
University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

# MQuicTT: evolving IoT at the transport layer

**Ivan Nikitin**

March 30, 2022

# Abstract

New advances in networked systems and programming languages are set to succeed existing industry standards. In the transport layer space, QUIC is set to supersede the TCP/TLS stack and is advantageous for performance and security by leveraging improved features such as a more straightforward, secure handshake and streams. While in the programming languages space, Rust, a new systems programming language, uses a strong type system to guarantee memory-safety and deadlock-freedom while still being performant. In this work, we analyse the feasibility of using a Rust based QUIC implementation as the basis for secure communication in IoT devices. To do so, in this work, we develop *MQuicTT* - a Rust based MQTT implementation using QUIC at the transport layer. We compare the performance of the resulting solution with existing MQTT implementations in various use cases and analyse the hardware resources needed to deploy it. We find that the performance of the implementation is on par with the baseline. Finally, we discuss the binary size of the implementation and the possible methods to generally reduce Rust binary sizes for IoT network firmware using QUIC.

# Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Ivan Nikitin Date: March 30, 2022

# Contents

# 1 Introduction

The Internet of Things (IoT) is rapidly becoming the most prominent form of computation, with the number of IoT devices projected to surpass 75 billion by 2025 (Statista 2016). IoT devices range from daily consumer gadgets such as wearables to sensors that are used in smart factories. The central theme of these devices is network connectivity. Network firmware installed on these devices has to be performant enough to send and receive massive amounts of data. Hence, the need for efficient, lightweight network firmware often means that manufacturers of these devices forego security for performance Ling et al. (2018). Therefore, the mass use of devices that are manufactured without security in mind creates a situation where many devices that are critical to infrastructure become vulnerable to cyber attacks, as seen in the 2015 attack on the Ukraine power grid (Liang et al. 2017). However, we can not just employ the usual methods to secure firmware on these devices due to the presented constraints in terms of their physical size, power consumption needs and available hardware resources. Hence, there is a need to develop protocols for secure, performant communication for IoT devices.

MQTT (OASIS 2014) is a popular message-passing network protocol designed to be lightweight for the IoT use case. While certain protocols have been developed to specifically act as data transfer protocols for IoT devices, MQTT remains the most widely used. MQTT's design ensures that the protocol's implementations have a small code footprint and take up minimal network bandwidth. Significantly, MQTT relies on a transport layer protocol to send data and ensure secure communication. The standard in current MQTT implementations is to use TLS to provide secure data transfer. As we present in this work, this adds overhead, which means that the advantages of MQTT are negated if we need secure communication.

There have recently been several developments in systems software that aim to succeed in existing industry standards. QUIC (Iyengar and Thomson 2021), a new transport layer network protocol initially designed at Google, is set to succeed TCP with a number of improvements. QUIC boasts advantages in both secure communication and performance by eliminating several overheads of TCP such as head of line blocking and requiring a much simpler handshake to establish a secure connection. In addition to this, the Rust programming language is a memory-safe systems programming language that aims to succeed C. Rust is secure by design due to its type-system and still provides the needed efficiency due to its region-based memory management approach adding no real overhead.

Hence, to take advantage of the developments made in the transport protocol and programming languages spaces, we must analyse if a Rust implementation of QUIC can be a viable alternative to existing widely-deployed implementations.

To do so, we introduce MQuicTT - a QUIC port of an MQTT library in the Rust programming language, discuss the design choices made during its development, analyse its performance and discuss the challenges IoT presents for network protocols. We show that *MQuicTT* is as performant as the baseline chosen TCP MQTT implementation in our testing scenarios. We further analyse the steps needed to create protocols for hardware constrained devices and present an analysis for the methodology of lowering the overhead of a QUIC stack. Finally, we analyse the libraries and features that contribute to the binary size of Rust implementations.

