University of Glasgow | School of Computing Science

The QUIC Transport Protocol

Emily Band, 2038561b

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 28 March 2018

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Introduction

1.1 Overview

Networked applications are effectively restricted to a choice of two transport protocols: TCP or UDP. These protocols are decades old, and were designed to support a much smaller volume of traffic than the vast amounts generated by modern Internet. Any inefficiency in these protocols generates a large amount of redundant traffic, a significant problem when systems have fewer clock cycles available to process each incoming packet.

Ossification of transport protocols has been enforced as a result of interference from middleboxes, which are ill-equipped to deal with new transport protocols built directly on top of IP; SCTP and DCCP are examples of failed attempts to deploy more effective replacements for TCP and UDP respectively.

This paper explores the development of the QUIC protocol, an attempt to subvert this ossification by building a userspace transport protocol on top of UDP: the use of a familiar protocol enables widespread deployment, while allowing sufficient freedom to build loss-recovery features, parallel streams for asynchronous content delivery, and a reduced latency handshake process compared to regular TCP stacks.

This project details the process of creating Mercury, a QUIC library implemented in Rust. Although Rust is not currently used for any implementations of QUIC[3], its region-based approach to memory management gives Mercury several advantages over existing solutions: a safer, more maintainable alternative to the implementations which use C or C++, and one which has faster performance than the ones which use Go.[5]

Another key focus of this project is reflecting on attempting to implement QUIC as a solo developer. The IETF specification for this protocol is extensive, with eight active documents on the IETF Datatracker page for the QUIC working group.[6] Learning and constructing a QUIC library is a very different experience for a student with considerable time constraints compared to working as part of a team of experienced engineers at an influential company.

1.2 Outline of Paper

This paper contains the following chapters:

- **Rust: Safer Systems Programming:** reflects on the process of learning a language with an unconventional approach to memory management, and evaluates its advantages and disadvantages.
- **The IETF: Open-Sourcing the Internet:** details the structure, conventions, and politics behind creating Internet standards.
- **The QUIC Protocol:** explores previous attemps to improve TCP, the mechanics of QUIC, and its benefits.
- **Project Scope and Requirements:** explains which sections of the QUIC protocol could be feasibly implemented in the time given.
- **Creating Mercury: A Rust-Based QUIC Library:** explores the technical details of creating a QUIC library for the chosen areas of the specification, along with reflections on which sections could be improved.
- **Status and Future Work:** summarises the current capabilities of the library, the areas of the specification which will be implemented in a future iteration of the Mercury project, and the issues which are likely to affect this.
- **Conclusions:** discusses the use of Rust as a tool for low-level networking, interacting with the IETF as a solo participant, the potential of moving transport protocols from the kernel into userspace, and the impact of encrypted transport protocols on the Internet as a whole.

Chapter 2

Rust: Safer Systems Programming

C and C++ are still the languages of choice for systems programming - the vast majority of current QUIC implementations are written in one of these.[3] Although programs written in these languages are fast and efficient, they are prone to memory leaks and segmentation faults due to incorrect memory management. Race conditions are common in multithreaded C and C++ programs due to restrictions on concurrent memory access not being strictly enforced.

Rust uses region-based memory management and ownership to allow efficient, predictable use of resources while minimising the risk of memory leaks, illegal memory accesses, and race conditions. This focus on safety and performance is perfect for a multithreaded transport protocol, but no IETF implementation has taken advantage of it yet.

2.1 Memory Management Strategies

There are several strategies which programming languages use to manage memory allocation:

[This might be better as a list?]

2.1.1 Reference Counting

[Discuss Python's abstraction of malloc and free - low-effort for programmers and good for small numbers of objects with lots of references, but terrible performance for lots of small objects. Requires intervention to detect and eliminate isolated cycles.]

2.1.2 Garbage Collection

[Discuss Java's 'stop the world' approach and generational collection - new used as abstraction for malloc. No effort required from the programmer but performance suffers. Makes Java unsuitable for systems programming due to slowdown and unpredictable use of resources.]

2.1.3 Manual Memory Management

[C's use of malloc and free - fast and efficient when done correctly, but requires caution from the programmer. Compiler does not check for memory leaks, segmentation faults, or buffer overflows caused by incorrect memory management. Multithreaded applications are even more complex to manage.] [Include sample code showing a buffer overflow error]

2.1.4 Region-Based Memory Management

Rust uses region-based memory management to free allocated heap memory as soon as the compiler detects that a value can no longer be accessed. In practice, this means that a Rust program will implicitly call `drop()` on a variable which is going out of scope when a function ends, and will free the heap memory which was allocated for its associated value.

