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 JÃűrg Ott
Advisor:	M.Sc. Teemu KÃďrkkÃďinen
Submission Date:	August 1st, 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 1st, 2018 Angelin Rashmi Antony Rajan

Acknowledgments

If someone helped you or supported you through your studies, this page is a good place to tell them how thankful you are.

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. Since QUIC runs on user space it is easy to deploy and each party can develop their own implementation from the specification. This thesis deals with analyzing the various IETF QUIC implementation and running various interoperability tests between them. Interoperability is the ability for two or more networks, systems, devices or applications to communicate.

Contents

List of Figures

List of Tables

1. Introduction

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. Whenever there is a release in new stardard or product, there is a need to perform interoperability testing. 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 frameowork 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.

Three framework designs are proposed according to the availabilty of resource for setting up the test bed. The framework's feasibilty 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.

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.

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. Related modes which help in achieving quick connection establishment are 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.

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 usually 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. 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 secure than TLS 1.2. TLS 1.3 use 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 component

- **A Handshake Protocol** This component is responsible for authenticating the communicating parties, negotiate the cryptographic suites and establish 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** This 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 it is delivered to the upper-level protocol.

In the next subsections, we will see the 3 basic key exchange modes supported by TLS 1.3. These modes are the main reason for QUIC to take less time for connection establishment.

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.

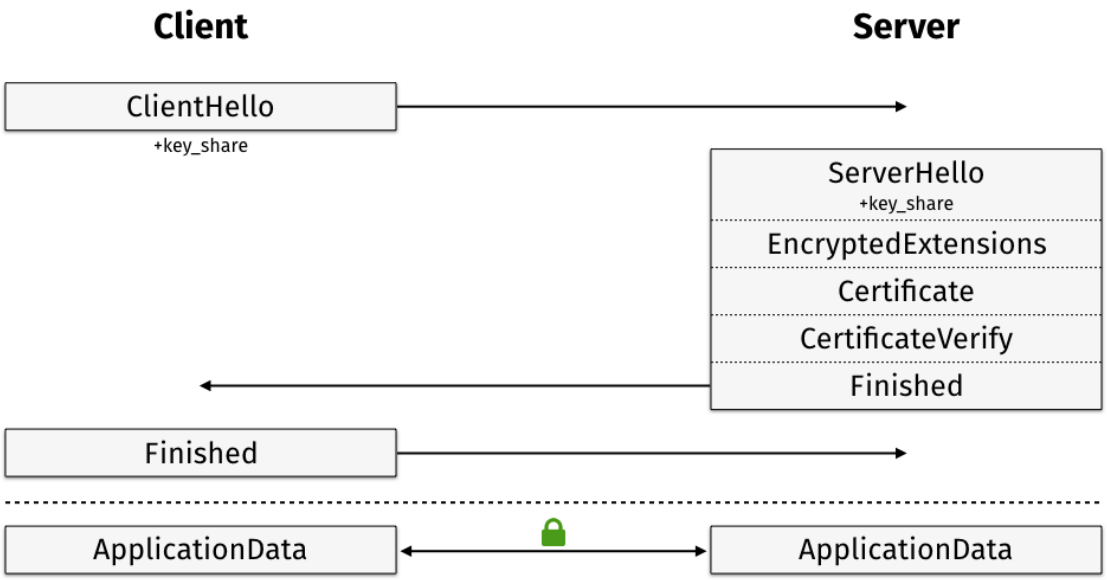

Figure 2.1.: TLS 1.3 1-RTT Full Handshake

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 2.2 depict the messages exchanged in TLS Session resumption mode.

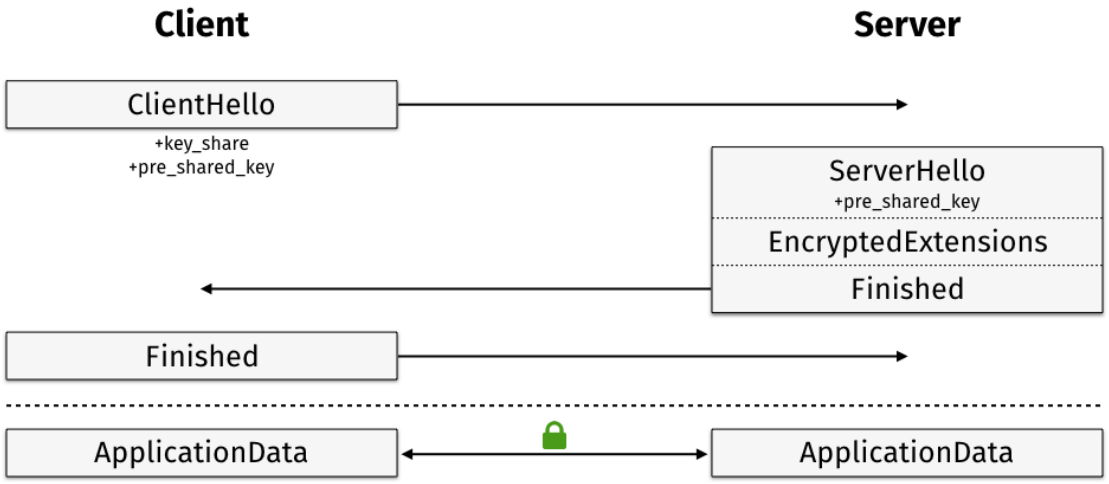

Figure 2.2.: TLS 1.3 Handshake Session Resumption

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.