## 1.1 Dissertation Outline

The rest of this dissertation is structured as follows:

- Chapter 2 provides a background on the recent developments in the transport layer network protocol space, IoT and the Rust programming language.
- Chapter 3 presents the requirements that were set to answer our research questions and the design of the implementation of *MQuicTT* and its ecosystem.
- Chapter 4 details the implementation of *MQuicTT* and *QuicSocket* – an intermediate QUIC API that we have developed. This chapter also presents a discussion of the choices made during implementation and the difficulties that we circumvented.
- Chapter 5 details the experiment design and analysis of *MQuicTT* and provides a discussion of results.
- Finally, Chapter 6 concludes and summarises important results and discusses avenues for future work.

# 2 Background

In this chapter, we present the background of topics that are fundamental for this work. We examine the current developments in the networks transport layer, the constraints surrounding IoT devices, hardware constrained network protocols, the MQTT protocol and the Rust programming language.

We demonstrate why it may be favourable to use QUIC as the transport layer protocol for IoT devices and how it can solve some of the challenges in designing protocols for hardware constrained devices. Additionally, we explain the benefits of Rust as a programming language in the context of creating efficient, secure systems code.

## 2.1 The evolution of the transport layer

This section presents a background of the network protocols at the transport layer, starting from the currently dominant standards and ending with new advancements in the field. We also consider how to provide secure communication at the transport layer using encryption.

First described by Cerf and Kahn (1974), the Transmission Control Protocol (TCP) has been the primary protocol of the Internet suite since its initial implementation. TCP provides a *reliable* and *ordered* delivery of bytes, ensuring that data is not lost, altered or duplicated and delivered in the same order as intended by the sender. TCP achieves this by assigning a sequence number to each transmitted packet and requiring an *acknowledgement* (commonly referred to as ACK) from the receiving side. If an ACK is not received, the data is re-transmitted. TCP can also use the sequence numbers to order packets in the order intended by the sender on the receiving side.

As TCP is a connection-based protocol, connection establishment must occur before any data can be transmitted. The receiving side (the server) must bind to and listen on a network port, and the sender (the client) must initiate the connection using the process of a *three-way handshake* as shown in Figure 2.1. In the first step of the handshake, the client sends a segment with a *synchronise sequence number* (SYN) that indicates the start of the communication and the sequence number that the segment starts with. The server responds with an acknowledgement - ACK, and the sequence number it will start its segment with - SYN. Hence, we refer to this step as the SYN-ACK. In the third and final step, the client must acknowledge the response. At this point, TCP establishes the connection and can transfer data on it.

In order to achieve *secure communication*, TLS (Rescorla 2018) is often used in the TCP stack. In order to do this, a separate TLS handshake has to occur to specify the version of TLS to use, decide on the cypher suites, authenticate the server via its public key and certificate authority's signature, and generate a session key that the protocol uses for symmetric encryption during communication. In the TLS handshake, the first step is for the client to send a *ClientHello* message that specifies the highest version of TLS that the client supports, a list of suggested cypher suites, compression methods and a random number. The server responds with a *ServerHello* message containing the selected TLS version, cypher suite, compression method, and random number. The server then sends its certificate and *ServerKeyExchange* message along with the *ServerHelloDone* message indicating that it has completed its part of the negotiation process. The client will respond

