Automating QUIC Interoperability Testing

Marten Seemann
Protocol Labs
marten@protocol.ai

Jana Iyengar
Fastly
jri@fastly.com

ABSTRACT

We present QuicInteropRunner (QIR) [1, 2], a test framework for automated and on-demand interoperability testing between implementations of the QUIC protocol [3]. QIR is a framework in which QUIC clients and servers interact with each other over a network that simulates various network conditions using ns-3 [4]. QIR automates QUIC interoperability testing by running a suite of test cases between containerized QUIC implementations. We describe the key constraints and insights that defined our work, recent innovations that made the framework possible, a high-level overview of our design, and a few exemplary tests. QIR is now supported and used by ten QUIC implementations as part of their development process, confirming our thesis that there is a need for automating interoperability testing and making it available on demand.

CCS CONCEPTS

• **Networks** → **Protocol testing and verification**; **Transport protocols**;

KEYWORDS

QUIC, interoperability testing, ns-3

ACM Reference Format:
Marten Seemann and Jana Iyengar. 2020. Automating QUIC Interoperability Testing. In *Workshop on Evolution, Performance, and Interoperability of QUIC (EPIQ '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3405796.3405826

1 INTRODUCTION

The Internet is a multi-vendor system defined by open standards. Interoperability testing has long been a cornerstone of the development of these open standards, for two reasons. First, different implementations of an open protocol interact with each other on the Internet and therefore need to be tested for those interactions. Second, it exposes gaps and ambiguities in the specification, as different implementations can make conflicting assumptions in such cases. For both of these reasons, the IETF, the primary standards body for Internet protocols, requires interoperability testing as a part of the development process [5].

Interoperability testing however has historically been a manual process. For example, SCTP [6] interoperability meetings entailed bringing computers into a room, wiring them to the same network, manually running various tests, and examining the outcomes locally. More recent efforts have relied on Internet infrastructure for testing where possible, removing the need for co-locating implementations. For instance, HTTP/2 and QUIC implementers would set up servers running their implementations on publicly accessible Internet endpoints, against which others could run tests. Nevertheless, the testing itself has remained manual [7].

Manual testing suffers from three significant scaling limitations. First, it limits the number of implementations that can be tested, since there are a quadratic number of combinations to test. Second, it limits the number of features that can be tested, a problem that is made worse by the complexity of protocols built for the modern Internet.

Finally, it limits the range of network conditions under which the protocol is tested. Ad-hoc testing over a local network or over the public Internet does not test the implementations' performance in the variety of network conditions under which the protocol is expected to perform well. As a result, various parts of the protocol, such as those designed to handle adverse network conditions, likely remain untested.

In the IETF's QUIC working group [8], these limitations meant that comprehensive interoperability testing was only performed roughly once every month. This interoperability did not include repeatable and precise tests of the implementations, and the outcomes of the tests were determined by manual inspection of logs [7].

```
+----------+---------+--------+----------+
|          | quic-go | quicly | picoquic |
+----------+---------+--------+----------+
| quic-go  | HSMZB3  |  HSMB  |  HSMZB3  |
|          |         |  Z3    |          |
|          |         |        |          |
+----------+---------+--------+----------+
|  quant   | HSMZB   |  HSMB  |  HSMZB   |
|          |   3     |  Z3    |    3     |
|          |         |        |          |
+----------+---------+--------+----------+
|  mvfst   |  HB3    |  HB    |   HB3    |
|          |   S     |  SZ3   |    S     |
|          |   MZ    |  M     |    MZ    |
+----------+---------+--------+----------+
```

Figure 1: Console output of a local run of the QUIC interop runner. Column headers refer to servers and row headers refer to clients. Tests are indicated by their letter symbols in each cell. Test outcomes are Success, Unsupported, or Failure, as shown in the top, middle, and bottom rows within each cell. Endpoint implementations and test cases can be specified via command line parameters, allowing implementers to focus their testing on specific pairs and interactions. Test cases shown here are H: Handshake, S: Retry, M: Multiplexing, B: Blackhole, Z: 0-RTT, 3: HTTP/3.