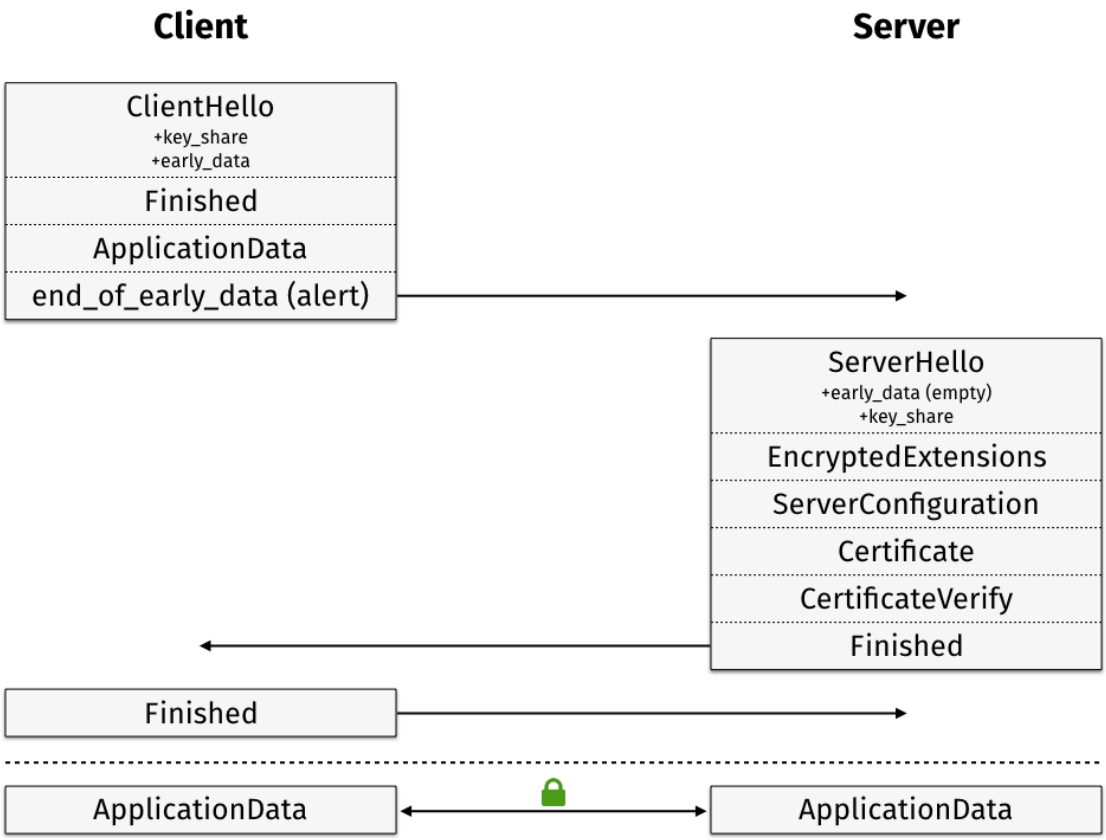

Figure 2.3.: TLS 1.3 Handshake 0-RTT

- **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.

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*. *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.

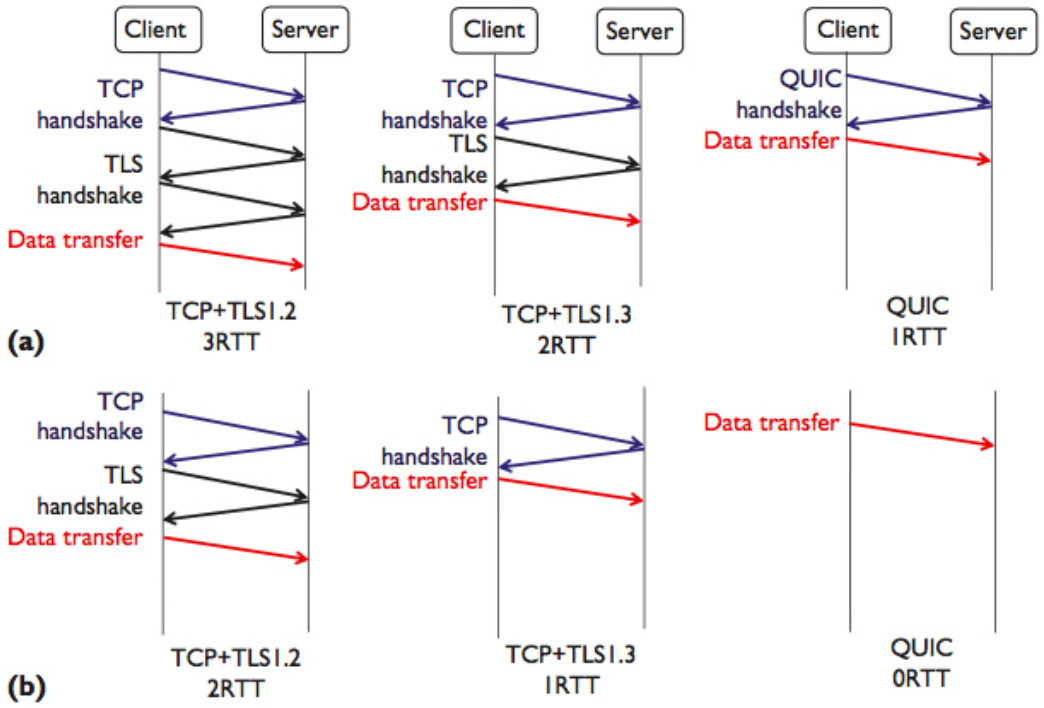

Figure 2.4.: Round Trip Time for different protocol a). First time connection b). Subsequent Connection

2.2.2. 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. 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 has 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. 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 level 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.

