

Formal specification and testing of QUIC

*Thesis submitted to the
Indian Institute of Technology Bhubaneswar
For award of the degree*

of

**B.Tech in Computer Science and Engineering and M.Tech in
Computer Science and Engineering**

by

Raghav Gade

20CS02003

Under the guidance of

Dr. Srinivas Pinisetty



**SCHOOL OF ELECTRICAL AND COMPUTER SCIENCES
INDIAN INSTITUTE OF TECHNOLOGY BHUBANESWAR**

May 2025

APPROVAL OF THE VIVA-VOCE BOARD

06/05/2025

Certified that the thesis entitled **Formal Specification and Testing of QUIC**, submitted by **Raghav Gade (20CS02003)** to the Indian Institute of Technology Bhubaneswar, for the award of the degree of *Master of Technology* in Computer Science & Engineering under the Dual-Degree Programme has been accepted by the examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

(Supervisor)

(Internal Examiner 1)

(Internal Examiner 2)

CERTIFICATE

This is to certify that the thesis entitled **Formal Specification and Testing of QUIC**, submitted by **Raghav Gade** to Indian Institute of Technology Bhubaneswar, is a record of bonafide research work under my supervision and I consider it worthy of consideration for the award of the degree of *Bachelor and Master of Technology* of the Institute.

Date:

Place: Bhubaneswar

Dr. Srinivas Pinisetty

Associate Professor

Department of Computer Sc. & Engineering

School of Electrical & Computer Sciences

Indian Institute of Technology Bhubaneswar

Odisha, India

DECLARATION

I certify that

- a. the work contained in the thesis is original and has been done by myself under the general supervision of my supervisor.
- b. the work has not been submitted to any other institute for any degree or diploma.
- c. I have followed the guidelines provided by the institute in writing the thesis.
- d. I have conformed to the norms and guidelines given in the ethical code of conduct of the institute.
- e. whenever I have used materials (data, theoretical analysis, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.
- f. whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Raghav Gade
20CS02003

Acknowledgments

I would like to express my sincere gratitude to everyone who has contributed to the successful completion of my project. Firstly, I would like to thank my project supervisor, Dr. Srinivas Pinisetty, for his invaluable guidance, mentorship, and continuous support throughout the project. His expertise and insights were instrumental in shaping the direction and quality of my work.

I would also like to express my appreciation to all my institution's professors, faculty members, and staff for their support and encouragement.

Last but not least, I extend my heartfelt thanks to my family and friends for their unwavering support and encouragement throughout the project. Thank you all for your valuable contributions, guidance, and support.

Raghav Gade
20CS02003

Abstract

QUIC is a modern transport protocol aimed at replacing the TCP + TLS stack. Google first introduced the protocol in 2013, and the IETF published it as RFC 9000. As QUIC serves as the basis for HTTP/3, it is likely to carry most internet traffic in the future. The IETF QUIC Working Group oversees protocol evolution and extensions. This thesis explores the formal specification and verification of QUIC. It examines Microsoft’s Ivy-based formal models, which check protocol conformance using raw wire data captured from live network traffic. These specifications were based on earlier QUIC drafts (draft-18, later draft-29) rather than the final RFC 9000. While Ivy excels at protocol verification through inductive invariants, it has limitations in verifying liveness properties. To complement this approach, a new formal specification using TLA+ was developed. TLA+, based on temporal logic, enables verification of both safety and liveness properties by analyzing protocol traces recorded in qlog format, an emerging logging format for QUIC currently in IETF draft status. Unlike the Ivy approach that works with raw network traces, the TLA+ model processes standardized qlog data from implementations like aioquic and picoquic, though computational constraints limit analysis to sample traces of modest length. The TLA+ specification remains under development. This work demonstrates how different formal methods with complementary strengths can provide more comprehensive verification of network protocols like QUIC.

Contents

Certification of Approval	i
Certificate	ii
Declaration	iii
Acknowledgments	iv
Abstract	v
List of Figures	x
List of Tables	xii
List of Abbreviations	xiv
1 Introduction	1
1.1 Motivation	1
1.1.1 TCP and its evolution	1
1.1.2 Why QUIC?	2
1.2 Problem formulation	2
1.2.1 Need for formal specification	2
1.2.2 Testing of transport layer protocols	3

1.3	Literature Survey	4
2	QUIC Protocol details	5
2.1	TCP and UDP	5
2.2	Problems with TCP	5
2.2.1	Connection overhead	5
2.2.2	Half-Open connection	6
2.2.3	Connection migrations	7
2.2.4	Head-of-line blocking	7
2.2.5	Kernel space implementation	8
2.3	History of QUIC	9
2.4	Protocol Stack	9
2.5	Features of QUIC	10
2.5.1	Connection Establishment	10
2.5.2	Improved Congestion Control	11
2.5.3	Multiplexing	12
2.5.4	Forward Error Correction	12
2.5.5	Connection Migration	12
2.5.6	Other features	13
2.6	Implementations of QUIC	14
3	RFC 9000	15
3.1	MUST/MUST NOT Statements	15

3.2	SHOULD/SHOULD NOT Statements	20
3.3	MAY/MAY NOT Statements	24
4	Traces	28
4.1	qlog	28
4.2	tcpdump and tshark	30
5	Methodology	33
5.1	Running Implementations	33
5.1.1	aioquic	33
5.1.2	picoquic	35
5.1.3	minquic	36
5.2	Interoperability Testing	38
5.3	Formal specification in Ivy	40
5.3.1	Specification details	40
5.3.2	Test setup	41
5.3.3	Results	43
5.3.4	Expected results	45
5.4	Formal specification in TLA+	46
5.4.1	qlog traces	46
5.4.2	Safety properties	47
5.4.3	Liveness properties	49
5.4.4	State variables	51

5.4.5	Model checking	52
5.4.6	Properties that cannot be satisfied	55
6	Conclusions and Future work	56
6.1	Conclusion	56
6.2	Comparison with Previous Approaches	57
6.3	Limitations and Areas for Further Work	57
	Appendix	59
A	Ivy concepts	59
A.1	Control	60
A.2	Non deterministic choice	61
A.3	Monitors	61
A.4	assume, require and ensure	62
B	TLA+ concepts	64
B.1	Introduction to TLA+	64
B.2	Setting Up TLA+	65
B.3	Key TLA+ Operators	65
B.4	Important Keywords and Constructs	66
B.5	Sets and Sequences	67
B.6	Temporal Operators and Properties	68
	Bibliography	70

List of Figures

2.1	Example of Head of line blocking in TCP.	8
2.2	Example of Head of line blocking and retransmission in TCP.	8
2.3	Protocol stack comparison	10
5.1	Client and Server output of aioquic on terminal	34
5.2	Running aioquic with Wireshark	34
5.3	Frame details of QUIC in Wireshark	34
5.4	Client and Server output of picoquic on terminal	35
5.5	Running picoquic with Wireshark with handshake packet information .	36
5.6	Running picoquic with Wireshark with payload packet information . .	36
5.7	Test program for minquic	37
5.8	Client and Server output of minquic on terminal	37
5.9	Running minquic with Wireshark with payload packet information . . .	38
5.10	Interoperability tests	39
5.11	Example of transfer test for interoperability between aioquic and pico- quic in Wireshark	39
5.12	Example of 0-RTT test for interoperability between aioquic and picoquic in Wireshark	40

5.13 Example of unsuccessful transfer test for interoperability between mvfst and msquic [1] in Wireshark	40
5.14 Test on picoquic server	43
5.15 Output generated on picoquic server test	44
5.16 Max stream test on minquic server	44
5.17 Output generated on minquic server test	44
5.18 Test on picoquic server inside docker container	45
5.19 Safety and liveness testing configurations in Toolbox	54
5.20 Results obtained after running the model checker	54
5.21 Model generating error trace after failing invariant	55

List of Tables

2.1	Comparison of TCP, UDP, and QUIC	11
2.2	Comparison of QUIC Implementations	14
3.1	Connection Management MUST Statements from RFC 9000	16
3.3	Version and Compatibilty MUST Statements from RFC 9000	17
3.4	Flow control MUST Statements from RFC 9000	18
3.5	Frame and Stream Handling MUST Statements from RFC 9000	19
3.6	Error Handling and Closure MUST Statements from RFC 9000	19
3.7	Crypto and Security MUST Statements from RFC 9000	20
3.8	Migration and Path Management MUST Statements from RFC 9000	20
3.9	Connection and Endpoint Behavior SHOULD Statements from RFC 9000	21
3.10	Packet Construction and Usage SHOULD Statements from RFC 9000	21
3.11	Error Handling and Robustness SHOULD Statements from RFC 9000	22
3.12	Acknowledgment and Congestion Control SHOULD Statements from RFC 9000	22
3.13	Migration and Path Management SHOULD Statements from RFC 9000	22
3.14	Security and Privacy SHOULD Statements from RFC 9000	23

3.15	Timers and Timeouts SHOULD Statements from RFC 9000	23
3.16	Miscellaneous SHOULD Statements from RFC 9000	24
3.17	Connection and Endpoint Management MAY Statements from RFC 9000	24
3.18	Packet Construction and Processing MAY Statements from RFC 9000	25
3.19	Migration and Path Management MAY Statements from RFC 9000 . .	25
3.20	Error Handling and Closure MAY Statements from RFC 9000	26
3.21	Frames and Streams MAY Statements from RFC 9000	26
3.22	Security and Privacy MAY Statements from RFC 9000	27
3.23	Miscellaneous MAY Statements from RFC 9000	27
5.1	Comparison of picoquic and quant test results	45
5.3	Error code distribution for picoquic server test cases	46

List of Abbreviations

ACK	Acknowledgement
FEC	Forward Error Correction
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
MQTT	Message Queuing Telemetry Transport
PTO	Probe Timeout
RAID	Redundant Array of Independent Disks
RFC	Request for Comments
RTT	Round trip time
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Chapter 1

Introduction

1.1 Motivation

1.1.1 TCP and its evolution

TCP/IP suite has been running the internet since TCP was standardized as RFC 761 in 1980. Soon after that it was implemented as part of operating system's networking stack, which resides in the kernel space. This was done to provide direct interaction to hardware and network interface cards and reduced overhead for tasks like packet routing, buffering and transmission. And also done to ensure uniform behaviour across all applications. TCP is connection-oriented, reliable, ordered transport protocol which also provides error checking, congestion control, flow control and in-order delivery. Since 1980, there have been several additions to TCP such as congestion control mechanisms, TLS integration and selective acknowledgments. TCP evolution is slow due to kernel integration requiring widespread OS updates. Still TCP suffers from major issues like head-of-line blocking, inefficiencies in handling retransmissions, lack of built-in security and middle-box interference.

1.1.2 Why QUIC?

QUIC protocol implements TCP-like properties at the application layer and uses UDP as transport. Many protocols preceded QUIC, such as SPDY and HTTP/2 via TCP, as well as improvements to the TCP protocol itself such as TCP fast open, but the fact that these predecessors were based on TCP limited their ability to manage transport layer level latency. QUIC was designed to combine the speed of UDP with the reliability and security features typically associated with TCP, along with the encryption capabilities of TLS-1.3 which is the next generation transport layer security protocol. As of 2023, 8.4% of all websites use QUIC, while majority of traffic is browser clients accessing Google services. It is intended as a replacement for the TLS/TCP stack for secure, authenticated communication over ordered channels. QUIC is designed to provide improved performance in various aspects, including initial latency, handling of multiple streams, mobility, data loss detection, and resistance to denial of service attacks. It will also serve as a basis for HTTP/3. As HTTP/3 becomes more widely adopted, QUIC will play an integral role in shaping the future of the web.

1.2 Problem formulation

1.2.1 Need for formal specification

QUIC is implemented in user-space in the application libraries as opposed to TCP to serve different implementations of QUIC based on one's use case and needs. This helps developers to push frequent updates with new features and optimizations without having to wait for kernel updates. Therefore, there exist several implementations of QUIC like Google QUIC, Mozilla, Cloudflare, MQTT over QUIC, msquic [1] by Microsoft and several open source implementations [2], [3], in many languages. These implementations need to conform to RFC 9000 which is a document published by IETF that formalizes the protocol's design, implementation guidelines, and standards to ensure

interoperability across the internet. A possible way to resolve the ambiguities and rigorously validate the protocol design is through formal verification, where a formal model of the protocol is first constructed and then analysed with respect to the specified security properties. Thus for the same, the RFC needs to be properly analysed, identifying protocol details such states, events, actions and constraint requirements. Message formats of the protocol, types of packets, header format, sequence numbers, flags and payloads should be properly defined. As QUIC protocol has in-built encryption layer, there should also be provision to manage the same. Entities participating in the model like client, server and middleboxes should be assigned proper roles. A proper formal modelling framework should be chosen to correspond protocol specifications. The formal model should be tested across different implementations to check their conformance to RFC specification.

1.2.2 Testing of transport layer protocols

Testing of transport layer protocols involve testing them to ensure protocol works according to the formal verification. Performance tests also include measuring efficiency of protocol under various conditions with parameters like throughput, latency, percentage of packet loss, congestion control, etc. Other tests include performance of protocol under different conditions of hardware and interoperability between different implementations of protocol. Security tests like injecting specific types of inputs to validate integrity of protocol. There are also various tools like Wireshark, netcat for that purpose. However, in this scenario, we need to test formal model across different implementations of QUIC. Therefore, it needs to be ensured that implementation to be tested in properly running the machine.

1.3 Literature Survey

We went through papers titled - Formal specification and testing of QUIC [4], Taking a long look at QUIC [5], Formal Analysis of QUIC Handshake protocol using Symbolic Model checking [6], QUIC not Quick [7]. There is also an update to the first paper which was looked upon later in time which works on draft-29 of QUIC [8]. These research papers were chosen based on the relevance to the study and number of citations. Formal specification and testing of QUIC [4] proposes a formal model in Ivy and is based on draft-18. Taking a long look at QUIC [5] conducts performance evaluation and discusses performance comparisons between TCP and QUIC. They try running QUIC across different scenarios, environment such as desktop and mobiles and network conditions. There also exists a replication to previous paper [9] which conducts performance evaluation between Google QUIC v37 and IETF QUIC v1. Along with these, brief references were taken from QUIC - A Quick Study, some research articles from ResearchGate and conferences from YouTube. Some of the implementations of QUIC including aioquic [2], picoquic [3], minq [10] were tested on my Arch Linux machine. A formal model based on Ivy [11] from the first paper was also tested based on the instructions given in this repository [11] and the formal model from the paper [8] is under process of testing in a docker container given in the repository [12].

Chapter 2

QUIC Protocol details

2.1 TCP and UDP

Transport protocol is of two types: Connection oriented and connection-less. Connection oriented requires additional work of setting up connections and provide reliability in terms of packet re-ordering, congestion control, reliable delivery of packets. On the other hand, connection-less transport protocol adapts, sends, and forgets mechanisms. Its entire purpose is to deliver the datagram without having to worry about its delivery. UDP is faster due to its lack of connection overhead, it is also less reliable and it's just IP with port addresses. TCP, on the other hand, ensures reliability but at the cost of speed due to its complex handshake and retransmission mechanisms.

