more than billion request in a day and was made to serve the clients like Chrome browser and YouTube app. It was found that the average google search response time was down by 8.0 % and reduced rebuffer rates in youTube by 18.0 % in desktops. Since QUIC was found to be safe and performant, as of 2017 QUIC was used for all users of Chrome and Android YouTube app [4]. In 2016 a working group was formed to standardize and modularize QUIC protocol. Currently, the protocol is in its 13 version of draft. Whenever there is a release in new standard or product, there is a need to perform interoperability testing. There are at least 16 groups working on the implementation of QUIC in various languages and are tested against each other for intereoperability.

These QUIC interop meetings happen on a predefined dates in an interval of three to four months and the tests are run manually to check if two implementations interoperte with each other. This is quite time consuming process as there are many implementations and many test cases to run and validate. It is estimated to take at least a day or two to run all the interoperability tests between implementations and publish the results. we propose to design an automated framework to run these interoperability tests regularly so that the time in conducting interoperability meeting is saved and the developers see much earlier if some of the test fails.

1.1 Goals of the thesis

The primary goal of this thesis is to design and implement an automated testing framework for QUIC's interoperability testing. This reduces the time taken for interoperability testing between different implementations. Often of the main challenge in a network protocol's interoperatbilty testing is to consider that there will be many implementations from different groups. Also, the QUIC IETF specification is still in progress and so it makes developing a test framework even more difficult. Furthermore there are no standard logging mechanism across different implementation. The framework is responsible for interpreting the logs and validating the test cases for each implementation. The framework should handle all such scenarios and scale up to any new implementations or new test scenarios without much changes.

In this thesis we propose three framework designs according to the availabilty of resource for setting up the test bed. The feasibility of the framework is demostrated by implementing and evaluating one of the design which is cost effective and can be set in much less time than the other setup. The evaluation of the design is done by using the implementation of the framework to run test against the public QUIC servers whose source code we don't have access to and by running tests against the open source QUIC servers. The evaluation can also show us how the interoperability of an implementation goes over time. The results of the interoperabilty test is displayed in a matrix and we also a provide a way in to assess the failure of a test case by providing the corresponding server and client logs.

1.2 Outline of the thesis

In Chapter 2, we see how TLS 1.3 works in comparison with TLS 1.2, the different modes in which TLS 1.3 can be operated. This helps in understanding how the connection establishment latency is reduced in QUIC with the use of TLS 1.3. We then talk about the various feautures offered by QUIC protocol and the packet types invloved in QUIC. This helps us in understanding how the protocol work. The next section focusses on the automated testing tools, objectives, steps for interoperability testing and problems faced in interoperabilty testing.

In Chapter 3, we present about the challenges in designing an Interoperability Testing Framework for QUIC protocol, conceptual model of the environment. We then talk about three proposed design setup for interoperability testing which are light weight, virtualized system and physical system. In the later parts we talk about the various test scenarios which covers the functionality of QUIC protocol. The flow of packets for the each test is explained clearly using a sequence diagram and some explanation for each packet and its contents.

In chapter 4, we see how the implementation for one of the design is done using Jenkins as a continuous integration tool, working of a log parser using state transition matrix and pandas library used to represent the test results in a data structure and apply stlying for the visual representation of the interoperability test matrix. In Chapter 5, we see the evaluation of it. In the end, Chapter 6 summarizes the thesis and presents some future work.

Chapter 2

Background

In this chapter, we will look at TLS 1.3 which is used to provide cryptographic security and message integrity for QUIC protocol. The related modes which help in achieving quick connection establishment is discussed. Later, we discuss the working of QUIC and the advantages of using it. We will also discuss the various implementations which currently participate in IETF QUIC interop testing. In addition, we will also see why we need inter-operability testing, objectives, and problems in interoperability testing.

Transmission Control Protocol and the Internet Protocol (TCP/IP) forms the backbone of the entire internet. TCP is a transport layer protocol and is responsible for providing host-to-host communication. The host can be the same machine or in the local network or remote network. TCP also provides connection-oriented transmission, flow-control, and reliable transmission. In order to get further benefits like security, TCP is used with other application protocols. The most commonly used is TLS over TCP to obtain data integrity and confidentiality. QUIC tries to get all the best practices of TCP, TLS and HTTP/2.

In the next section, we will see about TLS 1.3 as QUIC relies on them for authentication and negotiation of security parameters. QUIC and TLS 1.3 are co-dependent as QUIC uses TLS handshake data in CRYPTO frames and TLS uses ordered delivery and reliability from QUIC.

2.1 TLS 1.3

The goal of any transport layer security protocol is to establish a secure channel through which two parties can communicate for example a web browser as a client and a web server. The underlying transport protocol should provide reliability, data streams and ordered delivery.

TLS 1.2 was used for the past decade as a cryptographic protocol to provide security for the transport layer streams [5]. In order to improve the speed in connection establishment and mitigate the vulnerabilities caused by some cryptographic and message authentication algorithm in TLS1.2, TLS 1.3 was developed by the IETF group and standardized in March 2018. TLS 1.3 is not backward compatible with TLS 1.2 but the versioning mechanism allows the client and the server to negoatiate a common protocol which both of them supports. Any secure channel is expected to provide the following:

- **Confidentiality**: This property ensures that the data can be decrypted only by the endpoints participating in the communication. Some of the algorithms that are considered as legacy are obsolete in TLS 1.3 are SHA-1, RC-4, DES, 3-DES etc. This makes TLS 1.3 much more secure than TLS 1.2. TLS 1.3 uses only Authenticated Encryption with Associated Data(AEAD) algorithms like AES_GCM(128 and 256 bit key) , AES_CCM(128-bit and 256-bit key) and ChaCha20 and Poly1305. AEAD algorithms not only encrypts the data but also a provides a way to check the integrity of the data.
- **Authentication**: The endpoints should be able to prove their identity. In TLS the server is always authenticated and the client is optionally authenticated. This is done by using digital signatures eg: RSA, ECDSA, EdDSA or PSK.
- **Data Integrity**: Data cannot be modified by the attackers. If it is modified it can be easily identified by the endpoints. Usually this is provided by a separate set of algorithms called Message Authentication Code. TLS 1.3 uses AEAD algorithms that provide integrity along with confidentiality for the data transmitted.

TLS consists of two main components:

- **A Handshake Protocol** is responsible for authenticating the communicating parties, negotiating the cryptographic suites and establishing a shared secret key for communication. The cryptographic parameters that are needed for establishing a secure channel are produced by the TLS Handshake protocol.
- **A Record Protocol** uses the parameters established in handshake protocol to protect the traffic between the endpoints. The traffic is divided into a series of records and each there records are independently protected. The record protocol takes the message, fragments the data into blocks, protects the record and then transmits them. On the receiving side, the data is verified, decrypted, reassembled and then delivered to the upper-level protocol.

In the next subsections, we will see the three basic key exchange modes supported by TLS 1.3. These modes are the main reason for QUIC to take less time for connection establishment [6].

2.1.1 1-RTT Full Handshake

TLS 1.3 supports only forward secure Deffie-Hellman key exchange protocol as compared to RSA key exchange supported in TLS 1.2. Forward Secrecy provides the assurance that the session keys will not be compromised even if the private key of the server/client is compromised. The main change in TLS 1.3 compared to TLS 1.2 is the removal of ServerKeyExchange and ClientKeyExchange instead they are sent as an key_share extension in ClientHello and ServerHello message in TLS 1.3. This saves us 1-RTT time. Figure 2.1 shows messages exchanged in 1-RTT full handshake.

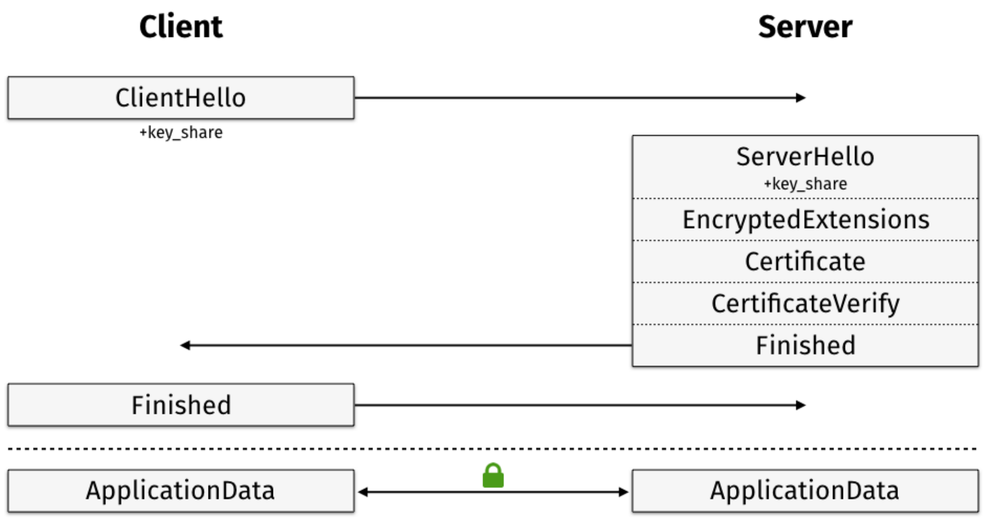


Figure 2.1: TLS 1.3 1-RTT Full Handshake [1]

The three phases invloved in a TLS 1.3 handshake is described below

In the first phase which is referred to as **Key Exchange Phase**, the client send the ClientHello which contains a random nonce, supported protocol version, a set of Diffie-Hellman key share in the key_share extention. The server processes the ClientHello and selects the appropriate cryptographic parameters for the connection. It sends the negotaited parameters in the ServerHello message along with the server's random nonce.

Server Parameters The server sends two messages to establish server paramters, Encrypted Extension which are encrypted using the keys derived. These contains extensions which are not needed to establish cryptographic context. The next message is CertificateRequest extension, it is send if the server requires client authentication. This is omitted if client authentication is not desired.

Authentication takes place after key exchange phase and server paramters are established. For the mandatory server authentication the server sends the Certificate of the server and CertificateVerfiy that contains the hash of all handshake messages exchanged so far, signed with the server's private key. The client verifies the signature using the Certificate's public key. The Finished contains a messge authenticaton code (MAC) over the entire handshake. This authenticates the identity of endpoints, bind the key exchanged to the endpoint's identity. On receiving the server's message the client responds with client's Certificate and CertificateVerfiy if the the server requested for client authentication and Finished. This is optional.

Now the handshake is complete and both the endpoints derive the secret key from the keying material to send the protected application data. The application data cannot be sent before sending Finished message except in 0-RTT mode where the application data is sent in the first packet itself.

2.1.2 Session Resumption

Session Resumption is done by using pre-shared key(PSK) mode. This can be either shared externally or can be established on a previous connection after the handshake is complete. Once a handshake is complete the server sends a unique key derived from the initial handshake. The client can use this pre_shared_key in a future connection along with the ClientHello message. The client also sends key_share extension in case the server wants to decline session resumption and do a full handshake. Server authentication does not happen in this mode and so Certificate or CertificateVerifiy message is not sent. In TLS 1.2 this functionality was provided by session IDs and session Tickets. Forward secrecy can be maintained by limiting the lifetime of the PSK. Since this also takes 1-RTT it is not very compelling to use. The figure 3.9 depict the messages exchanged in TLS Session resumption mode.

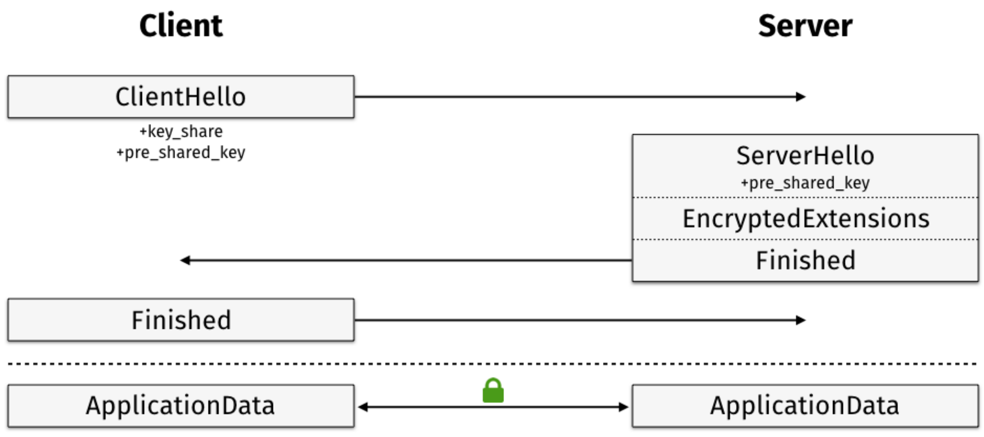


Figure 2.2: TLS 1.3 Handshake Session Resumption [1]

2.1.3 0-RTT

TLS 1.3 allows a client to send data on the first flight using a server configuration message. The configuration message contains an identifier, server's semi static EC(DH) parameters and expiration date. The client appends the application data encrypted using the static secret to the intial ClientHello. The server decrypts and then responds with the answer for the requested query. The 0-RTT data suffers from the following security weakness.

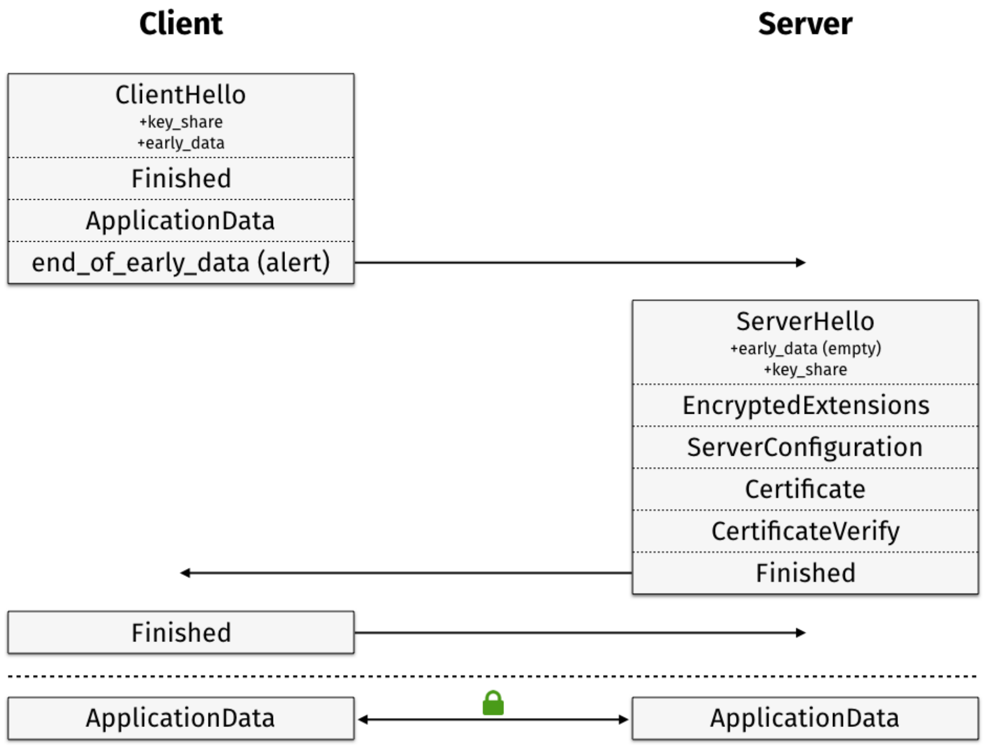


Figure 2.3: TLS 1.3 Handshake 0-RTT [1]

- **No Forward Secrecy** The encryption for the first flight data in 0-RTT is done using the static key which is derieved from the semi static DH key parameter given in the server configuration message. Thus in later stages if the attacker gets access to the PSK, the entire conversation of 0-RTT can be decrypted.
- **Replay Attacks** Replay protection in 1-RTT data is given by the server's random value which is shared in ServerHello message. But 0-RTT does not depend on ServerHello and so there is no guarantee for replay attacks. Replay protection will be guranteed after the ServerHello message is sent.

2.2 QUIC

There is a growing demand for low latency in connection establishment without compromise in security or reliablity. Currently, applications are often limited by the use of TCP as an underlying transport protocol. TCP without the support of any other protocol suffers from latency in connection establishment, head-of-line blocking etc. QUIC is a new secured transport layer protocol which is being developed by the IETF group to tackle the mentioned issue. It runs on top of UDP and provides the functionality similar to TCP, TLS and SCTP. Some of the features which QUIC provides is version negotiation, low-latency connection establishment, stream multiplexing, authenticated and encrypted payload, flow control on stream as well as connection level, connection migration etc. we will see in detail about all these advantages in the following subsections. Also, since QUIC runs on top of UDP, there will be no changes to the client operating systems and middleboxes. This enables rapid deployment.

QUIC protocol relays on TLS 1.3 handshake for agreeing on cryptographic protocols and exchanging ephimeral keys. The TLS handshake is also not only used to negotiate crypto parameters but also used to validate version negotiation and selection, agreeing on QUIC transport parameters and allows server to perform routeability checks on client. The version negotiation mechanism in QUIC is used to negotiate a version of QUIC before the completion of the handshake. Since this packet is not authenticated, there is a possibility for an attacker to force a version downgrade. QUIC overcomes this by copying the version information into TLS handshake and this provides integrity protection for the version negotiation [7].

