Modeling the Architecture of QUIC Implementations with rustviz

Leopold Jofer, Daniel Petri*, Marcel Kempf*,
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: leo.jofer@tum.de, petriroc@net.in.tum.de, kempfm@net.in.tum.de

***Abstract*—QUIC implementations differ vastly in their support for the different QUIC standards and extension-RFCs. A lack of documentation and an architectural overview make it difficult to judge which library supports what, how extensible they are and how they are structured. This paper examines two QUIC implementations: tquic by Tencent and neqo, developed by Mozilla for the Firefox web browser. Utilizing our custom tool, `rustviz`, we break down their architecture and subsequently model their design and intended usage via the C4 software model. From that, we infer their RFC support and extensibility and try to find potential anti-features or deterrents to including them in a project. Our findings were added to the QUIC Explorer, which makes it easy for developers to filter and assess the suitability of QUIC libraries.**

***Index Terms*—QUIC, QUIC Explorer, architecture, software, tquic, neqo, Rust, C4 Model**

1. Introduction

With the advent of the Web as a dominant platform in our lives, the speed, latency and reliability of web transport protocols have gained an increasingly important role. Traditional network communication, such as HTTP or SSH, use TCP for transmitting data. However, TCP was designed for the Internet of a different time and therefore has significant downsides and problems. Head-of-line blocking, connection establishment overhead and poor performance are just some of the baggage that TCP brings when using it [1]. As such, a need for a more efficient, faster and vertically integrated network stack has arisen. The next stage of this evolution is marked by QUIC, a new TCP alternative that aims to surpass these restrictions thanks to a more advanced design [2].

QUIC, a comparatively new protocol, solves and mitigates many of these problems, by integrating encryption, multiplexing streams and establishing connections in a more efficient manner. It is a user-space protocol, building on the properties of UDP, with various congestion control algorithms, that can detect packet loss across streams without blocking. On top of QUIC, HTTP/3 was designed as an efficient application layer protocol that aims to replace and improve HTTP/1.1 and HTTP/2 with better header compression and without head-of-line blocking [3].

However, when using QUIC in their application, developers may face challenges in finding a suitable implementation that fits their needs. Information about which libraries exist, what features they support and how standard-compliant they are is scattered, scarce and often out of date.

1.1. Adoption Landscape

Google, Cloudflare, Apple, Tencent and many more each have their own QUIC implementations with varying support for the different iterations and RFCs related to QUIC [4], [5]. As a developer, finding the right library or tool for your use case can therefore be quite hard, as public information is scattered across different websites, tables, repositories and documentations. Sometimes, the information given about what an implementation supports is quite scarce. Finding out what really is in the codebase is challenging.

1.2. Scope of the paper

In this paper, we cover two implementations, tquic by Tencent and neqo by Mozilla, which is used in firefox. We use our newly developed tool, called `rustviz`, to dive deeper into the codebase and find anomalies and features regarding the implementations. Our findings will be added to the QUIC explorer.

2. Background

QUIC is not a singular standard or RFC. Rather, it is composed out of multiple RFCs, extensions, and drafts that describe different parts and additions to the protocol. For example, HTTP/3's header compression with QPACK is specified in a separate RFC from the underlying QUIC protocol [6]. Some parts of the QUIC protocol itself (like congestion control) are described by the RFC, yet their concrete algorithm or implementation is not. Libraries can evoke security and compatibility concerns, like the choice of the TLS library used - which is a frequent target for attacks [7] - or implementation correctness.

2.1. QUIC Explorer

There exist many different QUIC implementations with different characteristics, maturity levels, and use cases. Judging which one is right for a certain use case can be a fairly complex problem. The QUIC Explorer [4] provides a central repository that enables users to filter QUIC implementations by feature and language and provides them with a simple overview.

2.2. C4 model

In this paper, we use the C4 modeling approach to examine the architecture of the libraries that are considered. The C4 model defines four abstraction layers:

- the *context* in which a software system exists,
- the *containers* of which the aforementioned software system is composed,
- the *components* of such a container, e.g. code modules
- the actual *code* of the application [8] [9]

C4 helps to frame the context in which a library is being used, as well as the inner workings of the library itself. Since we are examining the architecture of libraries — not entire software systems — in this paper, the relevant layers for this paper are the container layer and the component layer. Modeling the actual code would mostly yield low-level implementation details, which are not relevant for us. Instead, small peeks into the codebase when necessary are deemed adequate.