2.2 Problems with TCP

2.2.1 Connection overhead

Since TCP is the most important connection-oriented protocol, it has been a center point of the research. Several improved versions of TCP have been published in the

last couple of decades. These TCP versions have mainly focused on the throughput. Although throughput is the main factor for determining the quality of traffic, TCP might not be suitable for a lot of different types of traffic like IoT and IIoT. Traffic in these technologies is characterized by short-lived bursts of exchanging small data chunks. For such short-lived connections, establishing connections every single time before an exchange of short live burst data is a lot of overhead. Connection start up latency in TCP is high, as it requires two round trips (RTT) for declaring a peer-to-peer connection alive and ready.

Applications required to have connection-oriented transport protocol such as HTTPS have some security enabled on top of transport layer. One of the most popular protocols used for such purposes is TLS (Transport Layer Security). This protocol provides security over the TCP by encrypting everything in the application layer. Encryption is based on the keys exchange between end entities. This process adds one or two RTTs into the additional two RTTs in connection establishment between end points after connection establishment. Securing transport layer is necessary as it provides symmetric encryption which enables authenticated, confidential and integrity-preserved communication between two devices. TLS protocol has its own handshake like TCP and it is unique per connection. So, a total of 3-4 RTTs are required to establish and secure a connection-oriented connection between end-to-end hosts.

2.2.2 Half-Open connection

Receiving data in TCP is passive. It means that dropped connections are only detected by the sender and receiver has no way of detecting them. Suppose a connection is being established between client and server. If a server fails to receive an ACK from client, this port is continued to be occupied and resources will only be released once the TCP state transitions to Closed State after timer expires.

2.2.3 Connection migrations

Flow is generally identified by a combination of several factors to identify single connection. Parameters involved in identifying a flow are source IP address, destination IP address, source port number, destination port number and the protocol (called 5-tuple). Some of these parameters (port numbers) are unique for a connection while others are common across devices (ip address). Changing the application or device can lead to a different identifier than the previous one. This will eventually lead to tear down of the established TCP connection. Such condition will force devices to establish a new TCP connection and perhaps negotiate TLS handshake again.

The second limitation that arises due to flow migration is NAT rebinding. Generally, users are behind firewall in their LAN network, with non-routable IP (such as 192.168.x.x or 172.16.x.x), these IP and port make a NAT table in the outgoing router to identify each device. Upon connection migration, this table can be inconsistent and can lead to breach in LAN hosts securities.

2.2.4 Head-of-line blocking

Head-of-Line blocking problem is a limitation phenomenon, which occurs when resources are being held up by some entity to complete an action. It only occurs if a connection-oriented transport protocol is used. This out of delivery of packets can happen due to several reasons, such as the lossy nature of the network, different paths taken by the packets and being dropped. HOL can significantly increase the packet reordering delay and which leads to consuming resources and latency in packets being processed by network stack and application. TCP enforces in-order delivery of data, meaning a single lost packet can block the delivery of subsequent packets until it is retransmitted and received. This becomes problematic for modern applications requiring high throughput and low latency.



Figure 2.1: Example of Head of line blocking in TCP.

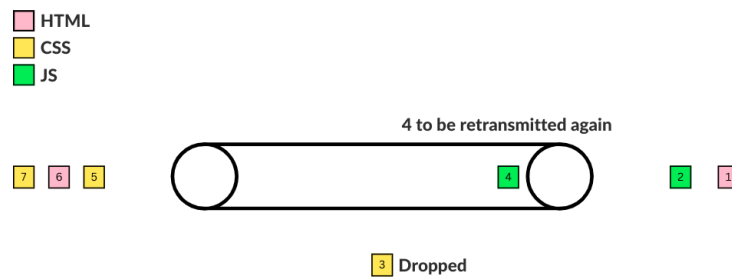


Figure 2.2: Example of Head of line blocking and retransmission in TCP.

As shown in Packets are coming from the server which are segregated into 3 different types of files by HTTP/2. Besides that, they will be transported through a single pipe (single TCP connection). For some reason, packet 3 which belongs to CSS is dropped. This further affects the next packet 4 from different type i.e. JS due to packet reordering.

2.2.5 Kernel space implementation

TCP has been integrated into kernel space since its early implementation in operating systems. and dates back to the initial adoption of the protocol in the 1980s. The integration allows direct interaction with hardware, network interface cards and efficient handling of tasks like packet routing, buffering, and retransmission. This also ensured uniform behaviour across all implementations. Updating TCP in kernel space requires OS updates and kernel recompilation, making protocol evolution slow. There are a growing number of middle-boxes on the Internet that block new TCP extensions.

2.3 History of QUIC

Google started working on QUIC in 2012. The next year they started small-scale experiments with the original versions of QUIC (Google QUIC now) with Chrome. In 2014, Chrome started a wide-scale deployment of Google QUIC in their browsers. The QUIC WG originated the specifications describing version 1 of QUIC. The WG acts as the reference point for any QUIC-related work in the IETF. Their tasks include maintenance and evolution of the QUIC base specifications that describe its invariants, core transport mechanisms, security and privacy properties, loss detection and recovery, congestion control, version and extension negotiation, etc. This includes the specification of new versions and extensions, and supporting the deployability of QUIC.

2.4 Protocol Stack

Delivering information over the internet is a complex operation that involves both the software and hardware level. One protocol cannot describe the entire communication flow due to the different characteristics of the devices, tools, and software used throughout the process. As a result, network communication is based on a stack of communication protocols in which each layer serves a different purpose. Although there are various conceptual models that describe the structure of protocol layers, such as the seven-layer OSI Model, the internet is based on the four-layer TCP/IP model, also known as the Internet Protocol Suite.

While it is connectionless at the lower level thanks to the underlying UDP layer, it is connection-oriented at the higher level due to its re-implementation of TCP's connection establishment and loss detection features that guarantee delivery. It integrates most features of the TLS v1.3 security protocol and makes them compatible with its own delivery mechanism, thus encryption moves down from the application layer to the transport layer.

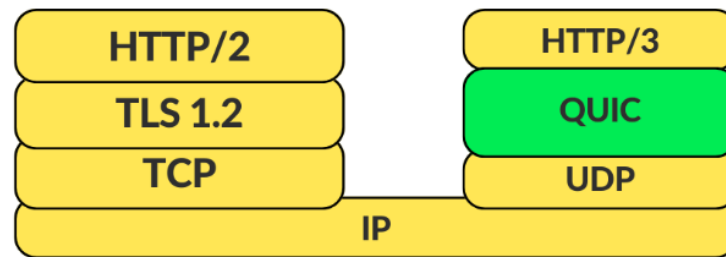


Figure 2.3: Protocol stack comparison

In the HTTP/3 stack, encryption is not optional but a built-in feature. HTTP/3 comes with a new underlying protocol stack that brings UDP and QUIC to the transport layer. HTTP/3 doesn't multiplex the connection between different streams as this feature is performed by QUIC at the transport layer. Transport-layer multiplexing removes the head-of-line blocking issue present in HTTP/2 (HTTP/1.1 doesn't have this issue because it opens multiple TCP connections and offers the option of pipelining instead of multiplexing, which turned out to have serious implementation flaws and was replaced with application-layer multiplexing in HTTP/2).

2.5 Features of QUIC

2.5.1 Connection Establishment

TCP in conjunction with TLS, requires 3-4 RTTs to establish connection before data being encrypted and sent. This overhead makes applications slower and with the increasing demands of bandwidth and application availability, this overhead can be a bottleneck. QUIC solves this problem by having at most 1 RTT for fresh connection and 0 RTT for repetitive connections. If a connection is being established between client and server for the first time, then it must perform 1 RTT handshake to get the necessary. Information. In 0-RTT scenario, server and client have communicated in the past and there is no need to establish connection again.

Feature	TCP	UDP	QUIC
Place in the TCP/IP model	On top of IPv4 or IPv6	On top of IPv4 or IPv6	On top of UDP
Connection type	Connection-oriented	Connectionless	Connection-oriented
Order of delivery	In-order delivery	Out-of-order delivery	Out-of-order delivery between streams, in-order delivery within streams
Guarantee of delivery	Guaranteed (lost packets are retransmitted)	No guarantee of delivery	Guaranteed (lost packets are retransmitted)
Handshake mechanism	Non-cryptographic handshake	No handshake	Cryptographic handshake
Security	Unencrypted	Unencrypted	Encrypted
Data identification	Knows nothing about the data it transports	Knows nothing about the data it transports	Uses connection IDs to identify the independent streams it transports

Table 2.1: Comparison of TCP, UDP, and QUIC

2.5.2 Improved Congestion Control

QUIC implements TCP cubic algorithm for congestion control. Each packet, whether it is original or re-transmitted, has a unique sequence number. This simple hack provides QUIC sender to distinguish between original and re-transmitted packets and solves TCP ambiguity problem. In addition to that QUIC ACKs also explicitly carry the delay between the acknowledgement sent and receipt of the packet. These sequence numbers are monotonically increasing numbers and do not get repeated in the connection lifetime.

2.5.3 Multiplexing

Client opens multiple TCP connections to server to fetch data from server. QUIC solves this problem by having multiple streams over one UDP pipeline. In QUIC, there is only one UDP connection for transport, while in the usage of TCP, there are multiple connections being established.

2.5.4 Forward Error Correction

Forward error correction mechanism is to recover the lost packets, without having to worry about re-transmission. QUIC can complement bunch of packets with an FEC packet. A perfect analogy would be RAID (Redundant Array of Independent Disks), just like RAID, FEC packet contains parity of the packets in the FEC group. If any packet is lost then content of the packet can be recovered from the FEC packet and the remaining packets in the group. It is up to the sender to decide whether to send the FEC packets for optimizing specific scenarios or not.

2.5.5 Connection Migration

TCP identifies flow by 5 tuple, connections in QUIC are being identified by a randomly generated 64 bit identifier called the Connection ID. In TCP, changing any parameter in 5-tuple can cause a connection break down and the session is no longer active. QUIC on the other hand, keep the Connection ID same and even if any parameter changes in 5-tuple underneath, it will not tear down the connection.

It enables connection mobility from one IP to other IP. For example, you are downloading a file at home through your Wifi, but now you have to travel in car, so you start using your mobile network.

2.5.6 Other features

QUIC is implemented in user space as opposed to TCP's implementation in the OS kernel. QUIC has been specifically designed to be deployable, evolvable and to have anti-ossification properties. Network stack sees QUIC packets as datagrams and doesn't require to reorder them. QUIC receives those datagrams and if implementation allows, can be pushed to application in any order necessary. This makes sure that packets are received as soon as network stack pushes them to the application. QUIC also supports multiplexing which makes these packets in the same UDP pipeline and once it is established, no additional connection establishment overhead is required.

2.6 Implementations of QUIC

Implementation	Language	Description
Chromium	C++	Source code of the Chrome web browser and the reference gQUIC implementation. Contains standalone gQUIC and QUIC client/server programs for testing.
ms-quic	C	A cross-platform QUIC implementation from Microsoft designed to be a general-purpose QUIC library.
mvfst	C++	Implementation of IETF QUIC protocol in C++ by Facebook.
kwik	Java	Client and server implementations of the QUIC protocol (RFC 9000) in 100% Java. Supports HTTP/3 (RFC 9114).
aioquic	Python	Library featuring an I/O-free API suitable for embedding in both clients and servers.
picoquic	C	Minimal implementation of QUIC aligned with IETF specifications. One of the best performers in interoperability testing. Needs picotls for encryption.
neqo	Rust	Implementation by Mozilla.
minquic	Go	Minq is a minimal implementation of QUIC which partly implements QUIC draft-05 with TLS 1.3 draft-20 or draft-21.

Table 2.2: Comparison of QUIC Implementations

Chapter 3

RFC 9000

3.1 MUST/MUST NOT Statements

Connection Management

Statement	Meaning
Endpoint MUST validate the source address before establishing a connection	To prevent spoofed addresses and attacks.
Clients MUST use a new connection ID for every connection attempt	Prevent linkability and confusion.
Endpoints MUST NOT reuse a connection ID on network paths.	Prevents tracking and security issues.
A QUIC endpoint MUST ensure that connection IDs are unpredictable.	To avoid connection hijacking.
A server MUST issue a new connection ID after receiving a packet with different destination connection ID	Allows for path migration and connection continuity.

Clients MUST NOT deliberately create connection with the same ID as another.	Prevents collisions and confusions.
Servers MUST NOT use connection IDs to authenticate or identify clients.	CIDss must not be treated as identity.
Endpoints MUST retire connection IDs that are no longer in use.	Maintains proper resource management.

Table 3.1: Connection Management MUST Statements from RFC 9000

Packet processing

Statement	Meaning
An endpoint MUST discard packets that are determined to be invalid.	No processing of corrupt or invalid data.
Receivers MUST NOT use unprotected packets to affect connection state.	Prevents injection of unencrypted data from causing issues.
Endpoints MUST support receipt of packets with non-zero length connection IDs.	Ensures compatibility and flexibility.
A client MUST NOT send a Retry packet.	Only servers issue Retry packets.
Endpoints MUST coalesce packets in a single UDP datagram, where possible.	To improve efficiency and reduce network load.
Endpoints MUST NOT send packets larger than the peer's maximum UDP payload size.	Prevents fragmentation and loss.
Endpoints MUST acknowledge received packets (send ACK frames).	Ensures reliability and enables retransmissions.

After sending CONNECTION_CLOSE, endpoints MUST NOT send more packets.	The connection is closed.
Endpoints MUST NOT send or accept packets for already closed connections.	Avoid resource and security issues.
Endpoints MUST NOT send Initial packets with source connection ID of zero length.	Prevents ambiguity at the connection start.
Endpoints MUST NOT send more than 3 coalesced packets in a single datagram.	Avoids UDP datagram size and processing issues.

Version and Compatibility

Statement	Meaning
Endpoints MUST verify QUIC packet version on receipt.	Ensures interoperability and correct protocol operation.
Clients MUST disregard Retry packets that are not valid for their connection.	Prevents hijacking or misuse.
Endpoints MUST respond to Initial packets with either an Initial or Version negotiation packet.	Initiates handshake or informs of incompatibility.

Table 3.3: Version and Compatibility MUST Statements from RFC 9000

Flow control

Statement	Meaning
Senders MUST NOT exceed advertised flow control limits from peers.	Prevents overwhelming the receiver.

Receivers MUST advertise increasing flow control limits as data is consumed.	Enables continued transmission.
Endpoints MUST check for flow control errors and terminate connections if detected.	Ensures safe buffer management.

Table 3.4: Flow control MUST Statements from RFC 9000

Frame and Stream Handling

Statement	Meaning
Endpoints MUST not send frames not defined in the spec unless negotiated.	Ensures only agreed frames are sent.
Receivers MUST advertise increasing flow control limits as data is consumed.	Enables continued transmission.
Endpoints MUST check for flow control errors and terminate connections if detected.	Ensures safe buffer management.
Implementations MUST process and properly ignore unknown or unrecognized frame types.	Ensures extensibility and future compatibility.
Endpoints MUST obey maximum stream and data limits set by their peer.	Prevents unintentional DoS.
Endpoints MUST treat stream data as ordered; streams MUST NOT deliver out-of-order data.	Ensures protocol semantics.
Endpoints MUST NOT exceed the peer's maximum number of permitted streams.	Prevents resource exhaustion.

