Vegvisir: A testing framework for HTTP/3 media streaming

Joris Herbots
Hasselt University – tUL – EDM
Diepenbeek, Belgium
joris.herbots@uhasselt.be

Mike Vandersanden
Hasselt University – tUL – EDM
Diepenbeek, Belgium
mike.vandersanden@uhasselt.be

Peter Quax
Hasselt University – tUL – Flanders Make - EDM
Diepenbeek, Belgium
peter.quax@uhasselt.be

Wim Lamotte
Hasselt University – tUL - EDM
Diepenbeek, Belgium
wim.lamotte@uhasselt.be

ABSTRACT

Assessing media streaming performance traditionally requires the presence of reproducible network conditions and a heterogeneous dataset of media materials. Setting up such experiments represents a complex challenge in itself. This challenge becomes even more complex when we consider the new QUIC transport protocol, which has many tunable features, yet is difficult to analyze due to its inherent encrypted nature. In this paper, we introduce Vegvisir, which aims to solve these aforementioned challenges by providing an open-source automated testing framework for orchestrating media streaming experiments over HTTP/3. We describe how users can steer the behavior of Vegvisir through its configuration system. We provide a high-level overview of its inner workings and its broad applicability by describing two use cases: one covering sizeable experiments spanning multiple days and another covering HAS evaluation scenarios.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Application layer protocols**; • **Information systems** → **Multimedia streaming**.

KEYWORDS

QUIC, HTTP/3, MPEG-DASH, qlog, qlog-has, testing framework

ACM Reference Format:
Joris Herbots, Mike Vandersanden, Peter Quax, and Wim Lamotte. 2023. Vegvisir: A testing framework for HTTP/3 media streaming. In *Proceedings of the 14th ACM Multimedia Systems Conference (MMSys '23), June 7–10, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587819.3592550

1 INTRODUCTION

Facilitating media streaming over the Internet has traditionally been accomplished by using RTMP, RTP/RTSP and the newer WebRTC if real-time performance was critical. When scalability became the main focus, the combination of HTTP and TCP was to be preferred, which led to HTTP Adaptive Streaming (HAS) protocols, such as MPEG-DASH and HLS. However, HAS over TCP is not a perfect solution and suffers from flaws inherited from the TCP-HTTP setting. Most HAS traffic is bursty by nature – due to the stream segmentation mechanism and playout buffer management – typically referred to as the ON-OFF cycle, which leads to poor bandwidth utilization and fairness issues [4–6]. For this and other reasons, the media streaming community is looking at QUIC as a possible replacement for TCP.

With the first stable version of QUIC having been standardized [14], we are at a key moment in time where significant innovation within the media streaming landscape can take place. The recently established Media over QUIC (MoQ) group – comprising key players such as Twitch, Cisco, Meta and several large CDNs – is working on a low-latency media delivery solution for ingesting and sharing media over QUIC [10]. Meanwhile, extensions to the QUIC protocol are being developed and deployed at scale, such as the Deadline-aware-Transport Protocol [8], the Unreliable Datagram Extension [22], the WebTransport API [9, 28], HTTP Datagrams and the Capsule Protocol [26]. Besides exploring novel ways to employ QUIC, others are already deploying existing media streaming implementations over QUIC without specific adaptations. Many of the major online platforms like Youtube and Facebook [25] use QUIC and HTTP/3 [7] – the successor to HTTP/2 to be used over QUIC – as their underlying protocols of choice for HAS. Undeniably, QUIC is here to stay and change the future landscape of Internet applications [25].

However, while media streaming over QUIC presents new opportunities, it also introduces new challenges. As Marx et al. posit, considerable heterogeneity exists between the available QUIC protocol implementations, resulting in measurable differences in performance [15]. Presently, 25 open-source QUIC implementations are available, most of them supporting HTTP3 [1]. Out of many factors influencing application performance, the specific choice of congestion control and flow control approaches are prime candidates to optimize. Regarding evaluating media performance over QUIC, we recognize that our experiments need to embrace this heterogeneity and look at the differences between existing QUIC implementations.

Profiling or debugging media streaming applications can be time-consuming for developers. The time required to establish a reproducible streaming experiment is composed of more factors than just manual labor for the analysis. A single test typically covers media playback of live media or some dataset, ranging from a few minutes to several hours. Considering the fact that a complete experiment, consisting of multiple tests, can span up to several days. Setting up

MMSys '23, June 7–10, 2023, Vancouver, BC, Canada

such an environment for each experiment is prone to human error as time progresses, directly impacting the experiments' results and reproducibility. Furthermore, analyzing media streaming over ideal network conditions tells very little about its behavior in real-world conditions. Establishing representative and reproducible network conditions to be used in these experiments can be cumbersome. Combine this with the QUIC implementation heterogeneity, and we achieve vast experiment sizes that preclude manual testing as a valid option.