During the connection establishment both the endpoints share their transport parameters which is integrity protected as they are included in the TLS handshake. These QUIC transport parameters are carried in the TLS extentions. Some of the mandatory transport parameters are *initial_max_stream_data*, *initial_max_data* and *idle_timeout* [8]. *Initial_max_stream_data* contains the initial value for the maximum amount of data that can be sent on any stream. This is also equivalent to sending a MAX_STREAM_DATA frame on all streams after opening. *Initial_max_data* contains the initial value for the maximum amount of data that can be sent on the entire connection. This is equivalent to sending a MAX_DATA frame on the connection immediately after completing the handshake. *Idle_timeout* contains the timeout value in seconds for which the server will keep the connection open when there is no packet is sent or received in the connection. Few other optional transport parameters are *max_packet_size*, *disable_migration*, *initial_max_bidi_streams*, *preferred_address* etc.

2.2.1 Connection Establishment

QUIC improves the connection establishment time as it uses TLS 1.3 to open a secure connection. In a normal TCP + TLS1.2 connection, it takes 1.5 RTT for a TCP handshake and 1.5 for a TLS connection ie., at least 3-RTT for a TCP + TLS1.2 connection. In contrast, QUIC improves on connection latency by integrating with TLS 1.3 and takes only 1-RTT for a first-time connection and 0-RTT for the subsequent connection.

From the figure 2.4 we can see that for the first time connection establishment QUIC takes 1RTT by carrying both the TLS handshake parameters and QUIC transport parameters in the first packet. 0-RTT is one of the important features which QUIC provides. Most of the time the client request a connection to the server with which it has interacted before. If the server remembers the client and the cryptographic key it is possible for the client to send a request to the server in the first packet itself. This saves the 1 round trip time.

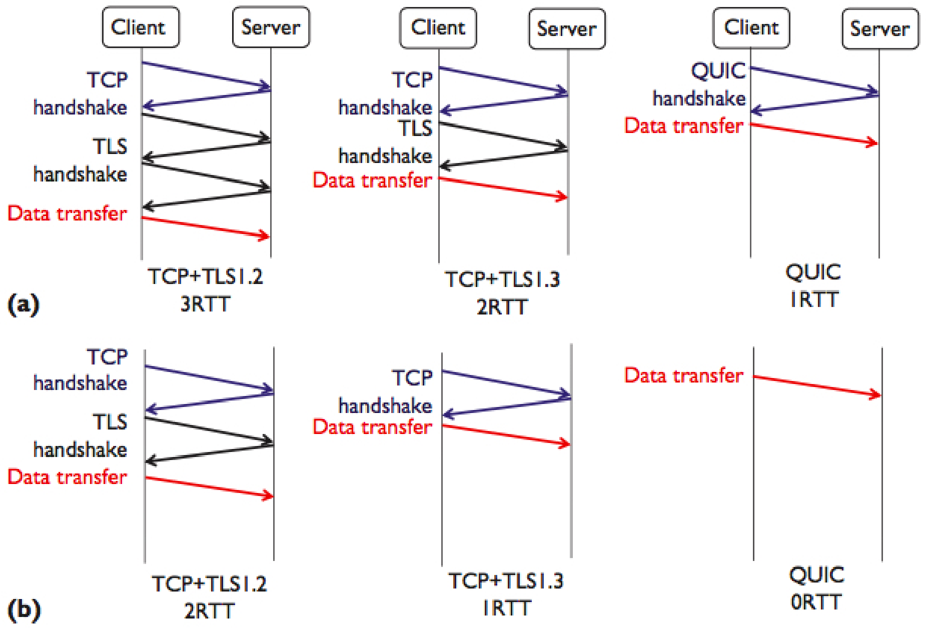


Figure 2.4: Round Trip Time for different protocol a). First time connection b). Subsequent Connection [2]

2.2.2 Connection Migration

In a TCP connection, a connection is identified by a 5 tuple(Source IP Address, Source Port Number, Destination IP Address, Destination Port Number, protocol). So, when any one of the parameter changes then the connection is closed due to timeout and a new connection is established. This change in a parameter can happen when a client moves from wireless network to wired or if the same client wants to establish a connection to the same server on another port etc. QUIC uses a 64-bit connection ID to uniquely identify a connection instead of the 5 tuple. This is how QUIC connection survives changes to IP addresses or port addresses. Usually, clients are responsible for initiating migration.

Connection migration should not be initiated before the handshake is finished and a 1-RTT key is established. An endpoint cannot initiate a connection migration if it's peer sent the *disable_migration* transport parameter during the TLS handshake. If the peer detects a connection migration even though the disable_migration parameter is sent, then it treats it as a connection error and closes the connection [8]. The process of connection migration is described below

- **Probing a New Path** The client verifies the reachability of the server from the new local address by using path validation mechanism. This is done prior to migration of the connection to the new address. The endpoint uses PATH_CHALLENGE frame to check the reachability of its peer. The frame contains an arbitary 8-byte data which is difficult to guess and it uses a new connection ID for the probes sent from the new local address. The server responds with a PATH_RESPONSE frame. If the content of both the frames does not match, then the client generates a connection error.
- **Initiating Connection Migration** The endpoint can migrate a connection to a new local address by sending frames in packets other than probing frames. When migrating the new path might not support the old path's sender rate so the congestion control values have to be reset.
- **Responding to Connection Migration** Receiving a non-probing frame in packet from a new client address indicates that the peer has migrated the connection to that address. The server skips the path validation if it has seen the address recently, otherwise it initiates a path validation by sending PATH_CHALLENGE frame to validate the ownership of the new address.

We stated before that only clients can perform connection migration, there is one case in which a server can perform connection migration. QUIC allows server to accept connections in one IP address and after completion of handshake the server can transfer the connection to a preferred address. This is done by including the *preferred_address* transport parameter in the TLS handshake. Once the *preferred_address* is conveyed to the client, the client waits till the handshake is completed. Now, the client initiates a path validation of the server's preferred address using the connection ID provided in the transport parameter. The client can start sending the future packets to the new server address as soon as the path validation is completed successfully and discontinue the use of old server address. If the path validation fails, then the client must send the future packets to the original server address. A server cannot migrate a connection to a new server address when it is already in the mid-connection.

2.2.3 Stream Multiplexing

Connection is a term used in QUIC to describe a conversation between two QUIC endpoints usually a client and a server that uses the same encryption parameters. A single connection can contain multiple streams. Connection is initiated by a QUIC client whereas streams can be initiated by either party. Stream multiplexing is achieved by using more than one STREAM frames from different streams in the same QUIC packet. This solves the problem of head-of-line blocking ie a loss in one QUIC packet halts only the streams involved in that packet and waits for retransimission to occur. Every other stream can continue making progress. It's upto the implementation to bundle as few streams as necessary so than the transmission efficiency is not affected. Figure 2.5 shows the comparison of multiplexing in HTTP1.2, HTTP/2 and QUIC by sending multiple streams of data over a single connection

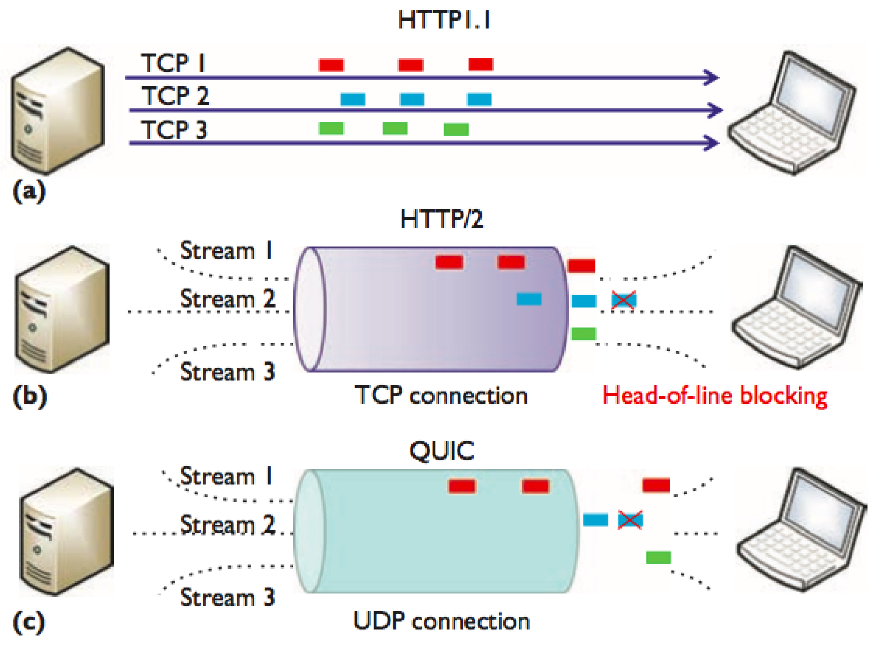


Figure 2.5: Multiplexing comparison (a) HTTP1.1, (b) HTTP/2, and (c) QUIC. [2]

Streams can also be prioritized by the application protocol which uses QUIC to make a significant effect on performance. Effective stream prioritization helps in getting a better performance.

2.2.4 Flow Control

QUIC supports flow control mechanism by using a credit-based flow control scheme and it is similar to HTTP/2 flow control. QUIC employs two levels of flow control in QUIC by making the receiver send the maximum octets it can receive on a given stream and the entire connection

- **Connection Level Flow Control**: In this a receiver sends a MAX_DATA frame which tells the peer that the maximum amount of data that can be sent on the entire connection. The endpoint terminates the connection with an error if the peer sends data more than the advertised value. This ensures that the sender does not exceed the receiver's buffer capacity.
- **Stream Level Flow Control**: For stream level flow control MAX_STREAM_DATA frame is used, in which it specifies the maximum amount of data that a peer can send on that particular stream. Stream level flow control ensures that a single stream doesnt consume the entire receiver's buffer capacity.

A receive can increase the maximum amount of data that it can recieve by sending again a MAX_STREAM_DATA or MAX_DATA frames with the larger offset value. At any point the receiver cannot advertise a smaller offset that it has already advertised. A BLOCKED or STREAM_BLOCKED frame is sent my the sender if it has data to send but it is blocked by the flow control limits. If the peer violated the flow control limits by sending more data than the advertised offset in either stream level or connection level then the receiver closes the connection with the FLOW_CONTROL_ERROR.

2.2.5 Congestion Control and Loss Recovery

In this section, we will see in high level how QUIC implements congestion control and loss recovery. These algorithms are similar to the well known TCP ones but there are few differences in the algorithms because of the following difference in QUIC.

- **Separate packet number spaces:** QUIC uses different packet number space for each encryption level and each endpoint maintains a separate packet number spaces for sending and receiving. The three spaces are initial space, handshake space and application data space.
- **Monotonically increasing packet numbers:** Every transmission in QUIC has a packet sequence number and this number never repeats within the same packet space in one connection, and are monotonically increasing. QUIC carries a new sequence number even for the retransmitted data, unlike TCP which has the same sequence number as the original packet. This design eliminates significant complexity.

- **More ACK ranges:** QUIC supports more acknowledgement range compared to TCP's three ACK range. This reduces false retransmissions and recovers fastly in a high loss environment.

QUIC uses ACK information and timeouts to detect packet loss [9]. The ACK based detection of packet loss is similar to TCP's Fast Retransmit and Early Retransmit algorithm but they are implemented differently in QUIC. In fast retransmit algorithm, if an endpoint receives an acknowledgment for a packet which is more than *kReorderingThreshold* than the unacknowledged packet, it means the unacknowledged packet is lost. The inital value of *kReorderingThreshold* is three which is derived from the TCP loss recovery. Fast retransmit algorithm does not work when there is a packet loss and there is no new packet to send. The receiver thinks that the end of data is reached where as the sender waits for *kReorderingThreshold* number of ACKs before retransmitting. QUIC overcomes this by marking the unacknowledged packets as lost after a small period of time.

QUIC follows TCP's congestion algorithm TCP NewReno to determine the congestion window [9]. Initially a connection starts with slow start and increases the congestion window depending on the number of bytes it acknowledged. When a loss is detected, it decreases the congestion window by half and sets the slow start threshold to the new value. It uses additive increase multiplicative decrease to avoid congestion.

2.2.6 QUIC Packet Types and Formats

In order to develop a test framework for QUIC we need to understand the packet formats and types. Any QUIC packet has either a short header or a long header. Long headers are used in the early part of the connection ie., before the establishment of 1-RTT keys and version negotiation. Short headers are used after version negotiation and 1-RTT keys are established [8].

2.2.6.1 Long Header

Long Header has Header Form bit set to 1. It contains a 7 bit packet type, 32-bit QUIC Version field, destination and source connection IDs, packet number , payload length and payload. The important field which helps in decoding the log files for the framework is packet type. There are 4 packet types defined in Table 2.1

1. **Initial Packet** uses long headers with packet type 0x7F. Initial packet carries the first cryptographic message sent by the client and has the packet number 0. All the subsequent packets contains packet number incremented by 1. The client populates the source connection ID with some random number. The client also populates the destination connection ID if it hasn't received a retry packet from

TYPE	NAME
0x7f	Initial
0x7e	Retry
0x7d	Handshake
0x7c	0-RTT

Table 2.1: Long Header Packet Type

the server before. If it has received a retry packet and this is a second initial packet then it populates the destination connection ID with the source connection ID from the retry packet. The initial packet has two additional fields than the long header token length and token. This fields are use when a client sends an initial packet in response to the Retry packet sent by the server.

2. **Retry Packet** uses long headers with packet type 0x7E. It is used by the server which wishes to perform stateless retry. It must contain at least two frames, one ACK frame to acknowledge the clients initial packet and the another STREAM frame containing the server's retry token. This token is opaque blob of data without any well defined format. It can contain information about the claimed client Ip address and timestamp, and other additional details needed by the server for validation. The server populates the destination connection ID with the source connection ID of the initial packet. The source connection ID is randomly selected by the server.

3. **Handshake Packet** uses long header with packet type 0x7D. It is used by both server and client to exchange cryptographic handshake messages and acknowledgements. A server sends one or more handshake packets in response to client's intial packet if it does not send retry packet. A client sends its subsequent cryptographic handshake messages in handshake packet and acknowledgements to the server.

4. **0-RTT Packet** uses long header with packet type 0x7C. These are sent by endpoints to send data protected with 0-RTT keys. The connection ID's should match with the values used in intial packet.

2.2.6.2 Short Header

Short Headers are used after 1-RTT key is established. It has Header Form bit set to 0. It contains only destination connection ID which is chosen by the intended recipient of the packet, packet number and protected payload. The packets with short header always contains 1-RTT protected data.

2.3 Implementations of IETF QUIC

In the below table 2.2 we can see the various implementations of IETF QUIC specification in different programming languages. Some of the implementations does not provide their source code, but it is possible to get the public servers IP address and port number which serves QUIC client. These implementations use any other application protocol other than IETF HTTP over QUIC eg. HTTP/0.9. The interop meetings are conducted periodically to ensure that the implementations interoperate with each other. The QUIC working group defines a set of implementation milestones for the successive interoperability test. Currently only 11 out of 16 implementations participate in interoprability test. For the interoperability testing each of these implementations is run as server and client in all possible combinations manually.

Implementation	**Language**	**Public Source Code**
picoquic	C	Yes
ngtcp2	C	Yes
quant	C11	Yes
mozquic	C++ with C interface	Yes
quicly	C	Yes
ATS	C++	Yes
winquic	C	No
f5	C	No
pandora	C	Yes
ngx-quic	C	No
applequic	C, Objective C	No
mvfst	C++	No
minq	Go	Yes
QUICker	NodeJS/Typescript	Yes
quicr	Rust	Yes
quinn	Rust	Yes

Table 2.2: IETF QUIC Implementations

2.4 Automated Testing Tools

Testing is an important process in software engineering. It helps in evaluating the quality of the software and also many aspects such as reliability, usability, portability, maintainability etc. This can be done with either manual or automation testing. Manual testing is more time consuming on the other hand test automation saves a lot of time and cost. Testing tools are designed to target one particular testing criteria, eg Selenium is one of the popular test automation tools for testing web applications [10]. Automation framework, on the other hand, provides an infrastructure to use different tools. Test automation is one of the important steps in test-driven development. In the following subsection, let's see one such framework using which we can achieve automated testing.

2.4.1 Continuous Integration

In a software devlopement project, there are usually many developers working on the same project and each person integrates his code multiple times a day. The practice of continuous integration came into existence to prevent integration problems which arises when merging with others code. These integration problems does not happen in a small project with only one developer working on it. It helps in merging all the working copies of the developers who are working on the same project with minimized integration problems.

Most of the continuous integration tool typically use a build server to automate the build process and to detect integration errors as quicly as possible. Once the code is built, a series of tests are run to validate that the commit did not break the application. Even though continuous integration can be done without any testing, software quality can be compromised if there is no test automation involved. Self testing of the build is one of the key practices for an effective continuous integration and delivery system.

Continuous Integration is a subset of continuous delivery and continuous deployment. Continuous Deployment is practice of keeping your application deployable at any point of time and automatically move the code to the test or production environment if all the test case passes. On the other hand continuous delivery refers to the practice that the application is deployable any time along with the necessary configurations needed for the production environment. Here the changes are moved to the test or production environment manually.