Endpoints MUST send a HANDSHAKE_DONE frame at handshake completion, and MUST NOT send it after.	Properly signals transition to secure state.
---	--

Table 3.5: Frame and Stream Handling MUST Statements from RFC 9000

Error Handling and Closure

Statement	Meaning
On detected protocol violation, endpoints MUST send CONNECTION_CLOSE frame and terminate connection.	Handles fatal errors robustly.
After sending CONNECTION_CLOSE, an endpoint MUST NOT send further packets.	Shuts down cleanly.
Endpoints MUST not send packets to already close/expired connections.	Avoids undefined behavior.

Table 3.6: Error Handling and Closure MUST Statements from RFC 9000

Crypto and Security

Statement	Meaning
All critical transport state transitions MUST be made only using encrypted packets.	Prevents state manipulation by attackers.
Endpoints MUST discard unprotected packets after handshake completion.	Ensures only secure data is processed.

Endpoints MUST close the connection on authentication failure.	Prevents MITM and tampering.
Endpoints MUST not disclose connection IDs or packet numbers that would compromise security.	Protects privacy and integrity.

Table 3.7: Crypto and Security **MUST** Statements from RFC 9000

Migration and Path Management

Statement	Meaning
Endpoints MUST validate new paths using <code>PATH_CHALLENGE</code> and <code>PATH_RESPONSE</code> during migration.	Prevents attacks and link spoofing.
Migration MUST NOT be allowed unless the peer is verified on that path.	Mitigates hijacking and redirection risks.

Table 3.8: Migration and Path Management **MUST** Statements from RFC 9000

3.2 SHOULD/SHOULD NOT Statements

Connection and Endpoint Behavior

Statement	Meaning
Endpoints SHOULD minimize the use of long-lived connection IDs.	To reduce tracking risk and improve privacy.
A server SHOULD NOT reissue previously retired connection IDs.	To prevent confusion/linkability.

Endpoints SHOULD respond to packets with a zero-length connection ID appropriately (i.e., by dropping or responding with stateless reset).	Ensures protocol correctness when an unexpected zero-length ID is seen.
Endpoints SHOULD close the connection if an invalid connection ID is used.	To avoid further miscommunication or exploitation.

Table 3.9: Connection and Endpoint Behavior SHOULD Statements from RFC 9000

Packet Construction and Usage

Statement	Meaning
Implementations SHOULD limit datagram size to avoid IP fragmentation.	Fragmented packets are more likely to be lost.
Endpoints SHOULD NOT coalesce more than three packets into a single packet. Packets are more likely to be lost.	Keeps UDP datagrams reasonable in size and complexity.
UDP datagrams containing Initial packets SHOULD be padded to the minimum required size.	Defends against certain attacks and ensures network compatibility.

Table 3.10: Packet Construction and Usage SHOULD Statements from RFC 9000

Error Handling and Robustness

Statement	Meaning
Endpoints SHOULD send CONNECTION_CLOSE for protocol violations, rather than silently discarding packets.	Improves debuggability and interoperability.

Endpoints SHOULD discard packets with unknown or unsupported versions.	Only process what is understood.
--	----------------------------------

Table 3.11: Error Handling and Robustness SHOULD Statements from RFC 9000

Acknowledgment and Congestion Control

Statement	Meaning
Endpoints SHOULD bundle ACKs with outgoing packets when possible.	Reduces overhead.
Endpoints SHOULD NOT send standalone ACK-only packets when a data packet transmitting piggybacked ACKs is imminent.	Further reduces unnecessary traffic.
When under high loss or delayed ACKs, endpoints SHOULD acknowledge frequently enough to avoid unnecessary retransmissions.	Improves performance and prevents congestion window collapse.

Table 3.12: Acknowledgment and Congestion Control SHOULD Statements from RFC 9000

Migration and Path Management

Statement	Meaning
Endpoints SHOULD use new connection IDs when changing network paths.	Enhances privacy and state management.
On path migration, endpoints SHOULD limit the amount of data sent until the new path is validated.	Avoids data loss and attacks during migration.

Table 3.13: Migration and Path Management SHOULD Statements from RFC 9000

Security and Privacy

Statement	Meaning
Endpoints SHOULD frequently change active connection IDs to prevent correlation of activity by observers.	Defends against long-term surveillance.
Endpoints SHOULD NOT use predictable connection ID sequences.	Protects against linkage and security risks.

Table 3.14: Security and Privacy SHOULD Statements from RFC 9000

Timers and Timeouts

Statement	Meaning
Endpoints SHOULD persist important transport parameters and state across connection migrations.	Keeps connections robust and recoverable.
Implementations SHOULD use exponentially increasing backoff intervals when re-sending handshake packets.	Prevents network overload.

Table 3.15: Timers and Timeouts SHOULD Statements from RFC 9000

Miscellaneous

Statement	Meaning
Endpoints SHOULD limit the rate of stateless resets.	Protects against denial of service amplification.

Endpoints SHOULD keep idle connection state for a time sufficient to receive delayed packets (default: at least 3 times the PTO).	Handles late/straggler packets gracefully.
---	--

Table 3.16: Miscellaneous SHOULD Statements from RFC 9000

3.3 MAY/MAY NOT Statements

Connection and Endpoint Management

Statement	Meaning
Endpoints MAY use different connection IDs on packets sent to different peers. (Sec 5.1.1)	Allows use of separate connection IDs for each peer to improve privacy and flexibility.
An endpoint MAY change the connection ID it uses for a peer at any time. (Sec 5.1.1)	Connection IDs can be rotated at any time for privacy or migration.
A server MAY choose to pad the first flight of packets sent to a client to a size larger than 1200 bytes to avoid amplification limits. (Sec 8.1.1)	Padding initial packets helps the server send more data up front.

Table 3.17: Connection and Endpoint Management MAY Statements from RFC 9000

Packet Construction and Processing

Statement	Meaning
-----------	---------

A packet's payload MAY contain multiple frames from different streams. (Sec 12.3)	A packet can multiplex frames from several streams for efficiency.
A sender MAY coalesce up to three packets into a single UDP datagram. (Sec 12.2)	Up to three packets can be combined in one UDP datagram to minimize overhead.
Endpoints MAY send multiple Initial packets between the first few flights of handshake packets. (Sec 8.1.1)	Multiple Initials can be sent for handshake reliability.
Implementations MAY use larger datagrams if they know that the PMTU supports them. (Sec 14.1)	Larger datagrams may be used if MTU is proven sufficient.
Endpoints MAY discard Key Phases that are no longer needed. (Sec 7.2)	Old encryption keys can be deleted once not needed.

Table 3.18: Packet Construction and Processing MAY Statements from RFC 9000

Migration and Path Management

Statement	Meaning
An endpoint MAY choose to reset the congestion controller and round-trip time estimator when migrating to a new path. (Sec 9.4)	Congestion and RTT settings can be reset after migrating paths.
An endpoint MAY limit migration as a defense against attackers. (Sec 9.6)	Migration can be selectively limited for security.

Table 3.19: Migration and Path Management MAY Statements from RFC 9000

Error Handling and Closure

Statement	Meaning
A server MAY send a Version Negotiation packet in response to a datagram with multiple Initial packets that specify different versions. (Sec 6.2)	Servers can signal for version negotiation if conflicting versions are received.
When a stateless reset is received, an endpoint MAY choose to reestablish the connection. (Sec 10.3.1)	Connection may be restarted if a stateless reset is seen.
Endpoints MAY send CONNECTION_CLOSE frames in response to other error conditions. (Sec 10.2.1)	CONNECTION_CLOSE may be used for any errors.

Table 3.20: Error Handling and Closure MAY Statements from RFC 9000

Frames and Streams

Statement	Meaning
A single STREAM frame MAY include data from multiple streams. (Sec 19.8)	STREAM frames can carry data from several streams (rare).
Endpoints MAY use application protocols that do not require reliable, in-order delivery of data, as provided by streams. (Sec 2.2)	QUIC may be used for unreliable or unordered delivery.

Table 3.21: Frames and Streams MAY Statements from RFC 9000

Security and Privacy

Statement	Meaning
-----------	---------

An endpoint MAY choose to stop sending packets on paths where it suspects the peer of address spoofing. (Sec 9.6)	Stop sending if peer's address is suspect.
---	--

Table 3.22: Security and Privacy MAY Statements from RFC 9000

*

Miscellaneous

Statement	Meaning
A client or server MAY discard any packet that cannot be authenticated. (Sec 5.7)	Unauthenticated packets may be immediately dropped.
Endpoints MAY accept packets with a reserved version number during development or experimentation. (Sec 6.4)	Reserved versions can be accepted in testing.
An endpoint MAY choose to defer retirement of connection IDs to avoid excessive signaling. (Sec 5.1.1)	Delaying Connection ID retirement reduces signaling overhead.

Table 3.23: Miscellaneous MAY Statements from RFC 9000

Chapter 4

Traces

4.1 qlog

Logging at the endpoints is an effective approach for observing and analyzing how applications interact using network protocols, especially when these protocols employ encryption that obscures internal behavior from external observers. Often, applications rely on proprietary or non-standard logging formats, which can limit the effectiveness of analysis tools and techniques. This becomes particularly problematic during tasks like diagnosing interoperability issues between different implementations. The absence of a unified log format hinders the creation and adoption of universal tools that can interpret logs across various systems. To address this, qlog provides a flexible and structured format for logging network protocol events. Its design promotes easier data sharing and supports consistent debugging and analysis across multiple tools and platforms. qlog is documented in draft-ietf-quicklog-main-schema with latest version as draft-11 [13]. The qlog-main-schema document defines the main schema for capturing logs of events in QUIC and related protocols (notably HTTP/3, H3, QPACK), intended for easier debugging, and analysis of protocol behavior. qlog is a structured, JSON-based schema that standardizes how events from QUIC (and similar protocols) are logged, so that different tools and implementations can read, process and visualize these logs easily.

Inspired by Google's vix and other network trace visualization tools.

Contains:

- Timing information for connections and streams
- Packet send/receive events
- Loss and recovery, congestion control
- Version negotiation, handshake, and O-RTT/1-RTT transitions
- Stream-level data (open/close, data, flow control etc.)
- Error and warning events

DOES NOT Contain:

- Arbitrary user data (e.g., decrypted application payload)
- Secret cryptographic material
- Deeply implementation-specific/internal-only state unless explicitly logged

There are two top-level formats in qlog: An array of events default in the original versions. An object format preferred for streaming and incremental logging. A single trace only contains events for a single logical QUIC connection for either the client or the server. According to the document every event must have time field associated with it which marks the timestamp at which the event started.

qlog object format structure:

- `qlog_format`: "NDJSON" — "json"
- `qlog_version`: String, e.g., "0.3"
- `title`: (optional) Human-readable name
- `description`: (optional)
- `traces`: Array of trace objects, usually one per connection
- Each trace object contains:
 - `vantage_point`: Perspective ("client", "server", or "network")

- `common_fields`: (optional) Shared info (e.g., `group_id`, `protocol_type`)
- `event_fields`: (optional) Specifies field order for events
- `events`: Object mapping event IDs to event objects
- Each event:
 - * `time`: Time since start (ms or us)
 - * `category`: E.g., "transport", "recovery"
 - * `event_type`: E.g., "packet_sent", "stream_opened"
- `event_data`: Detailed fields (packet numbers, stream IDs, etc.)

qlog categories include:

- transport (low-level: packets, frames, connection)
- recovery (loss detection, retransmission, congestion control)
- http, qpack, etc. (application-layer info)
- security (TLS, keys, handshake events)
- connectivity (address validation, shutdown, etc.)

Notable Implementations:

- aioquic (Python)
- picoquic (C)
- quicly (C)
- lsquic (C)
- ngtcp2 (C)
- quiche (Rust)
- msquic (C)

4.2 tcpdump and tshark

tcpdump is a command-line packet capture tool for Unix-like systems. It captures network packets on interfaces, allowing you to analyze network traffic in real time or

save it for later analysis. Outputs human-readable summaries of network traffic or saves full packet data in a file (PCAP format).

tshark is the terminal-based companion to Wireshark, the world's most popular GUI network protocol analyzer. It can read packet captures (PCAP files), capture new packets from interfaces, and apply powerful filtering and decoding. It gives structured decoding of packets and can output in various text and data formats.

tcpdump and tshark both intercept and log packets passing through a network interface. Use tcpdump or tshark to capture traffic to a .pcap file. They let you inspect the details of the packets, including headers (IP, TCP, UDP), payloads, and protocol-specific information. Can filter specific traffic: by IP, port, protocol, etc. Packet capture often requires sudo privileges. pcap files can grow large, especially on busy networks. Using capture filters reduces noise and file size. Captured packets might contain sensitive information; handle files securely. Legal Ensure you have authorization to capture network traffic on your system or network.

Capture on localhost (100) for a specific process

```
1 sudo tcpdump -i eth0 udp port 443 -w quic_trace.pcap
```

Human readable format

```
1 sudo tcpdump i 100w capture.pcap
```

Capture only UDP QUIC packets

```
1 sudo tshark -i ethe -f "udp port 443" -w quic_capture.pcap
```

Output as JSON (e.g. for scripting or analysis)

```
1 tshark-r quic_capture.pcap -T json
```

Workflow for capturing traces is:

- Capture with tcpdump or tshark

- Save to a .pcap file for later inspection or directly use tshark to store it as .json.
- Analyze with tshark or Wireshark
- Open thepcap file for protocol-level inspection, filtering, and visualization.

Chapter 5

Methodology

5.1 Running Implementations

All the implementations have been tested on Linux Machine with Arch Linux. The modules required are installed in a python environment and the certificates and private keys required to run the server and client are provided by some of the implementations. Instructions to install are provided in the respective repositories but minquic [10] has not been updated since long so the instructions to install will be given. Below given implementations are chosen as they were supposed to work with Ivy Model as given by the repository. aioquic [2] and picoquic [3] have QUIC implementation conforming to RFC 9000 as described in their respective repositories.

5.1.1 aioquic

aioquic is a library for the QUIC network protocol conforming to RFC 9000 are written in Python. It features a minimal TLS 1.3 implementation, a QUIC stack and an HTTP/3 stack. In order to run the examples, create a python virtual environemnt and install the picoquic library or build the library and install necessary packages. In the figure 3.1, examples of server and client instances are running, the client uses HTTP/3 to request for files.

