"Towards verification of QUIC and its extensions"

Crochet, Christophe ; Sambon, Jean-François

ABSTRACT

QUIC is a new transport protocol that aims to be widely used over the Internet. It is built above UDP and associates the benefits of TCP and TLS. The specification of this protocol just finished being standardized with the RFC9000. Compliance to this specification is essential to ensure future web safety and security An attempt to the verification of QUIC was done by McMillan & Zuck by using a methodology called "Network-centric Compositional testing". Ivy, a tool and language developed by Microsoft, was used in this methodology to perform a formal verification of the QUIC protocol. In this master thesis, we propose an extension to this work. First, we update McMillan & Zuck QUIC model to one of the latest draft. This draft is similar to the RFC9000. Then, we added the verification of 41 properties that were not present in the original model. One QUIC extension was also verified. Finally, we tested eight open source implementations to the compliance of all the modelled requirements. We found several errors and highlighted ambiguities that are still present in the RFC9000. The usefulness of our work was demonstrated by fixing a QUIC implementation with the errors we found. We proposed possible extensions to our work and we made publicly available all the resources needed to perform these extensions.

CITE THIS VERSION

UCLouvain

epl

École polytechnique de Louvain

Towards verification of QUIC and its extensions

Use of an automated tool for the formal verification of the QUIC protocol and extensions

Authors: **Christophe Crochet, Jean-François Sambon**
Supervisors: **Olivier Bonaventure, Axel Legay**
Readers: **Charles Pêcheur, Maxime Piraux**
Academic year 2020–2021
Master [120] in Computer Science

Acknowledgements

We would like to thank our supervisors Prof. Axel Legay and Prof. Olivier Bonaventure. Their reviews on our work were precious and helped us to improve the quality of our master's thesis. We are truly grateful to Maxime Piraux for his continuous support through the year. We also thank Tom Rousseaux for his suggestions on our work. Finally, we are grateful to Kenneth McMillan and L.D Zuck. This thesis would not have been possible without their previous work. We also thank Prof. Charles Pêcheur to have accepted to be the reader of our master's thesis. The configurations of the tested implementations were realized thanks to the help of François Michel.

Crochet Christophe & Sambon Jean-François

Abstract

QUIC *is a new transport protocol that aims to be widely used over the Internet. It is built above* UDP *and associates the benefits of* TCP *and* TLS*. The specification of this protocol just finished being standardized with the* RFC-9000 *[1]. Compliance to this specification is essential to ensure future web safety and security.*

An attempt to the verification of QUIC *was done by McMillan & Zuck by using a methodology called "Network-centric Compositional testing" [2].* Ivy*, a tool and language developed by Microsoft, was used in this methodology to perform a formal verification of the* QUIC *protocol [3]. In this master thesis, we propose an extension to this work [4]. First, we update McMillan & Zuck* QUIC *model to one of the latest draft. This draft is similar to the* RFC-9000*. Then, we added the verification of 41 properties that were not present in the original model. One* QUIC *extension was also verified.*

Finally, we tested eight open source implementations to the compliance of all the modelled requirements. We found several errors and highlighted ambiguities that are still present in the RFC-9000*. The usefulness of our work was demonstrated by fixing a* QUIC *implementation with the errors we found. We proposed possible extensions to our work and we made publicly available all the resources needed to perform these extensions.*

Glossary

Ack-eliciting Packet "A QUIC packet that contains frames other than ACK, PADDING, and CONNECTION_CLOSE. These cause a recipient to send an acknowledgment; see Section 13.2.1" [5]. 54

Connection ID "An opaque identifier that is used to identify a QUIC connection at an endpoint. Each endpoint sets a value for its peer to include in packets sent towards the endpoint." [5]. 3

Draining packet An endpoint enters the draining state if this endpoint sends a CONNECTION_CLOSE frame and this frame is acknowledged. If the sender is in the draining state, then we consider those packets as a "draining packet".. 55

Formal verification "Formal verification is a way to describe mathematically a system. It allows extracting the key properties and the ambiguities by highlighting *what* the system, i.e. QUIC, should do rather than *how* it should behave. However, it can be difficult to transform a protocol specification written in natural language form in a formal one.". 1

IETF stands for "Internet Engineering Task Force", it tasks is to publishes RFCs for methods, behaviours, research or innovations applicable to the Internet. 1

IP stands for "Internet Protocol". 3

Ivy "IVy is a tool for specifying, modelling, implementing and verifying protocols. IVy is intended to allow interactive development of protocols and their proofs of correctness and to provide a platform for developing and experimenting with automated proof techniques. In particular, IVy provides interactive visualization of automated proofs, and supports a use model in which the human protocol designer and the automated tool interact to expose errors and prove correctness."[6]. 2

Mirror "The mirror is the process that allows `Ivy` generating legal and random packets. "[2]. 20

Network-centric Compositional testing "Methodology used by McMillan and Zuck to verify the `QUIC` protocol. It allows testing the compliance of Network Protocol to a standard. It is done by doing a composition of all the specification of the different component of a system [2].". 10, 23

Out-of-order packet "A packet that does not increase the largest received packet number for its packet number space (Section 12.3) by exactly one. A packet can arrive out of order if it is delayed, if earlier packets are lost or delayed, or if the sender intentionally skips a packet number" [5]. 8

Probing packet " `PATH_CHALLENGE`, `PATH_RESPONSE`, `NEW_CONNECTION_ID`, and `PADDING` frames are "probing frames", and all other frames are "non-probing frames". A packet containing only probing frames is a "probing packet", and a packet containing any other frame is a "non-probing packet"." [5]. 52

QUIC stands for "Quick `UDP` Internet Connections". 1

QUIC Frame "The payload of `QUIC` packets, after removing packet protection, consists of a sequence of complete frames, as shown in Figure 10. Version Negotiation, Stateless Reset, and Retry packets do not contain frames."[5]. 15

QUIC Packet "QUIC endpoints communicate by exchanging packets. Packets have confidentiality and integrity protection; see Section 12.1. Packets are carried in `UDP` datagrams;"[5]. 12

Shim "The shim is the interface between `Ivy` abstraction and physical event on the wire."[2]. 28

Stream "A unidirectional or bidirectional channel of ordered bytes within a `QUIC` connection. A `QUIC` connection can carry multiple simultaneous streams. [5]". 15

TLS stands for "Transmission Control Protocol". 1

UDP stands for "User datagram protocol". 1

Contents

Glossary **iii**

1 Introduction **1**
1.1 Testing QUIC . . . 3
1.2 Formal verification . . . 4
1.3 Key steps . . . 5
1.4 Limitations . . . 6

2 State of the art **10**
2.1 `QUIC` . . . 10
2.1.1 `QUIC` history . . . 11
2.1.2 `QUIC` features . . . 12
2.2 Formal verification & `Ivy` . . . 19
2.2.1 Network-centric Compositional Testing . . . 19
2.2.2 Concepts of `Ivy` . . . 20
2.2.3 `QUIC` specification in `Ivy` . . . 23
2.3 Related work . . . 29

3 Contributions **30**
3.1 Update to `QUIC` 29 . . . 30
3.1.1 QUIC errors . . . 32
3.1.2 QUIC unknowns . . . 33
3.1.3 QUIC transport parameters . . . 34
3.1.4 IPv6 model . . . 34
3.1.5 Modelling a `QUIC` extension . . . 35
3.1.6 Shim update . . . 36
3.1.7 QUIC sending/receiving FSM . . . 36
3.2 Refactoring of the model . . . 37
3.3 Ivy improvements . . . 37
3.4 Server Tests . . . 38
3.4.1 Global tests . . . 38
3.4.2 Unknown situations . . . 39
3.4.3 Error handling . . . 39

3.5 Client Tests . . . 42
3.6 Future work & improvements . . . 43

4 Results **46**
4.1 Introduction . . . 46
4.2 Methodology . . . 46
4.2.1 Virtual Machine . . . 46
4.2.2 Tested implementations . . . 46
4.2.3 Experimental protocol . . . 48
4.3 Servers results . . . 48
4.3.1 Global tests . . . 49
4.3.2 Unknown types tests . . . 56
4.3.3 Errors handling . . . 57
4.4 Client results . . . 64
4.4.1 Global tests . . . 64
4.4.2 Unknown situations . . . 68
4.4.3 Errors handling . . . 69
4.5 Summary . . . 73

5 Discussion **75**
5.1 Key results . . . 75
5.2 `RFC` recommendations . . . 76
5.3 `SIGCOMM` 2021 Posters . . . 78
5.4 Limitations & improvements . . . 78
5.4.1 Real time properties . . . 78
5.4.2 Liveness properties . . . 78
5.4.3 Internal state properties . . . 79
5.4.4 Data types . . . 79
5.5 Further work . . . 79

A Results tables **i**
A.1 `aioquic` . . . ii
A.1.1 Server . . . ii
A.1.2 Client . . . iv
A.2 `quinn` . . . v
A.2.1 Server . . . v
A.2.2 Client . . . vii
A.3 `mvfst` . . . viii
A.3.1 Server . . . viii
A.4 `picoquic` . . . x
A.4.1 Server . . . x
A.4.2 Client . . . xii
A.5 `quic-go` . . . xiii
A.5.1 Server . . . xiii
A.5.2 Client . . . xv
A.6 `quant` . . . xvi

A.6.1 Server xvi
A.6.2 Client xviii
A.7 `quiche` xix
A.7.1 Server xix
A.7.2 Client xxi
A.8 `lsquic` xxii
A.8.1 Server xxii
A.8.2 Client xxiii

B Ivy concepts used for `QUIC` xxiv

C Theory complements xxxi
C.1 Decidability xxxi
C.1.1 Logic concepts xxxi
C.1.2 Weakest preconditions xxxii

D Paper reproduction xxxvi
D.1 Results xxxvi
D.1.1 Methodology xxxvi
D.1.2 Our results xxxvii
D.1.3 Comparison between our results and McMillan results xliii
D.2 Conclusion xliv

E Evolution of the results xlv
E.1 Server evolution xlv
E.2 Client evolution xlvii

F Google Docs xlviii

G Ivy project structure xlix

H Project installation guidelines li
H.1 TLS Keys for Wireshark li
H.2 Docker li

I Successful tests examples lii
I.1 Server lii
I.2 Client lxii

J Example's references lxix

List of Figures

1.1 Type of Connection 3
1.2 Reordering 8

2.1 QUIC stack [7] 11
2.2 Long Header 12
2.3 Initial Packet 13
2.4 Short Header 13
2.5 Protected Header 14
2.6 STREAM frame 15
2.7 Sending Stream States 16
2.8 Receiving Stream States 16
2.9 Channel 21
2.10 Mirror 22
2.11 The Shim [2] 22
2.12 Architecture 25

3.1 QUIC Transport - 268 MUST in total 31
3.2 QUIC Min Ack Delay - 17 MUST in total 36

4.1 Interaction summary between `QUIC` and `TLS` [8] 70

A.1 `lsquic` server errors xxii

List of Tables

2.1 Encryption levels - Packet number space [5] 14
2.2 Variable-Length Integer Encoding [5] 15
2.3 Streams Types [5] 15
2.4 Migration 18

3.1 Original tests proposed by McMillan 38
3.2 Global tests verifying normal execution of a feature 39
3.3 Tests verifying handling of unknown 39
3.4 Transport parameter errors tests 40
3.5 Violation of fields in frame tests 41
3.6 Violation of draft tests 42
3.7 Violation of draft tests 43
3.8 Transport parameter errors 43
3.9 Limitation 44

4.1 Draft 29 tested implementations 47
4.2 Server - Passed tests ratio 49
4.3 Server - General tests of the draft - Passed tests ratio 49
4.4 Server - Management of unknown frames and transport parameter . 56
4.5 Server - Transport parameter errors tests 57
4.6 Quant transport parameter: before/after 59
4.7 Server - Protocol violation errors tests 59
4.8 Server - Invalid fields frames errors tests 62
4.9 Client - Global test 64
4.10 Client - Global test 64
4.11 Client - Unknown situation 68
4.12 Client - Transport parameter errors tests 69
4.13 Client - Invalid fields frames errors tests 72
4.14 Client - Protocol violation errors tests 72
4.15 Quant transport parameter: before/after 74

A.1 `aioquic` server errors ii
A.2 `aioquic` client errors iv

A.3 quinn v0.7.0 server test errors v
A.4 quinn v0.7.0 server test errors without migration vi
A.5 quinn client errors vii
A.6 mvfst server errors viii
A.7 picoquic server errors x
A.8 picoquic client errors xii
A.9 quic-go server errors xiii
A.10 quic-go client errors xv
A.11 quant server errors xvi
A.12 quant client errors xviii
A.13 quiche server errors xix
A.14 quiche client errors xxi
A.15 lsquic client errors xxiii

D.1 VM Ubuntu - draft 18 xxxvii
D.2 Asus Linux Mint - draft 18 xxxvii
D.3 Dell Linux Mint - draft 18 xxxviii
D.4 VM Ubuntu - PicoQUIC xxxviii
D.5 VM Ubuntu - PicoQUIC xxxix
D.6 VM Ubuntu - Quant xxxix

E.1 picoquic draft 18 - Number of errors xlv
E.2 picoquic draft 29 - Number of errors xlvi
E.3 picoquic draft 18 - Number of errors xlvii
E.4 picoquic draft 29 - Number of errors xlvii

I.1 picoquic - quic_server_test_stream - no migration lii
I.2 picoquic - quic_server_test_stream - migration liii
I.3 quinn - quic_server_test_max liii
I.4 quinn - quic_server_test_reset_stream liv
I.5 quinn - quic_server_test_connection_close liv
I.6 quinn - quic_server_test_stop_sending liv
I.7 picoquic - quic_server_test_accept_maxdata lv
I.8 picoquic - quic_server_test_ext_min_ack_delay lv
I.9 picoquic - quic_server_test_unknown lvi
I.10 picoquic - quic_server_test_unknown_tp lvi
I.11 quinn - quic_server_test_tp_error lvii
I.12 quinn - quic_server_test_double_tp_error lvii
I.13 quinn - quic_server_test_tp_acticoid_error lvii
I.14 quinn - quic_server_test_tp_no_icid lvii
I.15 quinn - quic_server_test_blocked_streams_maxstream_error . . . lviii
I.16 picoquic - quic_server_test_retirecoid_error lviii
I.17 quinn - quic_server_test_stream_limit_error lviii
I.18 quinn - quic_server_test_newcoid_rtp_error lix
I.19 quinn - quic_server_test_newcoid_length_error lix
I.20 picoquic - quic_server_test_max_error lix

I.21 `picoquic` - quic_server_test_handshake_done_error lx
I.22 `quinn` - quic_server_test_new_token_error lx
I.23 `picoquic` - quic_server_test_token_error lx
I.24 `picoquic` - quic_server_test_max_limit_error lx
I.25 `picoquic` - quic_server_test_newconnectionid_error lxi
I.26 `picoquic` - quic_client_test_stream lxii
I.27 `picoquic` - quic_client_test_max lxiii
I.28 `picoquic` - quic_client_test_accept_maxdata lxiii
I.29 `picoquic` - quic_client_test_ext_min_ack_delay lxiv
I.30 `picoquic` - quic_client_test_unknown lxiv
I.31 `picoquic` - quic_client_test_unknown_tp lxv
I.32 `quant` - quic_client_test_tp_activoid_error lxvi
I.33 `picoquic` - quic_client_test_tp_error lxvi
I.34 `picoquic` - quic_client_test_no_ocid lxvi
I.35 `picoquic` - quic_client_test_double_tp_error lxvi
I.36 `picoquic` - quic_client_test_tp_prefadd_error lxvii
I.37 `picoquic` - quic_client_test_new_token_error lxvii
I.38 `picoquic` - quic_client_test_max_limit_error lxvii
I.39 `picoquic` - quic_client_test_blocked_blocked_limit_error lxviii
I.40 `picoquic` - quic_client_test_retirecoid_error lxviii

Chapter 1

Introduction

QUIC is a new network protocol that is indented to make the Internet faster, secure and more flexible. It is designed to replace the whole TCP/TLS/HTTP stack and is built above UDP. It is already used by more than 5.0% of all websites [9].

Implementing a new protocol should be done carefully. History has shown that many of the most used networks suffered from severe security breaches. As an example, the TCP protocol presented huge security issues soon after its release [10].

The Internet Engineering Task Force (IETF), in collaboration with the Internet community, created a RFC. This "Request For Commands" specifies the properties that an implementation of QUIC *must* or *should* respect.

The most common approach used to verify whether QUIC implementations follow properly the RFC is *interoperability testing*. It is an approach where an implementation is tested against others. This approach has been used by works such as the QUIC-Tracker test suite [11] and the QUIC-Interop-Runner [12].

Another way to verify that an implementation of QUIC respects this RFC is to perform a formal verification.

Our work consists in performing a formal verification of QUIC by extending the work already done by Kenneth McMillan and Lenore D. Zuck, "*Formal Specification and Testing of QUIC*" [4]. We updated it to one of the latest drafts and we modelled one QUIC extension. We also refactored the project to make it more maintainable in the future and we provided a better deployment for the project using Docker. Thanks to these improvements, it will be easier for new contributors to perform a formal verification of QUIC or other network protocols.

This document is divided into five parts. First, we explain more comprehensively the purpose of our work. We present the different notations that are used throughout this thesis and provides a brief explanation of what is the purpose of formal verification. Then, we present a quick overview of QUIC and the tools we used. We

end this section by explaining the key steps of our work and the limitation of the formalisation of RFCs.

Then, it covers the state of the art. There are two parts. The first part discusses QUIC. It gives a more precise explanation of this protocol. It starts by describing the purpose of QUIC and we present the historical challenges it solved. It covers the different parts of the protocol that we effectively tested using Ivy. In particular, it highlights the features of the protocol where we obtained interesting results.

The second part is about formal verification. It mainly focuses on Ivy, the language and tool we used to develop the formal verification of QUIC. It also makes references to other approaches used for the formal verification of QUIC. Our description of Ivy contains two parts. First, there is a theoretical one that is inspired by the papers authored by McMillan and Zuck. It covers how the formal verification of QUIC is done. The second part is practical: we explain how to apply Ivy to model the specification of QUIC.

Then, we analyse the McMillan work on which we based our work. It is followed by an explanation of his methodology and we describe what was the state of the model before our contributions. We give a detailed list of what he did and what remains to be done.

Afterwards, we present the enhancement we made to this work. First, we describe the main new properties from the QUIC specification that we added to the Ivy model. We also show how we updated the QUIC model and how we modified and refactor its architecture to make it more modular. We also present our methodology and describe the process we used to perform our final tests. In closing, we describe how we modelled a QUIC extension quic-delayed-ack [13].

The last part is about the tests we performed based on our model on several QUIC implementations and the results we obtained. We tested two of the three implementations that McMillan tested and added six other implementations (4.1). We do both a quantitative and qualitative analysis. We explain the different errors we detected in the implementations and how we interpret them.

The results we obtained highlight several types of errors. Some of them are non-implemented features. Others are violations of the specification or result from ambiguities in the draft. We can observe an ambiguity when encountering different behaviours adopted by implementations for a specific requirement.

Finally, we conclude with a final presentation of our results, a list of our accomplishments as well as a list of actions that remains to be done in subsequent work.

1.1 Testing QUIC

QUIC is a new transport layer protocol. The initial purpose of QUIC was to make it work with HTTP, that is acting as the application layer. QUIC has evolved and efforts are being deployed to make other applications compatible with QUIC. An example is DNS [14]. It is also possible to use MQTT, a protocol popular in IoT [15]. Here is a non-exhaustive list of some QUIC benefits:

1. QUIC uses UDP and is also a reliable protocol. It was *designed* to provide better performance [16] than TCP. However, efforts are still being deployed to make QUIC faster than TCP. Experiments showed that currently, QUIC has a worst page load time (PLT) than TCP [17]. There are still new mechanisms that improve QUIC speed. An example is a 0-RTT connection that allows an QUIC endpoint not to have to do a handshake if it already contacted a server in the past. With TCP and TLS, one handshake is needed for both protocols. In QUIC, there is a maximum of one handshake and 0 in the case of a 0-RTT connection, as shown in Figure 1.1 [18].

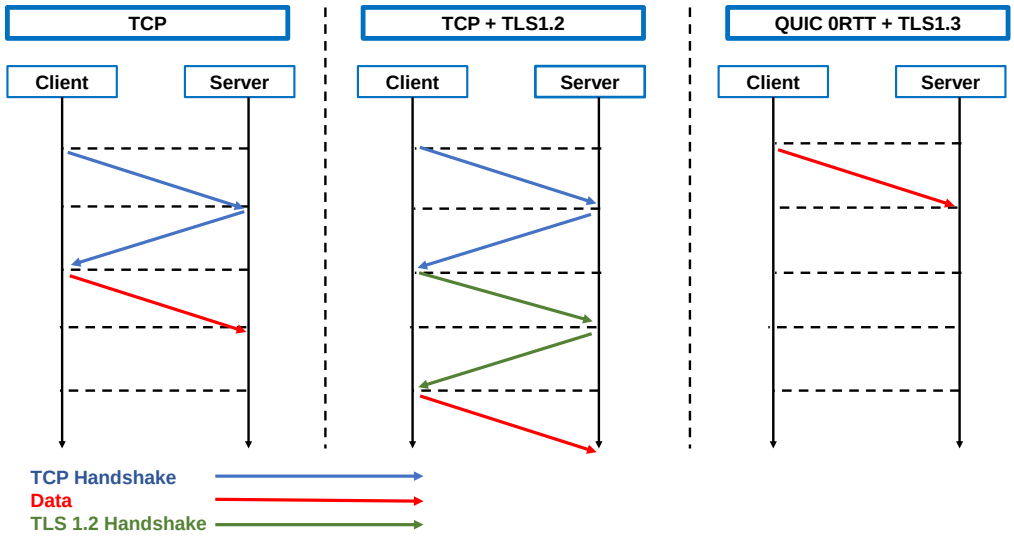


Figure 1.1: Type of Connection

2. QUIC does not only identify the connection by a tuple IP/Port like in TCP. Instead, it gives each connection a unique connection ID. QUIC connection is not linked to a specific port or IP address. This process of changing this tuple IP/Port while keeping the same connection ID is called migration.

3. Moreover, QUIC can do multiplexing with the concept of streams. There can be multiple streams in the same connection, each of them for example carrying a different kind of data. This can be used to avoid head-of-line blocking. It is a problem in TCP where one connection is used for multiple

different resources. If one resource is not available, it will block the whole flow. The other resources will be blocked. With QUIC, this issue does not exist. For example, on the loading of a website, we create a stream for the text and the images. If the images are not available or very slow to get, we still have the text in the other stream [19].

4. It is easier to extend QUIC than TCP. QUIC packets are encrypted and middle boxes cannot read the packet and modify it, while allowing mitigating their effects [20]. Furthermore, QUIC allows mechanisms to extend the protocol by adding new frames and transport parameters.

1.2 Formal verification

The most common way to check if an implementation of a protocol has the expected properties is to test it with other implementations of this protocol. This is called "*interoperability tests*". By testing an implementation with multiple other implementations, we can check if it reacts as expected. Such examples of work can be seen in [21]. This is the most common approach and it allows to detect many errors. However, history has also shown that this approach suffers from some drawbacks. It is important to note that the RFC, despite the hard work done by the Internet community, still contains some ambiguities. In interoperability tests, we do not explicitly explain how to interpret the QUIC specification. The information collected from those tests is unclearly saved, as the information is contained in the implementations themselves. In a formal verification method, we have to build a model and explicitly interpret those ambiguities. As a result, the model can be a complement to the specification to see how an implementation should and must behave. The test coverage is also lower for interoperability tests because most implementations do not implement all the properties that the standard allows. The last reason is that interoperability cannot be done with implementations that will be present in the future. It means that an implementation can pass all the interoperability tests but would fail to operate with a future implementation that would strictly respect the standards [4].

The tool we used to perform a formal verification of QUIC is Ivy. It is a language created by Microsoft and which is based on the Z3 solver, a famous SMT (satisfiability modulo theories) solver [22]. A network protocol is a high-level abstraction that includes many components: the server, the receivers, the network, etc. To test formally QUIC, it is easier to split the whole system in several components and see if each of them behaves correctly. Ivy makes it easy and convenient to design our formal proof that way, using its primitives. It also makes it easy to reduce all proof obligations to statements in decidable logic. We give a closer look at how ivy works in sections 2.2.2 and in appendix B [3].

The way we divided QUIC as Ivy components is called compositional testing. Our model does not model the RFC as a finite state machine (FSM) but keeps a global

state that allows us to determine if a specific action from a component is breaking the system or not. Ivy acts as a checker to see if all the actions are legal. It also acts as a generator. It sends legal packets by solving all the constraints fixed by the specification. It generates them with the help of Z3 that handles the different constraints whose output needs to be valid. [4]

Verification of QUIC using Ivy was performed in 7 main parts. There is one for each of the QUIC protocol layer: application, security, frame, packet and protection. There are also the shim which is describe in section 2.2.2, a serializer and a deserializer.

The serializer is used to transform the Ivy generated packet states into real packets that can be sent on the wire. The deserializer is similar: it transforms the real packets received into Ivy abstraction.

1.3 Key steps

The starting point for our work is the article by McMillan and Zuck [4] and the Ivy documentation [23]. We reproduced the results of McMillan using the methodology he presented. More details can be find in appendix D.

We first listed all the properties from the RFC and specified whether they were tested by McMillan and Zuck or not. A list of all those properties is provided in the appendix F. Then, we have changed the structure of Ivy to implement the other properties. An overview of the different components of our system is shown in Section 3. For example, we refactored the project to have clear distinctions between the different layers of the protocol or to be able to accept all transport parameters.

We identified four paths where we could extend the original model. First, McMillan and Zuck stopped working on it after the draft 18 [24]. One way to progress was to update the model to the newest draft. At the time of writing those lines, the current version is final and is in the RFC9000 [1]. It was published by the end of May 2021. We stopped our work at draft 29 [5]. As there are no important difference between the final version and the 29^{th} version, one can easily update our work to the latest version. The second way was to identify some parts of the specification that McMillan and Zuck did not implement and was left to do. We identified those in the two different articles they wrote [4, 2] on the subject as well as in some indication that were directly in the code [25]. The next source of improvement was to rework the project to make it more modular. As an example, we changed the output so that it is easier to identify in which state the system is when one property fails.

Finally, we started to verify one QUIC extension. McMillan verified no extensions. We first had to integrate them into our model. Then, we chose some properties to verify. Each QUIC extension comes with its specification and, at the difference of the core QUIC RFC, it is more likely to change. It was therefore important to identify properties that, in our opinion, would stay in the future.

Once we extended our model as described, we needed to test it on concrete implementations. There are two main reasons why this is important. First, we had to be able to detect misbehavior. It can allow the developers of these implementations to solve those issues. Second, we needed to check if the assumptions of our model were correct.

1.4 Limitations

It is important to note that a large part of the RFC properties is not testable. There are multiple reasons for this. Those reasons are not exclusive. One rule can be ambiguous and treat the internal state. In our explanation, we divided them into 3 categories and give a concrete example of why some properties are impossible to test formally.

Ambiguous

The first reason is that the specification is sometimes not clear enough. This property is an example of that:

> *"An address validation token MUST be difficult to guess. Including a large enough random value in the token would be sufficient, but this depends on the server remembering the value it sends to clients."*
>
> QUIC draft 29 section 8.1.4.[5]

There is not a formal definition of what "*difficult to guess*" means. From a formal point of view, one could argue that even if one manages to guess the value the right way, one does not break the rule if the process of finding it is complex. It is impossible to be formal on where to put the barrier. We can use some heuristics but we will never be able to know for sure whether an implementation respects this rule or not. Replacing *difficult to guess* by *unpredictable* would remove the ambiguity. However, the property would become undecidable.

Internal state

The second reason is that sometimes, we do not have enough information to see if a property is respected or not. We do not have access to the internal state of the

implementation that we test. In the RFC, many rules are focused on the point of view of the tested implementation. Here is an example of such rules:

> *"A packet MUST NOT be acknowledged until packet protection has been successfully removed and all frames contained in the packet have been processed. For STREAM frames, this means the data has been enqueued in preparation to be received by the application protocol, but it does not require that data is delivered and consumed"*
>
> QUIC draft 29 section 13.1.[5]

In our model, we consider the tested implementation as a black box; therefore, it is impossible to know if it has effectively removed the packet protection before processing the packet. It is one limitation of our model. However, one advantage of this approach is that we can use many different tested implementations without changing our model.

Undecidable properties

One example of undecidable property for an external observer is this property:

> *"The endpoint MUST use unpredictable data in every* PATH_CHALLENGE *frame so that it can associate the peer's response with the corresponding* PATH_CHALLENGE*."*
>
> QUIC draft 29 section 8.3. [5]

We can interpret "*unpredictable*" by the fact that there is no way to find a mathematical function to discover the value of all the future PATH_CHALLENGE frames. There exists no algorithm that can clearly answer if yes or no a specific function can be used to predict all the future values of the data in PATH_CHALLENGE frames. This problem is undecidable. However, we can use some heuristics to give an hint of a possible violation of this property. For example, if we find a mathematical relationship between the data in the PATH_CHALLENGE frame and previous data and that we manage to find the value of future PATH_CHALLENGE frames, we can say that the data is predictable. There is a problem with this approach: what can we conclude if in two different PATH_CHALLENGE frame there are two values that have a clear relation ? We can detect it as a violation of this property and take the risk of having a false positive result because the next PATH_CHALLENGE frame may not follow this pattern. If we don't detect it as an error, we don't verify at all this property.

Possible solutions

For each of these rules that are impossible to test formally in our model, we tried to find a solution. The first solution was to find a weaker rule that we could formally test. Let us take the rule we described previously:

> *"An address validation token MUST be difficult to guess."*
>
> QUIC draft 29 section 8.1.4.[5]

Even if we cannot say that this property is valid, we can say, in some cases, that it is invalid. If the implementation gives several times the same validation token, we can use this heuristic to state with a strong confidence interval that the rule is not respected.

Another solution is to put some hypothesis on our model that would make let us test some of these properties. Let us have a look at this property:

> *"Additional connection IDs are communicated to the peer using* `NEW_CONNECTION_ID` *frames. The sequence number on each newly-issued connection ID MUST increase by 1."*
>
> QUIC draft 29 section 5.1.1. [5]

It is impossible to verify formally this rule if the wire is not perfect. If the wire is not wire is not perfect, we could only verify this property by inspecting the internal state of the implementations. It is not possible in our methodology. We cannot verify it by only observing it on a not perfect wire because there may be reordering. It means that if we see that the packets are well ordered with increasing sequence numbers on the wire, we cannot conclude that the tested implementation respects this rule. Let us consider this scenario:

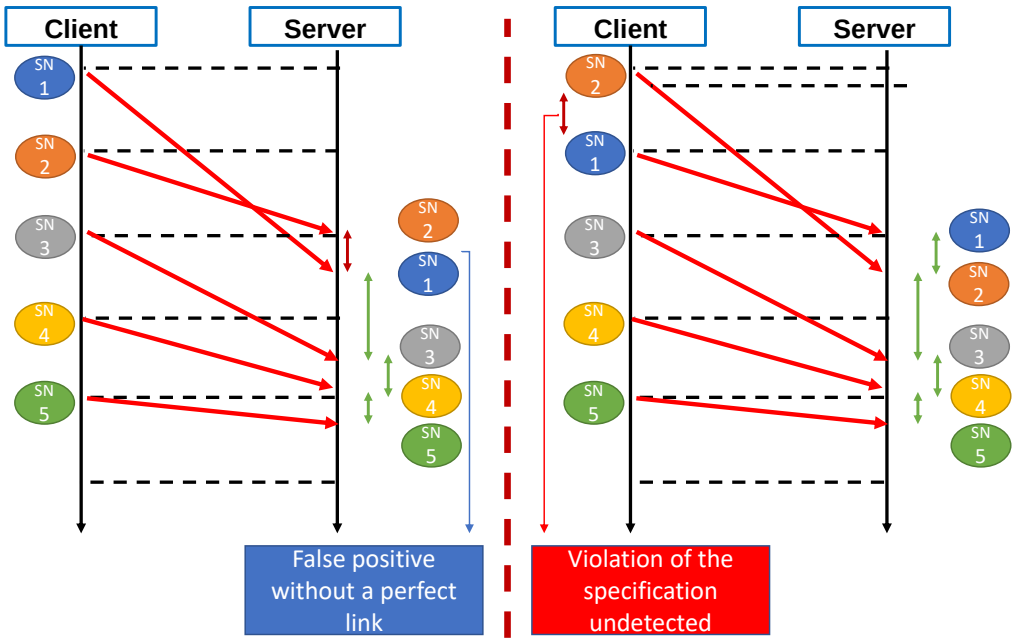


Figure 1.2: Reordering

In Figure 1.2 we observe a client that is sending packet to a server. The time goes from up to bottom. We see that the sender sends three packets. Each packet contains frames with a sequence number (`Sn`). On the figure, the sender is not respecting the rule: it sends packet number 2 before packet number 1. However, our model does not have access to this information. What we see is that frame number 2 comes before the number 1. This happens because the network can reorder the different packets containing the frame. Because of that, we cannot test formally this property. However, we do some heuristics using a hypothesis. In this case, we imagine that the network is perfect and that it does not reorder the packets. In the real world, this hypothesis does not hold, but by having this heuristic, if the testing protocol uses a network that is as reliable as possible, we can have some guess about whether the tested implementation respects this property. We cannot be formal on that, but it can give an idea.

Finally, the last solution would be to correct the draft itself. If the draft is ambiguous, we can propose a solution to reformulate it. This way, new implementations will know exactly what to do.

Chapter 2

State of the art

This chapter describes the state of the art for QUIC and the formal verification of QUIC.

First, we describe the QUIC protocol and the reasons that led the Internet community to choose this new protocol. We cover the different protocols that were the precursors of QUIC to understand better why there was a need for some QUIC features. We also explain what are the QUIC extensions and how they work. Then, we describe all the different features of QUIC that we test afterwards and that are needed to be stated to understand our final results.

Next, we explain how Ivy works. Ivy is a tool and language that was used by McMillan and Zuck to perform a formal verification of QUIC and of its extensions. We present the theoretical aspect of Ivy and explains the methodology that McMillan and Zuck used to perform the formal verification of QUIC using Ivy. It is called "Network-centric Compositional Testing" (NCT). In our work, we kept using this methodology. We also show how to use Ivy in practice. Appendix B explains the key concepts of Ivy. Examples of using Ivy are coming from the formal verification of QUIC.

2.1 QUIC

QUIC is a transport layer protocol that was first implemented by Google and launched in early 2013 [16]. It was created to solve issues that the Internet world has faced in the past 20 years [26]. The delivery of the Internet website has been dominated by the HTTP protocol. Over the years, the Internet has evolved: traffic has dramatically increased and the need for a faster page loading time (PLT) for the web has been crucial. Google laboratory decided to create two new protocols to take this challenge: SPDY [27] and later QUIC [16].

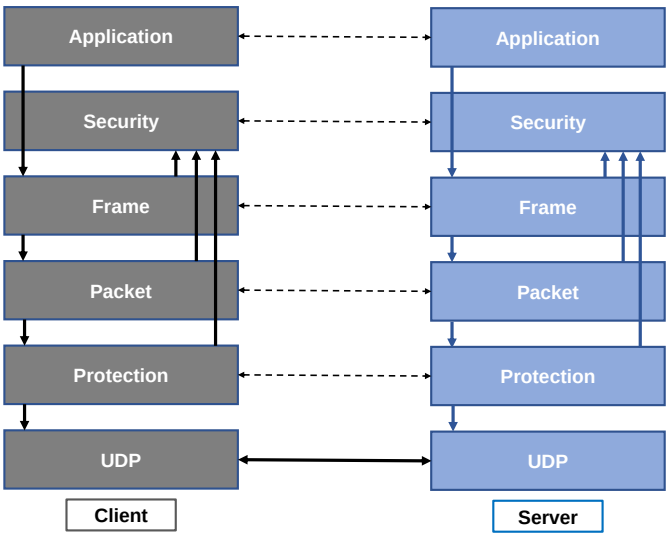


Figure 2.1: QUIC stack [7]

2.1.1 QUIC history

SPDY was designed in 2009 and came with many innovations to be able to improve HTTP. When QUIC was released, some of its features were reused. SPDY became the starting point of the second version of the HTTP protocol. Here is a list of a few key new features [28]:

1. Multiplexing: SPDY allows in a single connection to have multiple streams. In HTTP/1.0 with regular TCP connections, there was only one stream by connection. The only way to make some kind of concurrency is to create a new connection which leads to many problems for the latency: time needed to create a new connection, congestion control between connections, ...

2. Header compression; nowadays, the header of HTTP requests can have a size of up to 2 KB because more and more cookies are used by websites and more features are implemented in those headers generating a huge increase of this size. On low bandwidth connections, this can be a bottleneck and SPDY solves this problem by compressing the data in those headers.

3. Prioritization and protocol negotiation: since there can be multiplexing with SPDY, a user could need to put some priority on the receptions of some resources. SPDY handles highly priority requests and helps to control the congestion of a connection.

As we will see, most of those key features were reimplemented in the QUIC protocol. SPDY was discontinued in favour of HTTP over QUIC.

QUIC is a protocol that aims at replacing the HTTPS stack formed with TLS and TCP. QUIC is built over UDP. It came with various advantages that are designed

to solve the issues the Internet had with TCP [29]. The first one is the way it is implemented. TCP was part of the kernel and it was one of the main issues of the improvement of this protocol. QUIC was developed as part of the user space. The care with which the kernel is usually updated prevented the fast deployment of a major upgrade that could lead to a significant improvement of TCP. QUIC has been evolving quickly, as shown by the short lifetime of QUIC versions [30].

2.1.2 QUIC features

Some QUIC features are described to give a clear idea of how QUIC behaves. We focus on the key features as well as the one we effectively tested using Ivy.

QUIC packet

Let's first explain how a QUIC packet looks like. As QUIC is built above UDP, it is encapsulated in a datagram. There are two types of headers for QUIC packets: short ones and long ones.

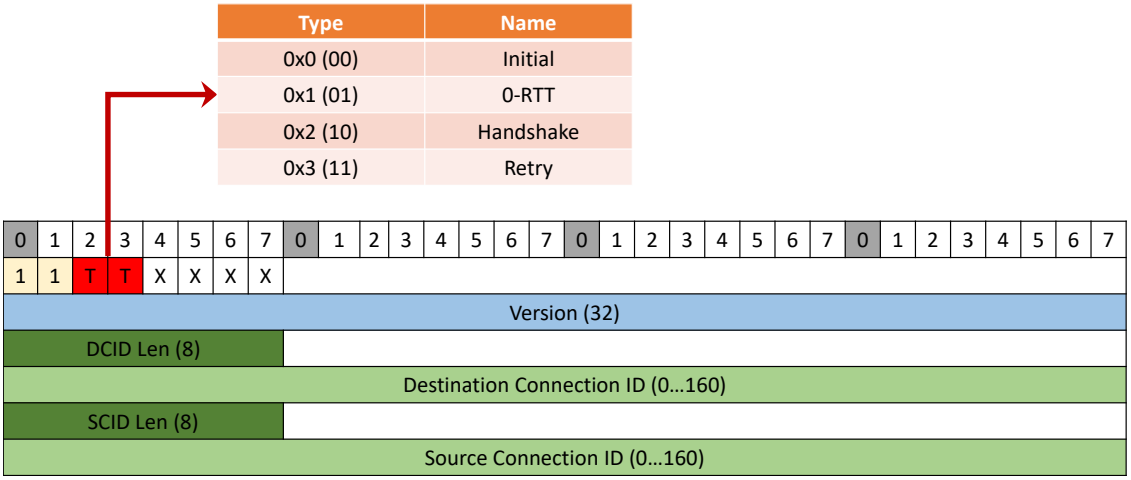


Figure 2.2: Long Header

Let us begin with the long header. On Figure 2.2, we see a graphical representation of it. The first line represents the index of the position of a bit in a specific byte. The header starts on the second line with a bit indicating if the packet is a long header (1) or not (0). The second bit is a fixed bit that must always be equal to one. Then the bits 2 and 3 define the packet type that follows the long header. The type is used for different situations and different encryption level. Each type of packet define different fields that are used. On Figure 2.2, we can observe the four types of packets. The Initial packets are used during connection establishment, the Handshake packets are used for the cryptographic handshake negotiation. The 0-RTT packets are used when the 0-RTT connection is employed and finally the Retry packets are present for address validation.

Here is the Initial packet:

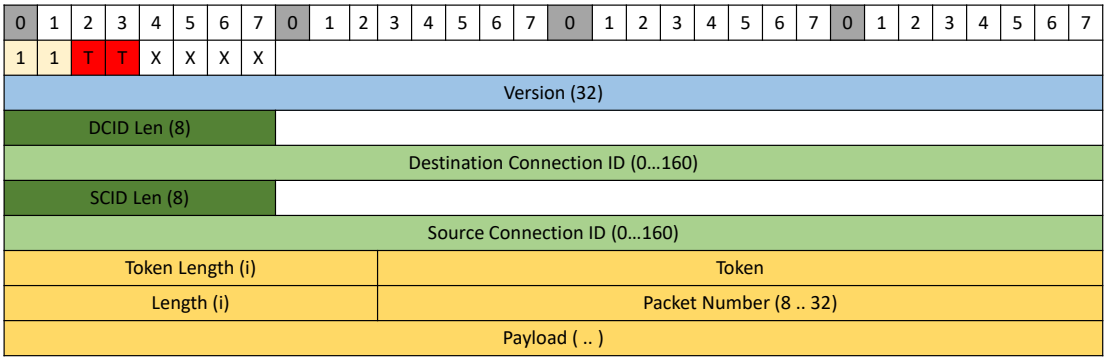


Figure 2.3: Initial Packet

The version field is used to determine which version of the QUIC protocol is used. The client sends a specific version number. Either the server accepts it and uses this version number or it sends to the client a version negotiation packet with the list of the version numbers that it supports. Then, the connection ID field identifies a connection. This connection number does not depend on the previous layers and allows a QUIC connection to migrate. For example, we can imagine that one endpoint could change its IP address while still preserving the QUIC connection. This would be useful for a user who is travelling and thus changing many times his IP address while still wanting to preserve the connection. The next field is the packet number. The use of this number is different from TCP. In QUIC, this number always increases and two packets can never have the same packet number even in the case of retransmissions.

The short header is used for packets after the handshake. It contains less information. It only includes some flags, the destination connection ID (DCID) and the packet number.