2.3. Encrypted Client Hello (ECH)

ECH is a mechanism to hide the site the user is visiting. When a client connects to a server with TLS, the Server Name Identification (SNI) as well as other TLS extensions are unencrypted, allowing potential adversaries to uncover which server the client is connecting to. ECH uses a public key (most commonly from a DNS record) to encrypt the SNI, therefore making it substantially more difficult to reveal what web address a client is visiting [10].Hence, it yields a considerable privacy benefit. The main threat actors that this protocol protects against are network observers and ISPs. It is however only effective when multiple websites use the same ECH origin. If every web origin/domain had its own ECH address, that address would still uniquely identify a visitor.

QUIC does not create connections layered over a TLS connection, but instead integrates it directly into the protocol. The initial handshake also establishes the encryption as well as the data connection. Encrypted Client Hello does not require a separate TLS layer to function. Instead, the QUIC-TLS handshake (`CRYPTO` packets, see [11]) is encrypted and wrapped in a so called ClientHelloOuter-Message. This outer message contains a SNI that is generic enough to be useless to an adversary. The receiving QUIC server decrypts this message, thus uncovering the actual destination of the packet and redirecting the stream.

2.4. Sans-I/O

The Rust package ecosystem is split into synchronous (using the standard library and threads) and asynchronous packages (using a third-party executor for the futures). The choice of executor furthers this fragmentation, as every executor has a slightly different I/O interface. For example, a library written for `mio`, Tokios I/O implementation is not compatible with `async-std`, another async executor [12].

To circumvent this issue, Rust libraries for protocols typically resort to feature flags to allow for switching executor. In other cases, they just pick one. Sans-I/O libraries instead expose a set of handles that hook into I/O for the caller to implement, like e.g. quiche by Cloudflare [13]. Another advantage of this approach is that it future-proofs the library to a certain extent and makes it easier to use in all kinds of environments, like embedded or IoT (Internet of Things) devices.

3. Design and Methodology

In order to not exceed the scope of the paper, go into meaningful depth and be able to make a sensible comparison, we chose the following two QUIC implementations: *neqo* by Mozilla and *tquic* by Tencent.

3.0.1. neqo by Mozilla. neqo is Mozilla's QUIC implementation, written in the Rust programming language [14]. Its primarily used in Firefox, tightly integrated as a part of the greater — not to be confused by name — `Necko` network stack [15]. In fact, neqo's primary purpose does not entail being used outside of the Firefox/`Necko` ecosystem. It is not published on crates.io, the Rust package registry. As a result, neqo on its own has not seen wider adoption beyond that scope. But, due to its use in Firefox, which has around 155 million monthly active users [16], we nonetheless consider it relevant to analyze.

3.0.2. tquic by Tencent. *tquic* is a performant QUIC library developed by Tencent, also written in Rust. Its primary use is in the Tencent cloud [17]. In contrast to neqo, tquic is being developed as a standalone library that can be integrated into various codebases and has also been released as a crate on crates.io [18]. Since existing public information on the library is sparse even though it is used quite widely, it is the second library that we analyze.

3.1. Diagram Creation

Creating component-level diagrams by hand is tedious and can be error-prone. It is easy to miss or misunderstand a part of the codebase and create a wrong schematic. We use a custom-made tool called `rustviz`[1] to generate these diagrams automatically. To reduce noise, our tool can filter it's output so that test suites and binaries can be omitted.

In addition, to showcase how the projects fit into the greater scope of the technical landscape and how they are used, we hand-create C4 diagrams on a container level. In select cases the source code of the implementation is examined as well.

3.2. Analyzed Features

Since our goal is to add our findings to the QUIC Explorer, the key features we examine partially stem from its list of features. Below is a list of the features that we look for in this paper and why.

- Encrypted Client Hello Support, since it is not yet part of the QUIC Explorer and relevant for privacy requirements.
- TLS Library choice, since old, untested or wrong crypto poses a huge threat to security.