In figure 3.2 and 3.3, Wireshark is set up to capture packets on loopback address and the communication between server and client is captured. Client and Server establish handshake in 1-RTT with CRYPTO as frame and immediately start sending


```

aioquic:zsh ~ aioquic:zsh ~ sudo ~
aioquic:python
> python examples/http1_server.py --certificate tests/cert.pem --private-key tests/ssl_key.pem
2024-11-29 00:27:24,669 INFO quic [5c8549c3e18d0235] Negotiated protocol version 0x00000001 (VERSION_1)
2024-11-29 00:27:24,674 INFO quic [5c8549c3e18d0235] ALPN negotiated protocol hq-interop
2024-11-29 00:27:24,678 INFO quic [5c8549c3e18d0235] HTTP request GET / [1] [1] [5]
2024-11-29 00:27:24,678 INFO quic [5c8549c3e18d0235] Connection close received (code 0x100, reason )
2024-11-29 00:27:23,778 INFO quic [0dfae25fb7fc47] Negotiated protocol version 0x00000001 (VERSION_1)
2024-11-29 00:27:23,783 INFO quic [0dfae25fb7fc47] ALPN negotiated protocol h3
2024-11-29 00:27:23,784 INFO quic [0dfae25fb7fc47] HTTP request GET /
2024-11-29 00:27:23,787 INFO quic [0dfae25fb7fc47] Connection close received (code 0x100, reason )

aioquic:zsh ~ aioquic:zsh ~
aioquic:python
> source aioquic-em/bin/activate
> python examples/http1_client.py --ca-certs tests/pycert.pem --legacy-http https://localhost:4433/
2024-11-29 00:27:24,674 INFO quic [5c8549c3e18d0235] Negotiated protocol version 0x00000001 (VERSION_1)
2024-11-29 00:27:24,678 INFO quic [5c8549c3e18d0235] ALPN negotiated protocol hq-interop
2024-11-29 00:27:24,678 INFO client New session ticket received
2024-11-29 00:27:24,678 INFO client Response received for GET / : 1106 bytes in 0.0 s (2.693 Mbps)
2024-11-29 00:27:23,783 INFO quic [0dfae25fb7fc47] Connection close sent (code 0x100, reason )
2024-11-29 00:27:23,783 INFO client New session ticket received
2024-11-29 00:27:23,784 INFO client Response received for GET / : 1196 bytes in 0.0 s (8.251 Mbps)
2024-11-29 00:27:23,785 INFO client Push received for GET /style.css : 0 bytes
2024-11-29 00:27:23,785 INFO quic [0dfae25fb7fc47] Connection close sent (code 0x100, reason )

```

Figure 5.1: Client and Server output of aioquic on terminal

their payload of files. In figure 3.2, the connection IDs for source and destination is highlighted and the transport to QUIC is observed as UDP and QUIC IETF is also mentioned. The destination connection ID switches between two values as observed.

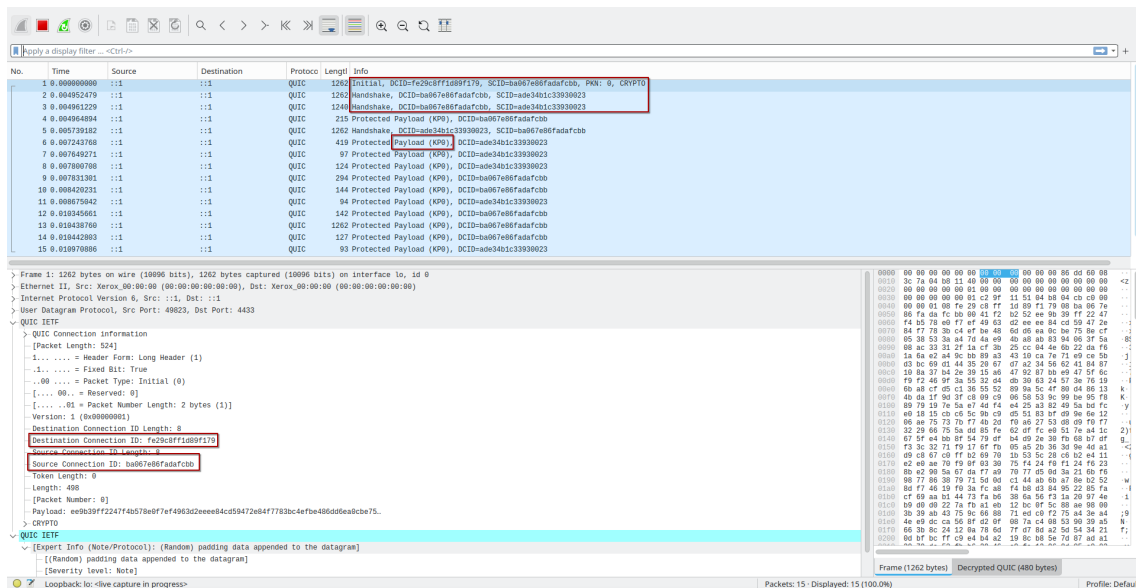


Figure 5.2: Running aioquic with Wireshark

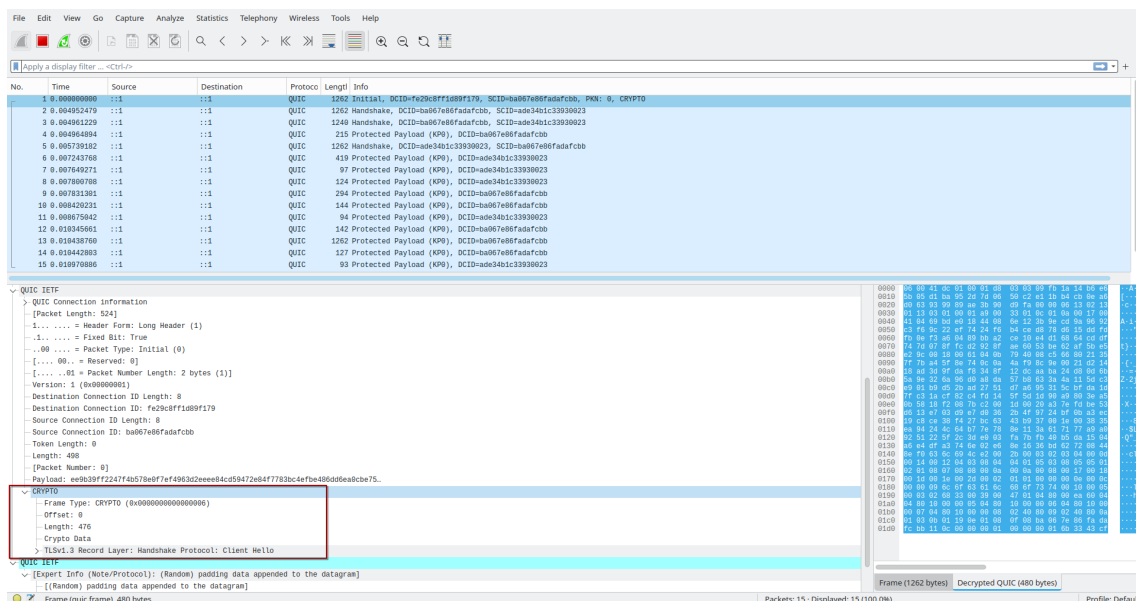


Figure 5.3: Frame details of QUIC in Wireshark

5.1.2 picoquic

The current version of Picoquic supports the QUIC specifications per RFC 9000, RFC 9001, RFC 9002, and RFC 8999. It is written in C. Building the project requires first managing the dependencies, Picotls and OpenSSL. Picotls is as TLS1.3 stack written in C. In order to run the implementation, build picotls as per the instructions. Then make build the picoquic implementation. In the figure 3.4, examples of server and client instances are running, the client uses HTTP/3 to request for files, this uses picoquicdemo instance to run.

Figure 5.4: Client and Server output of picoquic on terminal

In figure 3.5 and 3.6, Wireshark is set up to capture packets on loopback address and the communication between server and client is captured. Client and Server establish handshake in 1-RTT with CRYPTO, PING and PADDING as frames and immediately start sending their payload of files in the next transfer. In figure 3.5, the connection IDs for source and destination is highlighted and the transport to QUIC is observed as UDP and QUIC IETF is also mentioned. It uses Long header for handshake with packet number is also mentioned. In figure 3.6, the destination connection ID changes and short header used to give payload information. The packet number cannot be obtained as it is encrypted as shown. In the next half, you can observe ACK packets from the client but they are mentioned in the figures.

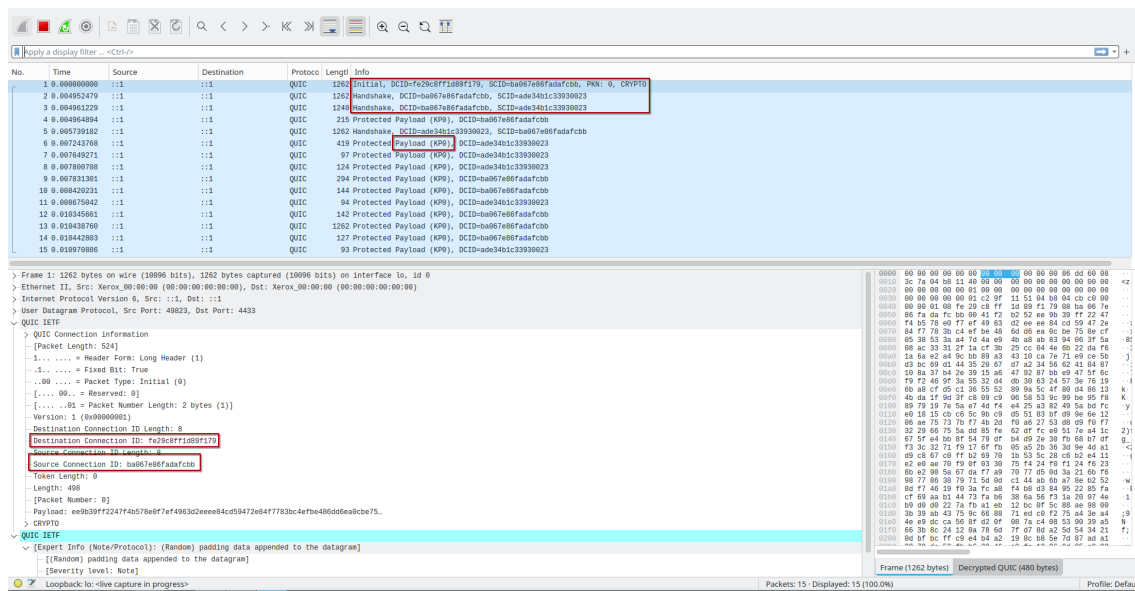


Figure 5.5: Running picoquic with Wireshark with handshake packet information

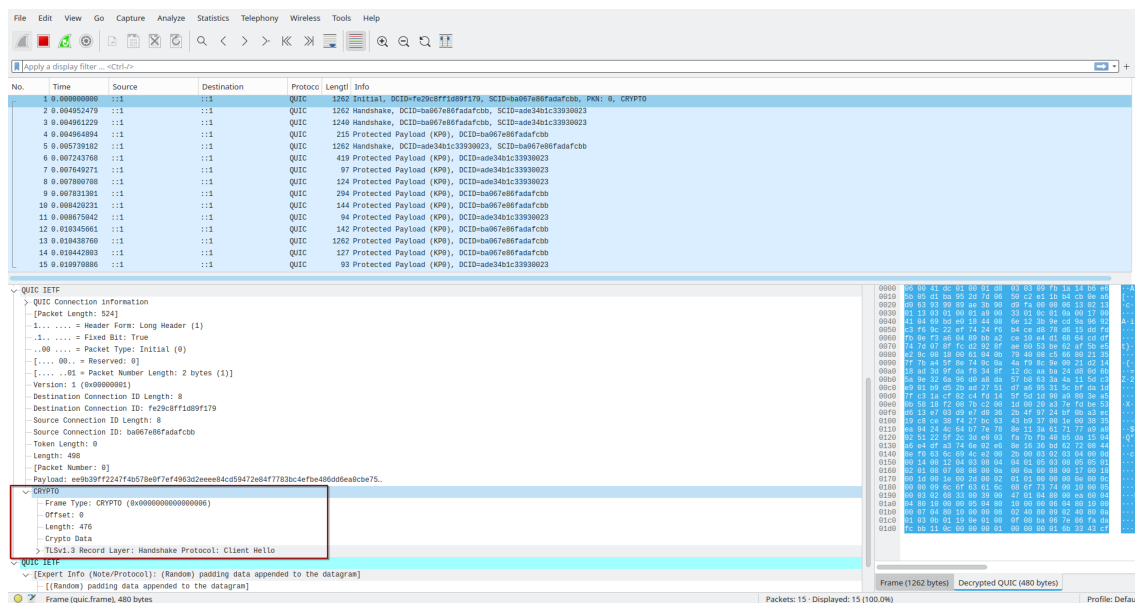


Figure 5.6: Running picoquic with Wireshark with payload packet information

5.1.3 minquic


Minq partly implements QUIC draft-05 (it advertises -04 but it's actually more like the editor's copy) with TLS 1.3 draft-20 or draft-21. It is written in Go and mentions some tests to ensure correct installation given in the figure 3.7. The installation instructions given in the repository uses deprecated methods of Go. For the correct installation, you have create a separate go file and include mint and minq as dependencies with their respective versions. Then install the dependencies or clone the github repositories and mention their paths. The implementation also requires you to include cloudflare cfssl

helpers v1.6.5 which is not mentioned in the repository.

[illegible]

Figure 5.7: Test program for minquic

There are two test programs that repository. The server is an echo server that upcases the returned data. The client is just a passthrough. The running instance is shown in figure 3.8. In figure 3.9, the packet information doesn't include QUIC as given in the examples of other implementations. The packet information only mentions it as UDP packet as this doesn't conform to RFC 9000.



```

go run ming/bin/server/main.go
2024/11/29 11:05:48 New connection
2024/11/29 11:05:48 State changed to: StateEstablished or role of unknown and known devices
2024/11/29 11:05:54 State changing
2024/11/29 11:05:55 State changed to: StateClosed
WARNING
Ming is absolutely not suitable for any kind of production use and should
of course, if explicitly doesn't violate conditions.

go run ming/bin/client/main.go
2024/11/29 11:05:48 PID: 8670
2024/11/29 11:05:48 Client com id=fba6142d7
2024/11/29 11:05:48 State changed to: StateWaitServerInitial
2024/11/29 11:05:48 State changed to: StateWaitServerFirstLight
2024/11/29 11:05:48 State changed to: StateEstablished
2024/11/29 11:05:48 Connection established server CID = 8fd99c29d
2024/11/29 11:05:54 State changed to: StateClosing
2024/11/29 11:05:54 Error Connection is closing

```

Figure 5.8: Client and Server output of minquic on terminal

No.	Time	Source	Destination	Protocol	Length	Info
1	4 0.000000000	127.0.0.1	127.0.0.1	ICMP	64	4433 Len=64
2	0.003550172	127.0.0.1	127.0.0.1	UDP	211	4433 -> 40441 Len=171
3	0.003595120	127.0.0.1	127.0.0.1	UDP	676	4433 -> 40441 Len=634
4	0.003784181	127.0.0.1	127.0.0.1	UDP	85	40441 -> 4433 Len=43
5	0.005015973	127.0.0.1	127.0.0.1	UDP	124	40441 -> 4433 Len=82
6	0.006039290	127.0.0.1	127.0.0.1	UDP	84	4433 -> 40441 Len=42
7	0.006087277	127.0.0.1	127.0.0.1	UDP	128	4433 -> 40441 Len=86
8	0.106608895	127.0.0.1	127.0.0.1	UDP	73	40441 -> 4433 Len=31

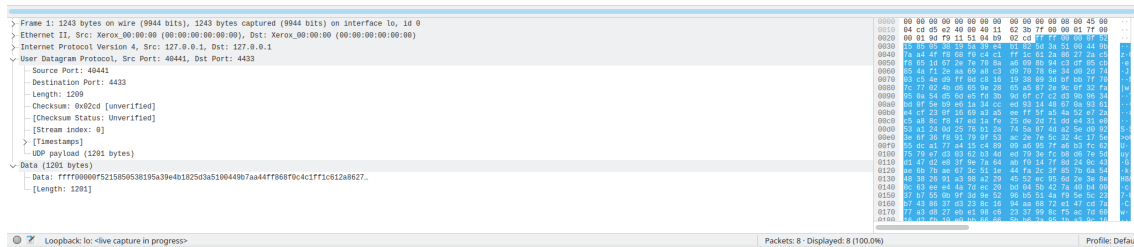


Figure 5.9: Running minquic with Wireshark with payload packet information

5.2 Interoperability Testing

This is a primary method used to date to validate QUIC. As QUIC standard exists as an English document, it is ambiguous and broadly open to interpretation. As a result, there are multiple independent implementations. These implementations represent a kind of commentary on the standard document, providing concrete interpretations where the language may be vague, unclear or contradictory. Interoperability is insufficient to guarantee that current implementations will be interoperable with future ones. For testing interoperability, QuicInteropRunner [14] is used.