To address the challenges mentioned in the paragraph above, we present Vegvisir [13]. Vegvisir is an automated testing framework that orchestrates client-server-based media streaming experiments over simulated network conditions. Because the media streaming domain is broad and we cannot inventory all potential use cases in advance, Vegvisir is designed to be broadly applicable. The version released alongside this paper is tested and specialized towards media streaming over HTTP/3. Nonetheless, due to careful design choices, Vegvisir does not enforce this use case in any way, also making it applicable for future use cases related to e.g. protocols proposed by the MoQ group and QUIC extensions in development. Our proposed framework makes it easy to define and set up experiments involving multiple QUIC implementations and different network scenarios. Vegvisir collects all output produced by an experiment in a human-interpretable structured folder format. Additionally, Vegvisir provides an extensible mechanism to allow users to program custom behavior to analyze the collected output. To help with adoption, our framework is compatible with existing Docker images from the QuicInteropRunner (QIR) project [27] (see Section 3), so out-of-the-box, Vegvisir always provides up-to-date QUIC-HTTP/3 client and server configurations. At the time of writing, this applies to 15 of out the 25 existing QUIC implementations [1, 3].

2 DESIGN GOALS

The concrete needs we identified when designing the proposed framework were transformed into the following five goals. We will mark the remainder of this paper with ①, ②, ③, ④ and ⑤ whenever they describe an element that adheres to that design goal.

① **Repeatable and shareable experiments:** Experiments should be performed under controlled conditions. Users wishing to replicate the results of shared experiments should be able to do so without a deep understanding of the experiment configuration.

② **Scalability:** The scalability goal covers two domains. Firstly, the framework should provide a high-level interface for defining sizeable experiments without verbosely writing down every single composing test. Users should be able to define their experiments through the use of parameters. Secondly, experiments spanning multiple hours or even days should be able to complete without human intervention.

③ **Broad applicability and future proof:** The primary focus of the framework is to facilitate experiments in any way or form and gather their output. The framework should be designed to integrate current and future QUIC-based protocols without significant changes to its underlying architecture.

④ **Extensible behavior:** The framework should provide hooks to allow users to program its behavior and go beyond its intended scope. Users should be able to extend the behavior such that the framework serves as a highly-specialized testing framework (i.e., a test framework that knows what output the experiments produce with the capability of analyzing it).

⑤ **Human interpretable results:** The framework should make experiment output easy to navigate so that configurations and results do not require intimate knowledge about the framework.

3 RELATED WORK

Mahimahi [20] is a highly-specialized testing framework for HTTP client-server interactions. By capturing HTTP-based interactions and storing them in a structured format, it can later replay these interactions using simulated network conditions. This enables its users to test the performance of HTTP-based applications under controlled conditions. Mahimahi fits our predefined goals ① and ②. The biggest downside, however, is that Mahimahi's current implementation lacks QUIC and HTTP/3 support. Constantin et al. tried using Mahimahi to analyze the influence of resource prioritization on HTTP/3 Head Of Line blocking. However, they ran into problems, resulting in them having to set up their own system, which was limited to a single QUIC implementation [24].

The Speeder framework [19] aims to evaluate sizeable input sets across multiple software implementations over various network conditions. However, while similar to our situation ① ②, Speeder focuses on amassing predefined web performance measurements, thus making it a highly-specialized testing framework. The authors have chosen to automate its experiments and facilitate client-server communication through TC NETEM's network emulation software. Nonetheless, due to the lack of flexibility in their setup, they have decided not to open-source their framework.