1. github.com/leopoldlabs/rustviz

doi: 10.2313/NET-2025-11-3_16

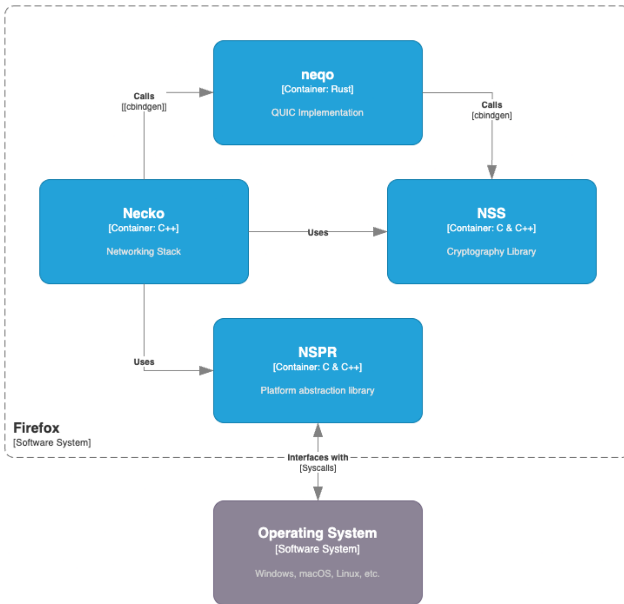

Figure 1: container view of neqos usage in Firefox

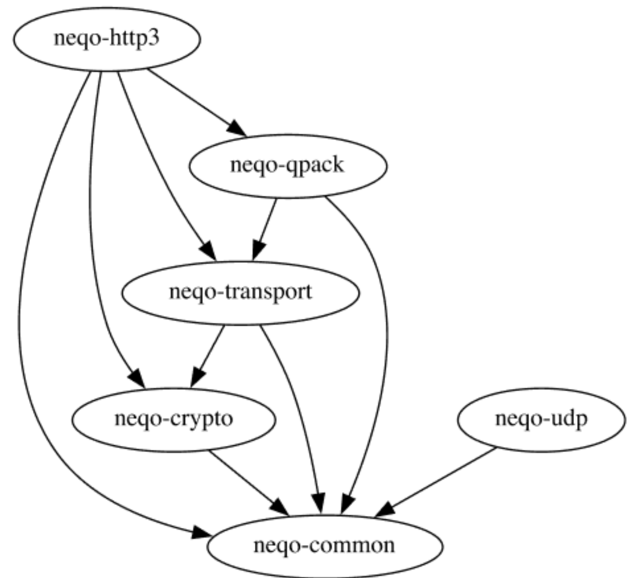

Figure 2: the individual crates that make up the neqo library. generated with `rustviz --filter fuzz,test-fixture,neqo-bin`, visualized with graphviz

4. Implementation

`rustviz` can operate on two abstraction layers of a Rust project. It is able to infer the dependencies between different crates in a cargo workspace by parsing and discovering the corresponding `Cargo.toml` files. From this workspace, an internal graph is built. Also, it can visualize the relationships between different modules, so it tracks which module imports which. This is done by generating documentation for a given crate with `rustdoc`. This documentation is then parsed and traversed with a breadth-first search in order to build a simplified module graph. In both cases, this Graph can then be output in textual form as C4-Diagrams or GraphViz-Diagrams. These can then be viewed with the MermaidJS Diagram Viewer or a GraphViz visualizer. Unnecessary modules can be omitted with the `--filter` option. This process uncovers architectural patterns and gives an easy overview of how the codebase is structured, making it easier to uncover support for different RFCs and standards. This provides a good entry point for exploring the codebase. The interactions between the various components of the codebase also unveil insights into its modularity and extensibility.

5. Evaluation

5.1. neqo

neqo's unique place in the Firefox codebase means that it depends on packages that are fairly unusual for the Rust ecosystem. Figure 1 shows that if the browser wants to establish a QUIC connection, Necko calls neqo via `cbindgen`. The sockets are handled by `NSPR`, a library that provides a cross-platform abstraction for sockets, files and other primitives [15], [14]. It does not implement I/O itself.