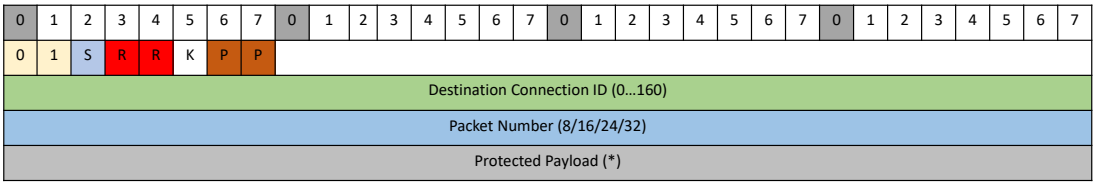


Figure 2.4: Short Header

The payload of a QUIC packet is constituted of frames.

QUIC packet protection

QUIC uses TLS to protect its packets. A complete explanation of this part can be found in [8]. We provide a summary of this section.

With a protected packet in QUIC, most of the information is encrypted using the key of the corresponding encryption level. The Figure 2.5 shows in red the elements

that are encrypted.

To encrypt a packet, an endpoint needs two elements. First it needs the encryption keys. It also needs to extract the first 128 bits of the payload. Once the sample is extracted, it is used with the keys to send the encrypted packet. The receiver of the packet need to have the corresponding decryption key. He also needs to extract the first 128 bits of the payload he receives.

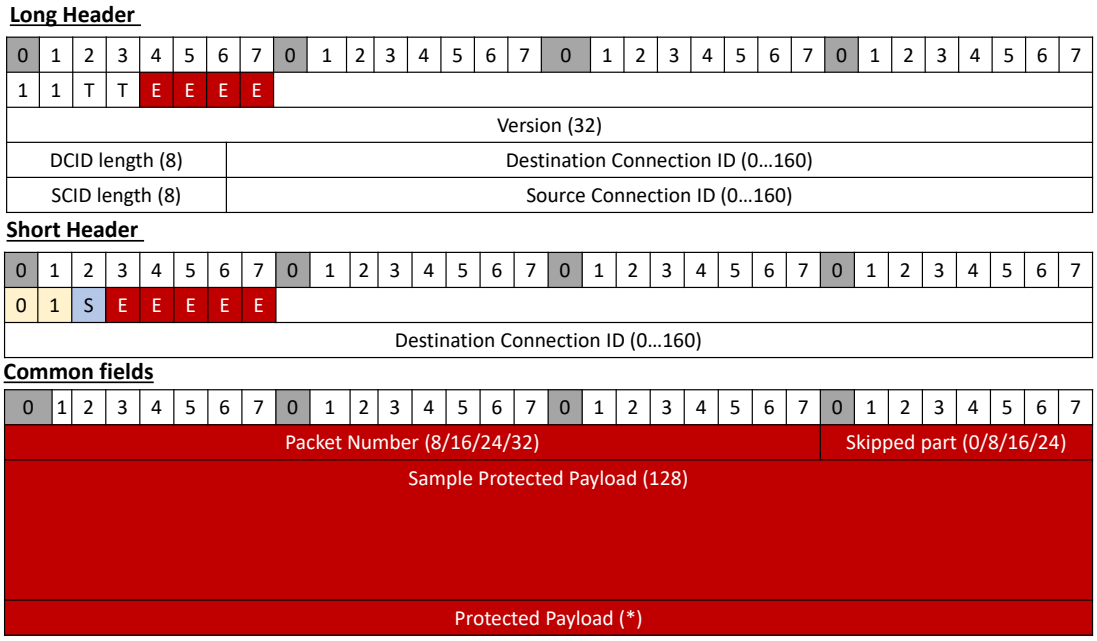


Figure 2.5: Protected Header

QUIC encryption level & packet number spaces

The encryption levels are used at specific moments of the connection and different keys are used. Different packet number spaces are also allowed, which means that we can use different sequence numbers in each space. Their detailed utilization is defined in [8]. Here is a summary containing the minimum required information. More details can be found in the draft:

Packet Type	Encryption Keys	PN Space
Initial	Initial secret	Initial
0-RTT Protected	0-RTT	Application data
Handshake	Handshake	Handshake
Retry	Retry	
Version negotiation		
Short Header	1-RTT	Application data

Table 2.1: Encryption levels - Packet number space [5]

QUIC frames

Frames can be of different types. Some examples are CONNECTION_CLOSE frames, to end a connection, ACK frame, to acknowledge the reception of a set of frames, or CRYPTO frames, that allow exchanging cryptographic handshake messages. The STREAM frame is used to send data between endpoints. A complete description of the frame and their usage can be found in [31].

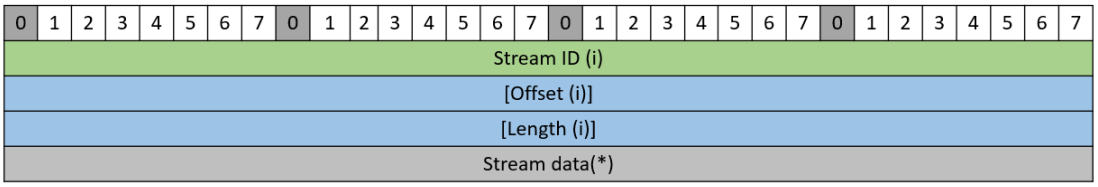


Figure 2.6: STREAM frame

This example shows the usage of a QUIC concept called "Variable-Length Integer Encoding", which is a clever way to encode data with, as expected, a variable length. It is indicated on Figure 2.6 by "(i)". As said in the draft, this method reserves the two most significant bits of the first byte to encode the length size in bytes. Then, the value is encoded in the remaining bits, in network byte order:

2bits	Length	Usable bit	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 2.2: Variable-Length Integer Encoding [5]

QUIC streams

A QUIC connection can carry one or several streams. They can go in both directions: both the server and the client can send data. Thus we have unidirectional and bidirectional streams. Every time a stream frame with a new STREAM ID is sent, a new stream is implicitly created.

The least significant bit of the stream ID identifies the initiator of the stream and the second least significant bit allows the distinction between bidirectional streams (0) and unidirectional streams (1):

Bits	Stream Type
0x0	Client-Initiated, Bidirectional
0x1	Server-Initiated, Bidirectional
0x2	Client-Initiated, Unidirectional
0x3	Server-Initiated, Unidirectional

Table 2.3: Streams Types [5]

Figures 2.7 and 2.8 are taken from the draft 29 section 3.1. and 3.2. They summarize how the streams are working:

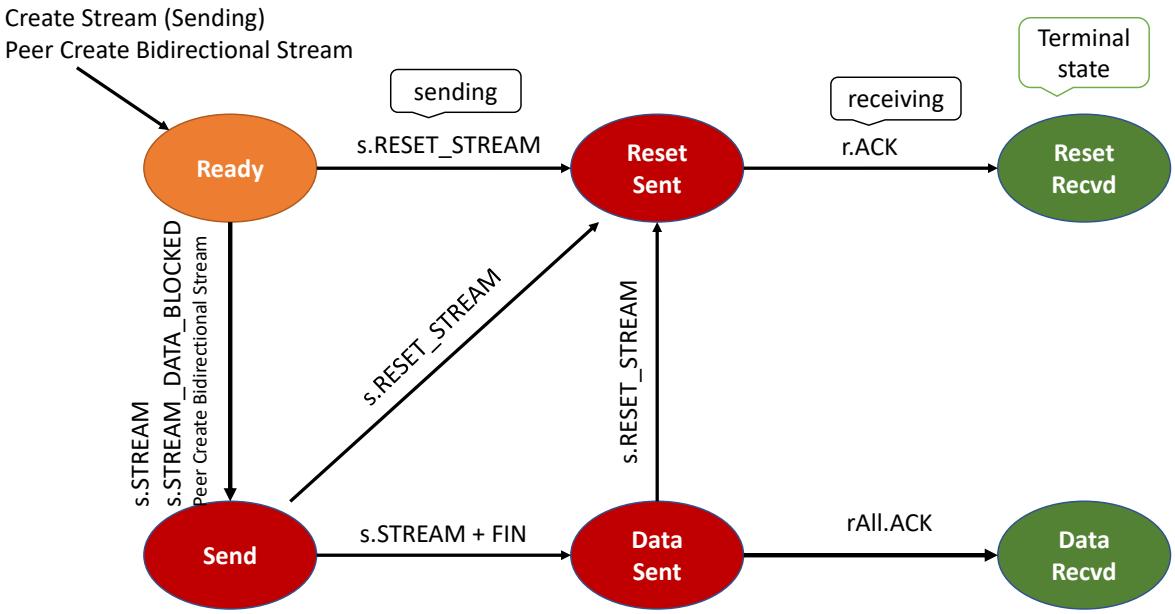


Figure 2.7: Sending Stream States

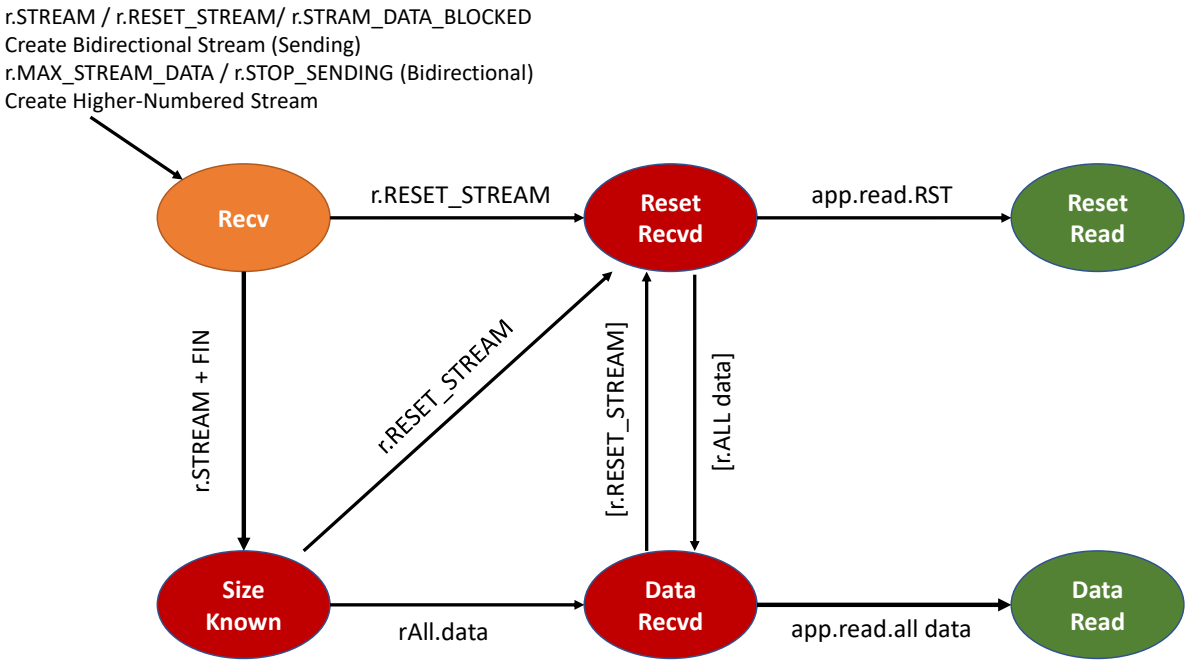


Figure 2.8: Receiving Stream States

QUIC connection

Let us explain how QUIC works. There are several ways to start a connection. We can have one 0-RTT connection if the client already knows the server and a 1-RTT connection if they do not know each other and they have to negotiate the QUIC version they will use.

Let us imagine this is the first time that the server and the client connect and that the server directly accepts the QUIC version that the client uses. We call this connection 1-RTT and it is illustrated by Figure 1.1. The client starts by sending a "Client Hello" message. This packet contains a list of transport parameters that informs the server of the modality of the connection. At any moment, both the server and the client can send a CONNECTION_CLOSE frame to end the connection in case of an error. The server answers with an Initial packet containing the "Server Hello" message. This message mainly contains the choice of the cipher suite algorithms to encrypt future packets. At the same moment, the server sends a Handshake packet that allows authenticating itself. The handshake packets are encrypted. A part of the QUIC header is also encrypted. It was not the case with TCP. An advantage is that it prevents interference with middleware that will be unable to modify this header. This is one drawback of QUIC. A disadvantage of encrypting the header is that it is more difficult to analyse QUIC connections if we detect some problems and need to troubleshoot them. Finally, the client ends the handshake with its handshake packet. The client and the server are now able to exchange data using 1-RTT packets.

The connection can be different if the server and the client have already communicated before. In this situation, it is possible to send data directly as shown in Figure 1.1. The client needs to keep in memory different information from the previous connection with the server such as the Diffie-Hellman value and send a cryptographic cookie that it cached. This allows the client to encrypt directly the payload of the first packet that can contain data. With this information, the server can decrypt the packet and continue the cryptographic session from the old connection [32]. However, it is needed to be careful with the data sent in the 0-RTT connection, it is not forward secure [33].

QUIC migration

We have seen that a QUIC connection is identified by a connection ID and not by a (IP address/port) tuple. One benefit of this approach is the possibility to migrate within the same connection. A migration happens when there is a change in the IP address/port tuple. A use case is when there is a WiFi connection and that there is a switch between several access points and thus IP addresses. With this migration, there is no need to open new connections on each IP address like TCP.

To start a migration, a QUIC client needs to change its (IP address/port) tuple. On the following trace 2.4, the client change its port on the packet 13. It is a new IP address/port tuple. The connection ID is the identifier of the connection.

The server receives this packet and detect a port change and notice it is the same connection ID. The server must send a PATH_CHALLENGE frame containing a random value of 8 bytes. The packet 7 is sent towards the client to verify the authenticity of the client. The client must at the reception of the PATH_CHALLENGE send a PATH_REPONSE frame. It is sent in packet 15. This frame must contain the same value that is inside the PATH_CHALLENGE frame. When the client receives a PATH_REPONSE, it checks if the value matches with the one that was sent in the PATH_CHALLENGE. The path is validated if it is the case. It is invalidated if not.

The purpose of the PATH_RESPONSE and PATH_CHALLENGE frames are to mitigate some attacks. An example is the amplification attack. An attacker could start a connection with a server that sends much data. Then, the initiator changes its address with a spoofed address. The spoofed address would directly receive many packets without asking anything. With the path validation, this attack is impossible.

src	dst	Information
4988	4443	1232, Initial, DCID=0000000000000001, SCID=0000000000000000, PKT: 6, CRYPTO
4443	4988	177, Initial, DCID=0000000000000000, SCID=af607353e17bdead, PKT: 0,
4443	4988	1069, Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKT: 2, CRYPTO
4443	4988	171, Initial, DCID=0000000000000000, SCID=af607353e17bdead, PKT: 3, CRYPTO
4443	4988	67, Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKT: 5, CRYPTO CRYPTO
4988	4443	131, Handshake, DCID=af607353e17bdead, SCID=0000000000000000, PKT: 12, CRYPTO
4443	4988	56, Protected Payload (KP0), DCID=0000000000000000, PKT: 6, DONE
4987	4443	65, Protected Payload (KP0), DCID=f5d90b5172d826f1, PKT: 13, STREAM(4)
4443	**4987**	36, Protected Payload (KP0), DCID=0000000000000000, PKT: 7, PC
4443	**4987**	33, Protected Payload (KP0), DCID=0000000000000000, PKT: 8,
4443	**4987**	40, Protected Payload (KP0), DCID=0000000000000000, PKT: 9, STREAM(4)
4443	**4987**	29, Protected Payload (KP0), DCID=0000000000000000, PKT: 10,
4987	4443	79, Protected Payload (KP0), DCID=af607353e17bdead, PKT: 15, PR
4443	**4987**	62, Protected Payload (KP0), DCID=0000000000000000, PKT: 11,
4443	**4987**	40, Protected Payload (KP0), DCID=0000000000000000, PKT: 12, STREAM(12)
4443	**4987**	29, Protected Payload (KP0), DCID=0000000000000000, PKT: 13,

Table 2.4: Migration

2.2 Formal verification & Ivy

This section is split in two parts: we explain and summaries the methodology of McMillan and Zuck [4, 2]. The examples that are shown in this section are come from those articles. We reused this methodology in our work. Afterwards, we explain how to model QUIC using Ivy.

2.2.1 Network-centric Compositional Testing

The most common way to test if an implementation of a protocol has the properties that we expect is to operate it with other implementations. We check if it has the expected behaviour. However, experience has shown that this approach suffers from several drawbacks. First, if an inconsistency of the specification is noticed between the two implementations, the errors will be corrected in the implementations themselves. There is no clear view of what an implementation should do to fit the specification. The test coverage is also low because most implementations do not implement all the properties allowed by the standard. The last reason is also that you cannot do interoperability with an implementation that will be present in the future. It could completely respect the standards but be impossible to operate with it.

We use another methodology. Our starting point is the QUIC specification [31]. We extract all the properties that an implementation should have. We consider all the properties as constraints to build a packet. Specific scenarios are created. As an example, the goal of a scenario could be to send data or to close a connection. The interaction between two implementations can be represented as a graph with nodes. To test an implementation, we create a node that is fully respecting the specification. We generate the set of all the packets that could be sent by this node at time T. We remove from this set all the packets that violate the specification. Then, we send one random packet from this set.

This node is a concrete representation of the protocol. It can send network packets that are visible on the wire. We form a system composed of an arbitrary number of nodes in which there are at least Ivy, the tested implementation, and the network. All nodes from this system have their local specification. All those specifications must be respected.

This methodology is called compositional testing because it composes all the nodes together. Each node has an input and an output part. When we compose two nodes together, the output of the first node becomes the input of the second one. All the assumptions we have from the formal specification on one node are treated as a guarantee for its output. If we notice at some point that a property is not respected for one node, there are two possibilities. First, the node which broke the global specification is misbehaving. It did an illegal action and the node should be corrected. It is its responsibility if the specification is not satisfied anymore.

The other reason could be that the specification is too weak and that we cannot compose the two nodes together. The only solution to solve this is to change the specification. We have to put some more rules because the input of one node needs more constraints.

The specification may not always be clear. It is sometimes subject to interpretation. With this methodology, we give a clear representation of which inputs and outputs are valid. While the specification is more focused on what an implementation should internally do, with Ivy, we have a clear specification of what an external observer should see on the network.

Compositional testing was inspired by other works , especially the assume/guarantee part [34]. It is not the only approach that has been used to test formally network protocols. Other approaches exist using other methodologies and were explained in section 1.5. However, most approaches commonly used for the formal verification of network protocols are not suited for QUIC specificities. As example, the size of QUIC messages is a challenge. With *Network-centric Compositional Testing*, it is possible to specify an unambiguous interpretation of the QUIC specification while testing different concrete implementations of QUIC.

2.2.2 Concepts of Ivy

(1) Processes

Processes are entities that are communicating value using channels. We want the communication to be synchronous. This way, we can verify the different properties on a sending or receiving event. Network make the exchange between two entities asynchronous: what is sent is not directly received. To solve this problem, Ivy considers that the network is also a process. This way, we can model the communication over a channel as synchronous: what we send is directly received.

Figure 2.9 shows the flow when Ivy generates a packet that is sent to a test implementation. First, we use the Ivy mirror. This entity allows generating random packets that are respecting the specification. To solve the CSP that contains all the properties to respect, Ivy uses the Z3 solver. After the generation of this packet, it is sent on the mirror channel. TLS Receives a packet. We consider TLS as a black box process: *we are just aware of its specification.* Once TLS finished the encryption of the packet, it sends it directly to the network. When the packet is received by the tested implementation, the network process sends its data to the tested implementation.

It is at the sending and the receiving event of each process that we can verify properties.

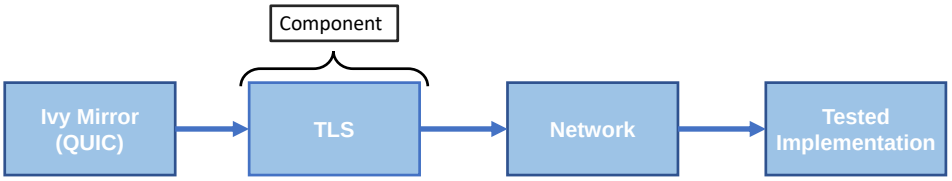


Figure 2.9: Channel

(2) Specification

The specification groups the set the properties that a particular process must respect. The global specification is the composition of all the specifications from all processes. It is composed of safety properties. These properties state that nothing bad should happen during the execution [35]. A property that a `QUIC` field must not be equal to zero is a safety property. There are also liveness properties [36]. These properties state that something good eventually happens. The property that says that an endpoint should eventually retransmit a specific packet if it is lost is a liveness property. Verifying `QUIC` liveness properties are complex. McMillan and Zuck chose not to test them. We decided to include some of them using heuristics. A typical heuristic that we used is the expiration of a timer. For example, we have a timeout for the handshake. After a given amount of time, we consider that the handshake is not completed. Heuristic can lead to false positive results that we analyse in section 3.6. However, it allows to verify more properties.

We represent each of the properties as an action. An action is a guarded command that may have some parameters and is represented using first-order logic. We can set guards before or after this command. The statement will not be executed if the guard is false. When some event occurs, we can update this action to modify its state. Those actions help us to generate some valid input as we apply our `Z3` solver to generate events that are valid for the different guards of the actions.

a. Mirrors

Mirrors are processes that help us to generate all the scope of possible values that will not make the global specification fail. If the process that we are testing is sending a message that will break the specification, we conclude that it is either its fault or that the specification is too weak.

On Figure 2.10, we see that Mirror is generating `QUIC frames`. Some of them are valid (green) while others are not (red). Using the `Z3` solver, `Ivy` selects only the frames that are respecting all the properties of the specification. Then, those frames are included in a `QUIC` packet. There is a random generation of packet that follows the same process but this is not shown here. Then, the packet containing the randomized frames are sent to the network process.

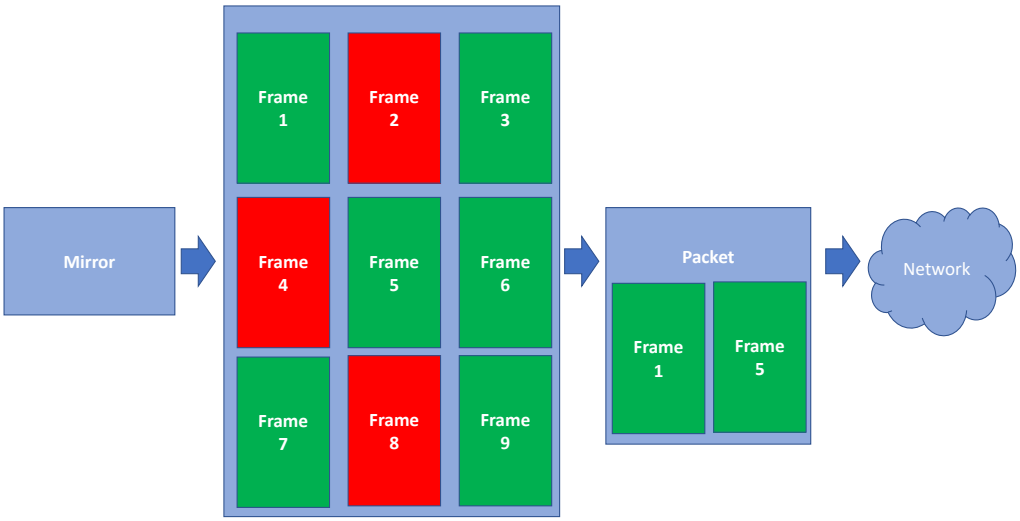


Figure 2.10: Mirror

b. Shim

The shim is the interface that allows us to translate the physical events that happen on the network into actions. It is first triggered on the receiving side of the network process. It orchestrates everything that needs to be done. The `Ivy` protection module is called to decrypt the packet using an `TLS` API. All the different relations and actions are updated and the properties verified. The shim is also used for the translation of the `Ivy` abstraction in a concrete packet. Once the mirror generates a packet to send, the shim encrypts it using the protection module. Then, it sends it to the network process.

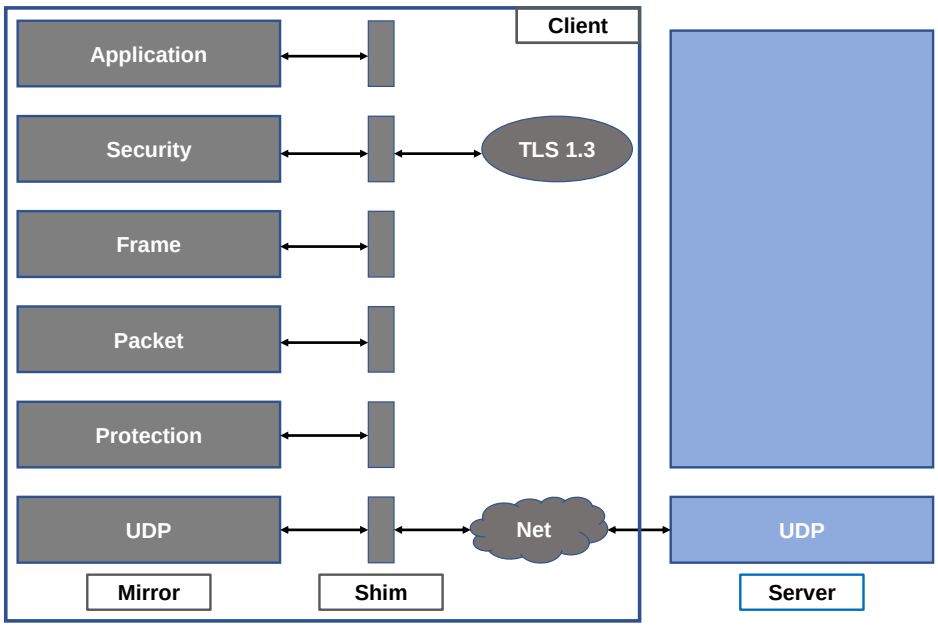


Figure 2.11: The Shim [2]

c. Randomized testing

NCT involves some non-deterministic choices. The first one is the choice of the mirror process: it can produce anything that will satisfy its guard. The second choice is the order of execution of Ivy events. Also, the tested implementation can have non-deterministic choices that we are unaware of.

If the choices were purely random, there would be some issues. When we test an implementation, it would not be convenient to have as much STREAM frames as STREAM_RESET frames in every scenario. There would not be much content to analyse. McMillan and Zuck created a solution to solve this problem. They put weights to the different kinds of frames to influence their distribution on generations.

Attributes weight

```
1 attribute frame.rst_stream.handle.weight = "0.02"
2 attribute frame.max_stream_id.handle.weight = "0.02"
3 attribute frame.max_stream_data.handle.weight = "0.02"
4 attribute frame.max_data.handle.weight = "0.02"
5 attribute frame.path_response.handle.weight = "5"
```

The weights are put at the begin of a scenario. If the value is high, the distribution of the frame will increase. The distribution is a Monte Carlo sampling based on the declared relative weight [37].

2.2.3 QUIC specification in Ivy

This section explains the main Ivy concepts used for the specification-based testing of QUIC implementations. It is based on the official documentation of Ivy [25]. We include some examples specific to the QUIC testing to point out the concepts presented in the Ivy version 1.7 [23]. We only present a subset of Ivy functionalities and encourages the reader to read the official documentation for a more detailed explanation.

We first cover the fundamentals of the Ivy language. Then, we more specifically address the way Ivy is used to test network protocols and QUIC specifically.

What is Ivy ?

"Ivy is a research tool intended to aid the interactive development of protocols and their proof of correctness. It also provides a platform for developing and experimenting with automated proof techniques. In particular, Ivy *provides interactive visualization of automated proofs, and supports a used model in which the human protocol designer and the automated tool interact to expose errors and prove correctness."*[25] It can also extract executable programs by translating them from python to C++ for efficiency. [38]

First, the Ivy language supports three kinds of objects [23]:

1. **Data values**, which are the variables stored in the state of our program.
2. **Function values**, which are mathematical functions: from a given input it always gives the same output and does not modify the environment.
3. **Procedure** can modify the environment which allows different results for different executions. It can return at most one value.

We can also use high-order programming with the Ivy language by putting into variable data and function values. There are no references in Ivy. Two variables always lead to two different objects.

The language is synchronous, which means that all actions occur in reaction to changes in the environment and that they are also all isolated. Consequently, they seem to occur instantly.

These features are present because most verification problems such as invariant checking and model checking rely on a subset of first-order logic formulas: decidable fragments [39]. More details are provided in the appendix C.

More information about Ivy concepts used for implementing the specification of QUIC can be found in the appendix B.

QUIC specification

We explore the specification of the different QUIC layers and summarize how they are implemented in Ivy:

1. Application over QUIC
2. TLS handshake security
3. QUIC Frame protocol
4. QUIC Packet protocol
5. QUIC Packet Protection for reliability
6. UDP

See Figure 2.12 and Appendix G for a graphical version of the architecture. QUIC specification is split into two parts. The first one contains a file for each layer of the QUIC protocol. The other part contains all the definitions used in the first one. For example, the transport parameters and the errors are defined there.

The shim coordinates these two parts. It uses the serializer and the deserializer to be able to generate and receive real packets on the wire. The shim is the conductor that allows every component to synchronize.

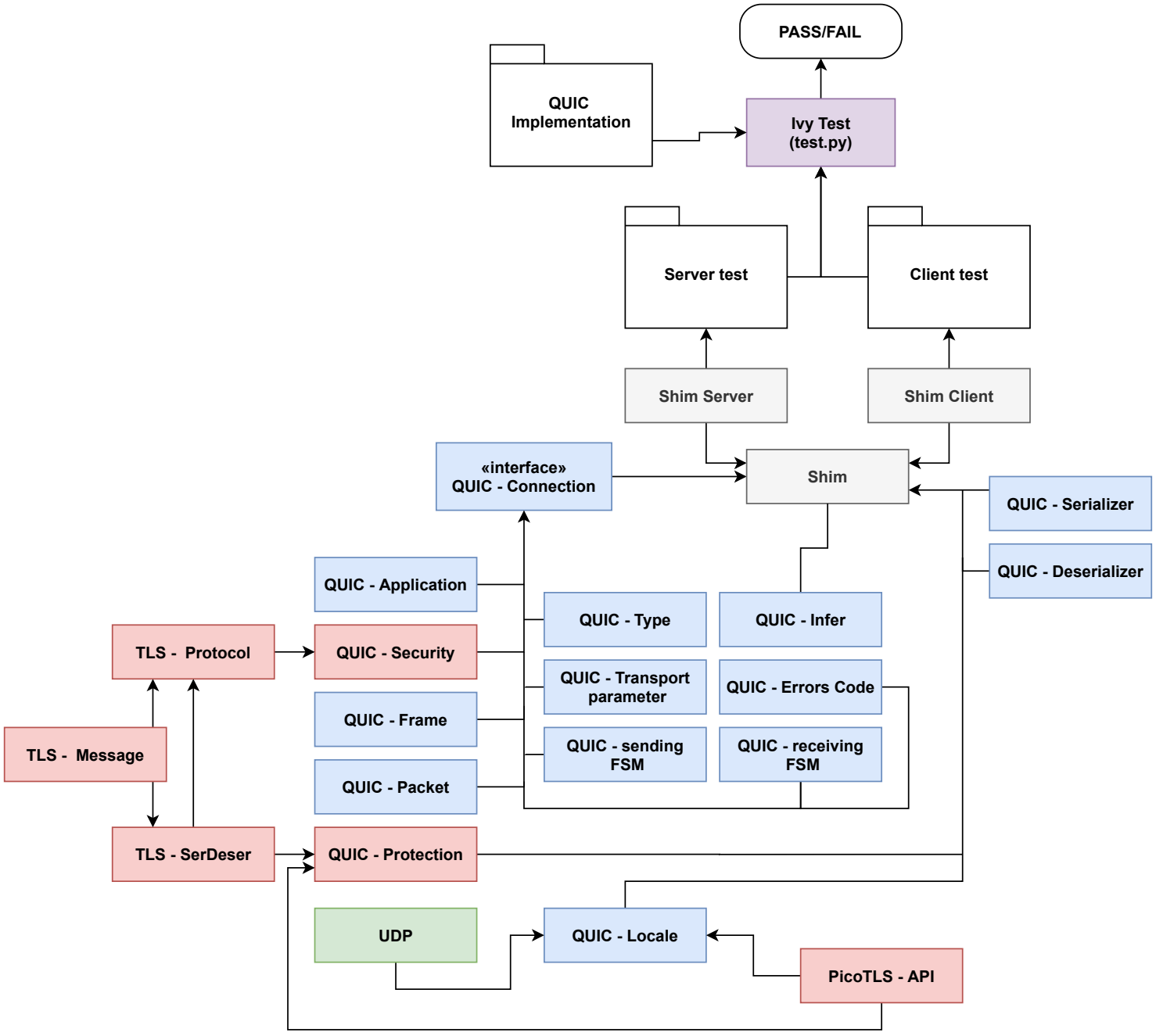


Figure 2.12: Architecture

A. QUIC stack

In this part we explain the different layers present in the QUIC stack from the top to the bottom. The highest level of the stack is the application and the lowest level is the UDP datagrams.

The main file for representing a QUIC connection can be found in quic_connection.ivy. This file is like an interface that includes all remaining components of the specification of Ivy.

1. QUIC application:

Provides flow-controlled streams for structured communication between clients and servers in a reliable and secure way. The main events are corresponding to opening new streams and the transfer of application data to be transmitted into QUIC streams. The corresponding file is quic_application.ivy.

2. QUIC security:

Implement the cryptographic handshake used to establish identities of the endpoints within a connection. It allows representing the establishment of the cryptographic keys for each encryption levels defined by the specification in the model. It uses the TLS protocol to deserialize, serialize and handle the TLS messages.

TLS protocol In tls_record.ivy and tls_protocol.ivy, one can find a model of the TLS v1.3. The first part defines all the different TLS message possible such as "Server Hello" or "Client Hello" for example. The second one how to handle TLS events. Finally, in tls_deser_ser.ivy, we define the parser with the serializer and deserializer of the TLS messages that allow to make the link between the raw data directly in the model and the wire.

3. QUIC frames:

The file quic_frame.ivy defines a type object to represent a general QUIC frame. This type object is shared among all the frames. We give more specific definitions of the different frames after. A specific frame is called a variant frame. We define them with their corresponding fields as in the specification. An associated action to explain how to handle them is also added. This action contains all the set of requirements associated to the frame.

Frames

```
1 object frame = {
2     type this # The base type for frames
3     object ping = { # (0x01), ping frames contain no data
4         variant this of frame = struct {
5         }
6     }
7 }
```

```
object frame = { ...
    object ping = { ...
        action handle(f:frame.ping, scid:cid, dcid:cid,
            e:quic_packet_type, seq_num:pkt_num) # Refinement
         around handle {
                require connected(dcid) & connected_to(dcid) =
                    scid;
                require e = quic_packet_type.one_rtt ->
                    established_1rtt_keys(scid);
                require num_queued_frames(scid) > 0 -> e =
                    queued_level(scid);
                ...
                call enqueue_frame(scid,f,e,false);
            }
    }
}
```

4. QUIC packets:

QUIC packets are modelled in `quic_packet.ivy` and contains a type object representing a QUIC packet:

Packets

```
object quic_packet = {
    type this = struct {
        ptype : quic_packet_type,
        pversion : version,
        dst_cid : cid,
        src_cid : cid,
        token : stream_data,
        seq_num : pkt_num,
        payload : frame.arr
    }
}
```

One main action is defined `packet_event` and include all the set of requirements that are specific to the packets. We also defined a series of different actions that allows handling the TLS extensions and the transport parameters. Different requirements and relations are associated and initialized from these messages.
It is not defined like the frames with variant for the different type of packets but we think we should proceed this way. However the current modernization is tightly coupled with the current serializer and deserialiser.

5. QUIC packet protection:

In `quic_protection.ivy`, one can find a representation of protected packets. We directly linked this component to the PicoTLS API component that allows decrypting and encrypt protected packets. All the required field needed for decryption and encryption defined in QUIC-TLS [8] such as the packet's sample are parsed here and send to the PicoTLS API.

6. UDP:

Finally in udp_impl.ivy, a generic implementation of an UDP endpoint is defined. We allow opening a socket and send and receive a packet on this socket. This part is not written in Ivy language but in C++.

B. QUIC utils

In order to refine the model, we can define new components.

QUIC Shim

As explained in section 2.2.2, it allows to translate the physical events that happen on the network into actions in Ivy. We orchestrate the whole connection by calling all other defined components. We also refine two sub-shims with specific actions for the server and for the client.

QUIC transport parameters

The different QUIC transport parameters are represented as different modules differentiated by their respective field and value. More details are provided in quic_transport_parameters.ivy.

QUIC errors code

Define all the QUIC errors and how to handle them. More details are provided in quic_transport_error_code.ivy.

QUIC FSMs

We also have an explicit representation of the stream FSM in quic_fsm_sending.ivy and quic_fsm_receiving.ivy. However, all requirements are not enforced due to liveness properties but it can be used to give an indication of the current state.

QUIC types

In this file (quic_types.ivy), one can find all the definitions of elementary common types that are used in the QUIC specification, such as Connection ID type cid, the protocol version version, the packet number pkt_num and many other ones.

QUIC infer

Some events are not directly observed on the wire but need to be inferred for the shim. For example, when we received a packet containing STREAM frames, we can infer and call application-level events. In the same reasoning is applied for TLS events.

QUIC locale

Contains the instances of the UDP and PicoTLS components that will be used during the connection.

QUIC Ivy serializer/deserializer

These are two important files used to put and get states/objects of the QUIC protocol (interact with the preceding concept). It is used for the concrete generation and parsing of the packets and frames.

QUIC tests

Finally we also define a set of tests detailed in section 3.4. We also present an example for the error in section 3.1.1. In these tests, we add specific requirements on the generated entities. For a server, but we apply the same logic for clients, most of the tests have a common set of requirements defined in `quic_server` and `quic_server_test`. Then we refine these basic structure with new requirements for the packets and frames generated.

2.3 Related work

There are many ways to test if a network protocol obeys an RFC. Some methods involve interoperability. A good reference of this method can be found here [40] where an active test suite is proposed and a large range of QUIC implementations tested. Another example of a tool suite is H3spec [41].

Besides, there are some approaches to test formally QUIC. As in our work, we do not test a large part of the TLS properties because we consider this component as a black box in our model, there was an attempt to provide a formal verification of the QUIC handshake and the TLS properties linked to it. Two state-of-the-art verifiers, Proverif and Veripal, were used. A design error of the QUIC handshake was found [42].

An attempt of the formal verification of TLS 1.3 has also been proposed [43]. Other works provide a formal verification of QUIC using symbol execution [44].

Chapter 3

Contributions

We now present our improvements to `Ivy`. There are three types of contributions. The first one is to update the `QUIC` module of `Ivy` from draft 18 [24] to draft 29 [5]. We present in section 3.1 the rules that were added or updated. We highlight the rules that were already present and the ones that we updated or created.

We concretely show our different contributions. We needed a better environment to help us update the `QUIC` draft. It will also allow other contributors from not being discouraged by the difficulty of installing `Ivy` if they want to contribute to it.

Note that as `Ivy` is not specific to `QUIC` and can be used to test other networking protocols, it is also possible to reuse our work without specifically focusing on `QUIC`.

The final part is about the `QUIC` extension that we verified. First, we show how it is possible to test new extensions of `QUIC`. Then, we show the issues and the limitations we faced.

We end by explaining the tests created for `QUIC` servers and clients.

We performed a reproduction of the results that McMillan presented in his paper "*Formal specification and testing of* `QUIC`". It is present in appendix D. We also have in appendix H the improvements to the deployment, the ease of use and the output of the testing of `QUIC` using `Ivy`.

3.1 Update to `QUIC` 29

The `QUIC` specification contains a series of *MUST*, *MUST NOT*, *SHOULD* and *SHOULD NOT* statements [5]. We list every *MUST* and *MUST NOT* statement of draft 29 and perform two checks. First, we verify if this rule is already implemented in `Ivy`. Then, we check if it is possible to verify it. Section F in the appendix summarizes our work with a colour code that specifies whether the property from the draft is supported by McMillan (dark green), by us (green), impossible to test

(blue), not implemented (red) or when only one part of the requirement is checked (orange). Usually, this is because the requirement is a mix between safety and liveness properties.

However, these types of requirements are not the only ones. There is also a larger set of rules where no "*MUST*" is present. An example is the following statement:

> "*An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.*"
>
> Draft 29 of QUIC section 9.3.

In QUIC draft 29, we listed a total of 268 "*MUST*" statements:

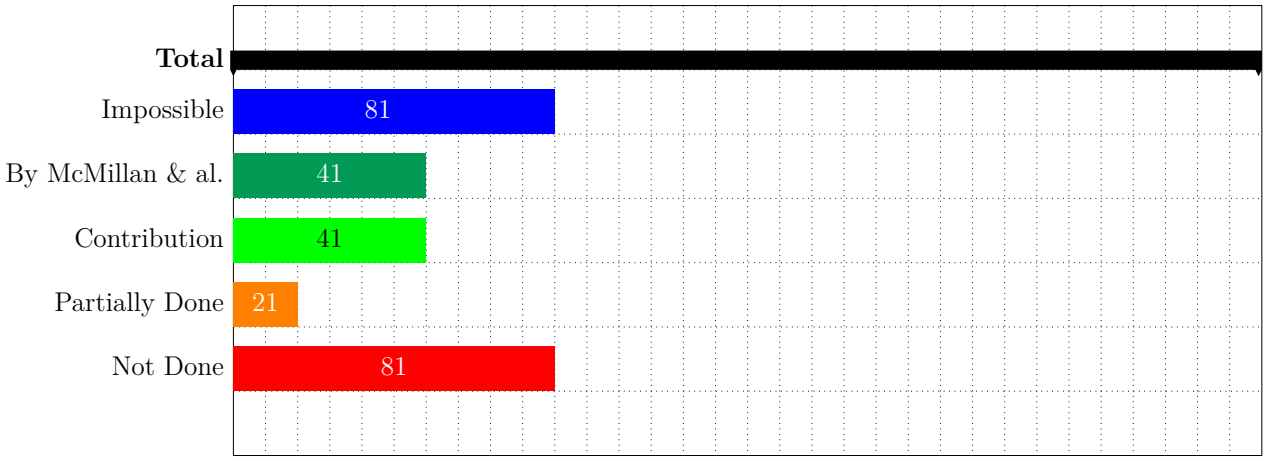


Figure 3.1: QUIC Transport - 268 MUST in total

Note that the contribution of McMillan is broader: he also implemented requirements that are not explicitly marked with a "*MUST*". We must consider the fact that McMillan did not explicitly list the requirements so we had to estimate the work done. We also implemented some of these requirements. Most of our contribution, only considering the QUIC specification, is for the errors and transport parameters where most of the possible requirements have been implemented.

