Observing the Evolution of QUIC Implementations

Maxime Piraux, Quentin De Coninck, Olivier Bonaventure
UCLouvain
{maxime.piraux,quentin.deconinck,olivier.bonaventure}@uclouvain.be

ABSTRACT

The QUIC protocol combines features that were initially found inside the TCP, TLS and HTTP/2 protocols. The IETF is currently finalising a complete specification of this protocol. More than a dozen of independent implementations have been developed in parallel with these standardisation activities.

We propose and implement a QUIC test suite that interacts with public QUIC servers to verify their conformance with key features of the IETF specification. Our measurements, gathered over a semester, provide a unique viewpoint on the evolution of the QUIC protocol and of its implementations. They highlight the introduction of new features and some regressions among the different implementations.

ACM Reference Format:
Maxime Piraux, Quentin De Coninck, Olivier Bonaventure. 2018. Observing the Evolution of QUIC Implementations. In *Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ'18), December 4, 2018, Heraklion, Greece*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3284850.3284852

1 INTRODUCTION

Internet transport protocols usually evolve slowly. Any significant evolution to TCP, the dominant transport protocol, takes years of efforts to be widely deployed. There are several factors that explain this slow evolution [19]. On one hand, TCP is usually implemented inside the operating system kernel and upgrading kernels remains costly and slow. On the other hand, there are a growing number of middleboxes on the Internet that block new TCP extensions [13]. QUIC, initially proposed by Google [15] addresses this ossification in several ways. First, QUIC runs above UDP, which implies that it can be distributed as a userspace library that can be easily upgraded. Second, QUIC encrypts most of the control information and payload in order to prevent middlebox interferences.

The results obtained by Google with QUIC [15] combined with its security features have convinced the IETF to standardise a new protocol starting from Google's initial design.

EPIQ'18, December 4, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6082-1/18/12...$15.00
https://doi.org/10.1145/3284850.3284852

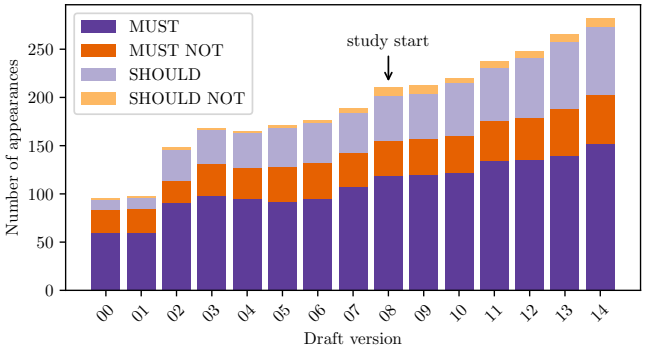

Figure 1: Keywords in `draft-ietf-quic-transport`.

The QUIC IETF working group was chartered in late 2016. It is currently finalising the specifications for the first standards-track version of QUIC. During this period, the IETF working group published fourteen versions of the main protocol draft and more than a dozen of QUIC implementations are publicly available.

The efforts of the QUIC designers and implementers is probably unique in the history of protocol design and implementation. Although many IETF protocols have been designed, very few have attracted so many implementers while the protocol was still being developed. As an illustration of the complexity of the QUIC protocol, Figure 1 plots the number of RFC2119 keywords (i.e. "MUST", "MUST NOT", "SHOULD", "SHOULD NOT") in the different versions of `draft-ietf-quic-transport`. We can observe that the number of these indicators has more than doubled since the first version of the specification.

Implementing network protocols is not a trivial task and several authors have proposed techniques to test and validate protocol implementations. Some of these techniques rely on formal methods to automatically derive the test suite from the implementation [7, 12]. However, it is difficult to apply them to Internet protocols since their specifications are informal. Researchers have proposed different solutions to test and validate protocol implementations. Some have proposed techniques to validate complete TCP implementations [1, 17]. Paxson et al. proposed specific tools to validate the conformance of TCP implementations [20, 21]. With TBIT, Padhye and Floyd developed techniques to infer the characteristics of TCP implementations by interacting with them with specially crafted packets [18]. These tools have played an important role in improving TCP implementations.