On the other hand, QuicInteropRunner (QIR) [27] is a recent open-sourced interoperability testing framework for QUIC implementations. The creators of QIR recognized that for QUIC development to advance more efficiently, testing the interaction of QUIC implementations against each other was paramount. Testing is done by running a batch of test cases to test protocol specifics between all participating QUIC implementation combinations. The framework then decides whether a test succeeded using a simple pass/fail mechanism to indicate the interoperability status between two implementations. Even though QIR is a highly-specialized testing framework that does not meet our predefined goals, we have decided to use it as the foundation for Vegvisir. QIR's core concept comprises QUIC-based client-server communication over a simulated network, which is an opportune starting point for our design goals ①, ②, ③ and ⑤. A secondary motivation for expanding upon QIR lies in how QIR delegates its maintenance to the QUIC implementers participating in interoperability testing. Internally, QIR's architecture makes use of Docker. Each implementer provides a Docker image via the Docker Hub and updates it according to their schedule. Most of these support an HTTP3 web server setup, fully equipped to serve requests by QUIC-compatible clients. In other words, Vegvisir is out-of-the-box compatible with QIR's Docker images and thus provides support for 15 out of the 25 QUIC-HTTP/3 implementations. By expanding upon QIR, we profit from already existing community efforts which in turn aids with the adoption of Vegvisir.

goDASH [21, 23] is an open-source headless DASH player written in golang, with an emphasis on being a validation framework

for HAS algorithms and QoE models. Under the hood, goDASH supports TCP- and QUIC-based streaming, comes equipped with numerous HAS adaptation algorithms and is designed to be easily customized to enable rapid testing of new HAS logic. When combined with goDASHBed, a testbed framework that emulates network behavior using Mininet, users can set up scalable experiments mimicking real-life scenarios. Although similar to Vegvisir, goDASH with goDASHBed is considered a highly-specialized test framework. Both are additionally, by design, limited to only work with the quic-go QUIC implementation. These downsides are problematic because they strongly hinder our design goal ③.

4 FRAMEWORK COMPONENTS

This section describes the different components that make up Vegvisir. We explain the configuration system, extensibility through customization, the engine for running experiments and how it enables reproducibility.

4.1 Setting up Experiments

Automated testing frameworks are typically associated with a single repository or configuration file containing test cases or conditions. Introducing changes to the involved software requires redefining test cases. Even worse, duplicate test definitions are required if multiple versions of the same software are used. This approach, however, does not meet our predetermined design goals ① and ②.

Therefore, we have designed Vegvisir to operate on two types of configurations, the `implementation` and `experiment` configurations, respectively. These define what software is available and how it is used. Their composition is explained in more detail below. We have chosen JSON ① as the data format because it is easy to interpret, edit and share with others. The experiment execution engine (explained in Section 4.3) operates on a three-tuple of entities that Vegvisir refers to as the server, the shaper and the client. The client and the server represent network endpoints and assume their prototypical roles. The shaper defines the characteristics of the network over which the client and server communicate.

The rationale behind splitting up the `implementation` configuration from the `experiment` configuration is to achieve loose coupling of software and experiments ① ②. For example, multiple implementation configurations can define the same client `chrome` entry, each addressing a different version. This way, the experiment configuration requires no changes – provided the implementation entry works the same way – so experiments can easily swap implementation versions or behavior. Section 4.4 further explains how Vegvisir utilizes this mechanism for sharing by freezing implementation configurations. Furthermore, splitting up these two configurations allows multiple experiments to use the same implementations without explicit duplication. These abilities drastically reduce the amount of configuration work while providing increased flexibility.

The `implementation` configuration defines the available server, shaper and client software to the experiment execution engine. To satisfy constraints ① and ③, we have chosen to represent these entities with the Docker container technology. The benefit of spinning up Docker containers is that entities are self-contained units that are easy to exchange through existing platforms such as the Docker hub or via traditional file sharing. However, we also recognize that container technology is not always a good fit for client entities. For example, wrapping a GUI-based client – such as the `chrome` browser entry in Listing 1 – in a container does not achieve the same results as executing it natively on the host machine. With that in mind, we have designed Vegvisir to allow client entries to be represented by both Docker images and locally compiled or installed binaries.

```
{
 "clients": {
  "chrome": {
   "command": "chrome␣--origin-to-force-quic-on=!{ORIGIN}:!{
        ↪ ORIGIN_PORT}␣!{REQUEST_URL}",
   "parameters": {"REQUEST_URL": true},
   "construct": [{"root_required": false, "command": "python␣./
        ↪ util/chrome-set-downloads-folder.py␣~/.config/google-
        ↪ chrome/Default/Preferences␣\"!{CLIENT_DOWNLOAD_DIR}\""}
        ↪ ]
  }
 },
 "shapers": {
  "tc-netem": {
   "image": "tc-netem",
   "scenarios": {
    "simple": {"command": "simple␣!{LATENCY}␣!{THROUGHPUT}", "
        ↪ parameters": ["THROUGHPUT", "LATENCY"]}
   }
  }
 },
 "servers": {"aioquic": {"image": "aiortc/aioquic-qns"}}
}
```

Listing 1: An `implementation` configuration showcasing one entity for each category. The client entity, `chrome`, defines how to use start the chrome browser via the `command` key together with its parameter set. The `tc-netem` shaper entry contains one scenario for introducing latency and rate limiting. The `aioquic` server entry is one of QIR's interoperability images that works out-of-the-box with Vegvisir.