2.2.4. Congestion Control and Loss Recovery

In this section, we will see in high level how QUIC implements congestion control and loss recovery. Every transmission in QUIC has a packet sequence number and this number never repeats in the same 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. QUIC uses ACK information and timeouts to detect packet loss.

The ACK based detection of packet loss is similar to TCP's Fast Retransmit, Early Retransmit algorithm.

2.2.5. 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 sngle connection

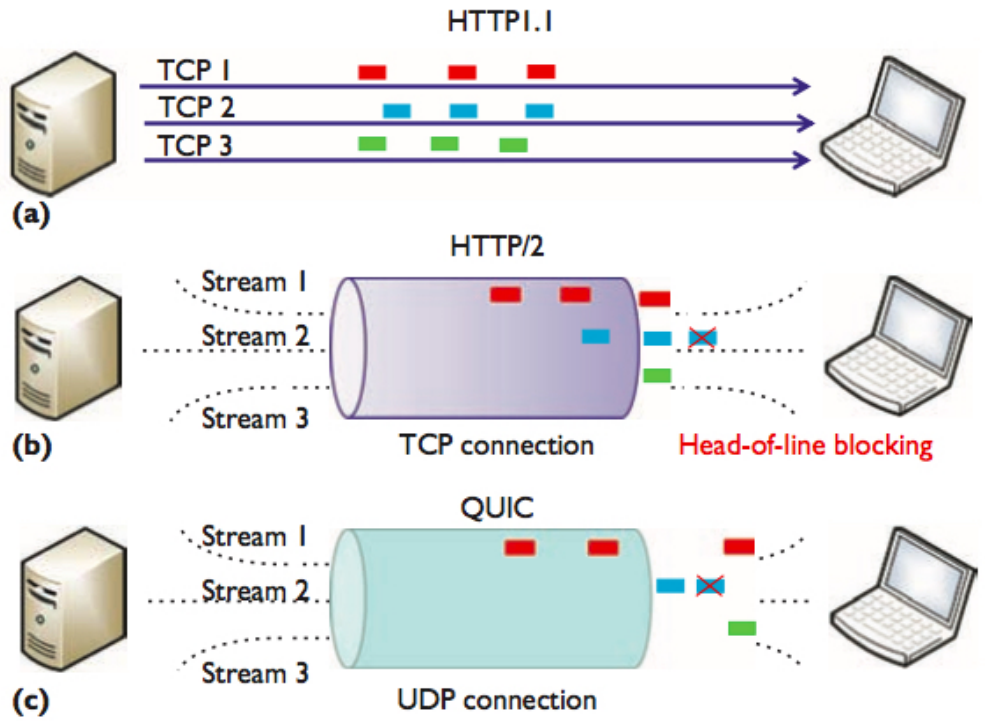

Figure 2.5.: Multiplexing comparison (a) HTTP1.1, (b) HTTP/2, and (c) QUIC.

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.

2.2.6. QUIC Packet Types and Formats

Inorder to develop a test framework for QUIC we need to understand the packet formats and type. 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 establishment of 1-RTT keys and version negotiation. Short headers are used after version negotiation and 1-RTT keys are established.

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** It 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

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

Table 2.1.: Long Header Packet Type

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 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.

2. **Retry Packet** It uses long headers with packet type 0x7E. It is used by the server which wishes to perform stateless retry. It can also carry cryptographic handhsake message and acknowledgements. Also, it must contains atleast 2 frames, one ACK frame to acknowledge the clients initial packet and the another STREAM frame containing the server's cryptographic stateless retry information. 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** It 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** It 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.

Short Header

Short Headers are used after 1-RTT key is established. It has Header Form 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. Security Considerations in QUIC

- **Slowloris Attack** It is type of denial of service attack in which the attacker tries to keep many connections to the target endpoint and try to hold them open as long as possible. The

attacker sends a partial connection request and never completes the connection. This forces the server to stop connection request from additional clients. This can be mitigated in QUIC by increasing the maximum number of clients the server will allow, imposing restrictions on the transfer speed, restricting the length of time a client is allowed to stay connected or by imposing restriction on number of connections from the same IP address.

- **Optimistic ACK Attack** In this method of attack the congestion control algorithm is expoited to cause denial of service. Usually ACK packet is used to adjust the congestion window size. The size of the congestion window increases are the ACK packet's are received in order and it is decreases as ACK's are received out of order. The attacking endpoint acknowledges the packet which it has not received and thus causing the server to increase it's congestion window and sending rate till the server cannot server any other client. The server can detect this attack by skipping packets intentionaly when sending packets. The server closes the QUIC connection with PROTOCOL_VIOLATION error if it receives an ACK.