The dozen QUIC implementations [8] that are actively being developed will likely face similar problems as TCP implementations during the last decades [21]. The interoperability tests that are regularly organised by the QUIC IETF

working group have helped to identify some ambiguities in the specifications and solve interoperability problems. In this paper, we contribute to this implementation effort with a publicly available and detailed test suite for QUIC. To our knowledge, this is the first public test suite for this new protocol.

We first describe the architecture of our test suite in Section 2. Section 3 analyses results collected during a 6-month period using our test suite on the public servers that already implement QUIC. Section 4 concludes this paper by reviewing the future prospects for our work and assessing how it can be freely improved, reused and extended.

2 THE QUIC TEST SUITE

In this section we first describe both the approach and the architecture of our test suite. We then depict the test scenarios that constitute it.

2.1 Testing approach

Network protocol testing approaches can be categorised according to two dimensions, black-box versus white-box testing and passive versus active testing. The first dimension defines the perspective chosen to evaluate an implementation, i.e. an external or internal perspective. Because we want to test all QUIC implementations without relying on source code availability, we chose the black-box approach. Our test suite only observes their external behaviours, i.e. the packets sent and receiver, to evaluate them.

The second dimension defines how the tool behave with respect to the implementation under test (IUT). The first approach is passive testing. It has been used in earlier works [14, 20, 24], but is of limited use with QUIC given that most of the packets are encrypted.

The second approach is active testing, in which the tool used for experiments actively exchanges messages with the IUT. It requires the IUT to be available when using the tool. Several studies have applied this approach to TCP. TBIT [18] was one of the pioneering work in this domain. It has been extended in later works [4, 27]. Conducting active tests with TCP is becoming more difficult today given the deployment of various types of middleboxes that may interfere with active tests [11, 13, 16]. By encrypting most of the control information and payload, QUIC exposes a smaller surface subject to ossification.

The objective of our test suite is to check the conformance of QUIC server-side implementations by only exchanging packets with them. Given that the IETF specifications are still being developed, we only cover a subset of them. We want the tool to be autonomous in two manners. It must be able to create QUIC packets on its own to perform the tests. It must also be able to appropriately present bug reports for an implementer to locate which mechanisms caused their occurrence.

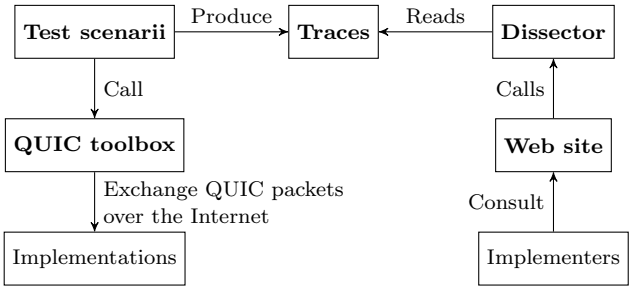

Figure 2: Tools forming the test suite.

2.2 Architecture

Figure 2 illustrates the architecture of our test suite. Akin to TBIT [18], our test suite is constituted of several self-contained scenarios. Each scenario is self-contained in the sense that it establishes a new QUIC connection to perform the test. A separate connection increases the isolation between two scenarios, in an attempt to accurately test a specific mechanism. Each test addresses a particular feature of the QUIC protocol. Combining several scenarios within a single connection is left as future work.

We implement the scenarios on top of a high-level API, i.e. the QUIC toolbox, that allows easily manipulating QUIC packets. Pieces of QUIC client behaviour are implemented as asynchronous message passing objects, called agents. We implemented 9 different agents responsible for, e.g. issuing `ACK` frames in response of received packets, retransmitting lost data, interacting with TLS, decrypting and parsing QUIC packets, bundling frames into packets and performing 1-RTT handshake.