As visible in Figure 1, neqo uses the *NSS* (Network Security Services) library for cryptographic operations, the well-maintained library that is also used elsewhere in Firefox. It does so with C bindings, generated by `cbindgen`. Since there is no significant abstraction layer over NSS, swapping out the cryptography library is not easy in neqo and would require drastic code changes. neqo's NSS dependency and lack of publicly available crate makes it harder to integrate into regular Rust projects, as the build process becomes more complicated. NSS and NSPR are external C/C++ dynamic libraries [19] that are often out of date on older distributions. Consumers of neqo can build their own version of the NSS and NSPR. This process is outlined in the neqo documentation [14].

Digging deeper, Figure 2 shows the internal structure of the library and how different features are compartmentalized into various crates. Common abstractions, utilities and other shared pieces of code, that are used throughout the library, such as QLOG support, are inside `neqo_common`. On the other hand, higher-level abstractions of QUIC such as streams, network methods and UDP datagram parsing are all concentrated inside `neqo_transport`.

In order to provide the necessary cryptographic primitives for QUIC and to interface with the cryptographic library `NSS`, neqo provides C bindings. The definitions for these bindings are defined in `neqo_crypto`. HTTP/3 support can be layered on top using `neqo_http3`, which uses the QPACK decoder and encoder from the separate `neqo_qpack` crate. neqo has full support for client-side HTTP/3 and experimental support for server-side HTTP/3, which is not meant for production use but rather client-code testing [14, neqo-http3/src/lib.rs:21-23]. Taking a closer look at the `neqo_transport` crate reveals that neqo has support for ECH. The `Connection` struct contains methods to enable ECH on the server and client side respectively, as well as methods to retrieve the current ECH configuration of the connection.

5.2. tquic

tquic is being developed for general usage in Rust projects on the server side and is also a sans-io QUIC

 doi: 10.2313/NET-2025-11-3_16

library [17]. In contrast to neqo, it is not divided into different crates with different functionalities, instead it is only organized with Rust modules. It hooks itself into a server (or client) as a user space library. When a connection reaches a server, the kernel of the server forwards those UDP packets to the respective web server. The web server then calls the delegated handlers provided by tquic, so that the task of parsing them and handling the connection is delegated from the main program to tquic, the library. Just like neqo, tquic also contains an HTTP/3 and a QPACK implementation, but they are hidden behind the `h3` feature flag and contained inside of the `h3` module. We are going to update this feature on the QUIC Explorer for both.

tquic uses Google's BoringSSL, a mature crypto library forked from openSSL that is also used in Google Chrome. It does so in conjunction with the `ring` crate, which has been explicitly marked as an experiment in it's README [20]. Presumably this was done to protect the author of any liability. Yet, the seeming risk of this dependency is unfortunately not mentioned anywhere in the tquic docs. tquic is not presented as experiment either. An explanation would be sensible and helpful in this context. It does not contain an ECH configuration. There is no mention of it in the codebase and official documentation.

6. Conclusion and future work

Both tquic and neqo are solid, architecturally sound choices when picking a QUIC library and they will stay so for the foreseeable future due to their large supporters.

Due to the QUIC specification and extensions being so expansive, the number of elements to consider when picking a QUIC library is substantial. Finding out the extent of what RFC and feature is actually implemented remains a tedious task. While the QUIC Explorer mostly mitigates this and provides a nice overview, it still needs to be kept up-to-date and expanded as well. Library developers should put more emphasis on compatibility documentation and provide more usage examples.

6.1. Future outlook

Code comments and docs can state that a feature is supported, but whether it is implemented lacklusterly or properly can hardly be inferred from that.

6.1.1. Spec compliance. An example for this kind of testing is the QUIC tests developed by Microsoft for their protocol testing language, Ivy. The tests can verify that a implementation is spec compliant.

The tquic developers claim to have used these tests for their library, however their test environment and test harness version were not published. Ivy itself is unmaintained [21], just like the tests that were published for QUIC within the ivy repository. Not only would the language require updates, the tests would also have to be rewritten in order to be compliant with the RFC, not just the old drafts.

6.1.2. Automated Testing. New versions of libraries can potentially create unintended regressions. While both neqo and tquic come with Unit tests, full integration tests, potentially using ivy have the potential to catch more errors. A public dashboard could show the percentage of passing tests, and how the resulting spec compliance evolves over time.

6.1.3. Better generation of C4 Diagrams. This is regarding the generation of code-level diagrams that showcase the relationships between functions, structs, traits, etc. This is presumably also useful in a broader scope than QUIC library analysis, for example as an educational help for Rust learners and teachers.