EPIQ '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-8047-8/20/08…$15.00
https://doi.org/10.1145/3405796.3405826

Run: 2020-05-05T16:21:25UTC
Start Time: 5/5/2020 4:21:25 PM UTC
Duration: 16:27:37

Interop

	quic-go	quicly	ngtcp2	quant	mvfst	quiche	picoquic	aioquic
quic-go	H D C M S R Z 3 B L1 L2 C1 C2	H D C M S R B L1 L2 C1 C2 Z 3	3 H D C M S R Z B L1 L2 C1 C2	H D C M S R Z B L1 L2 C1 C2 3	H D C M R Z 3 B L2 C2 S L1 C1	H D C M S R Z 3 B L2 C2 L1 C1	H D C M S R Z 3 B L1 L2 C1 C2	H D C S R 3 B L1 L2 C2 Z M C1
quicly	H D C M S R B L1 L2 C2 Z 3 C1	H D C M S R B L1 L2 C1 C2 Z 3	 Z 3 H D C M S R B L1 L2 C1 C2	 Z 3 H D C M S R B L1 L2 C1 C2	H D C L2 C2 S Z 3 L1 C1 M R B	H D C M S R B L2 C2 Z 3 L1 C1	 Z 3 H D C M S R B L1 L2 C1 C2	H D C S R B L1 L2 C1 C2 Z 3 M
ngtcp2	3 H D C M S R Z B L1 L2 C1 C2	 Z 3 H D C M S R B L1 L2 C1 C2	H D C M S R Z 3 B L1 L2 C2 C1	 3 H D C M S R Z B L1 L2 C1 C2	H D C M 3 B L2 C2 S L1 C1 R Z	H D C M S R Z 3 B L2 C2 L1 C1	H D C M S R Z 3 B L2 C2 L1 C1	H D C S R 3 B L1 L2 C2 Z M C1
quant	H D C M S R Z B L1 L2 C1 C2 3	H D C M S R B L2 Z 3 L1 C1 C2	 3 H D C M S R Z B L1 L2 C1 C2	H D C M S R Z B L1 L2 C1 C2 3	H D C R Z B L2 C2 S 3 L1 C1 M	H D C M S R Z B L2 C2 3 L1 C1	H D C M S R Z B L2 C1 C2 3 L1	H D C S R B L1 L2 C1 C2 Z 3 M
mvfst	H D C Z 3 B L2 C2 S L1 C1 M R	H D C L2 C2 S Z L1 C1 M R 3 B	3 S L1 C1 H D C M R Z B L2 C2	H D C Z B L2 C2 S 3 L1 C1 M R	H D C M Z 3 B L2 C2 S L1 C1 R	H D C Z L2 C2 S L1 C1 M R 3 B	H D C Z 3 B L2 C2 S L1 C1 M R	L2 C2 S Z L1 C1 H D C M R 3 B
quiche	H D C M S 3 B L2 C2 R Z L1 C1	H D C M S B L2 C2 R Z 3 L1 C1	H D C M S 3 B L2 C2 R Z L1 C1	 R Z 3 H D C M S B L1 L2 C1 C2	H D C 3 B L2 C2 S R Z L1 C1 M	H D C M S 3 B L2 C2 R Z L1 C1	 R Z H D C M S 3 B L1 L2 C1 C2	H D C S 3 B L2 C2 R Z M L1 C1
kwik	H D C M S R Z 3 B L1 L2 C1 C2	H D C M S R B L1 L2 C2 Z 3 C1	3 H D C M S R Z B L1 L2 C1 C2	H D C M S R Z B L1 L2 C1 C2 3	H D C M R Z 3 B L2 C2 S L1 C1	H D C M S R Z 3 B L2 C1 C2 L1	H D C M S R Z 3 B L1 L2 C1 C2	H D C S R 3 B L1 L2 C1 C2 Z M
picoquic	H D C M S R Z 3 B L1 L2 C1 C2	H M S R L1 L2 C1 C2 Z 3 D C B	H D C M S R Z 3 B L2 C2 L1 C1	H D C M S R Z B L1 L2 C1 C2 3	H D C R Z 3 L2 C2 S L1 C1 M B	H D C M S R Z 3 B L1 L2 C1 C2	H D C M S R Z 3 B L1 L2 C1 C2	H S R 3 L2 Z D C M B L1 C1 C2
aioquic	H D C M S R 3 B L1 C1 Z L2 C2	H D C M S R B L2 C2 Z 3 L1 C1	3 Z H D C M S R B L1 L2 C1 C2	 Z 3 H D C M S R B L1 L2 C1 C2	H D C M R 3 B L2 C2 S Z L1 C1	H D C M S R 3 B L2 C2 Z L1 C1	 Z H D C M S R 3 B L1 L2 C1 C2	H D C S R 3 B L2 C1 C2 Z M L1
neqo	H 3 C2 S R Z L1 C1 D C M B L2	H S R Z 3 L1 C1 D C M B L2 C2	 S R Z L1 C1 H D C M 3 B L2 C2	H S R Z 3 L1 C1 D C M B L2 C2	H M S R Z L1 C1 D C 3 B L2 C2	H D C 3 S R Z L1 C1 M B L2 C2	H 3 S R Z L1 C1 D C M B L2 C2	H D C 3 C2 S R Z L1 C1 M B L2

