UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

A test suite for QUIC

Dissertation presented by
Maxime PIRAUX

for obtaining the Master's degree in
Computer Science

Supervisor
Olivier BONAVENTURE

Readers
Quentin DE CONINCK , Marc LOBELLE

Academic year 2017-2018

Acknowledgements

I would like to express my gratitude towards my supervisor Prof. Olivier Bonaventure as well as Quentin De Coninck, who dedicated a part of their time to my thesis throughout the year. I am certain that their involvement has reflected in the quality of my work. The frequent meetings we had, the opportunity of participating to the IETF 101 Hackathon in London I received and their support during the writing of this thesis were invaluable in the achievement of my work.

I would also like to thank Quentin De Coninck and Marc Lobelle for accepting to be the readers of this thesis.

Finally, I am also thankful to my family and friends who supported me throughout my years of education.

Abstract

QUIC is a new transport protocol that is being standardised within the IETF. It is built on top of UDP and is designed to address the shortcomings of TCP. Google has already deployed an experimental version of QUIC and claims that it conveys 7% of the Internet traffic. The IETF QUIC working group in charge of the standardisation process counts fifteen implementations that provide feedback and improvements to the specification.

In this thesis, we propose an independent test suite that verifies the correctness of several QUIC mechanisms with regard to the specification. The test suite exchanges packets with QUIC implementations to evaluate them. We propose a methodology to extract test scenarii from the specification and implement a set of tools to easily create them. A web application allows to visualise the results of the test suite to ease the communication of bug reports to QUIC implementers.

We report the results we collected during a 3-month period starting in March 2018. We present the evolution of QUIC versions during the specification process. We assess the evolution of transport parameters and patterns of retransmission during connection establishment. We detail the bugs and regressions we found involving the flow control mechanism as well as the reordering mechanism of several implementations. We provide evidence that our work has been useful to the IETF QUIC working group and that it can be easily extended for future uses.

All our software is publicly available under an open-source license.

Contents

1 Introduction **1**
- 1.1 Internet and transport protocols . . . 2
  - 1.1.1 The Hypertext Transfer Protocol . . . 3
  - 1.1.2 Securing HTTP with Transport Layer Security . . . 4
  - 1.1.3 The shift of needs about transport protocols . . . 4
- 1.2 The need for a test suite . . . 5

2 State of the art **6**
- 2.1 Recent advances in transport protocols . . . 6
  - 2.1.1 Extending a transport protocol . . . 6
  - 2.1.2 The SPDY experimental protocol . . . 7
  - 2.1.3 Google's QUIC transport protocol . . . 8
  - 2.1.4 Standardising QUIC . . . 11
- 2.2 A detailed look at QUIC . . . 11
  - 2.2.1 Packet format . . . 11
  - 2.2.2 Packet framing . . . 12
  - 2.2.3 Stream multiplexing . . . 13
  - 2.2.4 Acknowledgements . . . 13
  - 2.2.5 Recovery . . . 14
  - 2.2.6 Congestion control . . . 14
  - 2.2.7 Flow control . . . 15
  - 2.2.8 Securing QUIC with TLS . . . 15
  - 2.2.9 Connection migration . . . 16
  - 2.2.10 Other control frames . . . 17
- 2.3 Passive testing and measurements analysis . . . 17
  - 2.3.1 Inferring TCP connection characteristics through passive measurements . . . 18
  - 2.3.2 A passive state-machine approach for accurate analysis of TCP out-of-sequence segments . . . 18
  - 2.3.3 Learning fragments of the TCP network protocol . . . 19
  - 2.3.4 Automated Packet Trace Analysis of TCP Implementations . . . 20
  - 2.3.5 A Finite State Machine Algorithm for Detecting TCP Anomalies . . . 20
  - 2.3.6 TCP Revisited: A Fresh Look at TCP in the Wild . . . 21
- 2.4 Active testing . . . 21
  - 2.4.1 On inferring TCP behavior . . . 22
  - 2.4.2 Misbehaviors in TCP SACK Generation . . . 23
  - 2.4.3 Rule-based Verification of Network Protocol: Implementations using Symbolic Execution . . . 23
  - 2.4.4 TCP Congestion Avoidance Algorithm Identification . . . 24
- 2.5 Alternative approaches . . . 24
  - 2.5.1 Rigorous Specification and Conformance Testing: Techniques for Network Protocols, as applied to TCP, UDP, and Sockets . . . 25

2.5.2 Coping with Nondeterminism in Network Protocol Testing 25

3 Methodology **27**
3.1 Testing approach 27
3.2 Architecture 28
3.3 Designing test scenarii 28
3.4 Tools developed for our task 28
3.4.1 A QUIC toolbox 28
3.4.2 Test traces 29
3.4.3 Test scenarii 31
3.4.4 A web application to visualise test results 33
3.4.5 A QUIC packet dissector 36
3.4.6 Developing Go bindings for picotls 39
3.4.7 Building a web crawler for QUIC hosts discovery 40
3.5 Deploying our work 40
3.6 Communicating with the implementers 41

4 Results **43**
4.1 Measurements and analysis 43
4.1.1 QUIC traffic inside the UCLouvain network 43
4.1.2 Deployment of QUIC during the specification process 45
4.1.3 QUIC transport parameters over time 46
4.1.4 Patterns of retransmission during connection establishment 48
4.2 Case studies 50
4.2.1 Flow control 50
4.2.2 Reordering stream transitions 53

5 Discussion **55**
5.1 Impact of the test suite on the WG 55
5.1.1 Attending the Hackathon 57
5.2 Testing without validation 57
5.3 Future prospects 58

A QUIC transport parameters recorded over time **59**

Chapter 1

Introduction

Recent advances in transport protocols have led to the design of QUIC, a new protocol built on top of UDP. It is currently in the process of standardisation within the IETF and is drawing significant interest from the scientific community. The QUIC working group, which is in charge of this task, is holding frequent meetings. Moreover, there exists fifteen implementations that provide feedback and improvements to the ongoing standard, which attests of this interest.

Implementing network protocols is not a trivial task, as reported by several IETF RFCs [1, 2] and articles of past works which conducted network protocol testing [3, 4, 5, 6, 7, 8]. The bugs they report arise from oversights inherent to programming but also from inconsistencies and ambiguities in the specification. Early implementations of network protocols usually performs testing between each other during the specification phase. This type of tests promotes interoperability, i.e. a shared interpretation and implementation of the specification, rather than correctness with regard to the specification. We intend to address this shortfall for the QUIC protocol.

We propose a test suite to provide feedback to QUIC implementers, in the hope that it will contribute to the specification efforts of QUIC. The test suite verifies the correct implementation of several mechanisms of QUIC and collects several metrics on the behaviour of QUIC implementations. We present in this document the results collected from the tests we conducted between the 8th of March and the 1st of June 2018.

This thesis is comprised of five chapters. The first chapter introduces the context of our work with an history of transport protocols and motivates the need for a test suite when developing network protocols. The second chapter summarises the state of the art of transport protocols and network protocols testing. The third chapter details the methodology we adopted for our work and the software we designed and implemented. The fourth chapter presents results on the evolution of the behaviour of QUIC implementations we collected using the tools developed in this thesis. The fifth chapter concludes this thesis and summarises the impact of our work on the QUIC community.

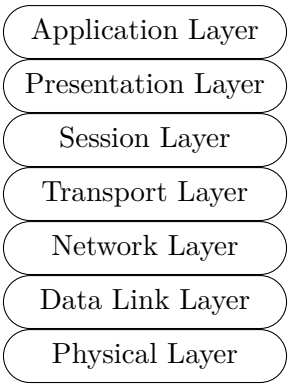


Figure 1.1: The OSI model

1.1 Internet and transport protocols

A commonly accepted definition of transport protocols is found within the fourth layer of the OSI model [9, 10]. This model is illustrated in Figure 1.1. It is composed of seven layers and aims at standardising communications across systems. The layers starts from the physical medium with the first layer to the end-user application at the last layer. The fourth is the *Transport Layer* which provides transparent transfer of data and relieves upper layers from any concern with the detailed manner in which reliable and cost effective transfer of data is achieved [10]. Transport protocols are thus critical for the efficient functioning of higher layers which interact with the end-user.

Transport protocols are ubiquitous in today's use of Internet. Past studies established that a very large amount of IP packets exchanged encapsulate transport protocols. Borgnat et al. studied the evolution of the Internet traffic between 2002 and 2009 on a trans-Pacific backbone link [11]. They found TCP and UDP, two protocols widely accepted as the most-used transport protocols, to be conveyed in more than 90% of the IP packets. This conclusion remains true in recent years.

Researchers also noted a clear dominance in terms of the amount of data exchanged by TCP over the amount of data exchanged by UDP. Lee et al. showed that the ratio of UDP traffic over TCP traffic in terms of bytes varied between 0.02 and 0.11 in 21 network traces captured at different locations during the past decade [12]. In fact, most of the major Internet applications, such as the Web, emails or file transfers rely on TCP while some of them, e.g. the DNS or WebRTC, use UDP. As a result, most of the scientific interest has been drawn by TCP rather than UDP. The bigger complexity of TCP, e.g. because it is a stateful protocol, is also an additional factor of interest.

Both protocols are different in terms of offered features. The Transmission Control Protocol (TCP) is a connection-oriented transport protocol which provides reliable end-to-end transport, data integrity, error recovery, multiplexing and flow control. The User Datagram Protocol (UDP) in comparison is a connection-less protocol that only offers multiplexing and data integrity. Both protocols are used on top of a *Network Layer* and more specifically the IP network.

A connection-oriented transport protocol requires the setup of a connection before application data can be exchanged. Establishing a connection is commonly implemented by exchanging packets to reach agreement on the connection opening. A transport protocol is said to be reliable when it provides notifications to the sender about the effective delivery of the data to the receiver. Data integrity gives the assurance that the data is delivered without being altered by an error during its transmission. End-to-end reliability extends these notifications to the two hosts using the protocol and not only the intermediate network links involved. Error recovery is a mechanism that allows a transport protocol to recover from a loss and to retransmit the missing data until it is acknowledged as received by the other peer. Multiplexing allows a shared medium, e.g. the

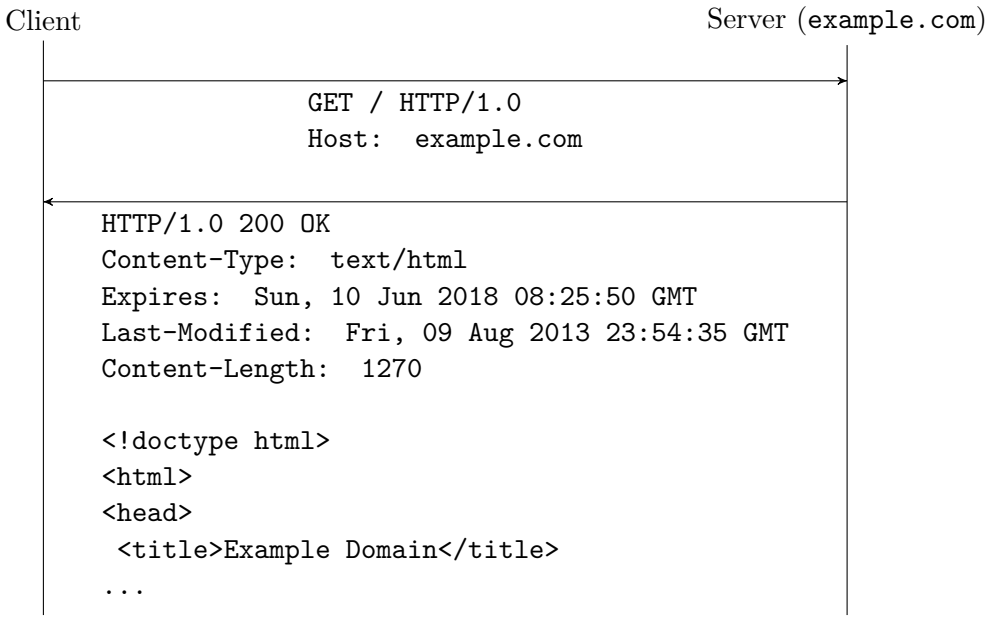


Figure 1.2: A part of an HTTP/1.0 request performed on `example.com` and its response

IP network, to convey multiple connections of this transport protocol between two hosts.

1.1.1 The Hypertext Transfer Protocol

The Hypertext Transfer Protocol, known as HTTP, is the most used application protocol in terms of volume transferred on the Internet [11]. As a result, we shortly tackle its functioning in order to better understand the considerations taken when designing or improving transport protocols. Its first standard version, HTTP/1.0, claims to be designed "with the lightness and speed necessary for distributed, collaborative, hypermedia information systems" [13]. HTTP is a generic, stateless and object-oriented protocol that can be utilised for many tasks through the extensions of its requests. Its syntax allows an open set of headers to define the purpose of a request and the meta-information of a transferred entity. Its most popular use is within the World Wide Web, where it is used to transfer HTML documents.

HTTP was designed as a human-readable protocol and was inspired by the design of the *Internet Text Messages* [14, 15, 16], known today as emails. The headers format of HTTP reuse the one defined in *Multipurpose Internet Mail Extensions* [15, 16], and is therefore constituted of plain text. HTTP uses a request and response paradigm of operation. A client that establish a connection with the server first sends its request using one of the defined HTTP methods and includes headers indicating information about the client, additional request modifiers and a body content if any. The server responds by indicating first the success status of the request, or the lack thereof. Then it includes the server information, the entity meta-information and the body content if any. After a request has been made, the underlying connection should be closed.

Figure 1.2 illustrates a request sent to `example.com` and the received response. We can see that the request indicates the GET method for the base document `/`. The server responds with a status code of 200 indicating the success of the request. The status code line is followed by headers indicating the content type of the entity transferred, its validity and its length. The headers and the body of the entity transferred are separated by a new line.

The version 1.1 of HTTP introduced persistent connections to allow multiple requests to be pipelined into a single connection [17], which greatly improved its performance when navigating web pages constituted of multiple entities.

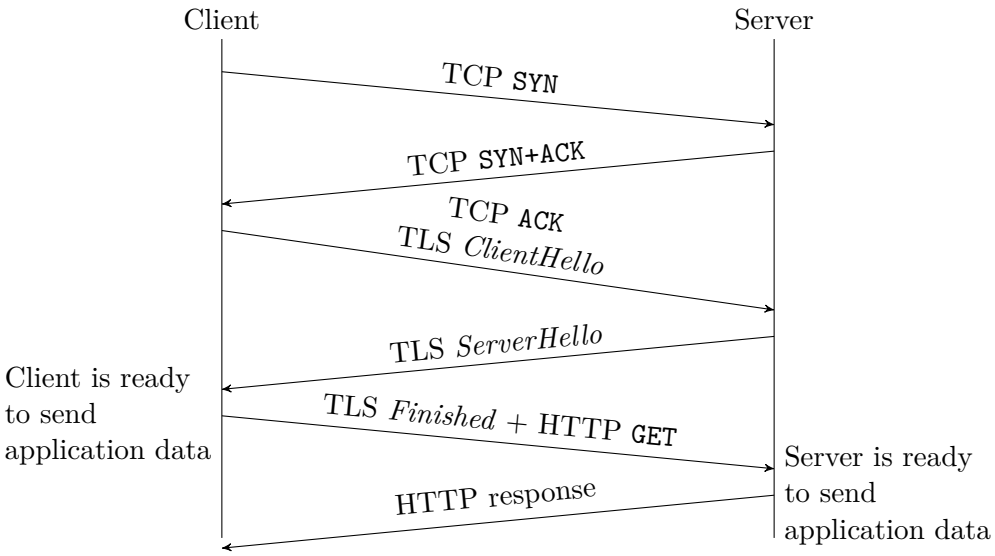


Figure 1.3: Connection establishment when using HTTPS

1.1.2 Securing HTTP with Transport Layer Security

Transport Layer Security is a cryptographic protocol that provides secure and private communications between hosts in a network. It is the successor of Secure Socket Layer, an earlier protocol which it has now replaced because of its design flaws and weak ciphers [18, 19, 20]. TLS and SSL are application independent. One of the use cases for SSL at its time of inception was the growth of online shopping. Because HTTP was consisting of plain text, payment information was sent in clear over the network which explained the reluctance of users at first. SSL was seen as a big improvement, if not a mandatory feature, for this growing market.

TLS is meant to be run on top of a reliable transport protocol [21]. Figure 1.3 illustrates how TLS is integrated between TCP and HTTP. First the peers have to agree on the cryptographic primitives they will use and to establish a shared secret. This is done by exchanging *ClientHello* and *ServerHello* TLS messages. In total, two round trips are required to establish a TLS connection atop TCP before being able to send application data, such as an HTTP request.

The integration of TLS with HTTP was standardised shortly after the TLS standard was established [22, 23]. It defined a new protocol identifier for URIs, i.e. `https`, changed the TCP destination port to 443 and required HTTP to be run over TLS instead of directly over TCP.

1.1.3 The shift of needs about transport protocols

Both TCP and UDP were designed more than three decades ago. A first IETF RFC document specifying TCP was published in 1974 [24], followed by an improved standard in 1981 [25]. UDP was defined in an IETF RFC document in 1980 [26]. This era is considered closer to the ARPANET than the Internet, a time where users, use-cases and devices were a lot more limited than today. ARPANET was designed to interconnect university researchers and super computers. Since then the Internet has grown in several dimensions. The average throughput of each device has greatly increased as well as the number of connected devices. But the nature of the devices themselves has also changed. They have evolved from the stationary computers of ARPANET to the mobile devices that are ubiquitous today. Mobile devices differ from them in many ways. They use wireless access points and can switch between them in a short amount of time. They are also able to use different technologies at the same time to access the Internet, such as WiFi and cellular data.

This evolution resulted in the emergence of other needs for transport protocols, such as multipath connectivity or connection migration. A transport protocol that supports multipath

allows a connection to use multiple network interfaces at the same time to convey its data. Connection migration allows a connection to use a new address without interrupting the flow of data.

New applications such a real-time video streaming also create new needs for transport protocols. In this case, when the loss of a packet happens, it is not always valuable to retransmit its data because of the additional delay induced. Instead the sender should use new techniques to ensure that its transmission is tolerant to packet loss without impacting the transmission delay. Forward error correction is one of these techniques. It adds redundancy when transmitting data to enable the receiver to reconstruct lost packets without requesting retransmissions.

1.2 The need for a test suite

Implementing network protocols is not a trivial task. The specification process, i.e. the production of an informal specification of the behaviour of a network protocol, often requires multiple iterations before reaching a stable point. At this point the specification document is believed to be understood in the same manner by each implementer and to cover the details of the intended behaviour and purpose of the protocol. But even considering the specification to be bug-free, network protocol implementations often reveal several bugs and interoperability issues throughout their lifespan. This can be due to remaining ambiguities in the document, but also because of various oversight inherent to programming in general. TCP has been widely implemented and recurring implementation problems have been catalogued [2]. The DNS specification has been clarified ten years after its inception [1]. Testing needs thus to be conducted at various stages of the protocol life-cycle.

IETF working groups often operate in a feedback loop in which they first lay out the behaviour of the protocol in a formal specification document, then implement it and perform interoperability testing. Based on this testing experience, the document may be updated and the feedback loop started again. Two *Best Current Practices* documents encourage such a way of operating [27, 28].

But other implementations are also created after the standard is established. This is obviously a purpose of standardising a network protocol. At this stage, the role of the test suite is to test for conformance to the specification. The holy grail of testing would be that the test suite is a concrete realisation of the specification. Passing the test suite would be equivalent to conforming to the established standard.

A third interest of having a test suite is for research purposes. A test tool could go beyond pure conformance testing and implement a way of collecting several metrics about a protocol. These metrics can then be used to evaluate the use of a particular feature for instance. This kind of measurement studies helps researchers to understand the actual state of Internet protocols and is very helpful when designing new systems. Finally, a test suite could help finding corner cases that are not addressed by the specification.

Chapter 2

State of the art

We summarise throughout this chapter the state of the art in the context of this thesis. We first review recent advances in transport protocols that inspired the design of QUIC. Then we describe the QUIC protocol and explain in detail the mechanisms needed to gain a better understanding of our work. In the remaining sections of this chapter, we present several scientific studies which conducted network protocol testing using different approaches. We extract several methodological choices and observations relevant to our work.

2.1 Recent advances in transport protocols

We first summarise the improvements that were made to TCP and report issues that arose during their designs and deployments. Then we explain the SPDY protocol. Its experimental results inspired the development of Google's QUIC protocol, for which we present its design document as well as experiments on its deployment at Google. We conclude this section by describing its standardisation process at the IETF.

2.1.1 Extending a transport protocol

A major part of the improvement researches of TCP were focused on improving its throughput. For instance, a clever retransmission mechanism is necessary to maintain a certain quality of service in case of data loss. Congestion control and congestion avoidance algorithms have also drown in significant attention from researchers. They involve a compromise between the pace at which a connection reaches its optimal data transfer rate and the risks of network congestion associated with the eagerness at which a transport protocol transmits data.

These changes involve upgrading the hosts of the Internet rather than the network itself, as the impact on the wire image is often minimal. The wire image is the sequence of messages expressed as a sequence of bits sent by each participant in the protocol, as defined by Trammel et al. [29].

But other improvements may alter the wire image of a protocol. TCP has been designed with the intent of being extensible. It includes an `Options` field in its header format [25]. This allows future extensions to add more state into TCP segments while staying in line with the original specification. However, several studies [30, 31, 32, 33] showed that despite a well-established standardisation of the TCP `Options`, the deployment of improvements reliant on them was much slower than what could have been expected. The reasons outlined are twofold.

Firstly, TCP is often implemented as part of the OS kernel. Upgrading TCP requires thus upgrading the kernel, a decision which is not taken lightly on production servers and not frequent on end-user machines. In turn, developers might not always consider using new improvements as they might not be widely available. Secondly, network devices along an Internet path may interfere and modify TCP segments. Such devices, called middleboxes, are now known to look

further than their intended scope, i.e. the Internet Protocol. It is not safe anymore to assume that a field of the TCP header may not be changed while in transit [34, 35]. Moreover, the rate at which these devices are upgraded is much slower than it is for regular hosts, which impacts the Internet traffic they relay for a long time.

As a result, most of the design effort of the latest TCP improvements has been put into taking these odd behaviours into account and creating a specification that is *de facto* accepted by the variety of middleboxes deployed in the Internet. The development and deployment of MPTCP are good examples of this problem, as well as examples of the work and tradeoff necessary to effectively deploy a major change to TCP [36]. This extension allows a TCP connection to use multiple network paths, with possibly different network addresses. This is a major departure from TCP which is bound to the pair of hosts IP addresses and ports. In 2015 Christoph Paasch from Apple reported that they decided to disable the TCP Fast Open option (TCP TFO) deployed on their devices after discovering that some middleboxes would blacklist sources that use it [37]. TCP TFO is an extension that allows the source to send data during connection establishment in order to cut its latency. Explicit Congestion Notification (ECN) is a joint extension of IP and TCP which has also seen difficulties of deployment since its standardisation [31, 38].

Extending TCP is now considered as rather difficult but not completely impossible. As a result, researchers investigated other ways to improve and evolve the transport layer. This has led to a renewed interest in UDP, which is supposed to be deployed and working while inducing minimal middle-box interferences because of its simplicity. UDP is envisaged as a encapsulation method to deploy new transport protocols. Edeline et al. have investigated this assumption and found it to be verified to a great extent [39]. A small part of Internet networks which are usually located at the edges of Internet, i.e. close to the end-user, do not allow all UDP traffic. They often restrict its use to known services such as DNS. In such cases UDP can only be used to develop new transport protocols if an alternative exists, e.g. a fallback to TCP is possible.

2.1.2 The SPDY experimental protocol

In 2009, Google announced through its research blog a new experimental application-layer protocol designed to improve content delivery over HTTP and minimise latency [40]. A whitepaper outlines three main features [41].

The first feature is stream multiplexing, which allows to create concurrent and interleaved streams of data within a single TCP connection. Navigating a web page involves retrieving the multiple elements that constitutes it. These elements are often fetched using multiple TCP connections. Establishing these connections induce an increased latency. By using multiplexing inside TCP, only a single connection is opened. It also allows to pack several HTTP requests into a single TCP segment.

The second feature is request prioritisation. When HTTP requests are performed in a single TCP connection, congestion control can be upgraded to take into account which resources are more important than others and prioritise their delivery in case of congestion. The HTML document is necessary for a browser to start rendering the page, while images may be delivered after CSS stylesheets and JavaScript scripts.

The third feature is HTTP header compression. HTTP/1.1 introduced document compression but the headers of the protocol remained in clear text. SPDY compresses the requests and responses headers to further reduce the packet size and increase the efficiency of request bundling.

Early experiments showed promising results with a speedup in delivery time ranging from 27% to 60% [41]. But several problems where also met, most of which were either related to TLS or TCP. Multiplexing inside a TCP connection is severely impacted by head-of-line blocking. Head-of-line blocking happens when a loss or reordering occurs during the transmission of a series of segments. Figure 2.1 illustrates this phenomenon. In this example, the client application request two pages from the web server in separate streams. The transmission of page `a` is unsuccessful but page `b` is received by the client. Because TCP delivers data in order to the

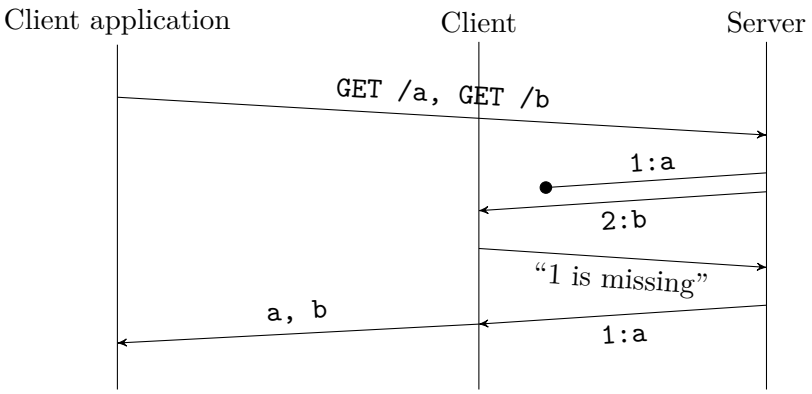


Figure 2.1: Head-of-line blocking phenomenon in TCP

application, it cannot deliver page `b` until page `a` has been received. It thus pause the delivery of data until the missing segment is eventually received. The two pages are delivered to the application in the order they were requested and not in the order they were effectively received. Because of multiplexing, the loss of a TCP segment may also impacts other SPDY streams than the ones bundled in the lost segment. A better multiplexed transport only delays the streams impacted by the loss.