7. Related Work

There are a few similar tools to rustviz.

- `cargo tree` in the Rust Toolchain outputs a tree visualization of a crate's dependency graph [22].
- `cargo-modules`, a separate tool, shows a tree overview of a crate's modules [23].

References

[1] M. contributors, "Head-of-line blocking - Glossary," 2025, [Online; accessed 11-Aug-2025]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Head_of_line_blocking

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

[3] G. Perna, M. Trevisan, D. Giordano, and I. Drago, "A first look at http/3 adoption and performance," *Computer Communications*, vol. 187, pp. 115–124, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366422000421

[4] M. Kempf, "QUIC Explorer," https://quic-explorer.net, accessed: 2025-08-12 (Commit 105a3a4). [Online]. Available: https://quic-explorer.net

[5] I. Q. W. Group, "implementations.md - quic working group." [Online]. Available: https://github.com/quicwg/quicwg.github.io/blob/main/implementations.md

[6] C. B. Krasic, M. Bishop, and A. Frindell, "QPACK: Field Compression for HTTP/3," RFC 9204, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9204

[7] "Vulnerabilites - OpenSSL," 2025, [Onlin,e accessed 12-Aug-2025]. [Online]. Available: https://openssl-library.org/news/vulnerabilities/index.html

[8] S. Brown, "The C4 Model for Software Architecture," 2018, [Online; accessed 21-June-2025]. [Online]. Available: https://www.infoq.com/articles/C4-architecture-model/

[9] ——, "The C4 model for visualising software architecture," 2019, [Online; accessed 21-June-2025]. [Online]. Available: https://c4model.com/

[10] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood, "TLS Encrypted Client Hello," Internet Engineering Task Force, Internet-Draft draft-ietf-tls-esni-25, Jun. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-esni/25/

[11] M. Thomson and S. Turner, "Using TLS to Secure QUIC," RFC 9001, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001

[12] M. Endler, "The State of Async Rust," 2025, [Online; accessed 20-Aug-2025]. [Online]. Available: https://corrode.dev/blog/async/

[13] A. G. (Cloudflare), "Enjoy a slice of QUIC, and Rust!" 2019, [Online; accessed 21-June-2025]. [Online]. Available: https://blog.cloudflare.com/enjoy-a-slice-of-quic-and-rust/

[14] Mozilla, "Neqo, the Mozilla Firefox implementation of QUIC in Rust," 2025, [Online; accessed 21-June-2025]. [Online]. Available: https://github.com/mozilla/neqo/tree/main

[15] ——, “Firefox Source Docs,” 2025, [Online; accessed 15-Aug-2025]. [Online]. Available: https://firefox-source-docs.mozilla.org/networking/http/http3.html

[16] ——, “Firefox User Activity Dashboard,” 2025, [Online; accessed 15-Aug-2025]. [Online]. Available: https://data.firefox.com/dashboard/user-activity

[17] Tencent, “tquic Documentation,” 2025, [Online; accessed 15-Aug-2025]. [Online]. Available: https://tquic.net/docs/intro/

[18] “tquic - Crates.io,” 2025, [Online; accessed 15-Aug-2025]. [Online]. Available: https://crates.io/crates/tquic

[19] “nspr: Summary - Mozilla Mercurial,” 2025, [Online; accessed 21-June-2025]. [Online]. Available: https://hg-edge.mozilla.org/projects/nspr

[20] B. Smith, “ring,” 2025, [Online; accessed 21-June-2025]. [Online]. Available: https://github.com/briansmith/ring

[21] Microsoft, “IVy is a research tool intended to allow interactive development of protocols and their proofs of correctness and to provide a platform for developing and experimenting with automated proof techniques.” 2020, [Online; accessed 21-June-2025]. [Online]. Available: https://github.com/microsoft/ivy

[22] “cargo-tree(1),” 2025, [Online; accessed 24-Aug-2025]. [Online]. Available: https://doc.rust-lang.org/cargo/commands/cargo-tree.html

[23] regexident, “cargo-modules,” 2025, [Online; accessed 24-Aug-2025]. [Online]. Available: https://github.com/regexident/cargo-modules

 doi: 10.2313/NET-2025-11-3_16