In figure 3.10, interoperability tests between mvfst, kwik, picoquic, aioquic and msquic are shown. Rows are the clients and columns denote server. There are several tests like handshake, 1-RTT, 0-RTT, File transfer, multiplexing, handshake loss between the instances. We can extract the pcap files from the tests for checking the packet information on Wireshark.

Figure 3.11 shows transfer test between aioquic [2] and picoquic [3] in Wireshark. There is initial handshake, with then files being sent in the form of payload and then ACK received. So, there exists interoperability between these two implementations. Figure 3.12 shows 0-RTT test between aioquic [2] and picoquic [3]. Initially there is a handshake with 1-RTT and then the connection breaks and then again forms with 0-RTT as mentioned with directly transferring payload from the first packet itself, a different connection ID is also observed. The packet numbers are encrypted. Figure

3.13 shows unsuccessful handshake test between mvfst and msquic [1]. The two implementations try for handshake couple of times as shown in Wireshark with destination and source connection IDs being not properly shared.

Interop Status

	mvfst	kwik	picoquic	aioquic	msquic
mvfst					
kwik					
picoquic					
aioquic					
msquic					

Measurement Results

	mvfst	kwik	picoquic	aioquic	msquic
mvfst	G: 9381 (± 24) kbps	G: 8718 (± 140) kbps	G: 9220 (± 10) kbps	G: 8810 (± 12) kbps	G: 8663 (± 67) kbps
kwik	G: 9322 (± 18) kbps	G: 8484 (± 164) kbps	G: 9079 (± 27) kbps	G: 9043 (± 10) kbps	G: 8370 (± 977) kbps
picoquic	G: 9357 (± 19) kbps	G: 8645 (± 258) kbps	G: 9262 (± 5) kbps	G: 9112 (± 14) kbps	G: 8976 (± 31) kbps
aioquic	G: 9314 (± 14) kbps	G: 8822 (± 206) kbps	G: 9342 (± 27) kbps	G: 9234 (± 11) kbps	G: 8265 (± 483) kbps
msquic	G: 9430 (± 13) kbps	G: 8753 (± 163) kbps	G: 9314 (± 4) kbps	G: 8845 (± 12) kbps	G: 9092 (± 1) kbps

Figure 5.10: Interoperability tests

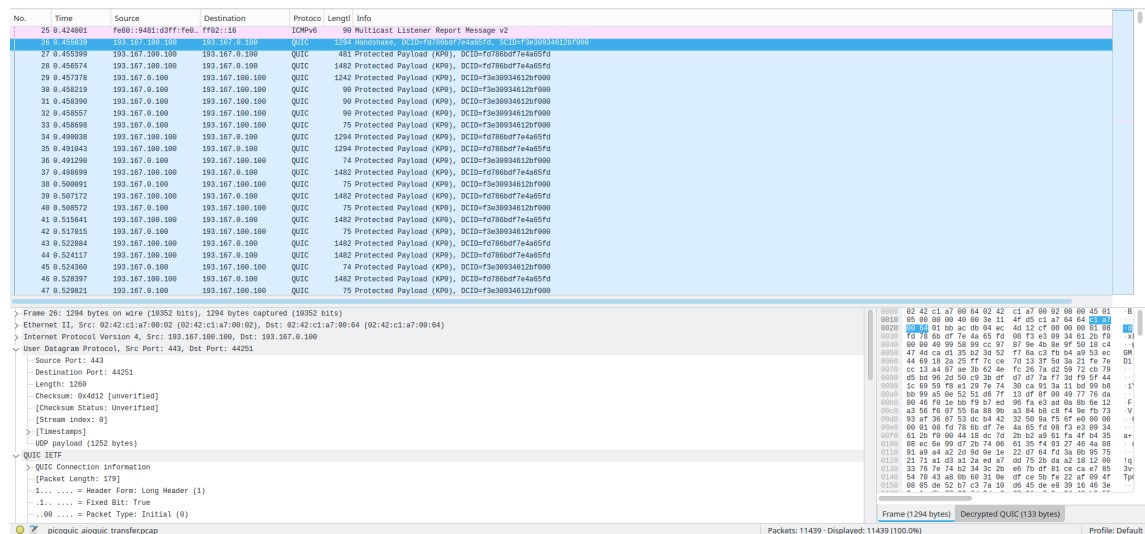


Figure 5.11: Example of transfer test for interoperability between aioquic and picoquic in Wireshark

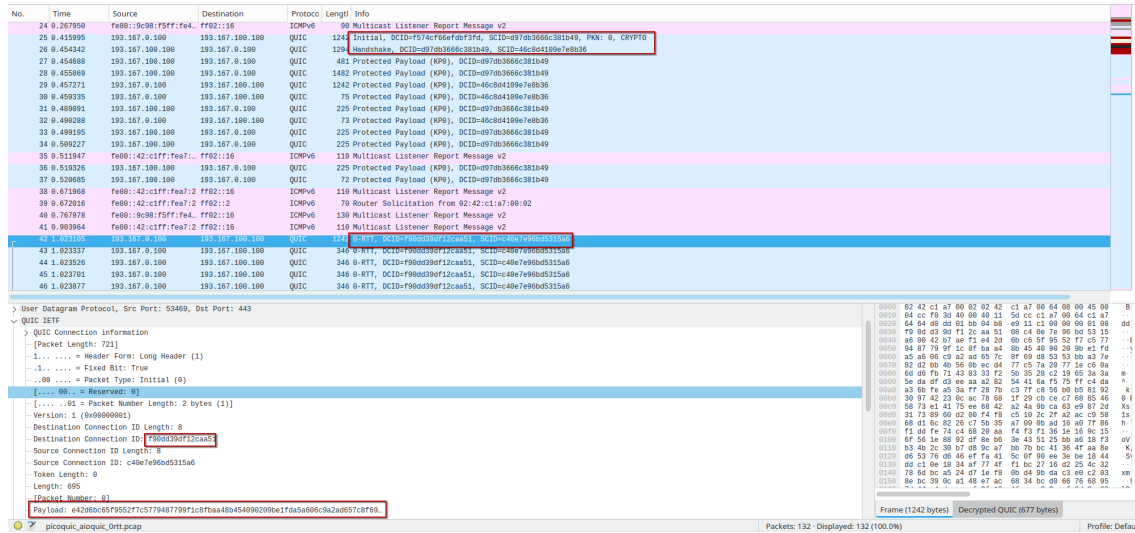


Figure 5.12: Example of 0-RTT test for interoperability between aioquic and picoquic in Wireshark

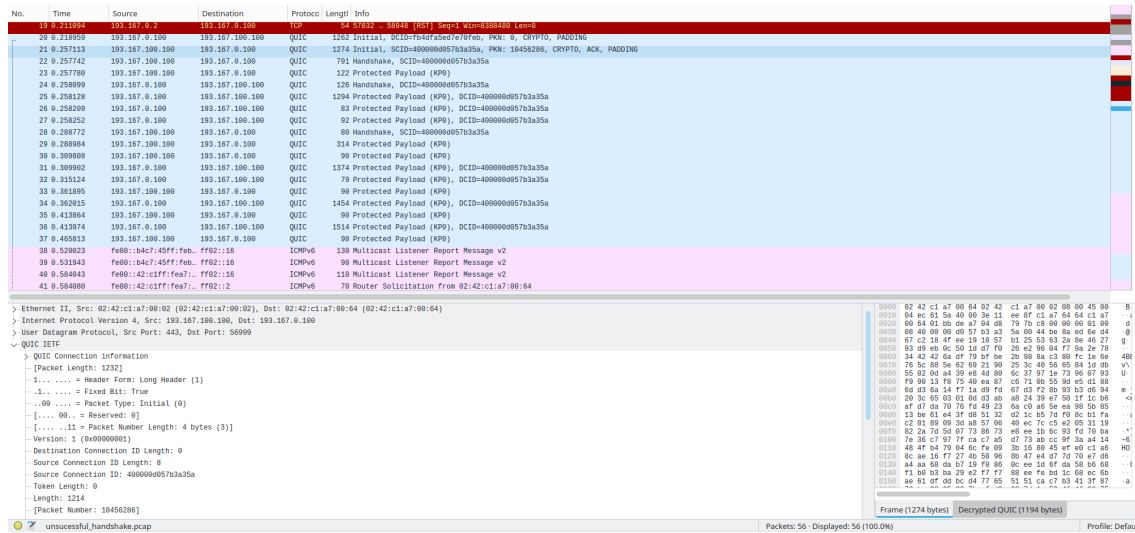


Figure 5.13: Example of unsuccessful transfer test for interoperability between mvfst and msquic [1] in Wireshark

5.3 Formal specification in Ivy

QUIC wire protocol is specified using an abstract machine that monitors protocol events. This specification [11] is coded in language called Ivy [15], given by Microsoft Research in the paper [4] which is archived since 2021.

5.3.1 Specification details

A protocol state is stored in a collection of mathematical functions and relations and the events associated with QUIC such as transmission are modeled by an action. This machine tracks all the QUIC-related events. It first consults its state to determine

whether the event is in fact legal according to the protocol. The Ivy tool can compile a generator that produces random sequences of events that conform to the specification. Specify a closed system of communicating agents, the same specification is used to generate tests for both client and server roles. Does not model QUIC as an FSM.

5.3.2 Test setup

The test setup is done in two systems - one following the draft-19 of QUIC specification [4] on Arch Linux and another updated version following draft-29 specification of QUIC [8]. The test setup of the draft-19 will be discussed here and the other specification currently runs inside an Ubuntu docker container and is under process. First steps of setup are setting up python environment to install or build Ivy cloning recursive submodules of the first work [11]. Then they have multiple tests on client and server for implementations and tests being written in ivy file need to be compiled. An example is given below. build stores the compiled tests and temp will store the results in .iev format.

```
1  mkdir build
2  mkdir test/temp
3  ivyc target=test quic_server_test_stream.ivy
```

Then set up environment variable QUICIMPLDIR pointing towards directory containing the QUIC implementations. According to test files, the tests should work on QUIC implementations such as picoquic [3], minquic [10] and quant. Among them, quant repository doesn't exist at all and thus no tests could be performed on quant. Following command is used to run tests on QUIC implementations. Among the tests compiled, the client tests have some issues in compiling. While in test.py, the imp module needs to be removed as it is deprecated for current versions of python.

```
1  cd test
2  python test.py iters=1 client=picoquic test=quic_client_stream
```


For the other setup, you need to clone the quic29 branch of the repository with updated implementation [12]. The dockerfile present needs to be updated with SSH keys and some the lines need to be removed. The container takes about 9 GB size with just picoquic and aioquic as implementations. As the size for volumes was increasing to about 16 GB, we are testing for limited implementations first.

1	<code>docker system df</code>				
2	TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
3	Images	18	1	8.968GB	6.935GB (77%)
4	Containers	3	0	42.24MB	42.24MB (100%)
5	Local Volumes	17	0	1.67GB	1.67GB (100%)
6	Build Cache	25	0	0B	0B

Run the setup with command where the dockerfile is located.

```
1  docker build --build-arg SSH_PRIVATE_KEY="$(cat ~/.ssh/id_rsa)" -t quic_env .
```

5.3.3 Results

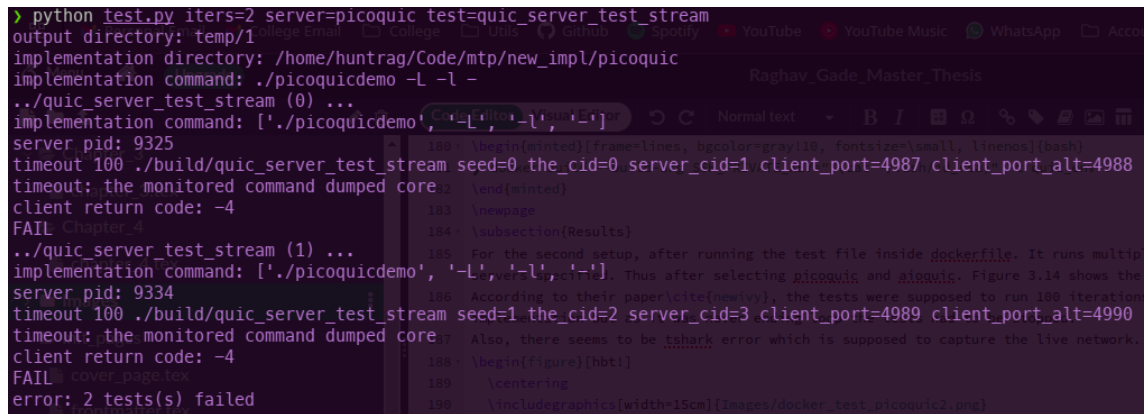
For the first setup, figure 3.15 gives the running instance of server test on picoquic and figure 3.16 gives the generated output. The output returns -4 and monitored command dumped core and error. The program was supposed to give below fragment as error with assumption being failed.

```

1  ../quic_server_test_max (0) ...
2  server pid: 9410
3  timeout 20 ./build/quic_server_test_max seed=0 the_cid=0 server_cid=1 client_port=4987 cl
4  quic_server_test.ivy: line 518: error: assumption failed
5  client return code: 1
6  FAIL
7  error: 1 tests(s) failed

```

Figure 3.15 and 3.16 show that the test started with creating server process and assigning ports with parameters being assigned for transfer of data, but there is an error. In the repository they have mentioned that the packet monitor doesn't work currently because the negotiated secrets from tls cannot be retrieved. Figure 3.17 and 3.18 give



```

> python test.py iters=2 server=picoquic test=quic_server_test_stream
output directory: temp/1
implementation directory: /home/huntrag/Code/mtp/new_impl/picoquic
implementation command: ./picoquicdemo -L -l -
../quic_server_test_stream (0) ...
implementation command: ['./picoquicdemo', '-L', '-l', '-']
server pid: 9325
timeout 100 ./build/quic_server_test_stream seed=0 the_cid=0 server_cid=1 client_port=4987 client_port_alt=4988
timeout: the monitored command dumped core
client return code: -4
FAIL
../quic_server_test_stream (1) ...
implementation command: ['./picoquicdemo', '-L', '-l', '-']
server pid: 9334
timeout 100 ./build/quic_server_test_stream seed=1 the_cid=2 server_cid=3 client_port=4989 client_port_alt=4990
timeout: the monitored command dumped core
client return code: -4
FAIL
error: 2 tests(s) failed

```

Figure 5.14: Test on picoquic server

the running instance of max stream server test on minquic and figure 3.16 gives the generated output. The output returns -4 and monitored command dumped core and error.