Most of the missing requirements are linked to the non-implemented QUIC packets. We explored the possibility to update this part. By lack of time and considering that the serializer and deserializer are highly dependent on the current packet representation, we preferred to leave it for future work.

Many assumptions of McMillan have also been resolved. For example, in the original version, only 8 bytes connection IDs were considered for the protected packets during the parsing.

3.1.1 QUIC errors

First of all, we tested the behaviour in case of the reception of QUIC packets that would break the specification. One example of such rule is:

> *"Token Length: A variable-length integer specifying the length of the Token field, in bytes. This value is zero if no token is present. Initial packets sent by the server MUST set the Token Length field to zero; clients that receive an Initial packet with a non-zero Token Length field MUST either discard the packet or generate a connection error of type PROTOCOL_VIOLATION. "*
>
> Draft 29 of QUIC section 17.2.2.

We created a test that would voluntarily generate packets that would trigger this error according to the draft. For that, we represent each type of error with an Ivy relation indicating if a specific error has been thrown by the tested implementation or not.

Handle error

```
1  action handle_transport_error(ec:error_code) = {
2      if ec = 0x0 {
3          is_no_error := true;
4      } else if ec = 0x1 {
5          is_internal_error := true;
6      }
7      # [...]
8      else if ec = 0xd {
9          is_crypto_buffer_exceeded := true;
10     } else {
11         is_crypto_error := true;
12     }
13 }
```

Then, in the frames handling errors such as CONNECTION_CLOSE, we call our function to set the global state of errors:

Connection close

```
1  object frame = { ...
2      object connection_close = { ...
3          action handle(f:frame.connection_close,scid:cid,dcid:cid,
4                        e:_packet_type,seq_num:pkt_num)
5   around handle{
6              require connected(dcid) & connected_to(dcid) = scid;
7              # [...]
8              require f.reason_phrase_length = f.reason_phrase.end;
9              conn_closed(scid) := true;
10             call handle_transport_error(f.err_code);
11       ...
12             call enqueue_frame(scid,f,e,false);
13         }
14     }
15 }
```

We can differentiate the packets and frames that we receive from the implementations with our own packets and frames. The keyword `_generating` is used to add additional requirements to the frames/packets generated by the `Ivy` mirror. See appendix B for more details:

Token Error

```
1  before packet_event(src:ip.endpoint, dst:ip.endpoint,
       pkt:quic_packet) {
2      if _generating {
3          # [...]
4          require ~(pkt.token.end = 0); # Wrong field generation
5      };
6      require pkt.long -> pkt.pversion = 0xff00001d
7  }
```

Finally, we test if the implementation *eventually* respects given requirements with the special instruction `_finalize`:

```
1  export action _finalize = {
2      require is_invalid_token | ~handshake_done_send;
3  }
```

This is a new approach compared to the one described by McMillan. We had before a specification that contained only requirements of the draft. Now we added requirements that are not present in the draft. It allows us to generate illegal packets. The Network-centric Compositional Testing model cannot be applied when we use this approach. We now use simple fuzzing. This open new doors for `Ivy`: we are now able to test the implementation against erroneous behaviour.

3.1.2 QUIC unknowns

We also defined a new type of frame which is not defined in the `QUIC` specification. Its purpose is to see how the tested implementations behave with an unknown type of frame:

unknown_frame

```
1  object unknown_frame = {    # 0x42
2      # contains nothing
3      variant this of frame = struct {
4      }
5  }
```

We applied the same idea to the transport parameters with a generic `unknown_transport_parameter` which allows us to be more flexible and not to trigger memory errors each time we encounter an unknown transport parameter. Here again, a modification of the structure to the serializer/deserializer was needed since it was not originally designed for that purpose. To achieve that, we perform some sort of double parsing: when we detect an unknown transport parameter, we first have to find the length of the transport parameter tag, then the length of the value itself.

unknown_transport_parameter

```
object unknown_transport_parameter = { # tag = -1
    variant this of transport_parameter = struct {
        unknown : stream_pos
    }
    instantiate trans_params_ops(this)
}
```

For the generation of unknown transport parameters, we defined the `unknown_ignore` transport parameter. We do not reuse `unknown_transport_parameter` because it is unadapted for serialization.

3.1.3 QUIC transport parameters

Another change was to update the transport parameters. They are all implemented but some are not enforced. A few of them are not tested since the QUIC specification does not state real properties for them. However, we can still parse them to avoid any issue. This update involved a rewriting of the serializer and deserializer because the encoding changed a lot since draft 18.
Here is an example of a `preferred_address` transport parameter:

preferred_address

```
object preferred_address = {
    variant this of transport_parameter = struct {  # tag = 4
        ip_addr: ip.addr, ip_port: ip.port,
        ip6_addr: ipv6.addr, ip6_port: ipv6.port,
        pcid_len: stream_pos, pcid : cid, pref_token: stream_data
    }
    instantiate trans_params_ops(this)
}
```

3.1.4 IPv6 model

We also make a simple model of IPv6 needed in the `preferred_address` transport parameter:

IPv6

```
object ipv6 = {
    type addr; type port; type protocol = {udp6,tcp6}
    object endpoint = {
        type this = struct {
            protocol : ipv6.protocol,
            addr : ipv6.addr, port : ipv6.port
        }
    }
    implementation {
        interpret addr -> longbv[1][128][3]
        interpret port -> bv[16]
    }
}
```

3.1.5 Modelling a QUIC extension

We also represent and define some properties for the `min_ack_delay` transport extensions defined in [13] and model most of the feasible properties. It is not possible to test every property since it usually makes an assumption on time or order. In Ivy, it is not yet possible to verify properties based on the time. We explore some possible solutions in section 5.4.

min_ack_delay

```
1  object ack_frequency = {
2       variant this of frame = struct {
3          seq_num : pkt_num,  packet_tolerence : stream_pos,
4          update_max_ack_delay : microsecs, ignore_order : bool
5      }
6  }
7  object frame = { ...
8      object ack_frequency = { ...
9          action handle(f:frame.ack_frequency,scid:cid,
10                       dcid:cid,e:quic_packet_type)
11   around handle {
12             require connected(dcid) & connected_to(dcid) = scid;
13             require e = quic_packet_type.one_rtt &
                   established_1rtt_keys(scid);
14             require num_queued_frames(scid) > 0 -> e =
                   queued_level(scid);
15             ...
16             var tp := trans_params(dcid);
17             var min : microsecs := 0;
18             if min_ack_delay.is_set(tp) {
19                 min := min_ack_delay.
20                         value(trans_params(dcid)).exponent_8
21             };
22             require f.packet_tolerence > 0;
23             require f.update_max_ack_delay > min;
24             require f.ignore_order = 1 | f.ignore_order = 0;
25             if first_ack_freq_received {
26                 require f.seq_num = 0;
27                 first_ack_freq_received := false;
28             } else {
29                 require f.seq_num > last_ack_freq_seq(scid);
30                 last_ack_freq_seq(scid) := f.seq_num;
31             };
32             call enqueue_frame(scid,f,e,false);
33         }
34     }
35 }
```

Unfortunately, most of the interesting requirements of this extension could not be modelled properly since time cannot be represented. However it shows that we can use formal verification with Ivy to test the safety properties of a QUIC extension.

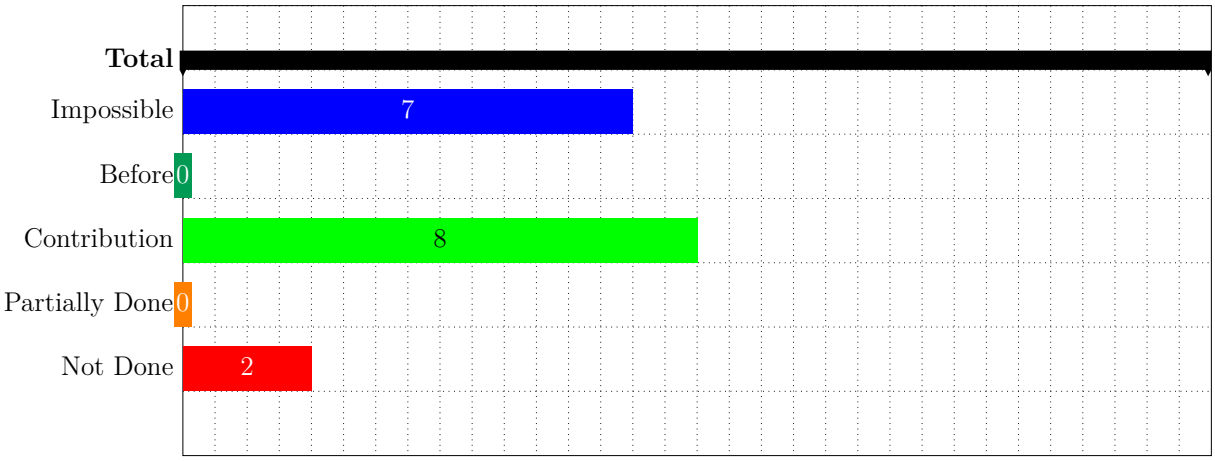


Figure 3.2: QUIC Min Ack Delay - 17 MUST in total

In the future, we could have implemented the `loss bit` extension [45] or the `grease bit` extension [46]. However they only define new transport parameters and a small set of requirements.

Another interesting case is the `Multipath QUIC` extension [47]. In retrospect it would have been more interesting to focus on this case but its higher complexity and the problems that we had with the standard `QUIC` specification made us reconsider this choice.

3.1.6 Shim update

We did many refactoring and updates to the shim described in 2.2.2 such that it was compatible with the current version of `QUIC`. For example, due to the previous implementation of the transport parameter, we had to update the way the client generates the transport parameter. Some transport parameters were already set before the sending of the Initial packet resulting in invalid connections. We also had to modify this part to be correct and be flexible with special implementations such as `mvfst` that for example use its version number. We had to adapt the salt for each case.

3.1.7 QUIC sending/receiving FSM

We also attempted to model the sending and receiving FSM of `QUIC` streams. However, during the development of these features, we realized that most of the feasible part had already been done. The impossible ones are the states where we have to make an assumption on time or order. However, even if no real interesting requirements have been added to this part, we now have an approximation of each state that can be retrieved in the `Ivy` logs. It is an "approximation" since we cannot guarantee them because some states are using heuristics. We also cannot be sure of the FSM of the tested implementation.

3.2 Refactoring of the model

We merged the shim for the client and the server and created two new files, `quic_shim_client` and `quic_shim_server` instead of the group of files that were distributed in several modules. We also created a subdirectory such that we do not have too many files in one directory and the files are grouped by layer. This was not trivial since `Ivy` only detects within the `include` statements the files that are in the same directory. We solved that issue by writing script that rearranges all files before and after the execution of a test.

3.3 Ivy improvements

In order to represent a value higher than 8 bytes as `Ivy` originally proposed, we updated `Ivy` itself to add a new type that we called `LongBV`.It allows representing a large range of integers using a `int128_t`. It is a bit vector. We explore solutions to have more than 16 bytes in Section 5.4.

This required to update the implementation of many C++ templates:

1. `ivy_cpp_types.py` where we define the python class `LongBV` directly.
2. `ivy_solver.py` that allows to link the `LongBV` python class and the `longbv` type in `Ivy` itself.
3. `ivy_to_cpp.py` where we reimplement all serializer/deserializer for the `int128_t` type. Here is an example:

Interface for the new Ivy type

```
1  struct ivy_ser_128 {
2      virtual void  set(int128_t) = 0;
3      virtual void  set(bool) = 0;
4      virtual void  setn(int128_t inp, int len) = 0;
5      virtual void  set(const std::string &) = 0;
6      virtual void  open_list(int len) = 0;
7      virtual void  close_list() = 0;
8      virtual void  open_list_elem() = 0;
9      virtual void  close_list_elem() = 0;
10     virtual void  open_struct() = 0;
11     virtual void  close_struct() = 0;
12     virtual void  open_field(const std::string &) = 0;
13     virtual void  close_field() = 0;
14     virtual void  open_tag(int, const std::string &) {
15         std::cout << "ivy_ser_128 open_tag deser_err" <<
               std::endl;
16      throw deser_err();
17     }
18     virtual void  close_tag() {}
19     virtual ~ivy_ser_128(){}
20 };
```

3.4 Server Tests

We have 24 different tests where the tested implementation is the server. Some of them are generic. For example, with the stream test, we test the complete specification and include the generation of the stream frame. Some other tests are more specific where we expect a specific event. With the error tests, we generate a packet that should trigger an error and we require the reception of the corresponding error. In each test, all the requirements from the specification that we implemented are checked. If one requirement is not met, we report it.

3.4.1 Global tests

First, we use the original tests developed by McMillan and Zuck. There were four tests for the server and one test for the client side. We adapted them to draft 29. `quic_server_test_stream` is a general scenario for testing streams. During this test, we allow multiple connection migrations. We can see if the tested implementation can handle it. Several streams are open to see if the stream management is correct. We create bidirectional streams by increasing the stream ID by four.

We have a second test, `quic_server_test_max` that is specific to the handling of the `MAX_STREAM_DATA` and `STREAM_DATA_BLOCKED` frames. We test if the data received on a stream is not over the `MAX_STREAM_DATA` threshold.

The last two tests, `quic_server_test_reset_stream` and `quic_server_test_connection_close` test respectively the management `RESET_STREAM` frame if the stream is well reset and how implementations close their connection.

quic_server_test_stream
Verifies all implemented requirements and generates STREAM frame and may perform connection migrations
quic_server_test_max
Verifies all implemented requirements and generates STREAM. Also MAX_STREAM_DATA, STREAM_DATA_BLOCKED and MAX_DATA frames with limited range field values.
quic_server_test_reset_stream
Verifies all implemented requirements and generates STREAM and RESET_STREAM frames with random error codes.
quic_server_test_connection_close
Verifies all implemented requirements and generates STREAM and CONNECTION_CLOSE frames with random error codes.

Table 3.1: Original tests proposed by McMillan

We also added `quic_server_test_maxdata` that is similar to `quic_server_test_max` except that we restrict to the `MAX_DATA` frame. More values are allowed in the frame's field domain. The coverage of the test is also higher than `quic_server_test_ max` for this frame.

The `STOP_SENDING` frame is also tested in the `quic_server_test_stop_sending`.

Finally, there is the `quic_server_test_ext_min_ack_delay` that holds all the properties of the `min_ack_delay` `QUIC` extension that we test. We did not test the core properties of the extension because these properties are liveness properties or are based on time. However, this test helped to show that we can easily add new frames and new behaviour for the extensions. Testing the properties of an extension has the same limitations as testing the core `QUIC` specification. Some properties are impossible to test.

quic_server_test_stop_sending
Verifies all implemented requirements and generates STREAM and STOP_SENDING frames.
quic_server_test_accept_maxdata
Verifies all implemented requirements and generates STREAM and MAX_DATA frames with full random field values. Endpoint should accept these frames even if they carry smaller maximum values.
quic_server_test_ext_min_ack_delay
Verifies all implemented requirements of the standard draft and the min_ack_delay transport extension.

Table 3.2: Global tests verifying normal execution of a feature

3.4.2 Unknown situations

The test `quic_server_test_unkown` checks the behaviour when receiving a frame of an unknown type after the handshake is confirmed. A `FRAME_ENCODING_ERROR` is expected according to the draft. For this test, we allow generation of this frame at the application encryption level and verify the good reception of the corresponding error.

`quic_server_test_unkown_tp` is similar but for the transport parameter. However, no error should be thrown for this test. We just verify if the tested implementation can handle it.

quic_server_test_unkown
Verifies handling of UNKNOWN frame type by the implementation. Only allowed in application encryption level so we must get a FRAME_ENCODING_ERROR.
quic_server_test_unkown_tp
Verifies handling of UNKNOWN transport parameter by the implementation. Must ignore it and continue execution. We disable migration in this case.

Table 3.3: Tests verifying handling of unknown

3.4.3 Error handling

Several tests concern the transport parameters. This is one of the parts of the specification that changed the most compared to the specification used by McMillan & Zuck. With `quic_server_test_tp_error`, we put an incorrect value in one of the transport parameters and expect a `TRANSPORT_PARAMETER_ERROR`.

We have a test for the `active_connection_id_limit` transport parameter in the test `quic_server_test_tp_acticoid_error` where we initiate a connection

with this transport parameter value set at a value less than 2. We expect a TRANSPORT_PARAMETER_ERROR.

We are also testing a scenario in quic_server_test_no_icid where we voluntarily do not set the initial_source_connection_id transport parameter and check that the tested implementation throws a TRANSPORT_PARAMETER_ERROR.

In test quic_server_test_double_tp_error, we set twice a given transport parameter and should expect a TRANSPORT_PARAMETER_ERROR. The rule in the draft is ambiguous as it does not explicitly say what we should do in such a case. For example, if we set twice the max_ack_delay and the implementation do not throw an error, which values will it consider?

We can also argue that not throwing an error is invalid since receiving twice the same transport parameter means that the protocol has been violated. We must not send a transport parameter more than once. We believe that the minimum behaviour is to send a PROTOCOL_VIOLATION or not to finish the handshake. But considering only the requirement presented hereafter we can also suppose that sending an error is not mandatory.

We interpret from the draft that we should send an error or that we should not finish the handshake. In other cases, the test fails. We created a heuristic that says that if we did not receive a HANDSHAKE_DONE when the test finished, then the handshake never completed. This is an example of a liveness property that we test. The requirement we consider is the following one:

> *An endpoint MUST NOT send a parameter more than once in a given transport parameters extension. An endpoint SHOULD treat receipt of duplicate transport parameters as a connection error of type* TRANSPORT_PARAMETER_ERROR.
>
> Draft 29 of QUIC section 7.4.

With more time it could be interesting to test all transport parameters to see the full coverage.

quic_server_test_tp_error
Invalid value for ack_delay_exponent transport parameter. Must have TRANS PORT_PARAMETER_ERROR.
quic_server_test_double_tp_error
Set twice a transport parameter which is not allowed. However draft says that we SHOULD consider a TRANSPORT_PARAMETER_ERROR. Since it is a "should" but is not allowed. We consider failure of handshake valid or any error.
quic_server_test_tp_acticoid_error
Invalid value for active_connection_id_limit transport parameter, Must have TRANSPORT_PARAMETER_ERROR.
quic_server_test_no_icid
Absence of initial_source_connection_id transport parameter. Must have a TRANSPORT_PARAMETER_ERROR.

Table 3.4: Transport parameter errors tests

Then we also define tests where we generate frames with invalid fields. For example, we have one test about the handling of STREAM_BLOCKED frames. We generate this frame with an invalid "maximum stream" field value. The draft states

that upon receiving the frame, the server should send a `STREAM_LIMIT_ERROR` or `FRAME_ENCODING_ERROR`. We verify if this is the case.

quic_server_test_blocked_streams_maxstream_error
Generation of STREAMS_BLOCKED frames with invalid "Maximum Streams" field. Must have STREAM_LIMIT_ERROR or FRAME_ENCODING_ERROR.
quic_server_test_retirecoid_error
Generation of RETIRE_CONNECTION_ID frames with invalid "Sequence Number" field. Must have PROTOCOL_VIOLATION error.
quic_server_test_stream_limit
Generation of STREAM frames with invalid "Offset" field. Must have FLOW_CONTROL_ERROR or FRAME_ENCODING_ERROR
quic_server_test_newcoid_rtp_error
Generation of NEW_CONNECTION_ID frames with invalid "Retire To Prior" field. Must have FRAME_ENCODING_ERROR
quic_server_test_newcoid_length_error
Generation of NEW_CONNECTION_ID frames with invalid "Length" field. Must have FRAME_ENCODING_ERROR
quic_server_test_max_error
Generation of MAX_STREAMS frames with invalid Maximum stream value. Must have STREAM_LIMIT_ERROR

Table 3.5: Violation of fields in frame tests

Finally, we also have a test that explicitly violates the draft. The simplest example is sending `HANDSHAKE_DONE` frames as clients which is not allowed.

The test `quic_server_test_newconnectionid_error` is a test that we use more as a fuzzing test. It is because we generate this frame with random "Retire To Prior" field used to mark connection ID that should be removed. This allows to see how the implementation reacts in nonconforming situations. We may expect an error if a value we generate does not make sense and break the specification. For example, if we generate a sequence number and that it contains an already used connection ID, we expect an error.

We have another test where we put a non-zero token field in an initial packet. If a server receives such a packet, it must either discard the packet or throw a `PROTOCOL_VIOLATION` error. We consider that the test succeeds if we received a `PROTOCOL_VIOLATION` error or if the handshake did not complete. In `quic_server_test_handshake_done_error`, we send several `handshake_done` frames at any time. According to the draft, a server should send this frame only before completing the handshake or return a `PROTOCOL_VIOLATION` error. This frame should only be sent by the server. Whenever we send this frame as a client, we should receive `PROTOCOL_VIOLATION` according to section 19.20 of the draft.

quic_server_test_handshake_done_error
Generation of HANDSHAKE_DONE frames as client. Must have a PROTOCOL_ VIOLATION.
quic_server_test_new_token_error
Generation of NEW_TOKEN frames as client. Must have a PROTOCOL_ VIOLATION.
quic_server_test_token_error
Generation of Initial packet with non-zero Token length field. Either discard of packet is allowed, so failure of handshake. Either PROTOCOL_VIOLATION.
quic_server_test_max_limit_error
Generation of STREAMS frames with stream ID limit exceeding the limit set by the peer. Must have STREAM_LIMIT_ERROR
quic_server_test_newconnectionid_error
Generation of NEW_CONNECTION_ID frames with random "Retire To Prior" field and random connection ID. Should at least throw any type of error since this break many rules. Allow to see how implementation react non conform situation.

Table 3.6: Violation of draft tests

3.5 Client Tests

McMillan & Zuck. initially made one test similar to their server counterpart. This is the `quic_client_test_max` where the scenario is like `quic_server_test_max`.

We added 14 more tests. Most of them are like their server counterpart: the scenario is the same. It is the case for the following tests : `quic_client_test_unkown`, `quic_client_test_unkown_tp`, `quic_client_test_tp_acticoid_error`, `quic_client_test_tp_error` , `quic_client_test_no_odci`, `quic_client_test_double_tp_error`, `quic_client_test_ext_min_ack_delay`, `quic_client_test_retirecoid_error`, `quic_client_test_max_limit_error` and `quic_client_test_accept_max data`.

For the client part, we never perform connection migration. The reason is that it is the client that triggers the migration and it is impossible to activate a connection migration for all the tested implementations without changing their code. We preferred to disable the migration even for the implementations that allowed this possibility. This way, we are able to compare more easily the results between all the tested implementations.

Two tests are specific to the client, `quic_client_test_tp_prefadd_error` and `quic_client_test_new_token_error`. For the first one, we verify that the `preferred_address` transport parameter contains zero-length connection ID and the tested implementation throw `TRANSPORT_PARAMETER_ERROR`. For the other one, we set the length field of the token to zero, which is not allowed by the draft.

quic_client_test_new_token_error
Generation of NEW_TOKEN frames with length field set to 0. Must have a PROTOCOL_VIOLATION.

Table 3.7: Violation of draft tests

quic_client_test_prefadd_error
Zero length CID for preferred_address transport parameter. Must have TRANSPORT_PARAMETER_ERROR.

Table 3.8: Transport parameter errors

3.6 Future work & improvements

Some points restricted our contributions and the coverage of our tests. First, originally, `Ivy` accepted variables of maximum eight bytes. It was not a problem for the implementations that McMillan sampled. We managed to improve `Ivy` to make it support data up to 16 bytes. In the last chapter, we explore some paths allowing us to overcome this problem. This peak value of 16 bytes for `Ivy` datatype limits some tests we could perform and is a constraint to generate a scenario for some implementations. One example is the Connection ID. The draft states that it can be between 0 and 20 bytes, but only a subset of implementations use a Connection ID longer than 16 bytes. The truncation of this value could lead to several issues. Consequently, we modified the implementations themselves. They now use a connection ID of at most 16 bytes. Generally, it consisted in adapting one global variable.

Another point we can make is that for some errors, we could have performed more tests. It would have significantly increased the time needed to launch all the tests without a great improvement in the quality of our results. For example, concerning global errors on transport parameters, we focus our tests by choosing an arbitrary value for a single transport parameter.

We saw some cases where the `Ivy` running time was causing a problem, especially when an implementation sends a batch of big packets in a short time. This slows `Ivy` and can cause the tested implementation to timeout. It could result in some false positives as shown with an example with `quic-go`, an implementation we present in the next chapter, where the following rule is apparently not respected:

```
require f.fin <-> (stream_app_data_finished(dcid,f.id)
  & offset+f.length = stream_app_data_end(dcid,f.id));
```

It indicates that when a `STREAM` frame has the `fin` bit set to 1, then we should have the corresponding stream finished. The total stream offset must also match, and inversely if the `fin` bit is set to 0.

In the QUIC specification, this corresponds to the following rule:

> *"The final size of a stream is always signalled to the recipient. The final size is the sum of the Offset and Length fields of a STREAM frame with a FIN flag, noting that these fields might be implicit."*
>
> Draft 29 of QUIC section 4.4.

src	dst	time	Information
4988	4443		1274 Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4988		1294 Handshake, DCID=0000000000000000, SCID=e87f172d, PKN: 0, CRYPTO
4443	4988		228 Handshake, DCID=0000000000000000, SCID=e87f172d, PKN: 1, CRYPTO
4443	4988		1294 Initial, DCID=0000000000000000, SCID=e87f172d, PKN: 1, PADDING, CRYPTO
4443	4988		1294 Initial, DCID=0000000000000000, SCID=e87f172d, PKN: 2, PADDING, CRYPTO
4988	4443		154 Handshake, DCID=e87f172d, SCID=0000000000000000, PKN: 13, CRYPTO, PADDING
4443	4988		1127 Handshake, DCID=0000000000000000, SCID=e87f172d, PKN: 2, ACK, CRYPTO
4443	4988		228 Handshake, DCID=0000000000000000, SCID=e87f172d, PKN: 3, CRYPTO
4443	4988		1394 Protected Payload (KP0), DCID=0000000000000000, PKN: 0, PADDING, PING
4443	4988		319 Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, NT, CRYPTO
4443	4988		319 Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, NT, CRYPTO
4443	4988		319 Protected Payload (KP0), DCID=0000000000000000, PKN: 4, DONE, NT, CRYPTO
4443	4988		1344 Protected Payload (KP0), DCID=0000000000000000, PKN: 5, PADDING, PING
4988	4443		103 Protected Payload (KP0), DCID=e87f172d, PKN: 15, STREAM(4), PADDING
4443	4988		74 Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK
4443	4988		1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 7, STREAM(4)
4443	4988		1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 8, STREAM(4)
4443	4988		1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 9, STREAM(4)
4443	4988		1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 10, STREAM(4)
4443	4988		187 Protected Payload (KP0), DCID=0000000000000000, PKN: 11, STREAM(4)
4443	4988		73 Protected Payload (KP0), DCID=0000000000000000, PKN: 12, STREAM(4)
4443	4988		319 Protected Payload (KP0), DCID=0000000000000000, PKN: 14, DONE, NT, CRYPTO
4443	4988		1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 15, STREAM(4)
4443	4988		1319 Protected Payload (KP0), DCID=0000000000000000, PKN: 16, PADDING, PING
4988	4443	41.63	88 Protected Payload (KP0), DCID=e87f172d, PKN: 32, ACK, PADDING
4988	4443	43.3848	113 Protected Payload (KP0), DCID=e87f172d, PKN: 36, STREAM(12), ACK, ACK, PADDING
4443	4988	43.3851	1303 Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, STREAM(4)
4443	4988	43.3852	1294 Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(4)
4443	4988	43.3852	187 Protected Payload (KP0), DCID=0000000000000000, PKN: 19, STREAM(4)
4443	4988	43.3855	73 Protected Payload (KP0), DCID=0000000000000000, PKN: 20, STREAM(4)

Table 3.9: Limitation

In trace 3.9, we can see that quic-go retransmits all the data from STREAM(4). Ivy considers that it is not retransmission because it acknowledged previously a frame containing the FIN bit of the stream. Consequently, Ivy considers that it received a new stream frame with an already used ID which is invalid.

After an analysis, we are in the presence of a false positive. In Ivy, we already acknowledged the last STREAM(4) sent by quic-go. At the same time, the quic-go retransmission timer expires. Then, quic-go retransmits some STREAM(4). In the Ivy state, this stream has already been processed. Consequently, we consider the stream finished and acknowledged. Finally, we report an error. This illustrates a limitation of Ivy, it took two seconds to process all quic-go messages which caused retransmission.

The heuristics we used to model certain QUIC specifications have a significant influence on the false positive rate.

These situations are rare but they exist. We could remove the heuristics but it would imply to remove the verification of interesting properties. Therefore, a manual analysis of the trace is also possible to check if we are not in presence of a false positive.

Chapter 4

Results

4.1 Introduction

In this chapter, we report the results obtained by following the experimental protocol of McMillan's paper entitled "Formal Specification and Testing of QUIC" applied to the draft 29 [5] for the server and the client implementations. We compare our findings with the preceding ones to see the evolution of the correctness of the different implementations.

4.2 Methodology

4.2.1 Virtual Machine

We use a virtual CentOS (7.5.1804) machine running on an x86_64 architecture. The environment was limited to 4 cores of 3100 MHz and 8 GB of RAM.

4.2.2 Tested implementations

We check out the quic_29 branch of our Ivy forked version [48]. Seven QUIC implementations are tested: quinn, picoquic, aioquic, mvfst, quiche, quic-go and lsquic. We select implementations written using different programming languages. These implementations use different TLS implementations. Some of them are mature implementations while others are newer. We have implementations intended for production and some used for scientific and feedback purposes. We used the official QUIC Working Group that lists all known implementations of QUIC [49].

Quinn is an implementation written in Rust. One of its creators wanted to show that this language could be used to tackle software challenges that need high performance [50]. It contains 41k lines of code.

picoquic (C) [51] ad23e6c3593bd987dcd8d74fc9f528f2676fedf4 **picotls** [53] 47327f8d032f6bc2093a15c32e666ab6384ecca2	**lsquic** [52] v2.29.4 **boringssl** [54] a2278d4d2cabe73f6663e3299ea7808edfa306b9
quic-go (golang) [55] v0.20.0	**quinn (rust)** [56] 0.7.0
aioquic (python) [57] 0.9.3	**quiche (rust)** [58] 0.7.0
quant (C) [59] 29	**mvfst (facebook/C++)** [60] 36111c1

Table 4.1: Draft 29 tested implementations

Quiche is the QUIC implementation of Cloudflare, a company specialized in website security. The quiche purpose is to be used at an industrial level. The language is also Rust and the number of lines of codes is 58k.

Picoquic is a minimalist implementation of QUIC. It is mainly developed by Christian Huitema [61]. One of the purposes of this implementation is to participate in the development of the QUIC standard by giving feedback [51]. It is written in C and consist of 81k lines of code.

We also use Aioquic, a Python implementation of QUIC and a minimal implementation of TLS 1.3. The intention of aioquic is also to be minimalist. It is written in only 19k lines of code. This implementation was created to provide a common QUIC implementation for all the servers and client libraries in Python [57].

Mvfst, pronounced "Move Fast", is the QUIC implementation created by Facebook. It is already used in the Facebook production networks [62]. The programming language is C++ and the codebase is large: more than 105k line of codes. We tested this implementation only for the server side.

Quant is an implementation that is designed for research purposes only. It uses C as a programming language. The number of lines of codes is low: 18k. However, it does not implement an HTTP/3 binding [59]. It has been used in IoT environment. [63] and was mainly developed by Lars Eggert, the current Chair of the IETF and the IETF's QUIC working group [64].

Quic-go, an implementation written in Go, was also included. It contains 73k lines of code. It has been under development since 2015 [65]. It is used both at a scientific and industrial level. It can be used with Caddy, an automatic HTTPS server [66]

The last tested implementation is lsquic. It is open source, written in C, and it contains 129k lines of code. It has been under development since 2017. We tested this implementation only for the client side. We could not operate with the server side because of an TLS internal error A.1.

4.2.3 Experimental protocol

We launched 100 iterations for each test and implementation. The test suite needed three days to complete. We focused more on the analysis of traces than on statistical analysis. Interpretation of statistical models based on our results could be biased by the distribution of some errors. We chose to answer the question, "how can we improve an implementation?" rather than "does my implementation have a good coverage of the QUIC specification properties ?". Ivy should be able to answer both questions. However, for the second one, a deeper statistical analysis should be done. The way we proceeded makes it impossible for us to extract "probability" and confidence interval of whether an implementation respects a large part of the QUIC specification. It should be possible to analyse statistically the results, but the different parameters of our experimental protocol should be reviewed.

4.3 Servers results

We present the errors found by category of test as presented by Ivy. Multiple client migrations are allowed during these tests. The complete tables summarizing the errors detected per implementation can be found in the appendices. See A.1 for aioquic, A.7 for picoquic, A.13 for quiche, A.3 for quinn, A.11 for quant, A.6 for mvfst and finally A.9 for quic-go. The complete list of failed requirements per implementations is included.

Table 4.2 summarizes the result of all the tests by implementation. It illustrates that some implementations are more advanced than others regarding the specification requirements. Some tests are passed by most implementations while others have a low success rate. We cover in this section the results of the different tests.

In order to prove that Ivy can be used to find errors that we can correct, we fixed the errors of one implementation when it was possible. Our choice for this task is aioquic because it is written in Python, a language that we master.

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
stream	79%	6%	56%	95%	18%	12%	97%
max	85%	3%	47%	39%	27%	21%	96%
reset_stream	29%	7%	61%	100%	24%	5%	98%
connection_close	95%	37%	81%	63%	78%	40%	100%
stop_sending	100%	4%	48%	33%	33%	8%	96%
accept_maxdata	77%	12%	50%	68%	43%	21%	96%
ext_min_ack_delay	80%	10%	38%	100%	26%	6%	98%
unkown	95%	99%	99%	96%	0%	0%	100%
unkown_tp	84%	59%	98%	100%	68%	100%	96%
double_tp_error	100%	0%	100%	100%	100%	3%	100%
tp_error	100%	100%	0%	100%	0%	0%	0%
tp_acticoid_error	100%	0%	0%	0%	0%	100%	0%
no_icid_error	100%	100%	100%	100%	100%	0%	0%
token_error	100%	98%	100%	100%	100%	100%	99%
new_token_error	100%	0%	0%	84%	100%	0%	0%
handshake_done_error	100%	92%	89%	0%	86%	2%	77%
newconnectionid_error	81%	85%	100%	9%	68%	93%	91%
max_limit_error	49%	41%	100%	0%	41%	16%	0%

blocked_error	70%	0%	0%	75%	0%	0%	100%
retirecoid_error	87%	0%	86%	85%	0%	0%	0%
stream_limit_error	100%	63%	99%	98%	99%	10%	0%
newcoid_length_error	84%	0%	2%	81%	0%	0%	91%
newcoid_rtp_error	91%	0%	0%	90%	0%	0%	0%
max_error	0%	90%	100%	0%	0%	0%	0%

Table 4.2: Server - Passed tests ratio

4.3.1 Global tests

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
stream	79%	6%	56%	100%	18%	12%	97%
max	85%	3%	47%	39%	27%	21%	96%
reset_stream	29%	7%	61%	100%	24%	5%	98%
connection_close	95%	37%	81%	63%	78%	40%	100%
stop_sending	100%	4%	48%	33%	33%	8%	96%
accept_maxdata	77%	12%	50%	68%	43%	21%	96%
ext_min_ack_delay	80%	10%	38%	100%	26%	6%	98%

Table 4.3: Server - General tests of the draft - Passed tests ratio

When inspecting Table 4.3, we observe that some implementations respect well the draft for the standard tests. `Quinn` and `quiche` are examples of that. However, we cannot conclude from this table that an implementation is significantly better than another one. For example, `mvfst` failed many tests, but when analysing the errors themselves, we can see that there is only one requirement that fails in most tests.

Observed errors

The first problem found with these tests is a perfect example of ambiguity in the draft. When we perform migration, sometimes `mvfst` returns a `CONNECTION_CLOSE` frame with `APPLICATION_ERROR` (`0x0c`) as an error code. The error message says: `"Migration disable"` and the log states:

```
4472 RecordLayer.cpp:82] Received handshake message Finished
4472 StateMachine-inl.h:41] Transition from ExpectingFinished to AcceptingData
...
633 LogQuicStats.h:60 server onPacketDropped reason=PEER_ADDRESS_CHANGE
633 QuicTransportBase.cpp:1870 onNetworkData Migration disabled
633 EchoHandler.h:53 Socket "error=Invalid migration"
633 LogQuicStats.h:91 server onConnectionClose reason=NONE
```

This error is unexpected. We did not receive the `disable_active_migration` transport parameter [ref. 1]:

src port	dst port	Information
4987	4443	1274 Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 10, CRYPTO, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480956, CRYPTO, ACK, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480958, CRYPTO, ACK, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480961, CRYPTO, ACK, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480963, CRYPTO, ACK, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480964, CRYPTO, ACK, PADDING
4987	4443	1274 Initial, DCID=400000d42cb02248, SCID=0000000000000001, PKN: 27, CRYPTO, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480965, CRYPTO, ACK, PADDING
4443	4987	814 Handshake, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480956, CRYPTO
Extension: quic_transport_parameters (drafts version) (len=89)		
Parameter: original_destination_connection_id (len=8)		
Parameter: initial_max_stream_data_bidi_local (len=4) 66560		
Parameter: initial_max_stream_data_bidi_remote (len=4) 66560		
Parameter: initial_max_stream_data_uni (len=4) 66560		

		Parameter: initial_max_data (len=4) 1048576
		Parameter: initial_max_streams_bidi (len=2) 2048
		Parameter: initial_max_streams_uni (len=2) 2048
		Parameter: max_idle_timeout (len=4) 60000 ms
		Parameter: ack_delay_exponent (len=1)
		Parameter: max_udp_payload_size (len=2) 1500
		Parameter: stateless_reset_token (len=16)
		Parameter: facebook_partial_reliability (len=1)
		Parameter: initial_source_connection_id (len=8)
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480967, CRYPTO, ACK, PADDING
4443	4987	1294 Initial, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480968, ACK, PADDING
4987	4443	150 Handshake, DCID=400000d42cb02248, SCID=0000000000000001, PKN: 7, CRYPTO, ACK, PADDING
4443	4987	89 Handshake, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480957, ACK
4443	4987	71 Protected Payload (KP0), DCID=0000000000000001, PKN: 5480956, DONE
4988	4443	95 Protected Payload (KP0), DCID=400000d42cb02248, PKN: 19, ACK, PADDING
4443	4987	92 Protected Payload (KP0), DCID=0000000000000001, PKN: 5480957, CC
4443	4987	105 Handshake, DCID=0000000000000001, SCID=400000d42cb02248, PKN: 5480958, CC

We observe that `mvfst` throws an error when we start migration just after receiving the HANDSHAKE_DONE. It considers that we break the specification. However, in the draft it is said that we cannot perform a migration before the handshake is confirmed:

> "*The design of* QUIC *relies on endpoints retaining a stable address for the duration of the handshake. An endpoint MUST NOT initiate connection migration before the handshake is confirmed, as defined in section 4.1.2 of [*QUIC-TLS*]."*
>
> Draft 29 of QUIC section 9.

However we can argue with the following requirement in QUIC-TLS for the client view:

> "*In this document, the* TLS *handshake is considered confirmed at the server when the handshake completes. At the client, the handshake is considered confirmed when a* HANDSHAKE_DONE *frame is received."*
>
> Draft 29 of QUIC-TLS section 4.1.2.

From the client point of view, we consider the handshake as confirmed when we received the HANDSHAKE_DONE frame. `Mvfst`, the server, can only know that the client consider the handshake as confirmed when he received the ACK for HANDSHAKE_DONE. `Mvfst` want to be sure that the client has received the HANDSHAKE_DONE before allowing it to migrate. For the server, the handshake is confirmed when the handshake is complete. It happens when the TLS stack has both sent a "Finished" message and verified the peer's "Finished" message. It is the case here (at least for the sending part).

> "*In this document, the* TLS *handshake is considered* ***complete*** *when the* TLS *stack has reported that the handshake is complete. This happens when the* TLS *stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion [also confirmation for server] of the handshake depends on the perspective of the endpoint in question."*
>
> Draft 29 of QUIC-TLS section 4.1.1.

Following the draft, an endpoint cannot migrate before the handshake is confirmed. If a client migrates just after it receives a HANDSHAKE_DONE, the client respects the draft. The server has to accept the migration before it receives the ACK for the HANDSHAKE_DONE.

Migration is the source of most errors detected. `Aioquic` returns us an `PROTOCOL_VIOLATION` error (`0xa`) indicating that the data in the `PATH_CHALLENGE` and `PATH_RESPONSE` frames do not match. Moreover, this problem arises half of the time. It affects the results of many of our tests. Here is a network trace highlighting this problem [ref. 2]:

src port	dst port	Information
4987	4443	1274 Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4987	1322 Handshake, DCID=0000000000000001, SCID=0191cbd30002ee7d, PKN: 1, CRYPTO
4443	4987	1111 Handshake, DCID=0000000000000001, SCID=0191cbd30002ee7d, PKN: 2, CRYPTO
4443	4987	1322 Handshake, DCID=0000000000000001, SCID=0191cbd30002ee7d, PKN: 4, CRYPTO
4443	4987	109 Handshake, DCID=0000000000000001, SCID=0191cbd30002ee7d, PKN: 5, CRYPTO
4987	4443	158 Handshake, DCID=0191cbd30002ee7d, SCID=0000000000000001, PKN: 15, CRYPTO, PADDING
4443	4987	98 Protected Payload (KP0), DCID=0000000000000001, PKN: 6, DONE, NCI
4987	4443	1274 Protected Payload (KP0), DCID=0191cbd30002ee7d, PKN: 8, STREAM(4), PADDING
4443	4987	75 Protected Payload (KP0), DCID=0000000000000001, PKN: 7, ACK
4443	4987	82 Protected Payload (KP0), DCID=0000000000000001, PKN: 8, STREAM(4)
4443	4987	71 Protected Payload (KP0), DCID=0000000000000001, PKN: 9, PING, PADDING
4988	4443	1274 Protected Payload (KP0), DCID=0191cbd30002ee7d, PKN: 12, ACK, ACK, STREAM(8), PADDING
4443	4988	113 Protected Payload (KP0), DCID=0000000000000001, PKN: 10, ACK, DONE, PC, NCI
PATH_CHALLENGE(0a7108f0135b0b18)		
4443	4988	82 Protected Payload (KP0), DCID=0000000000000001, PKN: 11, STREAM(8)
4987	4443	1274 Protected Payload (KP0), DCID=0191cbd30002ee7d, PKN: 32, PR, ACK, PADDING
PATH_RESPONSE(0a7108f0135b0b18)		
4443	4988	106 Protected Payload (KP0), DCID=0000000000000001, PKN: 12, CC
CONNECTION_CLOSE (Transport) Error code: PROTOCOL_VIOLATION		Frame Type: CONNECTION_CLOSE (Transport) (0x000000000000001c) Error code: PROTOCOL_VIOLATION (10) Frame Type: 27 Reason phrase Length: 33 Reason phrase: Response does not match challenge

During a `QUIC migration`, the client decides, after the handshake, to change its address, represented by a pair of IP address and port number. The connection to the server should be able to survive this event thanks to the connection ID. However, for security reasons, the server sends the client, in a frame, a challenge of unpredictable and encrypted data containing 8 bytes. The name of this frame is `PATH_CHALLENGE`. Then, the client should validate this challenge by responding with a `PATH_RESPONSE` frame containing the same data. It allows to validate the new path used by the client and be sure that it is not an attacker that has spoofed the source address of the client.