This increases modularity by defining the behaviour of a test without having to reimplement mechanisms shared by several tests. For instance, the `address_validation` scenario, which tests whether a server validates the client address before sending significant amount of data to it, does not send acknowledgements but retransmits lost data and derive session keys from the handshake to decrypt all received packets. Therefore, it disables the agent responsible for acknowledgements, but enables the agents interacting with TLS and sending retransmissions.

The QUIC toolbox depends on *picotls* [5] for using TLS 1.3 through a Go binding we wrote [23]. The interaction between TLS and the toolbox is isolated in a separate 115-line long agent. One can thus replace the TLS stack used by implementing a new agent providing equivalent functionalities.

We synchronise the different agents using specific events inside a connection, e.g. a packet has been received or sent out, a new encryption scheme is available. This paradigm allows attaching new behaviours upon reception of these events without requiring extra coordination with other agents or having to define a common event loop for each connection. Agents can also emit events as their connection progresses.

Each test outputs its result in a JSON trace. Each trace contains an error code that summarises its outcome and a clear-text trace of the packets exchanged during its execution.

The error code is not purely binary, i.e. passed or failed. It can be used to discern various cases of failures to help the implementer to locate which part of the tested mechanism was deemed as erroneously implemented. For instance, the `zero_rtt` test can report whether a valid resumption token was sent by the server, whether it allowed the test suite to effectively succeed a 0-RTT handshake and whether the test suite could perform an 0-RTT HTTP request.

A trace can also contain scenario-specific data, such as the list of the protocol versions that were announced during the `version_negotiation` test and the transport parameters received during the `transport_parameters` test. Using this feature, we instrumented several test scenarios to collect several metrics. We present three of them in the Section 3.

To track the evolution of QUIC implementations, we ran an instance of the test suite every day. The different scenarios are run in a randomised order to prevent a particular sequence of tests from repeatedly impacting the data collection. A publicly accessible web application allows visualising the test results [22]. It eases the communication of bug reports to implementers and hopefully encourages them to consult test results. The presentation of the results emphasises on the cause of the problem so that implementers can efficiently diagnose which mechanism was erroneously implemented. The website also provides a detailed description of each test.

Our web application embeds a packet dissector we implemented. We chose to develop our own because existing packet dissectors, such as those in Wireshark [6], do not consistently support QUIC. Being able to dissect the packets exchanged by the test suite greatly improves its ability to efficiently present bug reports. The dissector operates based on a specification of the protocol written in YAML and a cleartext trace of the packets exchanged. We maintain separate specifications for different QUIC versions in order to ensure backward compatibility.

Our QUIC toolbox consists of more than 3600 lines of Go code. The web application consists of 1100 lines of Python code, while the dissector is 300-line long supplemented by 1600 lines of YAML for protocol descriptions.

2.3 Testing the specification

We derive test scenarios from the QUIC specification. This process cannot be automated, because the specification is written in English in an informative style. We analyse the sentences containing strong indicators of importance as defined in RFC2119 [2], i.e. sentences containing the words "MUST" or "MUST NOT". We then extract rules from these sentences that should not be violated throughout the test. Based on these rules we design a scenario that ensures that these rules are not violated. The tests follow the evolution of the specification and are updated accordingly. We prioritise the design of tests that involve features chosen by the QUIC working group as part of the *Implementation Drafts* [26] to provide valuable feedback.

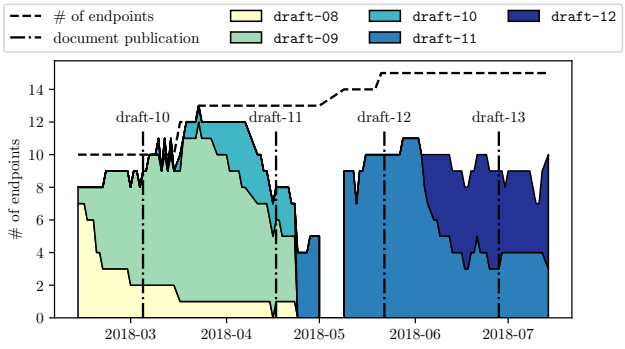

Figure 3: Number of endpoints announcing different draft versions.