For the second setup, after running the test file inside dockerfile. It runs multiple tests on servers specified. Thus after selecting picoquic and aioquic. Figure 3.14 shows

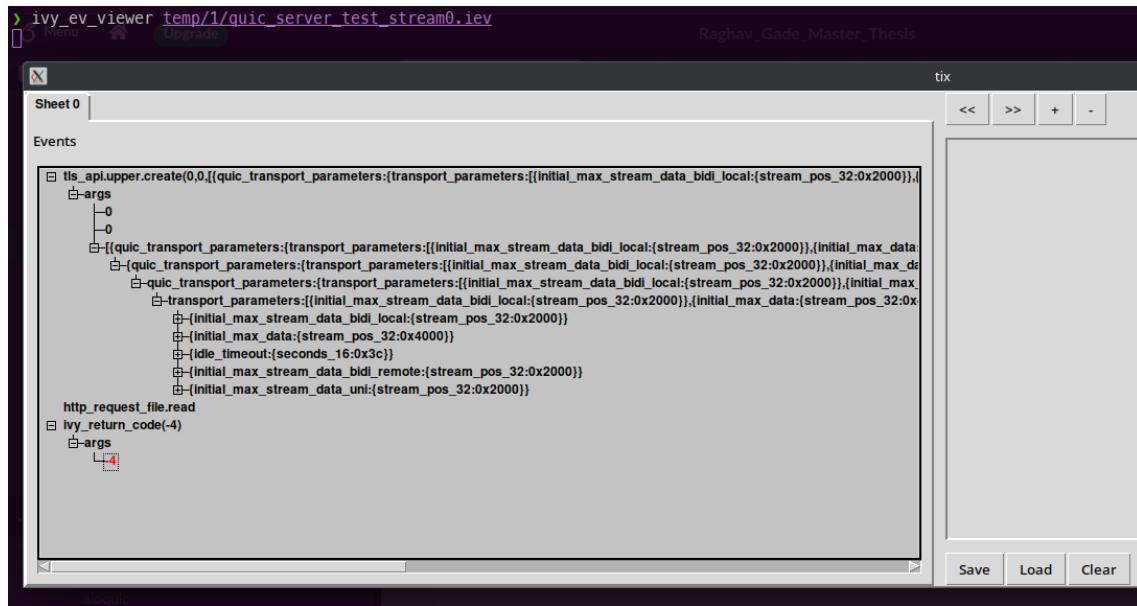


Figure 5.15: Output generated on picoquic server test

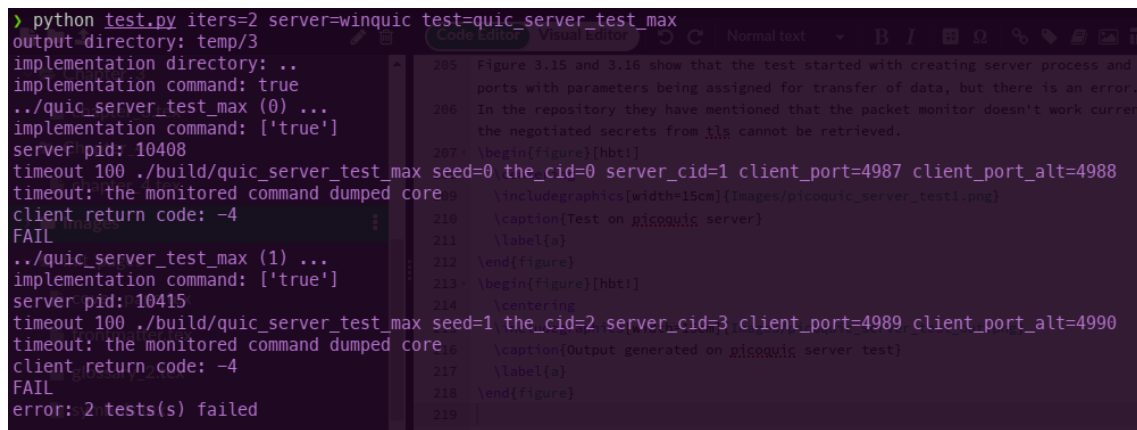


Figure 5.16: Max stream test on minquic server

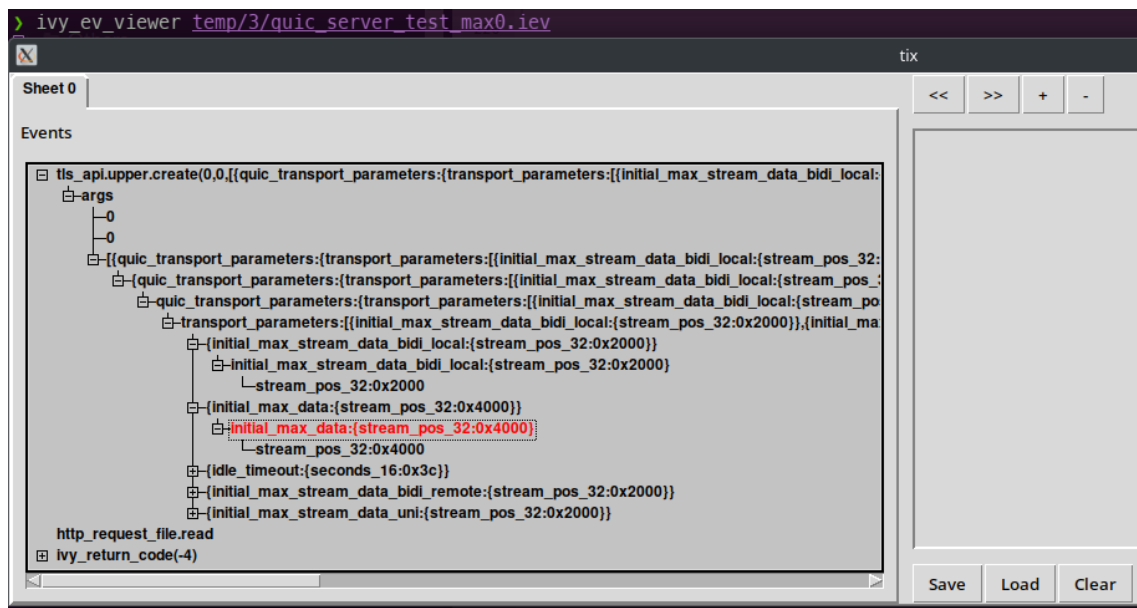


Figure 5.17: Output generated on minquic server test

the output. According to their paper [8], the tests were supposed to run 100 iterations of each implementation but as it was never ending loop the tests had to be stopped. Also, there seems to be tshark error which is supposed to capture the live network.

```

Running as user "root" and group "root". This could be dangerous.
Capturing on 'lo'
tshark: cap_set_proc() fail return: Operation not permitted
tshark: cap_set_proc() fail return: Operation not permitted

between random: 82942441 The default text editor in Ubuntu is GNOME Text Editor.
between random: 82942437 Features: Includes themes, dark mode, autosave, session restoration, and the
between random: 82942441 ability to zoom into text without changing the text's size
between random: 28081
" require f.data = crypto_data(scid,e).segment(f.offset,f.offset+f.length); # [2]
"
/usr/local/lib/python2.7/dist-packages/ivy/include/1.7/quic_frame.ivy: line 803: error: assumption failed
output directory: temp/13
implementation directory: /quic/picoquic
implementation command: ./picoquicdemo -l -D -L -q /results/picoquic_qlog
../quic_server_test_crypto_limit_error (0) ...
implementation command: ['./picoquicdemo', '-l', '-D', '-L', '-q', '/results/picoquic_qlog']
quic_process pid: 18614
timeout 100 ./build/quic_server_test_crypto_limit_error seed=495 the_cid=0 server_cid=1 client_port=4987 client_port_alt=4988
client return code: 1
FAIL
error: 1 tests(s) failed
test_server.sh: line 94: [: 15: unary operator expected

\Iteration => 15
\Implementation => picoquic
\Test => quic_server_test_crypto_limit_error

```

Figure 5.18: Test on picoquic server inside docker container

Results are not as expected and will require debugging. Currently, the docker environment is under work as it has an updated specification of QUIC and the environment supports deprecated version of modules required to test.

5.3.4 Expected results

Following results in table 3.1 were ran on Ubuntu-18 VM. These are results of 1000 iterations per test. These are the number of tests that passed.

Test Case	Picoquic	Quant
quic_server_test_stream	614/1000	362/1000
quic_server_test_max	682/1000	306/1000
quic_server_test_connection_close	92/1000	78/1000
quic_client_test_max	133/1000	0/1000

Table 5.1: Comparison of picoquic and quant test results

Following are the expected results in table 3.2 show the type of errors encountered for the tests we tried.

Test Case	Error Code	#
quic_server_test_stream	require $\sim_generating$ & $\sim_queued_non_ack(scid)$ $\rightarrow ack_credit(scid) \wedge 0$; [5]	242
	Ran out of values for type cid	121
	require $\sim_path_challenge_pending(dcid, f.data)$;	24
	bind failed: Address already in use	2
	require $conn_seen(dcid) \rightarrow hi_non_probing_endpoint(dcid, dst)$; [10]	1
	Total	390/1000
quic_server_test_max	require $stream_id_allowed(dcid, f.id)$; [4]	219
	Ran out of values for type cid	102
	require $\sim_path_challenge_pending(dcid, f.data)$;	8
	require $\sim_generating$ & $\sim_queued_non_ack(scid)$ $\rightarrow ack_credit(scid) \wedge 0$; [5]	12
	require $conn_total_data(the_cid) \wedge 0$;	31
	require $conn_seen(dcid) \rightarrow hi_non_probing_endpoint(dcid, dst)$; [10]	7
	bind failed: Address already in use	1
	Total	380/1000

Table 5.3: Error code distribution for picoquic server test cases

5.4 Formal specification in TLA+

5.4.1 qlog traces

TLA+ lets you write executable, mathematical models of distributed protocols like QUIC, capturing both intended behaviors and invariants precisely. We could catch subtle bugs in the protocol logic before implementation-especially for distributed, concurrent, or stateful systems by modelling them beforehand in TLA+. It lets you assert system invariants ("safety") and eventual guarantees ("liveness"), such as those required for QUIC. By loading traces, TLA+ provides a precise tool to retroactively check if trace logs comply with the RFC using formal logic (model-based trace validation). QUIC is highly asynchronous/parallel, with intricate packet and frame/stream relationships. Testing all possible combinations with ad-hoc methods is impractical.

The spec loads pre-captured QUIC event traces (ClientTrace and ServerTrace) and steps through them, simulating time. It models QUIC state transitions and then checks "safety" (invariant) and "liveness" (eventual guarantee) properties, many directly tied to QUIC RFC 9000 requirements.

5.4.2 Safety properties

Safety Properties (all must always hold; relate to RFC "MUST/MUST NOT").

Received packet must correspond to packet that was sent

Specification

```
Safety_NoUnknownRecv ==
  RecvPktNums(ClientTrace, ci-1) \subseq SentPktNums(ServerTrace, si-1) /\
  RecvPktNums(ServerTrace, si-1) \subseq SentPktNums(ClientTrace, ci-1)
```

Meaning: Each packet received by one endpoint must correspond to a packet actually sent by the other endpoint-this prevents packets "from the void".

RFC 9000 reference: *Section 12.2 Packet Processing: "QUIC packets that are determined to be invalid MUST be discarded."*

Section 7.2: "A QUIC receiver MUST ignore all packets that do not match a packet number expected as sent by the peer."

Packet numbers in sent packets must always increase

Specification

```
Safety_PktNosMonotonic ==
  LET
    cSentSeq == CollectSentIndices (ClientTrace, ci-1)
    cLen == Len(cSentSeq)
    sSentSeq == CollectSentIndices (ServerTrace, si-1)
    sLen == Len(sSentSeq)
  IN
    (cLen < 2 /\ (\A k \in 2..cLen :
      ClientTrace[cSentSeq[k]].data.header.packet_number >
      ClientTrace [cSentSeq[k-1]].data.header.packet_number))
    /\
    (sLen < 2 /\ (\A k \in 2..sLen :
```

```

ServerTrace[sSentSeq[k]].data.header.packet_number >
ServerTrace[sSentSeq [k-1]].data.header.packet_number))

```

Meaning: Enforces strict packet number ordering as required by QUIC.

RFC 9000 reference: *Section 12.3: "Packet numbers MUST be assigned in increasing order, beginning with the value 0 for the first packet sent on each connection."*

All stream IDs referenced in frames must have an explicit type set for them

Specification

```

Safety_AllUsedStreamsTyped ==
  StreamIDs(ClientTrace, ci-1) \subseq TypedStreams(ClientTrace, ci-1) /\
  StreamIDs(ServerTrace, si-1) \subseq TypedStreams(ServerTrace, si-1)

```

Meaning: Streams may only be used if their type has been declared-prevents undefined behavior.

RFC 9000 reference: *Section 3, Section 6.2 of HTTP/3, and stream handling in Section 2.2 of RFC 9000: "Stream type is set at creation, and endpoints MUST NOT send stream data on streams of unknown type."*

There can be at most one handshake_done or connection_close frame sent per trace.

Specification

```

Safety_AtMost1HandshakeClosePerTrace ==
  Cardinality(HandshakeSent(ClientTrace, ci-1)) <= 1 /\
  Cardinality(ConnCloseSent(ClientTrace, ci-1)) <= 1 /\
  Cardinality(HandshakeSent(ServerTrace, si-1)) <= 1 /\
  Cardinality(ConnCloseSent(ServerTrace, si-1)) <= 1

```

Meaning: Protocol allows at most one handshake completion and one connection close per endpoint trace.

RFC 9000 reference: *Section 10.2.1: "After sending a CONNECTION_CLOSE frame, endpoints MUST NOT send further packets." (Only one closure allowed per direction.)*

5.4.3 Liveness properties

Liveness Properties ("eventually...", usually "SHOULD"/"MUST" when an action is triggered):

Every sent packet must eventually be either received or dropped by the other endpoint

Specification

```
Liveness_EverySentPktHandled ==
  SentPktNums(ClientTrace, ci-1) \subseq (RecvPktNums(ServerTrace, si-1)
    → \cup DropPktNums(ServerTrace, si-1)) /\
  SentPktNums(ServerTrace, si-1) \subseq (RecvPktNums(ClientTrace, ci-1)
    → \cup DropPktNums(ClientTrace, ci-1))
```

Meaning: Ensures no sent packet is forever lost/unaccounted for in the logs.

RFC 9000 reference: *Section 12.2/12.3: "Receivers MUST track and acknowledge all packets received."*

Every stream whose type is set, is actually used in a frame

Specification

```
Liveness_AllTypedStreamsUsed ==
  TypedStreams(ClientTrace, ci-1) \subseq StreamIDs(ClientTrace, ci-1) /\
  TypedStreams(ServerTrace, si-1) \subseq StreamIDs(ServerTrace, si-1)
```

Meaning: Prevents spurious/pointless stream setups.