Measurements

	quic-go	quicly	ngtcp2	quant	mvfst	quiche	picoquic	aioquic
quic-go	G: 9513 (± 23) kbps C: 5429 (± 154) kbps	G: 9501 (± 16) kbps C: 6214 (± 236) kbps	G C	G: 8702 (± 157) kbps C: 3720 (± 212) kbps	G: 9334 (± 19) kbps C: 8852 (± 112) kbps	G: 9523 (± 11) kbps C: 5750 (± 130) kbps	G: 9474 (± 18) kbps C: 7179 (± 65) kbps	G: 9115 (± 21) kbps C: 4986 (± 351) kbps
quicly	G: 9566 (± 1) kbps C: 7248 (± 93) kbps	G: 9482 (± 32) kbps C: 5057 (± 323) kbps	G C	G C	G: 9290 (± 46) kbps C: 8943 (± 119) kbps	G: 9553 (± 0) kbps C: 6050 (± 343) kbps	G C	G: 9383 (± 11) kbps C: 4868 (± 189) kbps
ngtcp2	G C	G C	G: 9165 (± 18) kbps C: 5103 (± 184) kbps	G C	G: 7080 (± 1765) kbps C	G: 9203 (± 7) kbps C: 6092 (± 351) kbps	G: 9367 (± 13) kbps C: 7320 (± 393) kbps	G: 9128 (± 19) kbps C: 5377 (± 288) kbps
quant	G: 9451 (± 17) kbps C: 6908 (± 432) kbps	G: 9469 (± 7) kbps C: 6326 (± 265) kbps	G C	G: 8986 (± 76) kbps C: 4769 (± 170) kbps	G: 9284 (± 3) kbps C: 8887 (± 123) kbps	G: 9509 (± 11) kbps C: 6156 (± 223) kbps	G: 9417 (± 10) kbps C: 7086 (± 103) kbps	G: 9348 (± 10) kbps C: 5489 (± 405) kbps
mvfst	G C: 5490 (± 455) kbps	G: 9423 (± 7) kbps C: 4962 (± 184) kbps	G C	G C	G: 8918 (± 128) kbps C: 8494 (± 97) kbps	G C	G: 8939 (± 70) kbps C	G C
quiche	G: 9324 (± 15) kbps C: 6527 (± 60) kbps	G: 9324 (± 14) kbps C: 6760 (± 349) kbps	G: 9233 (± 16) kbps C: 5557 (± 117) kbps	G C	G: 9076 (± 76) kbps C	G: 9057 (± 36) kbps C: 6080 (± 106) kbps	G C	G: 8886 (± 21) kbps C: 5640 (± 165) kbps
kwik	G: 8819 (± 23) kbps C: 6626 (± 202) kbps	G: 8846 (± 17) kbps C: 6258 (± 139) kbps	G C	G: 6896 (± 217) kbps C: 4495 (± 264) kbps	G: 8205 (± 85) kbps C: 8253 (± 150) kbps	G: 8788 (± 212) kbps C: 6256 (± 160) kbps	G: 8903 (± 13) kbps C: 7402 (± 217) kbps	G: 8639 (± 62) kbps C: 5644 (± 269) kbps
picoquic	G: 9432 (± 54) kbps C: 6015 (± 343) kbps	G C	G: 9255 (± 15) kbps C: 5049 (± 295) kbps	G: 9084 (± 40) kbps C: 5081 (± 623) kbps	G: 9188 (± 63) kbps C: 8785 (± 58) kbps	G C	G: 9418 (± 4) kbps C: 7024 (± 1249) kbps	G C
aioquic	G: 9433 (± 20) kbps C	G: 9447 (± 9) kbps C: 5795 (± 166) kbps	G C	G C	G: 9248 (± 61) kbps C: 8910 (± 84) kbps	G: 9489 (± 0) kbps C	G C	G: 9315 (± 13) kbps C: 5681 (± 171) kbps
neqo	G: 8319 (± 17) kbps C	G: 8093 (± 37) kbps C	G C	G: 5603 (± 593) kbps C	G C	G: 7530 (± 22) kbps C	G: 8037 (± 73) kbps C	G: 7698 (± 46) kbps C