Our current test suite contains 18 test scenarios. In the interest of space, we do not present all of them but summarise the mechanisms they verify. Eight of them check several aspects of the QUIC handshake, such as the *0-RTT*, the exchange of the *Transport Parameters* and the *Version Negotiation*. Six of them focus on QUIC streams, e.g. bidirectional and unidirectional support, flow control and reordering in stream transitions. Two of them test the handling of acknowledgements and the support for Explicit Congestion Notification (ECN). Finally, two tests explore connection migration and new connection ID support.

By manually analysing the 14th version of `draft-ietf-quic-transport`, we identified 29 strong requirements covered by the test suite, i.e. "MUST" and "MUST NOT". We note that 41 of the 203 strong requirements are only applicable to QUIC client implementations and thus out of the scope of our tool. These 18 test scenarios are implemented in about 1200 lines of Go code.

3 TEST RESULTS

We used our QUIC test suite on the public endpoints of QUIC implementations during a 6-month period, starting from the 12th of February to the 15th of July 2018[1]. We updated the list of public endpoints when they were publicly announced on the communication channel dedicated to testing coordination [10].

We report our results into three phases. First, we provide a high-level view showing key metrics collected by our test suite. Then, we dig in two case studies on two specific test scenarios. Finally, we present a snapshot of the test results to show the diversity of behaviours between studied implementations.

3.1 Measurements

In this section, we present three metrics extracted from the data collected by the test suite, i.e. the announced QUIC version, the handshake success and the test outcome percentage. For each metric we explain how the measurements were conducted and what conclusion can be drawn from them.

[1] A bug was introduced on the 1st of May, preventing data collection until the 8th of May.

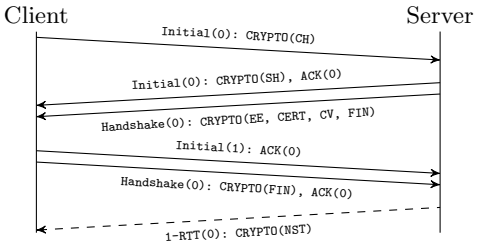

Figure 4: 1-RTT connection in our `handshake` test.

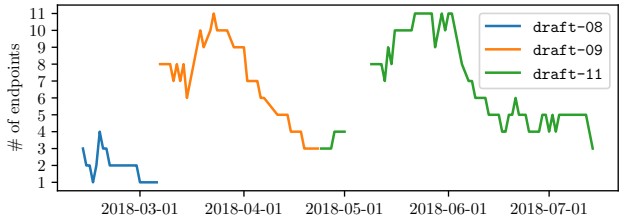

Figure 5: Number of endpoints succeeding our `handshake` test.

Deployment of QUIC. We first report the result collected by the `version_negotiation` scenario. This test triggers the version negotiation process and records the versions announced by the tested implementations. In a sense, this is similar to the measurements carried out by Rüth et al. to identify the number of servers that support Google's version of QUIC (gQUIC) over the entire IPv4 addressing space [25]. Figure 3 summarises our results over the 6-month period. We restrict the figure to the five main versions of the draft specifications [9]. It also indicates the number of endpoints we tested over this period.

We can observe that when a new version of the specification was published, most implementations chose to stop supporting the previous version in favour of the new one without maintaining backward compatibility. This is reflected in the figure by a simultaneous increase and decrease between two successive versions. This lack of backward compatibility is normal for prototype implementations. It contrasts with the findings of Rüth et al. about public gQUIC servers that gradually update the set of versions they support.

QUIC versions can introduce lot of changes, including modifications to the QUIC invariants about the public header format. `draft-11` is an example. It introduced a version negotiation process that is not backward-compatible. We updated the test suite to support `draft-11` shortly after its publication. From this point, we were unable to observe prior versions.

