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 Jrg Ott
Advisor:	M.Sc. Teemu Krkkinen
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

1. Introduction

1.1. Goals of the thesis

The primary goal of this thesis is to design and implement an automated system that compiles, builds and run interoperability tests between all the QUIC IETF implementations. The results of the interoperabilty test is displayed in a matrix.

1.2. Outline of the thesis

In Chapter 2 we see how QUIC protocol works, objectives and problems in interoperabilty testing. In Chapter 3 we see the propsed setup for interoperability testing as well as the proposed framework. In chapter 4 and 5 we see the implementation of the proposed design and evaluation of it respectively.

2. Background

In this chapter we will look briefly on how QUIC protocol works and 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.

2.1. QUIC

QUIC protocol relays on TLS 1.3 for agreeing on cryptographic protocols and exchanging ephimeral keys. Draft-11 is under development.

2.2. IETF Implementation of QUIC

In the below table we can see the various quic implementations in different languages. Currently only 11 out of 13 implementations participate in interoprability test.

Implementation	**Language**	**Public**	**Dependency**	**Current Draft**
picoquic	C	Yes	picotls	
ngtcp2	C	Yes	picotls	
quant	C11	Yes	picotls	
mozquic	C++ with C interface	Yes		
quicly	C	Yes	picotls	
ATS	C++	Yes	picotls	
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		

Table 2.1.: IETF QUIC Implementations

2.3. Automated Testing Tools

2.3.1. Continous Integration

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. Proposed Setup

The proposed design for running interoperability testing between a client and a server is shown in Figure 3.1. From the figure 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.

- 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.

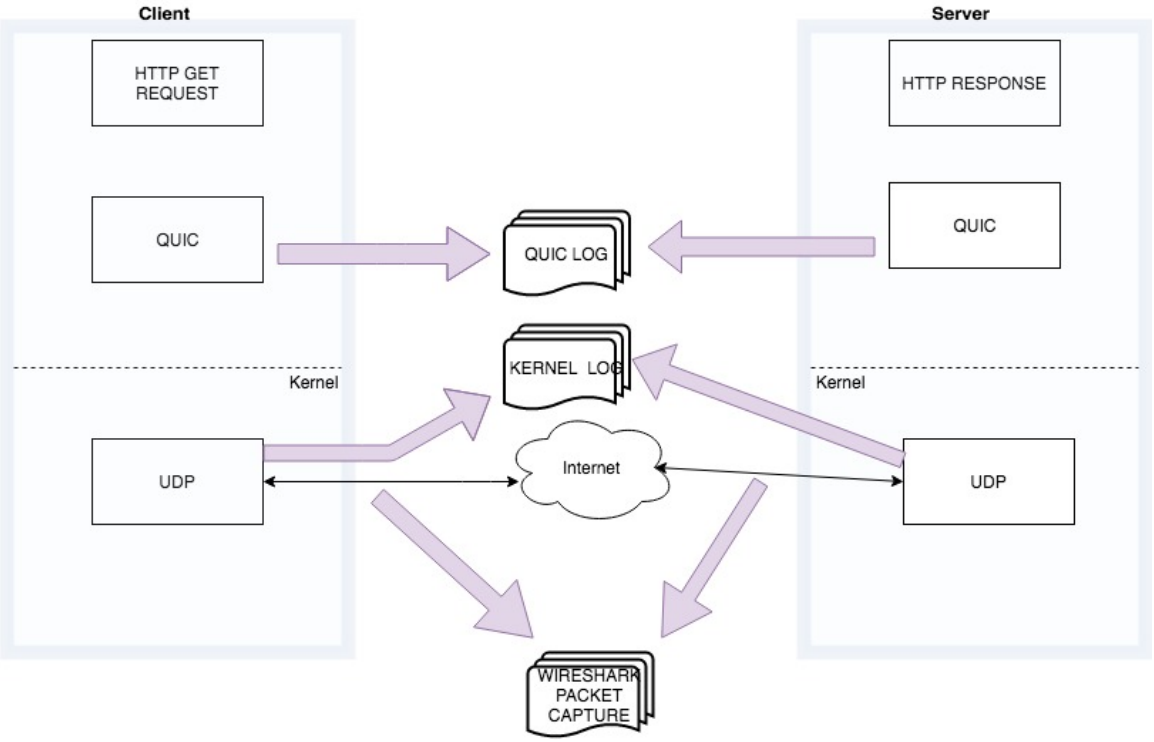

Figure 3.1.: Conceptual Model of the Environment

3.3. System Design

3.3.1. Light Weight

In the light weight system we run both client and the server in the same machine 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. This system is easy to setup and we don't need any extra devices.

3.3.2. Virtualized

In the virtualized system we have both the client and server running in virtual machines. The advantage of this approach is that we can test the protocol even on different operating system.

3.3.3. Physical Setup

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.

3.4. Test Scenario

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 unless it sends Retry packet. This is followed by the client sending handhake packet to server.

3.4.3. Stream Data

3.4.4. Connection 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.

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
–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

4.1. Jenkins

Jenkins is an automation server for continous integration and delivery of a project. It is a 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 has a lot of plugins.

5. Evaluation

6. Conclusion

Appendix

A. Detailed Descriptions