The problem is that `aioquic` should not throw an error. The data matches. We base our reasoning on the following `QUIC` specification rule that states:

> *"A new address is considered valid when a* `PATH_RESPONSE` *frame is received that contains the data that was sent in a previous* `PATH_CHALLENGE`*. Receipt of an acknowledgment for a packet containing a* `PATH_CHALLENGE` *frame is not adequate validation, since the acknowledgment can be spoofed by a malicious peer.* ***Note that receipt on a different local address does not result in path validation failure****, as it might be a result of a forwarded packet (see Section 9.3.3) or misrouting. It is possible that a valid* `PATH_RESPONSE` *might be received in the future."*
>
> Draft 29 of `QUIC` section 8.5

We fixed this problem considering the first rule only and created a pull request. The problem does not arise any more in the fixed version of `aioquic`:

src port	dst port	Information
4988	4443	1274 Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
4443	4988	1322 Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKN: 1, CRYPTO
4443	4988	1111 Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKN: 2, CRYPTO
4443	4988	1322 Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKN: 4, CRYPTO
4443	4988	109 Handshake, DCID=0000000000000000, SCID=af607353e17bdead, PKN: 5, CRYPTO
4988	4443	173 Handshake, DCID=af607353e17bdead, SCID=0000000000000000, PKN: 12, CRYPTO, ACK, ACK, ACK, PADDING
4443	4988	98 Protected Payload (KP0), DCID=0000000000000000, PKN: 6, DONE, NCI
4987	4443	107 Protected Payload (KP0), DCID=f5d90b5172d826f1, PKN: 13, STREAM(4), PADDING
4443	4987	78 Protected Payload (KP0), DCID=0000000000000000, PKN: 7, PC
4443	4987	75 Protected Payload (KP0), DCID=0000000000000000, PKN: 8, ACK
4443	4987	82 Protected Payload (KP0), DCID=0000000000000000, PKN: 9, STREAM(4)
4443	4987	71 Protected Payload (KP0), DCID=0000000000000000, PKN: 10, PING, PADDING
4988	4443	121 Protected Payload (KP0), DCID=af607353e17bdead, PKN: 15, PR, ACK, STREAM(12), PADDING
4443	4988	104 Protected Payload (KP0), DCID=0000000000000000, PKN: 11, ACK, DONE, NCI
4443	4988	82 Protected Payload (KP0), DCID=0000000000000000, PKN: 12, STREAM(12)
4443	4988	71 Protected Payload (KP0), DCID=0000000000000000, PKN: 13, PING, PADDING
4988	4988	108 Handshake, DCID=af607353e17bdead, SCID=0000000000000000, PKN: 29, ACK, PADDING
4443	4988	71 Protected Payload (KP0), DCID=0000000000000000, PKN: 14, PING, PADDING
4443	4988	71 Protected Payload (KP0), DCID=0000000000000000, PKN: 15, PING, PADDING
4988	4443	142 Protected Payload (KP0), DCID=f5d90b5172d826f1, PKN: 32, ACK, STREAM(16), ACK, ACK, STREAM(28), PADDING
4443	4988	91 Protected Payload (KP0), DCID=0000000000000000, PKN: 16, ACK, PING, STREAM(4)
..		

This error is probably due to a change in the draft. Before draft 20, we could only respond to the `PATH_RESPONSE` frame from the network path where the `PATH_CHALLENGE` was sent. Since this feature of `aioquic` has not changed for a long time, this could explain the error.

Another problem of the migration that arises in most implementations is about the following requirement that indicates that the server did not send packets to an address/endpoint from which the highest packet number of non-probing packets has been received:

```
require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst);
```

The following `aioquic` trace is an example [ref. 3]:

src port	dst port	Information
4988	4443	1274 Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4988	1322 Handshake, DCID=0000000000000001, SCID=21500f522a60b7b2, PKN: 1, CRYPTO
4443	4988	1111 Handshake, DCID=0000000000000001, SCID=21500f522a60b7b2, PKN: 2, CRYPTO
4443	4988	1322 Handshake, DCID=0000000000000001, SCID=21500f522a60b7b2, PKN: 4, CRYPTO
4443	4988	109 Handshake, DCID=0000000000000001, SCID=21500f522a60b7b2, PKN: 5, CRYPTO
4988	4443	163 Handshake, DCID=21500f522a60b7b2, SCID=0000000000000001, PKN: 6, CRYPTO, ACK, PADDING
4443	4988	98 Protected Payload (KP0), DCID=0000000000000001, PKN: 6, DONE, NCI
4987	4443	112 Protected Payload (KP0), DCID=0f2380bbb53779ee, PKN: 1, STREAM(4), ACK, PADDING
4443	4987	78 Protected Payload (KP0), DCID=0000000000000001, PKN: 7, PC

When we start the migration, we send a short header packet with 1 as sequence number, which is not erroneous since we change the packet number space. Using the preceding rule, `aioquic` should have sent the packet to the first address, as we received the highest-numbered packet. One hypothesis is that `aioquic` is confused as we migrate, but this is caused by the ambiguity of the draft.

In `Ivy`, we consider the following `QUIC` specification rule:

> "*An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.*"
>
> Draft 29 of QUIC section 9.3.

According to this rule, we should send the packet to the highest-numbered non-probing packets. As we did not have an indication about the encryption level to consider, we used the highest-numbered non-probing packet among all possible ones. Since migration is allowed only after the handshake completed, we should probably consider only the encryption level for the 0-RTT Protected or short header packets. It is not written explicitly.

However, in the same paragraph, we can find the following indication:

> "*Receiving a packet from a new peer address containing a non-probing frame indicates that the peer has migrated to that address. In response to such a packet, an endpoint MUST start sending subsequent packets to the new peer address and MUST initiate path validation (Section 8.2) to verify the peer's ownership of the unvalidated address..*"
>
> Draft 29 of QUIC section 9.3.

If we use this rule, then we must send responses of non-probing frames to the new peer address. Now, the question is the following: why do these rules, in the same paragraph, seem to contradict one another when we consider them in pairs ? What should we do if the highest-numbered non-probing packet is received on the old peer address ?

This error is present in most tested implementations. One of the main errors was detected in `picoquic`, `quant` and `quinn`.

Finally, the last major error detected related to migration is about pending `PATH_CHALLENGE` frames that we received twice with the data. It is represented by:

```
require ~path_challenge_pending(dcid,f.data)
```

We present a trace that we observed locally using `quinn` [ref. 4]. This error was also frequently observed with `quant`:

src port	dst port	time	Information
4987	4443		1274 Initial, DCID=0000000000000000, SCID=0000000000000001, PKN:0, CRYPTO, PADDING
4443	4987		1242 Handshake, DCID=0000000000000001, SCID=e09f058956664da8, PKN:0, CRYPTO, PADDING
4443	4987		96 Protected (KP0) , DCID=0000000000000001, PKN:0, NCI
4987	4443		158 Handshake, DCID=e09f058956664da8, SCID=0000000000000001, PKN:0, CRYPTO, PADDING
4443	4987		71 Protected (KP0) , DCID=0000000000000001, PKN:1, DONE, PADDING
4988	4443		1274 Protected (KP0) , DCID=e09f058956664da8, PKN:0, ACK, ACK, PADDING
4443	4987		1242 Protected (KP0) , DCID=0000000000000001, PKN:2, PC, PADDING
PATH_CHALLENGE (00fe42f00ca1b5a9), PADDING			
4988	4443		1274 Protected (KP0) , DCID=e09f058956664da8, PKN:16, PR, PADDING
4443	4988	37.38	1242 Protected (KP0) , DCID=0000000000000001, PKN:3, ACK, PC, PADDING
ACK, PATH_CHALLENGE (03a32cf4a177000f), PADDING			
4443	4988	40.58	1242 Protected (KP0) , DCID=0000000000000001, PKN:4, PING, ACK, PC, PADDING
PING, ACK, PATH_CHALLENGE (03a32cf4a177000f), PADDING			

We can see that the packet is retransmitted, as the payload of the packet is the same with only the packet number changing. Moreover, the time between the two packets seems to confirm this idea. This is not wanted and this is due to the `Ivy` computations that are responsible for the slow generation of packets. We explore solutions to this problem in the section 5.4.

This is in contradiction with the section "13.3 Retransmission of Information" of the draft 29. It is written that:

> "*A liveness or path validation check using* `PATH_CHALLENGE` *frames is sent periodically until a matching* `PATH_RESPONSE` *frame is received or until there is no remaining need for liveness or path validation checking.* `PATH_CHALLENGE` *frames* ***include a different payload each time they are sent.***"
>
> Draft 29 of `QUIC` section 13.3

However, there is this property from the retransmission section :

> "*New frames and packets are used to carry information that is determined to have been lost.* ***In general,*** *information is sent again when a packet containing that information is determined to be lost and sending ceases when a packet containing that information is acknowledged.*"
>
> Draft 29 of `QUIC` section 13.3

As a conclusion, when the implementation noticed that the `PATH_CHALLENGE` was lost, it tried to retransmit the frame. This is an error. Either this requirement is not implemented or this is a coding error.

A problem in acknowledgement management was also found for `quant`, `aioquic`, `quinn` and `quiche`.

```
require ~ _generating  ∧ ~ queued_non_ack(scid)
        -> ack_credit(scid)> 0; # [5]
```

Let us explain this requirement by dividing it by parts :

- `~ queued_non_ack(scid)`: `queued_non_ack(C)` is a relation that indicates that one or more of queued frames at aid `C` is not an `ACK` frame. With the negation rule, we have that we should not have an `ACK` frame for `C`.
- `ack_credit(scid)> 0`: The relation `ack_credit(scid)` is the number of ack-eliciting packets minus the number of non-ack-eliciting packets which is any packet containing other frames than `ACK`, `PADDING`, `CONNECTION_CLOSE`.

The final relation indicates that the number of non-ack-eliciting packets must not be greater than the number of ack-eliciting packets.

The specification states that:

> *"Since packets containing only ACK frames are not congestion controlled, an endpoint MUST NOT send more than one such packet in response to receiving an ack-eliciting packet."*
>
> Draft 29 of QUIC section 13.2.1

This network trace from quant shows that we receive an ACK for the same packet twice in a short time [ref. 5]:

src port	dst port	Information
4988	4443	1274 Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4988	1242 Initial, DCID=0000000000000001, SCID=5015dece, PKN: 0, ACK, PADDING, CRYPTO
4443	4988	1294 Handshake, DCID=0000000000000001, SCID=5015dece, PKN: 0, PADDING, CRYPTO
4443	4988	354 Handshake, DCID=0000000000000001, SCID=5015dece, PKN: 1, PADDING, CRYPTO
4988	4443	154 Handshake, DCID=5015dece, SCID=0000000000000001, PKN: 3, CRYPTO, PADDING
4443	4988	105 Protected Payload (KP0), DCID=0000000000000001, PKN: 0, DONE, NT
4988	4443	1274 Protected Payload (KP0), DCID=5015dece, PKN: 16, ACK, PADDING
4443	4988	74 Protected Payload (KP0), DCID=0000000000000001, PKN: 1, ACK
4988	4443	1274 Protected Payload (KP0), DCID=5015dece, PKN: 32, ACK, PADDING
4443	4988	73 Protected Payload (KP0), DCID=0000000000000001, PKN: 2, ACK
4988	4443	1274 Protected Payload (KP0), DCID=5015dece, PKN: 33, ACK, PADDING
4443	4988	73 Protected Payload (KP0), DCID=0000000000000001, PKN: 3, ACK
ACK (LA: 33, ACK Delay: 37, ACK Range Count: 0, First ACK range:0)		

We see that quant acknowledges an ACK frame which does not respect the property.

Besides, we highlighted that for the connection_close test, we have many errors for quant. Among them is an error of invalid state:

```
require conn_draining(scid)-> ~draining_pkt_sent(scid)&
                              queued_close(scid);
```

This rule indicates that if the sender is in the draining state, then we consider it as a "draining packet". An endpoint enters the draining state if this endpoint sends a CONNECTION_CLOSE frame and this frame is acknowledged. This implies no retransmission of these packets to avoid an infinite loop of CONNECTION_CLOSE exchanges. In this case, we can see that we have retransmissions of these packets [ref. 6]:

src port	dst port	Information
4988	4443	1274 Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1242 Initial, DCID=0000000000000000, SCID=d2676493, PKN: 0, ACK, PADDING, CRYPTO
4443	4988	1294 Handshake, DCID=0000000000000000, SCID=d2676493, PKN: 0, PADDING, CRYPTO
4443	4988	351 Handshake, DCID=0000000000000000, SCID=d2676493, PKN: 1, PADDING, CRYPTO
4988	4443	115 Handshake, DCID=d2676493, SCID=0000000000000000, PKN: 18, ACK, CC, PADDING
4443	4988	89 Handshake, DCID=0000000000000000, SCID=d2676493, PKN: 2, ACK, CC
4988	4443	104 Handshake, DCID=d2676493, SCID=0000000000000000, PKN: 31, ACK, PADDING
4443	4988	91 Handshake, DCID=0000000000000000, SCID=d2676493, PKN: 3, ACK, CC

This corresponds to the following QUIC specification rule:

> *"After receiving a* CONNECTION_CLOSE *frame, endpoints enter the draining state. An endpoint that receives a* CONNECTION_CLOSE *frame MAY send a single packet containing a* CONNECTION_CLOSE *frame before entering the draining state, using a* CONNECTION_CLOSE *frame and a* NO_ERROR *code if appropriate. An endpoint MUST NOT send further packets, which could result in a constant exchange of* CONNECTION_CLOSE *frames until the closing period on either peer ended"*
>
> Draft 29 of QUIC section 10.3.

Finally we also have implementations like mvfst, where we struggled to complete the handshake. This is because it implements the TLS handshake with the "Hello Retry Request" message. However, it seems that after some trials, it reverts back to the traditional "Server Hello" message. This could be why sometimes the handshake could be done and sometimes it could not [ref. 7]:

src port	dst port	time	Information
4987	4443	125.357	1274 Initial, DCID=0000000000000001, SCID=0000000000000000, PKN:0
4443	4987	125.358	1294 Initial, DCID=0000000000000000, SCID=40000087630fcf40, PKN:12755779
4443	4987	125.471	1294 Initial, DCID=0000000000000000, SCID=40000087630fcf40, PKN:12755781
4443	4987	125.682	1294 Initial, DCID=0000000000000000, SCID=40000087630fcf40, PKN:12755783
TLSv1.3 Record Layer: Handshake Protocol: Hello Retry Request			
4443	4987	125.683	1294 Initial, DCID=0000000000000000, SCID=40000087630fcf40, PKN:12755784
4987	4443	126.090	1274 Initial, DCID=40000087630fcf40, SCID=0000000000000000, PKN:16
4443	4987	126.091	1294 Initial, DCID=0000000000000000, SCID=40000087630fcf40, PKN:12755785
TLSv1.3 Record Layer: Handshake Protocol: Server Hello			

The cost of implementing this feature would be to update the serializer/deserializer of the TLS part and the associated Ivy model. We would also need to update the PicoTLS API interface with Ivy.

4.3.2 Unknown types tests

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
unkown	95%	99%	99%	96%	0%	0%	100%
unkown_tp	84%	59%	98%	100%	68%	100%	96%

Table 4.4: Server - Management of unknown frames and transport parameter

For the management of UNKNOWN frames, we should receive a FRAME_ENCODING_ERROR to pass the test. Usually, either an implementation reacts well, either it does not.

Quant for example, does not send an error when receiving the frame and closes the connection [ref. 8]

```
[0m[43m [0m [35merr_close[30m [34mconn.c:1675 [0mignoring new err 0x7; existing
    err is 0x7 (unknown 0x40 frame at pos 10)
[0m[1m6.120[0m [46m [0m [35mlog_pkt[30m [34mpkt.c:125 [0m[1m[34mRX[0m
    from=127.0.0.1:4987 len=79 0x43=[34mShort [0mkyph=0 spin=0 dcid=0:e741a4b2
    nr=[34m51[0m
[0m[43m [0m [35merr_close[30m [34mconn.c:1675 [0mignoring new err 0x7; existing
    err is 0x7 (unknown 0x40 frame at pos 10)
[0m[0m6.318[0m [43m [0m [35mq_close[30m [34mquic.c:751 [0mclosing serv conn
    0:e741a4b2 on 127.0.0.1:4443 w/err [0m0x0[0m
```

Others return the wrong error code. For example `aioquic` returns a `PROTOCOL_VIOLATION` error [ref. 9]:

src port	dst port	Information
4988	4443	1322, Initial, DCID=0000000000000002, SCID=0000000000000001, PKT: 7, CRYPTO PADDING
4443	4988	1111, Handshake, DCID=0000000000000001, SCID=2b5a88d1fc6c5918, PKT: 0, ACK ACK
4443	4988	158, Handshake, DCID=0000000000000001, SCID=2b5a88d1fc6c5918, PKT: 2, CRYPTO
4988	4443	98, Protected Payload (KP0), DCID=2b5a88d1fc6c5918, PKT: 6, CRYPTO PADDING
4443	4988	89, Protected Payload (KP0), DCID=0000000000000001, PKT: 3, DONE NCI
4987	4443	92, Protected Payload (KP0), DCID=2b5a88d1fc6c5918, PKT: 1, Unknown PADDING
4443	4988	1274, Initial, DCID=0000000000000001, SCID=0, PKT: 4, CC
		CONNECTION_CLOSE (Transport) Error code: PROTOCOL_VIOLATION
		Frame Type: CONNECTION_CLOSE (Transport) (0x000000000000001c)
		Error code: PROTOCOL_VIOLATION (10)
		Frame Type: 66
		Reason phrase Length: 18
		Reason phrase: Unknown frame type

It may also happen that an implementation detects the unknown frame. A `FRAME_ENCODING_ERROR` is usually sent. However, sometime no error is returned even through the frame was detected. An example with `quinn` is presented in this trace [ref. 10]:

src port	dst port	Information
4988	4443	1274, Initial, DCID=0000000000000002, SCID=0000000000000001, PKT: 7, CRYPTO
4443	4988	860, Handshake, DCID=0000000000000001, SCID=f4532ffbdf98e945, PKT: 0, CRYPTO
4443	4988	96, Protected Payload (KP0), DCID=0000000000000001, PKT: 0, NCI
4988	4443	108, Handshake, DCID=e170588d10dd0d09, SCID=0000000000000001, PKT: 6, ACK, PADDING
4988	4443	158, Protected Payload (KP0), DCID=0000000000000001, PKT: 15, CRYPTO
4443	4988	71, Protected Payload (KP0), DCID=0000000000000001, PKT: 1, DONE, PADDING
4987	4443	92, Protected Payload (KP0), DCID=f4532ffbdf98e945, PKT: 14, ACK, PADDING
4443	4988	1242, Protected Payload (KP0), DCID=0000000000000001, PKT: 2, PC, PADDING
4988	4443	131, Protected Payload (KP0), DCID=e170588d10dd0d09, PKT: 16, PR, ACK, Unknown
4987	4443	124, Protected Payload (KP0), DCID=e170588d10dd0d09, PKT: 18, ACK, STREAM(12), Unknown, PADDING
4988	4443	168, Protected Payload (KP0), DCID=e170588d10dd0d09, PKT: 32, ACK, STREAM(12), STREAM(8), STREAM(4), Unknown, PADDING

The logs confirm that the unknown frame is received:

```
[2mMay 19 23:09:08.816 [0m [31mERROR [0m server: connection failed: received a
    frame that was badly formatted in <unknown 42>: invalid frame ID
```

For the management of `UNKNOWN` transport parameters, we do not migrate to observe more clearly the result of this test. This feature is correctly implemented by most implementations.

4.3.3 Errors handling

Transport parameter errors

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
double_tp_error	100%	0%	100%	100%	0%	3%	100%
tp_error	100%	100%	0%	100%	0%	0%	0%
tp_acticoid_error	100%	0%	0%	0%	0%	100%	0%
no_icid_error	100%	100%	100%	100%	0%	0%	0%

Table 4.5: Server - Transport parameter errors tests

For the test `quic_server_tests_double_tp_error`, the `QUIC` specification is ambiguous on whether we should throw an error or not complete the handshake (3.4.3). The results seem to confirm that the rule is ambiguous. Either the implementations did not implement this property and continued the execution, either the succeeding implementations interpreted it according to one of the two possible valid endings.

Most implementations do not check the value of the `active_connection_id_limit` and `ack_delay_exponent` transport parameters.

For the same test, `aioquic` crashed unexpectedly. In the logs, we see that `aioquic` detects the correct error and closes the connection:

```
2021-04-29 13:32:58,066 DEBUG asyncio Using selector: EpollSelector
2021-04-29 13:33:02,939 DEBUG quic [0000000000000002] Network path ('127.0.0.1',
    4987) discovered
2021-04-29 13:33:02,942 DEBUG quic [0000000000000002]
    QuicConnectionState.FIRSTFLIGHT -> QuicConnectionState.CONNECTED
2021-04-29 13:33:02,957 DEBUG quic [0000000000000002] TLS
    State.SERVER_EXPECT_CLIENT_HELLO -> State.SERVER_EXPECT_FINISHED
2021-04-29 13:33:02,958 WARNING quic [0000000000000002] Error: 8, reason:
    active_connection_id_limit must be no less than 2, frame_type:
    QuicFrameType.CRYPTO
2021-04-29 13:33:02,958 DEBUG quic [0000000000000002]
    QuicConnectionState.CONNECTED -> QuicConnectionState.CLOSING
2021-04-29 13:33:04,460 DEBUG quic [0000000000000002] Discarding epoch
    Epoch.INITIAL
2021-04-29 13:33:04,461 DEBUG quic [0000000000000002] Discarding epoch
    Epoch.HANDSHAKE
2021-04-29 13:33:04,461 DEBUG quic [0000000000000002] Discarding epoch
    Epoch.ONE_RTT
2021-04-29 13:33:04,461 DEBUG quic [0000000000000002] QuicConnectionState.CLOSING
    -> QuicConnectionState.TERMINATED
```

We observe that the wrong packet type is sent and, consequently, the wrong encryption key, which implies that the client is not correctly informed of the error [ref. 11]:

src port	dst port	Information
4988	4443	1322, Initial, DCID=0000000000000002, SCID=0000000000000001, PKT: 11, CRYPTO PADDING
4443	4987	80, Protected Payload (KP0), DCID=0000000000000001, Protected Payload (KP0) (Undecryptable)

For the other tests about transport parameters, `aioquic` reports no error and continues the execution as if nothing happened.

`Quiche` chose to do nothing when an endpoint initiates connection with an invalid transport parameter value. We usually get a timeout, meaning, in this case, a local error [ref. 12]:

```
Finished dev [unoptimized + debuginfo] target(s) in 0.12s
Running `tools/apps/target/debug/quiche-server --cert tools/apps/src/bin/cert.crt
    --key tools/apps/src/bin/cert.key --no-retry --dump-packets /results/dump.txt
    --listen '127.0.0.1:4443'`
[2021-04-18T18:05:39.038882591Z ERROR quiche_server]
    d21987e22b078c1d8b4a32d0308c321f recv failed: InvalidTransportParam
```

But according to the draft,

> *"An endpoint MUST treat receipt of a transport parameter with an invalid value as a connection error of type* TRANSPORT_PARAMETER_ERROR.*"*
>
> Draft 29 of QUIC section 7.4.

Quiche does not conform to the draft concerning validation of the transport parameters. This can be problematic for the application if it needs this information to fall back towards TCP, for example.

Quant does not conform for most transport parameters tested. For quic_server_test_tp_error and quic_server_test_no_icid, we should get an error but instead quant issues a runtime error and crashes with the following error message [ref. 13]:

```
[0m[41m[0m[35m err_close[30m  [34m conn.c:1691 [0mack_delay_exponent 16040 invalid
/quic/quant/lib/src/tls.c:1465:9: runtime error: null pointer passed as argument
    7, which is declared to never be null
```

However in the latest version, in the master branch, these problems have been fixed:

	quant 29	quant master
double_tp_error	3%	100%
tp_error	0%	100%
tp_acticoid_error	100%	100%
no_icid_error	0%	100%

Table 4.6: Quant transport parameter: before/after

With quic_server_test_double_tp_error, mvfst ignores it and continue the execution. This is bad since we do not know which transport parameter is used.

Violation of the draft

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
token_error	100%	98%	100%	100%	100%	100%	99%
new_token_error	100%	0%	0%	84%	100%	0%	0%
handshake_done_error	100%	92%	89%	0%	86%	2%	77%
newconnectionid_error	81%	85%	100%	9%	68%	93%	91%
max_limit_error	49%	41%	100%	0%	41%	16%	0%

Table 4.7: Server - Protocol violation errors tests

Quic_server_test_token_error is the only test to pass nearly all the time. There are still some errors found. For example, with mvfst, we get an INTERNAL_ERROR in a CONNECTION_CLOSE frame [ref. 14]:

```
I0418 10:41:27.839520 15506 EchoServer.h:90] Echo server started at:
    127.0.0.1:4443
I0418 10:41:46.003042 15510 EchoHandler.h:48] Socket closed
*** Aborted at 1618742506 (Unix time, try 'date -d @1618742506') ***
```

```
*** Signal 15 (SIGTERM) (0x3c8f) received by PID 15506 (pthread TID
    0x7f30a8656ec0) (linux TID 15506) (maybe from PID 15503, UID 0) (code: 0),
    stack trace: ***
(error retrieving stack trace)
```

As a reminder, the draft says that:

> "*Token Length: A variable-length integer specifying the length of the Token field, in bytes. This value is zero if no token is present. Initial packets sent by the server MUST set the Token Length field to zero; clients that receive an Initial packet with a non-zero The token Length field MUST either discard the packet or generate a connection error of type* PROTOCOL_VIOLATION."
>
> Draft 29 of QUIC section 17.2.2.

Only two tests out of 100 generated this result.

Other implementations such as quant do not check if a frame can be sent by the server. Some implementations such as quic-go even return a wrong error for the HANDSHAKE_DONE test. The specification states:

> "*A* HANDSHAKE_DONE *frame can only be sent by the server. Servers MUST NOT send a* HANDSHAKE_DONE *frame before completing the handshake. A server MUST treat receipt of a* HANDSHAKE_DONE *frame as a connection error of type* PROTOCOL_VIOLATION"
>
> Draft 29 of QUIC section 19.20.

Moreover the error message indicates that the feature is not well implemented. The message says that we cannot send this frame with the "Handshake" encryption, which is true, but in this case, it was intentional. The most important is that it does not detect the fact that the frame was sent by a client [ref. 15]:

src port	dst port	Information
4988	4443	1274 Initial, DCID=0000000000000000, SCID=0000000000000001, PKN: 0, CRYPTO, PADDING
4443	4988	1294 Handshake, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 0, CRYPTO
4443	4988	465 Handshake, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 1, CRYPTO
4443	4988	213 Initial, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 1, CRYPTO
4443	4988	213 Initial, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 2, CRYPTO
4443	4988	1118 Handshake, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 2, CRYPTO
4443	4988	465 Handshake, DCID=0000000000000001, SCID=6d0df88cb463f2a1, PKN: 3, CRYPTO
4988	4443	109 Handshake, DCID=6d0df88cb463f2a1, SCID=0000000000000001, PKN: 0, DONE, DONE, DONE, DONE, DONE, PADDING
4443	4988	240 Protected Payload (KP0), DCID=0000000000000001, PKN: 0, CC
		CONNECTION_CLOSE (Transport) Error code: FRAME_ENCODING_ERROR
		Error code: FRAME_ENCODING_ERROR (7)
		Frame Type: 30
		Reason phrase Length: 60
		Reason phrase: HandshakeDoneFrame not allowed at encryption level Handshake

Quant encounters the same problem with the NEW_TOKEN frame and returns a TRANSPORT_PARAMETER_ERROR.

Picoquic and mvfst do nothing, we present the logs of picoquic [ref. 16]:

```
0000000000000002: Receiving packet type: 6 (1rtt protected), S0, Q1,
0000000000000002:     <ecfdcfcee9220860>, Seq: 36 (36), Phi: 0,
0000000000000002:     Decrypted 54 bytes
0000000000000002:     ACK (nb=0), 10
0000000000000002:     NEW TOKEN[2]: 0xda29
0000000000000002:     NEW TOKEN[2]: 0x891d
0000000000000002:     Stream 48, offset 0, length 16, fin = 1: 474554202f696e64...
0000000000000002:     ACK (nb=0), 11
0000000000000002:     padding, 16 bytes
0000000000000002: Server CB, Stream: 48, 16 bytes, fin=1
0000000000000002: Server CB, Stream: 48, Processing command: GET /index.html
```

Another point is for max_limit_error where we generate STREAM frames with stream ID exceeding the maximum value allowed by the transport parameters and by the MAX_STREAMS frame. We also received the STREAM_STATE_ERROR instead of a STREAM_LIMIT_ERROR. This is normal considering to the following rule:

> " *An endpoint MUST terminate the connection with error* STREAM_STATE_ERROR *if it receives a* STREAM *frame for a* ***locally-initiated*** *stream that has not yet been created, or for a send-only stream.*"
>
> Draft 29 of QUIC section 19.8.

This property cannot be tested with Ivy since we need local inspection of the implementation state.

We also faced a concurrency/thread issue with quinn [ref. 17]:

```
Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/examples/server /QUIC-Ivy/doc/examples/quic/ --listen
         '127.0.0.1:4443' --keylog`
listening on 127.0.0.1:4443
thread 'tokio-runtime-worker' panicked at 'called `Option::unwrap()` on a `None`
    value', quinn-proto/src/cid_queue.rs:99:34
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
thread 'tokio-runtime-worker' panicked at 'called `Result::unwrap()` on an `Err`
    value: "PoisonError { inner: .. }"', /quic/quinn/quinn/src/mutex.rs:138:42
stack backtrace:
...
  82:     0x7ff247e4a6db - start_thread
  83:     0x7ff24795b71f - __clone
  84:                0x0 - <unknown>
thread panicked while panicking. aborting.
```

This error happens with quic_server_test_newconnectionid_error, where we generate NEW_CONNECTION_ID with random connection ID and random "Retire to Prior" field. Here is another error with quant for the same test [ref. 18]:

```
[45m[37m[1m5.771    cid_retire cid.c:178 ABORT: assertion failed:
        id->available == false && id->retired == false
can only retire active cid [errno 9 = Bad file descriptor][0m
        /usr/lib/x86_64-linux-gnu/libasan.so.4(+0x558c0) [0x7fada9eda8c0]
    [34mutil_die at /quic/quant/lib/deps/warpcore/lib/src/util.c:398
[0m [34mcid_retire at /quic/quant/lib/src/cid.c:179
[0m [34muse_next_dcid at /quic/quant/lib/src/conn.c:222
[0m [34mdo_conn_mgmt at /quic/quant/lib/src/conn.c:525
[0m [34mtx_ack at /quic/quant/lib/src/conn.c:644
[0m [34mack_alarm at /quic/quant/lib/src/conn.c:1800
```

```
[0m [34mloop_run at /quic/quant/lib/src/loop.c:83
[0m [34mq_ready at /quic/quant/lib/src/quic.c:1010
[0m [34mmain at /quic/quant/bin/server.c:405
[0m /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xe7) [0x7fada84bcbf7]
 [34m_start at ??:?
```

Invalid frames

For the tests in 4.8, no migration was done to highlight the requirement tested.

	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
blocked_error	70%	0%	0%	75%	0%	0%	100%
retirecoid_error	87%	0%	86%	85%	0%	0%	0%
stream_limit_error	100%	63%	99%	98%	99%	10%	98%
newcoid_length_error	84%	0%	2%	81%	0%	0%	0%
newcoid_rtp_error	91%	0%	0%	76%	0%	0%	0%
max_error	0%	90%	100%	0%	0%	0%	0%

Table 4.8: Server - Invalid fields frames errors tests

Surprisingly, from the red boxes present in the table, we can conclude that most implementations do not check the validity of the frame fields. `Quinn` is the implementation with the least errors, reporting, most of the time, the good errors. The errors we get are usually produced before the specific requirements can be tested.

We observed that `quinn` and `quiche` return the wrong error for invalid value of the "maximum streams" of the `MAX_STREAM` frames with a `FRAME_ENCODING_ERROR` instead of `STREAM_LIMIT_ERROR`.

`Aioquic` and `quant` generally do not check the validity of the field content or return the wrong errors.

For the tests with invalid fields in `NEW_CONNECTION_ID` frames, `picoquic` and `mvfst` return `PROTOCOL_VIOLATION` error codes instead of `FRAME_ENCODING_ERROR`. For the same tests, `quiche` does not perform any check and no error is returned. For `mvfst` the reason in the textual information indicates that it detected the problem but the wrong error code is still returned.

`Mvfst`, for the test with invalid `RETIRE_CONNECTION_ID` frames, throws five times an `INTERNAL_ERROR`:

```
I0524 08:16:06.915892 29336 LogQuicStats.h:144] server onWrite size=61
I0524 08:16:06.915917 29336 QuicTransportFunctions.cpp:1063] Server sent close
    packetNum=6539041 in space=InitialSpace client CID=0000000000000000 server
    CID=4000008745bd0358 peer address=127.0.0.1:4987
I0524 08:16:06.915951 29336 LogQuicStats.h:144] server onWrite size=62
*** Aborted at 1621836967 (Unix time, try 'date -d @1621836967') ***
*** Signal 15 (SIGTERM) (0x3e800007281) received by PID 29329 (pthread TID
    0x7fc8d59e6ec0) (linux TID 29329) (maybe from PID 29313, UID 1000) (code: 0),
    stack trace: ***
(error retrieving stack trace)
```

We discovered that this is not linked to the frame since this error happens just after the handshake.

src port	dst port	Information
4987	4443	1232, Initial, DCID=0000000000000001, SCID=0000000000000000, PKT: 7, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539021, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539023, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539025, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539026, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539028, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539029, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539031, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539032, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539034, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539035, CRYPTO
4987	4443	1232, Initial, DCID=4000008745bd0358, SCID=0000000000000000, PKT: 15, CRYPTO
4443	4987	52, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539036, ACK
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539038, CRYPTO
4443	4987	1252, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539039, CRYPTO
4443	4987	54, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539040, ACK
4443	4987	46, Protected Payload (KP0), DCID=0000000000000000, PKT: 0, Protected Payload (KP0) (Undecryptable)
4443	4987	61, Handshake, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539021, CC(INTERNAL_ERROR)
4443	4987	62, Initial, DCID=0000000000000000, SCID=4000008745bd0358, PKT: 6539041, CC(INTERNAL_ERROR)

For the tests with nonconforming `STREAMS_BLOCKED`, `mvfst` and `picoquic` do not perform any check of the invalid field since no error is received.

Finally, all implementations, except `quant`, check and return the right error code when we generate `STREAM` frames with an invalid offset. Sometimes `mvfst` does not report any error and the connection is locally closed:

```
I0527 09:49:42.560974 23422 ServerStateMachine.cpp:967] Server received stream
    data for stream=0, offset=4611686018427387903 len=0 fin=0 client
    CID=0000000000000001 server CID=4000000724477788 peer address=127.0.0.1:4987
I0527 09:49:42.561013 23422 LogQuicStats.h:97] server onNewQuicStream
I0527 09:49:42.561041 23422 QuicStreamFunctions.cpp:67] Empty stream without EOF
```

Sometime it does report the correct error:

```
I0527 09:49:55.339404 23457 ServerStateMachine.cpp:967] Server received stream
    data for stream=4, offset=4611686018427387903 len=0 fin=1 client
    CID=0000000000000001 server CID=400000185893593b peer address=127.0.0.1:4987
I0527 09:49:55.339445 23457 LogQuicStats.h:97] server onNewQuicStream
I0527 09:49:55.339473 23457 StreamReceiveHandlers.cpp:36] Open: Received data
    with fin stream=4 client CID=0000000000000001 server CID=400000185893593b
    peer address=127.0.0.1:4987
I0527 09:49:55.339834 23457 QuicTransportBase.cpp:1870] onNetworkData Stream flow
    control violation on stream 4 client CID=0000000000000001 server
    CID=400000185893593b peer address=127.0.0.1:4987
```

After analysis, the result seems to depend on the stream ID of the invalid `STREAM` frame. If it is 0, then no error is reported. Otherwise, the good error is reported.

For the same test `quant` sometimes returns the expected error but for the wrong reason. It returns the error because the offset is greater than the announced `initial_max_stream_data` transport parameter.

4.4 Client results

We present in this section the client results. We decided to remove `mvfst` from the tested implementation since we are unable to decrypt correctly for the 1-RTT encryption level every time. We believe that this is linked to the fact that `mvfst` uses zero-length source connection ID which is unadapted to the current `Ivy` model and we cannot configure `mvfst` not to use this feature.

Furthermore, we note that we were unable to configure some implementations to set the source and destination connection ID to fewer than 16 bytes. Most of the time, only one of them could be modified. We therefore removed these tests from the set, which explains why we do not have 100 iterations for all the tests.

Besides, we encountered more issues with retransmission timers, since the implementation usually sends burst of requests. This introduces many computations in `Ivy` that delay the packet generation.

The complete tables summarizing the errors found per implementation can be found in the appendices. See A.2 for `aioquic`, A.8 for `picoquic`, A.14 for `quiche`, A.5 for `quinn`, A.11 for `quant`; A.10 for `quic-go` and finally A.15 for `lsquic`.

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
stream	99%	51%	100%	97%	85%	52%	92%
max	100%	15%	100%	98%	85%	34%	100%
accept_maxdata	100%	93%	100%	97%	95%	82%	83%
ext_min_ack_delay	100%	40%	99%	100%	100%	100%	95%
unkown	100%	96%	99%	0%	0%	100%	0%
tp_unkown	100%	34%	99%	99%	100%	99%	96%
double_tp_error	0%	100%	100%	0%	0%	0%	0%
tp_error	0%	0%	100%	0%	0%	0%	0%
tp_acticoid_error	0%	0%	0%	0%	100%	0%	0%
no_ocid	0%	100%	100%	0%	0%	0%	0%
tp_prefadd_error	0%	100%	0%	0%	0%	0%	0%
blocked_error	99%	0%	97%	0%	0%	91%	98%
retirecoid_error	99%	99%	100%	0%	0%	0%	98%
new_token_error	98%	94%	96%	1%	0%	87%	100%
limit_max_error	0%	88%	0%	0%	81%	0%	0%

Table 4.9: Client - Global test

4.4.1 Global tests

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
stream	99%	51%	100%	97%	85%	52%	92%
max	100%	15%	100%	98%	85%	34%	100%
accept_maxdata	100%	93%	100%	97%	95%	82%	83%
ext_min_ack_delay	100%	40%	99%	100%	100%	100%	95%

Table 4.10: Client - Global test

When we closely examine the errors, we see that they are rather similar to the ones with the server tests. This is normal as usual, the tests are almost the same but adapted for the client. All errors linked to migration are not present so it allows highlighting errors without migration. Generally, we also have fewer error of acknowledgement. Good examples are `aioquic` and `quant` where the rate for this error is much lower than for the server.

Then, we can directly see that there are fewer errors in general. This is probably because the migration was not activated. Not all the subsets of requirements directly linked to that part are verified. As for the implementations, no migration implies an easier scenario.

A way to add client migration would be to configure a client manually to perform migration on a determined port, so that we can easily detect it in `Ivy`. However, all implementations do not directly allow that. In consequence, we left it as future work. It is interesting in the sense that it somehow reflects the results that we might observe without migration.