Figure 2: The QUIC interop runner web interface [2]. Several tests are run for each client-server combination; tests are indicated by their letter symbols in each cell. In the Interop table, test outcomes can be Success, Unsupported, or Failure, as shown in the top, middle, or bottom row within each cell. The Measurements use a network bandwidth of 10Mbps, as described in Section 2.3. Results are reported as measured averages (with the standard deviation), and empty results represent failed runs. Log files from client and server generated during the test run, as well as log files and packet captures recorded by the network, are linked from each test case.

The QuicInteropRunner (QIR) is our attempt at overcoming these limitations in building performant and robust QUIC implementations [1]. Figure 1 shows the output of a single QIR run between three QUIC implementations for a selection of tests. Figure 2 shows the web output of a run between the ten implementations for our entire suite of interoperability and measurement tests.

QIR automates QUIC interoperability testing by running a suite of test cases between containerized QUIC implementations. QIR is a framework in which QUIC clients and servers interact with each other over a network that simulates various network conditions using ns-3 [4]. Each test case is described in the framework and made known to the implementations at run time. The outcomes of the tests are verified by the QIR framework via validation of transferred objects and programmatic inspection of packet traces. QIR also makes performance measurements under different network conditions possible. Importantly, QIR can be run locally, making both on-demand and continuous interoperability testing possible. To our knowledge, QIR is the first automated interoperability testing framework for a network protocol.

QIR includes several major QUIC implementations. As of this writing, ten QUIC implementations (two of which implement only client functions) are included in QIR. Any implementer can include their implementation in QIR by building a compatible container image, making it publicly available, and adding it to the list of implementations [9].

2 QIR DESIGN

QIR's design came out of our experience with the limitations of manual QUIC interoperability testing. We first go through the design constraints that shaped QIR's design, followed by detailed descriptions of QIR's components.

Since most QUIC implementations are in user space, we decided to focus our efforts on supporting user-space implementations. Since these implementations could all be built on Linux, we also chose to limit ourselves to that one platform instead of trying to build for multiple platforms. We acknowledge that this restricts us from testing existing kernel implementations or user-space ones that cannot be built on Linux.

2.1 Design constraints

QIR's design constraints were gleaned from our experience with manual interoperability testing with QUIC, and were as follows:

- *No source code*: QUIC implementations are written in different programming languages, under a variety of licenses, and with vastly different build environments and requirements. Building all implementations from their source for regular testing could require a significant amount of time and other resources. Importantly, we could not assume that the source code for all QUIC implementations would be available. As a result, our test framework could not expect the source code to be available, and would preferably not build implementations from source.
- *Maintenance delegation*: Each implementer would maintain and update their own endpoint, and make it publicly available on their own schedule.
- *On-demand and continuous testing*: To enable interoperability testing as part of a typical development workflow, the test framework would need to allow for on-demand testing of any set of implementations. This could then be extended to continuous testing of any set of implementations.
- *Repeatable performance testing*: Performance testing of the different implementations would need to use carefully constructed network scenarios and would need to be repeatable for debugging purposes.