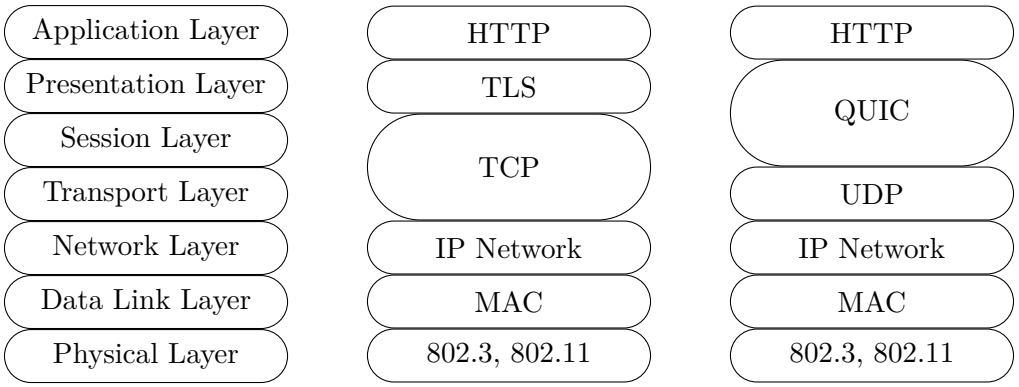
Furthermore, while grouping requests inside a single TCP connection allows to implement prioritisation, it also increase the impact of congestion avoidance. When a packet is lost inside this connection, the TCP congestion window is reduced by a certain factor. In the case where requests are split between several TCP connections, the bandwidth diminution resulting of a loss in one of them is a lot less severe than in the case of a multiplexed TCP connection, because the diminishing factor only affects one of the congestion window.

Lastly, TLS handshakes remains costly in terms of latency as SPDY do not address the encryption layer. A TLS handshake requires several round-trips before application data can be exchanged. This is purely an implementation requirement and not a security requirement. Improvements can therefore be made in this area to decrease the time necessary to establish an encrypted connection.

2.1.3 Google's QUIC transport protocol

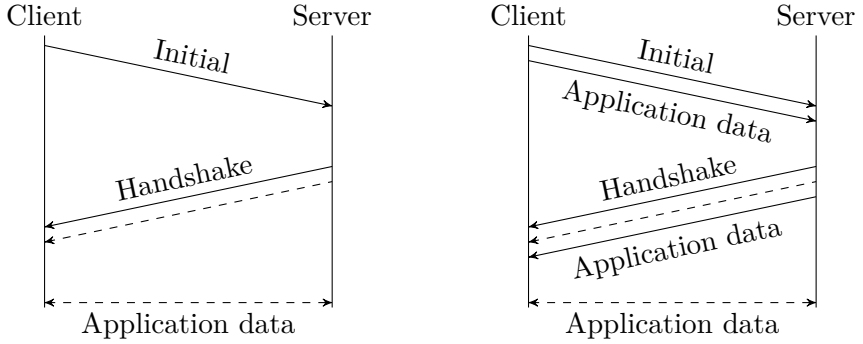
QUIC originally stands for Quick UDP Internet Connection. It first appeared in 2012 as part of the Google Chromium code base and was later announced in 2013 as an experimental transport protocol. A first document explaining the motivations, goals and design philosophy as well as implementation details of QUIC was published in the form of a Google document [42]. An article written by the design team was published later in August 2017 [43]. It explains the testing methodology they used when iterating over the design of QUIC as well as the performance improvements they measured on Google's services.

The design document clearly outlines that the first goal of QUIC is its viability at its time of inception. It is claimed to be designed with the knowledge that middleboxes will block or severely alter any new transport protocols not built upon existing and proven protocols. The choice of using UDP is backed by two main reasons. First, it does not require any change to the middleboxes to deploy a new transport protocol, as verified by Trammel et al. [39]. Secondly, UDP is widely available to common users without needing to update their operating systems or requiring elevated privileges. A QUIC implementation can be ran from user-space without any modification. It is expected that this will allow the protocol to be rapidly iterated upon and its implementation to be upgraded as a regular user program would be. User-space programs are also generally easier to debug and to instrument for experimental purposes.



(a) OSI model (b) A common HTTP stack (c) HTTP stack with QUIC

Figure 2.2: The OSI model compared a common HTTP stack and a QUIC stack



(a) 1-RTT connection establishment (b) 0-RTT connection resumption

Figure 2.3: Connection establishment in QUIC

QUIC is designed to address the shortcomings of TCP. More particularly, its initial intent is to improve the performance of TCP when serving HTTP content over TLS. Figure 2.2 compares a common HTTP stack with the proposed QUIC stack in terms of OSI layering. One of the main issue of the TCP-TLS-HTTP stack is the latency of establishing a connection. Network latency is function of the physical distance between two hosts, or more specifically the physical length of the network path between these hosts. The time for light to travel along this network path is a lower bound on the network latency. Other delays are added by middleboxes such as the delay required to process incoming packets. Reducing the latency is key to improve the responsiveness of user interactions. But despite the growth in bandwidth over the years, the network latency between hosts has not seen the same improvement [44]. A poorly designed application could induce a lot of latency by requiring many round-trips between the hosts. It is thus critical that network protocols include latency considerations into their design.

As shown previously in Figure 1.3, three round-trips are necessary to perform an HTTPS request and receive its response when no prior connection exists. QUIC reduces the added latency of TLS by embedding the encryption layer into the transport layer and by reworking the connection establishment mechanism. The default connection establishment is performed in a single round-trip, as illustrated by Figure 2.3a. After the Handshake packets are sent, the server is able to exchange application data. After their receipt, the client is able too.

QUIC also addresses the rise of mobile devices by making connection resumption less costly in terms of latency. A client that has connected at least once with a given server can resume a connection to it along with application data without waiting for any round trip, as illustrated by Figure 2.3b. The underlying assumption is that mobile devices tend to go offline and switch back online very frequently. But the services they offer may still be running throughout. When

connecting back using QUIC, these services will be able to resume their connections without having to wait any round-trip time before sending application data. This is known as 0-RTT connection resumption.

QUIC is built upon the experiments of SPDY and includes stream multiplexing. But it is able to avoid the head-of-line problem of SPDY over TCP because it implements the transport layer to enable in-order delivery of stream data rather than in-order delivery of packet data.

In their article [43], the design team explains the Internet-scale experiments they conducted inside Google's web browser Chrome, Google Search and YouTube mobile applications, and on their server fleet. They implemented QUIC inside a shared library used in the different services.

Three key application metrics were used to drive the development of QUIC. These are *Search Latency*, *Video Playback Latency* and *Video Rebuffer Rate*.

Search Latency is defined as the delay between the time at which a user enters a search term and the time at which all the search results are delivered to the user. This possibly includes delivering images in the search results. Google estimates an average search to generate a response of 100 kB on desktops, while being of 40 kB on mobile devices. They noted a mean reduction in *Search Latency* for QUIC users of 8% for desktops and 3.6% for mobile devices. They identified the 0-RTT connection resumption as the biggest contributor to reducing the latency. This also explains the difference between desktops and mobile client. As a mobile device changes networks, its IP address is likely to change. As 0-RTT connections can only be established with the same source address, mobile devices are less likely to benefit from it.

Video Latency is the delay between the time at which a user hit the "play" button and the time at which the video effectively starts playing. On the contrary of web searches, the data required to start a video playback cannot be accurately estimated because it depends on the bitrate of the video. The improvement are of 8% for desktops and of 5.3% mobile devices. The same justifications as for *Search Latency* apply here, but the richer signalling for loss recovery of QUIC can explain why the latency reduction for mobile devices is higher than with *Search Latency*. We explain this mechanism in detail in the section 2.2.

Video Rebuffer Rate is the ratio of time spent waiting for a video to resume due to the playback buffer running out of video data, divided by the time spent watching the video. This time is composed of the time waiting for the video to load added with the time spent watching the video effectively playing. This metric is often only appearing in the last percentiles of test results, simply because re-buffering does not happen at all for most of the users. At the 93rd percentile for desktop and the 94th percentile for mobile clients, QUIC was able to eliminate the video re-buffering, showing a 100% improvement. In further percentiles, the gain diminishes and the overall mean of reduction settles at 18% for desktops and 15.3% for mobile clients.

In addition to application metrics results, the design team also outlines areas of potential performance improvements. QUIC offers a small gain when operating in a network with a lot of bandwidth, a low loss rate and a low delay. It may actually even degrade performance compared to TCP. Because their implementation was focused on rapid development and ease of debugging, it resulted in a higher CPU cost. This cost becomes a bottleneck for high throughput connections, with speeds higher than 100Mbps for instance. Moreover, the operating system process scheduler might be unfair between a TCP stack implemented in the kernel and a user program running a QUIC stack. In addition, mobile devices seemed to have less benefited from QUIC than desktops. This is partially due to the nature of those devices. They are indeed more CPU-constrained, but they may also limit the amount of data retrieved because of small screen sizes, which constrains the possible improvements to be made. Mobile devices RAM constraints may also induce applications to stop and to restart in the background, which may disrupt connection state.

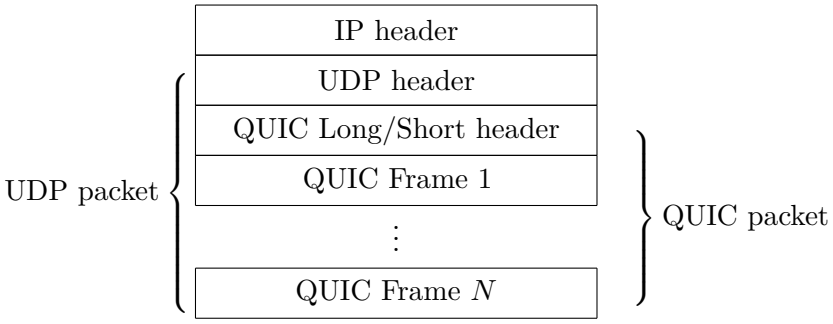


Figure 2.4: A QUIC packet conveyed in an IP network

2.1.4 Standardising QUIC

After starting their large-scale experiments in 2014, the QUIC design team from Google introduced an Internet-Draft to the IETF later in 2015 [45]. Their goal was to evolve from an experiment with interesting results to an Internet standard. An IETF working group was founded in 2016 and adopted the initial draft as a base document for the standardisation of QUIC [46]. Several others drafts detailing the loss recovery mechanism, congestion control mechanism, HTTP mapping and the security aspects of QUIC were adopted as a base [47, 48].

The working group specified its field of action and goals in a charter [46]. One important difference at the start of the working group existence is that while QUIC as implemented by Google uses a custom cryptographic library to secure it, IETF-QUIC uses TLS 1.3. This new version of TLS meets the goals of QUIC by re-working the connection establishment.

The charter also added features that were envisaged but not yet experimented with by Google. Forward error correction is now part of the key goals of QUIC, as well as a multipath extension. Forward error correction is an error recovery mechanism in which redundancy is added to the transmission of data to enable the receiver to correct errors or to retrieve missing packets using added redundancy instead of triggering retransmissions at the sender side. Multipath can be used when a client has more than one interface it wishes to use. A multipath extension should balance the load of the connection across all network interfaces.

The first important milestone for the QUIC working group is the delivery of the core specification documents in November 2018.

2.2 A detailed look at QUIC

We now describe and explains the mechanisms that constitutes QUIC, as defined in the 11th IETF draft of the core protocol [49]. We do not approach the protocol with as much details as in the specification documents, but we rather tackle what is necessary to understand our work. Fields of data structures and parts of the specification may be omitted in the interest of space.

2.2.1 Packet format

A QUIC packet, as illustrated by Figure 2.4, is constituted of a header followed by one or more frames. There are two types of header, a long and short one. Long headers are used at the connection establishment stage. Short headers are used after the connection is established. Figure 2.5 and 2.6 layout their content. Each header defines first the packet type they convey.

The version field is a 32-bit number that allows an implementation to negotiate the protocol version that will be used in accordance to its capabilities. Connection IDs are used to identify the QUIC connection this packet belongs to. Unlike TCP, connections are not identified by the tuple of source and destination ports and addresses but by this 64-bit number. Connection IDs are not shared by peers, each one establishes an ID they will use throughout the connection. The

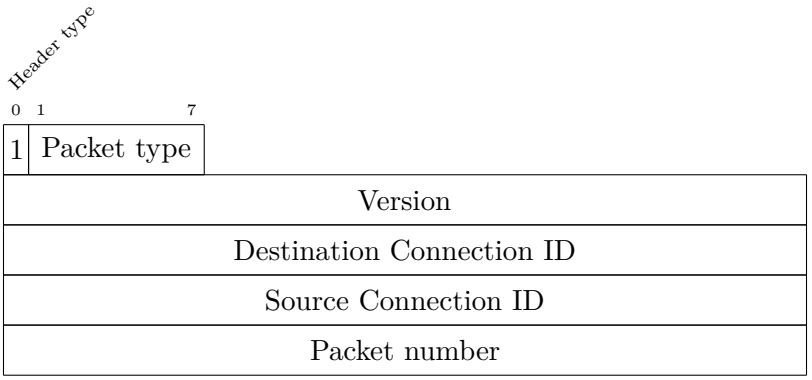


Figure 2.5: Long header

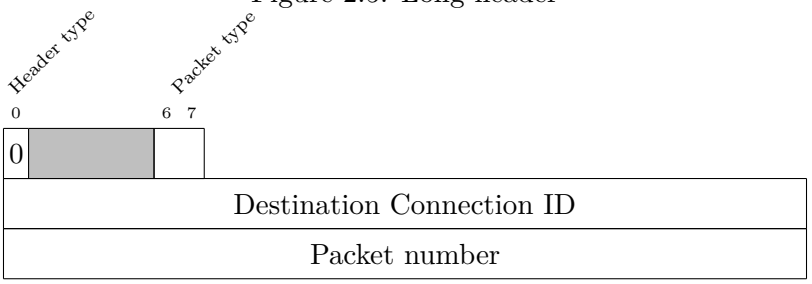


Figure 2.6: Short header

packet number is a monotonically increasing 62-bit number indicating the order in which packets were sent. The client and server also maintain separate packet numbers. A packet number may only be used once by each peer. A connection that runs out of packet numbers must be closed.

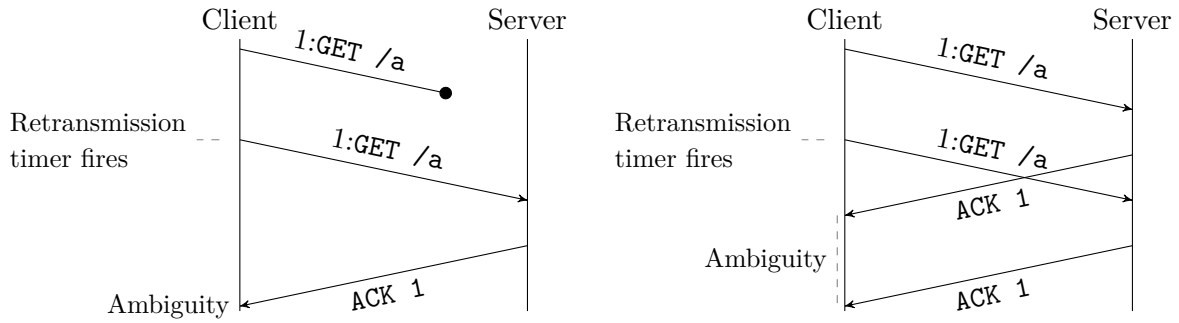
QUIC defines different types of packets for different purposes or cryptographic protections. Long headers are used to convey packets during the connection establishment, namely `Initial`, `Retry`, `Handshake` and `0-RTT Protected` packets. `Initial` packets are used by the client to start the connection. `Handshake` packets are used by both client and server for the rest of the connection establishment. `Retry` packets can be used by the server to continue the connection establishment in a stateless manner. `0-RTT Protected` packets carries protected application data during 0-RTT connection establishment. Short headers always carry packets protected data after 1-RTT connection establishment. The short packet type indicates how many least significant bytes of the packet number are included in the header.

2.2.2 Packet framing

QUIC packets convey frames of different types. Frames allow peers to exchange application data and to send control information. All frames begin with a single byte indicating the type of the frame. All fields after the type byte are frame-specific.

Lost packets cannot be retransmitted as a whole, because each packet sent must have a different packet number. Instead, the frames contained in the lost packet are sent in a new packet. As a result, retransmissions carry a different packet number than original transmissions. This design choice solves the *retransmission ambiguity* problem.

TCP suffers from this problem because retransmissions and original transmissions share the same sequence number. Figure 2.7 illustrates two ambiguous cases when retransmitting TCP segments. In the Figure 2.7a, the first transmission of segment 1 is lost. At some point, the retransmission timer fires and retransmits it. Note that the segment number is not incremented. Upon the receipt of the acknowledgement, the client cannot distinguish whether it corresponds to the acknowledgement of the original transmission or the retransmission. Discerning the two is important because the client may wish to increase the duration of its retransmission timer because the network has a higher RTT than expected in order to avoid unnecessary retransmissions.



(a) A segment has been lost

(b) RTT is higher than expected

Figure 2.7: TCP *retransmission ambiguity* problem

These restransmissions are called spurious retransmissions.

The second case, illustrated by Figure 2.7b, exhibits such a network. In this case the client is able to determine the RTT, but not before the second acknowledgement is received. Heuristics have been designed to mitigate this problem in TCP, but none is able to distinguish all spurious retransmissions.

2.2.3 Stream multiplexing

Streams in QUIC are lightweight and ordered byte-stream abstractions. One can draw a parallel on an abstract viewpoint from a TCP connection to a single QUIC stream. They are of two types, bidirectional and unidirectional streams. Streams are identified using a 62-bit number, of which the least significant bits indicate the nature of the stream as well as which peer has opened the stream. The parity bit is set to zero when the client initiated the stream, and set to one when it is the server that did. The second least significant bit is set to zero to indicate that the stream is bidirectional and set to one when it is unidirectional. The use of bidirectional and unidirectional streams are defined on a per-application basis.

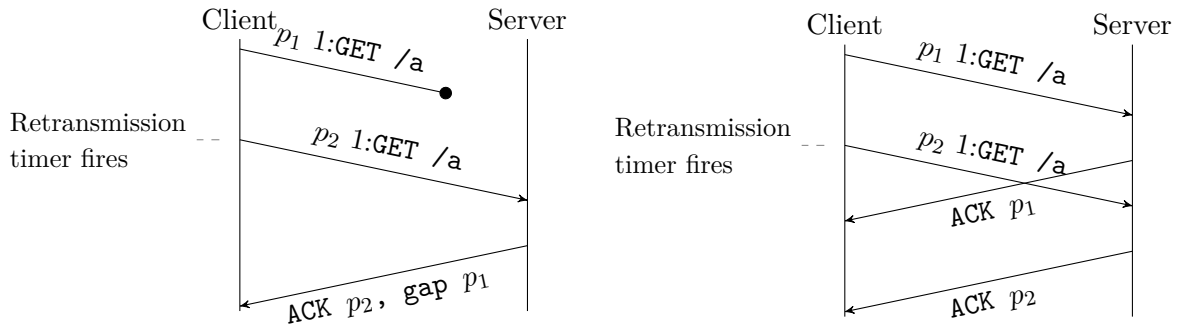
QUIC does not suffer from the head-of-line blocking that exists when multiplexing streams atop a single ordered-byte stream. It ensures that the loss of packet that contains several streams segment only impacts these specific streams. Other streams are not impacted by the loss and can continue to deliver data.

Stream data is carried in `STREAM` frames. A frame indicates the stream ID, the length and offset in the stream of the data it carries. A stream is opened by the simple mean of sending a `STREAM` frame.

2.2.4 Acknowledgements

A receiver uses `ACK` frames to signal to the sender which packets have been successfully received. An `ACK` frame is consisting of `ACK` blocks. Each `ACK` block defines a number of consecutive packets that have been received followed by a number of consecutive packets that were not received, called the gap. A receiver is able to infer which packets have been lost because the packet number is increased by one for each packet sent, thus defining the transmission order. Receiving a packet with a packet number higher than expected implies that a loss or reordering occured along the path. The largest packet number received is included in the `ACK` frame for reference.

`ACK` frames can also encode the delay between the receipt of the largest acknowledged packets and the sending of the acknowledgement. This allows the sender to explicit the existence of a timer for delayed acknowledgements. A peer must not send packets containing only `ACK` frames in response to a similar packet. However they have to acknowledge those packets in future `ACK` frames.



(a) A segment has been lost

(b) RTT is higher than expected

Figure 2.8: QUIC solving the *retransmission ambiguity* problem

In Figure 2.8, the acknowledgement mechanism of QUIC solves the retransmission ambiguity. `STREAM` frames that contain data are encapsulated in packets with a separate sequence number. Because QUIC defines acknowledgements on a packet-basis instead of on a frame-basis, the problem is solved.

2.2.5 Recovery

A separate document is dedicated to describing the loss detection and congestion control mechanisms of QUIC [47].

QUIC offers two kinds of detection mechanism for lost packets. The first one is based on the receipt of `ACK` frames. The receipt of an `ACK` frame that acknowledges packets past a threshold `kReorderingThreshold` of an unacknowledged packet marks it as lost. This is known as the "Fast retransmit" mechanism. The recommended initial value for `kReorderingThreshold` is 3. Because the last packets in a series may not be followed by at least `kReorderingThreshold` packets, another mechanism called "Early retransmit" has to complement it. It acts at the receipt of an acknowledgement for the last packet in a series and marks unacknowledged packets as lost after a short period of time. The delay added before the marking of packets prevents sending unnecessary retransmits when reordering occurs instead of a loss.

The second mechanism is based on timer expiration. It is helpful in case no acknowledgements are received. The first timer-based detection is the "Handshake timeout". Whenever a packet is sent to establish a connection, a timer should be set to retransmit unacknowledged data upon expiration. The second is the "Tail Loss Probe". Whenever the last packet in a series of retransmittable data is sent, a timer is set. Upon expiration, a *Tail Loss Probe* is sent to evoke an acknowledgement from the receiver. The probe takes the form of new data or unacknowledged data. The third and last one is the "Retransmission Timeout". It is the final mechanism for loss detection, and it is thus the one that triggers the last. The timer is scheduled when the *Tail Loss Probe* is sent. Upon expiration, two packets are sent to evoke acknowledgements from the receiver. Sending two packets instead of one makes the connection more resilient to single packet loss. The receipt of an acknowledgement for any of the two packets immediately marks all unacknowledged packets as lost. When the timer expires without receiving acknowledgements, unacknowledged data is retransmitted and it is re-armed with an exponential backoff.

2.2.6 Congestion control

Applying congestion control is a manner for a transport protocol to dynamically pace the transmission of data to a rate that does not overload the network. The mechanism of QUIC for congestion window determination is based on TCP NewReno [50]. The congestion window dictates how much bytes can be sent at once by a peer. Packets sent other than probing packets

account for the total of such bytes, usually called bytes in flight. Every QUIC connections starts in the slow start state. It exits this state upon packet loss. During this phase, the congestion window is increased by the number of acknowledged bytes. A threshold in bytes for the congestion window dictates when the connection transitions back to the slow start state.

A connection that exits the slow start state enters the congestion avoidance state. Akin to TCP NewReno, QUIC uses an additive increase multiplicative decrease (AIMD) approach. The congestion window is increased by the maximum packet size each time packets worth of an entire congestion window are acknowledged. When a loss is detected, the congestion window is halved and the slow start threshold is set to the new congestion window.

2.2.7 Flow control

Flow control allows the receiver to inform and adapt the sending rate of the sender to match its capabilities and resources. QUIC utilises a credit-based system to control the flow of data sent through the connection. It operates at two levels: the entire connection itself, in order to prevent the receiver buffer from exceeding capacity, as well as on each stream, to ensure that a single stream does not take up all the receiver connection buffer capacity. Note that the data exchanged during connection establishment is exempt of flow control.

At connection establishment, each peer declares how much bytes it is ready to receive on each separate stream as well as in total for the connection as a whole. The credits of the flow control system take the form of an absolute byte offsets for streams and a total of absolute byte offset for the entire connection. As a result, as the connection advances, a peer indicates that additional credits are available through the sending of `MAX_STREAM_DATA` frames for stream-level flow control and `MAX_DATA` frames for connection-level flow control. A peer cannot indicate an offset lower than one indicated in the past.

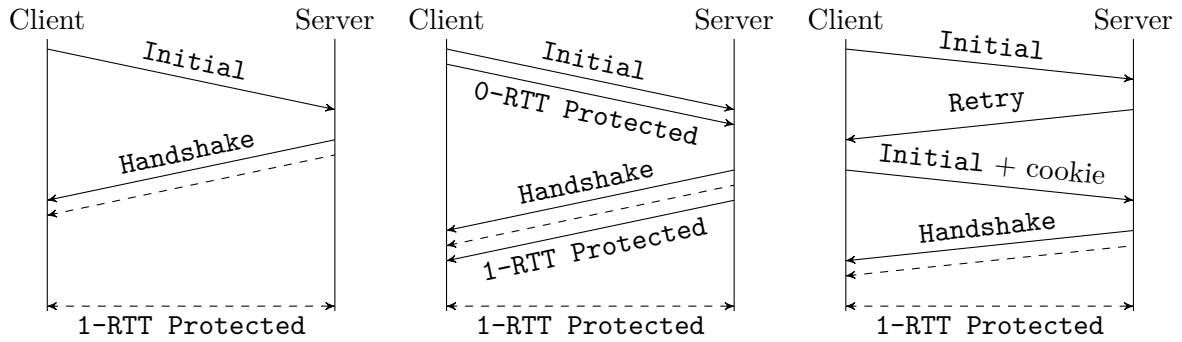
A peer can also control the number of streams that it is ready to make available to the other peer. At connection establishment, they declare how many streams of which types can be opened. After it succeeded, `MAX_STREAM_ID` frames can indicate a higher limit.