Failure of a requirement in the `QUIC` packet specification that we already get for other implementations in the server part but never for `picoquic` [ref. 19]:

```
require ~ _generating  ∧ ~ queued_non_ack(scid)
        -> ack_credit(scid)> 0; # [5]
```

src port	dst port	Information
47449	4443	1252, Initial, DCID=934af1a3b9ffb70d, SCID=e231a5b571d6a9cc, PKT: 0, CRYPTO, PADDING
47449	4443	1252, Initial, DCID=934af1a3b9ffb70d, SCID=e231a5b571d6a9cc, PKT: 1, CRYPTO, PADDING
4443	47449	1232, Initial, DCID=e231a5b571d6a9cc, SCID=0000000000000000, PKT: 7, CRYPTO, PADDING
47449	4443	53, Handshake, DCID=0000000000000000, SCID=e231a5b571d6a9cc, PKT: 0, PADDING
47449	4443	53, Handshake, DCID=0000000000000000, SCID=e231a5b571d6a9cc, PKT: 1, PADDING
4443	47449	1397, Handshake, DCID=e231a5b571d6a9cc, SCID=0000000000000000, PKT: 6, CRYPTO, PADDING
47449	4443	89, Handshake, DCID=0000000000000000, SCID=e231a5b571d6a9cc, PKT: 2, ACK
47449	4443	186, Protected Payload (KP0), DCID=0000000000000000, PKT: 0, NCI, PADDING
47449	4443	1440, Protected Payload (KP0), DCID=0000000000000000, PKT: 1, PING, PADDING
47449	4443	97, Protected Payload (KP0), DCID=0000000000000000, PKT: 2, STREAM(0), PADDING
4443	47449	46, Protected Payload (KP0), DCID=e231a5b571d6a9cc, PKT: 1, DONE, PADDING
47449	4443	55, Protected Payload (KP0), DCID=0000000000000000, PKT: 3, ACK, PADDING
47449	4443	183, Protected Payload (KP0), DCID=0000000000000000, PKT: 4, NCI, ACK, PADDING
47449	4443	1400, Protected Payload (KP0), DCID=0000000000000000, PKT: 5, PING, PADDING
47449	4443	97, Protected Payload (KP0), DCID=0000000000000000, PKT: 6, STREAM(0), PADDING
47449	4443	183, Protected Payload (KP0), DCID=0000000000000000, PKT: 7, NCI, ACK, PADDING
4443	47449	99, Protected Payload (KP0), DCID=0dd5652d0b593b48, PKT: 4, ACK, MD, ACK, PADDING
47449	4443	97, Protected Payload (KP0), DCID=0000000000000000, PKT: 8, ACK, PADDING
4443	47449	100, Protected Payload (KP0), DCID=e231a5b571d6a9cc, PKT: 9, STREAM(0), ACK, PADDING
47449	4443	97, Protected Payload (KP0), DCID=0000000000000000, PKT: 9, ACK, STREAM(4), PADDING
47449	4443	97, Protected Payload (KP0), DCID=0000000000000000, PKT: 10, STREAM(0), PADDING
4443	47449	50, Protected Payload (KP0), DCID=0dd5652d0b593b48, PKT: 16, ACK, PADDING
47449	4443	225, Protected Payload (KP0), DCID=0000000000000000, PKT: 11, NCI, ACK, PADDING
ACK(LA: 16)		
4443	47449	55, Protected Payload (KP0), DCID=0dd5652d0b593b48, PKT: 32, ACK, ACK, PADDING
47449	4443	55, Protected Payload (KP0), DCID=0000000000000000, PKT: 12, ACK, PADDING
ACK(LA: 32)		
4443	47449	50, Protected Payload (KP0), DCID=0dd5652d0b593b48, PKT: 33, ACK, PADDING

As can be seen in this trace, `picoquic` acknowledges `ACK`-only packets twice. This problem also arises in `aioquic` and `lsquic`. We think that migration increases the probability of this kind of error since the number of errors thrown by `aioquic` is considerably reduced compared to the test we performed with the server.

The problem of the retransmission timer is shown in the following output of `picoquic`:

```
6ed30c9cf9a32709: T= 19.337266, cwin: 5008,flight: 403,nb_ret: 19,rtt_min:
    1414454,rtt: 3000892,rtt_var: 2635210,max_ack_delay: 0,state: 14
6ed30c9cf9a32709: Sending 55 bytes to 127.0.0.1:4443 at T=19.337266
    (5c246da3f7367)
6ed30c9cf9a32709: T= 20.658154, Lost packet type 6, number 24, size 39, DCID
    <0000000000000000>, reason: timer
Received a request to close the connection.
Connection end with local error 0x434.
Quic Bit was NOT greased by the client.
Quic Bit was NOT greased by the server.
ECN was not received.
ECN was not acknowledged.
Received 8 bytes in 20.658195 seconds, 0.000003 Mbps.
6ed30c9cf9a32709: Received 8 bytes in 20.658195 seconds, 0.000003 Mbps.
Client exit with code = -1
```

This local error is only found with log inspections and the `CONNECTION_CLOSE` frame is not sent. We also faced similar problems with `quiche`. The performance of `Ivy` should be increased since this is not voluntary, but it still allows detecting some incorrect behaviours.

A problem with the `HANDSHAKE_DONE` was detected. We discovered that `quinn` returns a `CONNECTION_CLOSE` frame with error code 0, meaning that there is `NO_ERROR`. It is because we send a significant number of `HANDSHAKE_DONE` as soon as we can. It is not forbidden by the draft and it can be interesting to see how implementation reacts [ref. 20].

src port	dst port	Information
59841	4443	1242 Initial, DCID=55feed94e19a636ec3aa407b21982368, SCID=f6c43cccb16865d2, PKN: 0, CRYPTO, PADDING
4443	59841	1274 Initial, DCID=f6c43cccb16865d2, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
59841	4443	1242 Initial, DCID=0000000000000000, SCID=f6c43cccb16865d2, PKN: 1, ACK, PADDING
4443	59841	1463 Handshake, DCID=f6c43cccb16865d2, SCID=0000000000000000, PKN: 9, CRYPTO, PADDING
59841	4443	144 Handshake, DCID=0000000000000000, SCID=f6c43cccb16865d2, PKN: 0, ACK, CRYPTO
59841	4443	116 Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, STREAM(0)
59841	4443	88 Protected Payload (KP0), DCID=f6c43cccb16865d2, PKN: 11, DONE, PADDING
59841	4443	73 Protected Payload (KP0), DCID=0000000000000000, PKN: 1, ACK
59841	4443	96 Protected Payload (KP0), DCID=f6c43cccb16865d2, PKN: 32, DONE, DONE, DONE, DONE, DONE, DONE, DONE, DONE, DONE, PADDING
59841	4443	75 Protected Payload (KP0), DCID=0000000000000000, PKN: 2, ACK
59841	4443	103 Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, NCI
59841	4443	103 Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, NCI
59841	4443	97 Protected Payload (KP0), DCID=1cca369fdcfb0484, PKN: 36, ACK, ACK, PADDING
59841	4443	97 Protected Payload (KP0), DCID=1cca369fdcfb0484, PKN: 38, DONE, STREAM(0), DONE, PADDING
4443	59841	77 Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK
4443	59841	80 Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, MS
4443	59841	75 Protected Payload (KP0), DCID=0000000000000000, PKN: 7, CC
CONNECTION_CLOSE (Application) Error code: 0		
		Frame Type: CONNECTION_CLOSE (Application) (0x000000000000001d)
		Application Error code: 0
		Reason phrase Length: 4
		Reason phrase: done

The only requirements about these frames are:

> *"Note that it is not possible to send the following frames in 0-RTT packets for various reasons:* `ACK`*,* `CRYPTO`*,* `HANDSHAKE_DONE`*,* `NEW_TOKEN`*,* `PATH_RESPONSE`*, and* `RETIRE_CONNECTION_ID`*. A server MAY treat receipt of these frames in 0-RTT packets as a connection error of type* `PROTOCOL_VIOLATION`*."*
>
> Draft 29 of `QUIC-TLS` section 4.

> *"An endpoint MUST discard its handshake keys when the* TLS *handshake is confirmed (Section 4.1.2). The server MUST send a* HANDSHAKE_DONE *frame as soon as it completes the handshake."*
>
> Draft 29 of QUIC-TLS section 4.11.2

> *"The* HANDSHAKE_DONE *frame MUST be retransmitted until it is acknowledged."*
>
> Draft 29 of QUIC section 13.3.

> "*A* HANDSHAKE_DONE *frame can only be sent by the server. Servers MUST NOT send a* HANDSHAKE_DONE *frame before completing the handshake. A server MUST treat receipt of a* HANDSHAKE_DONE *frame as a connection error of type* PROTOCOL_VIOLATION"
>
> Draft 29 of QUIC section 19.20.

Following the draft, the handshake is confirmed at different conditions for the client and the server as explained in 4.3.1. For the client, the handshake is considered as confirmed when the HANDSHAKE_DONE is received. However, for the server, it is considered confirmed just when the TLS stack has both sent a "Finished" message and verified the peer's "Finished" message. It is done before the sending of HANDSHAKE_DONE.

Following the draft, an endpoint cannot migrate before the handshake is confirmed. If a client migrates just after it receives a HANDSHAKE_DONE, the client respects the specification. The server has to accept the migration before it receives the ACK for the HANDSHAKE_DONE.

However, we observed that some implementations did not interpret the draft this way. Some implementations wait to receive the ACK from the HANDSHAKE_DONE before allowing the client to migrate. Their interpretation could be that the only way to verify if the client has the right to migrate is to know if the client considers the handshake as confirmed. The reception of the ACK is the only way to have such information.

There is an issue with the current state of the draft because it prevents the server to know if a client finished its handshake before migrating.

From the client point of view, a constant retransmission of HANDSHAKE_DONE frames could indicate that the server does not know if the client considers the handshake as confirmed. The client could consider that the frame is retransmitted because the server did not receive the ACK. This may put complexity for some QUIC features.

An example is the migration of the client where the draft says that:

> "*The design of* QUIC *relies on endpoints retaining a stable address for the duration of the handshake. An endpoint MUST NOT initiate connection migration before the handshake is confirmed, as defined in section 4.1.2 of [*QUIC-TLS*].*"
>
> Draft 29 of QUIC section 9.

Other requirements would also be affected since we are limited when the handshake is not confirmed.

This problem also appears in other implementations in another shape. With `picoquic` we enter in a loop where few data is exchanged and most parts of the traffic is about ACK frames. There is a problem in the handling of those HANDSHAKE_DONE frames in such situation.

4.4.2 Unknown situations

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
unkown	100%	96%	99%	0%	0%	100%	0%
tp_unkown	100%	34%	99%	99%	100%	99%	96%

Table 4.11: Client - Unknown situation

The results are similar to the ones obtained with the server. We can explain the difference with the server for `tp_unknown` and `picoquic` that is caused by a retransmission timeout error.

Looking at `lsquic`, we observed that it does not respect the specification for the part about UNKNOWN frame. It must return a FRAME_ENCODING_ERROR but it returns a INTERNAL_ERROR code [ref. 21]:

src port	dst port	Information
4988	4443	1322, Initial, DCID=0000000000000002, SCID=0000000000000001, PKT: 7, CRYPTO PADDING
4443	4988	1111, Handshake, DCID=0000000000000001, SCID=2b5a88d1fc6c5918, PKT: 0, ACK ACK
4443	4988	158, Handshake, DCID=0000000000000001, SCID=2b5a88d1fc6c5918, PKT: 2, CRYPTO
4988	4443	98, Protected Payload (KP0), DCID=2b5a88d1fc6c5918, PKT: 6, CRYPTO PADDING
4443	4988	89, Protected Payload (KP0), DCID=0000000000000001, PKT: 3, DONE NCI
4987	4443	92, Protected Payload (KP0), DCID=2b5a88d1fc6c5918, PKT: 1, Unknown PADDING
4443	4988	1274, Initial, DCID=0000000000000001, SCID=0, PKT: 4, CC
		CONNECTION_CLOSE (Transport) Error code: INTERNAL_ERROR
		Frame Type: CONNECTION_CLOSE (Transport) (0x000000000000001c)
		Error code: INTERNAL_ERROR (1)
		Frame Type: 0
		Reason phrase Length: 16
		Reason phrase: connection error

4.4.3 Errors handling

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
double_tp_error	0%	100%	100%	0%	0%	0%	0%
tp_error	0%	0%	100%	0%	0%	0%	0%
tp_acticoid_error	0%	0%	0%	0%	100%	0%	0%
no_odci	0%	100%	100%	0%	0%	0%	0%
tp_prefadd_error	0%	100%	0%	0%	0%	0%	0%

Table 4.12: Client - Transport parameter errors tests

The transport parameter results are also like the ones obtained with the server. `Picoquic`, `quic-go`, `quant` and `aioquic` are at the same state as for the server tests.

`Quiche` also has the same requirement that does not pass like for the server part. An exception is for the `double_tp` test where it decides to continue the executions.

We could be surprised that `quinn` seems to fail all the tests for the clients but not for the server. However, it returns the good error code here. The problem is that it returns the error in 1-RTT encryption level when in fact we are not supposed to have installed those keys at the current stage of the execution:

src port	dst port	Information
33495	4443	1200, Initial, DCID=83ad071c1861c27019792bc59677f195, SCID=99ac07233a0d5b0d, PKT: 0, CRYPTO
4443	33495	1232, Initial, DCID=99ac07233a0d5b0d, SCID=0000000000000000, PKT: 7, CRYPTO
33495	4443	1200, Initial, DCID=0000000000000000, SCID=99ac07233a0d5b0d, PKT: 1, ACK
4443	33495	1472, Handshake, DCID=99ac07233a0d5b0d, SCID=0000000000000000, PKT: 6, CRYPTO
33495	4443	39, Protected Payload (KP0), DCID=0000000000000000, PKT: 0, CC (TRANSPORT_PARAMETER_ERROR) (Undecryptable)

When we look at Figure 4.1 taken from the `QUIC-TLS` draft. We see that as a server, we are supposed to install our receiving 1-RTT key only when it sends us the "Handshake" packet completing the handshake. Before that only Initial and Handshake encryption levels are allowed and thus the errors for the transport parameters cannot be sent in 1-RTT packets.
We can also use the following requirement:

> *"A server MUST NOT process incoming 1-RTT protected packets before the* `TLS` *handshake is complete. Because sending acknowledgments indicates that all frames in a packet have been processed, a server cannot send acknowledgments for 1-RTT packets until the* `TLS` *handshake is complete"*

Or: Draft 29 of `QUIC-TLS` section 5.7.

> *"Due to reordering and loss, protected packets might be received by an endpoint before the final* `TLS` *handshake messages are received. A client will be unable to decrypt 1-RTT packets from the server, whereas a server will be able to decrypt 1-RTT packets from the client. Endpoints in either role MUST NOT decrypt 1-RTT packets from their peer prior to completing the handshake."*

Draft 29 of `QUIC-TLS` section 5.7.

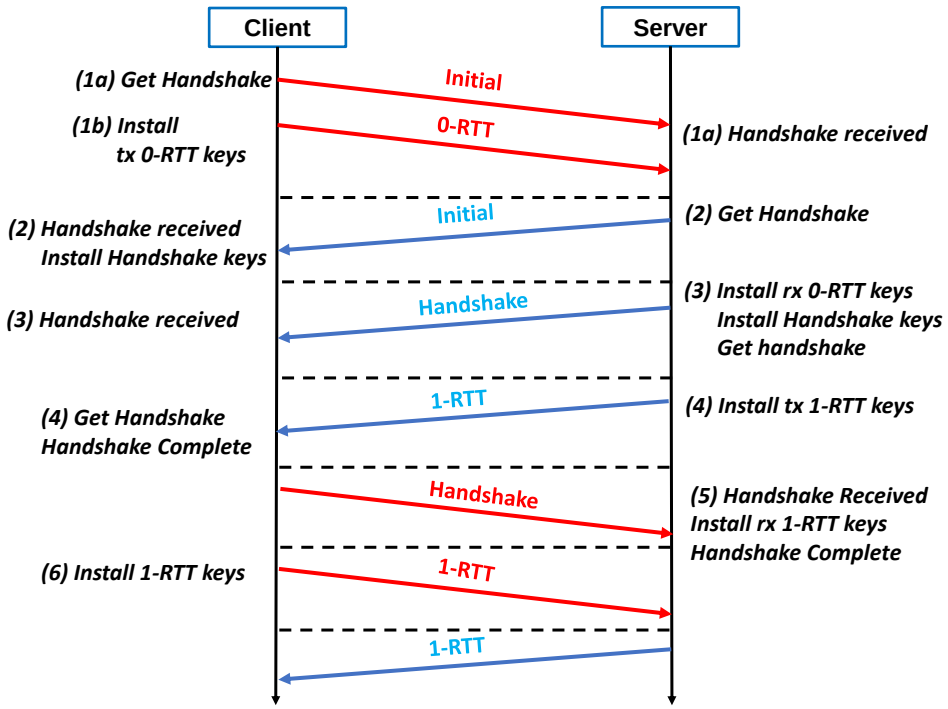


Figure 4.1: Interaction summary between QUIC and TLS [8]

An example of a valid execution is with picoquic:

src port	dst port	Information
46321	4443	1252, Initial, DCID=742a72c93a69f122, SCID=a7456b3c5a066271, PKT: 0, CRYPTO
4443	46321	1232, Initial, DCID=a7456b3c5a066271, SCID=0000000000000000, PKT: 7, CRYPTO
46321	4443	53, Handshake, DCID=0000000000000000, SCID=a7456b3c5a066271, PKT: 0, PADDING
4443	46321	1232, Handshake, DCID=a7456b3c5a066271, SCID=0000000000000000, PKT: 6, CRYPTO
46321	4443	54, Handshake, DCID=0000000000000000, SCID=a7456b3c5a066271, PKT: 1, CC(TRANSPORT_PARAMETER_ERROR)

Some implementations do not check the connection ID validity for the preferred_address transport parameter. Nothing is detected wrong for aioquic and quiche, as you can see our parameter are well formed:

src port	dst port	Information
59841	4443	1322 Initial, DCID=334504622eb55b3a, SCID=024cb26de12cf949, PKN: 0, CRYPTO, PADDING
4443	59841	1274 I nitial, DCID=024cb26de12cf949, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
59841	4443	92 Initial, DCID=0000000000000001, SCID=024cb26de12cf949, PKN: 1, ACK
4443	59841	1510 Handshake, DCID=024cb26de12cf949, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
Parameter: preferred_address (len=41) Type: preferred_address (0x0d) Length: 41 Value: 7f000001115b0000000000000007f0000017f000002115b0000000000000000007f0000... ipv4Address: localhost (127.0.0.1) ipv4Port: 4443 ipv6Address: ::7f00:1:7f00:2 (::7f00:1:7f00:2) ipv6Port: 4443 Length: 0 connectionId: <MISSING> statelessResetToken: 0000000000000007f0000017f000001		
59841	4443	202 Protected Payload (KP0), DCID=0000000000000001, PKN: 3, NCI
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 4, STREAM(0)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 5, STREAM(4)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 6, STREAM(8)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 7, STREAM(12)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 8, STREAM(16)

59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 9, STREAM(20)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 10, STREAM(24)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 11, STREAM(28)
59841	4443	90 Protected Payload (KP0), DCID=0000000000000001, PKN: 12, STREAM(32)
4443	59841	88 Protected Payload (KP0), DCID=024cb26de12cf949, PKN: 12, DONE, PADDING
...		

`Picoquic` and `quinn` report the good error but for `quinn` it is at the wrong encryption level. `Quic-go` is an outsider. It ignores the packet and tries to reconnect with "Client Hello" messages.

`Aioquic` also triggers a local exception for `client_test_no_ocid`:

```
2021-05-30 10:46:41,468 WARNING quic [94b28af9d0a43a37] Error: 8, reason:
    original_destination_connection_id does not match, frame_type:
    QuicFrameType.CRYPTO
2021-05-30 10:46:41,468 DEBUG quic [94b28af9d0a43a37]
    QuicConnectionState.CONNECTED -> QuicConnectionState.CLOSING
2021-05-30 10:46:42,972 DEBUG quic [94b28af9d0a43a37] Discarding epoch
    Epoch.INITIAL
2021-05-30 10:46:42,972 DEBUG quic [94b28af9d0a43a37] Discarding epoch
    Epoch.HANDSHAKE
2021-05-30 10:46:42,972 DEBUG quic [94b28af9d0a43a37] Discarding epoch
    Epoch.ONE_RTT
2021-05-30 10:46:42,972 DEBUG quic [94b28af9d0a43a37] QuicConnectionState.CLOSING
    -> QuicConnectionState.TERMINATED
Traceback (most recent call last):
  File "examples/http3_client.py", line 465, in <module>
    zero_rtt=args.zero_rtt,
  File "/usr/lib/python3.6/asyncio/base_events.py", line 484, in
      run_until_complete
    return future.result()
  File "examples/http3_client.py", line 313, in run
    wait_connected=not zero_rtt,
  File "/home/chris/TVOQE_UPGRADE_27/quic/aioquic/src/aioquic/asyncio/compat.py",
      line 17, in __aenter__
    return await self.gen.__anext__()
  File "/home/chris/TVOQE_UPGRADE_27/quic/aioquic/src/aioquic/asyncio/client.py",
      line 89, in connect
    await protocol.wait_connected()
  File
      "/home/chris/TVOQE_UPGRADE_27/quic/aioquic/src/aioquic/asyncio/protocol.py",
      line 127, in wait_connected
    await asyncio.shield(self._connected_waiter)
ConnectionError
```

It sends the good error but at the wrong encryption level:

src port	dst port	Information
43142	4443	1280, Initial, DCID=7296a6dbd8b6215d, SCID=23217d8b6f911c45, PKT: 0, CRYPTO
4443	43142	1232, Initial, DCID=23217d8b6f911c45, SCID=0000000000000001, PKT: 7, CRYPTO
43142	4443	50, Initial, DCID=0000000000000001, SCID=23217d8b6f911c45, PKT: 1, ACK
4443	43142	1405, Handshake, DCID=23217d8b6f911c45, SCID=0000000000000001, PKT: 6, CRYPTO
43142	4443	80, Protected Payload (KP0), DCID=0000000000000001, PKT: 0, CC (TRANSPORT_PARAMETER_ERROR) (Undecryptable)

Finally, `lsquic` never returns an error and locally closes the connection:

src port	**dst port**	**Information**
36511	4443	1200, Initial, DCID=e6ade284935dc9695c79b1d903b2eb, SCID=e2bb04c43c1514f5, PKT: 0, CRYPTO
36511	4443	1200, Initial, DCID=e6ade284935dc9695c79b1d903b2eb, SCID=e2bb04c43c1514f5, PKT: 1, CRYPTO
4443	36511	1232, Initial, DCID=e2bb04c43c1514f5, SCID=0000000000000000, PKT: 7, CRYPTO
36511	4443	1200, Initial, DCID=0000000000000000, SCID=e2bb04c43c1514f5, PKT: 2, ACK
4443	36511	1419, Handshake, DCID=e2bb04c43c1514f5, SCID=0000000000000000, PKT: 9, CRYPTO

With this message in the logs:

```
17:00:14.260 [DEBUG] [QUIC:E2BB04C43C1514F5] event: decrypted packet 9
17:00:14.260 [DEBUG] [QUIC:E2BB04C43C1514F5] event: packet in: 9, type:
    Handshake, size: 1419; ecn: 0, spin: 0; path: 0
17:00:14.260 [DEBUG] [QUIC:E2BB04C43C1514F5] event: CRYPTO frame in: level 2;
    offset 0; size 1354
17:00:14.261 [DEBUG] [QUIC:E2BB04C43C1514F5] event: handshake completed
17:00:14.261 [DEBUG] engine: decref conn E2BB04C43C1514F5, 'HTA' -> 'HA'
17:00:14.261 [DEBUG] engine: decref conn E2BB04C43C1514F5, 'HA' -> 'H'
17:00:14.261 [DEBUG] engine: incref conn E2BB04C43C1514F5, 'H' -> 'CH'
17:00:14.261 [DEBUG] engine: decref conn E2BB04C43C1514F5, 'CH' -> 'C'
17:00:14.261 [DEBUG] engine: decref conn E2BB04C43C1514F5, 'C' -> ''
17:00:14.261 [DEBUG] [QUIC:E2BB04C43C1514F5] event: full connection destroyed
17:00:14.261 [DEBUG] engine: destroying engine
```

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
blocked_error	99%	0%	97%	0%	0%	91%	98%
retirecoid_error	99%	99%	100%	0%	0%	0%	98%

Table 4.13: Client - Invalid fields frames errors tests

For the tests with frames containing invalid fields, the results are similar to the server side. `Lsquic` returns the good error code for each test and conforms to the specification.

	quinn	picoquic	quic-go	aioquic	quant	quiche	lsquic
new_token_error	98%	94%	96%	1%	0%	87%	100%
limit_max_error	0%	88%	0%	0%	81%	0%	0%

Table 4.14: Client - Protocol violation errors tests

For the last category of tests, we conclude that most implementations detect and return the good error when we send `NEW_TOKEN` frame with zero-length field. However, `aioquic` and `quant` do not.

For `limit_max_error` where we send `STREAM` frames with stream ID exceeding the limit set by the peer we get the same problem as for the server. Most implementations return a `STREAM_STATE_ERROR` instead of a `STREAM_LIMIT_ERROR`.

Most implementations indicating that they received this frame are in a disallowed state. Here is an example with `quinn`:

```
ERROR: failed to shutdown stream: connection closed: received a frame for a
    stream that was not in a state that permitted that frame: operation on
    unopened stream
```

Probably the stream was locally initiated by the client since it has prepared its requests for the server.

4.5 Summary

Let us summarize. The first point to highlight is that we were unable to test all implementations. It is the case for the server test with `lsquic` where we get a `TLS` internal error. It is also the case with the client and `mvfst` that used zero length connection ID. All features of `QUIC` and `TLS` are not available in `Ivy` and this can lead to such problems. However, we managed to improve the compatibility between `Ivy` and other implementations. When we look at other testing tools used for `QUIC`, this point seems normal [21].

We also decided to focus on the type of error rather than the number of errors since it could be misinterpreted or lead to a false conclusion. This perception is well reflected when we compare the result of the client and the server tests. As a reminder, with the server tests we allowed multiple migration per test and not at all with the client. Only looking at the number of tests passed for `aioquic` or `quant` could lead to the conclusion that they have a lot of faults. However, our analysis suggests that most of the errors are linked to the migration and removing this process increases a lot the success rate of both implementation.

Now considering the types of errors, they are various. We faced some implementations that simply do not implement a requirement in the specification. This result is well reflected by the error management for example where usually either the management of the error is implemented either it is not.

The error management also underlines another type of problem that is non-respect of the draft. Some implementations return the wrong error code, sometimes not the most appropriate one. Others return an error which does not correspond to the good case, an example is `quic-go` when we send `HANDSHAKE_DONE` frame as clients. The application layer could use those error codes to decide, for example, to fall back on a TCP connection. A good error management is important.

There is also the fact that some implementations send error codes with the wrong message. However these problems can only be found with a post-analysis of the results and not directly with `Ivy`.

Some implementation like `quant` usually detect the error locally without reporting an error which is not conforming to the specification. However, when we analyse the latest version, that problem has been solved:

	quant 29	quant master
double_tp_error	3%	100%
tp_error	0%	100%
tp_acticoid_error	100%	100%
no_icid_error	0%	100%

Table 4.15: Quant transport parameter: before/after

This example shows how `Ivy` can be used to track the evolution of an implementation. We also found errors for `aioquic` and managed to fix them. The evolution was also reported by `Ivy`.

For the tests that verify the specification as a whole, we do not detect too many errors. Most of the problems are linked to acknowledgement of acknowledgements and migration for the path validation. The first one is not critical, at least it never leads to a crash and no major security issues were detected. We supposed that it is also correlated with migration. In fact, this error is more present when the migration is allowed.

However the migration problem is more serious. Many security considerations are involved to guarantee authentication, confidentiality and integrity of the messages exchanged between endpoints. Many attacks are linked to the migration such as the "Peer Address Spoofing", "On-Path Address Spoofing" or the "Off-Path Packet Forwarding" describe in the `QUIC` specification. It is important that path and address validation are well implemented.

Another kind of problem is that sometimes the `QUIC` specification is not precise enough. It can also be ambiguous. This could lead to situations where the service is not available anymore. This problem was highlighted with `mvfst` because we are not sure when we are allowed to perform migration which makes fail the connection ~50%. See section 4.3.1 for more detail.

This problem is also reflected in our model where polysemous requirements lead to different valid formal interpretations (3.4.3). The best example is with the highest numbered non-probing packets to which an endpoint should send its packets. Is it the highest among all packets or among application packets ? Even if this question seems trivial for a network specialist, the ambiguity should be removed.

Another type of problem is that sometimes implementations use the wrong encryption level. This is mostly present for reporting the error where `aioquic` or `quinn` used the 1-RTT encryption that is not allowed at this moment.

Finally, there is also the fact that some implementations do not check if a specific frame is allowed by their peers as for the `HANDSHAKE_DONE` and `NEW_TOKEN` that are not allowed for the client.

Chapter 5

Discussion

5.1 Key results

We modelled the specification of QUIC specification into Ivy from draft 18 to draft 29. Some properties are impossible to verify formally due to different kinds of limitations explained through this thesis. However, we managed to add the verification of many rules that Kenneth McMillan & Zuck did not verify in their previous work as well as the adaptation and the creation of all the rules that changed between the two drafts. One QUIC extension was also verified.

We also showed that it was possible to extend McMillan's work and confirmed that Ivy was a tool that could be suited for the testing of network protocols. We did not manage to contact McMillan during our work and did not see anyone who continued this project. Therefore, we refactored the project to make it easier to understand and extend. This work can be useful for new contributors who want to keep the update of this formal verification of QUIC extensions in the new specifications. It is also possible for contributors who will want to use Ivy to test other network protocols to use our work. As we modelled and tested the properties of one extension, we show that there are no limitations for the testing of new QUIC extensions.

We can also see the evolution of one implementation according to the draft by testing different versions. We showed that with quant and observed that most problems linked to the transport parameters were fixed during the year.

Finally, we showed the usefulness of all our work. By testing eight QUIC implementations that are, for most of them, at a very advanced stage of development, we showed that some parts of the specification were not respected. We found different types of errors that are summarized in the section 4.5. We tried to solve some issues for one implementation aioquic. Another benefit of our work is that we highlight some parts of the specification that are not clear or ambiguous and that could be improved for the writing of new drafts.

5.2 RFC recommendations

During the work, we observed several limits to the way the RFC was expressed. This influences the formal specification of QUIC. We present in this section some requirements we faced that introduce different behaviours between the tested implementation, sometimes leading to complexity.

The first problem is when an endpoint sets a transport parameter twice. The QUIC specification indicates that:

> *An endpoint MUST NOT send a parameter more than once in a given transport parameters extension. An endpoint SHOULD treat receipt of duplicate transport parameters as a connection error of type* TRANSPORT_PARAMETER_ERROR.
>
> Draft 29 of QUIC section 7.4.

There are different interpretations of this requirement that lead to different results for an endpoint that receive a double transport parameter. The most basic case is when we literally applied the rule: we only *should* throw a TRANSPORT_PARAMETER_ERROR. If an implementation continues its execution when it receives this double parameter, it does not break the specification. In this case, an implementation cannot know which transport parameter to consider.

Besides, we can consider that we must throw at least a PROTOCOL_VIOLATION since the "*MUST NOT*" statement is not respected and this error should be reported with a TRANSPORT_PARAMETER_ERROR. With this approach, we avoid the problem of knowing which value to consider.

Both interpretations seem valid. The first one is the one we implemented in Ivy because it respects the semantic of the draft. However, we believe that the second option would lead to fewer problems. A better specification for the second option would be:

> *An endpoint MUST NOT send a parameter more than once in a given transport parameters extension. An endpoint MUST treat receipt of duplicate transport parameters as a connection error and SHOULD used a connection error type of* TRANSPORT_PARAMETER_ERROR.
>
> Draft 29 of QUIC section 7.4. ***modified***

Another point that should be more precise is about the confirmation of the handshake. It is a problem that we faced with mvfst. The QUIC specification indicates that:

> "*In this document, the TLS handshake is considered confirmed at the server when the handshake completes. At the client, the handshake is considered confirmed when a* HANDSHAKE_DONE *frame is received.*"
>
> Draft 29 of QUIC-TLS section 4.1.2.

> "*In this document, the TLS handshake is considered* ***complete*** *when the TLS stack has reported that the handshake is complete. This happens when the TLS stack has both sent a Finished message and verified the peer's Finished message. Verifying the peer's Finished provides the endpoints with an assurance that previous handshake messages have not been modified. Note that the handshake does not complete at both endpoints simultaneously. Consequently, any requirement that is based on the completion [also confirmation for server] of the handshake depends on the perspective of the endpoint in question.*"
>
> Draft 29 of QUIC-TLS section 4.1.1.

Following the draft, an endpoint cannot migrate before the handshake is confirmed. If a client migrates just after he receives a HANDSHAKE_DONE, the client respects the draft. The server has to accept the migration before he receives the ACK for the HANDSHAKE_DONE. However, the solution to wait that the HANDSHAKE_DONE is acknowledged is not a bad idea. This way, the server can know if the client respects the QUIC specification or not. It is better to be sure that both endpoints completed the handshake, but we have to wait a little bit more. The literal version of the specification allows an endpoint to migrate as soon as a HANDSHAKE_DONE frame is received. The endpoint could then acknowledge the frame from the migrated address. This is the behaviour we implemented in Ivy.

We can also consider the following requirement:

> "*The* HANDSHAKE_DONE *frame MUST be retransmitted until it is acknowledged.*"
>
> Draft 29 of QUIC section 13.3.

From the client point of view, a constant retransmission of HANDSHAKE_DONE frames could indicate that the server does not know if the client considers the handshake as confirmed. The client could consider that the frame is retransmitted because the server did not receive the ACK. This may add complexity for some QUIC features.

A simple but possible solution is inspired from TCP with the fast retransmit "dupack" strategy [67]. Instead of retransmitting a packet after a specific number of retransmissions, we stop the connection. With this solution, a problem in the handshake can be detected. Here is how the specification would look like:

> "*The* HANDSHAKE_DONE *frame MUST be retransmitted until it is acknowledged. A server MUST treat receipt of a* HANDSHAKE_DONE *frame as a connection error of type* CONNECTION_REFUSED *if it receives the frame more than 3 times*"
>
> Draft 29 of QUIC section 13.3. ***modified***

Many requirements are related to the completion of the handshake and such misinterpretation of the draft can lead to potential unwanted situation as with mvfst and quinn. It also adds complexity that can be removed with our solution.

Finally, there is a set of requirements that should be more precise because it involves

concepts that are unclearly written. A good example is the following rule:

> "*An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.*"
>
> Draft 29 of QUIC section 9.3.

For a network protocol implementer, it can seem obvious to consider only the 1-RTT and 0-RTT encryption levels. However, if we read the rule formally, nothing states that we should consider only those encryption levels. The draft can easily be modified to remove this ambiguity. We could be more precise by insisting on the encryption level considered:

> "*An endpoint only changes the address that it sends packets to in response to the highest-numbered non-probing packet in the 1-RTT and 0-RTT encryption levels. This ensures that an endpoint does not send packets to an old peer address in the case that it receives reordered packets.*"
>
> Draft 29 of QUIC section 9.3. ***modified***

5.3 SIGCOMM 2021 Posters

We prepared an introduction document to participate and present our results to the SIGCOMM [68] conference with participation and help of Maxime Piraux, Tom Rousseaux, Axel Legay and Olivier Bonaventure. We also plan to realize eventually a scientific paper in verification.

5.4 Limitations & improvements

Ivy suffers from some limitations that limited the scope of our work.

5.4.1 Real time properties

We could not verify properties that are based on time. This problem was already identified by McMillan & Zuck. There are some solutions to this issue. Temporal logic can be used for some properties [69]. Atomic predicate could also be used, as shown in [70]. However, it would require to introduce significant modifications to Ivy to be able to verify these rules.

5.4.2 Liveness properties

Liveness properties are difficult to verify by nature. In our work, we tried to model some of them by setting some heuristics. However, we found out that there was a balance to respect. If we put optimistic heuristics, we can observe more types of errors from the implementations for the liveness properties that we check. However, the false positive results increases. There is a balance to find.

5.4.3 Internal state properties

First, when we use `Ivy`, we have an external point of view of the implementations as we only see what is on the wire. It is impossible to do code inspections using `Ivy`, which restricts the set of requirements that can be tested. A possibility to overcome this problem could be to use `eBPF` [71]. We could use this tool to perform a static analysis of QUIC implementations and verify some rules we could not verify with `Ivy` [72].

5.4.4 Data types

Originally, `Ivy` accepts variables of maximum eight bytes. It was not a problem for the implementations that McMillan selected. We managed to improve `Ivy` to make it support data up to 16 bytes. This peak value of 16 bytes for `Ivy` datatype limits some tests we could perform and is a constraint to generate a scenario for some implementations. One example is the Connection ID. The draft states that it can be between 0 and 20 bytes, but only a subset of implementations use a Connection ID of more than 16 bytes. The truncation of this value could lead to several issues. Consequently, we modified the implementations themselves. They now take connection ID of maximum 16 bytes. Generally, it consisted in adapting just one global variable.

A possible solution to this problem would be to add a new data type and implement the C++ template as we did for `uint_128`. We can use for that a vector that will be used to represent data types of any length. However, we cannot simply extend the current template because we need to specify the "*length*" parameter. We can create a new module based on the old one and adapt all the function . That would be more versatile and allow any maximal size. However, we should be careful since a wrong implementation can lead to memory fault with arrays. Also, it would involve a rewriting of all the templates that is costly. .

5.5 Further work

There are several ways in which our work could be extended. Many extensions are currently in progress. One example could be `Multipath QUIC` [73], Forward Erasure Correction (`FEC`) [20] or unreliable datagram extension [74].

Completing `Ivy` with the missing packets is also possible. Examples of such packets are the Retry and Version Negotiation. It is possible to add them in the current model of `Ivy`. The serializer and deserialiser would need to be rewritten. Also, it is important to consider the performance issues when adding new features to the model. Adaptations could be needed.

Furthermore, we did not update to the latest draft. Currently, the latest draft is the 34^{th} one. The principal reason why we did not update to this draft is that it is

evolving fast. There were only months between draft 29 and draft 34. We preferred to stabilize our results on a specific version of the draft to simplify our work. It is also important to note that the speed at which the different implementations obey the latest draft is most of the time unknown. Fortunately, as `QUIC` evolves, there are fewer and fewer changes between the different drafts, which makes it less tedious to update the work from a draft version to another one.

A part of our work consisted in finding tricks to reduce to execution time of our testing approach. There is probably room to improve it. We did not manage to test some rules because of this limitation. If one manages to reduce significantly the time consumed by `Ivy` to run the tests or generate packets, new possibilities for testing untested rules would arise. Another way could be to have more control over the network to mitigate these problems, we could use a virtual network that would make our assumptions true by controlling the frequency of the packet [75].

It would also be possible to test our tool on other implementations than the ones we selected. We showed that the addition of new implementations to test was not an issue. It would be an easy way to search for more interesting results.

In the same vein, we could create a website where we would show the compliance of every implementation with the specification, by launching our tests that would be updated to each new draft for every implementation in our testing set. The same philosophy as QUIC-Tracker [21].

Development using `Ivy` could be improved by adding unit testing. There is currently no unit test for the `QUIC` module of `Ivy`. Due to time constraints, we left it as future work. However, we could observe that the verification of new properties we added was performing well by observing the results. Adding unit tests should remain a priority because it would improve the trust in automated testing and could increase the speed of development.

Bibliography

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt

[2] K. L. McMillan and L. D. Zuck, "Compositional testing of internet protocols," in 2019 IEEE Cybersecurity Development (SecDev). IEEE, 2019, pp. 161–174.

[3] "Ivy description," https://www.microsoft.com/en-us/research/project/ivy/, accessed: 2021-02-27.

[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, 2019, pp. 227–240.

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-29, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-29

[6] A. P. M. S. S. S. Kennet McMillan, Oded Padon, "Ivy: Home." [Online]. Available: http://microsoft.github.io/ivy/

[7] "Website of Kenneth McMillan," http://mcmil.net/wordpress/, accessed: 2021-02-27.

[8] M. Thomson and S. Turner, "Using TLS to secure QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-tls-29, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-29

[9] "Usage of QUIC in the world," https://w3techs.com/technologies/details/ce-quic, accessed: 2021-02-27.

[10] S. M. Bellovin, "Security problems in the TCP/IP protocol suite," ACM SIGCOMM Computer Communication Review, vol. 19, no. 2, pp. 32–48, 1989.

[11] M. Piraux, Q. De Coninck, and O. Bonaventure, "Observing the evolution of QUIC implementations," Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, Dec 2018. [Online]. Available: http://dx.doi.org/10.1145/3284850.3284852

[12] J. L. a. Marten Seemann, Lars Eggert, "Interop test runner," Oct. 2019. [Online]. Available: https://github.com/marten-seemann/quic-interop-runner

[13] J. Iyengar and I. Swett, "Sender control of acknowledgement delays in QUIC," Internet Engineering Task Force, Internet-Draft draft-iyengar-quic-delayed-ack-02, Nov. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-iyengar-quic-delayed-ack-02

[14] S. Dickinson and I. Sinodun, "Network working group c. huitema internet-draft private octopus inc. intended status: Standards track a. mankin expires: August 26, 2021 salesforce," 2021.

[15] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," Computer Networks, vol. 150, pp. 28–45, 2019.

[16] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar et al., "The QUIC transport protocol: Design and internet-scale deployment," in Proceedings of the conference of the ACM special interest group on data communication, 2017, pp. 183–196.

[17] K. Nepomuceno, I. N. de Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó, "QUIC and TCP: a performance evaluation," in 2018 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2018, pp. 00 045–00 051.

[18] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui, "QUIC: Better for what and for whom?" in 2017 IEEE International Conference on Communications (ICC). IEEE, 2017, pp. 1–6.

[19] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz, "Multipath QUIC: A deployable multipath transport protocol," in 2018 IEEE International Conference on Communications (ICC). IEEE, 2018, pp. 1–7.

[20] F. Michel, Q. De Coninck, and O. Bonaventure, "Quic-fec: Bringing the benefits of forward erasure correction to QUIC," in 2019 IFIP Networking Conference (IFIP Networking). IEEE, 2019, pp. 1–9.

[21] M. Piraux, O. Bonaventure, Q. DE CONINCK, and M. LOBELLE, "A test suite for QUIC," Ph.D. dissertation, Master's thesis. Ecole polytechnique de Louvain, Université catholique de . . . , 2018.

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[23] A. P. M. S. S. S. Kennet McMillan, Oded Padon, "Ivy: The language." [Online]. Available: http://microsoft.github.io/ivy/language.html

[24] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-18, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-18

[25] K. L. . a. McMillan, "Ivy," Oct. 2019. [Online]. Available: https://github.com/microsoft/ivy

[26] P. Megyesi, Z. Krämer, and S. Molnár, "How quick is QUIC?" in 2016 IEEE International Conference on Communications (ICC), 2016, pp. 1–6.

[27] Y. Elkhatib, G. Tyson, and M. Welzl, "Can spdy really make the web faster?" in 2014 IFIP Networking Conference. IEEE, 2014, pp. 1–9.

[28] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is {SPDY}?" in 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), 2014, pp. 387–399.

[29] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A performance perspective on web optimized protocol stacks: TCP+TLS+HTTP/2 vs. QUIC," in Proceedings of the Applied Networking Research Workshop, 2019, pp. 1–7.

[30] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A first look at QUIC in the wild," in International Conference on Passive and Active Network Measurement. Springer, 2018, pp. 255–268.

[31] J. Iyengar and M. Thomson, "QUIC frames," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-29, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-29#section-22.2

[32] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating transport with QUIC: Design approaches and research challenges," IEEE Internet Computing, vol. 21, no. 2, pp. 72–76, 2017.

[33] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 470–485.

[34] D. Giannakopoulou, C. Păsăreanu, and C. Blundell, "Assume-guarantee testing for software components," IET Software, vol. 2, no. 6, pp. 547–562, 2008.

[35] B. Alpern and F. B. Schneider, "Defining liveness," Information processing letters, vol. 21, no. 4, pp. 181–185, 1985.

[36] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no. 3, pp. 455–495, 1982.