To meet these constraints, our key insight was to use containers as QIR's basic building block. Containers give implementers control over their binary images, enabling them to bundle all build- and runtime-dependencies into their own, independent environments, and allowing them to publish updated images on their own schedule. Since they are distributed as binaries, containers also allow closed-source implementations to participate in the framework. Finally, containers enable implementers to make interoperability testing part of their development workflow, where they could run tests against other implementations at will.

2.2 QIR components

As shown in figure 3, QIR is a test harness that uses three Docker containers [10]: a client container, a server container, and a network container. Docker Compose [11] is used to orchestrate the three containers. QUIC implementers publish endpoint containers running their implementations on DockerHub [12], and each container can be instantiated as a server or as a client depending on the implementation's role in a test (the role is provided as an environment variable).

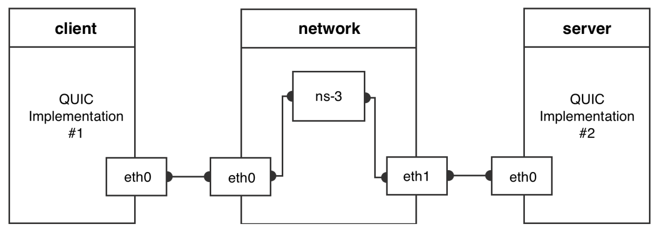

Figure 3: Network setup used in QIR tests. Boxes represent Docker containers [10] running a QUIC client, the network simulator, and a QUIC server. IP addresses and routes are configured such that packets between the client and the server have to pass through the network container, where ns-3 [4] is used to simulate different network conditions.

QUIC servers are expected to receive packets on UDP port 443 on a pre-specified IP address, configured as the address of the server container's virtual network interface. QUIC clients are expected to send requests to this pre-configured address.

The network interfaces of the server and client containers are on different IP subnets, to prevent the host operating system from forwarding packets directly between the two endpoint containers, and to force the packets to be forwarded through the network container instead. The network container has two network interfaces connecting to the server and client containers. All traffic between the endpoint containers passes through the network container, where various network conditions can be simulated.

2.3 Using ns-3 for network simulation

Within the network container, QIR uses the ns-3 network simulator [4], running in real-time simulation mode, to read and write packets from and to the two network interfaces, and to simulate a network topology between them. We chose ns-3 for the ease with which we could introduce new behaviors in the network simulation for various tests (see Section 3 for examples). We were also aware of its rich set of channel propagation and mobility models for different wireless and wired links, which we wanted to explore. Importantly, ns-3 allowed packets from the real world to be introduced into the simulated world and vice-versa. This would allow us to simulate any network condition or topology between the client and the server.

After much testing, we chose 10 Mbit/s as the bandwidth of the bottleneck link of the simulated network to ensure that a commodity laptop could run the QIR setup without using up all its compute power. This seems low, but we argue that it is adequate for our purposes[1]. For testing interoperability, any reasonable bandwidth would work. For testing performance, we would introduce competing traffic and network pathologies, and we would have expectations of how well an implementation ought to perform. We acknowledge that this setup does not allow for testing an implementation in a high-bandwidth environment. We are considering using a more compute-efficient simulator, such as Linux's netem, for such tests, but we leave that to future work.

In the simplest configuration, QIR uses ns-3 to simulate a fixed-bandwidth link with a finite queue size. Despite its simplicity, this setup exercises a fair bit of QUIC's machinery. Using a fixed-bandwidth link requires QUIC congestion controllers to determine the available bandwidth, typically by filling the queue at the bottleneck and reacting to any resulting packet loss. Other scenarios include inducing packet loss to test QUIC's loss recovery, both during the handshake and later in the connection; inducing packet corruption to test QUIC's ability to discard invalid packets; and temporary black-holing of the connection, to test QUIC's recovery from temporary outages.