Listing 1 showcases a simplified `implementation` configuration example. An `implementation` configuration always contains three root dictionaries that define the `clients`, `shapers` and `servers` entities. A name uniquely identifies each entity entry (e.g., `chrome`, `tc-netem` and `aioquic`[1] in Listing 1), which contains a dictionary that provides information on how to run it to the experiment execution engine. Server, shaper and client entities reference Docker Hub images or locally built images with the `image` key. Client entries can additionally reference a local binary via the `command` key. The value of this key consists of a standard command-line interface (CLI) command. Docker images can contain scripts that manage the entity setup and dismantling. To allow for similar behavior with `command`-based clients, the optional `construct` and `destruct` keys can be populated with a list of commands to be executed before and after the client command. The `chrome` client entity in Listing 1 showcases an example of one such construct command to set the download folder to one provided by Vegvisir.

The vision for server and client entities is to reference a single software unit. For example, a client entry could represent a browser with a specific set of options. Shaper entities, on the contrary, are less atomic and define a collection of network scenarios via the `scenarios` key. Similar to entity entries, a name uniquely identifies scenario entries within a shaper entry.

We have incorporated a parametric system into the `implementation` configuration to aid with ② and ③. Server and client entities, as well as shaper scenarios, can define these via the `parameters` key. Parameter entries for servers and clients can specify whether they are required or optional through an associated boolean value (e.g.,

[1] `aioquic` is one of the existing open-source QUIC-HTTP/3 implementations.

the REQUEST_URL parameter in the chrome client of Listing 1 is required). Shaper scenario parameters are always required. Within the implementation configuration, command-based client entities and shaper scenario commands use these parameters via the syntax !{NAME}. Our parametric system is powerful enough that parameter-provided values can, in turn, also contain references to other parameters (e.g., the parameter value provided to REQUEST_URL in Listing 2 references another parameter called ORIGIN). Vegvisir additionally provides system-specific parameters, such as logging paths, hostnames and ports to more dynamically describe implementations and experiments. For example, in Listing 1, Vegvisir will populate the ORIGIN and ORIGIN_PORT parameters referenced by chrome's command with the correct hostname and port during an experiment.

```
{
 "clients": [
  {
   "name": "chrome",
   "arguments": {"REQUEST_URL": "https://!{ORIGIN}/video.mp4"}
  }
 ],
 "shapers": [{"name": "tc-netem", "scenario": "simple_15_10"}],
 "servers": [{"name": "aioquic"}],
 "environment": {
  "name": "webserver-basic",
  "sensors": [{"name": "timeout", "timeout": 30}]
 },
 "settings": {"label": "paper_example"}
}
```

Listing 2: An experiment configuration showcasing a setup using the chrome, tc-netem and aioquic entities from the implementation configuration in Listing 1.

How an experiment should behave is finally defined in the experiment configuration. Listing 2 shows a simple example of such a configuration. Similar to the implementation configuration, the experiment configuration contains three root keys: servers, shapers and clients. Each contains a list of at least one object describing the entity involved in the experiment. Entities present in the experiment that define parameters in the implementation configuration must provide arguments for them in the experiment configuration. Using the !{NAME} syntax, as mentioned earlier, the experiment can dynamically steer its entities ② ③. The experiment execution engine will create all possible combinations from the three provided sets of entities ②. One such combination constitutes a single test within the experiment. The total amount of tests is, therefore, equal to the number of entries of each set multiplied by each other.

The experiment configuration root also contains the environment and settings object. The environment name is linked directly to the environment system explained in Section 4.2 ④. It also specifies optional sensors that monitor client entities during the different tests of an experiment. A test can end in either of two ways. Typically, the client's Docker container or command, monitored by Vegvisir, would simply finish its execution (thus implicitly indicating the end of a test). Alternatively, configured sensors can detect a specific state, signaling the end of a test, resulting in Vegvisir terminating the test. Vegvisir presently includes two sensors. The timeout sensor is a simple timer mechanism that halts a test after a specified amount of seconds. The browser-file-watchdog sensor is explicitly designed for browser clients to signal the end of a test if one of the provided filenames appears in Vegvisir's test download folder. Due to their sandboxed nature, extracting application metrics from a browser context is complex, and no mechanism exists to close a browser from within a webpage. However, browsers can download files, and JavaScript can automate such behavior, which we can use to pass through gathered information and signal the end of the test.