[37] A. Shapiro, "Monte carlo sampling methods," Handbooks in operations research and management science, vol. 10, pp. 353–425, 2003.

[38] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in Computer Aided Verification, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 190–202.

[39] O. McMillan Kenneth L, Padon, "Deductive verification in decidable fragments with Ivy," in International Static Analysis Symposium. Springer, 2018, pp. 43–55.

[40] M. Piraux, Q. De Coninck, and O. Bonaventure, "Observing the evolution of QUIC implementations," in Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, 2018, pp. 8–14.

[41] "h3spec is a tool to test error cases of QUIC and HTTP/3." [Online]. Available: https://github.com/kazu-yamamoto/h3spec

[42] J. Zhang, L. Yang, X. Gao, G. Tang, J. Zhang, and Q. Wang, "Formal analysis of QUIC handshake protocol using symbolic model checking," IEEE Access, vol. 9, pp. 14 836–14 848, 2021.

[43] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in 2017 IEEE Symposium on Security and Privacy (SP). IEEE, 2017, pp. 483–502.

[44] F. Rath, D. Schemmel, and K. Wehrle, "Interoperability-guided testing of QUIC implementations using symbolic execution," in Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, 2018, pp. 15–21.

[45] A. Ferrieux, I. Hamchaoui, I. Lubashev, and D. Tikhonov, "Packet loss signaling for encrypted protocols," Internet Engineering Task Force, Internet-Draft draft-ferrieuxhamchaoui-quic-lossbits-03, Jan. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ferrieuxhamchaoui-quic-lossbits-03

[46] M. Thomson, "Greasing the QUIC bit," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-bit-grease-00, Apr. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-bit-grease-00

[47] Q. D. Coninck and O. Bonaventure, "Multipath extensions for QUIC (MP-QUIC)," Internet Engineering Task Force, Internet-Draft draft-deconinck-quic-multipath-07, May 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-deconinck-quic-multipath-07

[48] [Online]. Available: https://github.com/ElNiak/QUIC-Ivy/tree/quic_29/doc/examples/quic

[49] QUICwg, "quicwg/base-drafts." [Online]. Available: https://github.com/quicwg/base-drafts/wiki/Implementations

[50] D. Ochtman, "A QUIC future in rust," https://paris.rustfest.eu/sessions/a-quic-future-in-rust, (Accessed on 05/22/2021).

[51] B. K. . a. Christian Huitema, steschu77, "picoquic," Jun. 2017. [Online]. Available: https://github.com/private-octopus/picoquic/tree/ad23e6c3593bd987dcd8d74fc9f528f2676fedf4

[52] L. T. Inc, "lsquic," Sep. 2017. [Online]. Available: https://github.com/litespeedtech/lsquic/tree/v2.29.4

[53] s. B. K. . a. Kazuho Oku, Christian Huitema, "picotls," Sep. 2016. [Online]. Available: https://github.com/h2o/picotls/tree/47327f8d032f6bc2093a15c32e666ab6384ecca2

[54] "boringssl." [Online]. Available: https://boringssl.googlesource.com/boringssl/

[55] L. C. . a. Marten Seemann, "quic-go," Apr. 2016. [Online]. Available: https://github.com/lucas-clemente/quic-go

[56] J.-C. B. . a. Benjamin Saunders, Dirkjan Ochtman, "quinn," Feb. 2018. [Online]. Available: https://github.com/quinn-rs/quinn/tree/0.7.0

[57] J. Lainé, "aioquic," Feb. 2019. [Online]. Available: https://github.com/aiortc/aioquic/tree/0.9.3

[58] "quiche." [Online]. Available: https://github.com/cloudflare/quiche

[59] L. Eggert, "quant," Sep. 2016. [Online]. Available: https://github.com/NTAP/quant/tree/29

[60] Facebook, "mvfst," Apr. 2019. [Online]. Available: https://github.com/facebookincubator/mvfst

[61] C. Huitema, "Christian huitema," http://www.huitema.net/fr-home.asp, (Accessed on 05/22/2021).

[62] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," arXiv preprint arXiv:1910.04054, 2019.

[63] L. Eggert, "Towards securing the internet of things with QUIC," in Workshop on Decentralized IoT Systems and Security (DISS). https://doi.org/10.14722/diss, 2020.

[64] E. Lars, "Lars Eggert website," https://www.eggert.org/, (Accessed on 05/22/2021).

[65] L. Clemente and M. Seemann, "QUIC-go," 2019.

[66] M. Masquelin, "Caddy et traefik: des concurrents sérieux à nginx et haproxy?" in 15ème journée thématique MIn2RIEN:" Retours d'expériences", 2018.

[67] O. Bonaventure, The Transmission Control Protocol, 2019, pp. 1–272.

[68] "ACM SIGCOMM 2021." [Online]. Available: https://conferences.sigcomm.org/sigcomm/2021/

[69] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, pp. 1–33, 2017.

[70] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," IEEE/ACM Transactions on Networking, vol. 24, no. 2, pp. 887–900, 2015.

[71] "What is ebpf? an introduction and deep dive into the ebpf technology." [Online]. Available: https://ebpf.io/what-is-ebpf#what-is-ebpf

[72] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 1069–1084.

[73] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and evaluation," in Proceedings of the 13th international conference on emerging networking experiments and technologies, 2017, pp. 160–166.

[74] T. Pauly, E. Kinnear, and D. Schinazi, "An unreliable datagram extension to QUIC," Internet Engineering Task Force.(September 2018). draft-pauly-quicdatagram-00, 2018.

[75] W. A. Fagen, J. W. Cangussu, and R. Dantu, "A virtual environment for network testing," Journal of network and computer applications, vol. 32, no. 1, pp. 184–214, 2009.

[76] Microsoft, "Z3." [Online]. Available: https://github.com/Z3Prover/z3

[77] Linas, "Propositional calculus," 2018. [Online]. Available: https://en.wikipedia.org/wiki/Propositional_calculus

[78] Z. Rakamarić, "Propositional Logic & SAT," 2014. [Online]. Available: https://utah.instructure.com/courses/362531/files/folder/lectures?preview=53847316

[79] A. P. M. S. S. S. Kenneth McMillan, Oded Padon, "Ivy: Decidability." [Online]. Available: http://microsoft.github.io/ivy/decidability.html

[80] "Classical Program Logics: Hoare Logic, Weakest Liberal Preconditions." [Online]. Available: https://www.lri.fr/~marche/MPRI-2-36-1/2013/poly1.pdf

[81] T. King, "Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic," 2014. [Online]. Available: https://cs.nyu.edu/media/publications/king_tim.pdf

[82] J. Cito, V. Ferme, and H. C. Gall, "Using docker containers to improve reproducibility in software and web engineering research," in International Conference on Web Engineering. Springer, 2016, pp. 609–612.

[83] "Toward verification of QUIC extensions." [Online]. Available: https://github.com/ElNiak/Toward-verification-of-QUIC-extensions/tree/master/installer/TVOQE

Appendix A

Results tables

In the following part, we present you the detailed version of the errors we found. Since we launch 100 iterations per test, we give the number of time each error appears. This allow to see the error frequency.

After analysing the data, we conclude that we should not consider the ratio of test that passed with those that failed. This would not be very convenient in this situation since some features are not well implemented for some implementation such as the migration. It can cause high differences in term of results. For example with `aioquic`, when we disable the migration, the implementation have much better score if we only consider the passed test. So this is not relevant.

It is more important to consider the type of error and their distribution. Also, some errors are linked between each other. For example, we know that some errors that we have are only linked to the migration. If an implementation correct this part, it will remove multiple errors at once.

A.1 aioquic

A.1.1 Server

Table A.1: aioquic server errors

Test	Error code	#	
stream	frame.connection_close:{err_code:0xa}		50
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		30
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		2
			82.0/100
unkown	frame.connection_close:{err_code:0xa}		100
			100.0/100
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;		84
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		9
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		7
			100.0/100
token_error			**0.0/100**
max	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		3
	frame.connection_close:{err_code:0xa}		42
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		28
			73.0/100
tp_error	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		14
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		11
	require is_transport_parameter_error;		75
			100.0/100
tp_acticoid_error	No decryption keys		100
			100.0/100
connection_close	require is_no_error		13
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		9
			22.0/100
reset_stream	frame.connection_close:{err_code:0xa}		52
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		22
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		2
			76.0/100
no_icid	require is_transport_parameter_error;		99
	ivy_return_code(134)		1
			100.0/100
accept_maxdata	frame.connection_close:{err_code:0xa}		26
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		26
	require conn_total_data(the_cid) >0;		2
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		3
			57.0/100
retirecoid_error	require is_protocol_violation		56
	ivy_return_code(134)		5
	require connected(dcid) & connected_to(dcid) = scid;		2
	frame.connection_close:{err_code:0xa}		1
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		2
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		1
			67.0/100
ext_min_ack_delay	frame.connection_close:{err_code:0xa}		37
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		4
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		33
			74.0/100
handshake_done_error	ivy_return_code(139)		5
	require is_protocol_violation		9
			14.0/100
stop_sending	frame.connection_close:{err_code:0xa}		34
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		28
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		5
			67.0/100

Test	Error code	#
double_tp_error	require is_transport_parameter_error | ~handshake_done_send;	78
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	18
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]	4
		100.0/100
newcoid_rtp_error	require connected(dcid) & connected_to(dcid) = scid;	89
	require is_frame_encoding_error;	11
		100.0/100
max_limit_error	frame.connection_close:{err_code:0x5}	51
		51.0/100
retirecoid_error	require is_protocol_violation;	100
		100.0/100
newconnectionid_error	ivy_return_code(134)	28
	require conn_total_data(the_cid) >0;	4
		32.0/100
new_token_error		**0.0/100**
newcoid_length_error	require is_frame_encoding_error;	77
	ivy_return_code(134)	23
		100.0/100
unkown_tp	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	32
	bind failed: Address already in use	1
		33.0/100
max_error	require is_stream_limit_error;	100
		100.0/100
stream_limit_error	require stream_id_allowed(dcid,f.id); [6]	1
		1.0/100

A.1.2 Client

Table A.2: aioquic client errors

Test	Error code	#	
new_token_error	frame.connection_close:{err_code:0xa}		87
	frame.connection_close:{err_code:0x5}		2
	require is_frame_encoding_error;		10
			99.0/100
tp_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	31
	require is_frame_encoding_error;		69
			100.0/100
retirecoid_error	require is_frame_encoding_error;		100
			100.0/100
no_odci	client_return_code(1)		100
			100.0/100
stream	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	3
			3.0/100
tp_prefadd_error	require is_frame_encoding_error;		100
			100.0/100
max	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	2
			2.0/100
ext_min_ack_delay			**0.0/100**
unkown	frame.connection_close:{err_code:0xa}		71
	require is_frame_encoding_error;		29
			100.0/100
tp_unkown	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			1.0/100
limit_max_error	frame.connection_close:{err_code:0x5}		93
	require is_stream_limit_error;		7
			100.0/100
tp_acticoid_error	require is_transport_parameter_error;		100
			100.0/100
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;		100
			100.0/100
double_tp_error	require is_transport_parameter_error;		100
			100.0/100
accept_maxdata	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	3
			3.0/100
handshake_done_error	require is_protocol_violation;		31
			31.0/100

A.2 quinn

A.2.1 Server

Table A.3: quinn v0.7.0 server test errors

Test	Error code	#	
stream	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		7
	require conn_total_data(the_cid) >0;		13
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		1
			21.0/100
unkown	require is_frame_encoding_error;		2
			2.0/100
max	require conn_total_data(the_cid) >0;		9
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		4
	require stream_id_allowed(dcid,f.id); [6]		2
			15.0/100
connection_close	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		3
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		2
			5.0/100
reset_stream	require stream_id_allowed(dcid,f.id); [6]		57
	require conn_total_data(the_cid) >0;		12
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		2
			71.0/100
accept_maxdata	require conn_total_data(the_cid) >0;		16
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		5
	require stream_id_allowed(dcid,f.id); [6]		2
			23.0/100
ext_min_ack_delay	require conn_total_data(the_cid) >0;		20
			20.0/100
double_tp_error			**0.0/100**
handshake_done_error	Timeout		1
			1.0/100
stop_sending			**0.0/100**
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;		30
			30.0/100
tp_limit_newcoid	require conn_total_data(the_cid) >0;		20
			20.0/100
token_error			**0.0/100**
tp_acticoid_error			**0.0/100**
tp_error			**0.0/100**
no_icid			**0.0/100**
newcoid_rtp_error	bind failed: Address already in use		9
			9.0/100
max_limit_error	require is_stream_limit_error;		100
			100.0/100
retirecoid_error	require is_protocol_violation;		13
			13.0/100
new_token_error			**0.0/100**

newcoid_length_error	bind failed: Address already in use	16
		16.0/100
unkown_tp	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	12
	bind failed: Address already in use	4
		16.0/100
max_error	frame.connection_close:{err_code:0x7}	100
		100.0/100
newconnectionid_error	require conn_total_data(the_cid) >0;	8
	server_return_code(-4)	11
		19.0/100

Table A.4: quinn v0.7.0 server test errors without migration

Test	**Error code**	**#**
stream		**0/100**
max	Handshake not completed	1
	require stream_id_allowed(dcid,f.id);	2
		3.0/100
reset_stream	Handshake not completed	57
	Handshake not completed	9
		66.0/100
connection_close		**0.0/100**
unkown	Handshake not completed	4
		4.0/100
tp_error		**0.0/100**
tp_acticoid_error		**0.0/100**
token_error		**0.0/100**
no_icid		**0.0/100**
handshake_done_error	ivy_return_code(134)	**1.0/100**
double_tp_error		**0.0/100**
blocked_streams_maxstream_error	Handshake not completed	30
		30.0/100
accept_maxdata	Handshake not completed	7
	require stream_id_allowed(dcid,f.id);	8
	require conn_total_data(the_cid) >0;	2
		17.0/100
ext_min_ack_delay	Handshake not completed	22
		22.0/100

A.2.2 Client

Table A.5: quinn client errors

Test	Error code	#
new_token_error	frame.application_close:{err_code:0}	2
		2.0/100
tp_error	client_return_code(1)	100
		100.0/100
retirecoid_error	frame.application_close:{err_code:0}	1
		1.0/100
no_odci	client_return_code(1)	100
		100.0/100
stream	Timeout	1
		1.0/100
tp_prefadd_error	client_return_code(1)	100
		100.0/100
max		**0.0/100**
ext_min_ack_delay		**0.0/100**
unkown		**0.0/100**
tp_unkown		**0.0/100**
limit_max_error	frame.connection_close:{err_code:0x5}	100
		100.0/100
tp_acticoid_error	client_return_code(1)	100
		100.0/100
blocked_streams_maxstream_error	frame.application_close:{err_code:0}	1
		1.0/100
double_tp_error	client_return_code(1)	100
		100.0/100
accept_maxdata		**0.0/100**
handshake_done_error	frame.connection_close:{err_code:0xc}	84
	require is_protocol_violation;	13
		97.0/100

A.3 mvfst

A.3.1 Server

Table A.6: mvfst server errors

Test	Error code	#	
stream	frame.connection_close:{err_code:0xc}		62
	require conn_total_data(the_cid) >0;		25
	ivy_return_code(134)		7
			94.0/100
unkown	frame.connection_close:{err_code:0xc}		1
			1.0/100
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;		63
	ivy_return_code(134)		3
	frame.connection_close:{err_code:0xc}		34
			100.0/100
token_error	frame.connection_close:{err_code:0x1}		2
			2.0/100
max	frame.connection_close:{err_code:0xc}		76
	require conn_total_data(the_cid) >0;		20
	ivy_return_code(134)		1
			97.0/100
tp_error			**0.0/100**
tp_acticoid_error	frame.connection_close:{err_code:0xc}		69
	require is_transport_parameter_error;		28
	ivy_return_code(134)		3
			100.0/100
connection_close	require is_no_error		59
	ivy_return_code(134)		1
	require connected(dcid) & connected_to(dcid) = scid;		3
			63.0/100
reset_stream	frame.connection_close:{err_code:0xc}		69
	require conn_total_data(the_cid) >0;		21
	ivy_return_code(134)		3
			93.0/100
no_icid			**0.0/100**
accept_maxdata	frame.connection_close:{err_code:0xc}		65
	require conn_total_data(the_cid) >0;		22
	ivy_return_code(134)		1
			88.0/100
retirecoid_error	frame.connection_close:{err_code:0xc}		70
	ivy_return_code(134)		5
	require is_protocol_violation		22
			97.0/100
ext_min_ack_delay	require conn_total_data(the_cid) >0;		21
	frame.connection_close:{err_code:0xc}		64
	ivy_return_code(134)		5
			90.0/100

handshake_done_error	require is_protocol_violation;	3
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	5
		8.0/100
stop_sending	require conn_total_data(the_cid) >0;	32
	frame.connection_close:{err_code:0xc}	61
	ivy_return_code(134)	3
		96.0/100
double_tp_error	frame.connection_close:{err_code:0xc}	35
	require is_transport_parameter_error | ~handshake_done_send;	65
		100.0/100
newcoid_rtp_error	frame.connection_close:{err_code:0xa}	87
	require is_frame_encoding_error;	9
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
		100.0/100
max_limit_error	frame.connection_close:{err_code:0x5}	49
	assumption_failed(quic_server_test_max_limit_error.ivy: line 542)	5
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	1
	ivy_return_code(134)	4
		59.0/100
retirecoid_error	require is_protocol_violation;	91
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
	frame.connection_close:{err_code:0x1}	5
		100.0/100
newconnectionid_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	3
	require conn_total_data(the_cid) >0;	12
		15.0/100
new_token_error	require is_protocol_violation;	97
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	3
		100.0/100
newcoid_length_error	frame.connection_close:{err_code:0xa}	73
	require is_frame_encoding_error;	25
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	2
		100.0/100
unkown_tp	require conn_total_data(the_cid) >0;	35
	No decryption keys	1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	5
		41.0/100
max_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
	require is_stream_limit_error;	5
	ivy_return_code(134)	1
		10.0/100
stream_limit_error	require is_flow_control_error | is_frame_encoding_error;	32
	require stream_id_allowed(dcid,f.id); [6]	1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
		37.0/100

A.4 picoquic

A.4.1 Server

Table A.7: picoquic server errors

Test	Error code	#	
stream	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	44
			44.0/100
unkown	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	1
			1.0/100
blocked_streams_maxstream_error	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	13
	require is_frame_encoding_error | is_stream_limit_error;		87
			100.0/100
token_error			**0.0/100**
max	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	52
	require conn_total_data(the_cid) >0;		1
			53.0/100
tp_error	require is_transport_parameter_error;		81
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	19
			100.0/100
tp_acticoid_error	require is_transport_parameter_error;		100
			100.0/100
connection_close	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	19
			19.0/100
reset_stream	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	38
	require conn_total_data(the_cid) >0;		1
			39.0/100
no_icid			**0.0/100**
accept_maxdata	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	42
	require conn_total_data(the_cid) >0;		8
			50.0/100
retirecoid_error	frame.connection_close:{err_code:0xa}		39
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	9
			48.0/100
ext_min_ack_delay	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	38
	require f.seq_num >last_ack_freq_seq(scid);		23
	require conn_total_data(the_cid) >0;		1
			62.0/100
handshake_done_error	ivy_return_code(139)		4
	require is_protocol_violation		7
			11.0/100
stop_sending	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	52
			52.0/100
double_tp_error			**0.0/100**
newcoid_rtp_error	frame.connection_close:{err_code:0xa}		100
			100.0/100
max_limit_error			**0.0/100**
retirecoid_error	require is_protocol_violation;		14
			14.0/100

newconnectionid_error		**0.0/100**
new_token_error	require is_protocol_violation;	100 **100.0/100**
newcoid_length_error	frame.connection_close:{err_code:0xa}	98 **98.0/100**
unkown_tp	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	2 **2.0/100**
max_error		**0.0/100**
stream_limit_error	require stream_id_allowed(dcid,f.id); [6]	1 **1.0/100**

A.4.2 Client

Table A.8: picoquic client errors

Test	Error code	#	
new_token_error	frame.connection_close:{err_code:0x4}		4
	frame.connection_close:{err_code:0xa}		1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			6.0/100
tp_error	Connection end with local error 0x434		45
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	51
	require is_frame_encoding_error;		4
			100.0/100
retirecoid_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			1.0/100
no_odci			**0.0/100**
stream	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	16
	Connection end with local error 0x434		33
			49.0/100
tp_prefadd_error			**0.0/100**
max	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	20
	Connection end with local error 0x434		65
			85.0/100
ext_min_ack_delay	require f.seq_num >last_ack_freq_seq(scid);		19
	Connection end with local error 0x434		40
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			60.0/100
unkown	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	4
			4.0/100
tp_unkown	Connection end with local error 0x434		64
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	2
			66.0/100
limit_max_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	12
			12.0/100
tp_acticoid_error	require is_transport_parameter_error;		43
	Connection end with local error 0x434		55
	frame.application_close:{err_code:0}		1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			100.0/100
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;		79
	Connection end with local error 0x434		16
	require connected(dcid) & connected_to(dcid) = scid;		4
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			100.0/100
double_tp_error			**0.0/100**
accept_maxdata	require connected(dcid) & connected_to(dcid) = scid;		4
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	2
	require conn_total_data(the_cid) >0;		1
			7.0/100
handshake_done_error	require is_protocol_violation;		14
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
			15.0/100

A.5 quic-go

A.5.1 Server

Table A.9: quic-go server errors

Test	Error code	#	
stream	require f.fin <-> (stream_app_data_finished(dcid,f.id)		**5.0/100**
unkown	require is_frame_encoding_error;	2	**2.0/100**
max	require f.fin <-> (stream_app_data_finished(dcid,f.id)	2	
	require conn_total_data(the_cid) >0;	59	**61.0/100**
connection_close	server_return_code(1)	37	**37.0/100**
reset_stream			**0.0/100**
accept_maxdata	require f.fin <-> (stream_app_data_finished(dcid,f.id)	4	
	require conn_total_data(the_cid) >0;	28	**32.0/100**
ext_min_ack_delay			**0.0/100**
test_blocked_streams_maxstream_error	Handshake not completed		**100/100**
double_tp_error			**0.0/100**
handshake_done_error	frame.connection_close:{err_code:0xa}	100	**100.0/100**
stop_sending	server_return_code(1)	100	**100.0/100**
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;	25	**25.0/100**
tp_limit_newcoid			**0.0/100**
token_error			**0.0/100**
tp_acticoid_error	require is_transport_parameter_error;	100	**100.0/100**
tp_error			**0.0/100**
no_icid			**0.0/100**
newconnectionid_error	ivy_return_code(134)	84	
		6	
	server_return_code(1)	1	**91.0/100**
stream_limit_error	require stream_id_allowed(dcid,f.id); [6]	2	**2.0/100**

retirecoid_error	require is_protocol_violation;	15
		15.0/100
max_error	frame.connection_close:{err_code:0x7}	90
	require is_stream_limit_error;	10
		100.0/100
new_token_error	require is_protocol_violation;	16
		16.0/100
unkown_tp		
		0.0/100
newcoid_rtp_error	require is_frame_encoding_error;	10
		10.0/100
max_limit_error	frame.connection_close:{err_code:0x5}	100
		100.0/100
newcoid_length_error	bind failed: Address already in use	12
	server_return_code(2)	5
	sending id: client addr: 2130706433 port: 4443+< prot.show_enc_level(0)	2
		19.0/100

A.5.2 Client

Table A.10: quic-go client errors

Test	Error code	#
new_token_error	frame.application_close:{err_code:0}	3
		3.0/78
tp_error		**0.0/70**
retirecoid_error		**0.0/71**
no_odci		**0.0/70**
stream		**0.0/60**
tp_prefadd_error	client_return_code(1)	75
		75.0/75
max		**0.0/66**
ext_min_ack_delay	require conn_total_data(the_cid) >0;	1
		1.0/69
unkown	require client_return_code(1)	1
		1.0/71
tp_unkown	frame.connection_close:{err_code:0xc}	1
		1.0/71
limit_max_error	frame.connection_close:{err_code:0x5}	64
	client_return_code(1)	2
		66.0/66
tp_acticoid_error	frame.application_close:{err_code:0}	66
	require is_transport_parameter_error;	1
		67.0/67
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;	2
		2.0/76
double_tp_error		**0.0/78**
accept_maxdata		**0.0/71**
handshake_done_error	frame.connection_close:{err_code:0x7}	52
	require is_protocol_violation;	2
	client_return_code(1)	8
		62.0/62

A.6 quant

A.6.1 Server

Table A.11: quant server errors

Test	Error code	#	
stream	require ~path_challenge_pending(dcid,f.data);		45
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		36
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		7
			88.0/100
unkown	require is_frame_encoding_error;		94
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		1
	ivy_return_code(134)		5
			100.0/100
blocked_streams_maxstream_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		25
	require ~path_challenge_pending(dcid,f.data);		33
	require is_frame_encoding_error | is_stream_limit_error;		32
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		10
			100.0/100
token_error			**0.0/100**
max	require ~path_challenge_pending(dcid,f.data);		47
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		30
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		2
			79.0/100
tp_error	server_return_code(1)		100
			100.0/100
tp_acticoid_error			**0.0/100**
connection_close	require ~path_challenge_pending(dcid,f.data);		20
	ivy_return_code(134)		12
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		20
	require ~draining_pkt_sent(scid) & queued_close(scid);		6
	require ~conn_closed(scid); [8]		1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		1
			60.0/100
reset_stream	require ~path_challenge_pending(dcid,f.data);		55
	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		39
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		1
			95.0/100
no_icid	server_return_code(1)		100
			100.0/100
accept_maxdata	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		18
	require ~path_challenge_pending(dcid,f.data);		59
	require conn_total_data(the_cid) >0;		1
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		1
			79.0/100
retirecoid_error	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		7
	require is_protocol_violation		85
	require ~path_challenge_pending(dcid,f.data);		6
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]		1
	frame.connection_close:{err_code:0x9}		1
			100.0/100
ext_min_ack_delay	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]		40
	require ~path_challenge_pending(dcid,f.data);		54
			94.0/100
handshake_done_error	require is_protocol_violation		98
			98.0/100

stop_sending	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst); [10]	42
	require ~path_challenge_pending(dcid,f.data);	45
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	5
		92.0/100
double_tp_error	server_return_code(1)	100
		100.0/100
newcoid_rtp_error	frame.connection_close:{err_code:0xa}	58
	require is_frame_encoding_error;	42
		100.0/100
max_limit_error	frame.connection_close:{err_code:0x5}	100
		100.0/100
retirecoid_error	require is_protocol_violation;	100
		100.0/100
newconnectionid_error	server_return_code(-6)	6
	require conn_total_data(the_cid) >0;	4
		7.0/100
new_token_error	require is_protocol_violation;	61
	frame.connection_close:{err_code:0x7}	39
		100.0/100
newcoid_length_error	frame.connection_close:{err_code:0x9}	42
	require is_frame_encoding_error;	58
		100.0/100
unkown_tp		**0.0/100**
max_error	require is_stream_limit_error;	100
		100.0/100
stream_limit_error	ivy_return_code(134)	19
	require is_flow_control_error | is_frame_encoding_error;	71
		90.0/100

A.6.2 Client

Table A.12: quant client errors

Test	Error code	#
tp_unkown		**0.0/74**
retirecoid_error	require is_frame_encoding_error;	63
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	1
		64.0/64
double_tp_error	client_return_code(1)	63
		63.0/63
tp_acticoid_error		**0.0/69**
tp_error	client_return_code(1)	68
	ivy_return_code(134)	1
		69.0/69
unkown	require is_frame_encoding_error;	60
	ivy_return_code(134)	14
		74.0/74
stream	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	2
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	6
	require conn_total_data(the_cid) >0;	2
		10.0/70
max	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	7
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	2
	require connected(dcid) & connected_to(dcid) = scid;	1
		11.0/72
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;	73
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	3
		76.0/76
tp_prefadd_error	require is_frame_encoding_error;	68
		68.0/68
ext_min_ack_delay		**0.0/75**
accept_maxdata	require connected(dcid) & connected_to(dcid) = scid;	2
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	1
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	1
		4.0/74
new_token_error	ivy_return_code(134)	7
	require is_frame_encoding_error;	65
		71.0/71
limit_max_error	client_return_code(1)	9
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;	1
		10.0/53
no_odci	client_return_code(1)	73
		73.0/73
handshake_done_error	require is_protocol_violation;	76
	client_return_code(1)	1
		77.0/80

A.7 quiche

A.7.1 Server

Table A.13: quiche server errors

Test	Error code	#
stream	No error	97
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	2
	Timeout	1
		3.0/100
unkown		**0.0/100**
blocked_streams_maxstream_error		**0.0/100**
token_error		**0.0/100**
tp_limit_newcoid	No error	93
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	7
		7.0/100
max	No error	96
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
		4.0/100
tp_error	Timeout	100
		100.0/100
tp_acticoid_error	Timeout	100
		100.0/100
connection_close		**0.0/100**
reset_stream	No error	98
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	2
		2.0/100
no_icid	Timeout	100
		100.0/100
accept_maxdata	No error	96
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
		4.0/100
retirecoid_error	require is_protocol_violation	97
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	3
		100.0/100
ext_min_ack_delay	No error	98
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	2
		2.0/100
handshake_done_error	ivy_return_code(139)	12
	require is_protocol_violation	11
		23.0/100
stop_sending	No error	96
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
		4.0/100
double_tp_error		**0.0/100**
newcoid_rtp_error	require is_frame_encoding_error;	94
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	6
		100.0/100
max_limit_error	require is_stream_limit_error;	95
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	5
		100.0/100

retirecoid_error	require is_protocol_violation;	92
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	7
	Timeout	1
		100.0/100
newconnectionid_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	9
		9/100
new_token_error	require is_protocol_violation;	94
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	6
		100.0/100
newcoid_length_error	require is_frame_encoding_error;	94
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	6
		100.0/100
unkown_tp	No error	96
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	3
	Timeout	1
		4.0/100
max_error	frame.connection_close:{err_code:0x7}	100
		100.0/100
stream_limit_error	Timeout	1
	require stream_id_allowed(dcid,f.id); [6]	1
		2.0/100

A.7.2 Client

Table A.14: quiche client errors

Test	Error code	#
new_token_error	frame.connection_close:{err_code:0x5}	2
	require is_frame_encoding_error;	10
	frame.connection_close:{err_code:0xa}	1
		13.0/100
tp_error	require is_frame_encoding_error;	100
		100.0/100
retirecoid_error	require is_frame_encoding_error;	100
		100.0/100
no_odci	Timeout	100
		100.0/100
stream	Timeout	32
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	16
		48.0/100
tp_prefadd_error	require is_frame_encoding_error;	100
		100.0/100
max	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	4
	require stream_id_allowed(dcid,f.id); [6]	62
		66.0/100
ext_min_ack_delay		**0.0/100**
unkown		**0.0/100**
tp_unkown	require conn_total_data(the_cid) >0;	1
		1.0/100
limit_max_error	frame.connection_close:{err_code:0x5}	94
	require is_stream_limit_error;	5
	Timeout	1
		100.0/100
tp_acticoid_error	require is_transport_parameter_error;	100
		100.0/100
blocked_streams_maxstream_error	require is_frame_encoding_error | is_stream_limit_error;	9
		9.0/100
double_tp_error	require is_transport_parameter_error;	99
	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	1
		100.0/100
accept_maxdata	require stream_id_allowed(dcid,f.id); [6]	18
		18.0/100
handshake_done_error	require connected(dcid) & connected_to(dcid) = scid;	20
	Timeout	10
	require is_protocol_violation;	1
		31.0/100

A.8 lsquic

A.8.1 Server

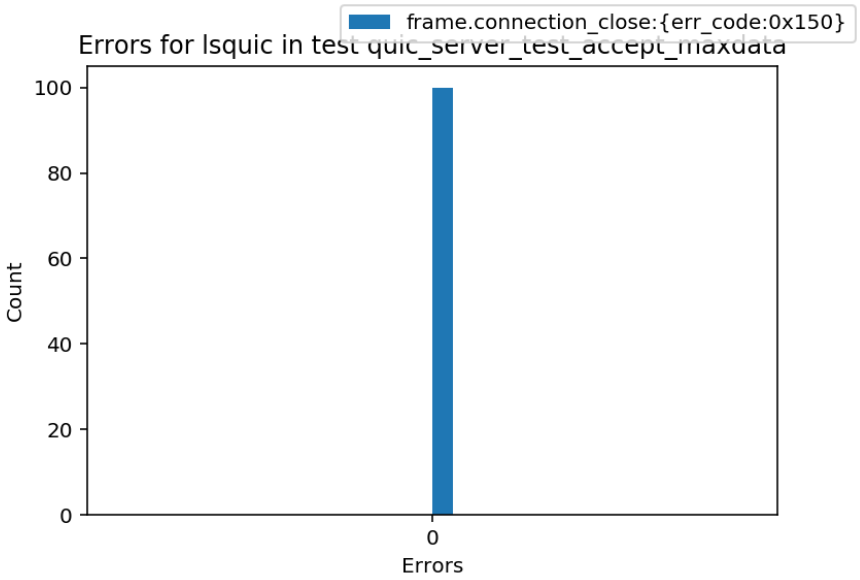


Figure A.1: lsquic server errors

A.8.2 Client

Table A.15: `lsquic` client errors

Test	Error code	#	
new_token_error			**0.0/100**
tp_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	89
	frame.connection_close:{err_code:0}		6
	require is_frame_encoding_error;		3
	frame.connection_close:{err_code:0x1}		2
			100.0/100
retirecoid_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	2
			2.0/100
no_odci	require is_transport_parameter_error;		100
			100.0/100
stream	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	6
	require connected(dcid) & connected_to(dcid) = scid;		2
			8.0/100
tp_prefadd_error	require is_frame_encoding_error;		100
			100.0/100
max			**0.0/100**
ext_min_ack_delay	ivy_return_code(134)		4
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;		1
			5.0/100
unkown	frame.connection_close:{err_code:0x1}		100
			100.0/100
tp_unkown	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	2
	require connected(dcid) & connected_to(dcid) = scid;		1
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;		1
			4.0/100
limit_max_error	frame.connection_close:{err_code:0x5}		99
	require is_stream_limit_error;		1
			100.0/100
tp_acticoid_error	require is_transport_parameter_error;		100
			100.0/100
blocked_streams_maxstream_error	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	1
	require is_frame_encoding_error | is_stream_limit_error;		1
			2.0/100
double_tp_error	require is_transport_parameter_error;		100
			100.0/100
accept_maxdata	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0;	[5]	9
	require connected(dcid) & connected_to(dcid) = scid;		6
	require pkt.long & connected(dcid) ->connected_to(dcid) = scid;		2
			17.0/100
handshake_done_error	require is_protocol_violation;		98
	frame.connection_close:{err_code:0x1}		2
			100.0/100

Appendix B

Ivy concepts used for QUIC

Type object, Action and State

In this example, we can already see the most used concepts of Ivy for testing QUIC protocols. First, Ivy uses the concept of "**type objects**" which have a "**state** which can be manipulated through "**action**".

State example

```
object frame = {
    type this # The base type for frames
    object stream = {  # Stream frames
        # Stream frames are a variant of frame
        variant this of frame = struct {
            off : bool, # is there an offset field ?
            len : bool, # is there a length field ?
            fin : bool, # is this the final offset ?
            id   : stream_id,     # the stream ID
            offset : stream_pos,# the stream offset
            length : stream_pos,# length of the data
            data : stream_data  # the stream data
        }
    }
}
```

Action example

```
object frame = { ...        # Action associated to frame
    action handle(f:this, # param_name:type
                  scid:cid, dcid:cid, e:quic_packet_type) = {
        require false;      # this generic action
    }
}
object frame = { ...  # Refinement of action for STREAM frame
    object stream = { ...
        action handle(f:this, scid:cid, dcid:cid,
            e:quic_packet_type) = {
            require connected(dcid) & connected_to(dcid)=scid;
            require e = quic_packet_type.one_rtt &
                established_1rtt_keys(scid);
            call enqueue_frame(scid,f,e,false);
        }
    }
}
action enqueue_frame(scid:cid, f:frame, e:quic_packet_type,
                     probing:bool) = { # Global action
        # Code
}
```

This example also shows that type objects are defined from a very abstract and generic point of view. Here, we start by defining a frame "type object", and then we define "variant" of the type object that sets the state of more "embedded" structure, here with STREAM frame, and any other type of frame that QUIC defines in the specification. The same idea is used through all layers of QUIC.

Actions are also defined from an abstract and generic point of view. They are then adapted to a specific case. Here, for the action `handle`, it is associated to frame type object and define this action for each type of frame. It can also be global such as the action `"enqueue_frame"`. The code of the action `"handle"` mostly defines a set of requirements/conditions that should be true at each step to continue the execution. This set is based on the requirements of the QUIC specification from the IETF.

Modules

Modules are like template classes in object-oriented programming that can be instantiated. For QUIC, this is used for defining a general class of objects but it can be used to capture more general theory and proof. Here is an example for the QUIC transport parameters:

Transport parameter module

```
type trans_params_struct
object transport_parameter = {
    type this
    action set(p:this,s:trans_params_struct)
    returns (s:trans_params_struct) = {}
}
module trans_params_ops(ptype) = {
    destructor is_set(S:trans_params_struct) : bool
    destructor value(S:trans_params_struct) : ptype
    action set(p:ptype,s:trans_params_struct)
    returns (s:trans_params_struct) = {
        is_set(s) := true; value(s) := p;
    }
}
```

Instantiating the module

```
object initial_max_stream_data_bidi_local = {
    variant this of transport_parameter = struct {
        stream_pos_32 : stream_pos  # tag = 0
    }
    instantiate trans_params_ops(this)
}
object initial_max_data = {
    variant this of transport_parameter = struct {
        stream_pos_32 : stream_pos # tag = 1
    }
    instantiate trans_params_ops(this)
}
```

In this example, we can see that the object can instantiate `trans_params_ops` automatically such that we can now check if a parameter is set or not. Then we can call it directly on transport parameter object such as:

Example of usage

```
initial_max_stream_data.is_set(fml:s)
initial_max_stream_data.value(fml:s)
initial_max_data.is_set(fml:s)
initial_max_data.value(fml:s)
trans_params(scid) := tps.transport_parameters
                        .value(idx)
                        .set(trans_params(scid));
```

Monitor

Before addressing in detail the specification statements, one important concept in 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.

Let us consider this example:

Monitor

```
around app_server_open_event {
    require conn_requested(dst,src,dcid);
    require ~connected(dcid) & ~connected(scid);
    ...
    call map_cids(scid,dcid);
    call map_cids(dcid,scid);
    ack_credit(scid) := ack_credit(scid) + 1;
}
```

Let us note that around is a shortcut keyword. The content above is the same as :

Equivalence 1

```
before app_server_open_event {
    require conn_requested(dst,src,dcid);
    require ~connected(dcid) & ~connected(scid);
}
after app_server_open_event {
    call map_cids(scid,dcid);
    call map_cids(dcid,scid);
    ack_credit(scid) := ack_credit(scid) + 1;
}
```

We can see 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 the fact 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 content each time an `app_server_open_event` is generated.

Here is a shortcut of this code that produces the same result :

Equivalence 2

```
action check_connection {
    require conn_requested(dst,src,dcid);
    require ~connected(dcid) & ~connected(scid);
}
action check_mapping {
    call map_cids(scid,dcid);
    call map_cids(dcid,scid);
    ack_credit(scid) := ack_credit(scid) + 1;
}
before app_server_open_event execute check_connection;
before app_server_open_event execute a2;
```

Control and expression

Here we define control flow statement such as sequential execution, calling actions, conditions and loops. Ivy also defines expressions which are represented by first order logic operator: and (&), or (|), implies (->), iff (<->) and equality (=) and negation (~). Let us consider a QUIC example of that:

CRYPTO Frame showing most operators

```
object frame = {
    ...
    object crypto = {
        ...
        action handle(f:frame.crypto, scid:cid, dcid:cid,
            e:quic_packet_type)
      around handle {
          #Implication
            require num_queued_frames(scid) > 0
                        -> e = queued_level(scid);
            #negation
            require ~conn_closed(scid);
            require ~(e = quic_packet_type.zero_rtt);
            #inequality
          require (f.offset + f.length)
                      <= crypto_data_end(scid,e);
          # /!\ equality /!\
          require f.data = crypto_data(scid,e)
              .segment(f.offset,f.offset+f.length);
          ...
          # Assigment
            var length := f.offset + f.length;
            # if statement
            if crypto_length(scid,e) < length {
                crypto_length(scid,e) := length
            };

            var idx := f.offset;
```

```
            # Loop: should not be used for initialisation
            while idx < f.offset + f.length {
                crypto_data_present(scid,e,idx) := true;
                idx := idx.next
            };
            call enqueue_frame(scid,f,e,false);
            if e = quic_packet_type.handshake {
                established_1rtt_keys(scid) := true;
            }
        }
    }
}
```

The sequential execution is defined by the ";" character. Every time we expect another requirement or action after a statement we put this character. We can call an action with the keyword "call", which executes its code.

Non-deterministic choice

Non-determinism at the QUIC transport parameters extensions

```
action handle_tls_extensions
    (src:ip.endpoint, dst:ip.endpoint,
     scid:cid, exts:vector[tls.extension],
     is_client_hello:bool) = {
    # We process the extensions in a message in order.
    var idx := exts.begin;
    while idx < exts.end {
        var ext := exts.value(idx);
        # For every `quic_transport_parameters` extension
        if some (tps:quic_transport_parameters) ext *> tps{
            call handle_client_transport_parameters(src,
                        dst,scid,tps,is_client_hello);
            trans_params_set(scid) := true;
        };
        idx := idx.next
    };
}
```

Non-deterministic choice can be represented with * and used for two situations:

1. On the right-hand side of an assignment, which means that the variable can take any possible value of the type.

2. It can be condition to create a non-deterministic branch, in the presented example this represents the fact that the extension supported by the client and the server are non-deterministic and means "if there is at least one transport parameter available in one of TLS extension proposed, then we call the action handle_client_transport_parameters" meaning that the proposed extension is non-deterministic in some way since it changes from implementation to implementation.

require and ensure

These are primitives actions of Ivy. Only `require` is really used for the QUIC specification. The actions `ensure` and `require` are similar in the sense that if the condition return false, then the action fails and so the execution. Ivy treats the `require` statement as an assumption and the `ensure` statement as a guarantee [23].