The typical workflow of a continuous integration task is show in figure 2.6 and is explained below [11].

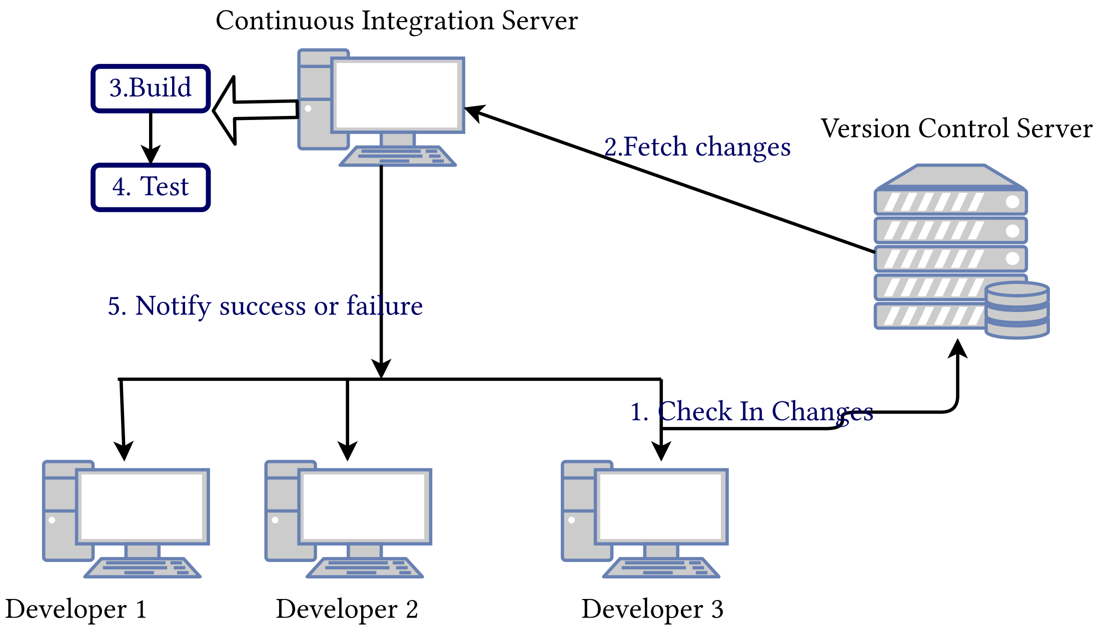


Figure 2.6: Workflow of Continuous Integration

1. During the application development lifecycle, the developers are working on their local copy and commit them to the central source code repository for version control like Git. This is the start of continuous integration pipeline.
2. A new release is triggered when the developer's branch is merged with the main branch.
3. Usually a continuous integration system monitors the version control system for the change in certain interval and if it finds some changes in the main branch, the build process is triggered.
4. The CI system pulls the latest code from the version control repository and the build step is responsible for compiling, fetching dependencies, linking and making the program executable. The program executable might run but it might not work as intended. This is why we need the next step in a CI system.
5. Unit tests are run to validate the changes and quality. This is a good way to catch bugs and further if needed a suite of automated tests is run to check the intended functionality of the software and if there any test failures, it fails the build process.
6. After the testing process, the CI notifies the team members of the project about the build failure or success. The notification can be either through email, pop-ups or SMS.
7. Finally, the application is deployed if all the tests are successful.

2.5 Interoperability Testing

Internet is growing day by day and the devices used for interconnection are from different vendors and the work on different platforms. These devices have to communicate and achieve the end to end functionality that is defined by the standard. Interoperability is the key to success when a new technology or a protocol is developed. Interoperability testing is needed to ensure that a particular protocol works as expected between these different hardware and platforms. This test helps in identifying if there are some induced errors while implementation or if there is some ambiguity in the standards. Each vendor does their own implementation of the standard and if there are some ambiguities in the specification, interoperability test can identify them. Other than that interop test can find if the desired performance is achieved according to the standards.

For this thesis, we are concerned with only interoperability testing. Often interoperability testing and conformance testing are coupled together. Conformance testing deals with determining whether an implementation behaves according to the standard's definition whereas interoperability testing deals whether two systems can interoperate with each other. Both of these tests are related in the sense that if an implementation passes the conformance testing there is more chance that the interoperability testing will be successful. If both the implementations conform to the standard then with a high level of confidence, they will interoperate. Interoperability testing finds issue outside the scope of the standard whereas conformance testing can find the issues within the scope of the standard. We will deal with developing a framework which tests interoperability and conformance testing is out of scope for this framework.

Figure 2.7 depicts the generic framework for automated interoperability testing defined by European Telecommunications Standard Institute(ETSI). This framework is mainly used for testing interoperability in telecommunication.

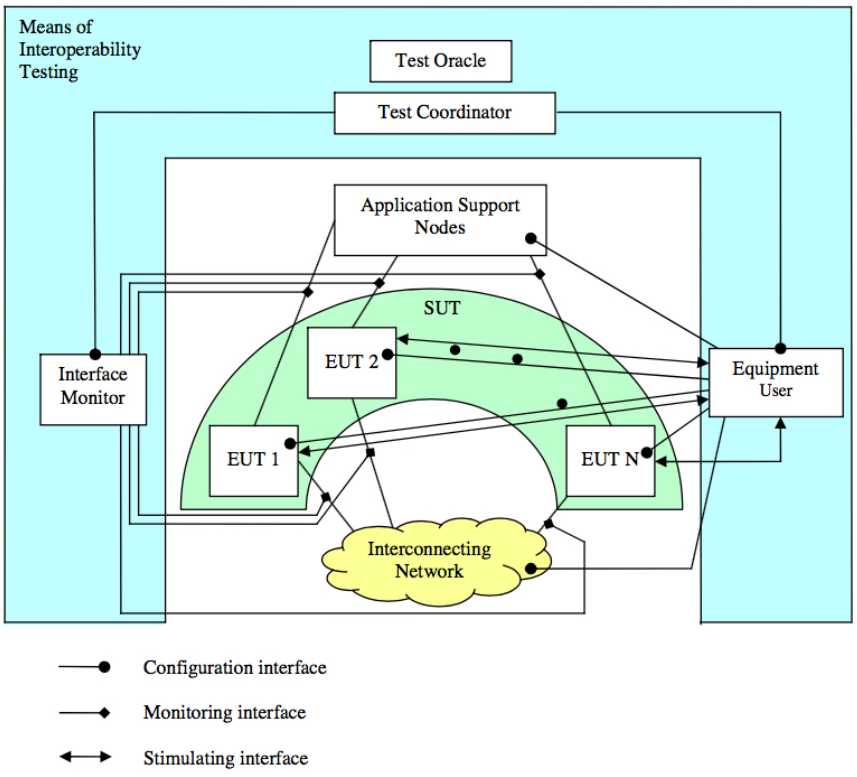


Figure 2.7: General Framework for Interoperability Testing [3]

The figure above shows the required concepts and the process for test setup and specification development and it is derived through the experience gained from performing interoperability testing in IPv6, Voice Over IP, WiMax etc., [3]. The framework says that at all the standardized interfaces connecting Equipement Under Test(EUT), there should be packet capturing or monitoring system. The user interactions are automated ie., the access of web page by the QUIC client to the QUIC server is automated. The test coordinator handles all the user interactions and traffic captured. The test oracle takes care of managing inteteroperability and conformance testing and it is also a part of test coordinator. Later, it is analysed and the test results are produced.

2.5.1 Objectives of Interoperability Testing

The objectives of the interoperability testing is to ensure that the network elements from different vendors can communicate with each other. These network elements mosly have different implementations of the same standards as they are from different vendors ie., independent implementation of the same standard.

2.5.2 Steps for Interoperability Testing

We need to understand the steps which are needed to do interoperability testing manually before developing a automated testing framework for interoperability testing.

1. **Setup** This phase is to define a formal statement of work and the necessary test arrangements are made. The configuration, product name of each concrete devices should be listed. Other physical test arrangements like the means of communication is also established. The necessary dependent tools are also set up.
2. **Plan** This is one of the important phases in any development process. It is very important to understand the entire application in our case it is the protocol specification to determine the procedure for interoperability testing. Usually, a test plan document is used to document the planning of interoperability testing. The number of test cases, the order of the test cases and their relationship is identified. The main focus is to test the end-to-end functionality of the application/protocol covering all possible scenarios. Apart from the functionality of the application the non-functional testing like performance, security etc., should also be tested. We will be covering this part of the process in Chapter 3.
3. **Develop Test Scenario** In this phase the test scripts to be automated is developed. These test scripts can cover one test case or a test suite. A test suite is a collection of test cases, one for each test purpose. We also develop a way to validate the test cases in this step ie Pass or Fail of a test case. This is covered in Chapter 4.
4. **Execute Test Scenario** The test cases or the test suites developed are executed in the real environment. The outcome of this phase which is either pass or fail of test case is recorded.
5. **Document and Report Test Results** A test report should summarize the testing activity and explicitly state that if the implementations interoperated with each other. The report should contain the timestamp, details of the test equipments and it's configuration and finally the outcome of the tests. The details about the failed test case is also reported with supporting logs. The main aspect in this type of testing is that the test results are properly reported to the parties participating in the interopertability testing or the organizer of an interoperability testing event.

2.5.3 Challenges in Interoperability Testing

There are number of problems that arise in interoperability testing. These challenges cannot be fully solved but it can be reduced with the appropriate processes or methodologies [12].

- **Network Complexity**: There are a lot of devices involved in between two end points which wants to communicate over the Internet. It is difficult to simulate such a big network instead we can only do a pair-wise testing.
- **Scalability of testing**: The cost and complexity of setting up an entire network is high. Pair-wise testing is the common form of interoperability testing and it is cost-effective comapred to setting up the entire network. Pair-wise testing also suffers from some complexity and disdvantages. It is Usually performed on two implementations from different vendors. Consider vendor X develops implementation A and vendor Y develops implementation B. The person doing interoperability testing should have the complete information of both the implementations A and B. The number of test case increases as the number of implementations increases. Suppose there are N implementations then we have at least$(N^2 - N)/2$ distinct combinations of tests to run. Also even though the implementations are tested against each other, the test may fail when it is combined in certain way.
- **Inadequate requirements**: The other issue is that each device may run a different implementation and/or on a different operating system. The test might pass when the protocol is run on one type of operating system but might fail on others. Also, the required performance might not be achieved when running on a specific device due to memory/bandwidth constraint.
- **Determining the root cause of the failure**: When a test fail, it is difficult to understand the root cause of it's failure. The test may fail due to incorrect setup of the test environment, one of the implementation not following the standard, the standard was ambiguous. Each of these possible scenarios should be analysed for addressing the root cause of the defect. This can be avoided to some extent if both the implementations are fully functionally tested, then the root cause of the failure is due to the environmental setup of the test.

2.5.4 Automate Interoperability Testing

There is a need to reduce the time to market a for new product, protocol or service in the current competitive environment. Since a lot of different hardware and implementation is involved, doing interoperability testing manually is time-consuming and involves a lot of repetitive tasks. During the early development stage of a product or protocol, there is a change in the design specification quite often. This leads to re-running the interoperability test quite often. Automating interoperability testing reduces the time for testing and reduces error which is caused because of repetitive manual testing. The start-up cost for automation is quite high and therefore be used only when the tests are run frequently.

Also, there is no guarantee that if an implementation A interoperates with implementation B and implementation B interoperates with implementation C then implementation A interoperates with implementation C ie the testing is not transitive. So, there is a need to run all combinations of tests. The above-mentioned reasons prove that there is a clear need to automate the interoperability testing.

2.6 Summary

In this section we breifly described TLS 1.3 which provides security to QUIC protocol. Later, we eloborated the various features which QUIC provide, packet format and types used in QUIC. The importance of automated testing and the need of intereoprability testing, the steps involved in it and as well as the challenges involved in intraoperability testing is listed. In addtion we described interoperability testing and its differences to conformance testing. The following chapter provides the main contribution of the thesis work, which is the framework design for the QUIC's intereoperability testing.

Chapter 3

Design

In this chapter, initially we talk about the various issues in designing a framework for QUIC interoperability testing and the conceptual model for collecting logs at various points. Later we propose three setups for running intereoperability test based on the available resources. Finally, we list of the basic test scenarios for testing QUIC's interoperability between different implementations.

3.1 Problem Description

One of the main issue with running intereoperability tests between different implementations is that the IETF QUIC Specification is still in progress and there are changes to the wire format whenever there is a new draft published. Each implementation is done by different group of people and hence one implementation might be feature complete and other implementations might still be in progress. This situation makes it difficult to run interoperability test as both the specification and implementations are continously evolving. Each implementations are evolving on a different pace.

The other issues is that some of the implementations are not open source. This makes it difficult to test some the scenarios. These implementations have a remote server running as QUIC Server or we have access only to the binaries for testing. In both cases we don't have access to the actual source code. This makes analyzing the log file and debugging difficult. With the implementations for which we have the source code, we have to find the necessary dependent libraries and build files to compile and build the project before starting the testing phase.

QUIC logs are not standardized. Each implementations does their own way of logging on server and client side. Some implementations do not provide the server logs. To find out what exactly is happening we have to analyze the logs manually and build a parser for each QUIC implementations. This is a time consuming process as well as not scalable when there are new implementations. Further with each updates in the source code, the structure of the log file can change drastically which forces to rebuild the parser. To overcome all this we propose a solution as a future work to standardize QUIC logging mechanism by generating a events at specific point in each of the implementation. This simplifies our testing process as we have a common log structure across all implementations. This approach is also scalable when there are new implementations if they follow the common logging interface.

3.2 Conceptual Model

The conceptual model for running interoperability testing between a client and a server is shown in Figure 3.1. From the figure, we can see that the client and the server use QUIC protocol communicate with each other over the Internet. We can collect logs at two levels and capture the packets exchanged between them. These details can be used to validate the test cases. In some cases we might need the logs from more than one layer to validate. Consider the test scenario of streaming data, to verify that, we need logs from the QUIC to see if the stream is opened and data is sent between the client and the server. We are not sure from the QUIC logs if the entire data is shared to the application layer. To verify this we also need logs from the application protocol.

- Using Kernel Logs

  Kernel logs can be obtained using various command line tools like netstat, tcpdump. Netstat can show all the TCP and UDP ports which have active connections. It can also show statistics about each protocol. This tool is not much useful because it does not provide more information about the packet's content. These tools can be used for trouble shooting or to make sure a particular port is open and is active. Tcpdump on the other hand can display the traffic that is either transimitted or received from the system. Tcpdump can be made to listen on a particular interface and display packets with a particular IP address and port number. These captured raw packet provides more information about the protocol flow. However, they are encrypted before transmitting and so we will will not be able to obtain more useful information.

- Using Server and Client QUIC Logs

  Each implementation logs the states, packet sent/received, error message in their own way for debugging. There is no standard structure followed in these implementations. Thus, they require a lot of time in understanding how each implementation follow their logging mechanism. There is also a need to parse the log generated by each implementations into standard events for further processing. We will see more about this in chapter 4.

- Using Wireshark Wireshark is a network analysis tool that helps in capturing packets on a specific interface. It can filter and sort packets according to certain criteria. It is possible to filter QUIC packets from interface eth1 alone. It is a great tool for troubleshooting the network.

  The current stable version of wireshark is 2.4.5. This still supports only Google QUIC. The latest development version 2.5.1 supports IETF QUIC. Wireshark development version is still at draft-12. 0-RTT decryption is still not supported, so we will not be able to use Wireshark for testing 0-RTT. For testing version negotiation, handshake, and stream data wire shark packet capture can be used. This way of validating is not much useful because the QUIC payload is encrypted.

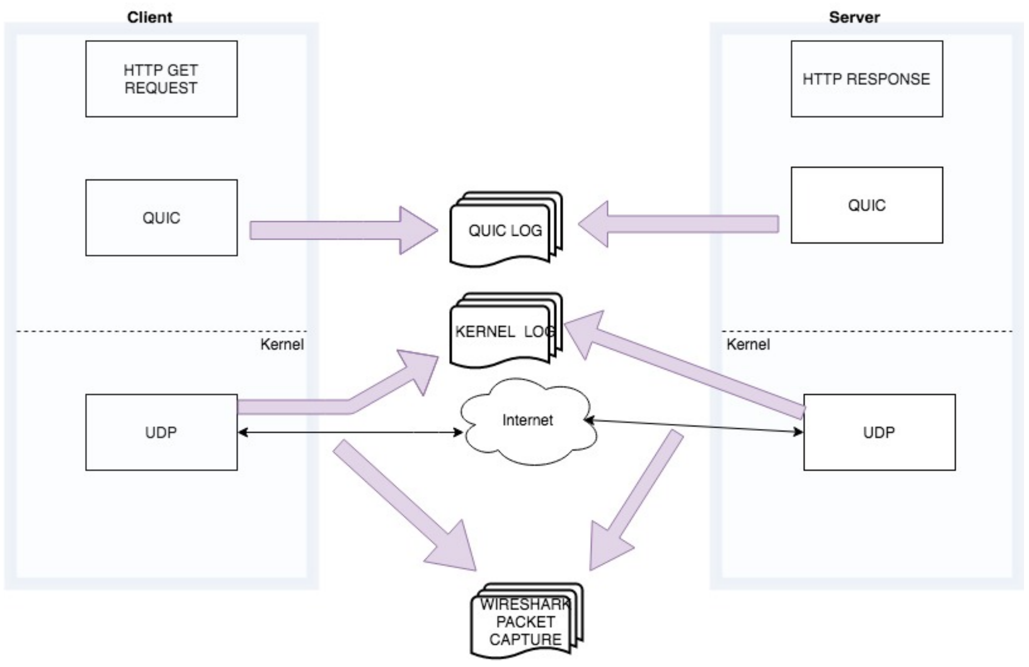


Figure 3.1: Conceptual Model of the Environment

3.3 System Design

In this section, we propose three system design for running interoperability test for QUIC protocols. These are ordered from low to high on the basis of test setup cost.