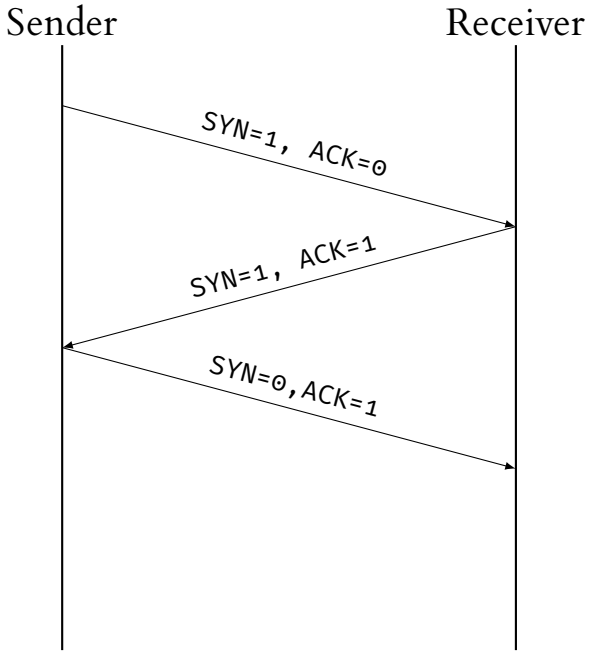


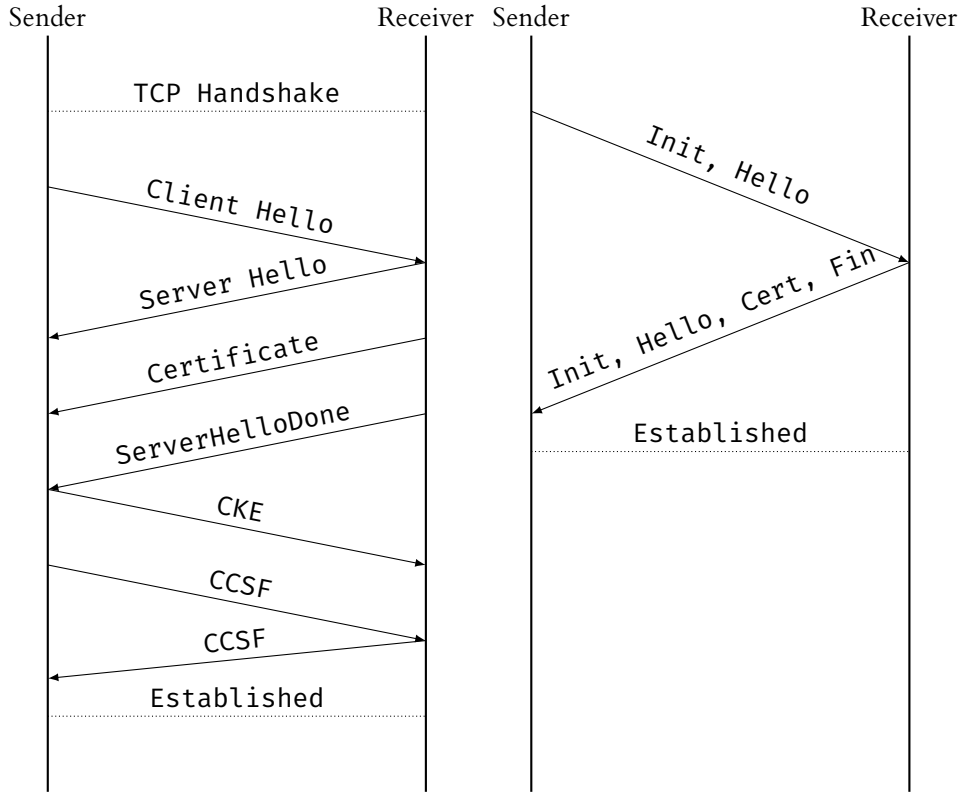

***Figure 2.1:*** *The TCP handshake needed for connection establishment. The values of the SYN and ACK fields being set indicate the kind of segment sent. For example, for the SYN-ACK stage both the SYN and ACK fields are set.*

with the *ClientKeyExchange* message (CKE), which may contain a public key depending on the chosen cypher suite. Following this, the client sends a *ChangeCipherSpec* message indicating that communication from this point is authenticated and encrypted. The client combines this message with a "client finished" message. The server responds with the same message, establishing the TLS connection.

Due to the establishment of communication and properties guaranteed by TCP, this process lengthens communication latency. Hence, in use-cases where reliability and connection state is not required, the User Datagram Protocol (UDP) (Postel 1980) is preferred. UDP uses a connectionless communication model with a minimal number of implementation semantics. The only mechanisms provided by UDP are port numbers and checksums in order to ensure data integrity. UDP is preferable for real-time systems as using TCP would cause overhead to latency and retransmission of packets that the application no longer needs.