3 QIR TESTS

A test case in QIR creates a *scenario* and observes the behavior of the QUIC endpoints, where a *scenario* is a specific network topology and behavior. For example, a simple test case could require a client to download a specific object from the server. This would mean that the client would have to successfully complete a QUIC handshake with the server, send a request for the object, process the server's response, and receive and store the object. More complex test cases require the client to establish a connection, receive a TLS Session Ticket [13] and transfer objects on a subsequently established 0-RTT connection.

In this section, we first describe QIR's workflow at run-time, followed by descriptions of a few exemplary tests. With the exception of the HTTP/3 test, all tests in QIR use a stripped-down HTTP/0.9 request-response format multiplexed onto QUIC streams for transferring objects. This allows for testing the QUIC protocol separately from HTTP/3. Furthermore, it allows QUIC implementations to participate in interoperability testing without requiring them to implement HTTP/3.

3.1 Scenarios

In QIR, a scenario represents a network topology and behavior, that we implement in C++ as a part of ns-3. QIR currently includes the following four scenarios, which are used in the tests it implements:

- *simple-p2p*: A simple point-to-point link with a specified bandwidth, delay, and queue size.
- *drop-rate*: A simple point-to-point link with a specified bandwidth, delay, queue size, and a configurable packet drop rate.
- *corrupt-rate*: A simple point-to-point link with a specified bandwidth, delay, queue size, and a configurable rate at which packets are corrupted (a random byte in the first fifty bytes of the QUIC packet is corrupted).
- *blackhole*: A simple point-to-point link with a specified bandwidth, delay, queue size, and configurable periods of time during which the link is either forwarding or dropping packets.

[1]Our investigation showed inefficiencies within ns-3 to cause this limitation, and we believe that performance optimization work within ns-3 can help. We leave this for future work.

3.2 QIR Workflow

Figure 4 shows QIR's workflow for each test. QIR first generates objects to be transferred for the test. These objects are of random sizes and content, and they are made available in the server container via a mounted directory. The client is expected to download these objects and store them into a separate mounted directory. At the end of each test, this setup allows QIR to access and validate the number and content of the downloaded objects.

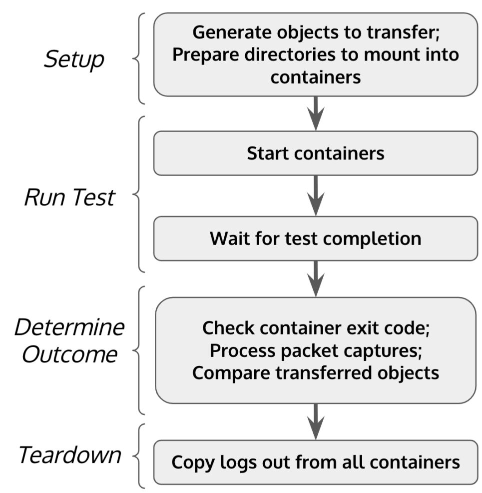

Figure 4: QIR workflow for running a test

In addition to these two directories, QIR sets up log directories to be used by the client and the server for recording their logs. Endpoints can record log files in their preferred logging format within the endpoint containers, and these are exported and made available by QIR after the test has completed. To facilitate analysis and debugging, the network container maintains various logs and detailed packet captures at both network interfaces. Some implementations export TLS secrets [14], which allows later decryption and analysis of these packet captures.

QIR then starts the containers up and waits until the test completes (or times out). QIR provides necessary configuration information, such as the name of the test and the names of the objects to download, to the endpoint containers using environment variables that are available within the containers. Test completion is indicated by at least one container shutting down. In most cases, this is the client shutting down after it has downloaded all objects.

QIR then checks container exit codes to determine the implementations’ self-reported outcomes of the test. If the containers claim to have completed the test successfully, QIR analyzes the packet captures and validates the downloaded objects against the ones that it generated.

3.3 Handshake Test

In this simple test, a client is expected to do the following:

(1) establish a QUIC connection with the server at the statically configured IP and port;
(2) request a single (small) file, the URL for which is specified in the REQUEST environment variable; for example, *https://server/xqsdfiuywerf*; and
(3) record the received object in a file with the same name, in the */downloads* directory in the client container.