2.2.8 Securing QUIC with TLS

QUIC uses TLS as its cryptographic handshake protocol and a separated document details its uses [48]. More specifically, it uses TLS version 1.3 which is also a work in progress at the IETF [51].

The major difference from TLS version 1.2 to version 1.3 is the addition of a 0-RTT mode. It permits the sending of application data without waiting for connection establishment. This comes at the expense of two security properties not being applicable during 0-RTT. Firstly, 0-RTT data is not forward secret, because it solely depends on a secret previously provided by the server. If the secret is compromised, e.g. the server certificate is stolen, the 0-RTT data can be decrypted. Secondly, 0-RTT data cannot be duplicated within a connection but it can be replayed between connections. The document recommends several mitigations that can be implemented by the QUIC implementation or the application using QUIC.

QUIC embeds TLS inside stream 0. It is a bidirectional and client-initiated stream reserved by the protocol. This stream carries information necessary for the TLS connection between the peers. A stateful 1-RTT connection establishment is illustrated in Figure 2.9a. The client sends its *ClientHello* TLS message on stream 0 in an `Initial` packet. The UDP datagram carrying the `Initial` packet must be at least 1200 bytes in length. Padding must be added by the client if necessary. Servers should ignore `Initial` packets that do not respect this requirement. This hinders the possibility of abusing QUIC servers in distributed reflective denial-of-service (DRDoS) by reducing the amplification factor. In DRDoS, attackers send several requests to public servers while spoofing the victim IP address as the source of these requests [52]. These servers reply to



(a) 1-RTT connection (b) 0-RTT resumption (c) Stateless establishment

Figure 2.9: Connection establishment in QUIC

the victim with more data than the requests themselves. They act as bandwidth amplificators for the attackers.

The server responds to the `Initial` packet by sending one or more `Handshake` packets with its TLS *ServerHello* on stream 0 and acknowledges the `Initial` packet received. At the receipt of this reply, the client is able to export forward-secret encryption and decryption keys from TLS. It responds to the server with a *Finished* TLS message to complete the TLS handshake. At this point, QUIC packets are ready to be protected with forward secret keys obtained from TLS. QUIC packets exchanged during the TLS handshake are also protected by a secret derived from the destination connection id and a version-specific salt. This allows to mitigate off-path attacks, i.e. attacks in which the attacker is only able to inject packets in the network but not able to retrieve packets.

0-RTT connection resumption is represented in Figure 2.9b. First the client sends a *ClientHello* on stream 0 in a `Initial` packet containing the resumption ticket provided by the server sent in a previous 1-RTT connection. Immediately after the sending of this packet, the client may send `0-RTT Protected` packets using the 0-RTT key provided by the server in a previous connection. The server replies to the `Initial` with `Handshake` packets and then may respond to the application data sent by the client in `1-RTT Protected` packets, as the handshake just completed. The server is not able to send `0-RTT Protected` packets.

Servers also have the choice to respond to the `Initial` packet without maintaining state during the validation of the client address, i.e. the validation of the fact that the client can receive packets at the address and port indicated as source in its `Initial` UDP packet. Figure 2.9c illustrates this mode of connection. The server responds with a `Retry` packet containing a cookie that must be echoed by the client to initiate a stateless connection establishment. This defers the cost of establishing the connection to the client, as an additional RTT is necessary.

TLS is also used to exchange authenticated declarations of initial transport parameters through the TLS extensions. Each peer has to comply unilaterally to the restrictions implied by the parameters declared by the other peer. These parameters include the initial maximum amount of data that can be sent on a stream and on the entire connection as well as the time of inactivity after which the connection will be considered as closed. Peers can also set the number of streams that can be opened and the maximum packet size they accept.

2.2.9 Connection migration

QUIC connections are not bound to the endpoint addresses, i.e. IP address and UDP port number. A client is thus able to migrate to a new network and continue the QUIC connections it took part in. This mechanism is not available until the cryptographic handshake has completed. A connection migration may be triggered voluntarily or involuntarily by the client. Because it might operate behind a NAT, its connection could be rebound to a different port after a certain

amount of time without network activity. NAT UDP timeout has been found to be much shorter than with TCP [53], despite the strong recommendation of a timeout of at least two minutes and a recommended default value of five minutes [54]. A peer may also be mobile and switch networks which often change its IP address.

The server can provide the client with new connections IDs after connection establishment in `NEW_CONNECTION_ID` frames. These can be used when voluntarily initiating a connection migration. An observer could deduce how the client is physically moving while changing networks based on the network paths in which a given connection ID appears. Changing the client connection ID avoids the leak of information through the correlation of paths and connection IDs. A deterministic, but cryptographically-secure, gap is also added to the sequence of packet number.

After a connection migration is initiated, i.e. packets arrived at the server with a new IP address or port but with a matching connection ID, the server is required to validate the new path before sending significant amount of data. This is done by sending a `PATH_CHALLENGE` frame containing 8 bytes of data that must be echoed by the client in a `PATH_RESPONSE` frame. Receiving the response on the same path on which the challenge was sent gives enough confidence in the usability of the path.

2.2.10 Other control frames

QUIC defines a richer set of control frames compared to TCP. There exist two types of frames, i.e. `CONNECTION_CLOSE` and `APPLICATION_CLOSE`, that allows an endpoint to close the connection and provide an error code as well as a human-readable explanation indicating the reasons that lead to the end of the connection. The two types allows to distinguish application-level from connection-level closures. `BLOCKED` and `STREAM_BLOCKED` frames allows a peer to signal that it is blocked due to connection-level or stream-level flow control. Similarly, a `STREAM_ID_BLOCKED` frame indicates that the peer cannot open new streams due to ID limitations. `STOP_SENDING` frames indicate that their sender will no longer deliver the data received on a particular stream to the application, and thus that their receiver should stop sending data on this stream. `RST_STREAM` frames abruptly close streams. `PING` frames are used to force the peer to send an `ACK` frame and thus verify that it is still alive. `PADDING` frames are consisting of a single null byte and allows a peer to pad packets to a desired size.

2.3 Passive testing and measurements analysis

We mention in this section past works in which passive testing and analysis has been conducted against network protocols. We introduce in the next paragraphs the methodology used and the results obtained when relevant in the context of our work. We detail their work in the subsections that follows. The names of the subsections are the names of the articles they describe.

Passive studies often try to quantify the performance of a network protocol, but some also investigated methods to perform conformance testing passively. A common assumption when adopting a passive approach is that observing the external behaviour of a network protocol is sufficient to infer the actions that were taken and the expected state the implementation is in. As a result this approach can only be used in very limited ways when the network protocol uses cryptography because its messages become hidden to an external observer.

Jaiswal et al. used a finite state machine (FSM) to identify the congestion control algorithm of TCP implementations [55]. They compared the actions taken by the FSM of each congestion control algorithm to the behaviour observed in the traces they studied. They found most of the connections not exhibiting a particular behaviour that their tool could discern to classify the congestion control algorithm they use.

Rewaskar et al. proposed a FSM to analyse TCP out-of-sequence segments [56]. During the

design of this state machine, the researchers found that the diversity of TCP stacks in the traces they studied should be addressed by their tool. They tailored four state machines corresponding to the prominent TCP stacks observed. They compared their tool to *tcpflows* [55] and found the latter to have a lower classification rate because of its inability to adapt to the specificities of several TCP stacks.

Fiteruau et al. learned automata modeling the TCP state machine from traces [57]. These automata showed several differences between the Windows and Linux TCP implementations.

Vern Paxson proposed *tcpanaly*, a tool that infers the behaviour of TCP implementations from traces. Paxson found the design of his tool to be more complicated than expected because of the specificities of different TCP implementations. He reported several critical bugs in TCP implementations.

Treurniet et al. proposed a FSM to detect deviations from the TCP specification in order to identify malevolent activities in network traces [3]. They found several attacks in known data sets.

Qian et al. conducted a large-scale study on TCP traffic. They investigated the evolution of flow sizes, durations rates and other metrics. They found a change in the distribution of several patterns of traffic, which they attribute to the increase of file size on web servers, multimedia streaming and gaming.

2.3.1 Inferring TCP connection characteristics through passive measurements

Jaiswal et al. proposed in 2004 a passive measurement methodology to deduce and track the changes of the sender's congestion window (`cwnd`) and the round trip time during the life of a TCP connection [55]. They argue that these properties allow to diagnose the end-user-perceived network performance.

Tracking the congestion window is done by updating a finite state machine based on the TCP state exposed by the packets they captured. At each step, their method considers the action that would take each congestion control algorithm and then records which algorithms do not coincide with the observations. Their assumption is that the number of bytes in flight reflects the `cwnd` at all time. The algorithm that violated the less their observations is chosen as the supposed congestion control algorithm of the sender. If no violations are found during the connection, the sender is said to have an indistinguishable congestion control algorithm.

They found the congestion control algorithm used to have a small impact on the sender throughput. They conducted comparison between Reno and NewReno, the later adding the "Fast Retransmit" mechanism, to assess to which extent hosts benefit from its enhanced capabilities. They found only 5% of the senders experiencing the receipt of a triple duplicate ACK, which should trigger this new mechanism present in NewReno and not in Tahoe or Reno. They were only able to distinguish the congestion control algorithm of 2.95% of the senders. In other scenarios, senders do not exhibit a different behaviour as no particular loss pattern that would induce such behaviours are experienced.

2.3.2 A passive state-machine approach for accurate analysis of TCP out-of-sequence segments

Rewaskar et al. proposed in 2006 a new tool for TCP traces analysis[56]. They focused on detecting and classifying out-of-sequence segments, i.e. TCP segments that have been retransmitted due to a loss.

They first detail their methodology for passively inferring losses. They tried to separate retransmitted segments perceived as lost from segments that were effectively lost. A retransmitted segment cannot always be considered as lost. Given the basic acknowledgement mechanism of TCP, a loss that occurs in the middle of a series of segments can trigger retransmissions of

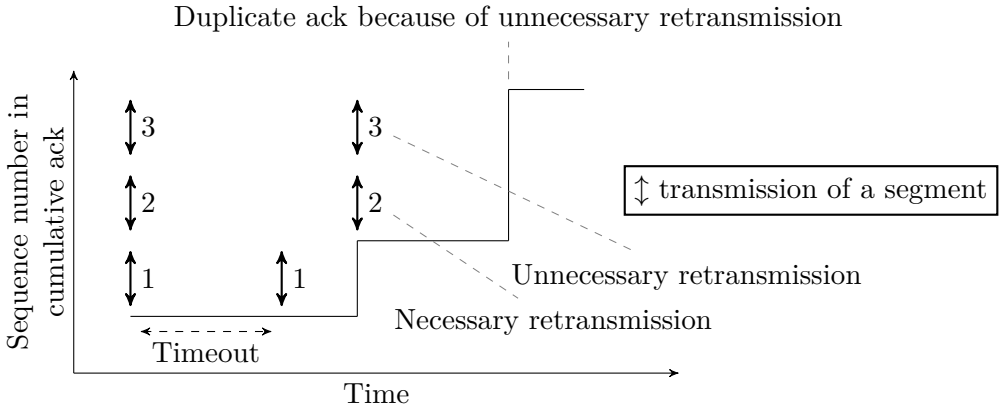


Figure 2.10: TCP implicit retransmission

segment past the loss. A previous work estimated the true loss rate of the connection but it did not identify which retransmissions were not necessary [58].

Figure 2.10 illustrates a TCP connection in which the first transmission of segment no. 1 and 2 is unsuccessful. As a result, the retransmission timer times out and the first segment is retransmitted. An acknowledgement is received and triggers the retransmission of segment no. 2 and 3. Because it is a cumulative acknowledgement, the sender is not able to know that segment no. 3 was received successfully. As a result, its second transmission is unnecessary.

They used a partial state machine modelling a TCP sender to which they input data and acknowledgement from the traces to detect the triggering of TCP recovery mechanisms. They augmented the state machine with the transmission order and timing of the packets previously sent to classify the retransmissions as necessary or not. Using this method, they claim to be able to classify which mechanism triggered the retransmission.

Because they first modelled the sender behaviour using a fixed state machine, they faced the challenge of the diversity of TCP stacks on the Internet. Using a passive fingerprinting tool called *p0f* [59], they identified four prominent TCP stacks. They studied their implementation details to tailor four state machines to replicate their behaviour. They then ran all four state machines against each connection trace and choose the machine that is able to classify all out-of-sequence segments to determine its algorithm.

They later investigated on the impact of taking these particularities into account and found *tcpflows*, the tool developed by Jaiswal et al. [55], unable to explain the nature of about 50% of all out-of-sequence segments while their tool was able to explain more than 90% of them. *tcpflows* is very close to the FreeBSD-specific state machine, because FreeBSD's TCP implementation follows carefully the established standard.

2.3.3 Learning fragments of the TCP network protocol

Fiteruau et al. applied automata learning techniques to the Windows 8 and Linux kernel TCP stacks by observing their external behaviour [57]. The two automata they constructed showed differences in their models, which allowed the researchers to fingerprint each operating system. They abstracted the large possibilities of valid TCP packets into a limited number of actions.

They reduced the state of each TCP segment to the set of flags, the sequence number and the acknowledgement number. Their model also incorporate the direction of the flow of data. Segments sent by the client are distinguished from the response segments sent by the server. They abstracted the numbers themselves by replacing them with a value indicating whether or not the number complies with the standard TCP flow.

They learned automata for a Windows server, a Ubuntu client and server. They combined their observations in a per-OS diagram. The two resulting diagrams exhibits the states of the

TCP specification as expected, but they found a slight variation between the two. A `RST` segment always carries a zero acknowledgement number for Ubuntu while the Windows implementation sets the last acknowledgement sent. They also noted difference under abnormal scenarios that impacted the flags, sequence and acknowledgement numbers sent.

2.3.4 Automated Packet Trace Analysis of TCP Implementations

In 1997, Vern Paxson proposed *tcpanaly*, a tool that analyses TCP traces to infer the behaviour of TCP implementations in unidirectional bulk transfer [4]. This approach required him to cope with ambiguities due to the distance between the measurement point, the sender and the receiver as well as accommodating a number of different behaviours in the various TCP stacks deployed on the Internet. Paxson details the differences between 8 major TCP implementations and explains why he failed to develop a fully general tool.

Firstly, Paxson investigated to measure the errors introduced by the packet filters that produce the traces used by *tcpanaly*, to allow it to detect them. Filtering can occur at two levels: the kernel-level, in which the OS has sufficient tools to reduce the stream of network packets to the sole TCP connection of interest, and the user-level which filters the packet after receiving and reading all of them in user memory, which can induce a high load and potential packet drops. Detecting filter drops is not easy, as one drop could be mistaken with a genuine network drop. TCP makes this task possible because it is a reliable transport protocol, as a result genuine network drops will be recovered but packet filter drops will go unnoticed.

tcpanaly was first designed as a one-pass tool, but this approach was deemed as difficult due the vantage point issue that cause the tool to be unable to tell if the actions observed are a direct response to the most recent packet received or to one more distant in the past. Attempts were made to remedy to this problem by using a k-packet look-ahead window. They were abandoned in favour of determining the sender window. Moreover, recognising generic TCP actions was deemed very hard when the large variations in TCP behaviour became apparent. Eventually, *tcpanaly* works in two passes over the trace, uses a k-packet look-ahead and look-behind to resolve ambiguities and is tailored to the specificities of TCP implementations.

Paxson analysed both receiver and sender behaviour through *tcpanaly* and reported the results. He noted minor variations across senders with regard to the update of the congestion window but also discovered several critical bugs in various operating systems.

In the light of his analysis, Paxson concluded that implementations that are written independently are often prone to bugs and that "implementing TCP correctly is extremely difficult".

2.3.5 A Finite State Machine Algorithm for Detecting TCP Anomalies

Treurniet et al. are members of Defence R&D Canada. In 2003, they proposed a finite state machine that detects deviations from the protocol specification, which allowed them to identify non-conforming implementations as well as malevolent activities in a collection of network traces [3].

Similarly to previous works mentioned in the subsections above, they modelled the TCP connection as a finite state machine with states, events and transitions. A connection that does not end in the *Closed* state is flagged as anomalous. Events that do not correspond to any transition for a given state are also reported as an anomaly failure. A distinction is made between connections that ends in known states different than *Closed*, and connections that violates the state machine. The former are considered as connections that timed out.

The researchers used two known data sets to run experiments with their model. They found a large-scale `SYN` scan, a `SYN`+`FIN` scan and traffic backscatter, i.e. traffic generated towards the victim in response to a spoofed request of the attacker, as part of the connections that were reported as anomalous. They also found potential evidence of a malfunctioning NAT and TCP stacks.

2.3.6 TCP Revisited: A Fresh Look at TCP in the Wild

In 2009, Qian et al. conducted a large-scale measurement study of TCP traffic[60]. Their motivation was to come up with a follow up to the last in-depth studies that were published 6 to 8 years ago. During this hiatus, the Internet and the user demand have grown in orders 1 to 2 of magnitude. TCP also saw the rise of other variants for its congestion control algorithm. They investigated the evolution of flow sizes, durations and rates, the initial congestion window distribution, the actions taken by the sender under packet losses, and the distribution of the TCP flow clock. They captured traces on a Tier-1 ISP, a VPN provider edge router and a DSL provider network.

A 2001 study established that only 4% to 10% of TCP flows are faster than 100kbps [61], the researchers found those flows to account for at least 17% in their measurements. They also found the flows to have increased in volume by about an order of magnitude. They attribute this increase to the file size increase on web servers, as most of the data flows have a source port of 80 which indicates HTTP traffic sent by from web servers. Long-lived flows are also more frequent, which are likely to be multimedia streaming or gaming as verified using IP addresses and port numbers. The researchers mention burstiness as the most striking difference. A 2006 study accounted 1% of the flow as bursty, contributing to 40% of the volume [62]. Qian et al. found such flows comprised between 0.04% and 0.26%, with traffic volume between 0.24% and 1.35%.

2.4 Active testing

We report in this section past works in which active testing has been conducted against network protocols. We introduce in the next paragraphs the methodology used and the results obtained when relevant in the context of our work. We detail their work in the subsections that follows. The names of the subsections are the names of the articles they describe.

A testing approach is said to be active when the tool used for experiments actively exchanges messages with the system tested. It does not require the availability of past measurements but requires the system tested to be available when using the tool.

Pahdye et al. proposed a tool for TCP behaviour inference called TBIT [5]. The tool is able to establish TCP connections autonomously. It is comprised of several test scenarii, each of them verifies a particular feature of TCP. They ran their tool against remote web servers to analyse the behaviour of TCP implementations commonly found on the Internet. They report the results of tests that infer the initial congestion window, the congestion control algorithm used, the support of Explicit Congestion Notification and selective acknowledgements (SACK).

Ekiz et al. [6] extended TBIT with several scenarii that detect misbehaviours of the SACK mechanism. They found a bug compromising the reliability of a TCP connection and several issues with the receipt and sending of SACKs.

Song et al. proposed a verification tool for network protocol implementations [7]. It performs symbolic execution of implementations combined with automata-based rule checking. They propose a method to extract rules from RFC specifications. Symbolic execution is used to generate test packets that covers a large part of the code of an implementation when used as an input.

Yang et al. built a tool upon TBIT for congestion avoidance algorithm identification [63]. It extracts the multiplicative decrease parameter and the window growth function used in connections from network traces. It compares the value extracted to training features to classify the algorithm used. Akin to TBIT, their tool can be used on web servers.

2.4.1 On inferring TCP behavior

In 2001 Pahdye and Floyd proposed TBIT, a TCP behaviour inference tool [5]. This tool has been used and extended in other studies afterward [56, 6, 63]. They developed TBIT with the intent of detecting bugs and non-compliant behaviours with regard to the TCP specification. They primarily focused their effort on the congestion control mechanism as the stability and performance of Internet is reliant on its use. They collected data of remote web servers, because of their large contribution to the Internet traffic and their ease of use, i.e. no particular privilege is necessary to use them.

They outline five motivations for TBIT. The first is to assess the use of the Reno variant of TCP, in order to establish the suitability of Internet simulations based on this variant. The second is to report the initial congestion windows employed by the web servers. Because it may be user-configured, one cannot confidently infer the ICW from the operating system of the host. The third is to be able to publicly identify hosts not conforming to congestion control to reinforce its use. The fourth is to aid the identification of bugs in TCP implementations. The fifth is to test the behaviour of the middleboxes on the path to the server, with an emphasis on situations where ECN is employed.

The researchers outlines two requirements for their tool. Firstly, it should be able to test any web server at any time. Secondly, the traffic generated by TBIT should not appear as hostile to the web server. As a result, it uses services and privileges available to the general public. It depart from tools like NMAP [64], which uses easily-detectable and extraordinary packet sequences, by only generating conformant TCP traffic.

TBIT consists of several tests, each of them is designed for a specific TCP feature. It isolates the tests from the OS TCP stack by establishing a raw IP connection to the host and by blocking this connection from reaching the kernel using the host firewall. TBIT is thus able to generate its own TCP packets, and also ensures the robustness of the tests against packet loss. It makes the server generate traffic by sending an HTTP GET request for the base page `/`. This is arguably the most common page served by all web servers. As most of the behaviour of TCP is based on the number of segments sent, TBIT sets a low maximum segment size (MSS) to ensure enough segments are generated to complete the test. The article details six tests that were implemented in TBIT. We describe the four most relevant of them in the context of our work.

The first test infers the value of the initial congestion window in terms of number of segments. TBIT does not acknowledge any packets after the connection establishment and records each packet send by the server. As the sender is not be able to advance its congestion window, the retransmission of the first packet marks the end of the full flight of the initial congestion window. They found 85% of the servers setting their ICW to 2 segments and 9.5% to a single segment.

The second test generates pre-determined loss patterns to distinguish the congestion control algorithm used by the server. TBIT requests a base web page with an MSS of 100 bytes and a receiver window of $5 * \text{MSS}$. It intentionally drops the 13th packet. Based on the segments received in a stream of 25 packets, TBIT is able to discern NewReno, Reno, Tahoe, a variant of Reno as well as TCP stacks without "Fast Retransmit", which have never been observed previously.

The third test discerns whether or not the host actually used the information contained in selective acknowledgements (SACKs). Verifying the announced support of SACKs can be done with passive testing, but only active testing can ensure that it is working properly. The researchers conducted this test on smaller, SACK-enabled, part of the servers. They found 58% of them not able to verify the usage of SACK information, i.e. unable to use the information they contain to improve their retransmissions.

The last test tracks the deployment of Explicit Congestion Notification [65], a mechanism in which IP routers set a bit in TCP packets to indicate if congestion has been experienced along the path. They found about 1% of the servers able to negotiate ECN but only less than a tenth of percent reported actual congestion notifications correctly. Close to 90% of the servers were

not ECN-capable and were able to gracefully refuse ECN, while 7% did not respond and %2 sent a `RST` in response.

2.4.2 Misbehaviors in TCP SACK Generation

In 2011 Ekiz et al. revisited the study of the SACK support in TCP[6]. While a fraction of the servers were SACK-enabled in the original TBIT study, all operating systems tested in this article were announcing support for SACK. They analysed the CAIDA trace files [66] and found the generation of SACKs to be potentially incorrect with respect to RFC 2018 [67]. The researchers extended TBIT to discern more subtle SACK misbehaviours and reported their results in this article. They found seven types of them, one of which may be impacting the transfer reliability of the connection. A TBIT test has been developed for each of the misbehaviour.

Their most critical finding was the observation of SACK blocks reappearing from a prior connection in later connections. If two consecutive connections were to have overlapping sequence number, the latter would send a SACK for a segment that was never received. This breaks the reliability of TCP. Reappearance of a SACK block was observed as late as 45 minutes after its first appearance.

The researchers concluded that while the handling of SACK is simple in theory, it is now proven to be complex to implement in reality.

2.4.3 Rule-based Verification of Network Protocol: Implementations using Symbolic Execution

Song et al. proposed SYMNV, a practical verification tool for network protocol implementations [7]. Their approach differs slightly from the previous works presented here. It combines symbolic execution of real-word implementations with automata-based rule checking. By replaying a set of input packets that result in a high source code coverage and checking potential violations of rules derived from the specification, they are able to find semantic bugs in network daemons such as three different Zeroconf daemon implementations.

The verification of an implementation with SYMNV is split into four steps. The first is the creation of packet rule. Such rules allow to define a correct sequence of packets based on the protocol specification. One approach to generate these rules would be to translate phrases and requirements of a RFC into rules. The researches adopted a black-box approach by assuming that the behaviour of an implementation is solely constituted of its output packets. By only checking the rules against its output, they do not consider its internal state and thus are able to use them across different implementations of the same protocol. The rules are expressed in a custom description language. Each rule describes a non-deterministic finite automata (NFA). Each state has an associated input packet, each of its transition read that packet.

The second step is to generate high-coverage test packets. SYMNV uses symbolic execution to achieve it. It replaces the actual data by symbolic values which represents a range of possible values. Then it explores threads of execution derived from symbolic variables. It is up to the developer to mark relevant fields of a network packet as symbolic, i.e. the fields that are likely to result in the highest coverage gains. They tie in SYMNV with an implementation by modifying the source code that receives packets. As most C implementations use a standard socket API, they found this modification easy to implement.

The third step is to replay the packets generated previously to the original implementation and to capture its output. All data exchanged on the network is captured for later verification.

The last step is the effective verification of the implementation correctness. All packets captured are used as input to the NFAs generated by the packets rules of the first step. An NFA that ends in a violating state causes SYMNV to generate a violation trace, i.e. to report the sequence of packets that lead to the violation of the packet rule corresponding to the NFA.

They validated SYMNV against three daemons implementing the Zeroconf protocol, as described at that time in documents that lead to RFC 6762 and RFC 6763 [68, 69]. They extracted rules from the documents based on phrases containing strong requirements indicators, e.g. words such as "MUST" and "SHOULD" as defined in IETF BCP 14 [70]. They filtered out any requirement that was not externally observable, e.g. requirements about the internal state of an implementation such as the caching behaviour.