3.3.1 Lightweight System

In the lightweight system we run both client and the server in the same machine as different user processes. As the name suggests, it is relatively simpler and faster to use than the other designs mentioned below. This system is easy to setup and we don't need any extra resources for testing and evaluating the design. The cost of setting up this environment is much lower than the other system designs.

The portability of the protocol cannot be tested under this setup. Most of the implementations are designed to be run on top of different operating systems. Mostly the sever and client doesn't have the same operating system but they are required to interoperate. This cannot be validated because we use only a single machine both as a client as well as the sever, and the tests are run on the underlying operating system it supports.

The traffic between the client and the server goes through the loopback interface and hence there is no network between them. So we can't examine how the protocol operates under different traffic conditions. To mimic the actual internet environment we can use traffic generator to increase the inflow of data to the network. These data are dummy packets with unique identifier generated by the tool. There are various tools to this job like iperf, bwping. This technique does not provide us any use because in the lightweight design we do not have a network and so using a traffic generator is not possbile. On the other hand, we can use traffic shaping to throttle the bandwidth, to prioritize a certain traffic and to drop low-priority packets. Traffic Shaping is a technique which manages the bandwidth according to the required traffic profile. This can be used to regulate the performance and quality of service. Traffic Shapping retains the flow of excess packets in a queue and later schedules for transmission when the network condition is favourable.

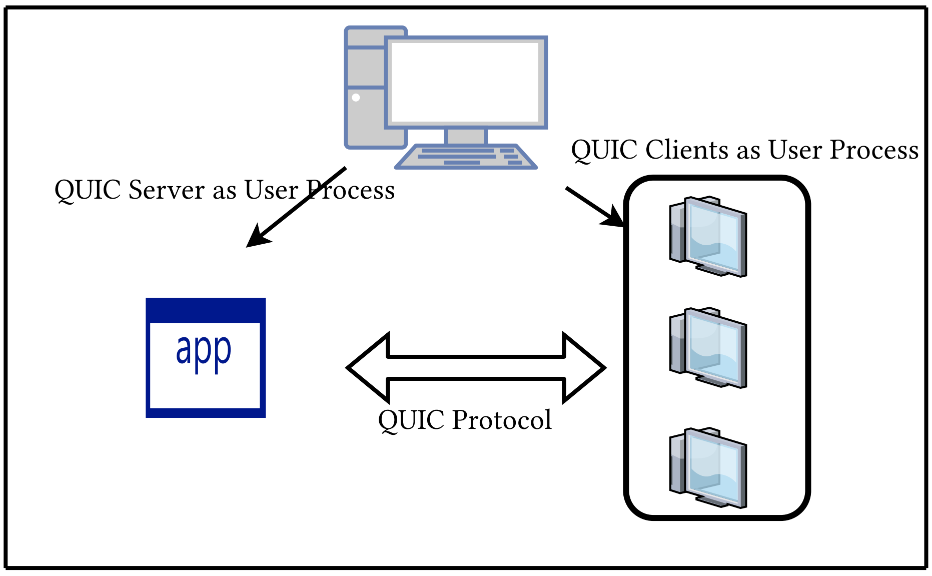


Figure 3.2: Lightweight System

Figure 3.2 show how a light weight system design would look like. From the figure we can see that, we need only one physical hardware and it acts as both the server and the client.

The disadvantage with the light weight design is that the measurement of paramters like latency for connection establishment will not be accurate. This is because there is no network between the client and the server. This can be emulated with a help of traffic shapers to some extent and then we can measure the latency of connection establishment. Another drawback with the lightweight system design is that we will not be able to test the implementation against different operating systems. Suppose we have an implementation running as a QUIC server on a windows operating system and we want to test the interoperability with another implementation running as a client on a unix-based operating system, it is not possible to test such scenarios.

3.3.2 Virtualized System

Using virtual machines for interoperability testing can help in emulating different machines with different operating systems on a single physical computer. There can be multiple guest operating system controlled by the hypervisor and the hypervisor is responsible for sharing and managing the hardware between the guest operating systems. Each of these virtual machines can be be QUIC server or client. The QUIC processes running on the virtual machines behave as if they were running on their own physical machine. The advantage of this approach is that we can test the protocol compatibility on different operating system without having to spend money on buying the hardware specifically for testing.

This system design makes it possible to setup a virtual network between the client and the server. It is also possible to move the virtual machines on different servers with physical network between them. This virtual network helps us to use traffic generators to increase the traffic in the network and see how QUIC works when there is congestion. We can also use traffic shappers to regulate the trafiic according to the test scenario.

The limitation of this setup is that the host machine's configuration should have enough memory and CPU to support multiple guests. This can be overcome by using increasing the number of servers and moving the virtual machines across different servers for testing.

3.3.3 Physical System

In this setup, we suggest to use a lot of different hardware devices where QUIC might be used in real world. Raspberry Pi is a low cost system and single board system that can be used for testing interoperabilty of QUIC. It supports both wired as well as wireless connection. Some model doesn't have an onboard ethernet port but it is possible to use an external user-supplied USB ethernet. The latest model Raspberry Pi 3 provides 2.4

GHz WiFi 802.11n and onboard ethernet port [13]. It recommends to use linux based operating system but some windows based OS like Windows 10 IoT core can also be run on it.

Mobile phone are used extensively to communicate with the outside world. The total number of site visits from mobile phones has increased by 7% in 2017 as compared to 2016 in US [14]. This shows that most the internet traffic is from the mobile phone and we need to verify that QUIC implementations interoperate with each other when they are runnning on a mobile phone.

The other thing to decide on the design aspect of the problem is what mode of communication to use ie., is it a wired connection between the devices or they communicate through a wireless medium. Again in wireless, the device communication can be through cellular network or through technology like WiFi which uses IEEE 802.11 standard. Wireless communication suffers from packet loss and so it makes it possible to understand how QUIC performs under packet loss without having to use tools for it.

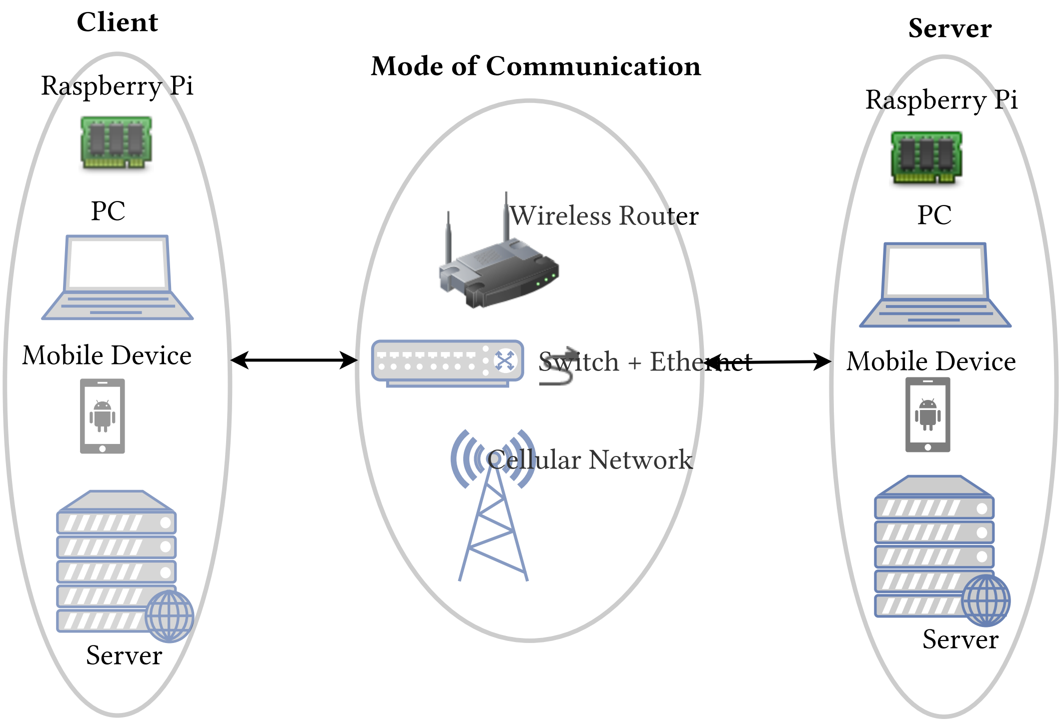


Figure 3.3: Physical Setup

Usually it is enough to run the implementations as QUIC client on mobile phones, PC and Raspberry Pi as they dont have the enough computation power and resources to run as server. Figure 3.3 shows that it is possible to run all devices as QUIC client as well as server. Physical system for testing interoperability suffers from a overhead that it is not possible to automate to run every day as it depends on the physical availabiity of a device. The mobile phones and raspberry pi should always be powered and ready for the interoperability testing.

3.3.4 comparison

In this section, we present the design comparison of the above three systems in table 3.1. The comparison is done based on the key parameters that is needed for deciding the appropriate model for implementation.

Parameter	**Lightweight**	**Virtualized**	**Physical**
Cost	less expensive	system with good configuration to run VM	more expensive than any other setup
Network traffic	cannot be simulated	possible	possible
Mode of Communication	localhost	virtual network	wired, wireless
Different OS	depends on the underlying OS	yes	yes

Table 3.1: Comaprison of the three design

3.4 Test Scenario

In this section we will see the some of the important test scenarios that we propose to determine the interoperability of two implementations. Test Scenario helps in testing the functionality of the entire protocol. This list is not exhaustive and there is a possibility to add more test scenarios to ensure that two implementations interoperate with each other. Each of the test scenario has further specific test cases which is most probably tested in the unit testing of the individual implementation. In this section we will see the test scenario in detail and how the protocol flow should look like for each scenario. This helps us in implementing the framework which will be discussed in next chapter.

3.4.1 Version Negotiation

QUIC Version Negotiation packet is sent by the server to the client in response to a client initial packet that contains a version which the server does not support. This test scenario can be made by explicitly setting the following reserved version *0x?a?a?a?a* by the client or by sending a version which the server does not support. The reserved version is not a real protocol version, the client uses this with the expectation that the server will initiate version negotiation. Currently, the version numbers are used to identify the IETF drafts ie., if the version number is set to 0xff00000D then it means it follows ietf draft 13.

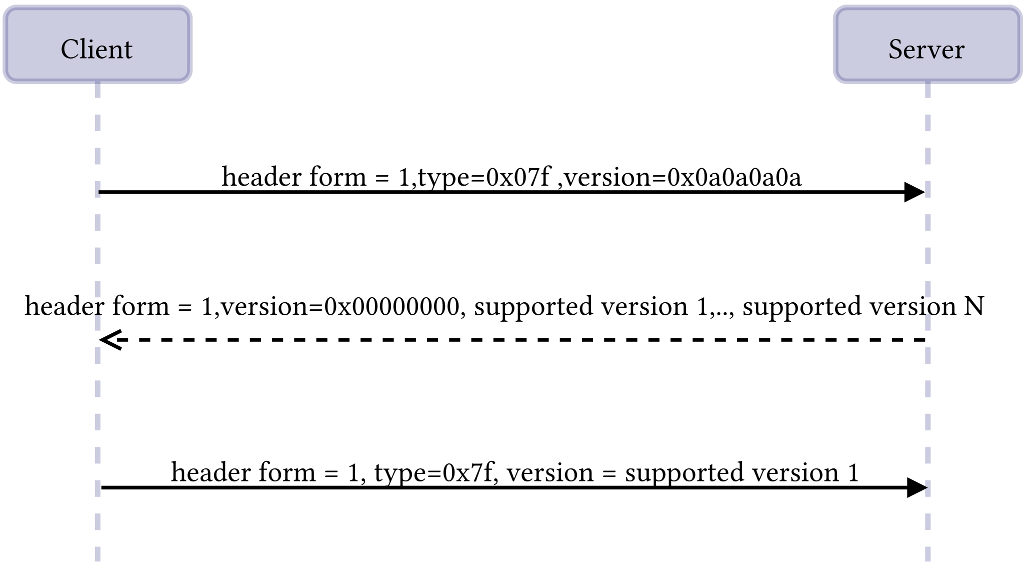

Client
Server
header form = 1,type=0x07f ,version=0x0a0a0a0a
header form = 1,version=0x00000000, supported version 1,.., supported version N
header form = 1, type=0x7f, version = supported version 1

Figure 3.4: Version Negotiation

The proposed sequence diagram for the packet exchanged for version negotiation between client and server is shown in the above figure 3.4.

- The client sends an initial packet with a reserved version *0x?a?a?a?a* to the server for connection establishment. This packet also carries the CRYPTO frame with *clienthello* message.
- The server on receiving the inital packet with a reserved version, sends a version negotiation packet. Version Negotiation packet is special packet which uses long header by setting the version field to 0x00000000. The server also sends a list of 32-bits versions it supports.
- The client from the list of versions, the one it supports and constructs a initial packet with the supported version. The new initial packet contains the initial cryptographic message for the handshake. The version negotiation packet is not acknowledged by ACK frame instead it is implicitly acknowledged when another initial packet is sent by the client.

Since version negotation packets are not encypted, it is possible to do validate this test case from wireshark capture of packets as well. This can be done by checking if the packet is long header and the version field is set to zero. This is possible because the long headers fields and version negotiation packet is not cryptographically protected.

Each implementation's development happens at a different pace ie., one implemetation can be much complete than other implementation. This makes the testing process difficult. Most of the QUIC implementations are not backward compatible. So if implementation A currently supports only draft-09 and implementation B supports draft-10, then they cannot be tested against each other as the version negotiation always fails. This is bound to happen when both the draft specification and implementation is continuously evolving.

3.4.2 Handshake

Cryptographic Handshake packet is used by the server and client to exchange cryptographic keys after agreeing on the QUIC version. The first cryptographic message is sent to the server from client in inital packet. The server responds with one or more handshake packet which contains cryptographic handshake message and acknowledgements. The cryptographic handshake can be present in either initial, retry or handshake packets and all these packets use long headers. This is followed by the client sending handshake packet to the server. Only CRYPTO frame is used for sending cryptographic handshake.

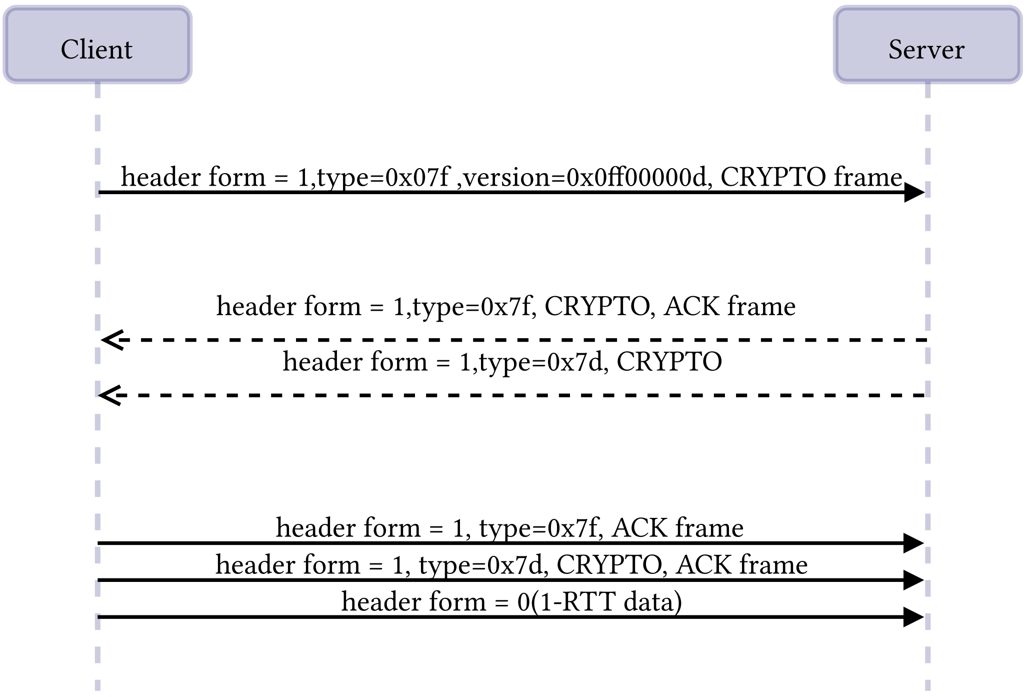

Client
Server
header form = 1,type=0x07f ,version=0x0ff00000d, CRYPTO frame
header form = 1,type=0x7f, CRYPTO, ACK frame
header form = 1,type=0x7d, CRYPTO
header form = 1, type=0x7f, ACK frame
header form = 1, type=0x7d, CRYPTO, ACK frame
header form = 0(1-RTT data)

Figure 3.5: Handshake

The exchange of QUIC packets to complete a handshake by the server and the client is depicted in the figure 3.5.