We can conclude from Figure 3 that the implementers of QUIC often need between fifteen days to a month to integrate the changes published in a new version of the specification to their implementations. As a result, tracking the behaviour of these QUIC implementations requires to regularly update the test suite.

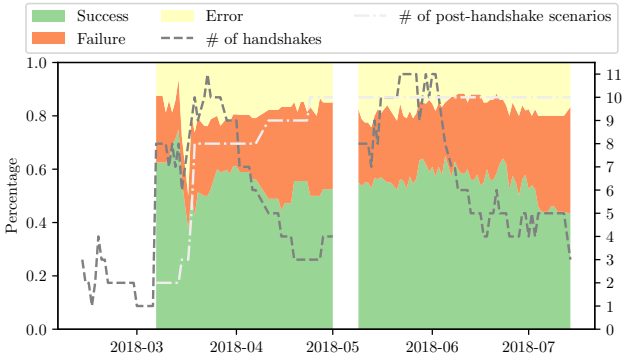

Figure 6: Percentage of outcomes for post-handshake tests.

Successful handshakes. Version negotiation and connection establishment being separate mechanisms, there could be a mismatch between the number of implementations that announce a particular version and the number that effectively support it. To investigate this possibility, we report the number of implementations that successfully performed a 1-RTT handshake with our test suite. It is based on the data collected by the `handshake` scenario which discerns various causes of 1-RTT handshake failure. Figure 4 illustrates the behaviour of our `handshake` test. We can observe that the test performs a complete 1-RTT handshake and derives the corresponding session keys. The server can send a *New Session Ticket* (NST) which will be decrypted by the test.

Figure 5 reports the number of endpoints that succeed our handshake test over the 6-month period. During this period, we implemented `draft-08`, `draft-09` and `draft-11` of the specification. We chose to not deploy `draft-10`, because most implementers indicated that they were willing to support the next version as soon as possible [10]. `draft-10` contained very few new features when compared to `draft-11`. We can indeed observe in Figure 3 that only a maximum of four implementations simultaneously announced its support.

Overall, the resulting graph contains several slight fluctuations when compared to Figure 3. These fluctuations are a result of the rapid pace at which changes are deployed amongst all the tested implementations. Some of these changes have caused interoperability problems. This is expected as implementing the version negotiation involves simpler mechanisms than the 1-RTT handshake.

Test suite outcome percentage. We present the evolution of the success, failure and error rates over the entire test suite during our data collection period in Figure 6. We computed the percentage over the tests that require the handshake to complete and only kept the implementation that succeeded this handshake. We overlay the number of these implementations as well as the number of these tests on this figure. *Success* corresponds to a successful execution of a test. *Failure* reports the tests that were violated and *Error* reports the tests for which a prerequisite was missing, e.g.

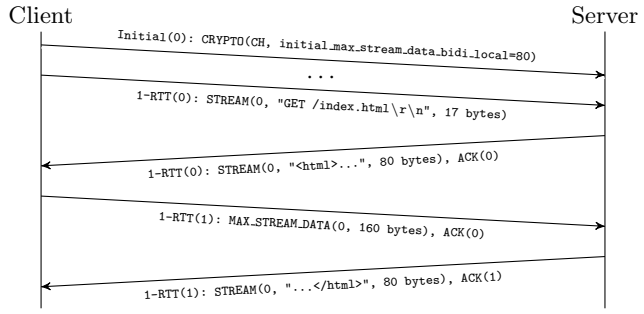

Figure 7: Example flow for our `flow_control` test.

the endpoint crashed, no IPv6 address could be resolved, no unidirectional streams were available, ...

We can note that most of the fluctuations occurred during March 2018 when most of the tests were introduced. Based on the feedback of implementers [10], we updated several tests to improve their correctness and address false positives.

The Figure illustrates that the *Success-Failure* ratio follows the number of implementations available. The most active ones rapidly move from one version to the next one and positively impact this ratio. Implementations that are slower to evolve usually have a lower ratio. Finally, the curves show many fluctuations that are indicative of the dynamic nature of these QUIC implementations.