The server is expected to accept incoming connections and respond to requests using objects located in the */www* directory in the server container.

In this test, QIR first generates and stores a 1KB file with a random name in the server’s */www* directory. Then, all three containers are brought up. The client is expected to finish the steps listed above and then exit, at which point QIR compares the original and the downloaded files, copies logs from the */logs* directory in all three containers, and copies off packet traces from the network container. If the validation of the downloaded file is successful, the test is a success. If the test concludes in any other manner, it is considered to have failed.

3.4 Retry Test

QUIC’s Retry mechanism is designed for a server to validate the client’s IP address prior to committing any state to the connection. This test extends the Handshake test to exercise the Retry mechanism. As in the Handshake test, the client requests an object from the server. The server is expected to validate the client’s IP address with a Retry packet prior to accepting the connection attempt.

In addition to the steps performed for the Handshake test, QIR needs to avoid misbehaving servers or clients from gaming this test. That is, QIR needs to verify that a Retry packet was in fact sent, and that the client’s post-Retry handshake attempt included information from the received Retry packet. To perform this verification, QIR programmatically examines packet traces to confirm that a retry did in fact occur. To examine packet traces, QIR uses pyshark [15], a Python wrapper around the Wireshark protocol analyzer [16].

3.5 Multiplexing Test

In the transport parameters sent during the handshake, QUIC endpoints declare the highest stream ID that the peer is allowed to open. This is used to limit the amount of resources dedicated to stream handling at any given time. If an endpoints wants to open more streams than this limit, it has to wait until the peer increases the stream ID limit. Typically, implementations increase the stream ID limit after previously used streams are closed and any resources associated with those stream has been freed.

The Multiplexing test extends the Handshake test to exercise QUIC’s stream multiplexing features. QIR generates 2000 small files (of 32 bytes each). Since servers commonly set a stream ID limit that is lower than 2000, clients will have to request a first batch of files, wait for the completion of the transfer and the increase of the stream ID limit from the server, and then issue requests for the next batch of files.

Due to the large number of files transferred, this test is particularly difficult during manual interoperability testing.

3.6 Performance Tests

The Throughput test is the first of QIR’s performance tests. This test is exactly the same as the Handshake test with the following modifications: the object transferred is large in size, and the test is repeated a number of times to show some statistical confidence in the results. Performance tests are not simply success or failure tests; the output of such a test is an expected value (in this case, throughput), with an error margin.

QIR allows building of more sophisticated performance tests. The second performance test reports on the server’s throughput when competing at a bottleneck link with a concurrent TCP flow. As shown in figure 5, we add two additional containers to the setup used so far, each running an iPerf [17] client or server, to generate TCP traffic. This TCP traffic uses the Cubic congestion controller and shares the ns-3-simulated bottleneck link with the QUIC traffic under observation. For perfect flow-fairness, TCP and QUIC are expected to each use half of the bottleneck link’s capacity. The interop runner computes the throughput of each flow from analyzing packet captures after completion of the test.

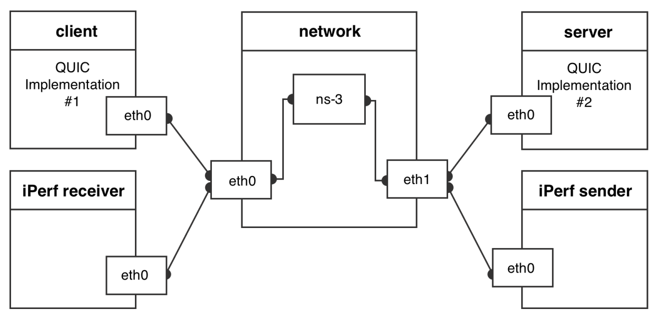

Figure 5: Network Setup for the cross-traffic test. In addition to the containers in Figure 3, this setup has two additional containers running an iPerf sender and an iPerf receiver to generate TCP traffic. TCP traffic competes with the QUIC traffic for bandwidth of the bottleneck link.

These are just four of the fourteen tests that are currently part of the QIR test suite, with many more tests under development. Scaling the number of interoperability tests is one of QIR’s key benefits, and we expect that increasing this number will accelerate the development and maturing of all QUIC implementations.