- The client sends an initial packet with a valid version which is supported by the server for connection establishment. This packet carries the CRYPTO frame with *clienthello* message.

- The server sends an initial packet with *serverhello* message along with the ACK frame for the previous initial packet sent by the client. The server also coalesce a handhshake packet in the same UDP datagram. This handshake packet carries a CRYPTO frame which contains encrypted extensions(EE), certificate(CERT), certificate verify(CV), finished(FIN) message.
- The client sends an ACK frame in initial packet as well as CRYPTO frame and ACK frame in handshake packet. The CRYPTO frame in handshake packet contains *fin* message and the ACK frame contains the acknowledgement for the previous handshake message from the server. The client can now send 1-RTT data as the shared key for the session is established.

When the above flow of packet is detected in QUIC log files then we can say that the QUIC cryptographic handshake is complete.

3.4.3 Stream Data

To create a stream and carry data STREAM frame is used. A single QUIC packet can have multiple STREAM frames belonging to one or more streams and thus stream multiplexing is achieved. A QUIC stream can be unidirectional streams or bidirectional streams. Each stream is individually flow controlled and the number of the streams that can be created is also controlled by the peer. This is negotiated by using MAX_STREAM_DATA and MAX_STREAM_ID frames.

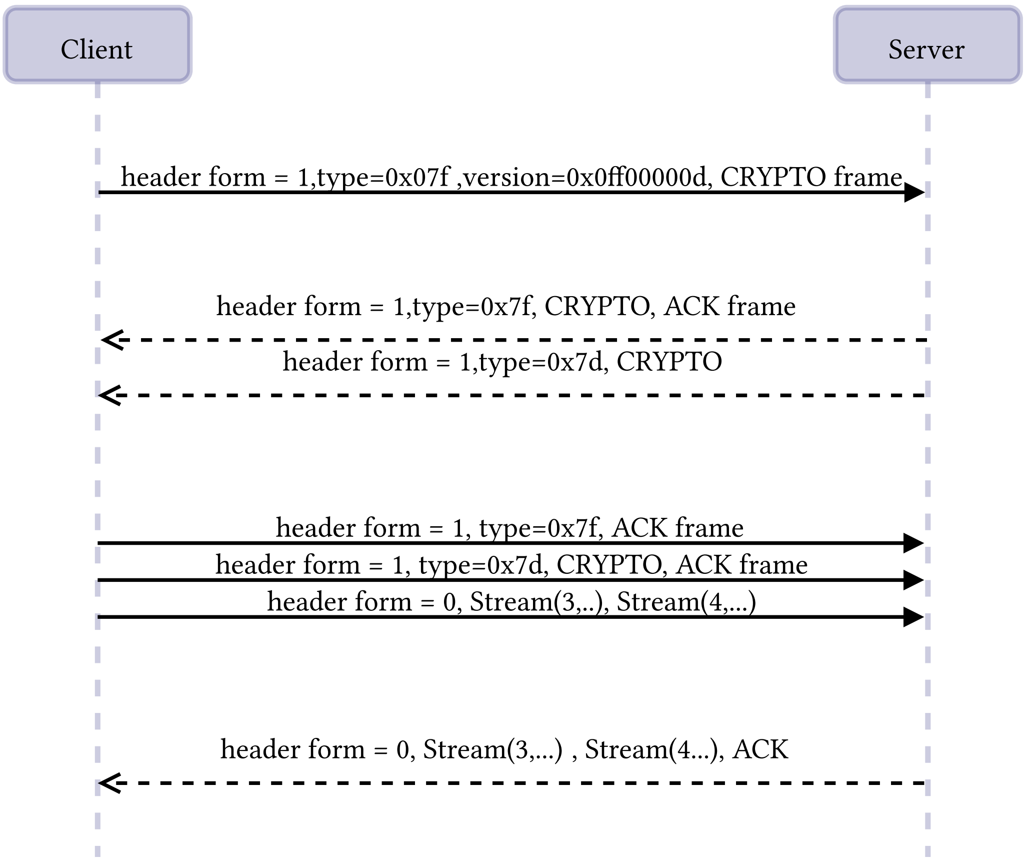

Client
Server
header form = 1,type=0x07f ,version=0x0ff00000d, CRYPTO frame
header form = 1,type=0x7f, CRYPTO, ACK frame
header form = 1,type=0x7d, CRYPTO
header form = 1, type=0x7f, ACK frame
header form = 1, type=0x7d, CRYPTO, ACK frame
header form = 0, Stream(3,..), Stream(4,...)
header form = 0, Stream(3,...) , Stream(4...), ACK

Figure 3.6: Multiple Streams

The sequence diagram for flow of packets to test multiple stream is shown in figure 3.6.

- The client sends an initial packet with a valid version which is supported by the server for connection establishment. This packet carries the CRYPTO frame with *clienthello* message.
- The server sends an initial packet with *serverhello* message along with the ACK frame for the previous initial packet sent by the client. The server also coalesce a handhshake packet in the same UDP datagram. This handshake packet carries a CRYPTO frame which contains EE, CERT, CV, FIN message.
- The client sends an ACK frame in initial packet as well as CRYPTO frame and ACK frame in handshake packet. The CRYPTO frame in handshake packet contains *fin* message and the ACK frame contains the acknowledgement for the previous handshake message from the server. The client can now send 1-RTT data as the shared key for the session is established. The 1-RTT data is sent in short header and the client opens two streams. The different streams are also coalesced into the same UDP packet.

- The server processes the requests in both the streams and sends back the appropriate data along with the ACK frame.

3.4.4 Connection Close

A QUIC connection once established can be terminated due to two different reasons. The first reason is that the server has served the client as expected and the client sends CONNECTION_CLOSE frame with NO_ERROR code. The other reason for connection close is when there is an error in the QUIC layer due to violation of protocol specification by any endpoint. An endpoint can close the entire connection even though only one stream caused the error. These two reasons causes immediate close of the connection. A connection can be terminated in one of the following three ways.

- **Idle Timeout:** is the value in seconds which is exchanged in transport Parameters during connection establishment by each endpoint. This is a mandatory field in the transport parameter and the maximum value is 600 second(10 min). If a connection remains idle longer than the idle timeout then it will be closed.
- **Immediate Close:** can be done by sending a CONNECTION_CLOSE frame or APPLICATION_CLOSE frame to terminate a connection immediately. If there are any open streams in the connection that are not explicitly closed , they are implicitly closed when a connection is closed. The connection close frame has a error code which says why the connection was closed. For example server sets the error code as 0x02 when the server is busy and closes the connection. 0x01 specifies it is an internal error and cannot continue with the connection. 0x00 says that the connection is closed abrubtly without any error.
- **Stateless Reset:** is the last option for an endpoint that do not have access to the connection state. An endpoint can lose the state of the connection due to crash or outage, but the peers continue to send messages which the endpoint cannot process. The stateless reset token is already shared by the server to the client in it's transport paramter. This token is protected and therefore only the client and the server knows this value. An endpoint which does not have the state of the connection and cannot process any packets sends a short header containing some random octets and the last 16 bytes contains the token. This makes it indistinguishable from a regular packet with a short header. The endpoint detects a stateless reset when a short header packet cannot be decrypted. Then, it compares the last 16 bytes to the stateless reset token in the server's transport parameter. If the values are same then the endpoint enters the draining state and does not accept further packet on that connection. If the comparison of the token failed then the packet is discarded.

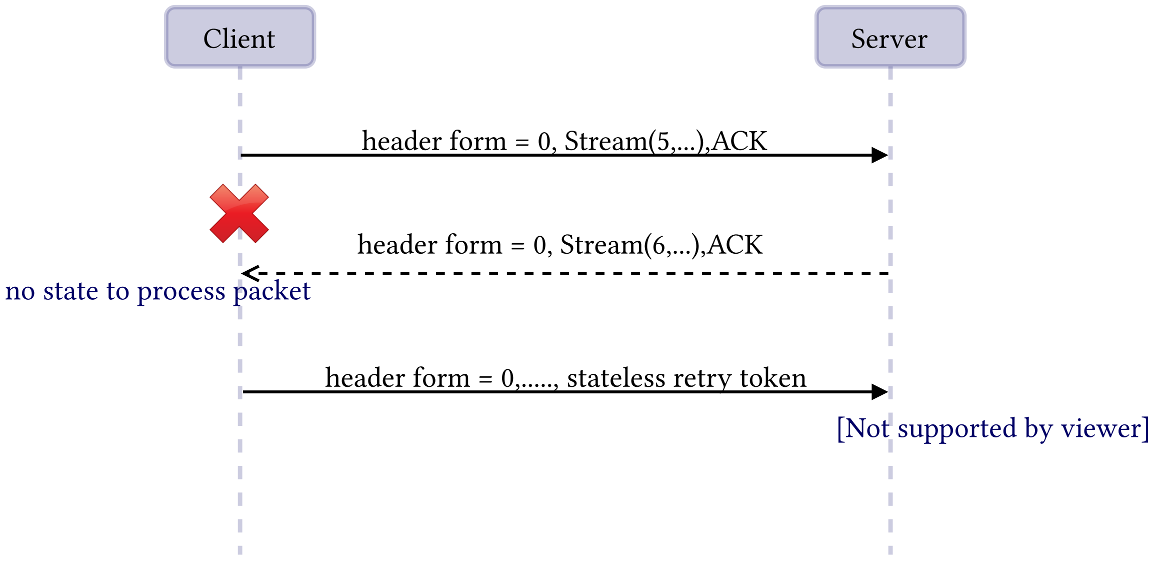

Client
Server
header form = 0, Stream(5,...),ACK
header form = 0, Stream(6,...),ACK
no state to process packet
header form = 0,....., stateless retry token
[Not supported by viewer]

Figure 3.7: Connection Termination through Stateless Reset

To identify connection close we need to depend on the client or the server log in particular to the streams that each packet contains. Even though some implementations logs the streams it received after decrypting, it is a complex to model. So, we present an idea on how to initiate a connection close but the implementation of this test scenario is skipped due to the time contraints.

3.4.5 0-RTT

QUIC supports 0-RTT packets which means that a client can send a data immediately following the handshake packet without waiting for a reply from the server. This can be done by first establishing a connection and then storing the transport parameters and session parameters which the server sent. This can later be used for resuming the session and sending 0-RTT packet.

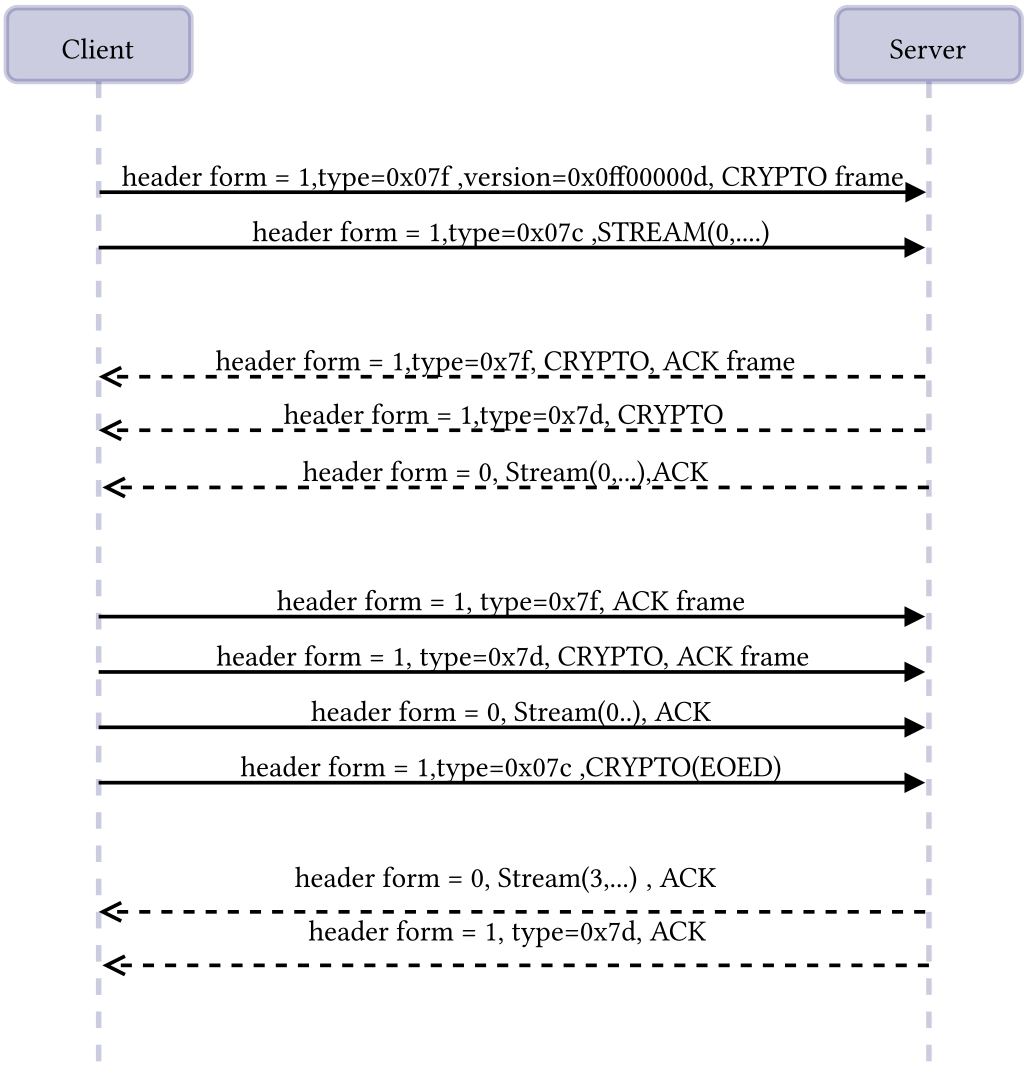

Client
Server
header form = 1,type=0x07f ,version=0x0ff00000d, CRYPTO frame
header form = 1,type=0x07c ,STREAM(0,....)
header form = 1,type=0x7f, CRYPTO, ACK frame
header form = 1,type=0x7d, CRYPTO
header form = 0, Stream(0,...),ACK
header form = 1, type=0x7f, ACK frame
header form = 1, type=0x7d, CRYPTO, ACK frame
header form = 0, Stream(0..), ACK
header form = 1,type=0x07c ,CRYPTO(EOED)
header form = 0, Stream(3,...) , ACK
header form = 1, type=0x7d, ACK

Figure 3.8: 0-RTT

The sequence diagram 3.9 depicts the sequence of packets exchanged in the second connection to send 0-RTT data.

- The client sends an initial packet with a valid version which is supported by the server for connection establishment. This packet carries the CRYPTO frame with *clienthello* message. This is used for 1-RTT key establishment in the later stage of the connection. Along with the initial packet, the client also sends 0-RTT packet(type 0x7c) with data in stream 0 to the server.
- The server replies back with the initial packet which has CRYPTO frame along with the ACK frame for client's initial packet. The CRYPTO frame contains *serverhello* message. The handshake packet carries the remaining cryptographic

paramterer in the CRYPTO frame. The EE, CERT, CV, FIN are carried in the CRYPTO frame. The 1-RTT packet contains the data for the clients query in the previous 0-RTT packet in STREAM frame and the server acknowledges the 0-RTT data in the 1-RTT encryption level. Note that the initial, handshake and 0-RTT packet are coalesced in a single UDP datagram.

- The client sends an ACK for the CRYPTO frame sent in initial packet and sends the CRYPTO frame with *fin* message and ACK frame for the handshake packet. The application data is sent in the 1-RTT protected packet. The client also sends an end of early data(EOED) message to convey that all 0-RTT data has been transmitted and the further data which it sends will be protected by the 1-RTT key established in this session.
- Both the parties can now communicate using the 1-RTT key established.

To test 0-RTT, we need to run our client program twice to establish a connection to the server. In the first connection, we save the tranport parameters and session parameters which is needed to connect to the same server next time. In the next subsequent connection we use the transport and session parameters to perform 0-RTT. When the above flow of packet is detected in QUIC log files in the second connection test then we can say that the QUIC 0-RTT works as expected.

3.4.6 Stateless Retry

A server can process the initial cryptographic handshake messages without commiting any state. This is done by the server to perform address validation on the the client or to avoid the connection establishment cost. To do statess retry the server sends the retry packet which is not encrypted in response the the client initial packet. This Retry packet has a long header with the type value 0x7E. The client reset its transport parameters but the remembers the state of the cryptographic handshake.