3.2 Case studies

We review in this section some test scenarios which reported bugs in several implementations. For each test, we first explain its intent, then we report the evolution of its results based on the feedback submitted to and received from the implementers. We concentrate on a 3-month period starting on the 1st of March 2018.

Flow control. Flow control is an important part of a transport protocol. It prevents a fast sender from overwhelming a slow receiver. A peer can signal flow control through two different mechanisms in QUIC. The first are the transport parameters. For instance, parameter `initial_max_stream_data_bidi_local` allows a client to limit the amount of data that a server can send on a stream initiated by the former. The second is the `MAX_STREAM_DATA` frame, which advertises higher limits.

The `flow_control` test, as illustrated in Figure 7, initiates a connection and sets the `initial_max_stream_data_bidi_local` parameter to 80 bytes. This limit has been chosen sufficiently low to be smaller than most of the web pages that are hosted by the endpoints. Once the connection is established, the test sends an HTTP request and waits for the server to send the first 80 bytes of the response. The server must not send more than 80 bytes because of the limit imposed by the transport parameter. Once these bytes are received, the test sends a `MAX_STREAM_DATA` frame that raises the limit to 160 bytes. The test ensures that the server resumes the sending of data after receiving the frame.

We found several implementations failing this test at different stages of the specification process. On the 10th and 17th of March 2018 two implementations entered a loop when running the `flow_control` scenario. We reported these bugs and discussed with their implementers. The first was repeatedly sending `ACK` frames due to an incorrect integration of flow control with other mechanisms. The second was sending empty `STREAM` frames, which is forbidden by the specification, because of a missing corner-case when clamping these frames according to flow control. The test results collected after the 20th of March indicated that both implementers had fixed the bug.

We found another implementation that incorrectly implemented flow control on the 23rd of March. It only divided its response into two pieces, the first being 80-bytes long. We reported the bug and were notified that a fix was implemented, which was confirmed by the test results shortly after. On the 18th of May 2018, after this implementation added support for `draft-11`, we observed a regression regarding this test. The implementation aggressively sent `STREAM_BLOCKED` frames and retransmissions of the second half of data requested. We did not observe the deployment of a fix before the end of our data collection. We later learned that its developer was not active any more during this period.

Stream transitions reordering. A QUIC implementation must be able to react appropriately when packet reordering occurs. We can discern two cases which can induce packet reordering. The first is introduced by the use of different network paths, due to, e.g. load balancers. The second one is caused by a packet loss during the transmission of a series of packets. The data of the lost packet will be retransmitted and received after the rest of the series.

The `stream_opening_reordering` test simulates the first type of reordering. It initiates a connection and then sends an HTTP request in two packets. The first packet contains the graceful closure of the client-side of the request stream. The second contains the data of the stream, which contains the request. The first packet is sent with a higher packet number than the second packet. The test successfully completes once the server has responded to the request.

We report three of the cases we observed during the 3-month period. The first one lead to a one-to-one conversation with an implementer. The scenario triggered a livelock in their implementation and the latter did not produce any kind of observable external behaviour. We provided assistance to install the test suite and run it against a local and better-instrumented version of the implementation.

On the 11th of May, we detected a regression for a particular implementation for which support of `draft-11` was recently added. We were not actively analysing the data on this day and thus did not report the bug. We later found that the implementer had consulted the test result and fixed the bug. We argue that this is an indication of the benefits of an autonomous test suite that runs on a daily basis and provides public results.