4.2 Custom Behavior with Hooks and Sensors

Due to ③, knowing the output of an experiment in advance would limit Vegvisir's scope. This is why we decided to make Vegvisir's primary goal to gather output from experiments ③ and make them as interpretable as possible ⑤. A side-effect of ⑤ is that users can automate the output analysis with scripts. It is precisely this feature that makes a testing framework a highly-specialized testing framework. We optionally allow users to program this behavior through hooks to bridge this gap ④. By extending the BaseEnvironment available in the environments module within Vegvisir's codebase, users can populate the pre_run_hook and post_run_hook callbacks with custom behavior. Even though our codebase is written in Python, users wishing to use existing scripts or programs written in other languages can also opt to call these from the hook callbacks using Python's subprocess module. The benefit of using our hooks is that they receive all download and logging paths pertaining to a test. This eliminates the need to crawl through the output folders manually.

While we provide some generic sensors that cover many use cases, we recognize that they will not cover everything or be specific enough for other use cases due to ③. As such, similar to hooks, users can program custom sensors by extending the ABCSensor[2] available in the environments module. This allows for a powerful mechanism to govern experiments. For example, a browser-based HAS client could send decoding metrics to an HTTP endpoint set up by a custom sensor. If this example sensor detects dropped frames, it can decide to preliminarily end the test and output its reason for doing so in the output log collected by Vegvisir. In cases where an experiment needs to process many tests that take a long time, such a mechanism is very beneficial.

4.3 Experiment Execution Engine

Since Vegvisir can host experiments that can span from a few minutes to even days, we have designed the experiment execution engine to dry-run the provided configurations before performing the actual tests. The dry-run phase will primarily catch missing parameters for entities involved in the experiment and check if the provided sensor configurations can be loaded. It is important to note that while – similar to a compiler – we can catch missing parameters, there is no system in place to check whether the provided values are in fact correct. If, for example, an experiment configuration provides a wrong URL in a client configuration, Vegvisir will quietly perform this test and collect its results. It is up to the implementer to confirm the validity of provided values.

After passing the dry-run phase, Vegvisir commences the experiment by sequentially running all test combinations as computed from the experiment configuration. Each test goes through 11 steps. Steps prefixed with a dagger (†) only take place when the respective test involves a client using a CLI command. A double-dagger (‡) indicates that the step happens if the specified environment hook behavior is populated.

[2]ABC stands for Abstract Base Class

(1) **Log collection setup:** Vegvisir creates the necessary logging structures to capture all output produced by the entities involved with the test. Server, shaper and client Docker containers directly mount their respective logging folder to which they can output anything they produce. For clients using CLI commands, Vegvisir provides system parameters containing the current logging paths[3].
(2) **Parameter setup:** The parameters for each entity are associated with the provided arguments from the experiment configuration. For Docker containers, the parameters are accessible via environment variables within the container. For client CLI commands, the parameters get directly substituted for the respective arguments.
(3) **‡Activate environment pre-hook:** The environment pre-hook is called and provided with the current test's log paths.
(4) **Start server and shaper:** Vegvisir boots the server and shaper configuration beforehand. The experiment execution engine is now at a branching point where clients using CLI commands require extra steps.
(5) **†Routing configuration:** Vegvisir reconfigures system routes, so traffic from a command-based client to the server passes through the shaper container. If all involved entities make use of Docker, Docker Compose automatically takes care of routing.
(6) **Gather system information:** To aid with ① and ⑤, Vegvisir will, during this step, execute a number of CLI commands to gather the following information: all kernel parameters and their values, current IP and route information (including that of the server and shaper container), Docker version and Docker compose version. Since certain system settings can influence the test, logging these for post-experiment analysis are essential to correctly replicate behavior.
(7) **†Sequentially perform client `construct` commands**
(8) **Start client and sensors:** Configured sensors are activated and monitor the client output as necessary. Manual termination of the test is also possible by sending an abort to the terminal.
(9) **Teardown:** After a client exit or sensor trigger, Vegvisir will halt all entities.
(10) **†Sequentially perform client `destruct` commands**
(11) **‡Activate environment post-hooks:** The post-hook is called and provided with all log paths involved with the current test.