We have already seen example of `require` which is related to **pre-condition**, so here is one for `ensure` which is used to represent **post-condition**, this example has been taken from the Ivy documentation:

```
1  action decr(x:t) returns (y:t) = {
2      require x > 0;
3      y := x - 1;
4      ensure y < x;
5  }
```

_generating and _finalize

Finally, the primitive action `_finalize` and the relation `_generating` are the last Ivy concepts used for the QUIC specification.

The `_generating` keyword is a primitive that return either true or false. Its value indicates if the evaluation of this relation has place in the mirror generation process. In this example, the mirror will apply a defined set of requirements to the generated entity.

generating

```
1  before frame.ping.handle(f:frame.ping,scid:cid,dcid:cid) {
2      if _generating {
3          require client.ep.addr ~= 0;
4          require scid = the_cid;
5      }
6  }
```

We can also verify some requirements that must eventually happen during the execution with the `_finalize` action. For example, the requirement of receiving data at the end of a scenario is placed in this action. Eventually receiving data means that at the end, more than 0 byte has been received.

finalize

```
1  export action _finalize = {
2      require conn_total_data(the_cid) > 0;
3  }
```

Appendix C

Theory complements

C.1 Decidability

Verifying and proving theorems and logical rules is a very complex and time-consuming task. This is why it is imperative to have an efficient algorithm and for that, McMillan and his teams decided to use Microsoft's Z3 theorem prover since they also developed it.[76]

C.1.1 Logic concepts

First of all, we will define some logical concepts that are needed to understand how Ivy works.

Propositional logic

In brief, the syntax of propositional logic defines the allowable sentences. The atomic sentences consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts. Here are the basic operators:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

We can also use different rules (e.g de Morgan) and axioms to solve problems (e.g Modus ponens). For those who are interests in the subject, you can check out this link.[77]

First-order logic

Also called predicate logic, where the main differences with propositional logic are:

1. Propositions are replaced by predicates;
2. Existential and universal quantifiers are introduced
3. It is possible to express relationships between the elements of a set in a concise way, even when the set is of infinite size

With this logic we can have expressions such as "All men are mortal, Socrates is a man so Socrate is mortal".

C.1.2 Weakest preconditions

A solver must be able to transform the predicate from the first-order logic to be able to infer the logical formula. There are two main ways to build valid deduction: **weakest-precondition** and **strongest-postconditions**. Ivy constructs his proofs by using the weakest liberal precondition. The most general (i.e., weakest precondition) P that satisfies[78]

$$\{P\}\ S\ \{Q\} \equiv \text{ Hoare Triple}$$

With S a statement, P the precondition and Q the postcondition of the statement. To check $\{P\}S\{Q\}$, we need to prove that $P \rightarrow wp(S, Q)$.

Here is a small resume of some of the rules:

$$wp(x := E, Q) = Q[E/x]$$
$$wp(\text{assert } P, Q) = P \wedge Q$$
$$wp(\text{assume } P, Q) = P \rightarrow Q$$
$$wp(s1; s2, Q) = wp(s1, wp(s2, Q))$$
$$wp(s1\ s2, Q) = wp(s1, Q) \wedge wp(s2, Q)$$

Weakest liberal preconditions

Weakest liberal precondition is denoted as wlp. If S is a program statement and Q is a some condition on the program state, $wlp(S, Q)$ is the weakest condition P such that, if P holds before the execution of S and if S terminates, then R holds after the execution of S.[79][80]

Again, for each triple $\{P\}S\{Q\}$, we have to prove that $P \rightarrow wps(S, Q)$ with the modified rules.

The difference with the weakest precondition wp is that in the liberal one we don't guarantee that S terminates.

WLP calculus and rules

$$wlp(skip, Q) = Q \ [\mathbf{1}]$$
$$wlp(X := E, Q) = Q[E/X] \ [\mathbf{2}]$$

($Q[E/X]$ is the substitution of every occurrence of X in Q state by the value E)

Ivy example :

- Let's have a Q state containing $x = 4$ and $y = 8$
- $wlp(x := y, Q) = Q\{x = 8, y = 8\}$

$$wlp(s1; s2, Q) = wlp(s1, wlp(s2, Q)) \ [\mathbf{3}]$$

This rule explains why we are going backward when we use the weakest precondition: the first WLP calculus rule that will be effectively applied will start with the last statement.

Ivy example :

- We want to prove that Y = 8.
- $wlp(x := 3; y = 5; y := x + y, Q) = Q = \{y = 8\}$
- $wlp(x := 3; y = 5, wlp(y := x + y, Q) = Q = \{y = 8\}$
- $wlp(x := 3; wlp(y = 5, wlp(y := x + y, Q)) = Q = \{y = 8\}$
- $wlp(x := 3, wlp(y = 5, Q)) = Q = \{x + y = 8\}$
- $wlp(x := 3; Q3) = Q = \{x + 5 = 8\}$
- $Q = \{8 = 8\}$

It is obvious that 8=8 we, therefore, proved that $y = 8$.

$$wlp(\text{if } C \text{ then } s1 \text{ else } s2, Q) = (C \to wlp(s1, Q)) \wedge (\neg C \to wlp(s2, Q)) \ [\mathbf{4}]$$
$$wlp(\text{assume } S, Q) = (S \to Q) \ [\mathbf{5}]$$
$$wlp(\text{assert } S, Q) = (S \wedge Q) \ [\mathbf{6}]$$

Quantifier-free linear integer arithmetic

Algorithms that solve the problem of determining whether a given first-order logic formula is exponential in time for the worst cast since this problem is NP-complete.

These algorithms handle a theory called "Quantifier-Free Linear Integer Arithmetic" (QFLIA). [81] It considers a set of variable X only containing integers which allows us to form combination of constraints define with operators "and" ($\wedge$), "or" ($\vee$) and "not" ($\neg$).

It uses a kind of proof by contradiction where as reminder, to prove proposition $P \to Q$ then:

- We assume P to be false, i.e $\neg P$
- We have to show that $\neg P \to (Q \wedge \neg Q)$ which is a contradiction, so P must be true.

The algorithm proceeds more or less the same way by searching if the negated proposition/condition has a solution or not meaning in the first case that the proposition is not valid.

Decidable fragment and Z3

First, let's explain what is a decidable fragment. A fragment in mathematical logic is a subset of this logic theory which is obtained by adding restrictions on it. We have a reduction from a problem to another one.

Proceeding like that allows taking advantage from the property that the time complexity of saying whether a fragment is satisfiable/decidable or not cannot be higher than for the original problem.

Z3 SMT solver use this and given enough time and memory, it should always be able to determine if formula in the fragment is satisfiable.

In practice, Z3 is very predictable for formula inside fragment than formula outside because it can easily have a reliable counter-models/negated proposition. It is also very efficient for small formulas as we can expect.

We also know that all quantifier-free formulas are in a decidable fragment. This allows us to reason on formula with a quantifier, for that let's take a look at a simple but good example of McMillan:

$$\forall X : f(X) > X$$

To prove this, we can instantiate the quantifier for some constant y like this:

$$f(y) > y$$

We know as a consequence of Herbrand's theorem that the method of instantiating quantifier is complete, so if we show that the verified condition is not satisfiable using instantiation, then this formula is unsatisfiable in general. But the inverse is not always true except for decidable fragment where it can be shown that there is always a finite set of instances such that if they are satisfiable then the formula is satisfiable.

Appendix D

Paper reproduction

In this section, we reproduce the results that we obtained by following the experimental protocol of the paper in the form of a report:

"Formal Specification and Testing of QUIC"
from *KL McMillan, LD Zuck*[1].

We also do a comparison between those results and the results we obtained with our upstarted version of `Ivy`.

D.1 Results

D.1.1 Methodology

We begin this task by creating "installer" files[2] in order to try to have similar setup than McMillan by checkout good commits for each modules/third party.

We don't have any guarantee that the setup is exactly the same than McMillan used. The version of Ivy/implementations that McMillan used are not shown in the article. This could change the final result. Also, the branch he used to update Ivy to the `QUIC` 18 version was used to update to many other versions. Again, the commits were not clear to see which is the final commit of each version. We sectioned the commits that were the most close to the `QUIC` 18 draft.

We first installed `Ivy` at the following branch `quic18_client` which mainly implement the `QUIC` draft 14 with some updated rules. There are also other `QUIC` 18 branches but those looked as experimental branches. We also decide to change a little bit the normal installation setup since it was not adapted for the draft 18 due to version errors. All of that can be checked in the installers.

[1] `http://mcmil.net/pubs/SIGCOMM19.pdf`

[2] `https://github.com/ElNiak/Toward-verification-of-QUIC-extensions/tree/master/installer`

Then we installed different version of QUIC but only tested the 3 first one:

picoquic (C) [51]
95dd82f
picotls [53]
4e6080b6a1ede0d3b23c72a8be73b46ecaf1a084
quant (C) [59]
b55051011a3a040ccc93e83add29dec46eceda54

As McMillan did in the paper, we ran a series of 1000 tests for each test to experiment on the long run the protocol implementation. We also do that on 3 machines:

1. One virtual machine on Ubuntu 18.0 LTS with virtual box
2. One recent Dell with Linux Mint 19.3 Tricia
3. And finally one old Asus with Linux Mint 19 Tara

D.1.2 Our results

Our results are in the following repository:

`https://github.com/ElNiak/Toward-verification-of-QUIC-extensions/tree/master/result`

We did our tests on different setup. Our results are located in the result folder. Inside it, there is one folder containing the results of each 1000 iterations per test for all machine configuration. We first present you the number of test that passed and then the errors we get:[3]

Table D.1: VM Ubuntu - draft 18

	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 D.2: Asus Linux Mint - draft 18

	picoquic	**quant**
quic_server_test_stream	610/1000	386/1000
quic_server_test_max	620/1000	297/1000
quic_server_test_connection_close	101/1000	85/1000
quic_client_test_max	138/1000	0/1000

[3]So this represent the number of test that passed

Table D.3: Dell Linux Mint - draft 18

	picoquic	**quant**
quic_server_test_stream	647/1000	351/1000
quic_server_test_max	584/1000	273/1000
quic_server_test_connection_close	93/1000	86/1000
quic_client_test_max	130/1000	0/1000

We can see that independently of the computer distribution, the tests generally produce the same amount of errors per implementation and per test which comfort us in the fact that our result are trustful.

We can also see that there are a lot of errors in the tested implementations, some of them have more errors than other. Even if PicoQUIC looks more compliant to the draft, many tests for both implementations fail. This observation is similar for other implementations that we tested and that are not shown here.

Summary of errors

Now let's see in more details the errors we obtain for the different tests on the Virtual Machine, we do not show the results for the other one but you can assume that they are quite similar and can be found here.[4] [5]

Table D.4: VM Ubuntu - `PicoQUIC`

test_connection_close	Ran out of values for type cid	412
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	53
	require ~conn_closed(scid); [2]	1
	bind failed: Address already in use	1
	require ~draining_pkt_sent(scid) & queued_close(scid);	249
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	4
	require conn_total_data(the_cid) > 0;	177
	require ~conn_closed(scid); [8]	2
		899/1000
quic_client_test_max	Ran out of values for type cid	862
	require src = server.ep & dst = client.ep;	4
	cipher for level 2 is not set	369
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	21
	require stream_id_allowed(dcid,f.id); [6]	4
	require f.err_code = 0	3
	cipher for level 3 is not set	629
	Segmentation fault	105
	timeout: the monitored command dumped core	105
		862/1000

[4] Google doc

[5] So this represent the number of error per test

Table D.5: VM Ubuntu - `PicoQUIC`

Test	Error code	#
quic_server_test_stream	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	242
	Ran out of values for type cid	121
	require ~path_challenge_pending(dcid,f.data);	24
	bind failed: Address already in use	2
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	1
		390/1000
quic_server_test_max	require stream_id_allowed(dcid,f.id); [4]	219
	Ran out of values for type cid	102
	require ~path_challenge_pending(dcid,f.data);	8
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	12
	require conn_total_data(the_cid) > 0;	31
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	7
	bind failed: Address already in use	1
		380/1000

Table D.6: VM Ubuntu - `Quant`

Test	Error code	#
quic_server_test_stream	require ~path_challenge_pending(dcid,f.data);	612
	require e = encryption_level.other -> ~pkt.hdr_long;	1
	bind failed: Address already in use	1
		614/1000
quic_server_test_max	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	6
	bind failed: Address already in use	1
	require f.err_code = 0	128
	require conn_total_data(the_cid) > 0;	45
	require ~path_challenge_pending(dcid,f.data);	523
		703/1000
test_connection_close	require ~conn_closed(scid); [8]	1
	require ~conn_closed(scid); [2]	3
	require ~path_challenge_pending(dcid,f.data);	106
	bind failed: Address already in use	1
	require ~draining_pkt_sent(scid) & queued_close(scid);	342
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	3
	require conn_total_data(the_cid) > 0;	310
	require e ~= encryption_level.initial;	149
		915/1000
quic_client_test_max	timeout	**1000/1000**

Explanation of errors

Finally, here are some explanation about the error codes that we got when we launch the test for draft 18:

Path challenge frames (QUIC18-19.17)

- `~path_challenge_pending(dcid,f.data);`

  It seems that this error happened when "Path Challenge" Frames which are used to request verification of ownership of an endpoint by a peer is required but failed and is never received in reality. This frame is used to check

reachability to the peer and for path validation during connection migration but also in order to minimize the risk of amplification attacks.

Application Close frames (QUIC16-19.4)

- `require e ~= encryption_level.initial;`

  This error appears during handling of an "Application close" Frames, it normally requires no (~) initial encryption level but it seems to not be the case here and there is some encryption.

  Not strange since not in the current QUIC 18 version.

Max Stream ID frames (QUIC16-19.7)

- `require ~conn_closed(scid); # [2]`

  This happen during the handling of "Max Stream ID" Frames, when the requirements that these frames may not occur in initial or handshake packets is not respected.

  Not strange since not in the current QUIC 18 version.

RST Stream frames (QUIC18-19.4)

- `require stream_id_allowed(dcid,f.id); # [4]`

  This happen during the handling of "RST Stream" Frames, when the requirements that the stream id must not exceed the maximum stream id for stream kind is not satisfied.

Stream frames (QUIC18-19.8)

- ```
  conn_total_data(dcid):= conn_total_data(dcid)
   + (length - stream_length(dcid,f.id)); # [12]
  ```

  This happen during the handling of "Stream" Frames that carry stream data, when the requirements that indicate that if the stream length (a) for a destination's frame is less than the total frame length (b) (offset included), then we should update the connection total data with the difference between (a) and (b) added to that but this is not satisfied.

- `require stream_id_allowed(dcid,f.id); # [6]`

  In theory, the stream ID must be less than or equal to the max stream ID for the kind of the stream. (QUIC16-19.7 not present anymore in the QUIC 18)
  ```

- `require ~conn_closed(scid); # [8]`

  This happen during the handling of "Stream" Frames that carry stream data, when the requirements that indicate that the connection must not have been closed the the source endpoint during the treatment of stream frame but this is violated here.

Packet Protocols

- `require ~draining_pkt_sent(scid)& queued_close(scid);`

  This may occur during the handling of packet events, we expect that during the draining state, we should make sure that a draining packet has not been previously sent and that the packet contains a "Connection Close" frame but this is not the case here. (QUIC18-10.1)

- `require ~_generating & ~queued_non_ack(scid)-> ack_credit(scid)> 0; # [5]`

  Also occurs during the handling of packet events, we have the requirement that a packet containing only "ACK" frames and padding is "ACK-only" (QUIC18-13.1.1.). For a given CID, the number of ACK-only packets sent from the source to destination must not be greater than the number of non-ACK-only packets sent from destination to the source which is not respected here.

- `require conn_seen(dcid)-> hi_non_probing_endpoint(dcid,dst); # [10]`

  Here again, we have the requirement during the handling of packet event that for a given connection, a server must only send packet to an address that at one time in the past sent the AS packet with the highest numbered packet thus far received. This concern the connection migration address (QUIC18-9) and path challenge frame (QUIC18-19.17).

  Note of McMillan *On see a packet form a new address with the highest packet number see thus far, the server detects migration of the client. It begins sending packets to this address and initiates path validation for this address. Until path validation succeeds, the server limits data sent to the new address. Currently we cannot specify this limit because we don't know the byte size of packets or the timings of packets. On detecting migration, the server abandons any pending path validation of the old address. We don't specify this because the definition of abandonment is not clear. In practice, we do observe path challenge frames sent to old addresses, perhaps because these were previously queued. QUESTION:* ***abandoning an old path challenge could result in an attacker denying the ability of the client to migrate by replaying packets, spoofing the old address****. The server would alternate between the old (bogus) and new address, and thus never complete the path challenge of the new address.*

- `quic_connection.ivy: line 698`

  We also have the requirement that no payload can be empty during connection, we should have at least one frame inside (QUIC18-12.4) which is not respected all the times.

- `quic_connection.ivy: line 687`

  Also during the handling of packet event but here it concern the encryption level during the handshake. It requires the long header packet type and "`0x20`" type bits which is used as Latency Spin Bit (QUIC18-17.3) in header but this is not respected here. This bit enables latency monitoring from observation points on the network path throughout the duration of a connection.

Server

- `require conn_total_data(the_cid)> 0;`

  At the end of the test, we check data some date was actually received from the server. This error indicate that no data were produce in total for the connection and a corresponding client CID.

- `require f.err_code = 0`

  During the generation of "Connection Close" frames by the server for the client, we are here in the case when the server is not in the "generating" frame state and so should return an error code "`NO_ERROR`", but this is not the case. (QUIC18-19.19)

Client

- `require f.err_code = 0`

  Same than `quic_server_test.ivy: line 518`

- `require src = server.ep & dst = client.ep;`

  Before handling stream frame, we need to ensure that the source IP address is equal to the server address and than the destination IP address is the client address which is not the case here.

Errors

- `Segmentation fault`

  There is some implementation that still have memory errors in their codes.

- `Ran out of values for type cid`

  Indicate that all the possible value for the CID has been used ??

- `bind failed: Address already in use`

  This happened when the server suddenly restart or something like that and that the IP address is already used by the last time. This should not happened.

- `timeout`

  Timeout launch by Ivy itself, it mean that 100 seconds has been reach and that nothing happened for the QUIC implementation, the tested implementation is either in wrong version, or the tested features is not implemented. Concerning `MinQUIC`, we think that the timeout is caused by wrong `TLS` implementation since it blocks at the cypher event.

- `cipher for level X is not set`

  Error with the `TLS` encryption, only occurs with `Quant`.

D.1.3 Comparison between our results and McMillan results

As said before, we perform our test in a similar fashion than McMillan, i.e that we run these test in batches of 1000 on the same server instance to test their long-run behavior too but on 3 machines to see any potential difference.

In a first overlook, it seems that we have more or less the same result than McMillan. We got in total **21** errors, all types included where McMillan revealed 27 errors so either we do not find all of them, either there are some minor difference between our version that correct some of them and also due to the fact that the `MinQUIC` implementation doesn't work for us, and finally since we only test draft 18.

We can find the complete details of McMillan results here.[6] Note that most of the tested version were anterior to the draft 18.

McMillan found that many protocols errors were due to unimplemented requirement. We got more or less the same result, we see that:

- One implementation, `PicoQUIC`, failed to respect a prohibition on sending a packet solely to acknowledges an ACK-only packet.
- Also two of them (`PicoQUIC` and `Quant`) failed to obey a rule that allows client migration to be detected only if a packet with the highest-observed sequence number arrives from a new address.
- We also have one implementation, `Quant`, that did not respect the error code in `CONNECTION_CLOSE` frame.

[6] `https://github.com/microsoft/ivy/blob/quic18_client/doc/examples/quic/anomalies`

D.2 Conclusion

We note that the analysis of the results is a tedious process and that we have to check manually all the output files to see which tests were failed. It is also to be noted that Ivy work as expected and is operational for the testing of QUIC implementation We improve that in our updated work with the automatic generation of results with graphs and clear output. We also see that we have trouble finding the good commits and that the deployment is not optimal and machine dependant. We fixed this problem using Docker. We improve the number of implementations with whom we operate Ivy. We also do the core of our work: updating to the QUIC 29 draft and add verification of extensions.

Appendix E

Evolution of the results

For this part we only present the evolution for `picoquic`.

E.1 Server evolution

We compare the evolution between draft 18 of the McMillan paper and draft 29. This only concern the tests that McMillan already defined. For the draft 18, we consider the reproduction of the results we have done since the results that McMillan used for its paper were not available. A complete version of these results is present in the appendix D.

Test	Error code	#
stream	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	242
	Ran out of values for type cid	121
	require ~path_challenge_pending(dcid,f.data);	24
	bind failed: Address already in use	2
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	1
		390/1000
max	require stream_id_allowed(dcid,f.id); [4]	219
	Ran out of values for type cid	102
	require ~path_challenge_pending(dcid,f.data);	8
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	12
	require conn_total_data(the_cid) > 0;	31
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	7
	bind failed: Address already in use	1
		380/1000
connection_close	Ran out of values for type cid	412
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	53
	require ~conn_closed(scid); [2]	1
	bind failed: Address already in use	1
	require ~draining_pkt_sent(scid) & queued_close(scid);	249
	require conn_seen(dcid) -> hi_non_probing_endpoint(dcid,dst); [10]	4
	require conn_total_data(the_cid) > 0;	177
	require ~conn_closed(scid); [8]	2
		899/1000

Table E.1: `picoquic` draft 18 - Number of errors

Test	Error code		#
stream	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	44
			44.0/100
max	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	52
	require conn_total_data(the_cid) >0;		1
			53.0/100
connection_close	require conn_seen(dcid) ->hi_non_probing_endpoint(dcid,dst);	[10]	19
			19.0/100

Table E.2: `picoquic` draft 29 - Number of errors

Only by looking at the number of the test passed and failed we cannot conclude that `picoquic` evolves well. However, when we look at the number of different errors, we can see fewer errors. We can see all the corrections made, for example with the migration and the `PATH_CHALLENGE` management. We conclude that we should not consider the ratio of tests that passed with those that failed. This is not very appropriate in this situation since some features are not always well implemented and occur in every test such as the migration. It can cause high differences in terms of results. For example, when we disable the migration, `aioquic` has a much better score if we only consider the successful tests. We can see that the distribution of errors changed from draft 18 to draft 29. For example, the ratio `conn_seen(dcid)->hi_non_probing_endpoint(dcid,dst)` errors increased. An explanation is given in the `picoquic` section.

These global results can induce false conclusions because all the errors do not have the same weight. Also, these results should be analysed while reading the analysis of traces that are put for each implementation. Some errors are not critical but strongly reduce the ratio of succeeded test for one implementation. On the other side, one implementation with critical but not frequent errors have a better ratio.

What we should consider is more the error itself, the number of different errors and how they are linked to each other. For example, we know that some errors are only related to the migration. Correcting this part will remove quite a large part of errors from some implementations.

E.2 Client evolution

As for the server, here is a more detailed version containing the number of errors per test and compare the evolution with draft 18 of the McMillan paper with draft 29.

Table E.3: `picoquic` draft 18 - Number of errors

Test	**Error code**	**#**
max	Ran out of values for type cid	862
	require src = server.ep & dst = client.ep;	4
	cipher for level 2 is not set	369
	require ~_generating & ~queued_non_ack(scid) -> ack_credit(scid) > 0; [5]	21
	require stream_id_allowed(dcid,f.id); [6]	4
	require f.err_code = 0	3
	cipher for level 3 is not set	629
	Segmentation fault	105
	timeout: the monitored command dumped core	105
		862/1000

Table E.4: `picoquic` draft 29 - Number of errors

Test	**Error code**	**#**
max	require ~_generating & ~queued_non_ack(scid) ->ack_credit(scid) >0; [5]	20
	Connection end with local error 0x434	65
		85.0/100

Most of the error found disappear but a new problem with retransmission appear probably linked to some performance problem of Ivy. It may hide errors that are still present but that we do not discover.

Appendix F

Google Docs

You can find the google doc that we used to list the requirement of the QUIC RFC and the extension in this link:

https://docs.google.com/spreadsheets/d/1O1TvLbuosTN5R6xtCBo8XaueozBih7BU0DtzR8OE0wk/edit?usp=sharing

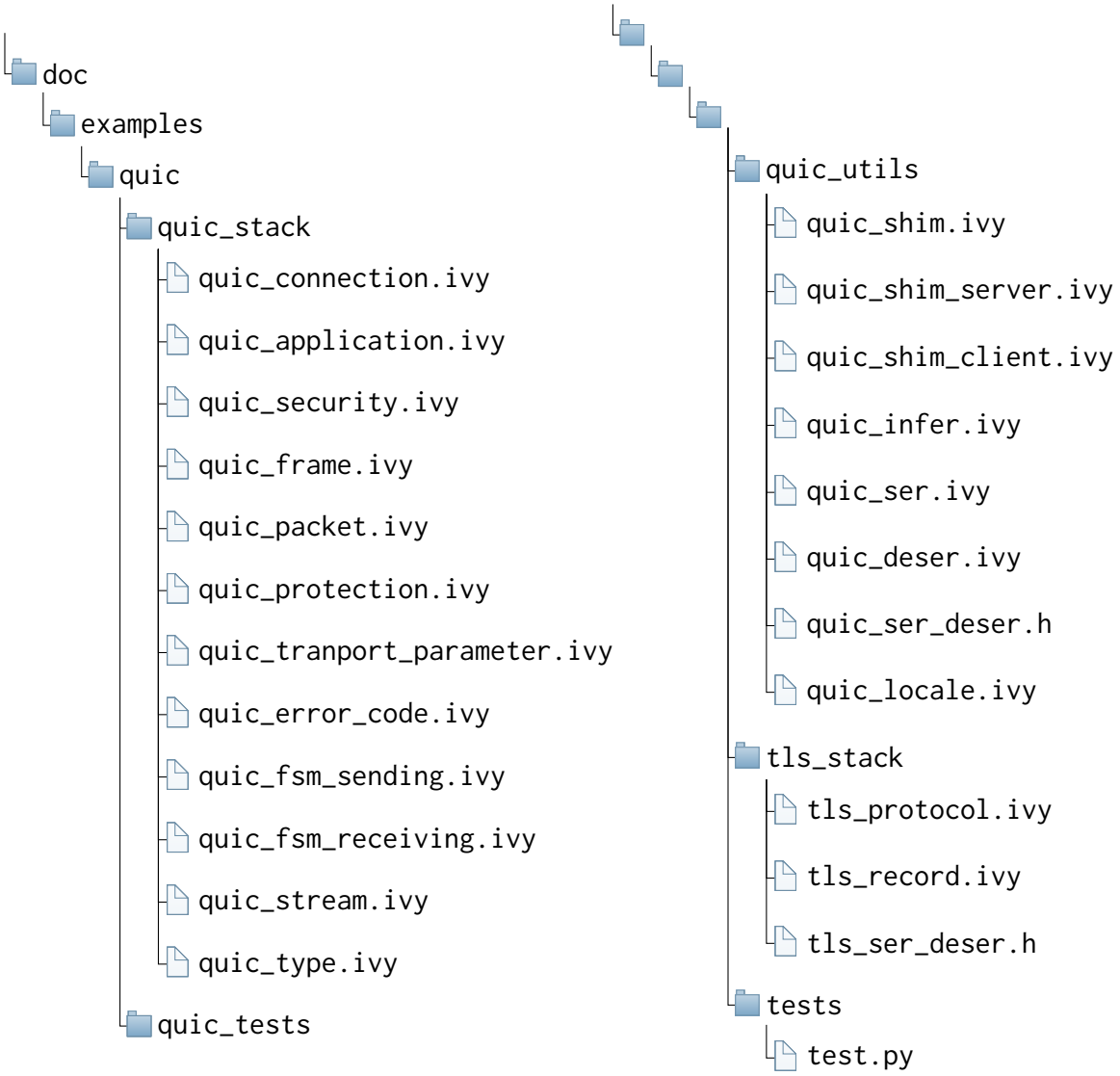
https://docs.google.com/spreadsheets/d/1Bsz6K5CkcfqIYaWWiTS1oba2iZ82upTZxTnRxKYvnS0/edit?usp=sharing

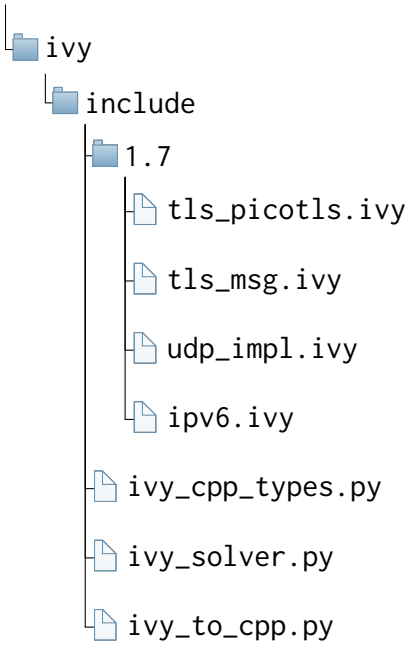
And you can find the google doc that we used to list for the reproduction of the result of McMillan here:

https://docs.google.com/spreadsheets/d/1WkqKCKSSSM3QD5_SshoD6J30v8t9DslmmHi1xmDJcsU/edit?usp=sharing

Appendix G

Ivy project structure




ivy
include
1.7
tls_picotls.ivy
tls_msg.ivy
udp_impl.ivy
ipv6.ivy
ivy_cpp_types.py
ivy_solver.py
ivy_to_cpp.py

Appendix H

Project installation guidelines

H.1 TLS Keys for Wireshark

Based on what has been done in `picoquic` implementation and adapted to the current API, we add the feature to Ivy to directly log the TLS keys in the `SSLKEYLOGFILE` environment variable if present. This code can be found in `tls_picotls.ivy`. This allows the decryption of packets on the fly with Wireshark.

H.2 Docker

Since the installation of Ivy is not trivial for many reasons, we decide to package our modified version of Ivy in a Docker container [82] which will help future development. It is important to keep the project maintainable as we can observe that not many open-source projects forked Ivy and that it is probably due to the difficulty to install the project. To illustrate this, we contacted some developers that wanted to contribute to the project. They told us that they didn't manage to make it work. Then, they moved on to another tool. [83] All we need to do now is resumed by one command:

```
python run-project.py -b true -m <client/server/all>
        -p <path_result> -i <nb_iteration>
```

With `-b` the parameter saying if we build or not the docker, `-p` specifying the path to save the results files and `-i` the number of iterations for each test. For each test, we record the `stdout/stderr` of Ivy and the tested implementation. We also recorded a network trace with the potential anomaly.

Appendix I

Successful tests examples

We present you typical successful scenario for each test. Note that the shape of trace, for a given test, are different between implementations which is expected. But also between instance of the same test with the same implementation since randomness is involved.

I.1 Server

`quic_server_test_stream`

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000001, SCID=2f07fe77c304d47a, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=2f07fe77c304d47a, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, PING, PADDING
4987	4443	107	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 2, STREAM(8), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, ACK, STREAM(8), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, NT, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 7, DONE, ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 8, STREAM(8), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 9, NT, ACK, PADDING
4987	4443	157	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 4, STREAM(20), STREAM(12), STREAM(4), ACK, ACK, PADDING
4443	4987	1249	Protected Payload (KP0), DCID=0000000000000001, PKN: 10, ACK, STREAM(4), STREAM(12), STREAM(20), PADDING
4987	4443	102	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 6, ACK, ACK, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 11, DONE, ACK, PADDING
4987	4443	92	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 8, ACK, PADDING
4987	4443	92	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 12, ACK, PADDING
4987	4443	107	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 32, STREAM(16), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 12, ACK, STREAM(16), PADDING
4443	4987	1249	Protected Payload (KP0), DCID=0000000000000001, PKN: 13, STREAM(4), STREAM(12), STREAM(20), PADDING
4987	4443	92	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 36, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 14, DONE, ACK, PADDING
4987	4443	92	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 38, ACK, PADDING
4987	4443	112	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 48, STREAM(24), ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 15, ACK, STREAM(24), PADDING
4987	4443	107	Protected Payload (KP0), DCID=2f07fe77c304d47a, PKN: 64, STREAM(28), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 16, ACK, STREAM(28), PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 17, DONE, ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 18, STREAM(24), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 19, STREAM(28), PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 20, DONE, ACK, PADDING

Table I.1: `picoquic` - quic_server_test_stream - no migration

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1294	Handshake, DCID=0000000000000000, SCID=7673707d27335969, PKN: 0, CRYPTO
4443	4988	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, NCI, PADDING
4443	4988	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4988	4443	158	Handshake, DCID=dcdae66f7c532908, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, NCI, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4988	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, NT, NCI, PADDING
4987	4443	92	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 14, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, DONE, ACK, PADDING
4443	4987	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, PC, PADDING
4443	4987	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, PC, PADDING
4987	4443	112	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 16, ACK, STREAM(8), PADDING
4443	4988	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, ACK, STREAM(8), PADDING
4988	4443	105	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 20, PR, PR, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, NT, NCI, ACK, PADDING
4987	4443	107	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 32, STREAM(24), PADDING
4443	4988	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, ACK, STREAM(24), PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, DONE, ACK, PADDING
4443	4988	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, STREAM(8), PADDING
4988	4443	112	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 36, STREAM(12), ACK, PADDING
4443	4988	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, ACK, STREAM(12), PADDING
4988	4443	127	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 40, STREAM(4), STREAM(32), PADDING
4443	4988	865	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, ACK, STREAM(4), STREAM(32), PADDING
4988	4443	107	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 48, STREAM(20), PADDING
4443	4988	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, ACK, STREAM(20), PADDING
4988	4443	132	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 64, STREAM(36), ACK, STREAM(40), PADDING
4443	4988	993	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, STREAM(36), STREAM(40), PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, NT, NCI, ACK, PADDING
4987	4443	107	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 65, STREAM(16), PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, ACK, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 20, DONE, ACK, PADDING
4443	4988	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 21, STREAM(24), STREAM(8), STREAM(12), STREAM(16)
4988	4443	137	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 66, STREAM(28), ACK, ACK, STREAM(44), PADDING
4443	4988	1121	Protected Payload (KP0), DCID=0000000000000000, PKN: 22, ACK, STREAM(16), STREAM(28), STREAM(44), PADDING
4987	4443	112	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 80, STREAM(48), ACK, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 23, NT, NCI, ACK, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 24, DONE, ACK, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 25, NT, NCI, ACK, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 26, DONE, ACK, PADDING
4988	4443	167	Protected Payload (KP0), DCID=dcdae66f7c532908, PKN: 96, ACK, ACK, ACK, STREAM(56), STREAM(52), ACK, ACK, ACK, ACK, ACK, PADDING

Table I.2: `picoquic` - quic_server_test_stream - migration

`quic_server_test_max`

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1242	Handshake, DCID=0000000000000000, SCID=b30979afc3801aea, PKN: 0, CRYPTO, PADDING
4443	4988	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4988	4443	158	Handshake, DCID=8b0c887cf0f63d46, SCID=0000000000000000, PKN: 17, CRYPTO, PADDING
4443	4988	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4988	4443	130	Protected Payload (KP0), DCID=b30979afc3801aea, PKN: 6, MS,, ACK, MS, ACK, MS, MS, PADDING
4443	4988	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, ACK
4988	4443	121	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 16, MS, ACK, MS, MS, PADDING
4443	4988	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK
4987	4443	158	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 17, ACK, STREAM(8), MSD(8), MSD(8), MSD(8), STREAM(12), PADDING
4443	4988	1242	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PC, PADDING
4988	4443	122	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 33, MSD(8), PR, STREAM(16), MSD(8), PADDING
4443	4988	164	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, MS, STREAM(12), STREAM(8)
4443	4988	78	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, MS
4443	4988	118	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, STREAM(16)
4987	4443	125	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 49, MS, MSD(8), MSD(8), MS, ACK, MSD(8), MS, MS, PADDING
4443	4988	1242	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, PC, PADDING
4988	4443	106	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 65, MSD(8), MS, MSD(8), ACK, MSD(8), PADDING
4987	4443	152	Protected Payload (KP0), DCID=8b0c887cf0f63d46, PKN: 73, PR, MS, ACK, MS, MS, MSD(8), STREAM(20), MSD(8), MS, MS, PADDING

Table I.3: `quinn` - quic_server_test_max

quic_server_test_reset_stream

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1242	Handshake, DCID=0000000000000000, SCID=12e0704c47dd57d9, PKN: 0, CRYPTO, PADDING
4443	4987	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4987	4443	158	Handshake, DCID=0134f7764b0d62c7, SCID=0000000000000000, PKN: 13, CRYPTO, PADDING
4443	4987	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4987	4443	119	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 12, ACK, RS(4), STREAM(16), PADDING
4443	4987	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, ACK
4443	4987	76	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, MS
4443	4987	122	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, MS, STREAM(4), STREAM(16)
4988	4443	232	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 16, STREAM(24), STREAM(12), RS(4), ACK, ACK, ACK, STREAM(56), STREAM(8), STREAM(48), STREAM(20), ACK, PADDING
4443	4987	1242	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, PC, PADDING
4988	4443	161	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 32, STREAM(52), RS(4), STREAM(44), RS(4), STREAM(36), PADDING
4987	4443	92	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 40, ACK, PADDING
4988	4443	123	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 48, STREAM(40), PR, RS(4), PADDING
4987	4443	94	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 64, RS(4), PADDING
4987	4443	139	Protected Payload (KP0), DCID=0134f7764b0d62c7, PKN: 72, STREAM(28), STREAM(32), RS(4), RS(4), PADDING

Table I.4: quinn - quic_server_test_reset_stream

quic_server_test_connection_close

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1242	Handshake, DCID=0000000000000000, SCID=8942c8d5e3f9c46f, PKN: 0, CRYPTO, PADDING
4443	4987	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4987	4443	158	Handshake, DCID=8942c8d5e3f9c46f, SCID=0000000000000000, PKN: 11, CRYPTO, PADDING
4443	4987	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4987	4443	103	Protected Payload (KP0), DCID=e1c36cf98995a12e, PKN: 8, CC(0x494a), ACK, PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, CC(NO_ERROR)

Table I.5: quinn - quic_server_test_connection_close

quic_server_test_stop_sending

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 19, CRYPTO, PADDING
4443	4988	1242	Handshake, DCID=0000000000000000, SCID=7ff6b19ea36e9fcc, PKN: 0, CRYPTO, PADDING
4443	4988	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4988	4443	158	Handshake, DCID=7ff6b19ea36e9fcc, SCID=0000000000000000, PKN: 18, CRYPTO, PADDING
4443	4988	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4987	4443	93	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 16, SS(4), PADDING
4443	4988	1242	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, PC, PADDING
4988	4443	103	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 32, SS(4), SS(4), SS(4), PADDING
4988	4443	103	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 48, SS(4), ACK, ACK, PADDING
4988	4443	193	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 64, STREAM(28), SS(4), PR, SS(4), STREAM(16), SS(4), ACK, SS(4), SS(4), STREAM(8), SS(4), PADDING
4443	4988	80	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK
4443	4988	126	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, MS, STREAM(28)
4443	4988	123	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, STREAM(16)
4443	4988	123	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, STREAM(8)
4443	4988	83	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, MS
4443	4988	83	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, ACK, MS
4988	4443	172	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 66, SS(4), SS(4), SS(4), STREAM(24), SS(4), ACK, SS(4), SS(4), PADDING
4443	4988	74	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, ACK
4443	4988	125	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, ACK, RS(4), MS, STREAM(24)
4443	4988	77	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, ACK, MS
4443	4988	82	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, ACK, RS(4), MS
4987	4443	110	Protected Payload (KP0), DCID=341bbf80c78e48cc, PKN: 72, ACK, SS(4), SS(4), SS(4), PADDING

Table I.6: quinn - quic_server_test_stop_sending

quic_server_test_accept_maxdata

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=c4a9e2f7fc28690f, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, NCI, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=c4a9e2f7fc28690f, SCID=0000000000000000, PKN: 15, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, NCI, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4988	4443	97	Protected Payload (KP0), DCID=d0916430ca6b3bc1, PKN: 16, SS(4), MD, MD, PADDING
4443	4988	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, PC, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, RS(4), PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, NT, NCI, ACK, PADDING
4443	4988	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, PC, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, DONE, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, RS(4), ACK, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, NT, NCI, ACK, PADDING
4443	4988	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, PC, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, DONE, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, RS(4), ACK, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, NT, NCI, ACK, PADDING
4988	4443	305	Protected Payload (KP0), DCID=d0916430ca6b3bc1, PKN: 32, MD, ACK, MS, SS(4), STREAM(20), MD, MD, MS, MD, MD, MD, MD, MD, SS(4), MS, STREAM(4), MS, MSD(4), MD, MD, PR, MD, STREAM(24), MD, PR, MSD(4), MD, MD, MD, STREAM(32), MS, STREAM(12), MSD(4), ACK, ACK, MD, PR, PADDING
4443	4987	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, STREAM(12), STREAM(20), STREAM(24), STREAM(32)
4443	4987	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(32), PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, DONE, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 20, RS(4), ACK, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 21, NT, NCI, ACK, PADDING
4987	4443	114	Protected Payload (KP0), DCID=d0916430ca6b3bc1, PKN: 40, ACK, SS(4), ACK, MD, MD, MD, MSD(4), MD, PADDING
4443	4987	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 22, STREAM(12), STREAM(20), STREAM(24), STREAM(32), STREAM(32)
4443	4987	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 23, STREAM(32), PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 24, DONE, ACK, PADDING

Table I.7: picoquic - quic_server_test_accept_maxdata

quic_server_test_ext_min_ack_delay

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=791ee9cbee0d818d, PKN: 0, CRYPTO
4443	4987	517	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, NCI, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=791ee9cbee0d818d, SCID=0000000000000000, PKN: 11, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, NCI, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, ACK_FREQ, PING, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, NT, NCI, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, DONE, PADDING
4987	4443	117	Protected Payload (KP0), DCID=791ee9cbee0d818d, PKN: 9, ACK, STREAM(12), ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, ACK_FREQ, PING, STREAM(12), PADDING
4988	4443	92	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 15, ACK, PADDING
4443	4988	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, PC, PADDING
4443	4988	1294	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, PC, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, NT, NCI, ACK, PADDING
4987	4443	127	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 26, ACK, ACK, STREAM(24), ACK, ACK, PADDING
4988	4443	146	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 27, STREAM(40), ACK, PR, STREAM(28), ACK, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, ACK, STREAM(12), STREAM(24), STREAM(28), STREAM(40)
4988	4443	92	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 31, ACK, PADDING
4443	4987	353	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, STREAM(40), PADDING
4987	4443	107	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 46, STREAM(16), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, ACK, STREAM(16), PADDING
4988	4443	121	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 67, ACK, PR, STREAM(8), PADDING
4987	4443	152	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 76, STREAM(56), STREAM(44), ACK, STREAM(48), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, ACK, STREAM(12), STREAM(24), STREAM(28), STREAM(40)
4987	4443	92	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 77, ACK, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, STREAM(40), STREAM(40), STREAM(16), STREAM(8), STREAM(44), STREAM(48)
4988	4443	112	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 91, STREAM(36), ACK, PADDING
4443	4987	1249	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, ACK, STREAM(36), STREAM(48), STREAM(56), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, STREAM(40), STREAM(40), STREAM(16), STREAM(8), STREAM(44), STREAM(48)
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(48), PADDING
4988	4443	102	Protected Payload (KP0), DCID=2f798ab6d368552c, PKN: 94, ACK, ACK, ACK, PADDING
4443	4987	1249	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, STREAM(36), STREAM(48), STREAM(56), PADDING

Table I.8: picoquic - quic_server_test_ext_min_ack_delay

quic_server_test_unknown

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4988	1294	Handshake, DCID=0000000000000001, SCID=071a8359e899cd54, PKN: 0, CRYPTO
4443	4988	506	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NT, NCI, PADDING
4443	4988	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
4988	4443	158	Handshake, DCID=eec31f59f191e729, SCID=0000000000000001, PKN: 11, CRYPTO, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, NT, NCI, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, DONE, PADDING
4443	4988	1442	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, PING, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, NT, NCI, PADDING
4987	4443	152	Protected Payload (KP0), DCID=071a8359e899cd54, PKN: 10, STREAM(4), ACK, UNKNOWN, STREAM(8), PADDING
4443	4987	73	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, CC(FRAME_ENCODING_ERROR)

Table I.9: picoquic - quic_server_test_unknown

quic_server_test_unknown_tp

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
Handshake Protocol: Client Hello Type: quic_transport_parameters (drafts version) (65445) Parameter: Unknown 0x0042 (len=0)			
4443	4987	1294	Handshake, DCID=0000000000000001, SCID=b0b71e6e598aadb9, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=b0b71e6e598aadb9, SCID=0000000000000001, PKN: 11, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, PING, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, NT, PADDING
4987	4443	102	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 12, ACK, ACK, ACK, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, ACK, PADDING
4987	4443	132	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 15, STREAM(4), STREAM(8), ACK, PADDING
4443	4987	865	Protected Payload (KP0), DCID=0000000000000001, PKN: 7, ACK, STREAM(4), STREAM(8), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 8, NT, ACK, PADDING
4987	4443	137	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 22, ACK, ACK, STREAM(12), STREAM(24), PADDING
4443	4987	865	Protected Payload (KP0), DCID=0000000000000001, PKN: 9, ACK, STREAM(12), STREAM(24), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 10, NT, ACK, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 11, STREAM(4), STREAM(8), STREAM(12), STREAM(24)
4987	4443	142	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 35, STREAM(16), STREAM(28), ACK, ACK, ACK, PADDING
4443	4987	1121	Protected Payload (KP0), DCID=0000000000000001, PKN: 12, ACK, STREAM(24), STREAM(16), STREAM(28), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 13, NT, ACK, PADDING
4987	4443	112	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 44, STREAM(20), ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 14, ACK, STREAM(20), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 15, STREAM(4), STREAM(8), STREAM(12), STREAM(24)
4443	4987	1121	Protected Payload (KP0), DCID=0000000000000001, PKN: 16, STREAM(24), STREAM(16), STREAM(28), PADDING
4987	4443	112	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 47, STREAM(52), ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 17, ACK, STREAM(52), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 18, NT, ACK, PADDING
4987	4443	127	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 63, STREAM(40), STREAM(36), PADDING
4443	4987	865	Protected Payload (KP0), DCID=0000000000000001, PKN: 19, ACK, STREAM(36), STREAM(40), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 20, STREAM(4), STREAM(8), STREAM(12), STREAM(24)
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 21, STREAM(24), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 22, STREAM(24), STREAM(16), STREAM(28), STREAM(52), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 23, NT, ACK, PADDING
4443	4987	865	Protected Payload (KP0), DCID=0000000000000001, PKN: 24, STREAM(36), STREAM(40), PADDING
4987	4443	122	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 71, STREAM(56), ACK, ACK, ACK, PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 25, ACK, STREAM(56), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 26, STREAM(4), STREAM(8), STREAM(12), STREAM(24)
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 27, STREAM(24), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 28, STREAM(24), STREAM(16), STREAM(28), STREAM(52), PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 29, NT, ACK, PADDING
4443	4987	865	Protected Payload (KP0), DCID=0000000000000001, PKN: 30, STREAM(36), STREAM(40), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000001, PKN: 31, STREAM(56), PADDING
4987	4443	137	Protected Payload (KP0), DCID=b0b71e6e598aadb9, PKN: 87, ACK, ACK, STREAM(32), STREAM(44), PADDING
4443	4987	993	Protected Payload (KP0), DCID=0000000000000001, PKN: 32, ACK, STREAM(32), STREAM(44), PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 33, STREAM(4), STREAM(8), STREAM(12), STREAM(24)
4443	4987	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 34, STREAM(24), PADDING

Table I.10: picoquic - quic_server_test_unknown_tp

quic_server_test_tp_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
TLSv1.3 Record Layer: Handshake Protocol: Client Hello Extension: quic_transport_parameters (drafts version) (len=50) Parameter: ack_delay_exponent (21276)			
4443	4988	102	Initial, DCID=0000000000000000, SCID=65018529e6f781f0, PKN: 0, CC(TRANSPORT_PARAMETER_ERROR)

Table I.11: quinn - quic_server_test_tp_error

quic_server_test_double_tp_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
TLSv1.3 Record Layer: Handshake Protocol: Client Hello Extension: quic_transport_parameters (drafts version) (len=50) Parameter: ack_delay_exponant Parameter: ack_delay_exponant			
4443	4988	102	Initial, DCID=0000000000000000, SCID=dfa7c6ad4b79c9f3, PKN: 0, CC(TRANSPORT_PARAMETER_ERROR)

Table I.12: quinn - quic_server_test_double_tp_error

quic_server_test_tp_acticoid_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
TLSv1.3 Record Layer: Handshake Protocol: Client Hello Extension: quic_transport_parameters (drafts version) (len=50) Parameter: active_connection_id_limit (len=1) 0			
4443	4988	102	Initial, DCID=0000000000000000, SCID=bd2312dd9b5a06aa, PKN: 0, CC(TRANSPORT_PARAMETER_ERROR)

Table I.13: quinn - quic_server_test_tp_acticoid_error

quic_server_test_no_icid

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
TLSv1.3 Record Layer: Handshake Protocol: Client Hello Extension: quic_transport_parameters (drafts version) (len=50) Parameter: initial_max_stream_data_bidi_local (len=4) 162506515 Parameter: initial_max_data (len=4) 162506515 Parameter: max_idle_timeout (len=4) 162506513 ms Parameter: initial_max_stream_data_bidi_remote (len=4) 162506515 Parameter: initial_max_stream_data_uni (len=4) 162506515			
4443	4988	102	Initial, DCID=0000000000000000, SCID=7a391e1a9579ae46, PKN: 0, CC(TRANSPORT_PARAMETER_ERROR)

Table I.14: quinn - quic_server_test_tp_no_icid

quic_server_test_blocked_streams_maxstream_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1242	Handshake, DCID=0000000000000000, SCID=2f203919ee32ddfb, PKN: 0, CRYPTO, PADDING
4443	4988	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4988	4443	158	Handshake, DCID=2f203919ee32ddfb, SCID=0000000000000000, PKN: 9, CRYPTO, PADDING
4443	4988	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4988	4443	112	Protected Payload (KP0), DCID=6fa6e5e197476bb1, PKN: 11, ACK, STREAM(4), PADDING
4443	4988	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, ACK
4443	4988	76	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, MS
4443	4988	116	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, STREAM(4)
4988	4443	101	Protected Payload (KP0), DCID=6fa6e5e197476bb1, PKN: 15, ACK, SB(Stream Limit: 4611686018427387903), PADDING
4443	4988	100	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, CC(FRAME_ENCODING_ERROR)

Table I.15: quinn - quic_server_test_blocked_streams_maxstream_error

quic_server_test_retirecoid_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=dd8fadf21fea198a, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=dd8fadf21fea198a, SCID=0000000000000000, PKN: 20, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, DONE, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, NT, PADDING
4987	4443	207	Protected Payload (KP0), DCID=dd8fadf21fea198a, PKN: 16, RCI(seqnum:1000), ACK, ACK, STREAM(12), STREAM(8), ACK, ACK, STREAM(4), ACK, ACK, PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, CC(PROTOCOL_VIOLATION)

Table I.16: picoquic - quic_server_test_retirecoid_error

quic_server_test_stream_limit

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000002, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
4443	4987	1242	Handshake, DCID=0000000000000001, SCID=3bf51258275b8c51, PKN: 0, CRYPTO, PADDING
4443	4987	96	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NCI
4987	4443	158	Handshake, DCID=3bf51258275b8c51, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
4443	4987	71	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, DONE, PADDING
4987	4443	98	Protected Payload (KP0), DCID=3bf51258275b8c51, PKN: 13, STREAM(8, offset:461168601842738790 3), PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, CC(FLOW_CONTROL_ERROR)

Table I.17: quinn - quic_server_test_stream_limit_error

quic_server_test_newcoid_rtp_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1242	Handshake, DCID=0000000000000000, SCID=d52fc3bc926c6382, PKN: 0, CRYPTO, PADDING
4443	4987	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4987	4443	158	Handshake, DCID=d52fc3bc926c6382, SCID=0000000000000000, PKN: 13, CRYPTO, PADDING
4443	4987	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4987	4443	127	Protected Payload (KP0), DCID=4d55bdad75f122fc, PKN: 5, STREAM(8), STREAM(4), PADDING
4443	4987	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, ACK
4443	4987	76	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, MS
4443	4987	119	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, MS, STREAM(4)
4443	4987	116	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, STREAM(8)
4443	4987	101	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, NCI
4443	4987	74	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, DONE, ACK
4987	4443	454	Protected Payload (KP0), DCID=4d55bdad75f122fc, PKN: 23, ACK, STREAM(20), NCI(Seq:32824, RTP:48365), ACK, STREAM(24), ACK, STREAM(32), STREAM(12), PADDING
4443	4987	81	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, CC(FRAME_ENCODING_ERROR)

Table I.18: quinn - quic_server_test_newcoid_rtp_error

quic_server_test_newcoid_length_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1242	Handshake, DCID=0000000000000000, SCID=360b559d01af5279, PKN: 0, CRYPTO, PADDING
4443	4987	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4987	4443	158	Handshake, DCID=360b559d01af5279, SCID=0000000000000000, PKN: 15, CRYPTO, PADDING
4443	4987	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4987	4443	131	Protected Payload (KP0), DCID=360b559d01af5279, PKN: 8, STREAM(4), NCI(CIDLength:0, CID<MISSING>), PADDING
4443	4987	81	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, CC(FRAME_ENCODIND_ERROR)

Table I.19: quinn - quic_server_test_newcoid_length_error

quic_server_test_max_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=50a9a26b4ab7f048, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=50a9a26b4ab7f048, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, NT, PADDING
4987	4443	107	Protected Payload (KP0), DCID=50a9a26b4ab7f048, PKN: 15, STREAM(28), PADDING
4443	4987	481	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, STREAM(28), PADDING
4987	4443	96	Protected Payload (KP0), DCID=50a9a26b4ab7f048, PKN: 16, MS(4611686018427387903), PADDING
4443	4987	80	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, CC(STREAM_LIMIT_ERROR)

Table I.20: picoquic - quic_server_test_max_error

quic_server_test_handshake_done_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1294	Handshake, DCID=0000000000000000, SCID=687edca1f72bdfa9, PKN: 0, CRYPTO
4443	4988	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, NCI, PADDING
4443	4988	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4988	4443	158	Handshake, DCID=687edca1f72bdfa9, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
4443	4988	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, NCI, PADDING
4443	4988	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4988	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4987	4443	88	Protected Payload (KP0), DCID=c4354d567c38dc23, PKN: 4, DONE, PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, CC(PROTOCOL_VIOLATION)

Table I.21: `picoquic` - quic_server_test_handshake_done_error

quic_server_test_new_token_error

port src	port dst	Length	Info
4988	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4988	1242	Handshake, DCID=0000000000000000, SCID=b157adbb675af597, PKN: 0, CRYPTO, PADDING
4443	4988	96	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI
4988	4443	158	Handshake, DCID=a2df8457283bb5e9, SCID=0000000000000000, PKN: 20, CRYPTO, PADDING
4443	4988	71	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, DONE, PADDING
4988	4443	96	Protected Payload (KP0), DCID=a2df8457283bb5e9, PKN: 6, NT, ACK, PADDING
4443	4988	93	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, CC(PROTOCOL_VIOLATION)

Table I.22: `quinn` - quic_server_test_new_token_error

quic_server_test_token_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
Token Length: 2 Token: 340f			
4443	4987	97	Initial, DCID=0000000000000000, SCID=69fa0eb67383066a, PKN: 0, ACK, CC(INVALID_TOKEN)

Table I.23: `picoquic` - quic_server_test_token_error

quic_server_test_max_limit_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=572821fb360eb9f4, PKN: 0, CRYPTO
Parameter: initial_max_streams_bidi (len=2) 513			
4443	4987	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=572821fb360eb9f4, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4987	4443	110	Protected Payload (KP0), DCID=572821fb360eb9f4, PKN: 18, STREAM(37756), PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, CC(STREAM_LIMIT_ERROR)

Table I.24: `picoquic` - quic_server_test_max_limit_error

quic_server_test_newconnectionid_error

port src	port dst	Length	Info
4987	4443	1274	Initial, DCID=0000000000000001, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
4443	4987	1294	Handshake, DCID=0000000000000000, SCID=990df1b10b43bbc8, PKN: 0, CRYPTO
4443	4987	506	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NT, PADDING
4443	4987	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
4987	4443	158	Handshake, DCID=990df1b10b43bbc8, SCID=0000000000000000, PKN: 19, CRYPTO, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, DONE, PADDING
4443	4987	1442	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, PING, PADDING
4443	4987	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, NT, PADDING
4443	4987	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, DONE, PADDING
4987	4443	167	Protected Payload (KP0), DCID=990df1b10b43bbc8, PKN: 6, STREAM(20), ACK, STREAM(12), ACK, NCI(Seq: 36854, RPT: 35143, CID Length: 4,CID: 00000008,SRT: 00000000000000000000000003018064b), PADDING
4443	4987	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, CC(PROTOCOL_VIOLATION)

Table I.25: picoquic - quic_server_test_newconnectionid_error

I.2 Client

quic_client_test_stream

port src	port dst	Length	Info
44683	4443	1294	Initial, DCID=518cf93c18f44e60, SCID=c2c9ebba45a1aea1, PKN: 0, CRYPTO, PADDING
44683	4443	1294	Initial, DCID=518cf93c18f44e60, SCID=c2c9ebba45a1aea1, PKN: 1, CRYPTO, PADDING
4443	44683	1274	Initial, DCID=c2c9ebba45a1aea1, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
44683	4443	95	Handshake, DCID=0000000000000000, SCID=c2c9ebba45a1aea1, PKN: 0, PADDING
4443	44683	1442	Handshake, DCID=c2c9ebba45a1aea1, SCID=0000000000000000, PKN: 10, CRYPTO, PADDING
44683	4443	186	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, PADDING
44683	4443	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, STREAM(0), PADDING
44683	4443	225	Handshake, DCID=0000000000000000, SCID=c2c9ebba45a1aea1, PKN: 2, CRYPTO, ACK, PADDING
4443	44683	93	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 6, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 15, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, PADDING
4443	44683	95	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 29, STREAM(0), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, STREAM(4), PADDING
4443	44683	92	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 31, ACK, PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 46, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, PADDING
4443	44683	107	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 57, ACK, DONE, STREAM(4), ACK, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, STREAM(8), PADDING
4443	44683	98	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 63, DONE, DONE, STREAM(8), DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, ACK, STREAM(12)
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, STREAM(8), PADDING
4443	44683	108	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 79, DONE, ACK, ACK, ACK, DONE, DONE, DONE, DONE, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, STREAM(12), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 91, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, STREAM(8), PADDING
4443	44683	102	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 104, DONE, STREAM(12), ACK, DONE, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, ACK, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, STREAM(12), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, STREAM(8), PADDING
4443	44683	99	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 120, ACK, DONE, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(16), PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 127, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 20, STREAM(12), PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 142, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 21, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 22, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 23, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 24, STREAM(12), PADDING
4443	44683	94	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 150, DONE, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 25, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 26, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 27, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 28, STREAM(12), PADDING
4443	44683	102	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 158, STREAM(16), DONE, ACK, DONE, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 29, ACK, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 30, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 31, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 32, STREAM(12), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 159, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 33, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 34, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 35, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 36, STREAM(12), PADDING
4443	44683	93	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 174, DONE, ACK, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 37, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 38, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 39, STREAM(8), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 182, ACK, PADDING

Table I.26: `picoquic` - quic_client_test_stream

quic_client_test_max

port src	port dst	Length	Info
46374	4443	1294	Initial, DCID=97559493a8c58eb3, SCID=2ba6d84031c3ac4c, PKN: 0, CRYPTO, PADDING
4443	46374	1274	Initial, DCID=2ba6d84031c3ac4c, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
46374	4443	95	Handshake, DCID=0000000000000001, SCID=2ba6d84031c3ac4c, PKN: 0, PADDING
4443	46374	1439	Handshake, DCID=2ba6d84031c3ac4c, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
46374	4443	186	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NCI, PADDING
46374	4443	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, STREAM(0), PADDING
4443	46374	88	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 7, DONE, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, NCI, ACK, PADDING
46374	4443	1442	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, PING, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, STREAM(0), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 7, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 8, STREAM(0), PADDING
4443	46374	117	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 23, MD, ACK, ACK, STREAM(0), ACK, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 9, ACK, STREAM(4), PADDING
4443	46374	92	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 29, ACK, PADDING
4443	46374	89	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 31, MD, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 10, ACK, PADDING
4443	46374	92	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 46, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 11, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 12, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 13, STREAM(0), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 14, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 15, STREAM(0), PADDING
4443	46374	125	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 67, ACK, ACK, STREAM(4), ACK, ACK, ACK, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 16, ACK, STREAM(8), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 17, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 18, STREAM(0), PADDING
4443	46374	102	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 76, ACK, ACK, ACK, PADDING
4443	46374	92	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 78, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 19, ACK, STREAM(8), PADDING
4443	46374	95	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 79, STREAM(8), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 20, ACK, STREAM(12), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 21, NCI, ACK, PADDING
4443	46374	100	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 94, ACK, MS, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 22, ACK, STREAM(0), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 23, NCI, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 24, STREAM(8), STREAM(12), PADDING
4443	46374	97	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 114, ACK, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 25, ACK, STREAM(0), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 26, NCI, ACK, PADDING
4443	46374	92	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 127, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 27, ACK, STREAM(8), STREAM(12), PADDING
4443	46374	95	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 140, STREAM(12), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 28, ACK, STREAM(16), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 29, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 30, STREAM(0), PADDING
4443	46374	97	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 148, ACK, ACK, PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 31, ACK, STREAM(8), STREAM(12), STREAM(16), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 32, NCI, ACK, PADDING
46374	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 33, STREAM(0), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 34, STREAM(8), STREAM(12), STREAM(16), PADDING
46374	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 35, NCI, ACK, PADDING
4443	46374	107	Protected Payload (KP0), DCID=2ba6d84031c3ac4c, PKN: 156, ACK, ACK, ACK, ACK, PADDING

Table I.27: `picoquic` - quic_client_test_max

quic_client_test_accept_maxdata

port src	port dst	Length	Info
47520	4443	1242	Initial, DCID=57dec30be87769f9d4f2f2f4e72cfd, SCID=a3c6c59b, PKN: 0, PADDING, CRYPTO, PADDING
4443	47520	1274	Initial, DCID=a3c6c59b, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
47520	4443	87	Initial, DCID=0000000000000001, SCID=a3c6c59b, PKN: 1, ACK
4443	47520	1442	Handshake, DCID=a3c6c59b, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
47520	4443	2062	Handshake, DCID=0000000000000001, SCID=a3c6c59b, PKN: 0, ACK, PADDING, CRYPTO
47520	4443	147	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, PADDING, STREAM(0), NCI
4443	47520	96	Protected Payload (KP0), DCID=a3c6c59b, PKN: 1, DONE, MD, MD, MD, MD, MD, MD, PADDING
47520	4443	74	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, ACK
4443	47520	107	Protected Payload (KP0), DCID=b9e6d88f, PKN: 5, MD, SS(4), MD, SS(4), ACK, MSD(4), PADDING
47520	4443	76	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, ACK
4443	47520	234	Protected Payload (KP0), DCID=a3c6c59b, PKN: 9, MD, SS(4), MD, SS(4), ACK, MSD(4), MD, MSD(4), MD, MD, SS(4), MD, MD, MD, MD, MD, MS, MD, MS, MSD(4), MD, MSD(4), MD, MS, MS, MSD(4), MSD(4), ACK, MD, STREAM(0)
47520	4443	74	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, ACK
47520	4443	123	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, PADDING, STREAM(8)
4443	47520	146	Protected Payload (KP0), DCID=b9e6d88f, PKN: 13, SS(4), MD, MS, SS(4), MSD(4), MS, MD, SS(4), MD, MD, STREAM(8), MS, MD, MS, MS, MSD(8), PADDING
47520	4443	76	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, ACK
47520	4443	123	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, ACK, PADDING, STREAM(12)
4443	47520	148	Protected Payload (KP0), DCID=b9e6d88f, PKN: 17, SS(4), MD, MS, SS(4), MSD(4), MS, MD, SS(4), MD, MD, STREAM(8), MS, MD, MS, MS, MSD(8), MD, PADDING
47520	4443	78	Protected Payload (KP0), DCID=0000000000000001, PKN: 7, ACK

Table I.28: `picoquic` - quic_client_test_accept_maxdata

quic_client_test_ext_min_ack_delay

port src	port dst	Length	Info
55535	4443	1294	Initial, DCID=67e31d5de8ca8200, SCID=fa7049971203cbdf, PKN: 0, CRYPTO, PADDING
4443	55535	1274	Initial, DCID=fa7049971203cbdf, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
55535	4443	95	Handshake, DCID=0000000000000000, SCID=fa7049971203cbdf, PKN: 0, PADDING
4443	55535	1449	Handshake, DCID=fa7049971203cbdf, SCID=0000000000000000, PKN: 11, CRYPTO, PADDING
55535	4443	186	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, PADDING
55535	4443	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, STREAM(0), PADDING
55535	4443	225	Handshake, DCID=0000000000000000, SCID=fa7049971203cbdf, PKN: 2, CRYPTO, ACK, PADDING
4443	55535	92	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 6, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK_FREQ, PING, PADDING
4443	55535	93	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 15, DONE, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, PADDING
4443	55535	92	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 35, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, PADDING
4443	55535	105	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 42, ACK, ACK, STREAM(0), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, STREAM(4), PADDING
4443	55535	92	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 46, ACK, PADDING
4443	55535	100	Protected Payload (KP0), DCID=fa7049971203cbdf, PKN: 47, STREAM(4), ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, STREAM(8), PADDING
4443	55535	95	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 63, STREAM(8), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, ACK, STREAM(12), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, STREAM(8), PADDING
4443	55535	97	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 84, ACK, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, ACK, PADDING
4443	55535	95	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 92, STREAM(12), PADDING
55535	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, ACK, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, STREAM(8), PADDING
4443	55535	97	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 93, ACK, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, STREAM(8), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, STREAM(8), PADDING
4443	55535	115	Protected Payload (KP0), DCID=fa7049971203cbdf, PKN: 110, ACK, STREAM(16), ACK, ACK, ACK, PADDING
55535	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, STREAM(20), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, STREAM(8), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 20, STREAM(20), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 21, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 22, STREAM(8), PADDING
4443	55535	102	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 111, ACK, ACK, ACK, PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 23, STREAM(20), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 24, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 25, STREAM(8), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 26, STREAM(20), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 27, STREAM(16), PADDING
55535	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 28, STREAM(8), PADDING
4443	55535	105	Protected Payload (KP0), DCID=fa7049971203cbdf, PKN: 127, STREAM(20), ACK, ACK, PADDING
55535	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 29, ACK, STREAM(24), PADDING
4443	55535	102	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 142, ACK, ACK, ACK, PADDING
4443	55535	92	Protected Payload (KP0), DCID=fa7049971203cbdf, PKN: 150, ACK, PADDING
4443	55535	105	Protected Payload (KP0), DCID=33e29d97213ea938, PKN: 157, ACK, ACK, STREAM(24), PADDING
4443	55535	92	Protected Payload (KP0), DCID=fa7049971203cbdf, PKN: 158, ACK, PADDING

Table I.29: `picoquic` - quic_client_test_ext_min_ack_delay

quic_client_test_unknown

port src	port dst	Length	Info
35232	4443	1294	Initial, DCID=455ff5a111dee88e, SCID=69ee40236608f9eb, PKN: 0, CRYPTO, PADDING
35232	4443	1294	Initial, DCID=455ff5a111dee88e, SCID=69ee40236608f9eb, PKN: 1, CRYPTO, PADDING
4443	35232	1274	Initial, DCID=69ee40236608f9eb, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
35232	4443	95	Handshake, DCID=0000000000000000, SCID=69ee40236608f9eb, PKN: 0, PADDING
4443	35232	1439	Handshake, DCID=69ee40236608f9eb, SCID=0000000000000000, PKN: 6, CRYPTO, PADDING
35232	4443	186	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, PADDING
35232	4443	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
35232	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, STREAM(0), PADDING
35232	4443	225	Handshake, DCID=0000000000000000, SCID=69ee40236608f9eb, PKN: 2, CRYPTO, ACK, PADDING
35232	4443	225	Handshake, DCID=0000000000000000, SCID=69ee40236608f9eb, PKN: 3, CRYPTO, ACK, PADDING
4443	35232	117	Protected Payload (KP0), DCID=dbe8e40b768aa23a, PKN: 14, PADDING, UNKNOWN
35232	4443	73	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, CC(FRAME_ENCODING_ERROR)

Table I.30: `picoquic` - quic_client_test_unknown

quic_client_test_unknown_tp

port src	port dst	Length	Info
44683	4443	1294	Initial, DCID=518cf93c18f44e60, SCID=c2c9ebba45a1aea1, PKN: 0, CRYPTO, PADDING
44683	4443	1294	Initial, DCID=518cf93c18f44e60, SCID=c2c9ebba45a1aea1, PKN: 1, CRYPTO, PADDING
4443	44683	1274	Initial, DCID=c2c9ebba45a1aea1, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
44683	4443	95	Handshake, DCID=0000000000000000, SCID=c2c9ebba45a1aea1, PKN: 0, PADDING
4443	44683	1442	Handshake, DCID=c2c9ebba45a1aea1, SCID=0000000000000000, PKN: 10, CRYPTO, PADDING
44683	4443	186	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, PADDING
44683	4443	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, STREAM(0), PADDING
44683	4443	225	Handshake, DCID=0000000000000000, SCID=c2c9ebba45a1aea1, PKN: 2, CRYPTO, ACK, PADDING
4443	44683	93	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 6, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, ACK, PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 15, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 4, ACK, PADDING
4443	44683	95	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 29, STREAM(0), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 5, ACK, STREAM(4), PADDING
4443	44683	92	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 31, ACK, PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 46, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 6, ACK, PADDING
4443	44683	107	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 57, ACK, DONE, STREAM(4), ACK, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 7, ACK, STREAM(8), PADDING
4443	44683	98	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 63, DONE, DONE, STREAM(8), DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 8, ACK, STREAM(12)
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 9, STREAM(8), PADDING
4443	44683	108	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 79, DONE, ACK, ACK, ACK, DONE, DONE, DONE, DONE, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 10, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 11, STREAM(12), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 91, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 12, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 13, STREAM(8), PADDING
4443	44683	102	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 104, DONE, STREAM(12), ACK, DONE, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 14, ACK, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 15, STREAM(12), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 16, STREAM(8), PADDING
4443	44683	99	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 120, ACK, DONE, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 17, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 18, STREAM(16), PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 127, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 19, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 20, STREAM(12), PADDING
4443	44683	88	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 142, DONE, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 21, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 22, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 23, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 24, STREAM(12), PADDING
4443	44683	94	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 150, DONE, DONE, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 25, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 26, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 27, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 28, STREAM(12), PADDING
4443	44683	102	Protected Payload (KP0), DCID=c2c9ebba45a1aea1, PKN: 158, STREAM(16), DONE, ACK, DONE, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 29, ACK, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 30, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 31, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 32, STREAM(12), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 159, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 33, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 34, STREAM(8), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 35, STREAM(16), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 36, STREAM(12), PADDING
4443	44683	93	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 174, DONE, ACK, PADDING
44683	4443	225	Protected Payload (KP0), DCID=0000000000000000, PKN: 37, ACK, PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 38, STREAM(20), PADDING
44683	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 39, STREAM(8), PADDING
4443	44683	92	Protected Payload (KP0), DCID=70fc0232fd27b982, PKN: 182, ACK, PADDING

Table I.31: `picoquic` - quic_client_test_unknown_tp

quic_client_test_tp_acticoid_error

port src	port dst	Length	Info
40508	4443	1242	Initial, DCID=2ed473bd96f7596dafd8b895a636d594, SCID=94b384db, PKN: 0, PADDING, CRYPTO, PADDING
4443	40508	1274	Initial, DCID=94b384db, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
Extension: quic_transport_parameters (drafts version) (len=64)			
	Parameter: active_connection_id_limit (len=1) 0		
40508	4443	87	Initial, DCID=0000000000000000, SCID=94b384db, PKN: 1, ACK
4443	40508	1446	Handshake, DCID=94b384db, SCID=0000000000000000, PKN: 2, CRYPTO, PADDING
40508	4443	104	Protected Payload (KP0), DCID=0000000000000000
40508	4443	122	Handshake, DCID=0000000000000000, SCID=94b384db, PKN: 0, ACK, CC(TRANSPORT_PARAMETER_ERROR)

Table I.32: quant - quic_client_test_tp_activoid_error

quic_client_test_tp_error

port src	port dst	Length	Info
55430	4443	1294	Initial, DCID=a7008226bff2f21c, SCID=a9f6146b9d2cda23, PKN: 0, CRYPTO, PADDING
4443	55430	1274	Initial, DCID=a9f6146b9d2cda23, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
55430	4443	95	Handshake, DCID=0000000000000000, SCID=a9f6146b9d2cda23, PKN: 0, PADDING
4443	55430	1443	Handshake, DCID=a9f6146b9d2cda23, SCID=0000000000000000, PKN: 2, CRYPTO, PADDING
Extension: quic_transport_parameters (drafts version) (len=57)			
	Parameter: ack_delay_exponent (len=2) 15287		
55430	4443	96	Handshake, DCID=0000000000000000, SCID=a9f6146b9d2cda23, PKN: 1, ACK, CC(TRANSPORT_PARAMETER_ERROR)

Table I.33: picoquic - quic_client_test_tp_error

quic_client_test_no_odci

port src	port dst	Length	Info
52640	4443	1294	Initial, DCID=ba758b55ac5be12d, SCID=95e3c51873a6358f, PKN: 0, CRYPTO, PADDING
52640	4443	1294	Initial, DCID=ba758b55ac5be12d, SCID=95e3c51873a6358f, PKN: 1, CRYPTO, PADDING
4443	52640	1274	Initial, DCID=95e3c51873a6358f, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
Extension: quic_transport_parameters (drafts version) (len=52)			
	Parameter: initial_max_streams_bidi (len=1) 10		
	Parameter: initial_max_stream_data_bidi_local (len=4) 213069144		
	Parameter: initial_max_data (len=4) 213069144		
	Parameter: max_idle_timeout (len=4) 213069143 ms		
	Parameter: initial_max_stream_data_bidi_remote (len=4) 213069144		
	Parameter: initial_max_stream_data_uni (len=4) 213069144		
52640	4443	95	Handshake, DCID=0000000000000001, SCID=95e3c51873a6358f, PKN: 0, PADDING
4443	52640	1438	Handshake, DCID=95e3c51873a6358f, SCID=0000000000000001, PKN: 18, CRYPTO, PADDING
52640	4443	96	Handshake, DCID=0000000000000001, SCID=95e3c51873a6358f, PKN: 1, ACK, CC(TRANSPORT_PARAMETER_ERROR)

Table I.34: picoquic - quic_client_test_no_ocid

quic_client_test_double_tp_error

port src	port dst	Length	Info
52640	4443	1294	Initial, DCID=9f973857dea25b36, SCID=6618b4747475f279, PKN: 0, CRYPTO, PADDING
52640	4443	1294	Initial, DCID=9f973857dea25b36, SCID=6618b4747475f279, PKN: 1, CRYPTO, PADDING
4443	52640	1274	Initial, DCID=6618b4747475f279, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
Extension: quic_transport_parameters (drafts version) (len=52)			
	Parameter: initial_max_streams_bidi (len=1) 10		
	Parameter: initial_max_streams_bidi (len=1) 20		
52640	4443	95	Handshake, DCID=0000000000000001, SCID=6618b4747475f279, PKN: 0, PADDING
4443	52640	1438	Handshake, DCID=6618b4747475f279, SCID=0000000000000001, PKN: 18, CRYPTO, PADDING
52640	4443	96	Handshake, DCID=0000000000000001, SCID=6618b4747475f279, PKN: 1, ACK, CC(TRANSPORT_PARAMETER_ERROR)

Table I.35: picoquic - quic_client_test_double_tp_error

quic_client_test_tp_prefadd_error

src port	dst port	Information
59841	4443	1322 Initial, DCID=334504622eb55b3a, SCID=6da5a4943d7ae98d, PKN: 0, CRYPTO, PADDING
4443	59841	1274 I nitial, DCID=6da5a4943d7ae98d, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
59841	4443	92 Initial, DCID=0000000000000001, SCID=6da5a4943d7ae98d, PKN: 1, ACK
4443	59841	1510 Handshake, DCID=6da5a4943d7ae98d, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
Parameter: preferred_address (len=41) Type: preferred_address (0x0d) Length: 41 Value: 7f000001115b00000000000000007f0000017f000002115b00000000000000000007f0000... ipv4Address: localhost (127.0.0.1) ipv4Port: 4443 ipv6Address: ::7f00:1:7f00:2 (::7f00:1:7f00:2) ipv6Port: 4443 Length: 0 connectionId: <MISSING> statelessResetToken: 00000000000000007f0000017f000001		
59841	4443	96 Handshake, DCID=0000000000000000, SCID=6da5a4943d7ae98d, PKN: 1, ACK, CC(TRANSPORT_PARAMETER_ERROR)

Table I.36: picoquic - quic_client_test_tp_prefadd_error

quic_client_test_new_token_error

port src	port dst	Length	Info
57409	4443	1294	Initial, DCID=adfeb1ff6495cd18, SCID=cf80ccb562ddb66e, PKN: 0, CRYPTO, PADDING
57409	4443	1294	Initial, DCID=adfeb1ff6495cd18, SCID=cf80ccb562ddb66e, PKN: 1, CRYPTO, PADDING
4443	57409	1274	Initial, DCID=cf80ccb562ddb66e, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
57409	4443	95	Handshake, DCID=0000000000000001, SCID=cf80ccb562ddb66e, PKN: 0, PADDING
4443	57409	1439	Handshake, DCID=cf80ccb562ddb66e, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
57409	4443	186	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NCI, PADDING
57409	4443	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
57409	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, STREAM(0), PADDING
4443	57409	88	Protected Payload (KP0), DCID=cf80ccb562ddb66e, PKN: 1, DONE, PADDING
57409	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, ACK, PADDING
57409	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, NCI, ACK, PADDING
57409	4443	1442	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, PING, PADDING
57409	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 6, STREAM(0), PADDING
57409	4443	225	Protected Payload (KP0), DCID=0000000000000001, PKN: 7, NCI, ACK, PADDING
4443	57409	102	Protected Payload (KP0), DCID=cf80ccb562ddb66e, PKN: 17, DONE, ACK, ACK, NT(Length:0, Token:<missing>), PADDING
57409	4443	81	Protected Payload (KP0), DCID=0000000000000001, PKN: 8, ACK, CC(FRAME_ENCODING_ERROR)

Table I.37: picoquic - quic_client_test_new_token_error

quic_client_test_max_limit_error

port src	port dst	Length	Info
60674	4443	1294	Initial, DCID=c80f58b46354d228, SCID=9eb5d4c36420500d, PKN: 0, CRYPTO, PADDING
60674	4443	1294	Initial, DCID=c80f58b46354d228, SCID=9eb5d4c36420500d, PKN: 1, CRYPTO, PADDING
Extension: quic_transport_parameters (drafts version) (len=83) Parameter: initial_max_streams_bidi (len=2) 513			
4443	60674	1274	Initial, DCID=9eb5d4c36420500d, SCID=0000000000000000, PKN: 7, CRYPTO, PADDING
60674	4443	95	Handshake, DCID=0000000000000000, SCID=9eb5d4c36420500d, PKN: 0, PADDING
4443	60674	1439	Handshake, DCID=9eb5d4c36420500d, SCID=0000000000000000, PKN: 11, CRYPTO, PADDING
60674	4443	186	Protected Payload (KP0), DCID=0000000000000000, PKN: 0, NCI, PADDING
60674	4443	1482	Protected Payload (KP0), DCID=0000000000000000, PKN: 1, PING, PADDING
60674	4443	97	Protected Payload (KP0), DCID=0000000000000000, PKN: 2, STREAM(0), PADDING
4443	60674	98	Protected Payload (KP0), DCID=9eb5d4c36420500d, PKN: 18, STREAM(20660), PADDING
60674	4443	72	Protected Payload (KP0), DCID=0000000000000000, PKN: 3, CC(STREAM_LIMIT_ERROR)

Table I.38: picoquic - quic_client_test_max_limit_error

quic_client_test_blocked_streams_maxstream_error

port src	port dst	Length	Info
52871	4443	1294	Initial, DCID=c87bdbc49fac256072621b16d8, SCID=c9654e8d945cbc81, PKN: 0, PADDING, CRYPTO
52871	4443	1294	Initial, DCID=c87bdbc49fac256072621b16d8, SCID=c9654e8d945cbc81, PKN: 1, PADDING, CRYPTO
52871	4443	1294	Initial, DCID=c87bdbc49fac256072621b16d8, SCID=c9654e8d945cbc81, PKN: 2, PADDING, CRYPTO
4443	52871	1274	Initial, DCID=c9654e8d945cbc81, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
52871	4443	1294	Initial, DCID=0000000000000001, SCID=c9654e8d945cbc81, PKN: 3, ACK, PADDING
4443	52871	1456	Handshake, DCID=c9654e8d945cbc81, SCID=0000000000000001, PKN: 6, CRYPTO, PADDING
52871	4443	129	Handshake, DCID=0000000000000001, SCID=c9654e8d945cbc81, PKN: 0, ACK, CRYPTO
52871	4443	85	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, STREAM(0)
52871	4443	124	Handshake, DCID=0000000000000001, SCID=c9654e8d945cbc81, PKN: 1, CRYPTO
52871	4443	124	Handshake, DCID=0000000000000001, SCID=c9654e8d945cbc81, PKN: 2, CRYPTO
4443	52871	88	Protected Payload (KP0), DCID=c9654e8d945cbc81, PKN: 14, DONE, PADDING
52871	4443	90	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, ACK, STREAM(0)
52871	4443	85	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, STREAM(0)
52871	4443	1394	Protected Payload (KP0), DCID=0000000000000001, PKN: 4, PADDING, PING
4443	52871	105	Protected Payload (KP0), DCID=c9654e8d945cbc81, PKN: 32, SB(Stream Limit: 4611686018427387903), SB, PADDING
52871	4443	125	Protected Payload (KP0), DCID=0000000000000001, PKN: 5, CC(FRAME_ENCODING_ERROR)

Table I.39: `picoquic` - quic_client_test_blocked_blocked_limit_error

quic_client_test_retirecoid_error

port src	port dst	Length	Info
39475	4443	1294	Initial, DCID=a79adcbebd6e803b, SCID=770585dc24dc0deb, PKN: 0, CRYPTO, PADDING
39475	4443	1294	Initial, DCID=a79adcbebd6e803b, SCID=770585dc24dc0deb, PKN: 1, CRYPTO, PADDING
4443	39475	1274	Initial, DCID=770585dc24dc0deb, SCID=0000000000000001, PKN: 7, CRYPTO, PADDING
39475	4443	95	Handshake, DCID=0000000000000001, SCID=770585dc24dc0deb, PKN: 0, PADDING
4443	39475	1439	Handshake, DCID=770585dc24dc0deb, SCID=0000000000000001, PKN: 19, CRYPTO, PADDING
39475	4443	186	Protected Payload (KP0), DCID=0000000000000001, PKN: 0, NCI, PADDING
39475	4443	1482	Protected Payload (KP0), DCID=0000000000000001, PKN: 1, PING, PADDING
39475	4443	97	Protected Payload (KP0), DCID=0000000000000001, PKN: 2, STREAM(0), PADDING
4443	39475	91	Protected Payload (KP0), DCID=770585dc24dc0deb, PKN: 2, DONE, RCI(1000), PADDING
39475	4443	72	Protected Payload (KP0), DCID=0000000000000001, PKN: 3, CC(PROTOCOL_VIOLATION)

Table I.40: `picoquic` - quic_client_test_retirecoid_error

Appendix J

Example's references

1. MVFST migration error (github.com)
2. AIOQUIC path_challenge error (github.com)
3. AIOQUIC highest probing packet error (github.com)
4. QUINN pending path_challenge error (github.com)
5. QUANT ack-credit error (github.com)
6. QUANT draining packet error (github.com)
7. MVFST no handshake (github.com)
8. QUANT unknown (github.com)
9. AIOQUIC unknown frames (github.com)
10. QUINN unknown frames no error reported (github.com)
11. AIOQUIC active_connection_id_limit error (github.com)
12. QUICHE transport parameters error (github.com)
13. QUANT transport parameters error (github.com)
14. MVFST token error (github.com)
15. QUIC-GO handshake_done error (github.com)
16. PICOQUIC new_token error (github.com)
17. QUINN new_connection_id error (github.com)
18. QUANT new_connection_id error (github.com)
19. PICOQUIC ack-credit error (github.com)

20. QUINN handshake_done error (github.com)

21. LSQUIC unknwown error (github.com)

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**