- **Stream Commitment Attack** In QUIC streams are identified by a 62 bit integer called Stream ID. Usually clients initiate an even-numbered stream ID whereas server initiate an odd-numbered stream ID. Clients are expected to open streams sequentially ie., when a stream with higher stream ID is opened then causes opening of all lowwer-numbered streams. The attacker can repeat this process in a large number of connection and cause something similar to TCP SYN flooding. To mitigate this attack, the number of concurrent transport parameter can be set efficiently. If it is set two low then the performance of application which needs a large number of open streams can be affected.

- **Spoofed ACK Attack** Usually transport protocol spends a round trip time to verifiy that a client can receive packets sent to its claimed IP address and port to avois spoofing of this information by an attacker. QUIC uses a token based address validation by providing a client with a token which cannot be easily guessed. If the client can return the token then it owns the transport parameter.

  In this attack, the attacker gets an address validation token from the server and later releases the IP address it used to acquire the token. Later, the attacker spoof the same address and initiates a 0-RTT connection on a victim's(new owner of the IP address) behalf. The attacker can send spoofed ACK frames anf cause the server to send excessive, unwanted data to the victim.

2.4. IETF Implementation of QUIC

In the below table we can see the various quic implementations in different languages. These implementations use any other application protocol other than IETF HTTP over QUIC eg. HTTP/0.9. Currently only 11 out of 13 implementations participate in interoprability test. For the interoperability testing each of these implementations is run as server and client in all possible combinations.

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
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

2.5. 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. 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.

2.5.1. 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 these integration problems. It helps in merging all the working copies of the developers who are working on the same project. Most of the CI typically use a build server to automate the building 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 as follows.

1. During the application development lifecycle, the developers are working on their local copy and commit them to tthe central source code repository for version control eg: Git.
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 and if it finds some changes in the main branch, the build process is triggered.
4. The CI system pulls the code from the 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, fails the build process.
6. Notify the developer about the test failure and is added to the defects tracking system.
7. Finally, the application is deployed if all the tests are successful.

2.6. Interoperability Testing

Internet is growing day by day and the devices used are from different vendor 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 work as expected between these different hardware and platforms. This test helps in identifying the implementation errors if there is some ambiguity in the standards. Each vendor does their own implementation of the standard and if there are some ambiguity in the specification, interoperability test can identify them. Other than that interop test can find if the desired performance is achieved.

Figure 2.6 depicts the generic framework for automated interoperability testing defined by European Telecommunications Standard Institute. This framework is mainly used for testing interoperability in telecommunication. (ETSI).

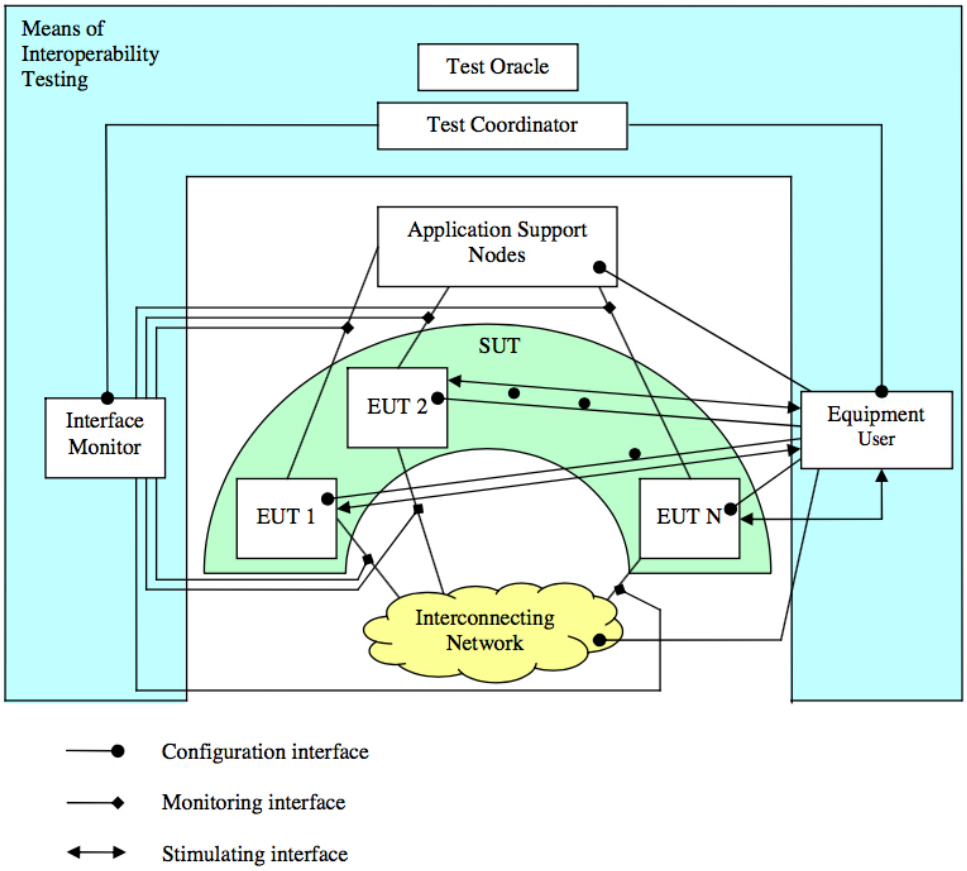

Figure 2.6.: General Framework for Interoperability Testing

2.6.1. 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 has different implementations of the same standards as they are from different vendors ie it independent implementation of the same standard. The other aim of interoperability testing is to verify that the required performance is achieved.