4.4 Sharing Experiments and Results

Repeatability and reproducibility are essential aspects when we apply Vegvisir to research-oriented experiments. Therefore Vegvisir provides a mechanism to freeze experiments ①. Freezing entails archiving Docker image versions specified in a provided `implementations` file. Freezing clients using CLI commands is not supported because there is no unified mechanism to support this in an easy-to-use way. We recommend that users manually create an archive of the locally compiled or installed program or its codebase as an alternative.

The output of the freezing process is an archive containing all Docker images referenced by the `implementation` configuration and a new `implementation` configuration in which the original `image` references are replaced with the frozen ones. The original and frozen `implementation` configurations provide the same functionality and are, at this point, provided that no changes were made to the current Docker images, completely interchangeable. More importantly, users can exchange these archives to replicate the experiments on other machines.

The logging output collected by the experiment execution engine can be interpreted without having any knowledge about Vegvisir. Output is organized using a hierarchical folder structure with readable names based on the unique names provided in the implementation configurations ① ⑤. Vegvisir automatically copies the configurations to the logging folders for each experiment. This way, users can reproduce the setup at a later point in time. Additionally, should the configurations irrecoverably change or get lost, no reverse engineering from the logs is required.

5 USE CASES

The following subsections illustrate two media streaming-related use cases for which Vegvisir has proven to be an invaluable research tool. The first use case comprises Vegvisir being deployed to efficiently discover transport-related issues in HAS streaming. Our second use case comprised finding the most suitable 3D mesh compression parameters for the Draco gLTF extension.

5.1 Evaluating HAS over HTTP/3

Evaluating browser-based HAS performance over HTTP/3 was our initial use case for creating Vegvisir. Having a way to manage an array of HTTP3 server implementations that have many tunable parameters, e.g., different congestion control algorithms, has allowed us to efficiently and rapidly prototype our ideas.

To evaluate HAS performance, we must compare it to a baseline set of measurements. To facilitate this process, we can depend on the qlog [17] unified logging format, which most QUIC implementations support. Tools like qvis[4] allow us to examine these qlogs using interactive visualizations. However, qlog only defines logging events for QUIC and HTTP/3 [16, 18]. No official event definitions for HAS currently exist. We therefore include our early work on qlog event definitions for HAS named `qlog-has` [11]. The idea behind `qlog-has` is to have a unified way to represent streaming metrics and events so that they can be used for performance evaluation and debugging. Additionally, we include a `dash.js` wrapper [12] that produces `qlog-has` logs for MPEG-DASH streams which we used during this experiment.

In this use case, we look at streaming diverse video datasets (e.g., constant bitrate encoding vs. variable bitrate encoding) over multiple simulated network environments that trigger adaptive bitrate algorithm logic using existing HTTP/3 server implementations. Vegvisir's experiment configuration allows us to define the above quickly and efficiently.

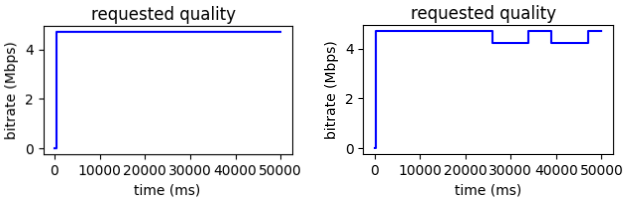
For example, in one of our experiments, we defined QIR's 15 HTTP/3 server implementations using their default setup, 3 simple constant throughput network scenarios that fell between bitrate ladder steps and 2 Firefox client configurations for two video datasets of 3 minutes each. Vegvisir produced a total of 90 test cases (15

[3]For people wishing to use Google Chrome and/or Firefox, we provide utility scripts to set the download paths to the folder created by Vegvisir.

[4]https://qvis.quictools.info/

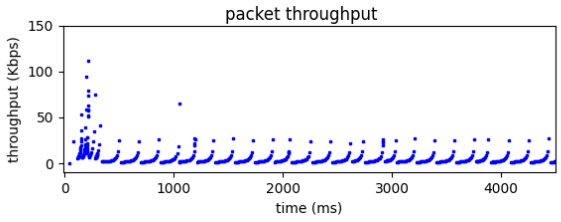
servers × 3 shapers × 2 clients) spanning a total time of approximately 4.5 running hours. The server implementations and Firefox produced qlogs containing QUIC and HTTP/3 metrics. Our `dash.js` wrapper produced qlog-has metrics for each test case.