The authors found seven bugs arising from implementation mistakes but also from ambiguities in the specification. They concluded that most of the bugs found are caused by different interpretations of the specification. They argued that extracting rules from the specification can eliminate ambiguities, as this only needs to be done once and can be left out to domain experts who are able to resolve them.

2.4.4 TCP Congestion Avoidance Algorithm Identification

Yang et al. proposed CAAI in 2014, a tool built upon TBIT for congestion avoidance algorithm identification [63]. Their study is useful when designing new algorithms, as they have to compete against other algorithms already already deployed. Traditionally, only the original TCP congestion avoidance algorithm was considered when evaluating their competing behaviours. Since more algorithms are deployed, it is necessary to adjust this comparative framework.

CAAI identifies the TCP congestion algorithm of a remote web server in three steps. Akin to TBIT, CAAI firsts actively gathers traces of TCP connections with the server under different emulated network environments. Then it extracts two features of TCP congestion avoidance algorithms. The first is the multiplicative decrease parameter which determines the threshold at which the slow start ends and the congestion avoidance starts. The second is the window growth function, which determines the increase of the congestion window in the congestion avoidance state. Based on these extract features, CAAI identifies the algorithm by comparing them to the training features collected in their lab test-bed.

They conducted measurements on the 5000 most popular web servers according to the Alexa traffic rank in February 2011. CAAI was unable to collect valid traces for 26% of the web servers, either because they do not serve enough content or they do not accept many HTTP requests on a single connection, letting a small chance for the congestion window to grow. The results of the classification they presented indicate a very diverse congestion avoidance ecosystem, with 7 out of the 13 different algorithms having more than 5% of usage across web servers.

2.5 Alternative approaches

In this section we tackle other works of interest when considering testing network protocols. These studies do not follow a strict active or passive approach but mention interesting considerations and finds for our work. We introduce in the next paragraphs the methodology used and the results obtained when relevant in the context of our work. We detail their work in the subsections that follows. The names of the subsections are the names of the articles they describe.

Bishop et al. proposed a technique for rigorous protocol specification testing. They tried to write a mathematical formalisation of the TCP specification but found it not covering all aspects of the behaviour of TCP implementations. They argue that this formalisation can be written during the specification phase.

Miller et al. approached the problem of testing an implementation for conformance to a specification as a sub-automaton identification problem. Given that the specification forms a non-deterministic FSM (NFSM), each implementation can be checked to be derived from this NFSM. They present mathematical bounds on the complexity of this task as well as the approaches that can be taken when considering active or passive testing.

2.5.1 Rigorous Specification and Conformance Testing: Techniques for Network Protocols, as applied to TCP, UDP, and Sockets

Bishop et al. proposed a technique for rigorous protocol specification that supports specification-based testing [8]. They developed and applied it to TCP, UDP, and the Sockets API after their specifications were conceived but they argue that it can also be used in the design phase of new protocols.

They chose a specification language that is able to capture the non-determinism of a specification. This language uses the HOL system [71] and consists of operational semantics in higher-order logic. HOL is not a fully automatic theorem prover or model checker, because higher-order logic is not decidable, but it allows the development of standalone tools adapted to specific domains.

They chose to pick events interacting with the specification at the segment level, i.e. events that occurs at the network interface and the Sockets API of an host. For TCP this corresponds to the receipt and sending of TCP segments. Focusing only on the Sockets API is not a problem because it is a *de facto* standard. Moreover, it allows the test to be as close to the implementation as possible and to minimise the amount of non-determinism in the system.

They first tried to write the mathematical formalisation of the TCP specification based on the relevant RFCs and POSIX standards. But they found those to omit important aspects of its behaviour, and many implementations do not fully comply with it. They changed their approach and considered TCP to be defined *de facto* by its common implementations, with the intended differences and bugs.

Based on this formalisation of the specification, they wrote a checker that performs symbolic evaluation over real-world network traces. Each trace is thus validated as conform to the specification by checking a set of constraints that needs to be satisfied throughout the connection. They iterated upon the formalisation of the specification until all valid traces were successfully validated. Trace checking is computationally intensive which made the researchers adopt a distributed architecture. 97.04% of the UDP traces were validated and 91.7% of the TCP traces were validated. Some traces hit a space limitation of HOL, some traces were erroneous because of a bug in their generation. But others contained genuine bugs. While finding bugs in implementations was not their original intent, they found several of them in BSD's TCP stack. For example, they found the receive window to be updated on receipt of a bad segment.

The authors conclude the article with a discussion about constructing a mathematically rigorous specification during the design phase of a protocol. They claim that such a specification presents several advantages such as making the complexity of the protocol apparent in the early stages of the design process. It also is a form of communication, and the more rigorous it is the less ambiguities it contains. Finally, machine-checked specification can be evaluated for completeness.

2.5.2 Coping with Nondeterminism in Network Protocol Testing

Miller et al. proposed a method to cope with non-determinism when testing network protocols [72]. They studied four cases complemented with real-world protocol examples, one of which required them to introduce a new method for deriving state machines from specifications. The main assumption of their work is that the specification forms a non-deterministic finite state machine (NFSM) and each implementation defines a deterministic finite state machine (DFSM). They transformed the conformance checking problem into a sub-automaton identification problem.

The researchers distinguish two kinds of NFSM. Some are said to be observable because they allow different responses according to an input (ONFSM). Others includes non-observable transitions and correspond to general NFSMs. Table 2.1 summarise the four cases developed in this article.

The easiest case is when the implementation is a derived machine of an observable NFSM. An

Case	Complexity	Active testing	Passive testing
Derived machine of ONFSM	$O(pn^2)$	✓	✓
Derived machine of NFSM	NP-hard	On-line exploration	Back-tracking
Conformance of ONFSM	k-way expansion	Expand SCC	?
Conformance of NFSM	Exponential	?	?

Table 2.1: Derived machine and conformance

active testing approach would derive the implementation machine by traversing the transitions with the same input from a state. The procedure takes $O(pn^2)$ where p is the number of inputs and n is the number of states. A passive approach would observe the current state of the specification NFSM while inputting the sequence of actions observed from the implementation machine. The time complexity is therefore in $O(L)$ where L is the length of the sequence.

In the case of a general NFSM, i.e. that contains non-observable transition, two approaches are proposed. Both involves NP-hard algorithms as the authors provided a proof by reduction from the Hamiltonian Path problem.

The two last cases involve implementation machines that are not restricted to a derivative of a specification machine. The researchers chose to prove that the implementation is derived from the k-way expansion of the specification, where k is the upper bound on the number of state in the implementation. A k-way expansion has k times the number of states and k^2 transitions for each transition in the reference machine. Each state in the implementation may then be constructed based on a copy of the specification machine. In this case only active testing can be considered. Implementation machines are derived from k-way expansion of strongly connected components in the specification machine.

If the implementation machine cannot be derived at all from the specification machine no matter how large k is, the verification is at least exponential and no computationally tractable algorithms have been established.

Chapter 3

Methodology

This chapter introduce the methodology we adopted for our work. We present the approach we chose for designing our test suite, its general architecture as well as the tools that constitute it. We will conclude this chapter by presenting how we deployed these tools as well as how we introduced our work to the QUIC working group.

3.1 Testing approach

We want to build an autonomous tool that checks whether a QUIC implementation conforms to the specification by the mean of exchanging packets with it. More specifically, we aim to test server-side implementations of QUIC. The tool should be usable on any QUIC endpoint without requiring any special access right. The traffic it generates should not appear as hostile to endpoints and to any middleboxes on their path. We restrict ourselves to the specification of QUIC as being developed at the IETF. We will not consider endpoints that use Google's version QUIC.

The tool follows thus an active testing approach, as it interacts with the implementation under test during its execution. This choice is backed by several reasons. First and foremost, QUIC has been designed as an encrypted protocol. The payload, which consists of frames conveying application data and control frames, is always encrypted. While the header is authenticated but not encrypted, it contains very few information that can be used for testing. One can argue that the use of TLS 1.3 to derive session keys allows to decrypt a QUIC connection if these are made available. We find this approach cumbersome. Moreover, as QUIC was in its design phase when we started our work and still is at the time of writing this thesis, passive analysis of current implementations would not be very insightful because connections to them are scarce. They are very few QUIC endpoints deployed today, and virtually none are used in production. We are not aware of public test endpoints that make available captured traces.

The tool considers the implementation under test as a black-box. It only observe its external behaviour, i.e. the packets that the test suite receives, to evaluate it.

The goal for this test suite is not to check the complete compliance of an implementation, but to track the evolution of the specification and to update its tests as it evolves in order to provide feedback and report bugs to the current implementers. At the time of writing, we note that they are fifteen implementations that have public test endpoints [73]. All of their implementers provide feedback about the quality of the specification and its suitability to address the design of QUIC. Ten of them also participate actively to the specification effort of QUIC, i.e. they comment and propose changes to it. As a result, and due to the limited time frame of our work, our end goal is to help the specification process of QUIC through the making of a test suite and most of our choices should be made toward this goal.

3.2 Architecture

Akin to TBIT [5], our test tool contain multiple self-contained tests. Each one address a specific feature of the QUIC protocol. Our test tool is autonomous in the sense that it can create QUIC packets on its own. It outputs the results of the test in a machine-readable format. The test suite runs every day to track the progress of each implementation in a fine-grained manner.

Each test is self-contained in the sense that it will establish a new QUIC connection to perform the test. The result of the test contains at least an error code that summarises its outcome and a trace of the packets exchanged during its execution. The error code is not purely binary, i.e. passed or failed. It can discern various cases of failure to help the implementer to locate which part of the tested mechanism was deemed as erroneously implemented.

A web site allows to visualise the test results. It eases the communication of bug reports to implementers and hopefully makes them able to consult test results on their own. The results are presented in such a manner that the cause of the problem can easily be established and that a local test that would reproduce the issue can be implemented. The web site also provides information about the purpose of each test.

3.3 Designing test scenarii

We build the different test scenarii and the requirements they should enforce based on the specification of QUIC. This process cannot be automated, because the specifications are not mathematically rigorous but are written in English in an informative style. Each test covers specific sections of the documents and ensures the correct behaviour of an implementation with respect to them. Based on the relevant sections for each test, we analyse phrases and requirements containing strong indicators of importance as defined in BCP 14 [70], i.e. phrases containing the words "MUST" or "MUST NOT", to extract rules that should not be violated throughout the test. This approach is also the one chosen by Song et al. in their methodology for their tool SYMNV. Akin to TBIT, we do not want our tool to appear as generating hostile traffic. We thus did not design tests that would involve a malevolent client.

Each test tries to simulate the behaviour of a compliant client and verifies that the server does not violate any of the rules established. Minor violations, i.e. violations of rules that do not prevent the connection from continuing to function, do not stop the test in an effort of detecting as much violations as possible at once. Major violations do stop the test and set the error code accordingly. All tests share common operations in a shared library, but they keep the actions they take directly apparent in their code. For example, packets are explicitly acknowledged by the tests, but the precise method for packet acknowledgement resides in a shared library.

The tests follow the evolution of the specifications and are updated accordingly.

3.4 Tools developed for our task

We now describe the major tools we created to support the goal of this thesis. Figure 3.1 presents them and summarise the manner they interact together. All the source code of the tools referred to in this section can be found publicly at `https://github.com/mpiraux/master-thesis`. It is subject to the GNU Affero General Public License [74].

3.4.1 A QUIC toolbox

The first step in establishing a QUIC test suite is to be able to create and parse QUIC packets. To do so, we chose not to use an existing QUIC implementation but to develop our own. We adopted the Go programming language for this task. It is a statically typed, memory safe and

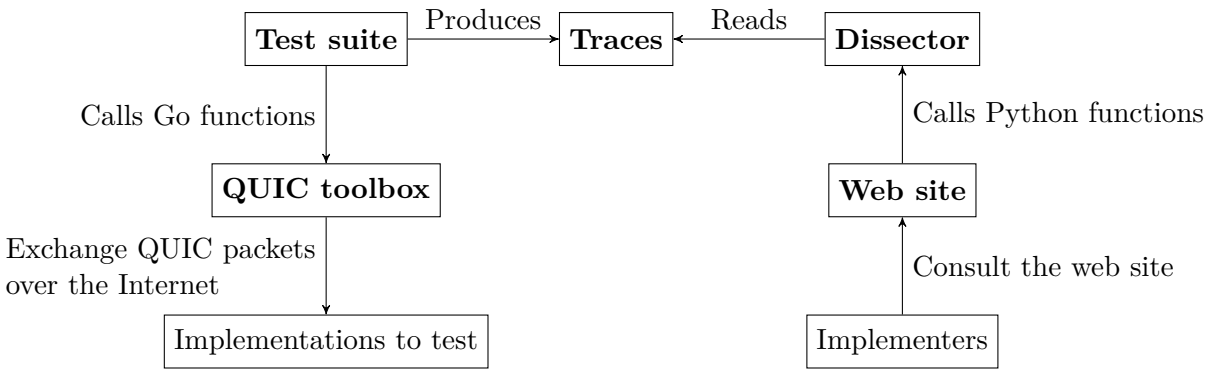


Figure 3.1: Summary of the tools constituting our work

garbage collected language created at Google. It has been designed in the tradition of C but offers a cleaner syntax [75].

We argue that the choice of implementing QUIC, or a least a part of, helped us gaining more understanding of the subject of this thesis. We read the specification documents and started implementing a set of functions to establish QUIC connections and send arbitrary packets at the beginning of September 2017. We did not implement TLS 1.3 because its functioning is beyond the subject of this thesis. Instead, we used *mint* [76], a Go implementation that tracks the evolution of the TLS 1.3 specification and counts its author as one of the contributors [51].

When designing our QUIC implementation, we kept a naming scheme for variables and structures as close to the specification as possible. The hope is that whenever implementers would read our source code, they will able to refer to the common knowledge established by the specification documents to understand the tests we are conducting. We limited ourselves to implementing mechanisms that are shared by the server and client as well as the client-specific ones.

At the time of writing, our implementation supports 1-RTT, both stateful and stateless, and 0-RTT connection establishments. All of these types of connection establishment can be ran on top of IPv4 and IPv6. Our implementation is able to parse all packets that can be sent by a server, and can create any packet that can be sent by a client. It can format `ACK` frames based on the packets received, though their transmission must be scheduled explicitly by the test. It contains a simple mechanism for frame retransmission based on a fixed timer which makes the tests resilient to packet losses.

3.4.2 Test traces

We established a unified machine-readable format called a trace, which contains the results of a test. All tests use this format. We chose the JSON data format [77], defined a trace as a JSON object and specified the fields that each trace contains. We chose a self-describing document approach to facilitate their uses in other applications. Figure 3.2 provides an example of a trace produced by a test scenario that records the versions announced by the server. The values of certain fields have been omitted as indicated by `...` in the interest of space.

We will now describe the format starting from the first field at the top of Figure 3.2 to its bottom. Each trace contains the hash of the current Git commit of our repository at the time at which the test was run. This greatly helps reproducibility when reporting bugs to implementers, as they will be able to read the source code and run it on their own as it was when conducting the test. It was also helpful during the analysis phase presented in the next chapter. Each test scenario has an human-readable id indicated in the `scenario` field as well as a version number. This allows to explicitly encode when a test has been improved or greatly modified and allows version-specific parsing of test results. The server host name and IP against which the test is ran

```
{
  "commit": "bd8d4cf4a268c9b6d414d962f25d4d311d8d4b00",
  "scenario": "version_negotiation",
  "scenario_version": 2,
  "host": "mozquic.ducksong.com:4433",
  "ip": "138.197.141.102",
  "started_at": 1526235130,
  "duration": 218,
  "error_code": 0,
  "results": {
    "supported_versions": [
      3926551082,
      4278190091,
      4045664453
    ]
  },
  "stream": [
    {
      "direction": "to_server",
      "timestamp": 1526235130120,
      "data": ...,
      "is_of_interest": false
    },
    {
      "direction": "to_client",
      "timestamp": 1526235130230,
      "data": ...,
      "is_of_interest": false
    },
    ...
  ],
  "pcap": ...,
  "client_random": ...,
  "exporter_secret": ...,
  "early_exporter_secret": ...
}
```

Figure 3.2: A trace produced by the `version_negotiation` test

are also recorded. The time at which the test begun and its duration are encoded respectively as seconds and milliseconds past the UNIX Epoch time. A scenario-specific error code allows to report a verdict on its execution.

The `results` field is a JSON object that allows the test to report scenario-specific data that can be consumed later by other applications. In this case, the `version_negotiation` scenario includes an array of numbers indicating which versions were announced, if any. Each test also records the packets that were sent and received in the `stream` field. It is an array of objects representing abstract packets. Each packet records to which peer and at which time it was sent as well as the data it contained. The `data` contains the full UDP payload in clear-text, i.e. before being encrypted if necessary. Finally, the test can report whether a packet is considered of interest to the reader. Packets that caused the test to fail or that violated a requirement of the specification will be marked. This is useful to guide the reader through the test result. Note that packets are sorted in the order they are read and written by the test suite. It can differ from the order they are received by the OS running the test suite.

The last part of the trace format was added based on the feedback of implementers [78, 79]. We added a packet capture to each test, the `pcap` field contains the content of the resulting `.pcap` file. Finally, the last three fields contain the TLS secrets required to decrypt it. This allows other applications such as Wireshark to decrypt the packet capture [80]. Based on these values, they are able to reproduce the state necessary to decrypt it.

A run of all the tests in the test suite produces an array of trace objects.

3.4.3 Test scenarii

We now briefly introduce the 17 scenarii we implemented. They are contained in a separate Go package named `scenario`. They are thus decoupled from the QUIC toolbox.

`version_negotiation` is one of the first test we implemented. It triggers the version negotiation process by setting a special version number in the `Initial` packet that is sent by the client. Versions that follow the pattern `0x?a?a?a?a` trigger a version negotiation from the server. If a `Version Negotiation` packet is received, it records the versions announced. The `Unused` field of the `Version Negotiation` packet is checked to be random as required by the specification by comparing the values received in two of such packets.

`transport_parameters` is another early test. It performs a 1-RTT handshake and records the transport parameters received from the server. It also checks whether the server accepts transport parameters in the range reserved for "Private Use" [81].

`handshake` as its name implies performs a 1-RTT handshake. It records whether it succeeded and which version was successfully negotiated with the server.

`handshake_v6` performs the same test but forces the test suite to connect using IPv6.

`ack_only` checks if the server send packets containing only `ACK` frames in response of packets that contain only `ACK` frames. Doing so could induce an infinite loop of acknowledgements. It forces the server to send data it will acknowledge by sending an HTTP GET request.

`http_get_and_wait` is a test that comprises several minor checks at once. It was introduced later than `ack_only` but we chose to not merge the two to ease the analysis of collected data. The test contains various checks for anomalies we noticed when inspecting the traces generated by the test suite. The protocol violations that are checked are for example the receipt of empty `STREAM` frames and the receipt of a `STREAM` frame on an unauthorised stream. It also checks whether the server is able to deliver content when sending an HTTP GET request. This is crucial for most of the other tests that need data to be delivered.

`multi_stream` sends several HTTP requests on up to four simultaneous streams at once. It does not impose the way data has to be delivered, as this is implementation-specific, but checks that all data is eventually delivered.

`http_get_on_uni_stream` sends an HTTP request on a send-only stream on the client, which corresponds to a receive-only stream for the server. Note that for now, the HTTP mapping

adopted by the implementers does not correspond to the full HTTP mapping as defined in a separate draft [82]. Despite the absence of agreements on using unidirectional streams for a basic HTTP server, we decided to test for abnormal behaviours instead of expecting a positive outcome in this case. The test verifies that no data is received on the unidirectional stream or on other unauthorised streams. If the server announces that is does not support unidirectional streams, the test ensures that the server effectively closes the connection when receiving data for these streams.

`stop_sending_frame_on_receive_stream` checks whether the server sends a `CONNECTION_CLOSE` frame with the appropriate error code when sending a `STOP_SENDING` frame on a stream that is receive-only for the server.

`unsupported_tls_version` initiates a 1-RTT handshake and will announce a TLS version that is unsupported by the server. We chose to announce the support of TLS 1.3 draft-00, as the version number is already reserved and very unlikely to see actual support in the future. The test verifies whether the server closes the connection with the appropriate error code.

`padding` is a simple test that sends an `Initial` packet containing only padding up to the length required by the specification. It records any response sent by the server but does not expect a particular outcome.

`zero_rtt` tests the support of 0-RTT connection establishment. To do so it first establishes a 1-RTT connection. If a session ticket has been sent by the server after the connection succeeded, it is used to open a 0-RTT connection. An HTTP request is sent in a `0-RTT Protected` packet and the test verifies that a response is received.

`handshake_retransmission` verifies that the server validates the client address before sending significant amount of data during the connection establishment. It can either send a `Retry` packet or include a `PATH_CHALLENGE` frame in its `Handshake` packets. If the server does not perform the validation, the test does not acknowledge any received packet and records the amount of data sent by the server. An amplification factor is calculated at the end of the test.

`flow_control` checks that the server complies to limits sent by the client. The test sets a very low limit of 80 bytes per stream at connection establishment. It then sends an HTTP GET request. Once the 80 bytes are received, it raises the limit and checks that the rest of the data is sent. An implementation that does not respect the limits imposed fails the test.

`stream_opening_reordering` simulates reordering during the opening of a stream. It sends an HTTP GET request and the graceful closure of the stream in two separate packets. The first has a lower packet number but is sent after the packet containing the closure. The test verifies whether the server performs state transitions after reordering the packets accordingly and answers the request eventually.

`connection_migration` simulates a client NAT rebinding by changing its source port. The test verifies whether the server responds and validates the new path.

`new_connection_id` checks that using the new connections IDs provided by the server and making the packet gap as indicated in the specification is supported by the server.

A timer is set for all tests to ensure they complete in a timely manner. Once a test has completed, it will keep the UDP connection open until the timer expires. This allows to observe any abnormal behaviour that could occur past the test itself.

Descriptions of each test and the meaning of its error codes are centralised in a YAML file. This allows multiple applications to present a human readable description of the test and its outcome.

Specification coverage of the tests

We extracted 169 phrases containing the words “MUST” or “MUST NOT” from the main specification document of QUIC [49]. We classified the requirements they expressed in four categories. Figure 3.3 illustrates this classification and its repartition in the specification document as well as in our test suite.

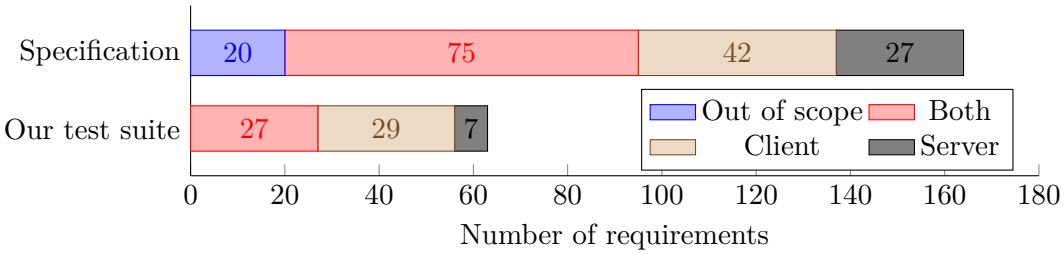


Figure 3.3: Coverage of the `draft-ietf-quic-transport-11` document [49]

The first is the category of requirements that cannot be tested. It can be for example because they involve behaviours that are hard to distinguish externally, such as "A stateless reset MUST NOT be used by an endpoint that has the state necessary to send a frame on the connection". It can also be because they express requirements for which no test case can be established, e.g. "An address validation token MUST be difficult to guess". Finally, there exist phrases that express requirements for the evolution of QUIC itself, such as "The position and the format of the version fields in transport parameters MUST either be identical across different QUIC versions, or be unambiguously different".

The second category contains requirements to which both client and servers are required to comply. Our goal for these requirements is to make sure the client we simulate does not violate any, and to check the server to comply to them.

The third contains requirements for the client, for which we should ensure that our test suite does not violate any. The last category contains requirements for the server. We can establish test cases to verify that the server does not to violate them.

Based on this classification, we identified the requirements that we cover with the scenarii described in section 3.4.3. We estimated that 43% of the requirements are met and checked for by the test suite. One could argue that the coverage of our test suite is quite low. We think there is effectively a lot of work remaining to be done to develop a complete conformance-checking tool. We state here again that the purpose of our effort was not to build such a tool, but rather to help the specification process of QUIC through its making.

Furthermore, we limited ourselves to the identification of requirements based on keywords. But other phrases that do not contains these keywords can also state implicit or explicit requirements. Some of these requirements could already be effectively checked by the tests. We chose not to consider them when evaluating the coverage of the tests because no unambiguous methodology for extracting them can easily be established.

Finally, some of the requirements extracted expect a peer to close the connection with a specific error code whenever the other peer is non-compliant. When observing abnormal behaviour from the server, our test suite does not always issue the correct code nor does it gracefully close the connection in every case. We argue again that our goal is not to implement a fully compliant client but to report bugs in implementations. We thus focused our effort on their identifications.

3.4.4 A web application to visualise test results

Using the tests built on top of our toolbox, we are able to generate traces of test results. We then developed an application that presents them in an human-readable manner. We chose the Python programming language and the Flask web framework to build it. Our personal experience with Python combined to the simplicity of Flask [83] allowed us to focus our effort on developing features, instead of requiring large amount of code to be created before being able to solve our initial problem. We chose AdminLTE for the front-end part [84]. It is a library built on top of Bootstrap that is dedicated to building administration interfaces and dashboards. It combines the ease of use of Bootstrap with various commonly used components of web interfaces.