2.6.2. Steps for Interoperability Testing

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

1. **Setup** This phase is to define a formal statement of work and to find all the different applications that are part of the network. The necessary automation tools are set up.

2. **Plan** This is one of the important phase in any development process. It is very important to understand the entire application in our case it 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 main focus to test the end-to-end functionality of the applicaton/protocol covering all possible scenario. Apart from the functionality of the application the non-functional testing like performance, security etc should 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 Chanpter 4.

4. **Execute Test Scenario** The test cases ot test suites developed are executed in the real environment. The outcome of this phase is either pass or fail.

5. **Document and Report Test Results** The output of a test in interoperability testting is a complex combination of information from different sources invloved. 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. The test logs which we got from the previous steps are analyzed, in particular the failed test cases. In our case we document the test results in the form of matrix for each implementation.

2.6.3. Challenges in Interoperability Testing

One of the main issues in interoperability testing is network complexity ie., there might be a lot of devices involved in between when two devices communicate on the internet. It is difficult to simulate such a big network instead we can only do a pair-wise testing. Suppose there are N implementations then we have (N2-N)/2 distinct combinations to test.

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. When the tests fail, it is also difficult to identify the root cause of the issue.

2.6.4. Automate Interoperability Testing

There is a need to reduce the time to market a 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.

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.

3. Design

In this chapter, the proposed setup for running intereoperability test and the frameowrk is discussed. We also see the test plan to test some features of QUIC protocol. We cover only the basic few test scenarios in this thesis.

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. Each implementation is done by different group of people and hence one implementation might be feature complete and others are still in progress. This situation makes it difficult to run interoperability test as both the specification and implementations are continously evolving.

The other issues is that only some of the implementations are not open source. This makes it difficult to setup the test scenario. These implementations have their remote server running as QUIC Server or we have access only to the binaries. In both cases we dont have access to the actual source code. With the implementations for which we have the source code, we have to find a way to compile and build the project before testing.

QUIC logs are not standardized. Each implementations does their own way of logging on server and client side. Some implementations doesn't even provide the server logs. To find out what exactly is happening we have to analyze and build a parser for each quic implementations. This is a time consuming process as well as not scalable when there are some new implementations for the protocol. 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 to standardize quic logging meachanism 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 commom 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 3.1 we can see that we can collect logs at two levels, capture the packets exchanged between them. These details can be used to validate the test cases

- Using Kernel Logs

- Using Server and Client QUIC Logs

  Each implementation logs the states, packet sent/received, error message in their own way. There is no standard strucrure 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 gnerated by each implementations into standard events for further processing. We will see more about this in section 4.2.

- Using Wireshark

  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. 0-RTT decryption is still not supported, so we will not be able to use Wireshark for testing that. For testing version negotiation, handhsake, and stream data wire shark packet capture can be used. This way of validating is not much useful because the QUIC payload is encrypted.

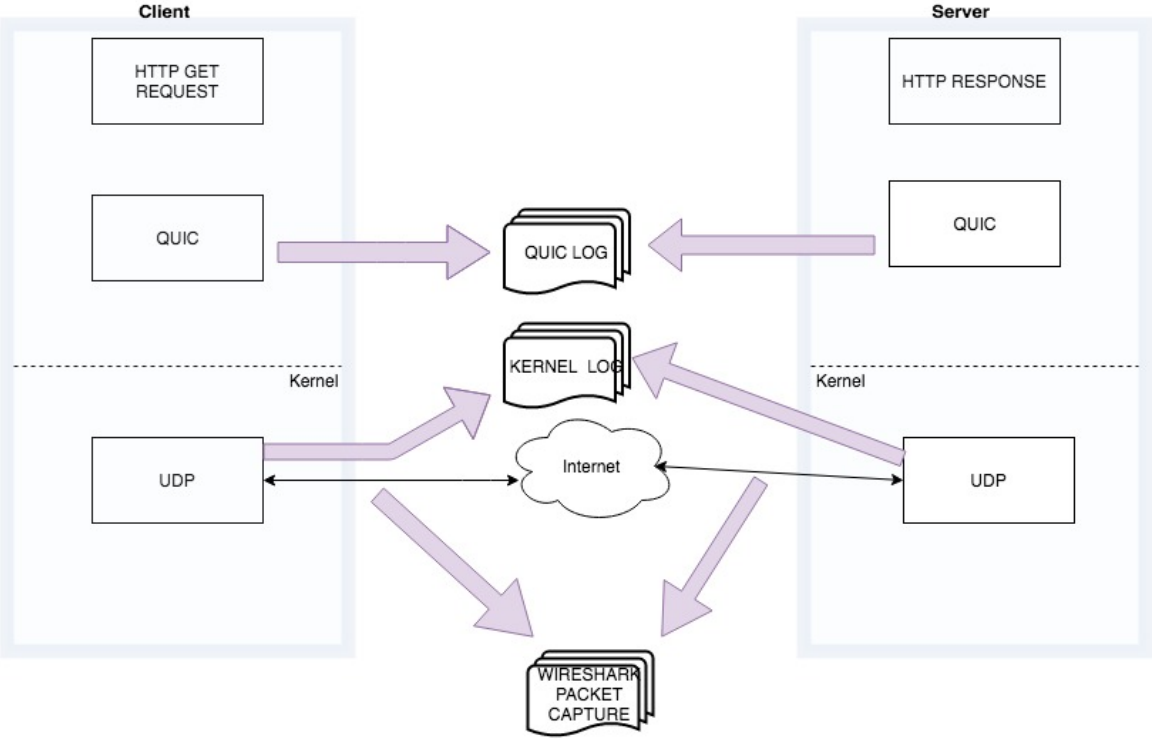

Figure 3.1.: Conceptual Model of the Environment

3.3. System Design

3.3.1. Lightweight System

In the lightweight system we run both client and the server in the same system as a different user processes. There is no network traffic between the client and the server and so we can't examine how the protocol operates under different traffic conditions. As the name suggests, it relatively simpler and faster than the other designs mentioned below. This system is easy to setup and we don't need any extra resources for testing aand evalutaing the design.