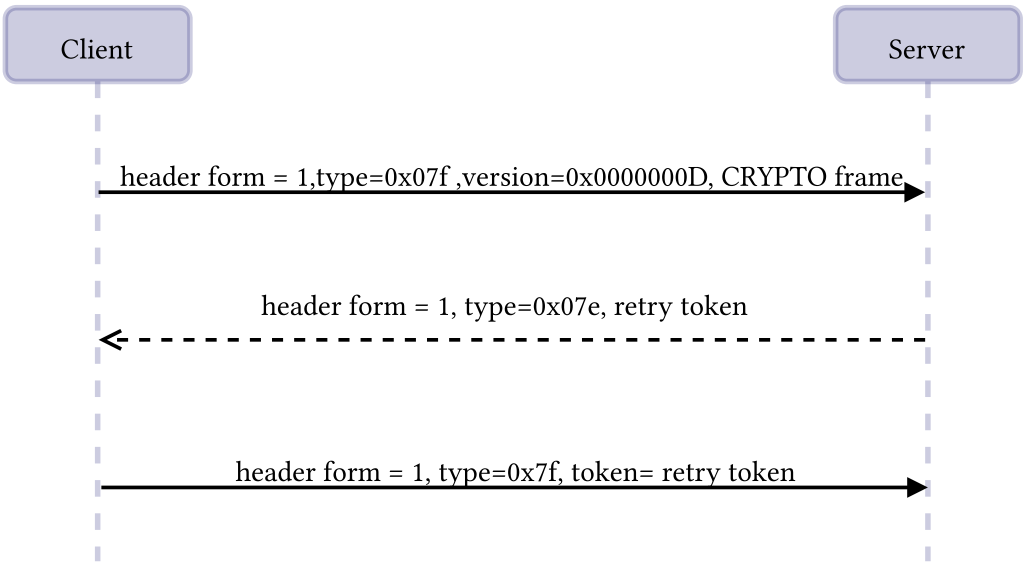

Client
Server
header form = 1,type=0x07f ,version=0x0000000D, CRYPTO frame
header form = 1, type=0x07e, retry token
header form = 1, type=0x7f, token= retry token

Figure 3.9: Stateless Retry

The sequence diagram in the figure 3.9 depicts the sequence of packets exchanged for stateless retry.

- The client sends an initial packet with a valid version which is supported by the server for connection establishment. This packet carries the CRYPTO frame with *clienthello* message.
- The server wishes to perform stateless retry to validate the source address. This helps in verifying that a client can recieve a packet to its claimed IP and port address and protects against spoofing the address by an attacker. The server sends back an retry packet with a retry token and the packet number of the retry packet should be set to 0.
- Only a legitimate client will receive the above retry packet and will respond with a client initial packet containing the token from the retry packet. After receiving a valid token, a server can abort the connection or proceed by sending a intial packet containing it's cryptographic parameters.

To test stateless reset, we need to set the server to perform stateless reset on clients who wants to establish an connection. This can be set by passing certain arguments to the executable. One such example is picoquic's implementation which enables server to perform stateless reset when the QUIC server is started with -r argument. Not all implementations has provided an interface to do this.

3.5 Summary

In this chapter, we discussed about the design challaneges in developing a framework for QUIC's interoperability testing. Later, we projected the conceptual model for running the test and the various ways to collect logs for validating the test cases. We discussed about the three design setup and listed out the advantage and disadvantage of using each setup. Finally, we list out some of the test scenrio to test the interoperability of QUIC implementations. The test scenario was explained briefly using sequence diagrams of the packet flow between the server and the client.

Chapter 4

Implementation

In the previous chapters we discussed about the working of QUIC protocol, automated testing tools and the importance and challenges of interoperability testing. We also discussed about the three design models: Lightweight, Virtualized and Physical Setup for setting up a test framework for QUIC's interoperability testing. In this chapter we will discuss about the implementation of our test framework by using one of the popularly used continuous integration tools and the plugins it offers, a log parser which is used to parse the test logs and validate the test cases for lightweight design model.

4.1 Continuous Integration Server Setup

Of all the continuous integration tools available, we need one that supports a wide variety of source control management(SCM) also called as version control management(VCM). SCM is an essential part of the CI system. Jenkins is a popular CI tool which supports most of the SCM, so it was clearly a better choice than any other CI tool. It helps in scalability of the framework as follows. Suppose there is new implementation in the future which uses less popularly used SCM like BitKeeper or Surround, they are not supported by the other commonly used continuous integration tools like Bamboo or Teamcity [15]. After analysing the tools in the market, Jenkins was choosen because of the following reasons

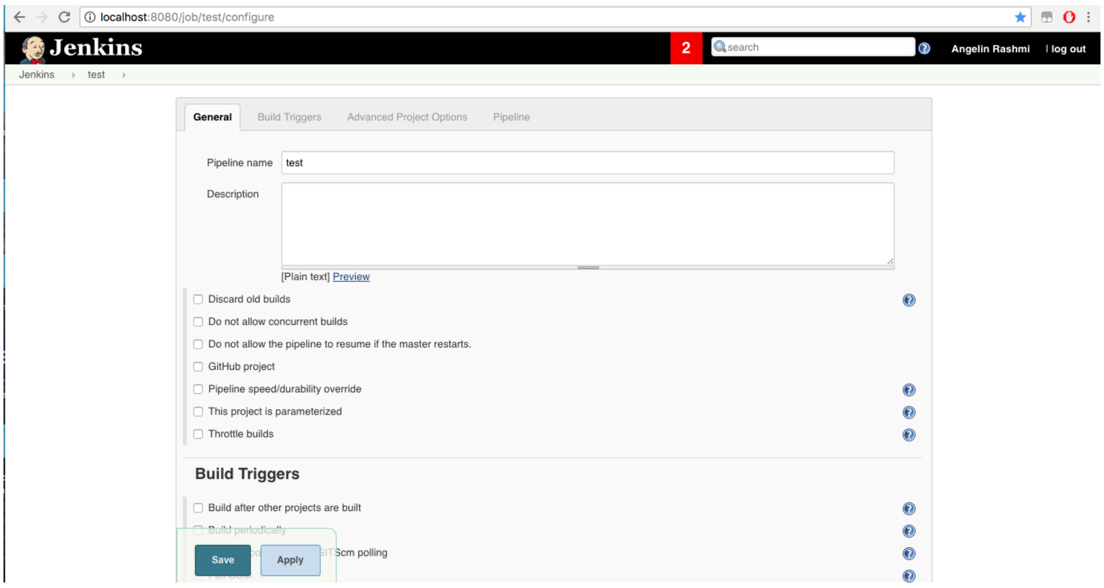
- It is is platform independent, Java-based program, with the possibility to run on windows, Mac OS X and any other Unix like operating system.
- It is easy to install and configure using their web graphical user interface. It requires little maintenance and plugin updates can be done through the same web interface.
- It is licenced under MIT and therefore it is free to use and distribute.

- The tool is very flexible because of feature extensions through plugins. There are over 400 plugins to support every aspect of software development life cycle with good documentation. It is also possible to create new plugins to customize for a particular scenario.
- It is possible to distribute builds or tests to multiple machines which are running under different operating systems.
- It also has some reporting features about the job and it's status ie., you can customize to send email or notify using pop-ups when the job fails or completes successfully.

In the following sections we will discuss the four phases of the test interoperability test framework in detail and how they help us in achieving the tasks.

4.2 Build Job Setup

Once Jenkins is installed, we need to create a build job which builds and compiles all the open source implementation, runs the test scripts and generates client logs. Later, the logs are processed and the interoperability matrix is generated. This is all defined in the Jenkinsfiles and this will be pulled from the Git repository. In the Jenkins dashboard, a new pipeline project is created and the repository for the framework is given in the advanced project option. Exact instructions on how to setup the job can be found in the jenkins homepage or in the readme section of the source code [16]. Figure 4.1 shows the web interface used to setup a Jenkins pipeline job.



Pipeline name: test

Description:

[Plain text] Preview

☐ Discard old builds
☐ Do not allow concurrent builds
☐ Do not allow the pipeline to resume if the master restarts.
☐ GitHub project
☐ Pipeline speed/durability override
☐ This project is parameterized
☐ Throttle builds

Build Triggers

☐ Build after other projects are built
☐ Build periodically
☐ GITScm polling

Figure 4.1: Pipeline Job Setup

Before we proceed to look into each phases in the framework, we need to understand about pipeline plugin which drives the entire framework's flow.

Pipeline Plugin is one of the important plugins which is used to develop the framework. This plugin helps to automate the process of getting the source code of each implementation from the version control system to publishing the interoperability matrix through a set of different plugins.

This plugin supports two syntaxes, declarative pipeline and scripted pipeline. Declarative pipeline syntax is more strict with pre-defined structure and it is used for simple continuous delivery process. We use scripted pipeline syntax for our framework because it is more flexible and extensible. The definitions of Jenkins Pipeline is written to a file called Jenkinsfile and is checked into the framework repository. Jenksinsfile drives all the phases of the testing framework. The stages that we designed are show in figure 4.2 and later explained in detail in the next sections.

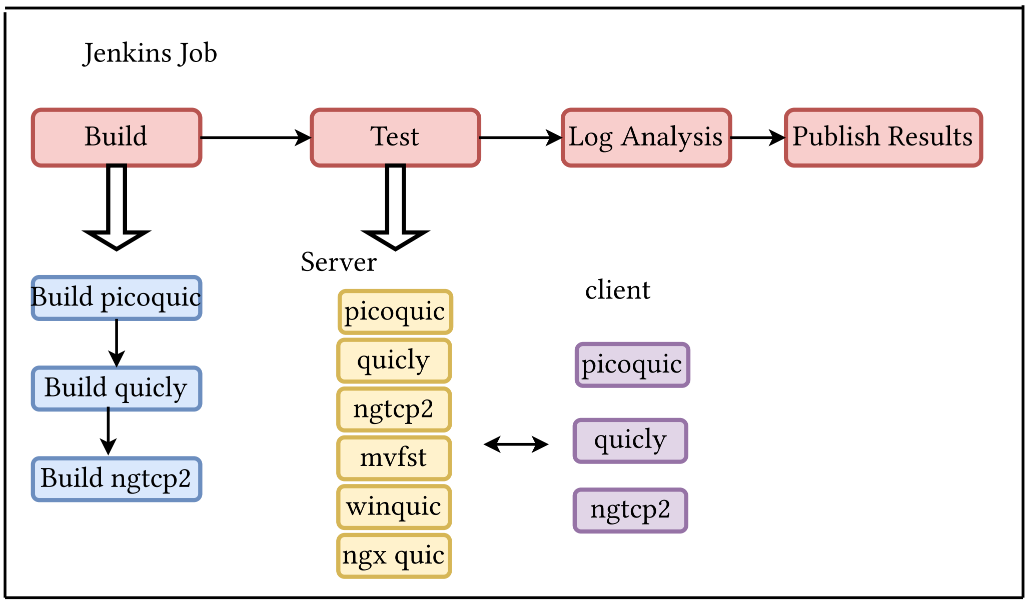


Figure 4.2: Jenkins Stages

4.3 Build Phase

This phase includes a stage for each implementation of QUIC for which we have access to the repository using a SCM. Each stage has list of tasks that conceptually should be run together. In the below step we can see the stage for picoquic implementation. The tasks for the stage is checking out the source code of the master branch from the repository, cmake and make for building the project. These tasks depends on the implementation language and so it might differ for each implementation.

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false,
extensions: [[$class: 'SubmoduleOption', disableSubmodules: false,
parentCredentials: true, recursiveSubmodules: true,
reference: '', trackingSubmodules: false]], submoduleCfg: [],
userRemoteConfigs:[[credentialsId:'8607f52e-a446-45e9-b4d1-02bc8ef0ab8b',
url: 'https://github.com/private-octopus/picoquic.git']]])
cmakeBuild 'InSearchPath'
sh 'make'
```

Source Code 4.1: Build stage of picoquic

timeout When two implementations communicate with each other, there are times when they don't interoperate or the servers are not reachable. The clients continue to send initial packet eventhough the server doesnt respond. We need a mechanishm to detect this and stop the client, otherwise it runs to an infinite loop where it keeps sending packets. Jenkins has a mechanism to detech this issue a timeout after certain seconds. timeout can be set for a block of code and if the time limit is reached an exception is thrown and aborts the step

```
try{
    timeout(time: 10, unit: 'SECONDS') {
        sh './client msquic.westus.cloudapp.azure.com 4433
        &> ../winquic/ngtcp2-client.log'
    }
} catch(err) {
    echo "Error"
}
```

Source Code 4.2: Example of timeout step

GIT Plugin This plugin helps to use Git as a build Source Control Management(SCM). We can configure this plugin with a Repository URL which is same as the syntax which we use in git clone command. There is also way to clone submodules using this plugin by setting *recursiveSubmodules* field to true. The other two inputs are the Credentials and Branches. Jenkins support various version control systems like Mercurial, Subversion etc and so based on the system which the implementation uses, we have to install the particular plugin and use it.

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false,
extensions: [[$class: 'SubmoduleOption', disableSubmodules: false,
 parentCredentials: true, recursiveSubmodules: true,
reference: '', trackingSubmodules: false]], submoduleCfg: [],
userRemoteConfigs:[[credentialsId:'8607f52e-a446-45e9-b4d1-02bc8ef0ab8b',
url: 'https://github.com/private-octopus/picoquic.git']]])
```

Source Code 4.3: Usage of GIT plugin

Pipeline Nodes and Process plugins provides a solution to run a shell script as a task. This is needed to run the appropriate QUIC client or server for testing. dir step changes the current working directory.

4.4 Test Phase

Once the build phase is complete, we have the excutable for QUIC server and client for each implementation. Some implementation provides with only one executable for server and client. The functionality of the executable changes based on the command line arguments. Since it is an interoperability test we need to run each implementation against every other implementation. The server is first run as a background process and we run each client against the server and record the logs. If the client runs for more than 10s we abort the current task and proceed with the next one as discussed before using timeout.

```
stage('Picoquic as Server'){
    dir('./picoquic') {
        sh './picoquicdemo -p 4444 &'
        sh './picoquicdemo -l picoquic-client.log  localhost 4444'
        sh 'cat picoquic-client.log'
        dir('../quicly'){
            try{
                timeout(time: 2, unit: 'SECONDS') {
                    sh './cli -p /logo.jpg -v 127.0.0.1 4444
                    &> ../picoquic/quicly-client.log'
                    sh 'cat ../picoquic/quicly-client.log'
                }
            }
            catch(err) {
                    echo "Error in running picoquic"
                }
        }
        dir('../ngtcp2'){
            try{
                timeout(time: 10, unit: 'SECONDS') {
                    sh './examples/client 127.0.0.1 4444
                    -d ../picoquic/http09_index.html
                    &> ../picoquic/ngtcp2-client.log'
                    sh 'cat ../picoquic/ngtcp2-client.log'
                }
            } catch(err) {
                echo "Error in running ngtcp2"
            }
        }
        sleep time: 100, unit: 'MILLISECONDS'
        sh 'kill $(lsof -t -i :4444)'
        }
    }
```

Source Code 4.4: Test stage for picoquic as server

4.5 Log Analysis Phase and Publish Result

After test phase, we have to analyse the logs that we received from the test, validate the test case and then document the result in interoperability test matrix. Even though logs can provide various insights in analysing about what is happening between the client and the server, there are lot of challenges in extracting valuable information from them. Most of the QUIC implementations does not provide server log. So we had to design a way to validate a test case by looking at the client logs only. This is not a major roadblock as it is possible to get a lot of information by looking only at the client logs.

There is a need for a log parser for each implementation. This is because the QUIC log's are not standardized and it differs from implementation to implementation. Apart from the log parser for each implementation, we need a way to validate the output of log parser and assert if the test cases passed or failed. This is done by developing a state transition matrix containing the acceptable transitions based on QUIC protocol's specification.

4.5.1 State Transition Diagram and Matrix

To verify whether a test case passed or not, we need to verify that the sequence of packets exchanged between the client and the server follows the QUIC standard. QUIC standard is quite complex and we have to find a way to describe the behavior of the QUIC protocol. State diagrams are the way to describe the behavior of the system when there is only one object is invloved, in our case it is only the client log. It is useful in depicting an event-driven system and shows how the object moves through various stages in its lifetime.

A state diagram can be used only when there is a finite number of states in the system. The following are the 7 states that we have come up with after analyzing the QUIC standard **Client Initial State**, **Stateless Retry State**, **Handshake State**, **1-RTT State**, **0-RTT State**, **Version Negotiation State** and **Error State**. The system receives event and causes the machine to move from one state to another state. Figure 4.3 shows the state transition diagram that we have designed after reading the IETF QUIC specification document. This helps in understanding the overall packet flow in QUIC protocol.

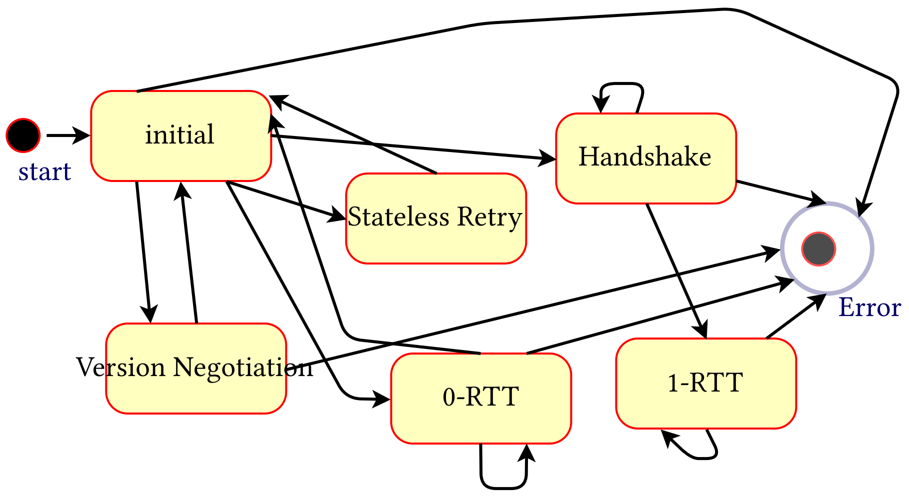


Figure 4.3: State Transition Diagram for IETF QUIC

In the above diagram, the initial state is represented as a single circle and states are represented with rounded rectangles that are labeled with the state name. Transitions are marked with arrows that move from one state to the another based on the content in the processed packet. A double circle terminates the process ie., it has reached the end state.