Figure 3.4: The set of results from the test suite on the 14th of May 2018

<table>
<tr><td colspan="2">Packets stream (Fig. 3.6)</td><td>Test results (Fig. 3.7)</td></tr>
<tr><td>Packet content (Fig. 3.8)</td><td colspan="2">Packet details (Fig. 3.9)</td></tr>
</table>

Figure 3.5: Layout of the web page for single trace visualisation

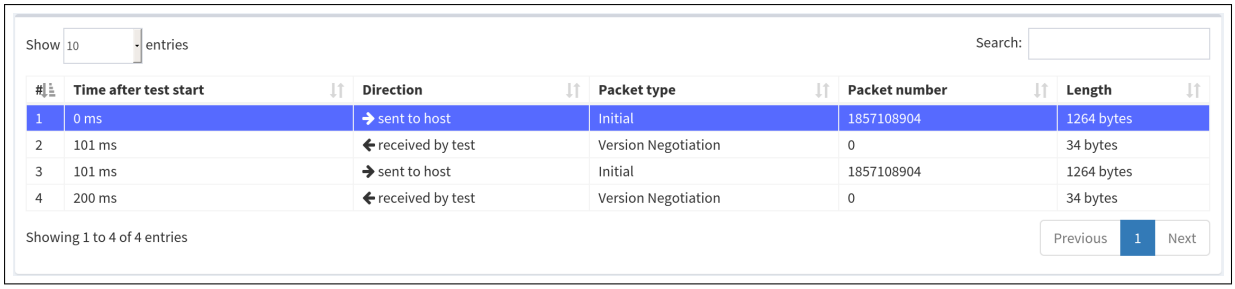


Figure 3.6: The packets stream of `version_negotiation` run against `mozquic.ducksong.com` on the 14th of May 2018

There are two types of pages in our application. Figure 3.4 contains a screen capture of the first one presenting all the results of a test suite run[1]. This page allows implementers to navigate to a particular test scenario and to consult the summary of all results for their implementation. The first column defines against which host the test was run, the second defines which scenario was run. The "Results" column contains an explanation based on the error code set in the trace. The table can be sorted and searched in. The state of the table, i.e. which page was displayed, how much results were displayed at once, which terms were searched for, is maintained across page loads.

The second type is a detailed page of a test trace, which can be navigated to by clicking on a particular row in the table presented in Figure 3.4. We introduce it piece by piece. Figure 3.5 presents the different parts of this page. We will look at a series of screen captures in Figures 3.6, 3.7, 3.8 and 3.9 to detail each part. These were captured from the page presenting the results of running the `version_negotiation` test against the public test endpoint of *mozquic* [85] on the 14th of May 2018. This page is available at `https://quic-tracker.info.ucl.ac.be/traces/20180514/86`.

Packets stream

The packet streams in Figure 3.6 lists the packets that were received and sent during the test. It allows to quickly grasp the overall course of the test. In Figure 3.6, we can see that four packets were exchanged. The first was an `Initial` packet sent by the client, to which the server responded with a `Version Negotiation` packet. Packets that are marked as of interest are indicated by a triangle containing an exclamation mark.

Test results

The panel in Figure 3.7 contains four tabs and serves several purposes. The first tab displays the result of the test, i.e. a summary of the trace object and a human-readable explanation of the error code. In this case the test succeeded. The second tab contains a description of the test

[1] `https://quic-tracker.info.ucl.ac.be/traces/20180514`

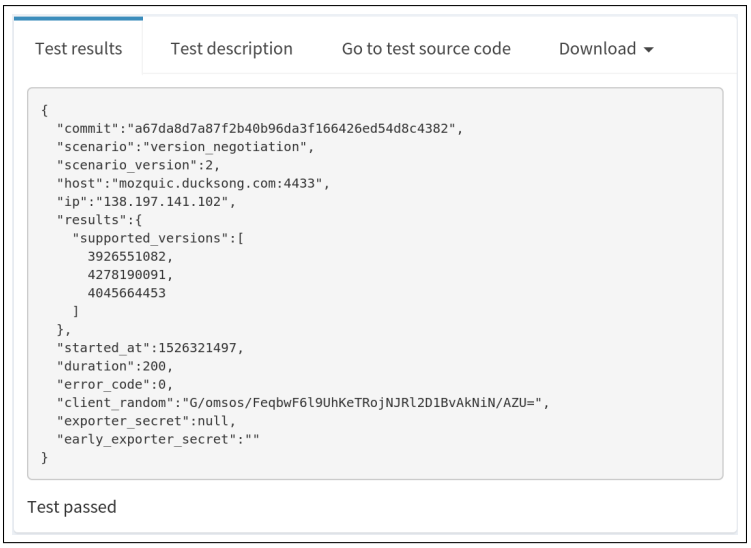


Figure 3.7: The test results of `version_negotiation` run against `mozquic.ducksong.com` on the 14th of May 2018

with details about how it take place and what are the rules that are checked. The next tab is a link to the test source code on Github. It will be presented as it is on the commit indicated in the test result. Finally, the last tab allows to download the `.pcap` file captured during the test and the secrets required for its decryption. The secret output format is the NSS key log format [86], as it is compatible with Wireshark. In our case, no secrets are made available because the `version_negotiation` test does not initiate a secure handshake.

Packet content

The packet content panel in Figure 3.8 shows the raw bytes contained in the packet highlighted in the packet stream. The content is presented in a common hexadecimal manner with a corresponding ASCII text translation. While it may not be the most practical, it is very similar to one of the views that Wireshark would offer for example. The packet that is shown in this panel can be changed by clicking in another packet in the packets stream. Note that the figure does not show the full content of the panel in the interest of readability, as an `Initial` packet mostly consists of padding.

Packet details

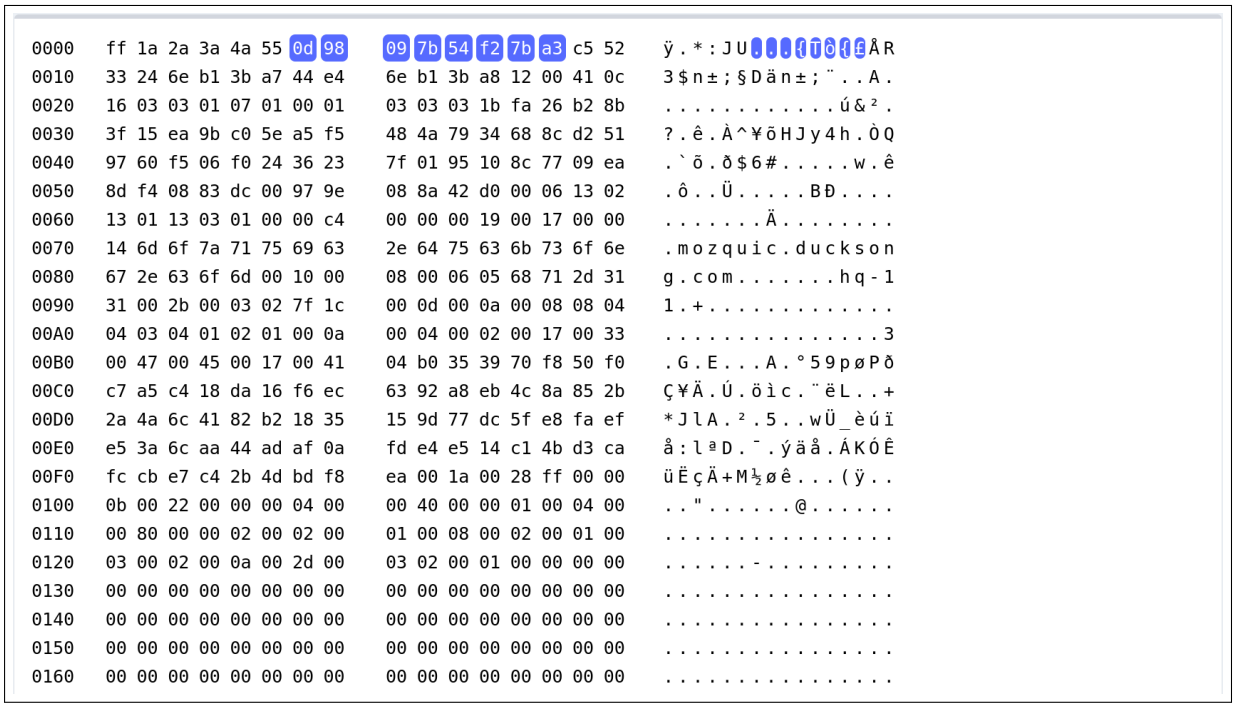
The last panel illustrated in Figure 3.9 presents a dissection of the QUIC packet highlighted in the packets stream.

The dissection is a key-value dictionary representation of the packet content in an human-readable manner. The name of the keys have been chosen directly from the QUIC specification [49]. Clicking on a particular key-value pair highlights it and its counterpart in the packet content. We can see that by clicking on the "Destination Connection ID", the 8 corresponding bytes are highlighted in the packet content. An indication about which specification version was used to dissect the packet is present in the top right corner .

We present the inner-workings of our dissector in section 3.4.5.

3.4.5 A QUIC packet dissector

We chose to write a dissector to make the visualisation application autonomous in the sense that it would be sufficient for an implementer to consult it in order to understand what behaviour was



```
0000   ff 1a 2a 3a 4a 55 0d 98    09 7b 54 f2 7b a3 c5 52   ÿ.*:JU.˜.{Tò{£ÅR
0010   33 24 6e b1 3b a7 44 e4    6e b1 3b a8 12 00 41 0c   3$n±;§Dän±;¨..A.
0020   16 03 03 01 07 01 00 01    03 03 03 1b fa 26 b2 8b   ............ú&².
0030   3f 15 ea 9b c0 5e a5 f5    48 4a 79 34 68 8c d2 51   ?.ê.À^¥õHJy4h.ÒQ
0040   97 60 f5 06 f0 24 36 23    7f 01 95 10 8c 77 09 ea   .`õ.ð$6#.....w.ê
0050   8d f4 08 83 dc 00 97 9e    08 8a 42 d0 00 06 13 02   .ô..Ü.....BÐ....
0060   13 01 13 03 01 00 00 c4    00 00 00 19 00 17 00 00   .......Ä........
0070   14 6d 6f 7a 71 75 69 63    2e 64 75 63 6b 73 6f 6e   .mozquic.duckson
0080   67 2e 63 6f 6d 00 10 00    08 00 06 05 68 71 2d 31   g.com.......hq-1
0090   31 00 2b 00 03 02 7f 1c    00 0d 00 0a 00 08 08 04   1.+.............
00A0   04 03 04 01 02 01 00 0a    00 04 00 02 00 17 00 33   ...............3
00B0   00 47 00 45 00 17 00 41    04 b0 35 39 70 f8 50 f0   .G.E...A.°59pøPð
00C0   c7 a5 c4 18 da 16 f6 ec    63 92 a8 eb 4c 8a 85 2b   Ç¥Ä.Ú.öìc.¨ëL..+
00D0   2a 4a 6c 41 82 b2 18 35    15 9d 77 dc 5f e8 fa ef   *JlA.².5..wÜ_èúï
00E0   e5 3a 6c aa 44 ad af 0a    fd e4 e5 14 c1 4b d3 ca   å:lªD.¯.ýäå.ÁKÓÊ
00F0   fc cb e7 c4 2b 4d bd f8    ea 00 1a 00 28 ff 00 00   üËçÄ+M½øê...(ÿ..
0100   0b 00 22 00 00 00 04 00    00 40 00 00 01 00 04 00   ..".......@......
0110   00 80 00 00 02 00 02 00    01 00 08 00 02 00 01 00   ................
0120   03 00 02 00 0a 00 2d 00    03 02 00 01 00 00 00 00   ......-.........
0130   00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00   ................
0140   00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00   ................
0150   00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00   ................
0160   00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00   ................
```

Figure 3.8: The first packet content of `version_negotiation` run against `mozquic.ducksong.com` on the 14th of May 2018

- Long Header: Dissected using draft-11.yaml
  - Header Form: 1
  - Long Packet Type: 0x7f
  - Version: 0x1a2a3a4a
  - DCIL: 5
  - SCIL: 5
  - Destination Connection ID: 0xd98097b54f27ba3
  - Source Connection ID: 0xc55233246eb13ba7
  - Payload Length: 1252
  - Packet Number: 1857108904
  - Payload: STREAM
    - Frame Type: 0x2
    - OFF bit: 0
    - LEN bit: 1
    - FIN bit: 0
    - Stream ID: 0
    - Length: 268
    - Stream Data: b'\x16\x03\x03\x01\x07\x01\x00\x01\x03\x03\x03\x1b\xfa&\xb2\x8b?\x15\xea\x9b\xc0^\xa5\xf5HJy4h\x8c\xd2Q\x97`\xf5\x06\xf0$6#\x7f\x01\x95\x10\x8cw\t\xea\x8d\xf4\x08\x83\xdc\x00\x97\x9e\x08\x8aB\xd0\x00\x06\x13\x02\x13\x01\x13\x03\x01\x00\x00\xc4\x00\x00\x00\x19\x00\x17\x00\x00\x14mozquic.ducksong.com\x00\x10\x00\x08\x00\x06\x05hq-11\x00+\x00\x03\x02\x7f\x1c\x00\r\x00\n\x00\x08\x08\x04\x04\x03\x04\x01\x02\x01\x00\n\x00\x04\x00\x02\x00\x17\x003\x00G\x00E\x00\x17\x00A\x04\xb059p\xf8P\xf0\xc7\xa5\xc4\x18\xda\x16\xf6\xecc\x92\xa8\xebL\x8a\x85+*JlA\x82\xb2\x185\x15\x9dw\xdc_\xe8\xfa\xef\xe5:l\xaaD\xad\xaf\n\xfd\xe4\xe5\x14\xc1K\xd3\xca\xfc\xcb\xe7\xc4+M\xbd\xf8\xea\x00\x1a\x00(\xff\x00\x00\x0b\x00"\x00\x00\x00\x04\x00\x00@\x00\x00\x01\x00\x04\x00\x00\x80\x00\x00\x02\x00\x02\x00\x01\x00\x08\x00\x02\x00\x01\x00\x03\x00\x02\x00\n\x00-\x00\x03\x02\x00\x01'
  - Payload: PADDING
    - Length: 964

Figure 3.9: The details of the first packet of `version_negotiation` run against `mozquic.ducksong.com` on the 14th of May 2018

```
top:
  - Structure A
  - Structure B
Structure A:
  - First field:
      length: 8
      values:
        neq: 0
      triggers:
        - Second field:
            length: set
  - Second field:
      format: hex
  - next: Structure B
Structure B:
  - First field:
      length: 8
      values: 0
  - Some bytes:
      length: 2
      parse: Byte
  - next: Structure B
Byte:
  - Value:
      length: 8
```

(a) The protocol description

```
04 0A 0B 0C 0D 00 01 02
00 03 04
```

(b) The input

- Structure A:
  - First field: 4
  - Second field: 0xa0b0c0d
- Structure B:
  - First field: 0
  - Some bytes: Byte
    - Value: 1
  - Some bytes: Byte
    - Value: 2
- Structure B:
  - First field: 0
  - Some bytes: Byte
    - Value: 3
  - Some bytes: Byte
    - Value: 4

(c) The output of the dissector rendered by the application

Figure 3.10: An example of inputs and output of the dissector

flagged as anomalous and how a local test case that reproduces the issue can be implemented. Furthermore, when we considered the making of a dissector, no tools were able to dissect encrypted QUIC packets. Wireshark developers later announced an experimental decryption support on the 29th of March [87], a month after we decided to develop the dissector.

The dissector is capable of extracting values and annotating sequence of bytes from the clear-text content of a QUIC packet. It is capable of parsing packets starting from 9th version [88] up to the 11th version of the specification [49]. In order to minimise the amount of work needed to incorporate changes of the wire image of QUIC into the dissector, we decoupled its specification from the dissector itself. The dissector operates based on a protocol specification written in YAML [89]. It is a superset of JSON and is commonly used in configuration files given its improved readability and ease of writing compared to JSON.

Figure 3.10 presents an example of the use of the dissector. We first describe the protocol used in this example, then explain how we modelled its YAML representation and finally describe the steps the dissector takes to produce the abstract representation of the packet.

The example protocol contains two kinds of structure that can appear at the beginning of any packet, `Structure A` and `Structure B`. The first one is always followed by the second if present. The two are discerned based on the first byte. A non-null first byte indicate that `Structure A` is present first. It also defines the length in bytes of its second field. `Structure B` starts with a null byte and is followed by two bytes. To demonstrate some of the capabilities of the dissector, we introduce a new structure `Byte` that represents a single byte. The second field of `Structure B` contains two of these structures. Finally, `Structure B` can be followed by itself until the packet ends.

Figure 3.10a contains the protocol description. The `top` keyword indicates which structures can appear at the beginning of a packet. Other keywords at the top level of the YAML file defines structures. Each structure is constituted of a list of fields. For each field, requirements about its length, possible values and conditions for its presence can be expressed. The `values` attribute of

the field defines possible values or conditions on possible values. The first field of `Structure A` is required to be non-null while the first field of `Structure B` must be null. The `triggers` attribute defines actions that are taken once the parsing of a field completes. Here, we set the length in bytes of `Structure A`'s `Second field` to the value of `First field`. The length of fields is specified in bits in the YAML file, because network protocols sometimes contains fields that are shorter than a byte. When a structure is embedded inside a field, such as the `Byte` structure inside the `Some bytes` field, then the length of the field specifies how many times the structure is expected when parsing this field. The `format` attribute defines how the value of the field will appear to the user. Values are shown as integers when no format is indicated.

For this example, we input a 10-bytes packet to the dissector as shown in Figure 3.10b. When faced with multiple structures that can appear in a packet, the dissector first tries to parse one and backtrack to the next if it encounters an error, i.e. a value that cannot be accepted or an empty buffer despite an incomplete structure. `Structure B` will not be parsed first as an attempt will generate an error when parsing the first field because the first byte of the input is non-null.

Once the dissector has successfully parsed the packet, i.e. when all the packet content has been read and no violations were encountered, it outputs an abstract representation of the packet. It is is a list of tuples in the form $(name, value, start_{offset}, end_{offset})$. The *name* is the human-readable name of the field or structure parsed. $start_{offset}$ and end_{offset} specify which consecutive bytes it spans. *value* can either be a number or a set of bytes that were extracted, but it can also be a list of such tuples which then indicates that it represents a structure and not a field, because it embeds other fields or structures.

The dissector has many more features to express a more complex protocol wire image, but we do not introduce all of them in the interest of space. Its output is used by the web application to produce the HTML content shown on Figure 3.10c.

We wrote the dissector in Python to ease its use inside the web application. We wrote a separate protocol description for each of the specification versions. When parsing a packet of a trace, the application first tries the most recent version. If the dissector is unable to parse the packet, it tries an earlier version. One can argue that the version field can be used to select the appropriate specification version, but it is not present in all packet types. Moreover, the 11th version has changed its location in the header. Choosing the version of the specification on a per-packet basis allows us to dissect packets of multiple versions in the same test and therefore gain a better understanding of the behaviour of an implementation. It also allows to parse custom version numbers defined by implementers which are *de facto* using a particular version of the specification. Custom versions are often used to negotiate the use of a particular feature for interoperability testing.

Because incremental updates to the protocol specification usually introduce a small amount of changes, we generally start from the previous version and add the changes when writing a new version specification in YAML. For example, the YAML file of the 9th version of the specification is 297-lines long. Writing the 10th version consisted of changing 55 of these lines and effectively reusing 80% of the previous version.

3.4.6 Developing Go bindings for picotls

In early March 2018, months after having settled on using *mint* as the TLS back-end for our test suite, we faced the fact that its development had slowed down. Furthermore, the specification process of TLS 1.3 continued at its normal pace. Its 23rd draft introduced breaking changes that other implementers reflected in their TLS backend. It made our tool unable to handshake with the most active implementations. We decided first to incorporate those changes into *mint* by patching it locally. Due to novelty of the *mint* implementation and our lack of detailed understanding of TLS 1.3, it took us a relatively long time to achieve a working patch when compared to its substantial content.

Later on, we chose to develop our own bindings for *picotls* [90], a more active TLS implementation written in C. This choice is backed by several reasons. Firstly, it is used by most of the QUIC implementations we are aware of. This guarantees that it will be maintained in the long term, as most of the implementers rely on it. *mint* in comparison was only used by two Go implementations. Secondly, developing Go bindings is entirely within our reach when compared to updating *mint* to support the latest TLS specification. *picotls* is used in several open-source QUIC implementations, which makes consulting diverse examples of use easy. There also exist detailed documentation on how to use it [91].

We wrote the bindings with the same approach we took for our QUIC toolbox. We implemented the parts necessary for writing QUIC clients but left out any other parts of the API that were not necessary. The source code is located in another Github repository at `https://github.com/mpiraux/pigotls`.

Writing these bindings allowed us to continue testing implementations that supported the 11th draft. We argue that the choice of changing our TLS backend for *picotls* is a winning move. At the time of writing, *mint* has still not been updated and cannot be used to test most recent QUIC implementations.

3.4.7 Building a web crawler for QUIC hosts discovery

Next to our effort of building a test suite, we also wanted to discover QUIC-capable hosts on the Internet. We wrote web crawler that checks for the announced support of QUIC among a list of hosts. RFC 7838 defines a mechanism that allows a web site to announce that it supports alternatives services to deliver its content [92]. It takes the form of the `Alt-Svc` HTTP header. The crawler will connect to each site using regular HTTP, retrieve the base page and parse the header if present. It will record which versions of QUIC were announced.

We extended the web application to present these results. We used the *Cisco Umbrella Popularity list* which contains the top one million domain names queried inside Cisco's Umbrella global network as a base for our search of hosts. Other lists can be used as the crawler sole input requirement is a newline-separated list of domain names.

We implemented it in Python, given its support for asynchronous I/O and the existence of the asynchronous HTTP library `aiohttp` [93]. As a result, it is able to open several concurrent connections to speedup the data collection without the need of creating costly POSIX threads.

3.5 Deploying our work

In early October 2017, we reserved a dedicated machine in the INGI department at UCL. We received access to a computer running an *Intel® Core™ 2 Duo CPU E8400* with 4 GB of RAM and a 256 GB SSD. It also has two dedicated public addresses, one for IPv4 and one for IPv6. Later on, we reserved a domain name, `quic-tracker.info.ucl.ac.be`, and the corresponding certificate to enable HTTPS on our web application.

We use the machine to run the web crawler and the test suite daily since mid-November 2017. The test suite runs at 8PM and the web crawler runs at 11PM local time. The crawler is ran against the first 100 000 domains in the Umbrella list. At the time of writing, fifteen different QUIC implementations are checked by the test suite as reported in Table 3.1. The table lists their names, the company they are affiliated to if any, their software license whenever they are public and the date at which they were added to the test suite. Each day, 255 tests results are produced.

Each test scenario is ran against all implementations until all scenarii have been tested. We chose to test on a scenario-basis rather than on an implementation-basis to give enough time to the endpoints to recover from the previous test before running the next one. Moreover, this

Implementation	Company	Public license	Date of addition
ats	Apache Software Foundation	Apache License 2.0	09/11/2017
minq		MIT	09/11/2017
mozquic		Mozilla Public License 2.0	09/11/2017
ngtcp2		MIT	09/11/2017
quant	NetApp	BSD 2-clause	09/11/2017
quicly	H2O	MIT	09/11/2017
winquic	Microsoft		09/11/2017
mvfst	Facebook		09/11/2017
pandora	TUM		08/02/2018
picoquic	Private Octopus	MIT	08/02/2018
ngxquic	Cloudflare		17/03/2018
f5	F5		17/03/2018
quicker	UHasselt		22/03/2018
quicr			01/05/2018
quinn			20/05/2018

Table 3.1: QUIC implementations tested

approach combined with the inactivity timeout of each test mitigates the risk of overloading the network if we were to run all tests at once against each endpoint.

We deployed the web application on the 8th of March using Apache and `mod_wsgi`, a module which can host Python web applications inside Apache.

3.6 Communicating with the implementers

Shortly after deploying the web application, which was the last part of the architecture proposed in Figure 3.1 left to be deployed, we determined it was time to formally present our work to the public. On the 9th of March, we published an announcement on the mailing list of the IETF QUIC working group [94]. We shortly described our work, linked the web site and invited implementers to provide feedback about the test suite.

Later on, we joined the `quicdev` Slack. Slack is a messaging application to facilitate team collaboration. It is used within the working group to coordinate testing between QUIC implementations. Every implementation has a dedicated channel within the `quicdev` Slack. We made a shorter announcement about our work in the general communication channel and created our own to centralise feedback and future announcements.

We used channels of implementations to discover new test endpoints that were not listed in the dedicated section of the working group wiki [73]. Several implementers also directly contacted us to add their implementation to the list of endpoint tested [95, 96, 97, 98]. We actively reported failed test scenarii to each implementer and answered to any question about the test suite or a particular scenario. Until the 1st of June, we reported a total of 53 failures to 11 different implementations. When reporting the failure of a test, we shortly describe the test itself and include the address at which the result of the test can be consulted as presented in Figure 3.5. When possible, we consult the public logs of the endpoint related to the test and try to locate and report the cause of the bug.

19 out of the 107 persons registered on the `quicdev` Slack are members of the `quic-tracker` channel. Until the 1st of June, 196 messages where exchanged in this channel. Only 70 of them are ours.

Tool	URL
QUIC toolbox	`https://github.com/mpiraux/master-thesis`
Test scenarii	—"—`/tree/master/scenarii`
Web application	—"—`/tree/master/quic_tracker`
Dissector	—"—`/blob/master/quic_tracker/dissector.py`
	—"—`/tree/master/quic_tracker/protocol`
Traces-to-CSV scripts	—"—`/tree/master/quic_tracker/postprocess`
Web crawler	—"—`/blob/master/scripts/faster_alt_svc_scrapper.py`
picotls Go bindings	`https://github.com/mpiraux/pigotls`

Table 3.2: Summary of the tools developed in this thesis and the URLs of their source code

Chapter 4

Results

In this chapter we present and describe results on the evolution of the behaviour of the QUIC implementations we tested. First we review the data we collected using test traces and present the global evolution of several metrics. Then we report two case studies of test scenarii. For each of them we list the implementations that were deemed as anomalous, explain the use of our work made by the implementers and study the cause of the underlying bug observed when possible.

4.1 Measurements and analysis

In this section, we present several metrics we extracted from the data collected by the test scenarii and from the Internet traffic at UCLouvain. For each of the metrics, we explain how the measurements were conducted and what are the conclusions that can be drawn from them.