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 eg: iperf, bwping etc. Traffic Shaping technique which manages the bandwidth according to the required traffic profile. This can be used to regulate the performance and qualitu of service. Traffic Shapping retains the flow of excess packets in a queue and later scheduled for transmission when the network condition is favourable.

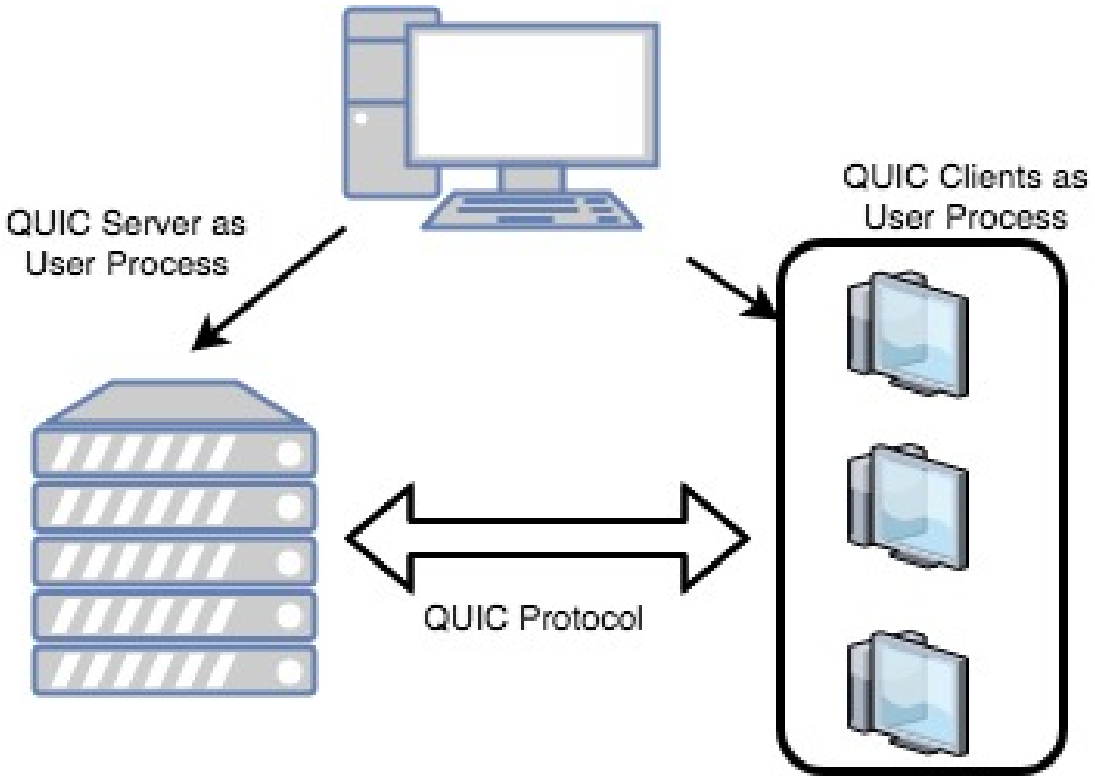

Figure 3.2.: Lightweight System

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.

3.3.2. Virtualized System

Using virtual machines for interoperability testing can help in emulating different maachines with different operating systems on a single physical computer. In the virtualized system we have both the client and server running in different virtual machines. The 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 just for testing.

3.3.3. Physical System

In this setup we create a physical network of two devices which are connected to each other through a switch or establish a wireless connection between them. This approach helps in examining how the protocol operates in the actual physical devices and network. The devices can be a server, mobile device or even a rasberry pi. 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.