The state diagrams cannot be used directly in the log parser to validate the test results. We have to represent the pictorially represented state transistion diagram as a state transition matrix so that the log parser can use it. We have used 2D array data structure in python to represent our transition matrix.

```
transition=[ [0,1,1,0,1,0,0], #Client Initial state
                  [1,0,0,0,0,0,0], #Stateless Retry state
                  [0,0,1,1,0,0,0], #Handshake State
                  [0,0,0,1,0,0,0], #1-RTT State
                  [0,1,0,0,1,0,0], #0-RTT State
                  [1,0,0,0,0,0,0], #Version Negotation State
                  [0,0,0,0,0,0,0] ] #Error State
```

Source Code 4.5: 2D data structure to represent state transition matrix

We have a $7X7$ matrix, where 7 is the number of states that we defined in the state diagram. The row represents the current state and the column represents the future state. Based on the action ie., the type of packet received, we consult the transition matrix which has two values 0 or 1. 0 means that the current state transition is not allowed and so the test case fails. 1 means the transition is valid and it can proceed.

4.5.2 Picoquic Parser

In this section, an idea of how to write a parser for a log file is described. Below is the sample client log of picoquic.

```
   Sending 1252 bytes to 127.0.0.1:4444
 at T=0.000000 (56bff0dda8c7c)
    Type: 2 (client initial), S0, Version ff00000b, <dd9ab640b94d191d>,
     <567a2719584b44fa>, Seq: 8b62cb, pl: 1224
    Decrypted 1208 bytes
    Stream 0, offset 0, length 246, fin = 0: 16030300f1010000...
    Padding, 958 bytes

Select returns 1252, from length 16
Receiving 1252 bytes from 127.0.0.1:4444
 at T=0.016444 (56bff0ddaccb8)
    Type: 4 (handshake), S0, Version ff00000b, <567a2719584b44fa>,
     <6e05828ed4e0e99a>, Seq: 5df0b673, pl: 1224
    Decrypted 1208 bytes
    ACK (nb=0), 8b62cb
    Stream 0, offset 0, length 1198, fin = 0: 160303008b020000...
```

Source Code 4.6: Picoquic's sample log

From the log, we can see that picoquic logs the each packet it send as well as receives in a particular format with the number of bytes it received or sent in the first line, the time at which it received or sent the packet in the second line and in the third line it logs the type of packet it received. This is very important for parsing and we have to find a mapping to this type and the states which we defined before. Table 4.1 shows the mapping which we have defined for picoquic.

PICOQUIC TYPE	STATE
Type: 2	Client Initial State
Type: 3	Stateless Retry State
Type: 4	Handshake State
Type: 6	1-RTT State
Type: 5	0-RTT State
Type: 1	Version Negotiation State
Type: 0	Error State

Table 4.1: Mapping Picoquic Packet Types to the States

Once the mapping is done, the validity of the action is checked using the state transition matrix and test case is declared pass or fail. In order to write a new parser for another implementation we have to analyze the logs it generates by running it manually and define a mapping similar to the table 4.1. This is possible only for the implementations which logs the packets which it sends or receives. For implementations which does not follow any logging mechanism, the framework will not be able to validate the tests.

4.5.3 Pandas

Pandas is a open source python library for data manipulation and analysis. It also provides easy to use data structures for processing numerical tables, time series etc., [17]. Dataframe is a two dimensional data structure which has labeled rows and columns. The column's data can be of different type as well. We use dataframes to store the test results for each client. In the below source code, the DataFrame is constructed from the dict of Series. Each series is labelled with the client implementation name and the resilts of the test cases are passed.

```python
    import pandas as pd
    d = {'picoquic' : pd.Series(picoquic_result),
                'ngtcp2' : pd.Series(ngtcp2_result),
                'quicly' : pd.Series(quicly_result)}
        df = pd.DataFrame(d)
```

Source Code 4.7: Test Result stored in Dataframe

One of the main reason for using pandas library is that it also allows visual styling of the DataFrames [18]. This can be done with the help of *Dataframe.style* property. This is useful for formatting and displaying the test results as matrix in interoperability test report ie., it provides CSS classes to the data. *Dataframe.style* property returns a Styler object and this object can be used to render it to HTML by using *.render()* method. The limitations of styling in pandas are, style can be applied only to the values not to the columns or indexes and there is no possibility to insert new HTML entitites.

```python
1   def color_zero_red(val):
2           color = 'red' if val.find(">0<") is not -1 else 'green'
3           return 'background-color: %s' % color
4
5   styles = [ dict(selector="th", props=[("text-align", "center")]),
6           dict(selector="", props=[("margin", "50px"),
7                                    ("display","inline-block")]) ]
8   dfWithStyle = df.style.applymap(color_zero_red)
9                  .set_caption(implementation_name+' as Server')
10                 .set_properties(**{'border': '2px solid black'})
11                 .set_table_styles(styles)
12                 .render()
```

Source Code 4.8: Panda's Style for displaying Interoperability matrix

In line 8, *applymap* function is used for applying styling elementwise. It takes a function as an argument and applies the style to the entire dataframe. In our example the function *color_zero_red* takes each element of the dataframe sets the background color red if the value is zero else the background color is green by returning the CSS attribute and value pair.

In line 9, *set_caption* adds a regular caption for the table and this does not depend on the data as in the previous function ie., they are non data driven functions. *set_property* is also one such function used when the style does't depend on the values.

In line 11, *set_table_styles* is used for applying Table Styles. They are applied to the entire table as a whole. *table styles* are list of dictionaries which are defined in line 5 and 6. Each dictionary as a CSS *selector*, a pattern used to select the elements which we want to style and *props* keys. The value for props is a list of tuples *('attribute','value')*. Later the string *dfWithStyle* will be writen to a HTML file for publishing. The main flow logic of log parser program is as follows:

```
for server_index in range(0,number_of_implementations):
        parsepicoquic(implementation_name[server_index])
        parsengtcp2(implementation_name[server_index])
        parsequicly(implementation_name[server_index])
        drawtable(implementation_name[server_index],result)
```

Source Code 4.9: Main loop of the Parser

For each implementation the corresponding function parses the log generated and populates the test result in an array. It is very easy to add new implementation to the log parser. It can be done by adding the corresponding parser specific to it's log file. Finally the interopertbilty test results are published in the Jenkins dashboard using the below mentioned plugin.

4.5.4 HTML Publisher Plugin

This is used for publishing HTML reports that either the build task or test task generates. It publishes only static HTML page and all the CSS is disabled by default due to content security policy in Jenkins. Since we have used some CSS using pandas for the resulting interoperability test report, it has to be enabled by typing the following command in Jenkins->Script Console

```
System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "")
```

Source Code 4.10: Command to enable styling

Once the pipeline job completes there will be a link to the interoperability test report in the jenkins dashboard and it is available for viewing.

```
publishHTML([allowMissing: false, alwaysLinkToLastBuild: false,
includes: '**/*.log,**/*.html', keepAll: true, reportDir: '.',
reportFiles: './framework/result.html',
reportName: 'InterOperability Report'])
```

Source Code 4.11: Usage of HTML Publisher plugin

- *allowMissing* if not set fails the build if the report is not generated/missing.
- *alwaysLinkToLastBuild* publishes the link on the project level even if the build failed.
- *KeepAll* if the field is set then all the reports are archived for the successful builds, otherwise they are deleted and only the most recent one is available.
- *reportDir* This should be the path of the HTML report directory relative to the Jenkins workspace.
- *reportFiles* The file in the report directory for which the links are provided.
- *reportName* The name of the report to display for the build/project

The above described implementation of the interoperability test framework helps in overcoming the challenges described in chapter 2. Pair-wise testing of implementation is made possible by use of Jenkins's pipeline script. It is also easy to add a new implementation as the script is more structured. The scalability of test scenarios can be handled easily in the jenkins script as well as in the log parser by adding the appropriate function for parsing the implementation's log file. Jenkins HTML publisher plugin and the pandas styling library provides a way to embed the log files as a link in the interoperability test matrix. This provides all the details for manually analyzing the root cause of the test fail.

4.6 Source Code

The source code for the QUIC automation interoperability test framework can be found at *https://github.com/angelinrashmi2000/QUIC-TEST-FRAMEWORK*. The instructions available are for setting up a Jenkins server locally. The list of needed Jenkins plugins for running the framework is available in the README section. Further details to creating pipeline job and building the project can also be found in detail.

4.7 Summary

In this chapter, we discussed about the implementation of the lightweight design approach of the framework presented in Chapter 3. We discussed about setting up the continuous integration tool called Jenkins which forms the backbone of our framework. The important plugins needed for the framework is also presented. We also discussed about the log parser component and the state transition model for QUIC protocol specification which helps us in validating the test case. In the next section we will see how this components are actually used for evaluation.

Chapter 5

Evaluation

In the previous chapters, we designed and implemented the various components needed for the interoperability testing framework. In this chapter, we will evaluate the lightweight system design described in section 3.2.1.

5.1 Experimental Setup

In the lightweight system design, we saw that both the client and server will be run as user processes in the same machine. We chose this approach for testing our proof of concept as it was cost-effective. For our evaluation, we need to set up the Jenkins server on the target machine. The minimum system configuration needed for setting up a Jenkins server is described below

- **Memory:** at least 2GB RAM
- **Disk Space:** depends on how many latest build history Jenkins has to maintain. This count can be configured on the job configuration page.
- **Operating System:** Windows, Mac OS X, Unix based OS.
- **JDK:** 1.5 or above

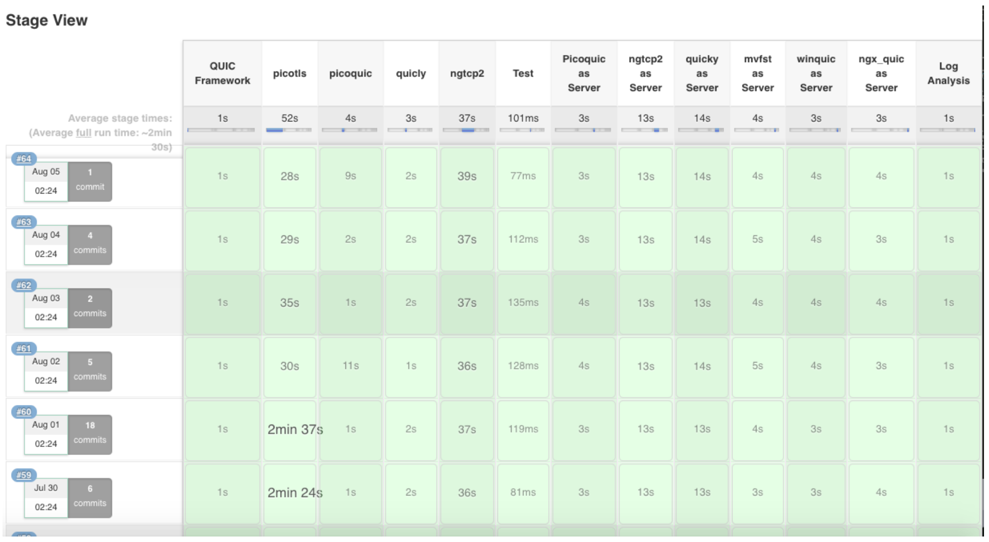
For the evaluation, the test setup was run on a MAC OS X with 4 GB RAM and with more than 32 GB disk space. We installed the necessary library and tool needed for building the implementations. We decided to use three open source QUIC implementations and they are *picoquic*, *ngtcp2*, *quicly* to evaluate our framework design. Apart from that, we selected three other public QUIC servers for which we do not have access to the source code and they are *mvfst*, *ngx_quic*, *winquic*. This selection helps in covering the two possible scenarios that we discussed in section 3.1. Table 5.1 defines the role which the implementations take in testing.

Implementation Name	**Client**	**Server**
picoquic	yes	yes
ngtcp2	yes	no
quicly	yes	yes
mvfst	no	yes
winquic	no	yes
ngx_quic	no	yes

Table 5.1: Role of Implementations

5.2 Test Against Open Source QUIC Server

Initially, in the build phase, the three implementations were pulled from their current repository. All the three implementations use CMake to manage the build process and make to generate executables from the source files. The entire job fails even if one of the build processes fails. It is possible to restart the job once the build error is corrected. It is also possible to checkout the previous version in the GIT for the failing process. This will be needed in the case where a new commit makes the build fail. Once the build phase is complete, the test phase begins where both the implementations *picoquic* and *quicly* are run as a server and tested for interoperability against each other and *ngtcp2*. Failure of a test does not stop the build process, it aborts the current test and proceeds with the next one.


Stage View

	QUIC Framework	picotls	picoquic	quicly	ngtcp2	Test	Picoquic as Server	ngtcp2 as Server	quicky as Server	mvfst as Server	winquic as Server	ngx_quic as Server	Log Analysis
Average stage times: (Average full run time: ~2min 30s)	1s	52s	4s	3s	37s	101ms	3s	13s	14s	4s	3s	3s	1s
#64 Aug 05 02:24 1 commit	1s	28s	9s	2s	39s	77ms	3s	13s	14s	4s	4s	4s	1s
#63 Aug 04 02:24 4 commits	1s	29s	2s	2s	37s	112ms	3s	13s	14s	5s	4s	3s	1s
#62 Aug 03 02:24 2 commits	1s	35s	1s	2s	37s	135ms	4s	13s	13s	4s	4s	4s	1s
#61 Aug 02 02:24 5 commits	1s	30s	11s	1s	36s	128ms	4s	13s	14s	5s	4s	3s	1s
#60 Aug 01 02:24 18 commits	1s	2min 37s	1s	2s	37s	119ms	3s	13s	13s	4s	3s	3s	1s
#59 Jul 30 02:24 6 commits	1s	2min 24s	1s	2s	36s	81ms	3s	13s	13s	3s	3s	4s	1s

Figure 5.1: Stage View

From the figure 5.1, we can see that the build process for each implementation takes less than 40 seconds and this shows that even if the number of implementation increases, the time taken for the entire build phase to finish will be much less that manually compiling the projects. Also, the test phase takes much less time to complete and overall this proves that this lightweight system design for automating the interoperability test saves a lot of time. This time can be further reduced by parallelizing the tasks across multiple Jenkins slave.

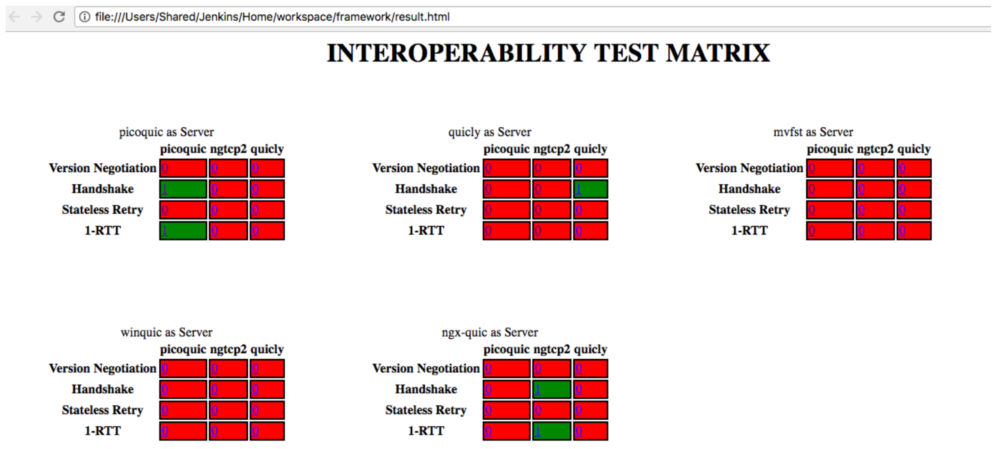
5.3 Test Against Public QUIC Server

There is no build phase for implementations *mvfst*, *ngx_quic*, *winquic* as they are not open sourced. The developers instead run QUIC servers on public IP address which can be used for testing interoperability. The table 5.2 below shows the hostname and port number for all the three implementations. The public QUIC server listens to the port and serves all the clients which want to communicate with it. Note that it is not possible to run these implementations as client as the developers of these implementations do not provide access to the executables of the client.