[Include a code fragment and diagram to illustrate this]

References can be used to allow a function to borrow values, allowing it to use them without removing data from the heap after exiting:

[Include a code fragment and diagram to illustrate mutable and immutable borrowing]

2.2 A Steep Learning Curve

Rust has acquired a reputation for being difficult to learn.[2][4][9] I would broadly agree with these sentiments, with the following four areas being the most notable obstacles: ownership, lifetimes, strings, and compiler errors.

2.2.1 Ownership

Attempting to use a variable after it no longer points to a value on the heap will result in a compiler error:

[Include example of attempting to use value after move]

While this is easier to debug than encountering a segmentation fault or undefined behaviour during runtime, it is initially confusing why this throws an error in the first place when this code would be legal in many other popular languages. The answer to this lies in Rust's approach to ownership of data, which is best explained through comparing two similar traits: `Copy` and `Clone`.

`Copy` and `Clone`

Values which have a constant value known at compile time, such as integers, are assigned memory on the stack instead of the heap and do not need to call `drop()`. The `Copy` trait is present for variables which have stack-allocated values (ie. anything which doesn't implement the `Drop` trait). Making a copy of

a stack-allocated value is guaranteed to be a computationally cheap operation given its known size at compile time, so stack-allocated values can be easily assigned to multiple variables without ownership conflicts:

[Include code fragment showing copy behaviour]

[Include diagram of pointer and data changes for copy]

`Clone`, by contrast, is an explicit function call which creates a deep copy of a value on the heap and allows a variable to take ownership of it:

[Include code fragment showing .clone()]

[Include diagram of pointers and data manipulations for clone]

Attempting to treat a heap value in the same way as a stack value creates a scenario where `drop()` would be called twice on the same section of memory after the function terminates: Rust's version of a double free error in C.[8] This illegal conflict of ownership throws the compiler error shown in (fig no. here):

[Include diagram of illegal pointer configuration]

Assigning a new pointer to a copy of a stack value is guaranteed to be a computationally cheap operation compared to copying a potentially large section of heap-allocated data, so custom types should use `Copy` rather than `Clone` where possible.

`Box<T>`

2.2.2 Lifetimes

2.2.3 `String` and `str`

2.2.4 Compiler Errors

[Rust has a list of 644 compiler errors, all documented on the official site.[1] 629 of these can still be emitted by the compiler.

There's even functionality in Cargo to detail a specific compiler error using the `--explain` command.]

2.3 `if` and `match`

2.4 `Result` and `Option`

[These abstractions are an additional layer of safety. Simplifies error handling - a wide range of errors can be encapsulated with `Err`, values which may or may not be present can be handled using `Some`. `.unwrap()` can cause runtime errors, but learning how to avoid these is relatively straightforward. `.expect(&str)` can be helpful for debugging.]

2.5 Traits

[Allows for a lot of flexibility for custom types: define functionality for one or two base functions, and Rust will know how a range of variants on these should work. Discuss Read and Write as examples w/ code fragments.]

2.6 `unsafe` Blocks

[Discussion of raw pointers, doubly-linked lists, and FFI]

2.7 Cargo

[Makes building projects and including code from other crates very easy. Rustdoc generates documentation which follows a consistent format - makes unfamiliar libraries easier to learn how to use and reduces the potential for errors in the documentation (typos in code, outdated types, etc).]

Chapter 3

The IETF: Open-Sourcing the Internet

3.1 QUIC Working Group

[Talk about mailing lists, github repos, and meetings - anyone is free to participate, no restrictions on entry]

3.2 TLS Working Group

3.3 Rough Consensus and Running Code

> ”We reject: kings, presidents and voting. We believe in: rough consensus and running code.”

Focus placed on getting working implementations: no delays caused by taking official votes, development is guided by convincing people about ideas on the mailing list and/or at meetings.

3.4 Corporate Interests

[GQUIC variant exists distinct from IETF-standardised QUIC (which in turn could be argued to be heavily influenced by Mozilla) - transport which has the potential to vary service-by-service is new territory, could have similar implications to Net Neutrality?]

Chapter 4

The QUIC Protocol

4.1 Existing Work to Improve TCP

[Standard TCP stacks are slow and most do not use TLS 1.3 as standard

NetMap and StackMap have been attempted as kernel-level improvements on TCP, but they have restrictions (taking up an entire networking interface, potential problems with encryption]

QUIC aims to become a userspace improvement on TCP: uses UDP as a base transport and adds flow control, loss recovery, and encryption with TLS 1.3

Initial version developed by Google and still in use for their services in Chrome (5% of internet traffic), standardised general-use version in development by IETF (representatives from Mozilla, Google, Facebook and BBC active on mailing list, a range of different implementations)]

4.2 Protection For Legacy Systems

[Recent operating systems have TCP stacks with theoretically comparable performance to QUIC (TCP fast-open), but some systems cannot be upgraded (eg. NHS systems forced to continue using Windows XP). Userspace protocols allow at least some degree of improved performance and protection.]