Finally, this test triggered a bug in the `ACK` frame generation of an implementation on the 22nd of May. We believe

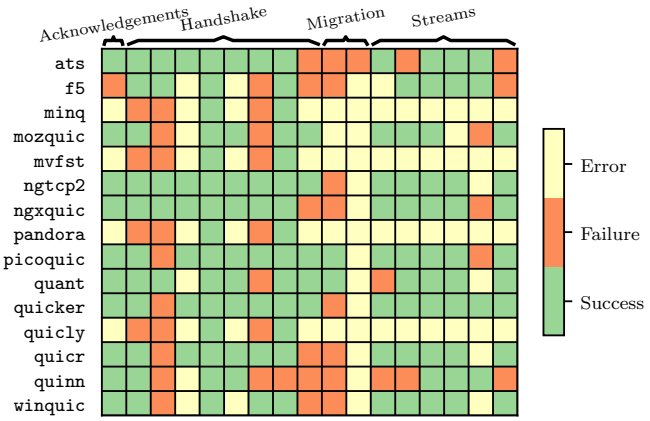

Figure 8: Results grid on the 1st of June 2018.

that the bug was discovered shortly after its introduction, as the results from the past days did not reveal it. The bug caused the generated `ACK` frame to report $2^{64} - 1$ missing packets, probably due to an overflow induced by the reordering of packets. Indeed, considering that one could determine the gap between the received packet and the last received packet by subtracting the received packet number with the last received packet number, reordering then causes an overflow. The implementation source code not being public, we could not confirm this hypothesis.

3.3 Results grid

We conclude this section by presenting Figure 8 which summarises the outcomes for the different tests and QUIC implementations. The grid is a snapshot captured on the 1st of June 2018.

Those results show the diversity of the outcomes generated by the available implementations. We can observe that three of them, i.e. `minq`, `pandora` and `quicly` only succeeded two scenarios. These scenarios only collect metrics and do not enforce requirements when the IUT is unavailable. We can also note that most of the runs of the two connection migration tests were either unsuccessful or could not execute. While the mechanisms tested were part of `draft-11`, the implementers did not include them in the corresponding *Implementation Draft* [3].

4 DISCUSSION

In this paper we have proposed a first active test suite for the QUIC protocol based on the current IETF specification. We detailed its architecture and the supported test scenarios. We presented the results collected using this test suite and reported two case studies. The test suite has already been used by the QUIC community. Its source code is publicly available under an open-source licence[2]. Two implementers have already integrated the test suite as part of their workflow, independently of our public instance.

The tool being open also implies that it can be reused, extended and improved by the QUIC community. Due to the very modular design of our architecture, it can be extended in different ways. New scenarios can be implemented to cover new features of the protocol and collect new metrics. The QUIC toolbox can be reused for other purposes. It can also be extended with new features that are not currently supported, such as sending coalesced packets, or improved with a better user-facing API. The visualisation application can also be improved, e.g. by adding more feedback to the implementers based on the trace format. We note that one is not required to use this web application, and can instead consume the test results in the traces using other applications. For instance, we developed a set of scripts that allows generating CSVs based on these traces, which were used to produce Figure 3 and Figure 5[3].

We plan to continue to update the test suite to track the evolution of the IETF specification and later to detect how QUIC server implementations have been tuned with heuristics for, e.g. retransmissions and congestion control schemes. However, we limit our approach only to QUIC servers. The protocol also requires the compliance of QUIC clients to the specification. Including them in our study would raise several challenges beyond the additional implementation efforts.

How to initiate connections from clients to the test tool ? We chose a black-box approach as the source code of QUIC implementations is not always available. Applying this approach to client testing requires some techniques to encourage diverse clients to connect to the test tool.

How to identify the various implementations connecting to the test tool ? In the server-side approach, we know to which servers the test tool connects to. If many clients can anonymously connect to the test tool, correctly identifying the tested implementations becomes critical for providing relevant feedback to the QUIC implementers.

Which QUIC clients are widely-deployed today ? While several server implementations are known thanks to their participation in the IETF interoperability tests, there is no equivalent for clients as of this writing. However, this is likely to change as the QUIC specification is finalised.

We intend to include QUIC clients to our approach in the future and we hope to be able to capture the full diversity of the emerging QUIC ecosystem, with as many interesting behaviours as we presented in this study.