Figure 1 illustrates the requested quality expressed in bitrate for two test cases using the same network conditions and client configuration, which we retrieved from our qlog-has output. Even though the conditions were the same, we observed differences in the requested qualities. Because Vegvisir captures QUIC and HTTP/3 server metrics with qlog, we can immediately examine this behavior by looking at the respective server output. Figure 2 illustrates the instantaneous throughput of the packets sent by the `quic-go` and `ngtcp2` implementations. A difference in burstiness and pacing is observable between the two.

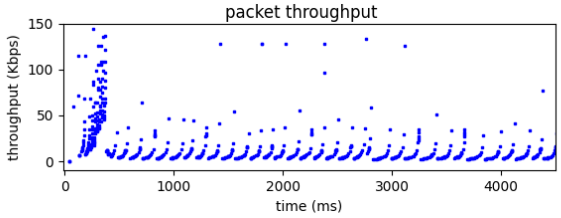


(a) `ngtcp2` HTTP/3 server **(b) `quic-go` HTTP/3 server**

Figure 1: Quality changes during a HAS session.



(a) quic-go



(b) ngtcp2

Figure 2: Packets of a HAS session, points represent sent-time and measured throughput.

5.2 Finding Suitable Encoding Parameters for Virtual Environment Streaming

In this use case, we focus less on analyzing the heterogeneity between existing QUIC-HTTP/3 implementations for HAS but instead showcase Vegvisir's potential in managing sizeable experiments. Our project aimed to find the most suitable Draco[2] compression parameters for gLTF-encoded 3D-meshes to enable virtual environment streaming.

By default, gLTF does not apply any compression on its 3D-meshes. This puts significant strain on the network to transfer this kind of data. Extensions like Draco allow compressing the vertex attributes, normals, colors and texture coordinates, resulting in faster transmission rates. Choosing the correct glTF encoding parameters for objects of the virtual scene is crucial. Different combinations of various encoding techniques for meshes and textures have varying performances based on the quality of the network. The highest compression, for example, might not result in the best time-to-interactive because of the overhead decompression introduces.

To find the best-fitting encoding parameters for our use case, we created an experiment configuration containing multiple HTTP/3 servers and network conditions. All possible encoding parameters were applied to our dataset and inserted as arguments for our browser client to utilize during the experiment. This resulted in a total of 8232 test combinations which Vegvisir robustly ran during a period lasting 48 hours. All tests were completed successfully without human interaction. The experiment provided us with enough log data to analyze and summarize quickly. No compression resulted in the worst time-to-interactive, while level 5 Draco compression produced the best time-to-interactive across different network scenarios. Vegvisir's involvement in this elaborate experiment saved many person-hours and ensured the project progressed with substantiated choices.

6 CONCLUSION AND FUTURE WORK

In this paper, we have introduced Vegvisir [13], a Python-based automated testing framework to aid researchers and developers in addressing the challenges posed by setting up sizeable experiments using the QUIC and HTTP/3 protocols. We hope this framework will be helpful for others looking into automating their research and development, whether QUIC-related or for other purposes.

Since Vegvisir is already rooted in our research and projects, development shall continue steadily at its GitHub repository [13]. We encourage everyone to play around with it, use it in their research, report potential issues, and contribute features.

As for future work, we plan to extend the experiment configuration to allow simultaneous multi-client setups. While it is already possible to do this by starting a script that initiates multiple clients, we want to create a more robust mechanism that allows a user to describe the client arrival process. A second feature we wish to implement is a playground mode that allows users to spin up a client or server with a shaper indefinitely to aid in developing and testing the other endpoint.

ACKNOWLEDGMENTS

Joris Herbots (BOF19OWB07) and Mike Vandersanden (BOF22OWB17) are Ph.D. candidates at Hasselt University, supported by the Special Research Fund (BOF). Special thanks go to Olaf Van Bylen for his work on exploring front-ends for Vegvisir. The research leading to the results in Section 5.2 has received funding from the European Union's Horizon Europe Programme under grant agreement 101070072, MAX-R (Mixed Augmented and eXtended Reality media pipeline).

REFERENCES

[1] 2023. Active QUIC implementations. https://github.com/quicwg/base-drafts/wiki/Implementations.
[2] 2023. Draco. https://google.github.io/draco/

[3] 2023. QUIC Interop Runner status matrix. https://interop.seemann.io/.