RFC 9000 reference: *Stream usage/multiplexing requirements, e.g. Section 2.2: "Endpoints MUST NOT use a stream before its type is set."*

After handshake_done is sent, at least one 1-RTT packet is sent

Specification

```

Liveness_HandshakeEnables1RTT ==

  \A t \in {"Client", "Server"}:

    LET

      trace == IF t = "Client" THEN ClientTrace ELSE ServerTrace

      hsidx == IF HandshakeSent(trace, Len(trace)) # {} THEN

        ↪ Min(HandshakeSent(trace, Len(trace))) ELSE Len(trace) + 1

      has1RTT == \E i \in SentEvents(trace, Len(trace)): i > hsidx /\

        ↪ trace[i].data.header.packet_type = "1RTT"

    IN hsidx <= Len(trace) => has1RTT

```

Meaning: Ensures the protocol transitions into the encrypted/1-RTT data phase post-handshake.

RFC 9000 reference: *Section 7.3 Section 4.9: "Endpoints MUST NOT send 1-RTT packets prior to handshake completion; endpoints MUST transition to 1-RTT after handshake_done."*

When a CONNECTION_CLOSE is sent, the peer eventually sees (observes) it

Specification

```

Liveness_ConnCloseEventuallyObserved ==

  \A t \in {"Client", "Server"}:

    LET

      trace == IF t = "Client" THEN ClientTrace ELSE ServerTrace

      other == IF t = "Client" THEN ServerTrace ELSE ClientTrace

      closeldx == IF ConnCloseSent(trace, Len(trace)) # {} THEN

        ↪ Min(ConnCloseSent(trace, Len(trace))) ELSE Len(trace) + 1

      recvClose == \E i \in ConnCloseRecv(other, Len(other)): i >= closeldx

    IN closeldx <= Len(trace) => recvClose

```

Meaning: Ensures connection closures are observable (no "silent close").

RFC 9000 reference: *Section 10.2.1: "An endpoint that receives a CONNECTION_CLOSE frame MUST close the connection."*

Every used/typed stream eventually sees a "fin" (finished) marker.

Specification

```

Liveness_TypedStreamEventuallyFin ==

  \A s \in (TypedStreams(ClientTrace, ci-1) \cup TypedStreams(ServerTrace,
    \rightarrow si-1)):
    (\E i \in 1..ClientLen:
      HasFrames(ClientTrace[i]) /\ \E f \in ClientTrace[i].data.frames:
        "stream_id" \in DOMAIN f /\ f["stream_id"] = s /\ "fin" \in DOMAIN f
        \rightarrow /\ f["fin"] = TRUE)
    /\
    (\E i \in 1..ServerLen:
      HasFrames(ServerTrace[i]) /\ \E f \in ServerTrace[i].data.frames:
        "stream_id" \in DOMAIN f /\ f["stream_id"] = s /\ "fin" \in DOMAIN f
        \rightarrow /\ f["fin"] = TRUE)

```

Meaning: All opened streams eventually complete and close.

RFC 9000 reference: *Section 3.1: "Streams are closed when all the data has been transmitted and received. FIN must be set at the end." Other properties (commented out/optional): Unique ack validation, unique stream type, etc. these could be enabled for stricter compliance.*

5.4.4 State variables

The state variables are indexes (ci, si) into the log for client and server. The Init state is the start of both traces. The state transitions are: ClientStep: Advance client by one event (simulate that endpoint processing). ServerStep: Advance server by one event. Stutter: No further steps when traces are complete (indexes at end).

[Next]_<<ci, si>> defines allowed next states. WF_<<ci, si>>(Next): Weak fairness (prevents starvation) is used so both traces will be advanced. Stutter state: When both traces are done, the Stutter transition keeps the model in its final state with no further activity, so properties like invariants remain checkable ("no more work to do").

5.4.5 Model checking

The TLA+ Toolbox provides an integrated environment for developing, simulating, and model checking TLA+ specifications. For a project such as the formal analysis of QUIC traces, some setup and customization are important to ensure the model checker works effectively on your specification and trace data.

Customizing the Toolbox for Model Checking

After writing the specification, the next step is to configure a *model* in the Toolbox. A model in TLA+ serves as a context or workspace for the model checker and simulation engine. It determines which properties to check, what constant values to use, and what exploration options are turned on.

Defining Constants and Supplying Traces

Constants in TLA+ are given meaning in the toolbox model, usually via a model's *What is the model?* tab. For a trace-based protocol analysis such as this, the constants `ClientTrace` and `ServerTrace` are populated with the concrete sequence of events you want to check. These are usually pasted in as TLC expressions (e.g., tuples or sequences) or loaded from a separate configuration file (`.cfg`). Example (inside the Toolbox's model editor):

```
CONSTANTS
```

```
ClientTrace = << [...], [...] >>
```

```
ServerTrace = << [...], [...] >>
```

It's good practice to keep the traces relatively small (typically 10–50 entries per trace for comfortable performance), as TLA+'s explicit state search will grow with the amount of input data.

Specifying Properties to Check

Properties-such as invariants (safety) and liveness conditions, are written in the TLA+ module as named definitions, for example:

```
SafetySpec == Spec => []SAFETY
LivenessSpec == Spec => <>LIVENESS
```

To have the Toolbox check these, you include them under the *What to check?* tab when creating or editing your model: For invariants, list the name(s) such as **SafetySpec** or any subproperty (e.g., **Safety_PktNosMonotonic**). For liveness properties, ensure the higher-level implication (as above) is selected, or list them individually or you can check several properties together. It is a good idea to start with invariants alone, as liveness checking is more resource intensive.

Model Checking: Options and Execution

In the *How to run?* tab, you can set model checking options:

- **Deadlock checking:** Enable by default to detect if the model can get stuck.
- **Depth limit:** Can optionally restrict the depth of state search, but is typically unlimited for full traces.
- **Symmetry:** Not needed for trace checking.
- **Workers / Parallelism:** If using a multicore CPU, increase the number of workers for faster state exploration.

For trace-driven analysis, most defaults work well, but avoid enabling features such as *random simulation*, as you want exhaustive checking of the input trace. To run the check, select "Run TLC" or "Model Check," and the Toolbox will begin enumerating all possible state progressions allowed by your specification, using the exact traces provided as constants.

How Verification Works

During its run, the TLC model checker: Enumerates all valid "next" state transitions over the input traces. At every step, evaluates your specified invariants and liveness

conditions. If an invariant is always true during all reachable states, it is *verified*. If not, TLC will halt with a counterexample, showing which step(s) fail and allowing you to inspect exactly where and why. For liveness, TLC ensures there is no execution where the desired eventual property is permanently violated. After checking, the Toolbox will present the state-space statistics, and indicate which properties were verified, and which (if any) failed. If a property fails, the tool will produce an error trace or counterexample for interactive exploration.

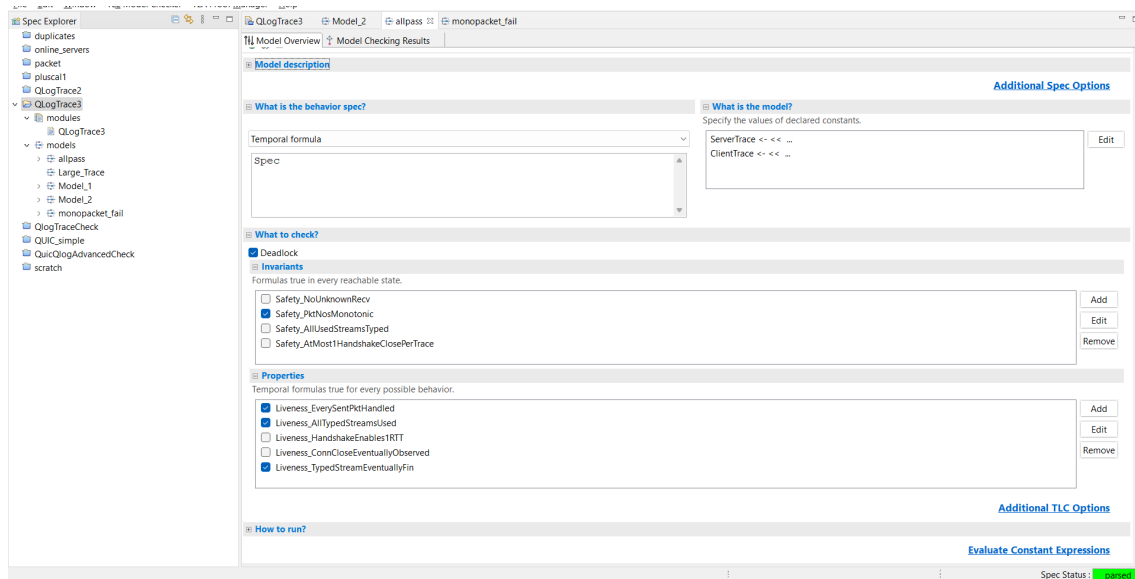


Figure 5.19: Safety and liveness testing configurations in Toolbox

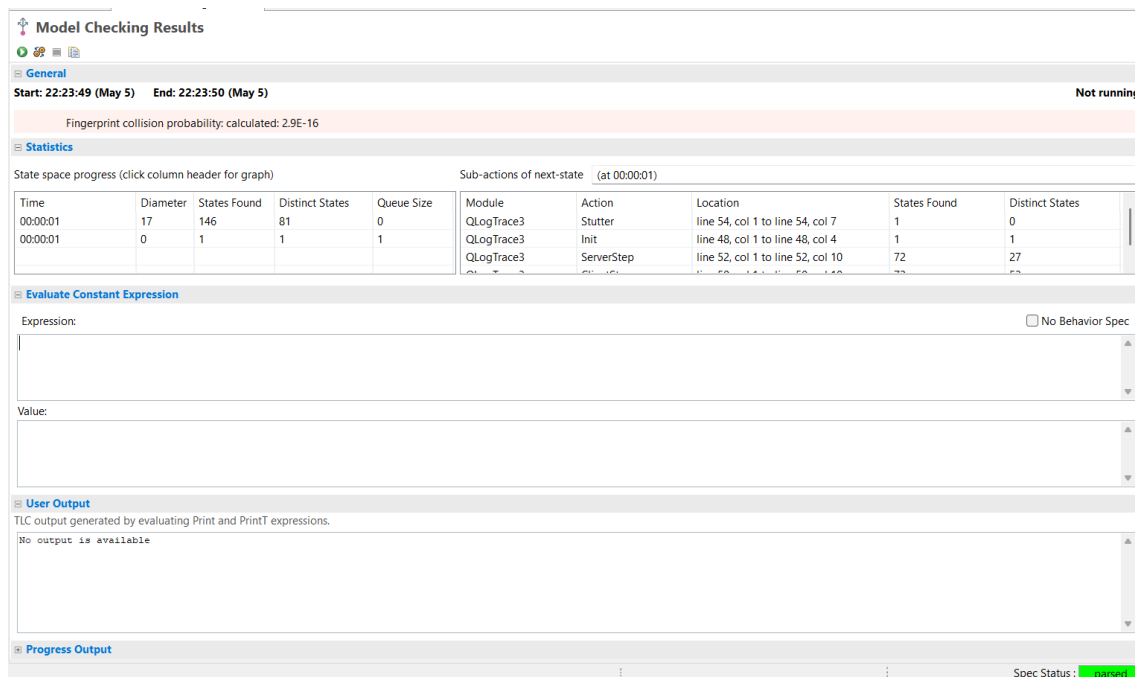


Figure 5.20: Results obtained after running the model checker

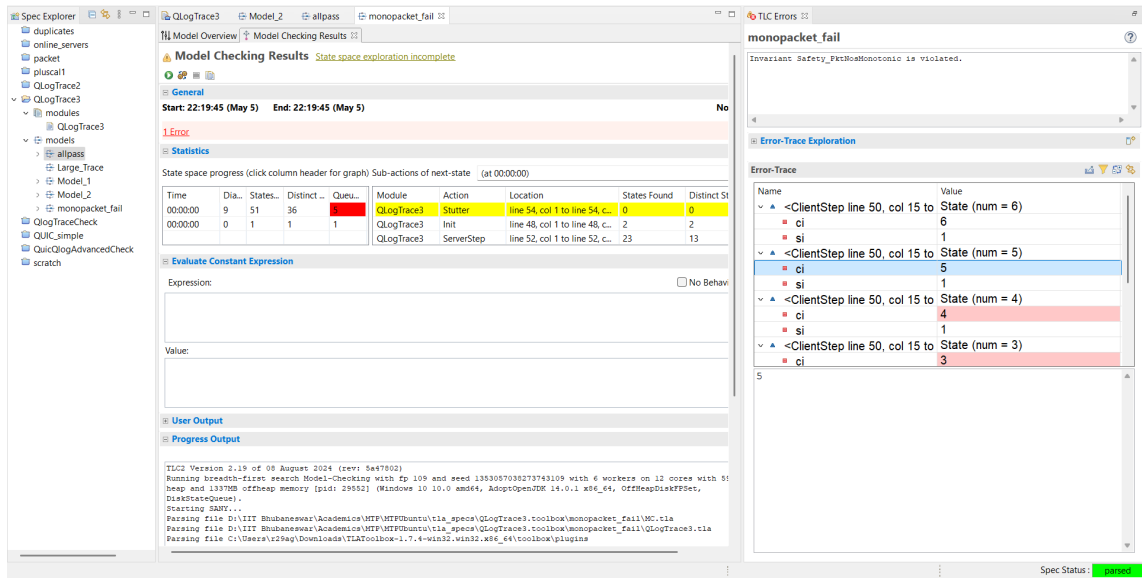


Figure 5.21: Model generating error trace after failing invariant

All the relevant code, examples traces and constants is available at https://github.com/huntrag/quic_spec.

5.4.6 Properties that cannot be satisfied

Incomplete Traces: If either trace is incomplete (missing close, unhandled streams, etc), liveness properties may fail (e.g., not all streams FIN'd).

Out-of-Order Logging: If log tool mismatches sending and receiving (e.g., due to clock skew), properties like NoUnknownRecv may be violated.

Corrupt Trace Data: E.g., duplicate packet numbers, streams with no types, orphaned FIN-would falsify monotonicity or typing invariants.

Violations Unobservable From Trace Alone: Some RFC properties (like cryptographic checks, anti-replay, key transitions) require data not present in event logs or not modelable from trace alone.

This TLA+ spec formalizes the core packet-exchange, stream, and state progress of QUIC and checks that the collected traces comply with several key RFC invariants.

Chapter 6

Conclusions and Future work

6.1 Conclusion

This thesis presents a formal approach to the analysis and validation of the QUIC protocol using TLA+. The primary goal of this work was to verify that concrete executions, captured as qlog traces from real client and server interactions, conform to the expected invariants and liveness properties derived from RFC 9000. Through systematic modeling and model checking, we gain both practical assurance in the protocol’s correctness and deeper insights into its operational guarantees and potential weaknesses. Our TLA+ specification encodes essential behaviors such as packet sending and receiving, proper assignment of stream types, monotonic increasing packet numbers, handshake progression, and correct management of connection closures. Each of these properties corresponds closely to either a "MUST" or "SHOULD" requirement in RFC 9000. When supplied with well-structured traces, our specification upholds all safety and liveness properties: packet numbers are always assigned in a strictly increasing manner for each endpoint, each typed stream is properly finished, handshaking transitions smoothly to the data transfer phase, and connections are closed in a way observable by both endpoints. These results confirm the protocol’s robustness under compliant conditions and the soundness of our formal approach.