ACKNOWLEDGEMENTS

We would like to thank the QUIC implementers and the participants to the IETF Hackathon in London who provided feedback on the test suite. This work was partially supported by funding from the Walloon Government (DGO6) within the MQUIC project.

[2] See https://github.com/QUIC-Tracker

[3] https://github.com/QUIC-Tracker/web-app/tree/master/quic_tracker/postprocess

REFERENCES

[1] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *ACM SIGCOMM Computer Communication Review*, Vol. 35. ACM, 265–276.

[2] Scott Bradner. 1997. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119.

[3] Lars Eggert. 2018. 5th Implementation Draft. (2018). https://github.com/quicwg/base-drafts/wiki/5th-Implementation-Draft

[4] Nasif Ekiz, Abuthahir Habeeb Rahman, and Paul D Amer. 2011. Misbehaviors in TCP SACK generation. *ACM SIGCOMM Computer Communication Review* 41, 2 (2011), 16–23.

[5] Kazuho Oku et al. 2018. `picotls` – TLS 1.3 implementation in C. (2018). https://github.com/h2o/picotls

[6] Alexis La Goutte. 2018. Bug 13881 - Add (IETF) QUIC Dissector. (2018). https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=13881

[7] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. 2003. An introduction to the testing and test control notation (TTCN-3). *Computer Networks* 42, 3 (2003), 375–403.

[8] QUIC Working Group. 2018. QUIC Implementations. (2018). https://github.com/quicwg/base-drafts/wiki/Implementations

[9] QUIC Working Group. 2018. QUIC Versions. (2018). https://github.com/quicwg/base-drafts/wiki/QUIC-Versions

[10] QUIC Working Group. 2018. quicdev Slack. (2018). https://quicdev.slack.com/

[11] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. 2013. Are TCP extensions middlebox-proof?. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. ACM, 37–42.

[12] Gerard J Holzmann and William Slattery Lieberman. 1991. *Design and validation of computer protocols*. Vol. 512. Prentice hall Englewood Cliffs.

[13] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is it still possible to extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 181–194.

[14] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. 2004. Inferring TCP connection characteristics through passive measurements. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, Vol. 3. IEEE, 1582–1592.

[15] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 183–196.

[16] Alberto Medina, Mark Allman, and Sally Floyd. 2004. Measuring interactions between transport protocols and middleboxes. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 336–341.

[17] Madanlal Musuvathi, Dawson R Engler, et al. 2004. Model Checking Large Network Protocol Implementations.. In *USENIX NSDI'04*. 12–12.

[18] Jitendra Pahdye and Sally Floyd. 2001. On inferring TCP behavior. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 287–298.

[19] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic, et al. 2017. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials* 19, 1 (2017), 619–639.

[20] Vern Paxson. 1997. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review* 27, 4 (1997), 167–179.

[21] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. 1999. *Known TCP Implementation Problems*. RFC 2525. RFC Editor.

[22] Maxime Piraux. 2018. QUIC-Tracker web application. (2018). https://quic-tracker.info.ucl.ac.be

[23] Maxime Piraux. 2018. `pigotls` – A very minimal Go binding for `picotls`. (2018). https://github.com/mpiraux/pigotls

[24] Sushant Rewaskar, Jasleen Kaur, and F Donelson Smith. 2006. A passive state-machine approach for accurate analysis of TCP out-of-sequence segments. *ACM SIGCOMM Computer Communication Review* 36, 3 (2006), 51–64.

[25] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. 2018. A First Look at QUIC in the Wild. In *Passive and Active Measurement*. Springer International Publishing, 255–268.

[26] Martin Thomson and Lars Eggert. 2018. 7th Implementation Draft. (2018). https://github.com/quicwg/base-drafts/wiki/7th-Implementation-Draft

[27] Peng Yang, Juan Shao, Wen Luo, Lisong Xu, Jitender Deogun, and Ying Lu. 2014. TCP congestion avoidance algorithm identification. *IEEE/ACM Transactions On Networking* 22, 4 (2014), 1311–1324.