4.3 Overcoming Ossification

[Reasons why middlebox-enforced ossification exists and strategies to avoid it (encryption, greasing)]

”Encryption *is* our defence against network ossification [...] That’s the only defence we have left against network middleboxes at this point.” [7]

4.4 Anatomy of a QUIC Connection

4.4.1 UDP Sockets

4.4.2 QUIC Packets

QUIC uses two types of packet: LongHeader and ShortHeader.

4.4.3 LongHeader

Used for exchanging 0-RTT key-protected data and initial connection setup.

[Include packet format diagram and explain sections]

4.4.4 ShortHeader

Used for exchanging 1-RTT key-protected data.

[Include packet format diagram and explain sections]

4.4.5 Handshake Process

4.4.6 Payload Frames

4.4.7 Connection Teardown

Chapter 5

Project Scope and Requirements

[Can only implement Handshake in time given - most implementations are created by experienced IETF participants, not novice undergrads]

Chapter 6

Creating Mercury: A Rust-Based QUIC Library

[NOTE: Mercury is quic-08 draft compliant]

6.1 The `mio` Crate

Enough flexibility for the project, simple to learn. `tokio` is more complex but may be a better choice for multithreaded work, might convert to this to prepare for work with multiple streams.

6.2 `header` module

[Detail the custom functions here, encoding and decoding process]

6.3 Porting TLS 1.3 to UDP

6.3.1 Selecting a TLS Library

Mozilla NSS and OpenSSL are highly customisable, but far too heavyweight for the project. picotls has been developed specifically for QUIC, but is written in C. Could use FFI to integrate it into this project, but adapting rustls preserves memory safety through minimising use of raw pointers, and is a better learning experience for understanding how TLS is supposed to work in QUIC.

6.4 Modifying the `rustls` Server Example

6.4.1 Events Poll

6.4.2 Simulating Connections in UDP

6.4.3 Custom Socket Type: `QuicSocket`

6.4.4 Consolidating Handshake TLS Messages

6.5 Modifying the `rustls` Client Example

Chapter 7

Status and Future Work

7.1 Current Status

7.2 Version Negotiation

7.3 Frames

7.4 Streams and Concurrency

7.5 Loss Recovery

7.6 Interop Testing

7.6.1 Unexpected IPv6 Behaviour

7.7 Version 10

7.8 Deciding between DTLS and TLS

7.8.1 Conflicting Advice from TLS Working Group

Chapter 8

Conclusions

8.1 Rust for Low-Level Networking

[Excellent choice after getting past the initial learning curve, huge potential if enough people support it.]

8.2 Working on a Solo Project in the IETF

[Impossible to complete a project alone as a student, but very good as a learning experience.]

8.3 Potential of Userspace Protocols

[Choices beyond basic TCP and UDP - applications can talk to a shim layer (eg. NEAT) to select a transport according to their current needs, no longer have to set up sockets directly. QUIC is just one of these possibilities.]

[Corporate influences over transport as mentioned in IETF section - could lead to segregated Internet]

8.4 The Future of QUIC

[Currently very fashionable and gaining influence. Direct backing from Google and heavy influence from Mozilla and Facebook suggests it will become widely adopted.]

8.4.1 A Less Open Internet?

[Difficult to monitor due to end-to-end encryption - great for user privacy, but much less data about network traffic available to track network performance. Restricted monitoring available with spin bit.

Encryption by default is going to be interesting with current UK government stance against E2E encryption.]

Bibliography

[1] Rust compiler error index. https://doc.rust-lang.org/error-index.html. Accessed 11/03/2018.

[2] Why is rust difficult? https://vorner.github.io/difficult.html. Accessed 11/03/2018.

[3] quicwg implementations. https://github.com/quicwg/base-drafts/wiki/Implementations, February 2018. Accessed 10/03/2018.

[4] Andrew Brinker. Setting expectations for rust's difficulty. http://www.suspectsemantics.com/blog/2017/01/26/setting-expectations-for-rusts-difficulty/, January 2017. Accessed 11/03/2018.

[5] The Computer Language Benchmarks Game. Rust programs versus go. https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=rust&lang2=go, February 2018. Accessed 10/03/2018.

[6] QUIC Working Group IETF. Quic working group documents page. https://datatracker.ietf.org/wg/quic/documents/, March 2018. Accessed 10/03/2018.

[7] Jana Iyengar. Quic: Replacing tcp for the web. https://youtu.be/BazWPeUGS8M?t=40m50s, January 2018. Accessed 10/03/2018.

[8] OWASP. Double free vulnerability. https://www.owasp.org/index.php/Double_Free, June 2014. Accessed 11/03/2018.

[9] Aaron Turon. Rust should have a lower learning curve. https://github.com/rust-lang/rust-roadmap/issues/3, November 2016. Accessed 11/03/2018.