4.1.1 QUIC traffic inside the UCLouvain network

In December 2017, we started collecting IP network traffic from the six core routers at UCLouvain. Our goal is to determine the amount of traffic exchanged between the university network and QUIC-capable servers. Rüth et al. [99] established to which extent Google's version of QUIC was used by web servers. Akin to their work, we study traffic on UDP port 443. We aggregate packets to sum the IP traffic from and towards hosts on this protocol and port. We compute this sum every five minutes.

We note that UCLouvain is also using UDP port 443 to provide a VPN service. The traffic we observe can thus be partly related to this service. Students on the campus access the UCLouvain network through eduroam [100], a joint effort by universities to provide a roaming wireless network to their members. UCLouvain imposes restrictions on the ports and transport protocols that can be used inside eduroam [101]. UDP port 443 is not authorised. The traffic we collect on this port is thus only exchanged by researchers, employees and other equipments at UCLouvain.

Based on this data, we present Figure 4.1 and Figure 4.2. The first figure represents the evolution of the percentage of UDP port 443 traffic over the total IP traffic in terms of bytes exchanged in the UCLouvain network. We computed an average per weekday restricted to the work hours, i.e. between 9 AM and 6 PM from Monday to Friday. We chose to not consider traffic outside of this time window because it may be traffic that is not relevant in the study of a large Internet network. For example, it may contain internal backups that are performed over the network and other local transfers necessary for the operations of UCLouvain.

We can observe that the UDP port 443 traffic constitutes between 1 and 2% of the total traffic. A group of outliers between the 24th of December and the 3rd of January can be attributed to the Christmas-New year break. The dashed line is a linear regression starting from the 3rd of January. We chose to not include past data into the linear regression given the outliers it

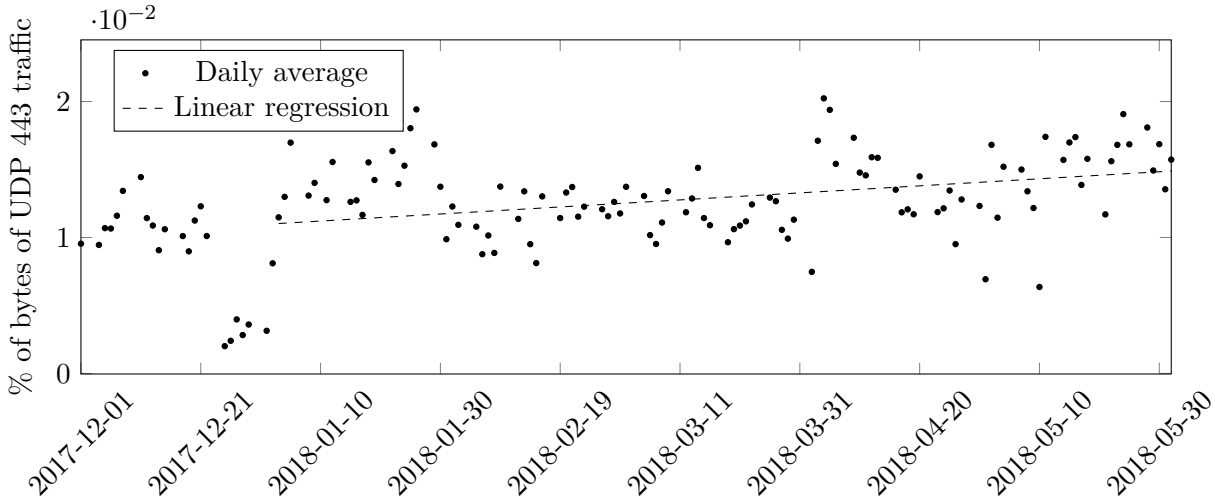


Figure 4.1: Percentage of UDP traffic on port 443 during work hours in the UCLouvain network

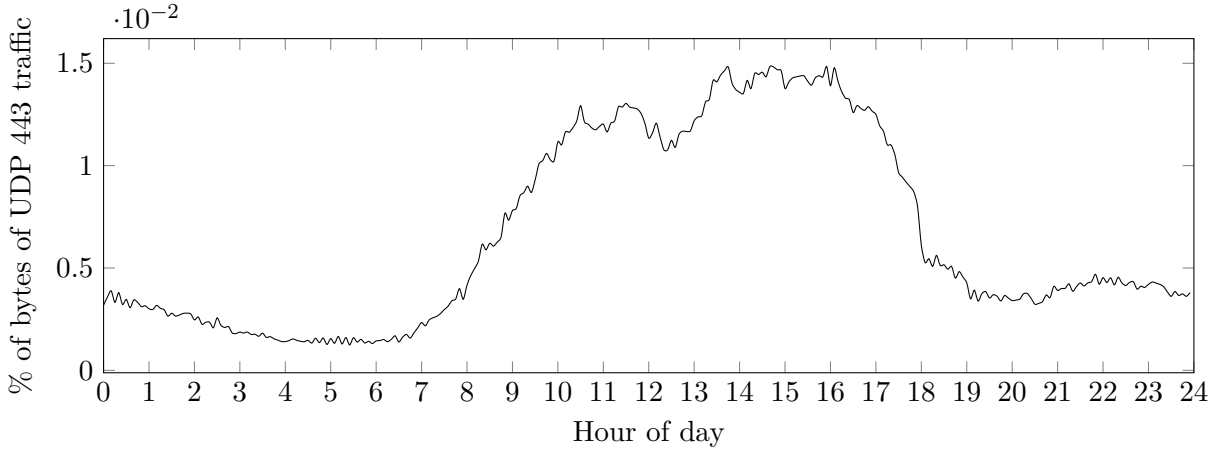


Figure 4.2: Average percentage of UDP traffic on port 443 during week days in the UCLouvain network

contains. We can observe a slow trend upwards of UDP port 443 traffic, though it is not very significant.

Rüth et al. observed the traffic of a major European ISP in August 2017. The researchers found UDP port 443 traffic to account for 7.8% of the total traffic. Overall, the average percentage of traffic we observe is lower.

Figure 4.2 shows the average of percentage of UDP port 443 traffic during weekdays. We computed this average over all the weekdays between the 1st of December 2017 and the 1st of June 2018.

We can observe that UDP traffic on port 443 is influenced by human activities inside UCLouvain. The outliers of Figure 4.1 constitute a first indication of this observation, as fewer employees and researchers are on the campus during this period. Figure 4.2 better illustrates this observation. We can see that during work hours, the average percentage of UDP port 443 traffic triples in average when compared to traffic during the night. This observation is expected as only two major companies had deployed QUIC on a large scale during the period of our data collection. Google deployed QUIC on services such as Youtube and Google Search while Akamai started enabling QUIC on their web servers in May 2018 [102]. These services are unlikely to be used by automated scripts and non-interactive programs.

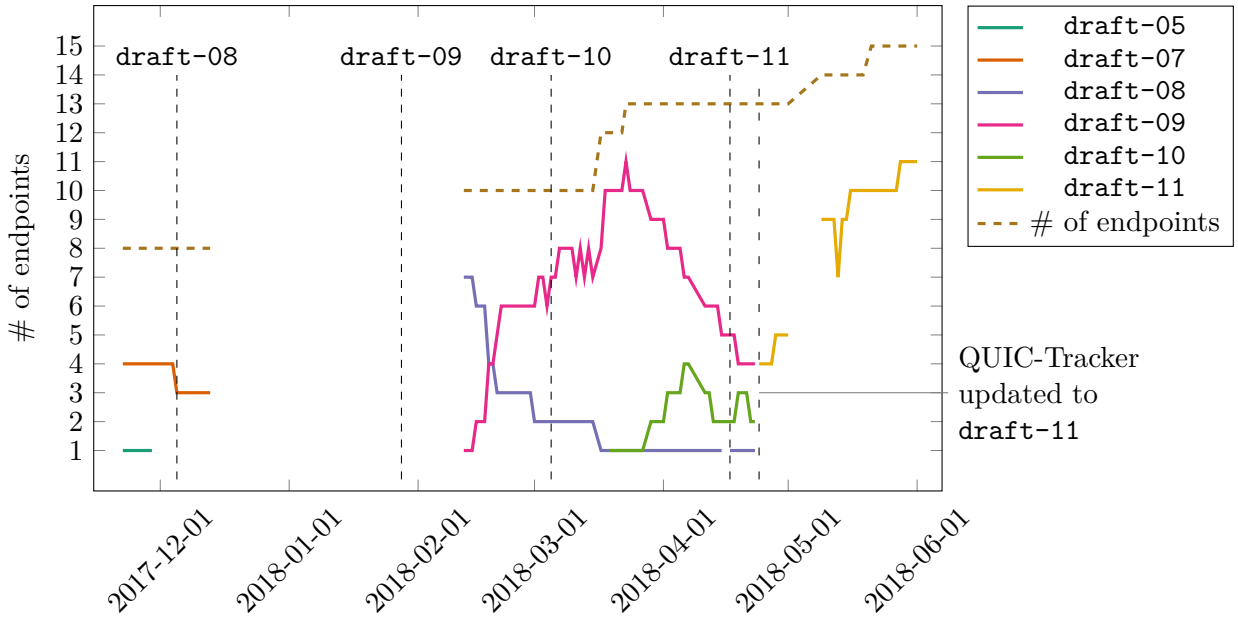


Figure 4.3: Number of endpoints announcing different draft versions

4.1.2 Deployment of QUIC during the specification process

We report here the results collected by the `version_negotiation` test scenario presented section 3.4.3. The test records which versions are announced by the server through the version negotiation mechanism. Then we compare them to the results of the `handshake` scenario. This allows us to compare the announced support of a version to its effective support.

Figure 4.3 illustrates the evolution of the different draft versions that were announced by the implementations tested. For several versions of the QUIC specification, we report the number of implementations that announced their support. The figure also indicates how many endpoints were tested. The publication of a new version of the specification is indicated by a densely dashed black vertical bar.

We were unable to collect data between the 13th of December and the 12th of February. While the test suite to runs every day, new changes incorporated to implementations after the 13th of December made our tool unable to collect data. A bug was introduced on the 1st of May, preventing data collection until the 8th.

We can observe that whenever a new version of the specification was published, most implementations choose to drop the support of the previous version in favour of the new version without maintaining backward compatibility. This is reflected in the figure by a simultaneous increase and decrease between two versions.

The loosely-dashed vertical bar on the right of the figure marks the date at which the test suite was upgraded to the 11th version of the specification. This version introduced changes to the invariants of QUIC. They made the version negotiation mechanism incompatible with previous versions. As most implementers, we chose to not maintain backward compatibility when implementing the support of the latest specification. As a result, starting from the 24th of April, we are unable to observe hosts advertising versions preceding the 11th version.

We can conclude from Figure 4.3 that new versions of the QUIC specification are published at a regular pace. Implementers often need between fifteen days and a month to integrate the changes into their implementations. As a result, tracking the behaviour of QUIC implementations requires to be regularly active in the maintenance of the tool during its specification process.

Figure 4.4 reports the number of implementations that successfully handshaked with our test suite. It also indicates which version was used. As the figure illustrates, we started implementing

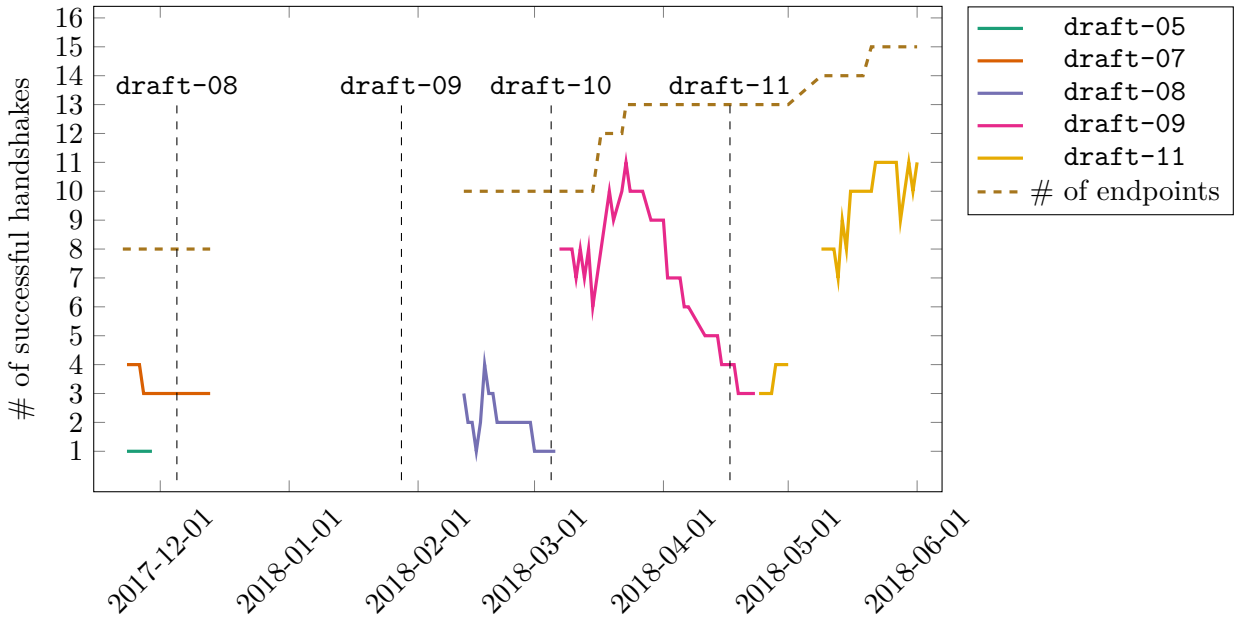


Figure 4.4: Number of endpoints succeeding 1-RTT handshakes

the 5th version of the specification and maintained backward compatibility when implementing the 7th. During the second semester, we first implemented support for the 8th version. We rapidly upgraded the test suite to the 9th version. We chose to not deploy the 10th version, because we learned through the `quicdev` Slack that most implementers were willing to support the next version as soon as possible. Deploying it over the 9th version would have significantly reduced the data collected during this period of transition. Figure 4.3 confirms this fact, as only a maximum of four implementations announced its support on the same day. Finally, we deployed the support for the 11th version a week after its publication in an effort to provide feedback and bug reports to implementers as early as possible.

We can identify two periods of interest for simultaneously comparing implementations. They correspond to two peaks around the 23rd of March and 22nd of May. These peaks do not directly corresponds to particular events of the working group, such as a day reserved for interoperability testing.

4.1.3 QUIC transport parameters over time

We now study the evolution of QUIC transport parameters over time. Transport parameters are exchanged by both peers during connection establishment. They are used to inform their receiver of the restrictions set by their sender. We collected the transport parameters sent by each implementation using the `transport_parameters` test scenario.

Figure 4.5 presents the evolution of the `initial_max_stream_data` parameter of several implementations. This parameter indicates how many bytes can be received on a single stream after connection establishment. We can see that there are very few changes over time. *quant* is the only implementation which increased its limit during our analysis. We can observe that while some implementation chose a value very close to others, there exists a variety of different values for this parameter, ranging from 2048 bytes as announced by *ats* to 1 250 000 bytes as announced by *quicr*.

Figure 4.6 reports the evolution of the `initial_max_stream_id_bidi` parameter. It indicates how many bidirectional streams can be opened by the client. Before the 11th version of the QUIC specification, the parameter indicated the maximum stream ID that can be opened by a client. The 11th version changed the parameter definition to represent the number of bidirectional

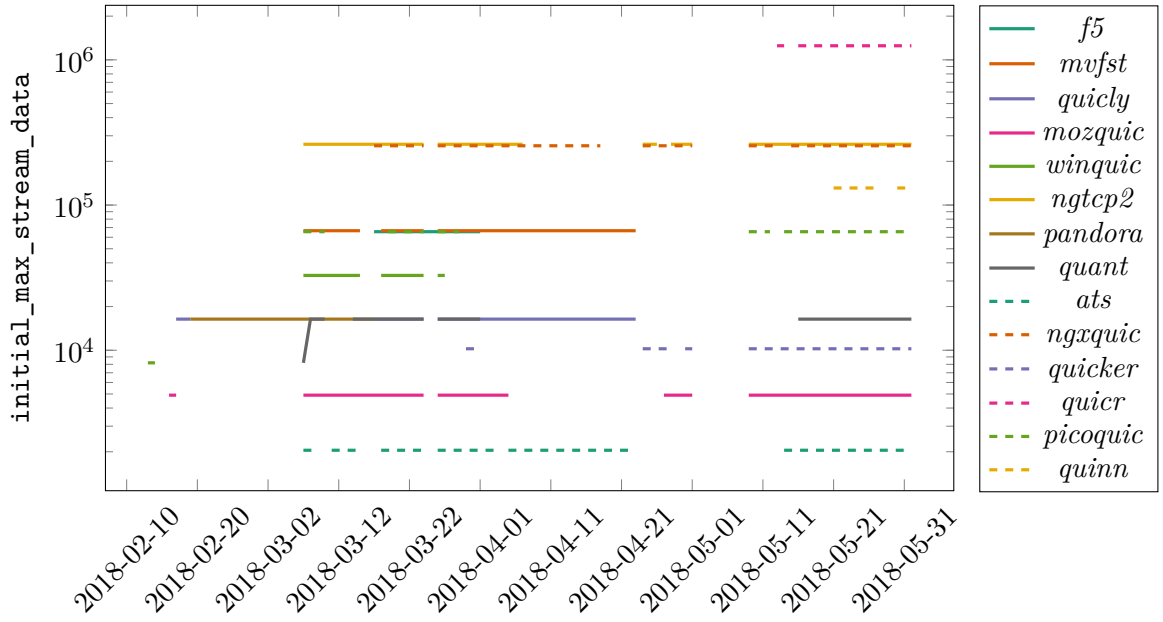


Figure 4.5: Transport parameter `initial_max_stream_data` announced by endpoints

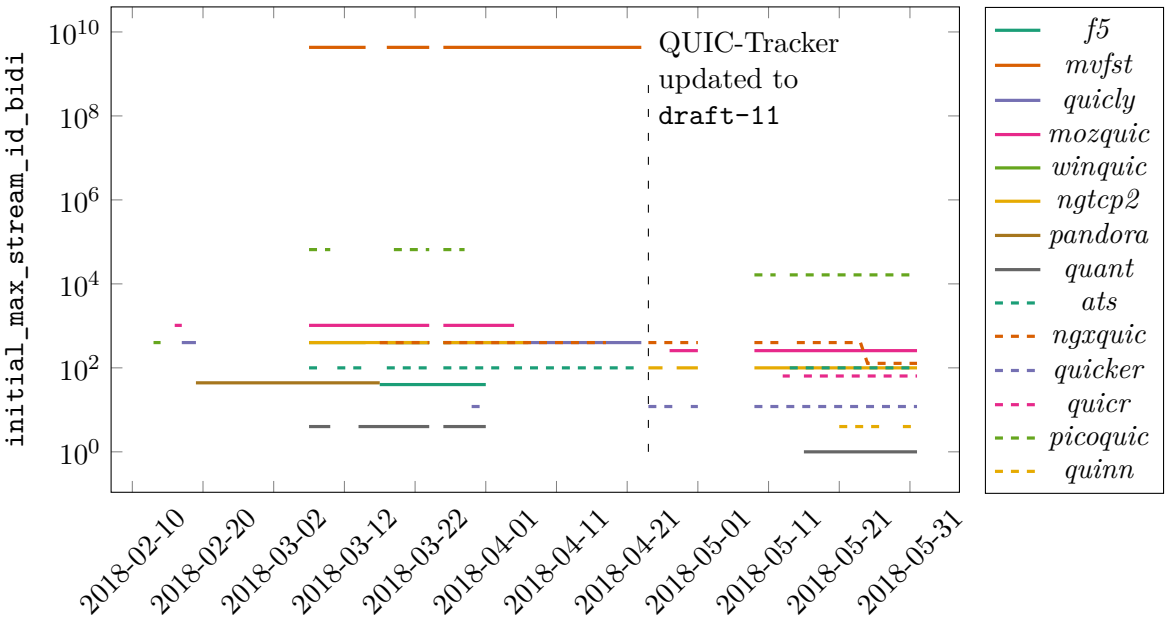


Figure 4.6: Transport parameter `initial_max_stream_id_bidi` announced by endpoints

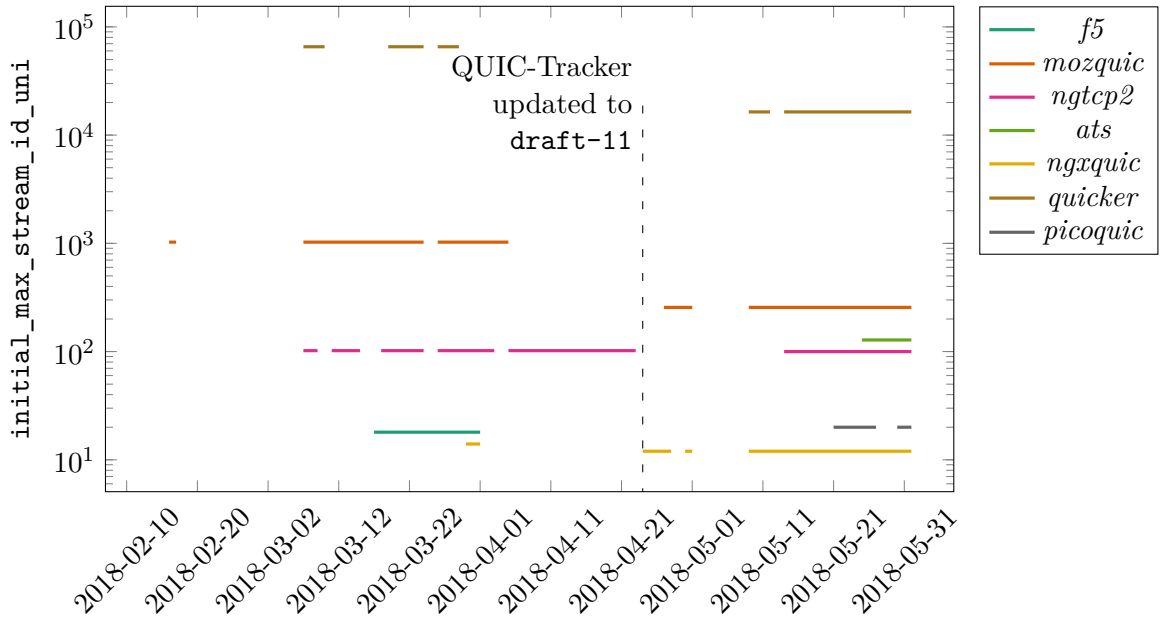


Figure 4.7: Transport parameter `initial_max_stream_id_uni` announced by endpoints

streams that can be opened by the client. We can observe that while the semantic of the parameter changed, some implementations did not adapt their value. One did adapt its value a month after its support of the 11th version. *ngxquic* lowered the value it announced from 400 to 128. We can again observe that the values chosen by the implementations cover a large value space.

Figure 4.7 presents the evolution of the `initial_max_stream_id_uni` parameter. It indicates how many unidirectional streams can be opened by the client. Akin to `initial_max_stream_id_bidi`, its definition was changed in the 11th version to represent the number of streams rather than the maximum stream ID. We observe that most implementations reflected this change by lowering the announced value. Overall, few implementations announce the support of unidirectional streams. This observation is expected as the implementers did not agree on testing them. Features that should be tested during interoperability tests are reported in implementation drafts [103].

Additional parameters are reported in appendix A.2 and A.1. We do not describe them in the interest of space as they show similar behaviours to the ones presented previously.

4.1.4 Patterns of retransmission during connection establishment

Using the `handshake_retransmission` scenario, we recorded the behaviour of implementations when retransmitting packets during connection establishment. We present here two snapshots recorded around the two peaks we identified in Figure 4.3. Then we describe the evolution of the amplification factor computed based on the amount of data received after initiating a connection.

Figure 4.8 reports the time at which each retransmission of the content of the first packet sent by the server is received. We observe the retransmissions sent by seven implementations during the ten seconds of duration of the test. We force the server to send retransmissions by not acknowledging any of its packet.

The figure also indicates what the default behaviour recommended by the specification is. It recommends the initial timer for handshake retransmission to be set to twice the initial RTT, which should be set to 100ms when no previous RTT has been observed for this connection. Each time the timer fires, i.e. a retransmission is sent, it should be rearmed with a doubled duration.