4 RELATED WORK

While there has been a lot of work in network simulation and emulation, there is very little work on interoperability frameworks. The closest related work to ours is QUIC Tracker [18]. This project uses a custom-built QUIC client to run tests against public QUIC servers, to test their compliance to the QUIC specification. Similarly, the

BoringSSL TLS implementation includes a separate endpoint that is used for running compliance tests against TLS endpoints [19]. These are both examples of compliance testing. QIR does not implement a QUIC endpoint, but provides a framework for interoperability testing across all participating implementations.

5 CONCLUSIONS AND FUTURE WORK

While interoperability testing has been one of the hallmarks of open standards and protocol development, the process itself remains woefully inadequate and limited. Using simple container orchestration and network simulation, QIR makes it possible to meet the constraints of implementers while automating this process for on-demand, continuous, and repeatable interoperability and performance testing.

QIR has considerably reduced the amount of time that implementers need to spend on setting up and running interoperability tests. This is now a fixed amount of manual effort for each implementer, irrespective of the number of other implementations. QIR is now supported by ten QUIC implementations, with some more forthcoming, and it has already become a part of the development workflow of several implementers.

We are planning on expanding the framework presented here to include more tests. Test cases of interest are:

- A high bandwidth test, testing QUIC flows at bandwidths up to 100 Mbit/s.
- A handshake latency test, measuring the time to handshake completion at various packet loss rates, averaged over multiple runs.
- A test case simulating a NAT rebinding, verifying that servers are able to handle this situation gracefully.
- A test case in which the client performs a connection migration from one network address to the other.
- A more sophisticated ns-3 scenario simulating characteristics of WiFi and cellular networks.
- A performance comparison between HTTP/2 and HTTP/3, using realistic web pages.

In addition to enabling implementers to run interoperability tests as frequently as they like, QIR makes it possible to run continuous tests across all implementations. We now run the entire suite of tests on a dedicated server to generate an interoperability matrix as frequently as once per day; see [2]. Using a web interface, this matrix shows the results of a complete interoperability test, with access provided to logs and packet captures for debugging purposes.

While QIR was developed for testing QUIC implementations, the central ideas, components, and even code can be re-purposed for testing other protocols as well. We hope to see this happen in the future.

REFERENCES

[1] *QUIC Interop Runner.* https://github.com/marten-seemann/quic-interop-runner.
[2] *QUIC Interop Runner Web Interface.* https://interop.seemann.io.
[3] J. Iyengar and M. Thompson. QUIC: A UDP-Based Multiplexed and Secure Transport. February 2020. https://tools.ietf.org/html/draft-ietf-quic-transport-27.
[4] *The ns-3 Network Simulator.* https://www.nsnam.org/.
[5] S. Bradner. RFC 2026: The Internet Standards Process – Revision 3. October 1996.
[6] R. Stewart. RFC 4960: Stream Control Transmission Protocol. September 2007.
[7] *QUIC Interop Wiki.* https://github.com/quicwg/base-drafts/wiki/17th-Implementation-Draft.
[8] *QUIC Working Group.* https://quicwg.org.
[9] *QUIC Interop Runner Documentation.* https://github.com/marten-seemann/quic-interop-runner/.
[10] *Docker.* https://docker.com.
[11] *Docker Compose.* https://docs.docker.com/compose/.
[12] *Docker Hub.* https://hub.docker.com.
[13] E. Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. August 2018.
[14] *NSS Key Log Format.* https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format.
[15] *pyshark: Python wrapper for tshark.* https://kiminewt.github.io/pyshark/.
[16] *Wireshark protocol analyzer.* https://www.wireshark.org/.
[17] *iPerf - The ultimate speed test tool for TCP, UDP and SCTP.* https://iperf.fr/.
[18] M. Piraux, Q. De Coninck, and O. Bonaventure. Observing the Evolution of QUIC Implementations. August 2018.
[19] *Boring SSL test runner.* https://boringssl.googlesource.com/boringssl/+/refs/heads/master/ssl/test/runner/.