[4] Saamer Akhshabi, Lakshmi Anantakrishnan, Ali C. Begen, and Constantine Dovrolis. 2012. What Happens When HTTP Adaptive Streaming Players Compete for Bandwidth?. In *Proceedings of the 22nd International Workshop on Network and Operating System Support for Digital Audio and Video* (Toronto, Ontario, Canada) *(NOSSDAV '12)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/2229087.2229092

[5] Sangwook Bae, Dahyun Jang, and KyoungSoo Park. 2015. Why Is HTTP Adaptive Streaming So Hard?. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 12, 8 pages. https://doi.org/10.1145/2797022.2797031

[6] Divyashri Bhat, Amr Rizk, and Michael Zink. 2017. Not so QUIC: A Performance Study of DASH over QUIC. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video* (Taipei, Taiwan) *(NOSSDAV'17)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3083165.3083175

[7] Mike Bishop. 2022. HTTP/3. RFC 9114. https://doi.org/10.17487/RFC9114

[8] Yong Cui, Chuan Ma, Hang Shi, Kai Zheng, and Wei Wang. 2022. *Deadline-aware Transport Protocol.* Internet-Draft draft-shi-quic-dtp-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-shi-quic-dtp/06/ Work in Progress.

[9] Alan Frindell, Eric Kinnear, and Victor Vasiliev. 2023. *WebTransport over HTTP/3.* Internet-Draft draft-ietf-webtrans-http3-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-webtrans-http3/04/ Work in Progress.

[10] James Gruessing and Spencer Dawkins. 2022. *Media Over QUIC - Use Cases and Requirements for Media Transport Protocol Design.* Internet-Draft draft-gruessing-moq-requirements-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-gruessing-moq-requirements/03/ Work in Progress.

[11] Joris Herbots and Mike Vandersanden. 2023. HTTP Adaptive Streaming event definitions for qlog. https://github.com/JorisHerbots/qlog-has.

[12] Joris Herbots and Mike Vandersanden. 2023. qlog-has wrapper for dash.js. https://github.com/JorisHerbots/dashjs-qlog-has.

[13] Joris Herbots and Mike Vandersanden. 2023. Vegvisir. https://github.com/JorisHerbots/vegvisir.

[14] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[15] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 14–20. https://doi.org/10.1145/3405796.3405828

[16] Robin Marx, Luca Niccolini, Marten Seemann, and Lucas Pardue. 2022. *HTTP/3 and QPACK qlog event definitions.* Internet-Draft draft-ietf-quic-qlog-h3-events-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-h3-events/03/ Work in Progress.

[17] Robin Marx, Luca Niccolini, Marten Seemann, and Lucas Pardue. 2022. *Main logging schema for qlog.* Internet-Draft draft-ietf-quic-qlog-main-schema-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/04/ Work in Progress.

[18] Robin Marx, Luca Niccolini, Marten Seemann, and Lucas Pardue. 2022. *QUIC event definitions for qlog.* Internet-Draft draft-ietf-quic-qlog-quic-events-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-quic-events/03/ Work in Progress.

[19] Robin Marx, Peter Quax, Axel Faes, and Wim Lamotte. 2017. Concatenation, Embedding and Sharding: Do HTTP/1 Performance Best Practices Make Sense in HTTP/2?.. In *WEBIST*. 160–173.

[20] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 417–429.

[21] John O'Sullivan, Darijo Raca, and Jason J. Quinlan. 2020. Godash 2.0 - The Next Evolution of HAS Evaluation. In *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 185–187. https://doi.org/10.1109/WoWMoM49955.2020.00043

[22] Tommy Pauly, Eric Kinnear, and David Schinazi. 2022. An Unreliable Datagram Extension to QUIC. RFC 9221. https://doi.org/10.17487/RFC9221

[23] Darijo Raca, Maëlle Manifacier, and Jason J. Quinlan. 2020. goDASH — GO Accelerated HAS Framework for Rapid Prototyping. In *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*. 1–4. https://doi.org/10.1109/QoMEX48832.2020.9123103

[24] Constantin Sander, Ike Kunze, and Klaus Wehrle. 2022. Analyzing the Influence of Resource Prioritization on HTTP/3 HOL Blocking and Performance. (2022).

[25] Sandvine. 2023. 2023 Global Internet Phenomena Report. (2023). https://www.sandvine.com/global-internet-phenomena-report-2023.

[26] David Schinazi and Lucas Pardue. 2022. HTTP Datagrams and the Capsule Protocol. RFC 9297. https://doi.org/10.17487/RFC9297

[27] Marten Seemann and Jana Iyengar. 2020. Automating QUIC Interoperability Testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3405796.3405826

[28] Victor Vasiliev. 2023. *The WebTransport Protocol Framework.* Internet-Draft draft-ietf-webtrans-overview-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-webtrans-overview/05/ Work in Progress.