Implementations that retransmit handshake data earlier than the default behaviour will be indicated by curves below the dashed line while implementations that retransmit later will be

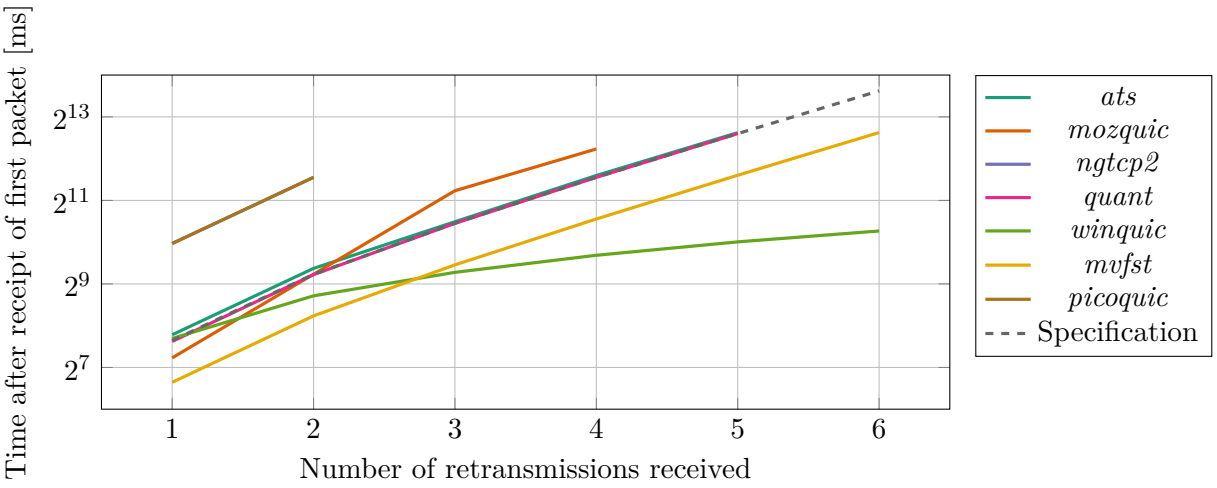


Figure 4.8: Retransmission of `Handshake` packets on the 22nd of March

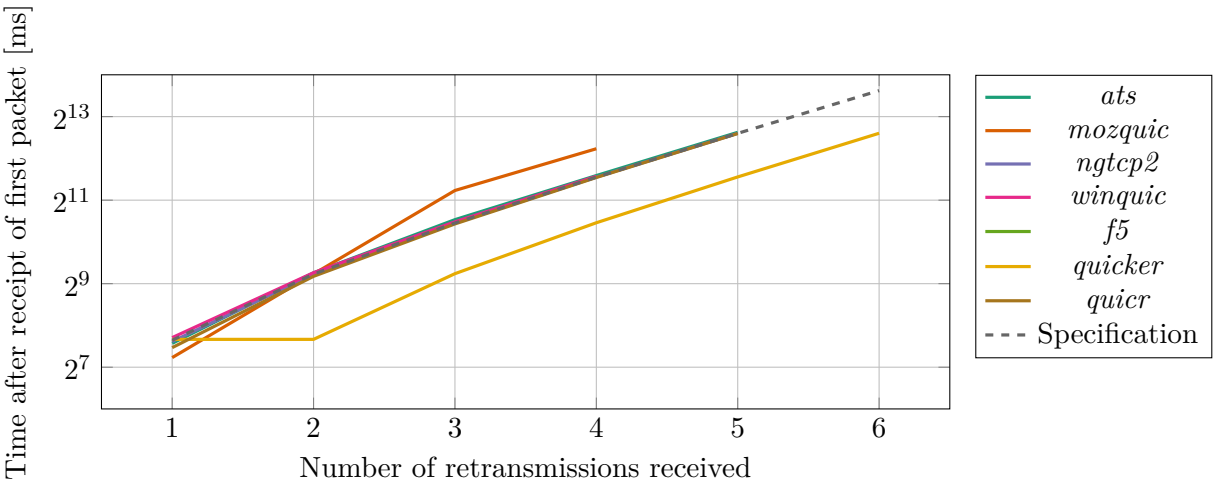


Figure 4.9: Retransmission of `Handshake` packets on the 21st of May

represented above the dashed line. We can observe that while three of them implement the default behaviour, the four remaining exhibit different behaviours.

Figure 4.9 represents the behaviour of seven implementations two months after Figure 4.8. We can observe that *winquic* changed its retransmission behaviour to the one recommended by the specification. New implementations such as *f5* and *quicr* were also added in the meantime, most of which implement the default behaviour correctly.

The 11th version of the QUIC specification introduced new requirements that a server should respect to prevent the protocol from being used as an attack amplificator. In this type of attack, a service is used to generate more traffic in response to a smaller request. The attacker spoofs the address of the victim inside its request to redirect the traffic towards them. The specification now requires servers that are willing to send more than three packets in response to a connection initiation to validate first the address of the peer before sending more than three packets. Servers can either send a `Retry` packet, or include a `PATH_CHALLENGE` frame in each of their packets. Address validation is complete once the client sends a new `Initial` packet in the first case and when receiving a `PATH_CHALLENGE` frame with echoed content in the second case.

We updated the `handshake_retransmission` scenario to compute the amplification factor during the course of the test. Figure 4.10 reports the factors of ten implementations. A threshold of 3 is indicated in the figure.

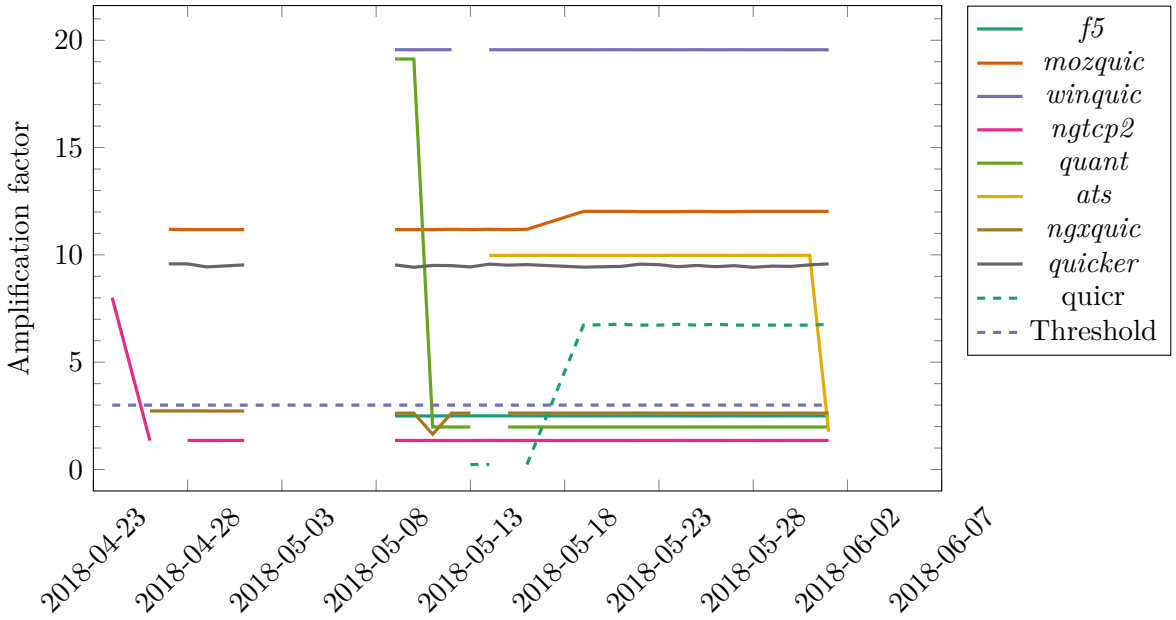


Figure 4.10: Amplification factor of several QUIC implementations over time

We can see that some implementations respected the specification requirements from the start of our measurements. Others did not implement it yet. *quant* shows a drop from a factor of 19.12 to 1.98 when implementing the requirements. *ats* also implemented the requirements on the 1st of June, showing a similar drop. The rise of *quicr* from a factor of 0.23 to 6.72 is explained by its implementation of stateful connection establishment without address validation. Prior to this rise, *quicr* only implemented stateless connection establishment and thus achieved the lowest amplification factor.

The different distances between implementations above the threshold is explained by their difference in certificate lengths. *winquic* uses a valid certificate on its public test endpoint. It is not self-signed and contains others intermediate certificates. The others implementations either use a self-signed certificate or a certificate with fewer or smaller intermediate certificates. Despite having similar behaviours in terms of time before retransmission, as shown in Figure 4.9, implementations that have larger certificates will retransmit more data in total.

4.2 Case studies

We now review some of the test scenarii which reported bugs in several implementations. For each test, we first explain the intent of the mechanism tested. We also shortly introduce again its formalisation in QUIC. We report the evolution of the test based on feedback received from implementers and the introduction of new specification versions. Then we study the underlying causes of bugs observed.

4.2.1 Flow control

Flow control is an important part of a transport protocol that prevents a fast sender from overwhelming a slow receiver. A peer can signal flow control through two mechanisms in QUIC. The first is transport parameters. The parameters `initial_max_data` and `initial_max_stream_data` sets the initial value for the maximum amount of data that can be sent on the entire connection and on each stream. The second is by sending `MAX_DATA` and `MAX_STREAM_DATA` frames once the connection is established, which advertises higher limits for the two previous parameters.

The `flow_control` test initiates a connection and sets `initial_max_data` to 160 bytes and

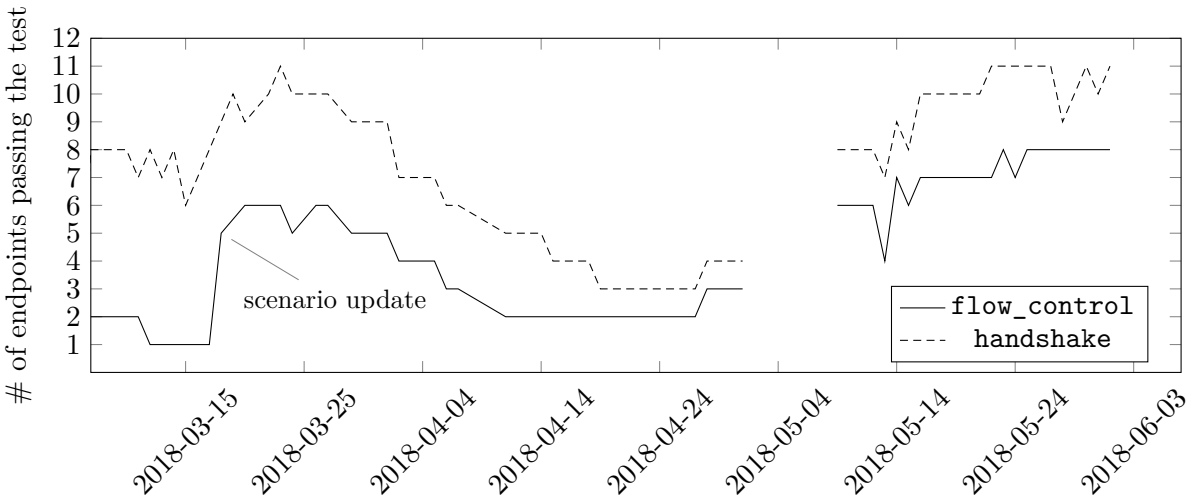


Figure 4.11: Number of endpoints succeeding the `flow_control` test

`initial_max_stream_data` to 80 bytes. After the connection is established, it sends an HTTP request and waits for the server to send the first 80 bytes. The server must not send more than 80 bytes per stream given the value of `initial_max_stream_data`. Once the 80 bytes are received, the test sends a `MAX_DATA` and `MAX_STREAM_DATA` frame to raise the limits to 320 and 160 bytes.

These limits were chosen to be sufficiently low to force the server to adapt its behaviour even if it only serve small files. We contacted one implementer to indicate that their implementation was serving too small files to conduct the test [97]. He increased the size of the files served in response.

Figure 4.11 reports the number of implementations that succeeded the test. The number of implementations that succeed a 1-RTT handshake is also reported to discern increases or decreases specific to the `flow_control` scenario. We can observe a important rise in successes after the 18th of March. We updated the test to stop requiring `BLOCKED` or `STREAM_BLOCKED` frames to be sent. After discussing with implementers [104], we removed this requirement because the specification did not indicate it as an absolute requirement but only as a strong recommendation.

Bugs reported

We now explain some of the most interesting and abnormal behaviours we observed when conducting this test.

picoquic was the first implementation we found to behave erroneously when tested with the `flow_control` scenario. On the 10th of March[1], we found it to have exchanged 20 811 packets during the ten seconds of the test. The implementation seemed to comply to the limits at first, because it sent the first 80 bytes of the page requested. But after sending this packet, it entered into a loop and kept sending `ACK` frames continuously.

We learned the cause of the bug after discussing and reporting the bug to the implementer of *picoquic* [105]. The implementation of flow control was incorrectly integrated with other QUIC mechanisms. Once *picoquic* had received the HTTP request, it stored the data to be sent in response in a buffer. Next it prepared a packet, adding an `ACK` frame first, then it added a `STREAM` frame limited to the 80 bytes we imposed. Once the packet was sent, *picoquic* verified in the buffer if there was data left to be transmitted, which was the case. It prepared the next packet with an `ACK` frame first, and then it tried to add a `STREAM` frame but failed given the flow control limitations. A final verification was missing to ensure that the packet was actually making the connection progress by either acknowledging new packets received or sending new

[1] `https://quic-tracker.info.ucl.ac.be/traces/20180310/69`

data, both of which were not verified in this case. After this ACK-only packet was sent, the loop would start again until the endpoint crashed. We reported the bug on the 17th of March and the results of the 20th indicated that it was fixed[2].

A week later[3], we found nghttp2 to exhibit a similar repetitive erroneous behaviour. 23034 packets were exchanged during the course of the test. It also seemed to comply to the limits at first, but then entered a loop and kept sending empty STREAM frames. We confirmed to the implementer of nghttp2 that a verification was missing for preventing empty STREAM frames from being sent [106]. The specification requires an endpoint to avoid sending such frames [49]. We reported the bug on the 18th of March, we were notified that a fix was implemented on the 19th and the results of the 20th confirmed that the bug was fixed[4].

On the 23rd of March, a few hours after the quicker implementation was announced on the quicdev Slack, we reported to its implementer several bugs including one involving the flow_control test[5]. Two observations can be made regarding the behaviour of quicker during this test. First, it reported being blocked by flow control on the stream used for HTTP and indicates that the blockage occurs at the beginning of the stream. While STREAM_BLOCKED frames are only informational, one could expect the blockage to be reported at an offset of 80 bytes rather than 0. Secondly, two STREAM frames are present in this packet. The first frame contains 80 bytes of data corresponding to a part of the response to the HTTP request as expected. The second frame contains 106 bytes that constituted the remaining part of the response data. We were notified that a bug fix had been implemented on the 26th but we were only able to confirm that it resolved the problem when the test endpoint went back online on the 30th of March[6].

We later detected a regression in quicker on the 18th of May. During the test, it aggressively sent STREAM_BLOCKED frames and retransmissions of the second half of data we requested. In total, 3405 packets were exchanged in ten seconds. We reported the bug on the 21st of May but we did not observe an improvement in later test runs. This is probably due to the fact that quicker is currently developed by a master student from UHasselt, who could be finishing the writing of his thesis during that time.

f5 is an implementation to which we reported several bugs on the 25th of March. We observed that it was unable to complete the connection establishment when running the flow_control test[7]. The implementation only sent an ACK frame during the course of the test. Because it was able to complete the handshake scenario, we indicated in the bug report that it could be due to the implementation enforcing flow control during the connection establishment, i.e. restricting the amount of data sent during connection establishment because of the transport parameters we set. While the version of the specification tested did not exempt connection establishment from flow control, there was an active discussion at that time to add it to the specification[8].

We also note that two days after having reported this bug, the implementer of *f5* opened an issue on the QUIC working group's Github questioning this exemption[9]. 9 participants took part in the discussion and 29 comments were exchanged. At the time of writing, the issue is not solved and implementers planned to talk about this issue in person in their next meeting in Sweden[10]. We are not aware of other implementers who conduct flow control testing. The *QUIC interop matrix*[11], a document that report interoperability testing results between implementations, do not report status about flow control. We argue that the results of this test scenario against *f5* is a direct cause of this discussion.

[2] https://quic-tracker.info.ucl.ac.be/traces/20180320/131
[3] https://quic-tracker.info.ucl.ac.be/traces/20180317/87
[4] https://quic-tracker.info.ucl.ac.be/traces/20180320/123
[5] https://quic-tracker.info.ucl.ac.be/traces/20180323/142
[6] https://quic-tracker.info.ucl.ac.be/traces/20180330/142
[7] https://quic-tracker.info.ucl.ac.be/traces/20180324/140
[8] https://github.com/quicwg/base-drafts/pull/1082
[9] https://github.com/quicwg/base-drafts/issues/1252
[10] https://github.com/quicwg/wg-materials/blob/master/interim-18-06/arrangements.md
[11] https://docs.google.com/spreadsheets/d/1D0tW89vOoaScs3IY9RGC0UesWGAwE6xyLk0l4JtvTVg

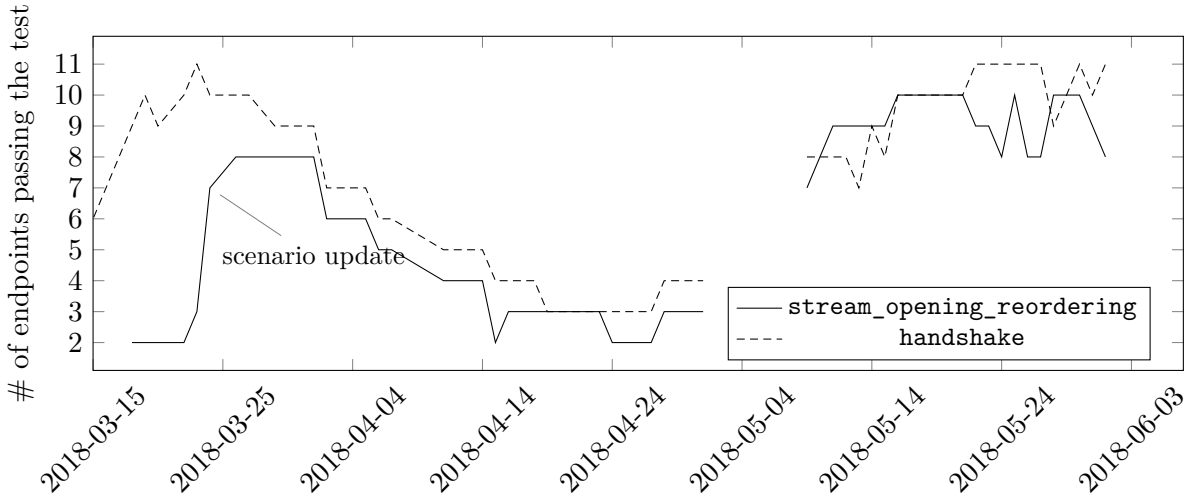


Figure 4.12: Number of endpoints succeeding the `stream_opening_reordering` test

The bug has not been fixed at the time of writing, as we have been observing it again since the 13th of May, which corresponds to our first measurement point after *f5* had been updated to support the 11th version of the specification. We believe that the implementer is not willing to fix it until the discussion aforementioned takes place.

4.2.2 Reordering stream transitions

A QUIC implementation must able to react appropriately when packet reordering occurs. Packet reordering is the alteration of the order at which packets are received from their transmission order. We can discern two cases which can induce packet reordering. The first is introduced by middleboxes, which contain heuristics that are supposed to improve the performance of certain applications or transport protocols they convey. These heuristics can introduce packet reordering, such as moving the tail of a series of packet to its head. These reordered QUIC packets can be easily detected, as the packet number encodes the transmission order. The second is introduced by a packet loss in the transmission of a series of packets which will be retransmitted and received after the rest of the series.

The `stream_opening_reordering` test scenario simulates the first type of reordering. It initiates a connection and sends an HTTP request in two packets. The first packet contains the data of the request and the second contains the closure of the sender side of the stream. The second packet will be sent first. The test verifies that the server is reordering the packets and performing the state transitions accordingly. The test completes once the server has responded to the request.

Figure 4.12 reports the number of implementations that passed the test. The number of implementations that succeed a 1-RTT handshake is also reported to discern increases or decreases specific to the `stream_opening_reordering` scenario. We updated the test on the 24th of March and greatly increased the number of successes. We changed the mechanism used to close the stream in the test. Before the 24th, it sent a `RST_STREAM` frame which corresponds to an abrupt stream closure. After discussing with implementers [107, 106], we learned that the decision to deliver stream data to the application, i.e. the HTTP server, is application-specific. Moreover, the HTTP mapping adopted by the working group for interoperability testing does not correspond to the specification of the mapping of HTTP/2 to QUIC [82, 103]. The implementers did not agree to respond to HTTP requests when the stream is abruptly closed during interoperability testing.

After implementing the support for the 11th version of the specification, we observed that the number of endpoints succeeding the `stream_opening_reordering` test is higher than the number

of successful 1-RTT handshakes. The order in which test scenarii are run is fixed, in this case the `handshake` scenario is run after the `stream_opening_reordering` scenario. We observed endpoints to crash and require manual restarting after running certain scenarii in between the two. Crashes are likely to be due to the lack of testing against 11th-version implementation at that time.

Bugs reported

We will now explain some of the most interesting abnormal behaviours we observed when running this test.

We engaged into a one-to-one conversation with the implementer of `quicker` after finding on the 27th of March that this scenario made their endpoint crash. We found that the reordering of packets triggered a livelock in their code. During the livelock, the implementation did not send any packets nor produced any kind of observable external behaviour. The implementer could not find its root cause based on their logs. We provided assistance to help him install the test suite on his development machine. Doing so allowed him to test the scenario against his local implementation, which was better instrumented for debugging. With our help, he was able to install it and found the cause of the bug when running the test suite. He fixed the bug on the 30th and our test results confirmed it[12].

On the 8th of May, which is our first measurement point after the *quant* implementation was updated to the 11th version of the specification, the test suite detected a regression for this test. We were not actively consulting data during that period and thus did not report the bug. We later found the implementer to have consulted the test result and to become aware of the bug on his own. He was able to fix it without the need of further explanations from us[13]. We argue that this is a proof that the test suite and the visualisation web application are autonomous, in the sense that they are able to detect and present bugs in an appropriate manner for an implementer to locate the erroneous mechanism and fix it without our intervention.

We found the test suite entering a livelock when conducting the test against the *f5* implementation on the 22nd of May. After investigation, we found *f5* to sent erroneous `ACK` frames when receiving packets out of order[14]. These frames reported a gap of $2^{64} - 1$ missing packets. When receiving `ACK` frames, our implementation iterated over the packet numbers reported as missing and scheduled retransmissions when the number corresponds to a packet effectively sent. A very large number of packets reported as missing induced a very large iteration. We concluded that the cause of the bug is that the mechanism for determining the gap between two packets received was not resilient to reordering. A possibility is that the most recent packet number received is decremented by the previous packet number received to determine the gap between the two. Doing so introduces an overflow when the most recent packet has a lower packet number than the previous one. We believe that the bug was introduced closely to the time at which it has been found, as our mechanism for `ACK` processing described previously exists since the 28th of March[15].

When reporting the bug, we learned that it had already been discovered during interoperability testing and a fix was supposedly implemented. But because we were able to reproduce the bug using this test scenario, while it was only triggered by real-world reordering during interoperability testing, we confirmed that the issue was not fixed.

[12] `https://quic-tracker.info.ucl.ac.be/traces/20180330/25`
[13] `https://github.com/NTAP/quant/commit/e7b53093b707e8a043469cd01d5218153887ee53`
[14] `https://quic-tracker.info.ucl.ac.be/traces/20180524/55`
[15] `https://github.com/mpiraux/master-thesis/commit/abe0e7b987860bf7cf80bb7cba398a7137c4614b`

Chapter 5

Discussion

In this chapter we summarise the impact of our work on QUIC implementers and the QUIC working group. We report our experience and the outcomes of our participation to the IETF 101 Hackathon. Then we further support our approach for testing QUIC. Finally we conclude this thesis by discussing further prospects for our work.

5.1 Impact of the test suite on the WG

In this section we provide further evidence that our work has been useful to the QUIC working group and has been acknowledged as such. We review the feedback we have received throughout our interactions with several implementers.

During the week after we published our initial announcement on the mailing list of the QUIC working group [94], we received several private emails from implementers. Three members actively involved in QUIC implementations contacted us, all of them were pleased by our test suite for QUIC. Nick Banks from Microsoft indicated that we should close the sending side of the stream after sending the HTTP request data in order for their implementation to respond to it. Subodh Iyengar from Facebook provided us with the URL of the document their implementation was serving, in order for the test suite to be able to generate traffic when necessary for the tests.

Alexis La Goutte from Wireshark enquired about adding a `.pcap` file to download for each test result. We agreed on developing this feature and invited him to provide details about how to export the secrets for TLS decryption and suggested the NSS key log format [86]. He replied that the mapping for QUIC was not already defined. On the 22nd of March, `.pcap` files were available with each test result. When announcing this new feature on the `quicdev` Slack, one of the developers of the Wireshark dissector for QUIC provided us with the newly defined mapping necessary to export secrets to Wireshark. We patched *mint* to export them and deployed secret exporting on the 28th of March. We also added this feature when developing our Go bindings for *picotls*. Later on, we reported a specific test scenario that produced results that were not properly dissected by Wireshark. We added an example packet capture file and its secrets to the Wireshark bug-tracker [1].

When joining the `quicdev` Slack, we noticed that the URL of our website was pinned in the general channel. We observed that only the *QUIC interop matrix* and some test vectors for TLS were pinned previously. When submitting bug reports to the implementers in their dedicated Slack channel, all of them greeted the test suite. *f5*'s implementer noted the tests as "extremely helpful" and enquired about adding retransmissions to the test suite in order for the tests to be more reliable. This feature was already on our roadmap and we prioritised it. He later discovered that a bug with *f5*'s cryptographic backend prevented it from reading the first packet sent after the completion of the handshake.

[1] `https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=13881#c84`

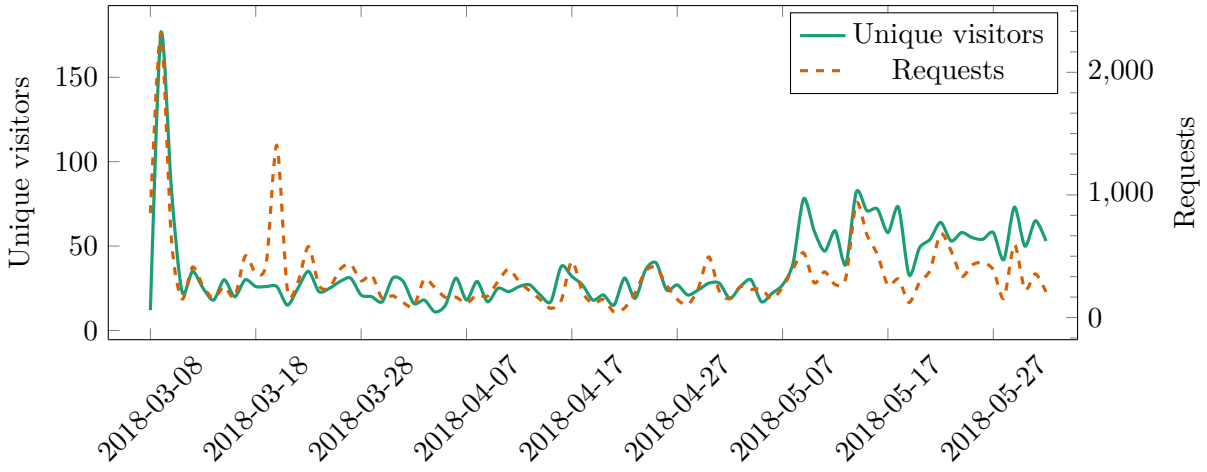


Figure 5.1: Unique visitors and requests on `https://quic-tracker.info.ucl.ac.be`

During the early phase of the implementation of *quinn*, its implementer asked on the general Slack channel whether there was an example stream of messages to illustrate the QUIC protocol. Some RFCs such as RFC 7515 include these into the specification itself [108]. One implementer spontaneously mentioned our test suite and linked its web site as well as instructions on how to download the `.pcap` files.