Implementation Name	**Hostname**	**Port Number**
mvfst	fb.mvfst.net	4433
ngx_quic	quic.tech	4433
winquic	msquic.westus.cloudapp.azure.com	4433

Table 5.2: Details of Public QUIC Server

In the test phase the three implementations are run as a server and the other three implementations *picoquic*, *ngtcp2* and *quicly* run as a client and the logs are saved for parsing and validating the results.


file:///Users/Shared/Jenkins/Home/workspace/framework/result.html

INTEROPERABILITY TEST MATRIX

picoquic as Server

	picoquic	ngtcp2	quicly
Version Negotiation	🟥 0	🟥 0	🟥 0
Handshake	🟩 1	🟥 0	🟥 0
Stateless Retry	🟥 0	🟥 0	🟥 0
1-RTT	🟩 1	🟥 0	🟥 0

quicly as Server

	picoquic	ngtcp2	quicly
Version Negotiation	🟥 0	🟥 0	🟥 0
Handshake	🟥 0	🟥 0	🟩 1
Stateless Retry	🟥 0	🟥 0	🟥 0
1-RTT	🟥 0	🟥 0	🟥 0

mvfst as Server

	picoquic	ngtcp2	quicly
Version Negotiation	🟥 0	🟥 0	🟥 0
Handshake	🟥 0	🟥 0	🟥 0
Stateless Retry	🟥 0	🟥 0	🟥 0
1-RTT	🟥 0	🟥 0	🟥 0

winquic as Server

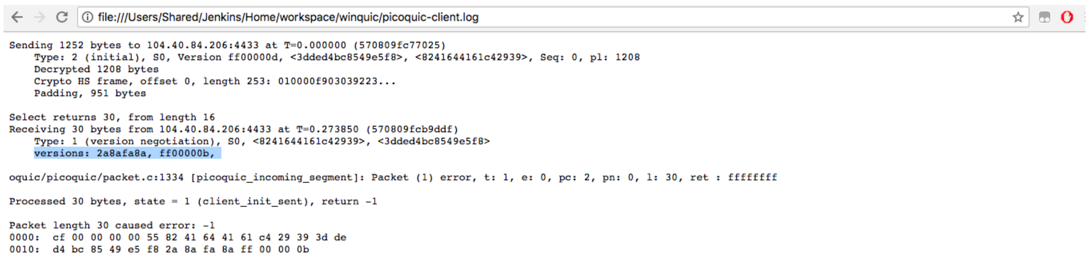
	picoquic	ngtcp2	quicly
Version Negotiation	🟥 0	🟥 0	🟥 0
Handshake	🟥 0	🟥 0	🟥 0
Stateless Retry	🟥 0	🟥 0	🟥 0
1-RTT	🟥 0	🟥 0	🟥 0

ngx-quic as Server

	picoquic	ngtcp2	quicly
Version Negotiation	🟥 0	🟥 0	🟥 0
Handshake	🟥 0	🟩 1	🟥 0
Stateless Retry	🟥 0	🟥 0	🟥 0
1-RTT	🟥 0	🟩 1	🟥 0

Figure 5.2: Interoperability test matrix

In figure 5.2, we can see the result of all the interoperability tests. It displays in green color when the test passes successfully else it is in red color. It is possible to click the links on the result to see the corresponding client logs. This can be used for analyzing the cause of the test failure. In the figure 5.2, all the test cases fail when *mvfst* and *winquic* is run as server because both the implementation is still supporting old version. This is found by analyzing the log file shown in figure 5.3. In the packet received from the QUIC server, we can see that the packet is a version negotiation packet and the versions it supports are *2a8afa8a, ff00000b*. These versions cannot be processed by the QUIC client as it supports only version *ff00000d* and it is not backward compatible.


file:///Users/Shared/Jenkins/Home/workspace/winquic/picoquic-client.log

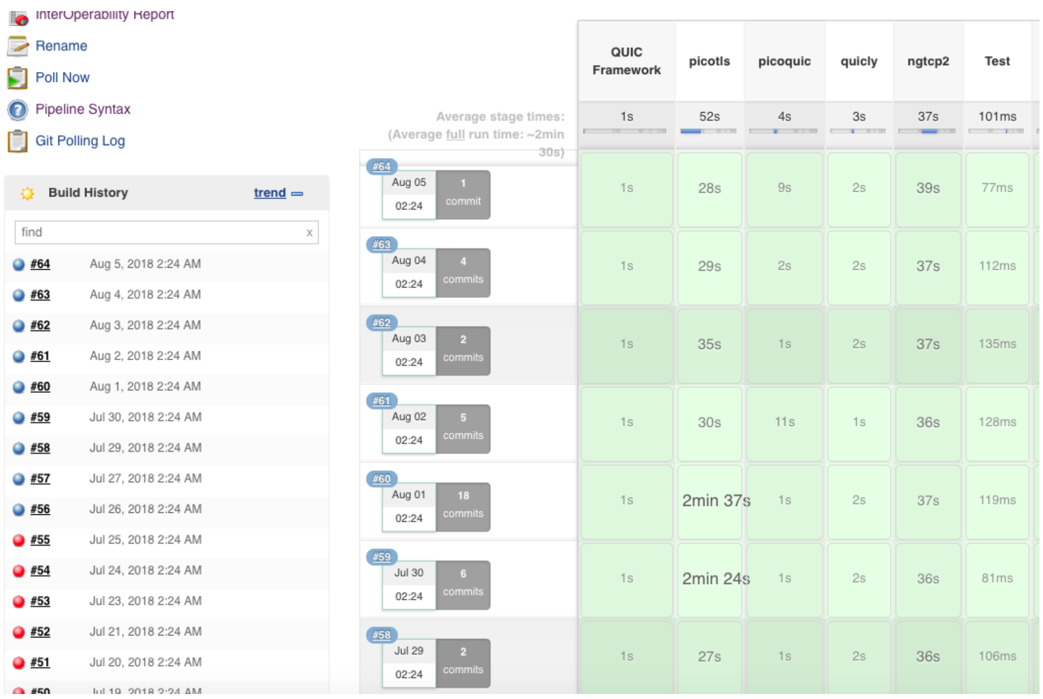
```
Sending 1252 bytes to 104.40.84.206:4433 at T=0.000000 (570809fc77025)
    Type: 2 (initial), S0, Version ff00000d, <3dded4bc8549e5f8>, <8241644161c42939>, Seq: 0, pl: 1208
    Decrypted 1208 bytes
    Crypto HS frame, offset 0, length 253: 010000f903039223...
    Padding, 951 bytes

Select returns 30, from length 16
Receiving 30 bytes from 104.40.84.206:4433 at T=0.273850 (570809fcb9ddf)
    Type: 1 (version negotiation), S0, <8241644161c42939>, <3dded4bc8549e5f8>
    versions: 2a8afa8a, ff00000b,

oquic/picoquic/packet.c:1334 [picoquic_incoming_segment]: Packet (1) error, t: 1, e: 0, pc: 2, pn: 0, l: 30, ret : ffffffff

Processed 30 bytes, state = 1 (client_init_sent), return -1

Packet length 30 caused error: -1
0000:  cf 00 00 00 00 55 82 41 64 41 61 c4 29 39 3d de
0010:  d4 bc 85 49 e5 f8 2a 8a fa 8a ff 00 00 0b
```

Figure 5.3: Sample of Log File

When the tests are run with ngx_quic implementation as server, ngtcp2 successfully completes handshake and 1-RTT where as picoquic fails because it support only *ff00000d* where as ngtcp2 and ngx_quic supports version *ff00000c*

5.4 History of Interoperability Test Matrix

The evolution of interoperability test can be seen from the build history in the Jenkin's dashboard. The job is scheduled to run every day at midnight. This can be set in the job configuration page and it can also be triggered by a git commit to the test framework. In figure 5.4, we can see the list of jobs that had run. Once you click the job number, we can see the logs of the build process, test process and the generated interoperability matrix. This helps us to understand how the interoperability test changes everyday based on the new commits from different implementations. It helps the developers to keep track of their implementation's interoperability with other QUIC implementations. Thus it helps in saving a lot of time by automating the entire process.



🔵	#64	Aug 5, 2018 2:24 AM
🔵	#63	Aug 4, 2018 2:24 AM
🔵	#62	Aug 3, 2018 2:24 AM
🔵	#61	Aug 2, 2018 2:24 AM
🔵	#60	Aug 1, 2018 2:24 AM
🔵	#59	Jul 30, 2018 2:24 AM
🔵	#58	Jul 29, 2018 2:24 AM
🔵	#57	Jul 27, 2018 2:24 AM
🔵	#56	Jul 26, 2018 2:24 AM
🔴	#55	Jul 25, 2018 2:24 AM
🔴	#54	Jul 24, 2018 2:24 AM
🔴	#53	Jul 23, 2018 2:24 AM
🔴	#52	Jul 21, 2018 2:24 AM
🔴	#51	Jul 20, 2018 2:24 AM
🔴	#50	Jul 19, 2018 2:24 AM

	QUIC Framework	picotls	picoquic	quicly	ngtcp2	Test
Average stage times: (Average full run time: ~2min 30s)	1s	52s	4s	3s	37s	101ms
#64 Aug 05 02:24 1 commit	1s	28s	9s	2s	39s	77ms
#63 Aug 04 02:24 4 commits	1s	29s	2s	2s	37s	112ms
#62 Aug 03 02:24 2 commits	1s	35s	1s	2s	37s	135ms
#61 Aug 02 02:24 5 commits	1s	30s	11s	1s	36s	128ms
#60 Aug 01 02:24 18 commits	1s	2min 37s	1s	2s	37s	119ms
#59 Jul 30 02:24 6 commits	1s	2min 24s	1s	2s	36s	81ms
#58 Jul 29 02:24 2 commits	1s	27s	1s	2s	36s	106ms

Figure 5.4: History of Pipeline Job

When a test fail, it is mostly due to the fact that the framework is not changed according to the new log format or change in packet header in the latest draft. There should be some constant effort to regulate the mismatch between the framework's test process and log parser. When there is a change even in one of the arguments of the client or server program, we have to update the framework to reflect it. Also, a change in the log format also needs to be addressed immediately because the log parser are designed to expect the logs in certain way.

5.5 Scalability

The QUIC specification is still under development and there still might be a lot of new groups starting to work on the implementation. The There can also be new functionality which QUIC aims to achieve. The framework should scale up to new implementations and test scenarios. The next section evaluates the implemented framework with respect to the scalability of new implementations and test cases.

5.5.1 Implementations

The amount of effort needed to add a new implementation is much less because our test framework depends on the pipeline architecture of Jenkins. The following are the steps that needed to be done to add a new implementation.

- Analyze the build steps needed for the new implementation and then add the build step to the build phase of the pipeline plugin.
- Add the script to run the new implementation as a server and client in the test phase of the pipeline script. It also has to run as a client against all the existing implementations participating in the interoperability test.
- Manually analyze the log of the implementation and build a log parser by creating a mapping between the seven states described in chapter 4 and to the details available in the implementation's log.

There is a possibility that the build phase takes much time when there are a lot of implementations to build one by one. Jenkins supports master-slave architecture to manage distributed build. In this scenario, the Jenkins server acts as a master and schedule build jobs to other machines. This offloads some of the build processes to the slave machines. It also takes care of monitoring the tasks running on the slave and then once finished it presents the build results. The advantage of this approach is that the slaves can run on a different operating system and so it is possible to test the implementation on different operating systems. It is also possible that the Jenkin's slave can be a virtual machine, and this saves us some cost also.

5.5.2 New Test Scenario

There is a possibility to add more test scenarios to test the interoperability of any two QUIC implementation. The current design and implementation of the framework are particular to the test scenarios which we mentioned in the previous chapter. These scenarios can be tested by looking at the headers of the QUIC packet and without having to decrypt the QUIC packet. More complex scenarios need to look into the frames carried in each packet which needs decryption. Modeling such specific test scenario is out of scope for this framework.

Some of the new test scenario requires changes to the log parser, this can be done by either by adjusting the state transition matrix or by changing the parser for each implementation to address the new scenario. This will need some understanding of the existing parsing mechanism for each implementation. On the other hand, when a new scenario is tested by changing the network's parameters to check how the QUIC performs when there is delay or loss in packets, it becomes much complex to model.

5.6 Summary

In this chapter, we evaluated our framework design by setting up a test machine with the Jenkins server. Subsequently, we ran the pipeline job to run test on both open source QUIC server and public QUIC server. We also presented the final interoperability test matrix which the framework generates. Later, we evaluated the framework based on scalability of new implementations as well as new test scenarios. The interoperability test framework scales up well to new implementations but it might not be able to support every new test scenario.

Chapter 6

Conclusion

In this thesis, we have designed and implemented an automated framework for interoperability test of IETF QUIC. At first, we present the overview of TLS 1.3 which provides cryptographic security and message integrity for QUIC protocol. Then, we present the three modes which help TLS 1.3 in achieving reduced connection latency. Later, we discuss the important features provided by QUIC protocol. This helps to understand the overall functionality of QUIC. We also discussed the various packet formats and types which QUIC uses and this helps us in designing the test scenario for the interoperability testing.

After discussing the two protocols which are needed for designing a QUIC interoperability test framework, we look into the general process involved in automated testing. In addition, we discussed the importance of automated testing tool and how continuous integration tool can be used for automation of building and testing. Subsequently, we showed the objective and the need for interoperability testing when developing a new protocol. Finally, the challenges of interoperability testing were highlighted.

In the design phase, we listed out all the challenges in designing an automated framework for interoperability test of QUIC protocol. Keeping that in mind we design the conceptual model of the environment and show how different logs can help us in validating the tests. Later, we propose three design setup for the framework and present the advantage and disadvantage of each setup. Finally, we decide on the test scenario's which determines whether two implementations interoperate or not. The list of test scenarios are not exhaustive, there can be more test scenario's added to the framework. We described the objective of each of the test scenario, depicted the flow of packets for the test scenario and explained the packet flow step-by-step for better understanding.

In the implementation of the framework, we choose a continuous integration tool which helps in overcoming the challenges of the interoperability testing. Later, we present that Jenkins is the best choice and provide arguments to show that the challenges of the interoperability testing like network complexity, test scalability, and root cause analysis of the failure can be overcome by using Jenkins as a continuous integration tool. Subsequently, we use the pipeline structure of Jenkins to design our framework into a build phase, test phase, and finally parse the log and publish the interoperability test result. Jenkins offers the plugins to write most of the task in build phase and test phase. The log parser phase is written in python and we show an idea of how to write a parser for the log file using picoquic as an example. Finally, we also show that we can represent the result of the test in the dataframe data structure which the pandas library offer. This library also helps us in styling the interoperability test matrix which is later published using the HTML publisher plugin.

We evaluate the framework by running a test environment with three implementation that has a open source code on GIT and three implementations whose source code is not available but we have the IP address and the port number where it's QUIC server is running. We proved that the framework is working as expected in both these cases. Later, we also evaluate the framework's scalability on two criteria which is when either a new implementation or a new test scenario is added for interoperability testing.

6.1 Future Work

In this thesis, we designed and implemented a log parser that parses the log files generated by the QUIC server and client. This is done to validate the test case and to verify if the packet exchange followed as per QUIC protocol specification. Analyzing the log files of each implementation and writing the parser is a time-consuming process. The future work for this thesis is to design a common logging interface and log structure. This can be used by all the implementations to use a standard format. This forces all logs to be in the same format and the log parser need not be changed when there is a new implementation provided the new implementation uses the new logging mechanism.

Bibliography

[1] T. Taubert, *MORE PRIVACY, LESS LATENCY Improved Handshakes in TLS version 1.3*. [Online]. Available: https://timtaubert.de/blog/2015/11/more-privacy-less-latency-improved-handshakes-in-tls-13/

[2] C. L. X. W. Y. Cui, T. Li and M. KuĬĹhlewind, *Innovating Transport with QUIC: Design Approaches and Research Challenges*. IEEE Computing Society, 2017.

[3] O. B. et al., *Ensuring Interoperability with Automated Interoperability Testing*. [Online]. Available: https://portal.etsi.org/Portals/0/TBpages/CTI/Docs/Ensuring%20Interoperability%20with%20Automated%20Interoperability%20Testing_rev6a.pdf

[4] A. L. et al., *The QUIC Transport Protocol: Design and Internet-Scale Deployment*. SIGCOMM, Aug 2017.

[5] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*. [Online]. Available: https://tools.ietf.org/html/rfc5246

[6] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-28*. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-tls13-28

[7] M. Thomson and S. Turner, *Using Transport Layer Security (TLS) to Secure QUIC draft-ietf-quic-tls-13*. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-tls-13

[8] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-13*. [Online]. Available: https://tools.ietf.org/html/rfc6749

[9] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control draft-ietf-quic-recovery-13*. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-recovery-13

[10] H. K. al., *Comparative Study of Automated Testing Tools: Selenium, Quick Test Professional and Testcomplete*. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.448.6743&rep=rep1&type=pdf

[11] *Continuous Integration Essentials*. [Online]. Available: https://codeship.com/continuous-integration-essentials

[12] F. S. N. Griffeth, *An approach to best-in-class interoperability testing*. [Online]. Available: http://comet.lehman.cuny.edu/griffeth/papers/itea.pdf

[13] *Raspberry Pi*. [Online]. Available: https://en.wikipedia.org/wiki/Raspberry_Pi

[14] *Mobile vs Desktop Usage in 2018: Mobile takes the lead*. [Online]. Available: https://www.stonetemple.com/mobile-vs-desktop-usage-study/

[15] *Comparison of continuous integration software*. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software

[16] *Source Code of QUIC Test Framework*. [Online]. Available: https://github.com/angelinrashmi2000/QUIC-TEST-FRAMEWORK

[17] *pandas: powerful Python data analysis toolkit*. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/index.html

[18] *Pandas: Styling*. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/style.html