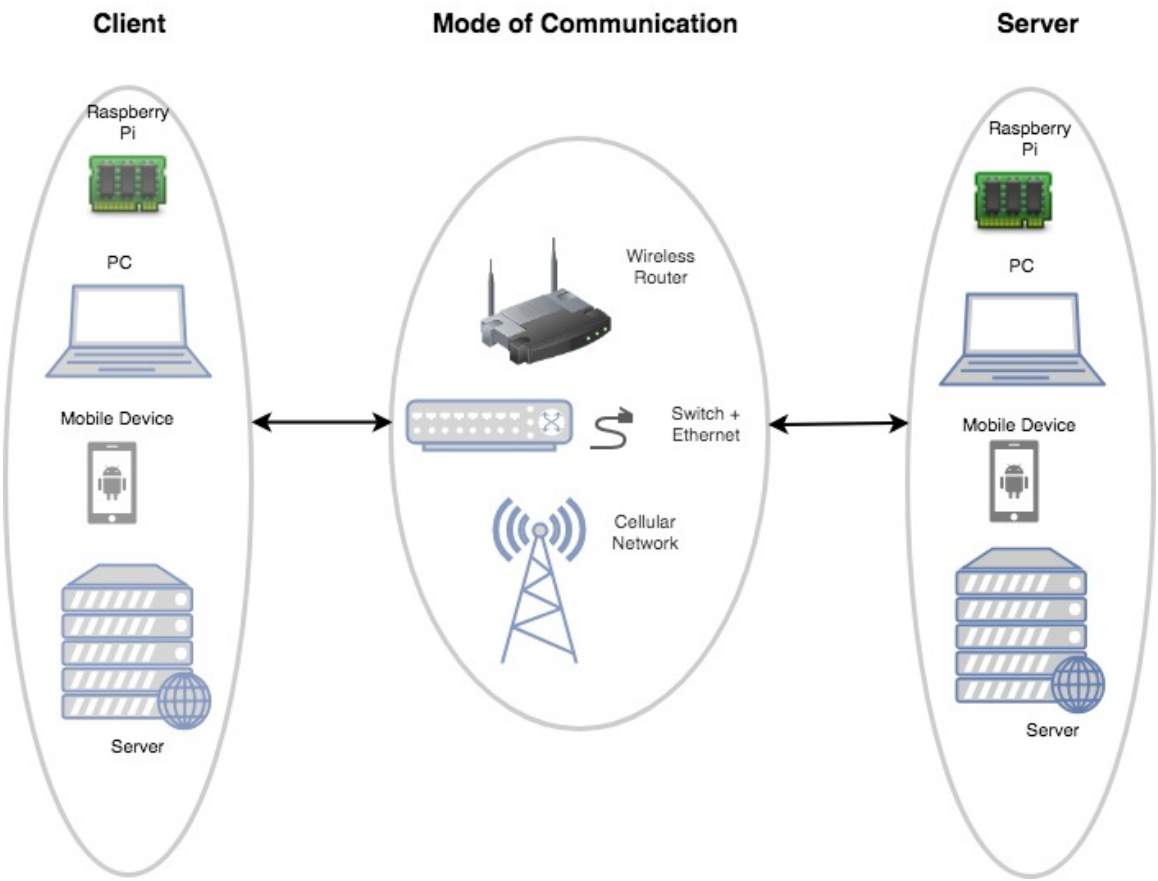

Figure 3.3.: Physical Setup

3.4. Test Scenario

In this section we will see the some of the important test scenarios that we have come up with to determine the interoperability of two implementations. This list is not exhaustive and there is a possibility to add more test cases to ensure that two implementations interoperate with each other. Each of the test scenario has further specific test cases.

3.4.1. Version Negotiation

QUIC Version Negotiation packet is sent by the server to the client in response to a client packet that contains a version which the server does not support. This test scenario can be made by explicitly setting a version by the client which the server does not support and wait for the version negotiation packet from the server. Since version negotation packets are not encypted, it is possible to do validate this test case from wireshark capture as well. This can be done ny

checking if the packet is long header Each implementation 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 implementation1 currently supports only draft-09 and implementation2 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 continously evolving.

3.4.2. Handshake

Cryptogrpahic 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 Stream 0 is used for sending cryptographic handshake.

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.

3.4.4. Connection Close

A QUIC connection can be terminated in one of the following 3 ways

- idle timeout 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 TP and the maximum value is 600 second(10 min). If a connection remain idle longer than the idle timeout will be closed.
- immediate close CONNECTION_CLOSE frame is used 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

3.4.5. Resumption

3.4.6. 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. ngtcp2 provides a way to resume a session and send 0-RTT packets in their example/client implementation. This is done by first establishing a connection with a server using the below command

./examples/client 127.0.0.1 4444 –session-file /Users/Rashmi/Documents/workspace/sample/ngtcp2/sess.txt –tp-file /Users/Rashmi/Documents/workspace/sample/ngtcp2/tp.txt

The above command stores the transport parameter and session ticket locally. This can later be used for resuming the session and sending 0-RTT packet.

3.4.7. Stateless Retry

A server can 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 in response the the client initial packet. This Retry packet has a long header with the type value 0x7E. This carries cryptographic handshake message of the server and acknowledgements. The client reset its transport parameters but the remembers the state of the cryptographic handshake.

4. Implementation

In the previous chapters we discussed about QUIC protocol, automated testing tools and the importance of interoperability testing. We also discussed about the three design models Light Weight, 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 one of the design model Light Weight Setup.

4.1. Continuous Integration Server Setup

The framework uses Jenkins as continuous integeration server as it is a widely used tool for building, automating tests and delivery of projects. It requires little maintenance and also has a graphical user interface for updates of plugins. 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.

Jenkins 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 configure and its basic functionality can be extended by using additional plugins. It has nearly 400 plugins to support almost every aspect. In the following sub sections we will discuss about some of the important plugins which is needed for the implementation of our framework.