The implementer of *quant*, who is also one of the chairs of the QUIC working group, stated that the tests we developed were "really helpful" [109]. He also enquired about how to install the test suite. We provided him with an installation guide that is now available with our the source code. We helped to troubleshoot portability issues that arose during the installation our Go bindings on BSD-derivative systems[2]. Using the test suite locally, he was able to run the `unsupported_tls_version` test on a better-instrumented version of *quant*. While there was no initial intent of deploying the test suite on machines out of our control, it is now known to be working on macOS, BSD-derivative and Linux systems. The test suite is also part of *quant*'s automated testing matrix, which runs multiple clients against multiple servers locally[3].

We were contacted by a bachelor student at UHasselt who asked for support when running the test suite locally [110]. He installed both the test suite and the web application on his own but was missing instructions on a particular step of their usage. We helped the student and improved our installation guide. Later on, we asked him the usage he made of our work. He replied that he ran the test suite to observe the behaviour of the server tested under different scenarii and to get a better understanding of QUIC.

We analysed the Apache log files of the machine hosting `quic-tracker.info.ucl.ac.be`. Figure 5.1 reports the number of visitors and the number of requests received in total. We can see the biggest peak happening on the 9th of March, which corresponds to the day we published the announcement on the mailing list. Our supervisor also linked our work on Twitter[4], which explains the magnitude of the peak. We note a second peak in requests during the 17th and 18th of March, which corresponds to the IETF 101 Hackathon, during which interoperability testing was performed. We detail our participation to the hackathon in section 5.1.1. Finally, we note a steady increase in both visitors and requests starting from the 8th of May. We attribute this rise to the new bugs discovered in the implementations of the 11th version of the specification we reported during this period.

When analysing the `Referer` field of the incoming HTTP requests, we found evidences that our work was mentioned in the private wikis inside Cloudflare and Baidu.

[2] `https://github.com/mpiraux/pigotls/issues/1`
[3] `https://github.com/NTAP/quant/blob/11/bin/test.sh`
[4] `https://twitter.com/OBonaventure/status/972063084645896193`

5.1.1 Attending the Hackathon

We had the chance to participate to the IETF Hackathon organised as part of the IETF 101 meeting. This event took place in London during the weekend of the 17th and 18th of March. The members of the QUIC working group were invited to participate on site or remotely to interoperability tests during the weekend. During the week that follows, open meetings were scheduled to discuss specific topics regarding the design of QUIC. We participated in order to meet the QUIC implementers and obtain feedback on our work. It was also a good opportunity for them to engage discussions about the test suite.

We received several feedback from implementers while participating to the hackathon. For instance, the first bug we report in section 4.2.1 lead to a discussion with the implementer of *picoquic*. Other implementers enquired about the approach we took to test implementations [111, 112]. We presented them the general architecture of our work.

During the hackathon, we also improved the test suite and implemented new test scenarii together with François Michel, a master student at UCLouvain who joined us for the weekend. He had experience with Google's version of QUIC but never read the IETF specification of the protocol. Within a day, he was able to implement the `stop_sending_frame_on_receive_stream` test scenario. Most of his time was spent on reading the specification, designing the test scenario and verifying that it behaved as expected, not on understanding the architecture and the tools we had designed. The second day, he implemented the `http_get_and_wait` and `http_get_on_uni_stream` scenarii, which validates several requirements of the specification. We argue that this is a proof that new scenarii can be easily added. For complete transparency, his contributions kept his authorship[5].

Because our mode of operation differs from the other QUIC implementers, i.e. we did not participate in the interoperability testing in a strict sense, we argue that attending the IETF Hackathon had a positive outcome. It was a good opportunity for implementers to get to know about our work and for us to show that our tool was of interest to them. We argue that without our attendance, we would have received far less feedback during the course of our work.

5.2 Testing without validation

When implementing both the toolbox and the test scenarii, we did not develop unit tests to ensure the correctness of their components. We motivate this choice with several reasons. First, the toolbox is mostly consisting of small functions that performs byte formatting, i.e. taking various arguments and outputting a stream of bytes representing the wire image of the structure. We found manual inspection to give enough confidence about the correctness of these functions. Secondly, given the limited time frame for our thesis, we estimated that our time was better invested in developing new test scenarii and extending the tools such as the web application for visualisation than to ensure the total correctness of our implementation.

Our approach is inspired from a widely-accepted IETF motto: "We believe in rough consensus and running code" [113]. Moreover, "Implementation experience provides critical feedback to the standardization process". We argue that we fulfilled this vision by prioritising the implementation of features that provide feedback to the implementers. Then, through discussions such as those reported in section 4.2.1 and 4.2.2, we reached consensus on what the tests should assess. Discussing the tests is a critical part of the process. As reported, implementers gave guidance on the specification that sometimes conflicted our interpretation. Since the beginning of our work, all of these conflicts arose from an erroneous interpretation from our part, but we argue that this process allows to detect inconsistencies in the specification that would be translated to inconsistencies in implementations.

[5] `https://github.com/mpiraux/master-thesis/commits?author=francoismichel`

Finally, we do not contest the usefulness of unit tests in the long term. Moreover, if we had to establish a list of objectives for the future of our work, we would certainly add the implementation of unit tests. They are important to gain credibility in order to attract new developers to contribute to the project. However, because we valued other objectives to support our goal, this work is left to be done.

5.3 Future prospects

In this section, we conclude this thesis by discussing the future prospects of our work. There are several parts of our work that are likely to continue to generate interest in the future.

Firstly, we implemented a number of scenarii that are not widely passed by implementations. For instance, the `new_connection_id` test has been implemented on the 15th of March, but no implementer has tried to develop the mechanism yet. Other tests, such as the `connection_migration` test, indicate that certain mechanisms are sometimes partially implemented. In this example, most implementations that do respond on the new path do not validate it after connection migration. Secondly, the test scenarii that collect the data we presented in section 4.1 will continue to do so. This will allow evaluating these metrics on a longer period of time. Thirdly, the introduction of new versions of the specification and the implementation of their support is likely to introduce regressions, such as those we reported in section 4.2. While the coverage of the test suite cannot be considered as high, we were still able to report regressions that were not detected by implementers.

However, as we noted previously, maintenance is required as the specification process continues. We hope to be able to dedicate some time to this task in the future, because of the personal interest for the subject we have developed during our work.

We established a methodology and a set of tools to implement new test scenarii and to be able to report bugs to implementers. We argue that our work is now valued inside the QUIC working group. However, the protocol specification is far from finished, as the base documents are supposed to be delivered in November 2018 [46]. Even then, several extensions are planned for QUIC such as multipath, forward error correction and explicit congestion notification. We hope that the assets we built will be used to continue to help the specification process of QUIC in the future.

Appendix A

QUIC transport parameters recorded over time

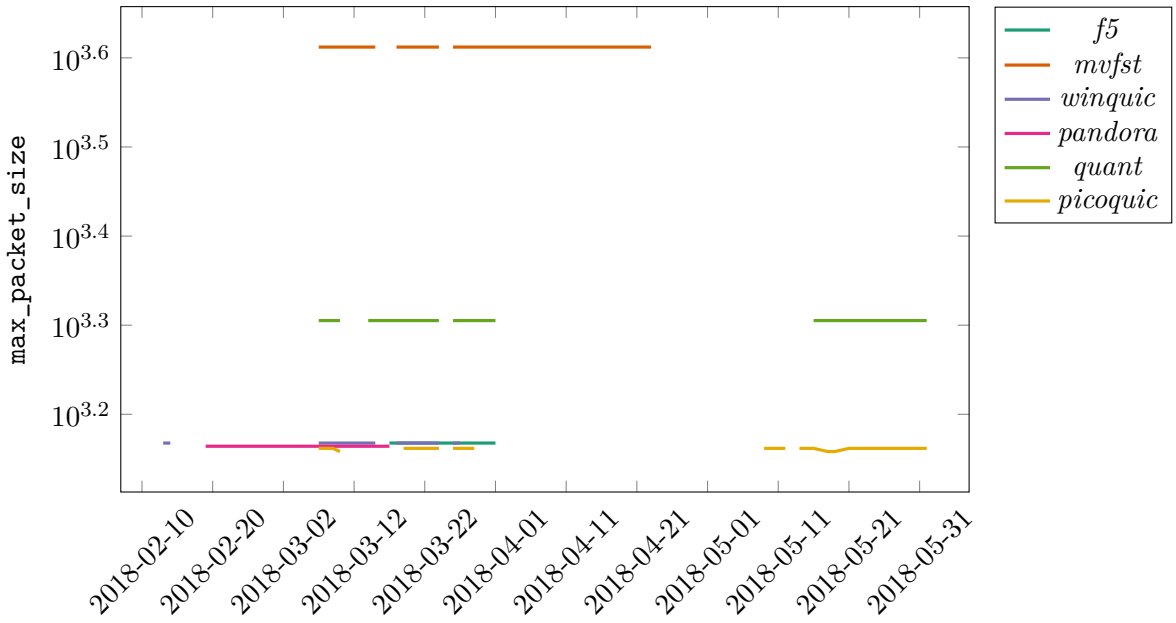


Figure A.1: Transport parameter `max_packet_size` announced by endpoints

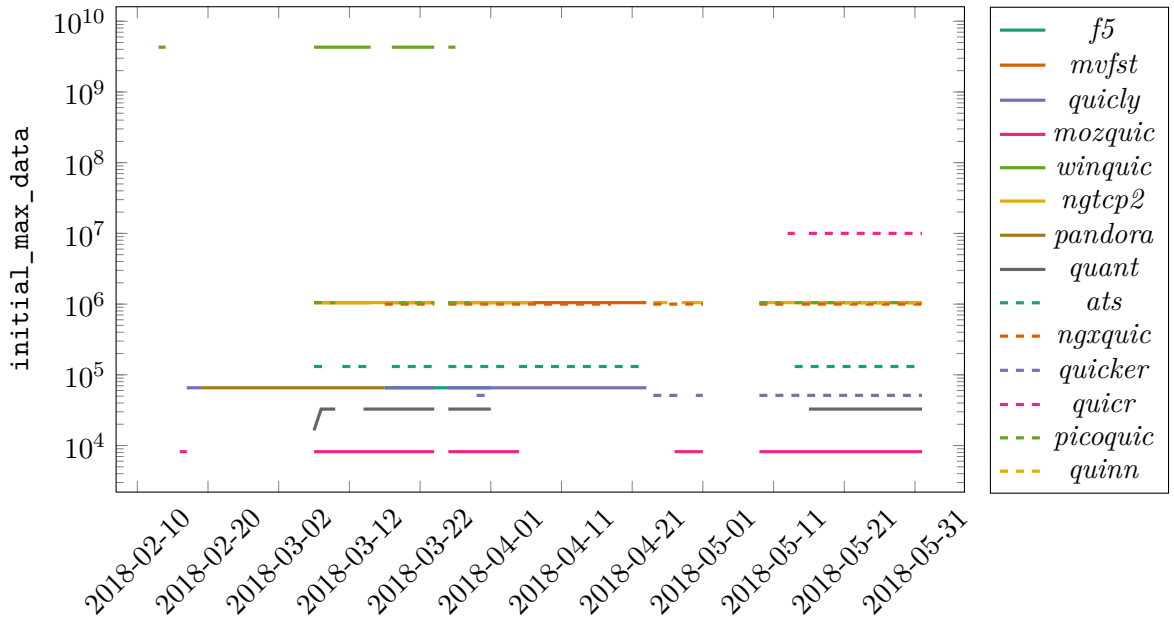


Figure A.2: Transport parameter `initial_max_data` announced by endpoints

Bibliography

[1] R. Elz and R. Bush, "Clarifications to the DNS specification," RFC 2181, RFC Editor, July 1997.

[2] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz, "Known TCP implementation problems," RFC 2525, RFC Editor, March 1999.

[3] J. R. Treurniet and J. Lefebvre, *A finite state machine model of TCP connections in the transport layer*. 2003.

[4] V. Paxson, "Automated packet trace analysis of TCP implementations," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 4, pp. 167–179, 1997.

[5] J. Pahdye and S. Floyd, "On inferring TCP behavior," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 287–298, 2001.

[6] N. Ekiz, A. H. Rahman, and P. D. Amer, "Misbehaviors in TCP SACK generation," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 2, pp. 16–23, 2011.

[7] J. Song, T. Ma, C. Cadar, and P. Pietzuch, "Rule-based verification of network protocol implementations using symbolic execution," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pp. 1–8, IEEE, 2011.

[8] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets," in *ACM SIGCOMM Computer Communication Review*, vol. 35, pp. 265–276, ACM, 2005.

[9] H. Zimmermann, "OSI reference model – The ISO model of architecture for open systems interconnection," *IEEE Transactions on communications*, vol. 28, no. 4, pp. 425–432, 1980.

[10] I. Standardization, "ISO/IEC 7498-1: 1994 information technology – Open systems interconnection – Basic reference model: The basic model," *International Standard ISOIEC*, vol. 74981, p. 59, 1996.

[11] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho, "Seven years and one day: Sketching the evolution of Internet traffic," pp. 711 – 719, 05 2009.

[12] D. Lee, B. E. Carpenter, and N. Brownlee, "Observations of UDP to TCP ratio and port numbers," in *Internet Monitoring and Protection (ICIMP), 2010 Fifth International Conference on*, pp. 99–104, IEEE, 2010.

[13] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, "Hypertext Transfer Protocol – HTTP/1.0," RFC 1945, RFC Editor, May 1996. `http://www.rfc-editor.org/rfc/rfc1945.txt`.

[14] D. Crocker, "Standard for the format of ARPA INTERNET text messages," STD 11, RFC Editor, August 1982.

[15] N. Borenstein and N. Freed, "MIME (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies," RFC 1521, RFC Editor, September 1993.

[16] K. Moore, "MIME (multipurpose internet mail extensions) part two: Message header extensions for non-ASCII text," RFC 1522, RFC Editor, September 1993.

[17] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, RFC Editor, June 1999. `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[18] C. Meyer and J. Schwenk, "Lessons learned from previous SSL/TLS attacks – a brief chronology of attacks and weaknesses.," *IACR Cryptology EPrint Archive*, vol. 2013, p. 49, 2013.

[19] B. Möller, T. Duong, and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," *Security Advisory*, 2014.

[20] S. Turner and T. Polk, "Prohibiting Secure Sockets Layer (SSL) version 2.0," RFC 6176, RFC Editor, March 2011. `http://www.rfc-editor.org/rfc/rfc6176.txt`.

[21] T. Dierks and C. Allen, "The TLS protocol version 1.0," RFC 2246, RFC Editor, January 1999. `http://www.rfc-editor.org/rfc/rfc2246.txt`.

[22] E. Rescorla, "HTTP over TLS," RFC 2818, RFC Editor, May 2000. `http://www.rfc-editor.org/rfc/rfc2818.txt`.

[23] R. Khare and S. Lawrence, "Upgrading to TLS within HTTP/1.1," RFC 2817, RFC Editor, May 2000.

[24] V. Cerf, Y. Dalal, and C. Sunshine, "Specification of Internet Transmission Control Program," RFC 675, RFC Editor, December 1974. `http://www.rfc-editor.org/rfc/rfc675.txt`.

[25] J. Postel, "Transmission Control Protocol," STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[26] J. Postel, "User Datagram Protocol," STD 6, RFC Editor, August 1980. `http://www.rfc-editor.org/rfc/rfc768.txt`.

[27] L. Dusseault and R. Sparks, "Guidance on interoperation and implementation reports for advancement to draft standard," BCP 9, RFC Editor, September 2009.

[28] Y. Sheffer and A. Farrel, "Improving awareness of running code: The implementation status section," BCP 205, RFC Editor, July 2016.

[29] B. Trammell and M. Kühlewind, "The Wire Image of a Network Protocol," Internet-Draft draft-trammell-wire-image-03, Internet Engineering Task Force, Apr. 2018. Work in Progress.

[30] K. Pentikousis and H. Badr, "Quantifying the deployment of TCP Options — A comparative study," vol. 8, pp. 647–649, 10 2004.

[31] M. Kühlewind, S. Neuner, and B. Trammell, "On the state of ECN and TCP options on the Internet," in *International Conference on Passive and Active Network Measurement*, pp. 135–144, Springer, 2013.

[32] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure, "Are TCP Extensions middlebox-proof?," in *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '13, (New York, NY, USA), pp. 37–42, ACM, 2013.

[33] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend tcp?," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pp. 181–194, ACM, 2011.

[34] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing middlebox interference with tracebox," in *Proceedings of the 2013 conference on Internet measurement conference*, pp. 1–8, ACM, 2013.

[35] A. Medina, M. Allman, and S. Floyd, "Measuring interactions between transport protocols and middleboxes," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 336–341, ACM, 2004.

[36] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? Designing and implementing a deployable Multipath TCP," pp. 29–29, 04 2012.

[37] C. Paasch, "Deploying TCP Fast Open in the wild," 2015. `https://www.ietf.org/proceedings/94/slides/slides-94-tcpm-13.pdf`.

[38] B. Trammell, M. Kühlewind, D. Boppart, I. Learmonth, G. Fairhurst, and R. Scheffenegger, "Enabling Internet-wide deployment of explicit congestion notification," in *International Conference on Passive and Active Network Measurement*, pp. 193–205, Springer, 2015.

[39] K. Edeline, M. Kühlewind, B. Trammell, E. Aben, and B. Donnet, "Using UDP for Internet transport evolution," 12 2016.

[40] R. P. Mike Belshe, "A 2x faster web," 2009. `https://research.googleblog.com/2009/11/2x-faster-web.html`.

[41] "SPDY: An experimental protocol for a faster web." `https://dev.chromium.org/spdy/spdy-whitepaper`.

[42] J. Roskind, "Quic: Quick UDP Internet connections: Multiplexed stream transport over UDP," 06 2013.

[43] A. Langley, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, A. Riddoch, W.-T. Chang, Z. Shi, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, and I. Swett, "The QUIC transport protocol: Design and Internet-scale deployment," pp. 183–196, 08 2017.

[44] L. Kleinrock, "The latency/bandwidth tradeoff in gigabit networks," *IEEE communications magazine*, vol. 30, no. 4, pp. 36–40, 1992.

[45] J. Iyengar and I. Swett, "Quic: A UDP-based secure and reliable transport for HTTP/2," Internet-Draft draft-tsvwg-quic-protocol-00, Internet Engineering Task Force, 06 2015. Work in Progress.

[46] "IETF – QUIC (quic) – milestones." `https://datatracker.ietf.org/wg/quic/about/`.

[47] J. Iyengar and I. Swett, "QUIC loss detection and congestion control," Internet-Draft draft-ietf-quic-recovery-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-11.txt.

[48] M. Thomson and S. Turner, "Using Transport Layer Security (TLS) to secure QUIC," Internet-Draft draft-ietf-quic-tls-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-11.txt.

[49] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," Internet-Draft draft-ietf-quic-transport-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-11.txt.

[50] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno modification to TCP's fast recovery algorithm," RFC 6582, RFC Editor, April 2012. http://www.rfc-editor.org/rfc/rfc6582.txt.

[51] E. Rescorla, "The Transport Layer Security (TLS) protocol version 1.3," Internet-Draft draft-ietf-tls-tls13-28, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-28.txt.

[52] C. Rossow, "Amplification hell: Revisiting network protocols for DDoS abuse.," in *NDSS*, 2014.

[53] G. Halkes and J. Pouwelse, "UDP NAT and firewall puncturing in the wild," in *International Conference on Research in Networking*, pp. 1–12, Springer, 2011.

[54] F. Audet and C. Jennings, "Network Address Translation (NAT) Behavioral requirements for unicast UDP," BCP 127, RFC Editor, January 2007. http://www.rfc-editor.org/rfc/rfc4787.txt.

[55] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP connection characteristics through passive measurements," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1582–1592, IEEE, 2004.

[56] S. Rewaskar, J. Kaur, and F. D. Smith, "A passive state-machine approach for accurate analysis of TCP out-of-sequence segments," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 3, pp. 51–64, 2006.

[57] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Learning fragments of the TCP network protocol," in *International Workshop on Formal Methods for Industrial Critical Systems*, pp. 78–93, Springer, 2014.

[58] M. Allman, W. M. Eddy, and S. Ostermann, "Estimating loss rates with TCP," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 3, pp. 12–24, 2003.

[59] M.Zalewski, "Passive OS finger printing tool," 2006. http://lcamtuf.coredump.cx/p0f3/.

[60] F. Qian, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: a fresh look at TCP in the wild," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pp. 76–89, ACM, 2009.

[61] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *ACM SIGCOMM Computer Communication Review*, vol. 32, pp. 309–322, ACM, 2002.

[62] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Computer Networks*, vol. 50, no. 1, pp. 46–62, 2006.

[63] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP congestion avoidance algorithm identification," *IEEE/Acm Transactions On Networking*, vol. 22, no. 4, pp. 1311–1324, 2014.

[64] G. F. Lyon, "Remote os detection via TCP/IP stack fingerprinting," *Phrack Magazine*, vol. 8, no. 54, 1998.

[65] K. Ramakrishnan and S. Floyd, "A proposal to add explicit congestion notification (ECN) to IP," RFC 2481, RFC Editor, January 1999.

[66] "CAIDA Internet data – Passive data sources." `www.caida.org/data/passive/`.

[67] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2018, RFC Editor, October 1996.

[68] S. Cheshire and M. Krochmal, "Multicast DNS," RFC 6762, RFC Editor, February 2013. `http://www.rfc-editor.org/rfc/rfc6762.txt`.

[69] S. Cheshire and M. Krochmal, "DNS-based service discovery," RFC 6763, RFC Editor, February 2013. `http://www.rfc-editor.org/rfc/rfc6763.txt`.

[70] S. Bradner, "Key words for use in rfcs to indicate requirement levels," BCP 14, RFC Editor, March 1997. `http://www.rfc-editor.org/rfc/rfc2119.txt`.

[71] M. J. Gordon and T. F. Melham, "Introduction to HOL – a theorem proving environment for higher order logic," 1993.

[72] R. E. Miller, D.-L. Chen, D. Lee, and R. Hao, "Coping with nondeterminism in network protocol testing," in *IFIP International Conference on Testing of Communicating Systems*, pp. 129–145, Springer, 2005.

[73] "Github `quicwg / base-drafts` Wiki – Implementations." `https://github.com/quicwg/base-drafts/wiki/Implementations`.

[74] "GNU affero general public license." `https://www.gnu.org/licenses/agpl-3.0.en.html`.

[75] "The Go programming language." `https://golang.org/doc/`.

[76] "`mint` – a minimal tls 1.3 stack." `https://github.com/bifurcation/mint/`.

[77] "Introducing JSON." `http://json.org/`.

[78] A. L. Goutte. personal communication.

[79] P. Wu. personal communication.

[80] "Wireshark." `https://www.wireshark.org/`.

[81] M. Cotton, B. Leiba, and T. Narten, "Guidelines for writing an IANA considerations section in RFCs," BCP 26, RFC Editor, June 2017. `https://tools.ietf.org/html/rfc8126#section-4.1`.

[82] M. Bishop, "Hypertext transfer protocol (HTTP) over QUIC," Internet-Draft draft-ietf-quic-http-11, IETF Secretariat, April 2018. `http://www.ietf.org/internet-drafts/draft-ietf-quic-http-11.txt`.

[83] "Flask – A Python microframework." http://flask.pocoo.org/.

[84] "AdminLTE." https://adminlte.io/.

[85] P. McManus, "mozquic – mozquic is an ietf quic library in c++." https://github.com/mcmanus/mozquic.

[86] "NSS key log format." https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format.

[87] P. Wu, "Wireshark decryption support for QUIC." https://www.ietf.org/mail-archive/web/quic/current/msg03692.html.

[88] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," Internet-Draft draft-ietf-quic-transport-09, IETF Secretariat, January 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-09.txt.

[89] "YAML: YAML Ain't Markup Language." http://yaml.org/.

[90] "picotls – TLS 1.3 implementation in C." https://github.com/h2o/picotls.

[91] K. Oku and C. Huitema, "Using picotls." https://github.com/h2o/picotls/wiki/Using-picotls.

[92] M. Nottingham, P. McManus, and J. Reschke, "Http alternative services," RFC 7838, RFC Editor, April 2016.

[93] "aiohttp – Asynchronous HTTP client/server for asyncio and Python." https://aiohttp.readthedocs.io/en/stable.

[94] M. Piraux, "A test suite for QUIC." https://www.ietf.org/mail-archive/web/quic/current/msg03555.html.

[95] M. Duke. personal communication.

[96] A. Ghedini. personal communication.

[97] K. Pittevils. personal communication.

[98] B. Saunders. personal communication.

[99] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A first look at QUIC in the wild," *arXiv preprint arXiv:1801.05168*, 2018.

[100] "eduroam – world wide education roaming for research & education." https://www.eduroam.org/.

[101] "Trafic autorisé pour les étudiants et les visiteurs qui se connectent via 'eduroam' ou 'visiteurs.uclouvain'." https://intranet.uclouvain.be/fr/myucl/services-informatiques/wifi-trafic-autorise-pour-etudiants-et-eduroam.html. The page requires an UCLouvain account.

[102] "Akamai community – FAQ: QUIC native platform support for media delivery products." https://community.akamai.com/customers/s/article/FAQ-QUIC-Native-Platform-Support-for-Media-Delivery-Products?language=en_US.

[103] "5th implementation draft." https://github.com/quicwg/base-drafts/wiki/5th-Implementation-Draft.

[104] C. Huitema and S. Iyengar. personal communication.

[105] C. Huitema. personal communication.

[106] T. Tsujikawa. personal communication.

[107] P. McManus. personal communication.

[108] M. Jones, J. Bradley, and N. Sakimura, "JSON web signature (JWS)," RFC 7515, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7515.txt.

[109] L. Eggert. personal communication.

[110] J. Reynders. personal communication.

[111] E. Rescorla. personal communication.

[112] S. Iyengar. personal communication.

[113] "IETF – how we work – running code." https://ietf.org/how/runningcode/.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**