6.2 Comparison with Previous Approaches

Historically, most protocol verification, especially for earlier protocols like TCP, relied on English-language descriptions, reference implementations, or limited model checking on abstracted versions of the design. These methods often missed corner cases or subtle concurrency bugs. QUIC was designed to overcome some limitations of its predecessors by introducing features like connection migration, stream multiplexing, and integrated transport-level encryption, but also introduced more complexity. Prior formal efforts-such as those using IVY, the tool by Kenneth McMillan-focus on deriving implementations or proofs from abstract state machine descriptions. IVY excels at discovering invariants and producing refinement mappings that can be used for protocol synthesis or deep verification. Our TLA+ approach, by contrast, is optimized for direct trace validation. It excels at asserting that a series of real-world messages and events, as recorded in production or test deployments, actually correspond to valid protocol executions per the RFC. This makes it valuable not only for academic specification but also for practical regression and interoperability testing. In comparing the two, we note that IVY's strengths in protocol synthesis and refinement could nicely complement TLA+'s strengths in trace-driven validation. For example, invariants derived in IVY could be imported into a TLA+ model for runtime checking, or executions validated in TLA+ could be used as test cases for synthesized protocols in IVY.

6.3 Limitations and Areas for Further Work

Although the TLA+ model robustly captures trace-level correctness for the most critical safety and liveness properties, several meaningful challenges remain. First, the translation from qlog or production trace to TLA+ data structures, though highly automated in our tools, remains a potential source of subtle error. In this work, we specifically used qlog as the input format for traces. Qlog is now widely adopted in the QUIC community and provides a normalized way to represent protocol events. However, qlog is designed for debugging, and not all protocol information is recorded in standard traces. Fields may be missing, abbreviated, or implementation-specific;

timestamps can be inconsistent or ambiguous. Mapping qlog events precisely to all the protocol-level invariants required by the formal model can thus be challenging and can introduce translation gaps, especially for edge cases or when the trace does not record internal state transitions in sufficient detail. Furthermore, some protocol features-such as cryptographic key transitions, congestion control nuances, or detailed retransmission logic-are only partially observable or not present at all in typical qlog output, and may warrant deeper formalization or richer trace capture in the future. A further limitation encountered in practice is related to resource usage by the TLA+ model checker (TLC). The checker assumes all relevant events are captured and that event order and timing are faithfully preserved. In reality, traces captured directly from production implementations can quickly exceed reasonable memory and time limits when used as TLA+ constants, especially for long-running or high-throughput sessions. In our work, we found that using raw traces from implementations often led to TLC running out of memory or becoming unresponsive, even on systems with 8GB or more of RAM. As a result, we relied on hand-crafted or filtered traces, using qlog as an intermediate, to keep the analysis tractable. This restriction means our analysis focuses on smaller, manageable slices of protocol behavior, rather than capturing the full range of a real deployment's complexity. For very large traces (hundreds or thousands of steps), state space reduction techniques or more scalable encoding could help, but those were outside the scope of this thesis. Another limitation is that non-deterministic, probabilistic, or adversarial network behaviors-such as random loss, reordering, or active attacks-are not described explicitly in our present models. While the current properties can be extended to cover some of these behaviors by introducing additional fairness constraints and nondeterminism, Further work is necessary to model these situations realistically capturing every relevant threat and fault model remains an open research problem. and to adapt verification techniques so that they remain both meaningful and tractable when analyzing such complex scenarios.

Appendix A

Ivy concepts

Introduction

Ivy's language is designed to let you to specify and implement systems in a way that is both convenient and conducive to automated verification. Technically, this means that important verification problems like invariant checking and bounded model checking fall within a decidable fragment of first-order logic. Ivy divides the world into three categories of things: Data values, Function values and Procedures.

Type Object, State and Action

Ivy uses the concept of "type objects" which have a "state" which can be manipulated through "action". `link` is a set of pairs (X,Y) where X and Y are nodes. Identifiers beginning with capital letters are logical variables, or place-holders. "action" is Ivy's notion of a procedure that mutates the values of state variables.

```
1  type node
2  relation link(X:node,Y:node)
3  type id
4  function node_id(X:node) : id
5  individual root:node
6  type color = {red,green,blue}
7
```

```

8  action connect(x:node, y:node) = {
9      link(x,y) := true
10 }
11
12 #below action removes links from x to all nodes in the set failed:
13 action clear_failed(x:node) = {
14     link(x,Y) := link(x,Y) & ~failed(Y)
15 }

```

A.1 Control

We can execute two actions sequentially by separating them with semicolon. A semicolon is a sequential composition operator in Ivy. Ivy uses the call-by-value convention. Calls are executed in left-to-right order.

```

1  #following syntax can be used
2  #to find a element of a type that satisfies some condition:
3  if some x:t. f(x) = y {
4      z := x + y
5  }
6  else {
7      z := y
8  }
9  #we can find an element of a set s with the least key like this
10 if some x:t. s(x) minimizing key(x) {
11     ...
12 }

```

A.2 Non deterministic choice

“*” can be used to represent non-deterministic choice in Ivy in two situations. First, on the right-hand side of an assignment and second in a conditional to create a non-deterministic branch.

```

1  #first
2  link(x,Y) := *
3  #second
4  if * {
5      link(x,y) := true
6  } else {
7      link(x,z) := true
8  }
```

A.3 Monitors

Ivy is the monitor. The QUIC specification is based on that concept. Monitors are useful to separate the specification of an object from the object itself. Monitors are objects in Ivy that contain the specification but that do not execute them. They enable us to control the flow of the specifications’ executions.

```

1  around app_server_open_event {
2      require conn_requested(dst, src, dcid);
3      require ~connected(dcid) & ~connected(scid);
4
5      call map_cids(scid, dcid);
6      call map_cids(dcid, scid);
7      ack_credit(scid) := ack_credit(scid) + 1;
8  }
9
10 before app_server_open_event {
```

```

11     require conn_requested(dst, src, dcid);
12     require ~connected(dcid) & ~connected(scid);
13 }
14
15 after app_server_open_event {
16     call map_cids(scid, dcid);
17     call map_cids(dcid, scid);
18     ack_credit(scid) := ack_credit(scid) + 1;
19 }

```

There are two important keywords: **before** and **after**. What is inside the **before** Statement will be executed each time there is an event `app_server_open_event`. Consequently, this example requires the presence of a connection request and ensures that the client's CID for this connection and the CID selected by the server for this connection are not already part of an active connection. The **after** statement is similar: it executes its content each time an `app_server_open_event` is generated.

A.4 assume, require and ensure

The primitive actions `require` and `ensure` allow us to write specifications. These actions fail if the associated condition is false. In `require`, `ensure` and `assume` actions, any free variables are treated as universally quantified.

```

1  # require
2  # connect action to handle only the case where
3  # the node y is not in the failed set.
4  action connect(x:node, y:node) = {
5      require ~failed(y);
6      link(x,y) := true
7  }
8
9  #ensure takes responsibility of the action itself

```

```
10  # to ensure the truth of the formula
11  action increment(x:node) returns (y:node) = {
12      y := x + 1;
13      ensure y > x
14  }
15
16  #assume
17  # action non-deterministically chooses
18  # a non-failed node and connects x to it
19  action connect_non_failed(x:node) = {
20      y := *;
21      assume ~failed(y);
22      link(x,y) := true
23  }
```

Appendix B

TLA+ concepts

B.1 Introduction to TLA+

TLA+ (Temporal Logic of Actions) is a formal specification language developed by Leslie Lamport for modeling and verifying concurrent and distributed systems. It combines first-order logic, set theory, and temporal logic to describe system behaviors at a high level of abstraction.

Why Use TLA+

Engineers and researchers use TLA+ for several compelling reasons. TLA+ allows detection of subtle issues in system designs before implementation. Specifications serve as unambiguous documentation of system behavior. TLA+ excels at modeling complex concurrent systems where traditional testing often fails. The TLA+ model checker (TLC) can exhaustively check specifications against properties. TLA+ models focus on essential aspects of system behavior, abstracting away implementation details.

How TLA+ Works

TLA+ describes systems as state machines where states are assignments of values to variables, behaviors are infinite sequences of states, actions are relations between consecutive states, and properties are expressed as temporal formulas over behaviors. The TLA+ approach typically involves modeling system states using variables, defining

initial conditions, specifying transitions between states (actions), expressing properties the system should satisfy, and verifying these properties using the TLC model checker.

B.2 Setting Up TLA+

To work with TLA+, you need the TLA+ Toolbox, which can be downloaded from <https://lamport.azurewebsites.net/tla/toolbox.html>. After installation, you can create a new specification in the Toolbox. A basic structure of a TLA+ module is shown below:

```

MODULE Example

EXTENDS Naturals, Sequences, TLC

VARIABLES var1, var2

Init ==

/ var1 = 0
/var2 = <<>>

Next ==

\\var1' var1 + 1
/var2' Append(var2, var1)

\ / \ var1' = 0
/\ var2' = <<>>

Spec == Init /\ [] [Next]_<<var1, var2>>

```

B.3 Key TLA+ Operators

TLA+ provides powerful operators for expressing complex behaviors.

Logical Operators

Logical operators include \wedge (conjunction, AND), \vee (disjunction, OR), \sim (negation, NOT), \Rightarrow (implication), and \models (equivalence).

Action Operators

Action operators include priming (') which indicates a variable's value in the next state, `[A]_v` which means action A or variables in v remain unchanged, and `<A>_v` which means action A must occur and variables in v must change.

Set Operators

Set operators include `\in` for set membership, `\subseteq` for subset relationship, `\union` for set union, `\intersect` for set intersection, `\` for set difference, and `x \in S: P(x)` for set comprehension.

B.4 Important Keywords and Constructs

CHOOSE

Selects an arbitrary element from a set that satisfies a predicate:

```
CHOOSE x \in S: P(x)
```

IN

Used in local definitions and set comprehensions:

```
{x \in 1..10 x % 2 = 0}
```

LET

Defines local variables within an expression:

```
LET x == 42 y == x + 1 IN x * y
```

CARDINALITY

Returns the number of elements in a finite set:

```
CARDINALITY({1, 2, 3}) = 3
```

VARIABLES

Declares state variables of the specification:

```
VARIABLES counter, queue, status
```

CONSTANT

Defines model parameters that remain fixed:

```
CONSTANT MaxClients, Nodes
```

EXTENDS

Imports definitions from other modules:

```
EXTENDS Integers, Sequences, FiniteSets
```

DOMAIN

Returns the domain of a function:

```
DOMAIN [a -> 1, b -> 2, c -> 3] = {"a", "b", "c"}
```

RECURSIVE

Defines recursive functions:

```
RECURSIVE Sum(_)
```

```
Sum(S) == IF S = {} THEN 0 ELSE LET x == CHOOSE n \in S: TRUE IN x + Sum(S \ {x})
```

B.5 Sets and Sequences

Sets

Sets are fundamental in TLA+:

```
\* Set enumeration
```

```
Colors == {"red", "green", "blue"}
```

```

/* Set comprehension
EvenNumbers == {n \in 1..100 n % 2 = 0}

/* Set operators
AllNumbers == OddNumbers \union EvenNumbers
Common Elements == Set1 \intersect Set2

```

Sequences

Sequences (ordered tuples) are represented as:

```

/* Sequence literal
seq1 == <<1, 2, 3, 4>>

/* Sequence operations
Head(seq1) = 1
Tail(seq1) = <<2, 3, 4>>
Append(seq1, 5) = <<1, 2, 3, 4, 5>>
seq1[2] = 2 * 1-indexed
Len(seq1) = 4

```

B.6 Temporal Operators and Properties

Safety Properties

Safety properties state that "nothing bad happens" and can be expressed as invariants:

```

/* Invariant: queue never exceeds maximum capacity
QueueSizeInvariant == Len(queue) <= MaxQueueSize

/* Type invariant
TypeInvariant ==
/ counter \in Nat
/ queue \in Seq (Message)
/ status \in {"ready", "busy", "failed"}

```

Liveness Properties

Liveness properties state that "something good eventually happens":

```
\* Eventually messages are delivered
```

```
MessageDelivery == [] <> (queue = <<>>)
```

```
\* If a request is received, it is eventually processed
```

```
RequestProcessing == [] (requested => <> (processed))
```

Bibliography

- [1] microsoft, “msquic implementation,” 2024. [Online]. Available: <https://github.com/microsoft/msquic>
- [2] aiortc, “aioquic implementation,” 2024. [Online]. Available: <https://github.com/aiortc/aioquic>
- [3] private octopus, “picoquic implementation,” 2024. [Online]. Available: <https://github.com/private-octopus/picoquic>
- [4] K. L. McMillan and L. D. Zuck, “Formal specification and testing of quic,” in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 227–240. [Online]. Available: <https://doi.org/10.1145/3341302.3342087>
- [5] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, “Taking a long look at quic: An approach for rigorous evaluation of rapidly evolving transport protocols,” in *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. New York, NY, USA: Association for Computing Machinery, November 2017, pp. 290–303. [Online]. Available: <https://doi.org/10.1145/3131365.3131368>
- [6] J. Zhang, X. Gao, L. Yang, T. Feng, D. Li, and Q. Wang, “A systematic approach to formal analysis of quic handshake protocol using symbolic model checking,” *Security and Communication Networks*, vol. 2021, pp. 1–13, August 2021, academic Editor: Ruhul Amin. [Online]. Available: <https://doi.org/10.1155/2021/1630223>
- [7] X. Zhang, S. Jin, Y. He, A. Hassan, Z. M. Mao, F. Qian, and Z.-L. Zhang, “Quic is not quick enough over fast internet,” in *Proceedings of the ACM Web Conference 2024 (WWW '24)*. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 2713–2722. [Online]. Available: <https://doi.org/10.1145/3589334.3645323>
- [8] C. Crochet, T. Rousseaux, M. Piraux, J.-F. Sambon, and A. Legay, “Verifying quic implementations using ivy,” in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC (EPIQ '21)*. New York, NY, USA: Association for Computing Machinery, December 2021, pp. 35–41. [Online]. Available: <https://doi.org/10.1145/3488660.3493803>
- [9] N. Muthuraj, N. Eghbal, and P. Lu, “Replication: ”taking a long look at quic”,” in *Proceedings of the 2024 ACM on Internet Measurement Conference (IMC '24)*. New York, NY, USA: Association for Computing Machinery, November 2024, pp. 375–388. [Online]. Available: <https://doi.org/10.1145/3646547.3688453>

-
- [10] ekr, “minq implementation,” 2024. [Online]. Available: <https://github.com/ekr/minq>
 - [11] K. McMillan, “formal specification of quic in ivy,” 2018. [Online]. Available: <https://github.com/kenmcmil/ivy/tree/master/doc/examples/quic>
 - [12] ElNiak, “formal specification of quic in ivy- draft 29,” 2021. [Online]. Available: https://github.com/ElNiak/QUIC-Ivy/tree/quic_29/doc/examples/quic
 - [13] R. Marx, L. Pardue, T. Pauly, and D. Schinazi, “qlog: main schema,” <https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/>, March 2024, work in Progress, IETF Internet-Draft draft-ietf-quic-qlog-main-schema-19.
 - [14] M. Seemann and J. Iyengar, “Automating quic interoperability testing,” in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ '20)*. New York, NY, USA: Association for Computing Machinery, August 2020, pp. 8–13. [Online]. Available: <https://doi.org/10.1145/3405796.3405826>
 - [15] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: Safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 614–630. [Online]. Available: <https://doi.org/10.1145/2908080.2908118>