4.1.1. Plugins

- **Pipeline Plugin** This is one of the important plugin 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 as follows

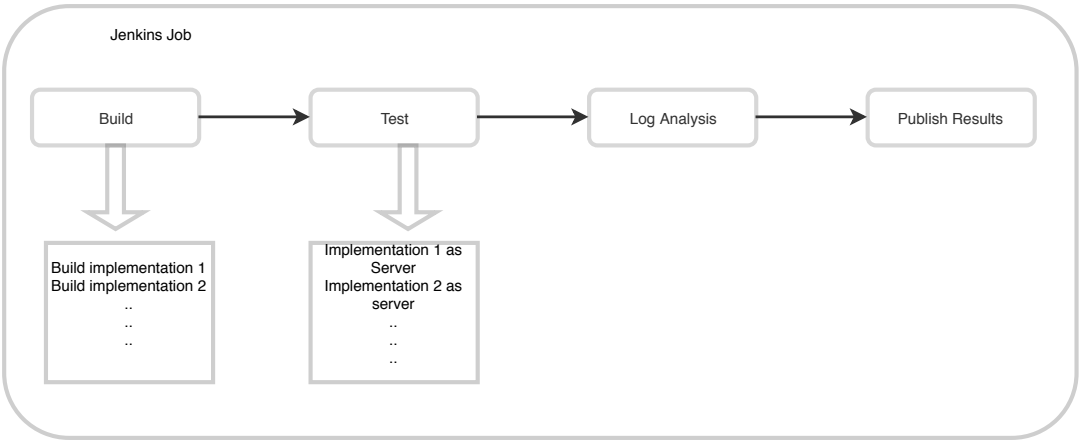

Figure 4.1.: Jenkins Stages

build-> Select Server -> Run all other implementaion as client one by one -> Logs the result from client -> Log Parser -> Generate events -> Publish the interop matrix

- **Git Plugin** This plugin helps us 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. The other version control systems which Jenkins supports are Mercurial, Subversion etc.

```
checkout([$class: 'GitSCM', branches: [[name: '*/master']],
doGenerateSubmoduleConfigurations: false, extensions: [[$class: 'Submodul
disableSubmodules: false, parentCredentials: true, recursiveSubmodules: t
reference: '', trackingSubmodules: false]], submoduleCfg: [],
userRemoteConfigs: [[credentialsId: '8607f52e-a446-45e9-b4d1-02bc8ef0ab8b
url: 'https://github.com/private-octopus/picoquic.git']]])
```

- **HTML Publisher Plugin** This is used for publishing HTML reports.

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

4.2. Log Parser

Talk about QUIC header and the types and Most of the implementations does not provide server log. So we had to design a way to evaluate a test case only by looking at the client log. There is a need for a log parser for each implementation. This is because the QUIC log's are not standardized and it differs implementation to implementation. This is a python code written to parse each log file and generate the interop matrix as an HTML file. The published HTML reports are available in the Jenkins dashboard.

4.2.1. State Transition Diagram and Matrix

In the previous section we talked about having access to only client log. State diagrams are the way to describe the behaviour of the system when there is only object is invlolved, in out case it is only the client log. 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. We have come up with the state transistion diagram to verify from the logs that a particular test case is successful or not. Figure shows the state transition diagram that we have designed after reading the IETF QUIC specification document

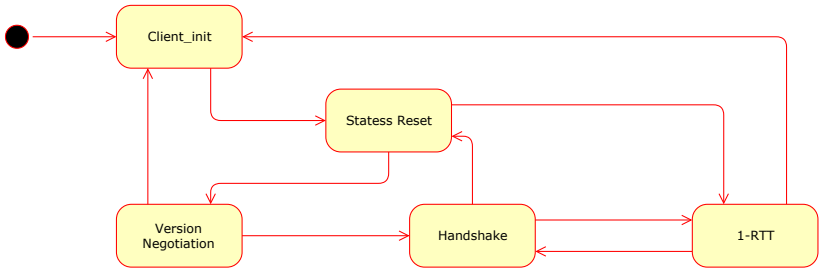

Figure 4.2.: State Transition Diagram for IETF QUIC

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

4.2.2. 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. Dataframe is a two dimensional data structure which has labeled rows and columns. The colums 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.

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

One of the main reason for using pandas library is that it also allows visual styling of a DataFrames. 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 file 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.

```
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"),("display","inlin
7  dfWithStyle = df.style.applymap(color_zero_red)
8                 .set_caption(implementation_name+' as Server')
9                 .set_properties(**{'border': '2px solid black'})
10                .set_table_styles(styles)
11                .render()
```

In line 7, *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 8, *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 10, *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.

4.3. Build Job Setup

Follow the to install jenkins. Once installed, we need to create a build job which builds and compiles all the open source implementation, runs the test cases 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 README section.

4.4. 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 for setting up the Jenkins server, the list of needed plugins for Jenkins is available in the README section. Further details to create pipeline job and building the project can also be found in detail.

4.5. 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.

5. Evaluation

In the previous chapter we saw about the various components needed for developing the testing framework. In this chapter we will evaluate the light weight system design described in section 3.2.1.

5.1. Experimental Setup

In the light weight 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 setup the Jenkins server on the target machine. The system configuration is described below

We have decided to use *picoquic*, *ngtcp2*, *quicly* to evaluate our framework design. Apart from the three implementations we have selected three other standalone server *mvfst*, *ngx_quic*, *winquic*. This selection helps in covering the two possible scenarios that we discussed in section 3.1.

The following are the combinations in which we run the tests

-jenkins server details -participating implementations, their language etc

5.2. Test Against Public QUIC Server

5.3. Test Against Open Source QUIC Server

5.4. History of Interoperability Test Matrix

5.5. Results and Analysis

6. Conclusion

Appendix

A. Detailed Descriptions