While TCP and UDP have been the dominant standards of the internet since their creation, QUIC is a relatively new general-purpose transport layer protocol. QUIC was first designed by Roskind (2012) at Google as part of the Chromium web engine and standardised by the IETF in 2021 (Iyengar and Thomson 2021). It aims to improve upon, and eventually make obsolete, TCP by using the concept of multiplexing over a UDP connection. Multiplexing is a method of combining several signals or channels of communication over one shared medium. QUIC makes use of multiplexing by facilitating data exchange on the UDP connection through the concept of *streams*. Streams are an ordered byte-stream abstraction used by the application to send data of any length. Any number of QUIC streams, unidirectional or bidirectional, can be created by either side during the connection. Hence, QUIC allows an arbitrary number of streams to send arbitrary amounts of data on a single UDP connection, subject to the constraints imposed by flow control.

By doing so, QUIC also achieves other performance benefits. For example, QUIC lifts congestion




*(a) TCP/TLS* *(b) QUIC*

***Figure 2.2:*** *Handshakes required to establish secure data transmission in the TCP/TLS stack (a) and the QUIC stack (b). In the case of TCP/TLS, we can see that the handshake is substantially more complex and that the TLS handshake requires the full TCP handshake before it can proceed. In both cases, these handshakes can be made quicker. In the case of TLS, version 1.3 allows for one less round trip before data can be sent, and in the case of QUIC 0-RTT, connection re-establishment may be used in some cases, allowing to send data in the first packet.*

control algorithms from the kernel space to the userspace. Hence, congestion control algorithms can evolve without being tied down to kernel level semantics and constraints.

Compared to TCP/TLS, QUIC combines the transport and cryptographic handshakes to minimise the time needed for connection establishment; Figure 2.2 shows a comparison of the handshakes. QUIC still uses TLS functionality to secure communication as described by Thomson and Turner (2021) unless the developer specifies a different cryptographic protocol; however, this is done differently to TCP. The initial QUIC handshake keeps the same handshake messages as TLS; however, it uses its framing format, replacing the TLS record layer. This use ensures that the connection is always authenticated and encrypted, unlike TLS, where the initial handshake is vulnerable. The combination also means that QUIC typically starts sending data after one round-trip, achieving security by default and lower latency.

## 2.2 The Internet of Things

There is no set definition for what constitutes an IoT device. However, an IoT device is generally a small chip that possesses some processing power and may have embedded sensors. The key

aspect of IoT devices is that they facilitate data exchange with other devices and systems over the Internet. The modern version of IoT can be attributed to Weiser's (1991) work on ubiquitous computing, although the term itself first appeared in a speech by Peter T. Lewis in 1985. IoT has applications in various fields, including smart home automation, healthcare, consumer applications, etc.

In terms of classifications within networking and IoT technologies, we can generally split them into wireless and wired, with the former split into short-range, medium-range, and long-range.

Short-range wireless IoT technologies include Bluetooth mesh networks, Z-wave, ZigBee and Wi-Fi, and other lesser-used technologies. These technologies are the most prevalent in consumer IoT devices, such as smart home applications. Medium range networks are used heavily in mobile devices with technologies such as LTE and 5G. The technologies again present an interest due to the amount of traffic that the Internet sees from mobile devices. On the other hand, long-range networks are rather specific in their applications; for example, VSAT - a satellite communication technology that uses small dish antennas is not something we would necessarily think of when encountering IoT. Due to the limited application of long-range technologies, we opted to leave them out of our analysis.

Ethernet remains the dominant general-purpose networking standard in terms of wired technologies used by IoT devices. Although wired technologies provide advantages in terms of data transfer speed, they limit deployments due to the physical wiring constraints.

Having defined what we mean by an IoT device in a network, we now consider the constraints that apply to these devices. Due to the use cases of IoT, the form factor of these devices should be small. For example, a processing unit inside a home assistant has to fit in its enclosure. Additionally, many of these devices have to run for long periods, sometimes on a single lithium battery, hence needing to consume as least energy as possible. Many use cases of IoT devices also require many of them connected in a mesh network. For example, Ericsson (2018) estimated that 0.5 connected devices were used per square meter in a smart factory, with demand growing. Large scale deployments add economic constraints to IoT devices as they need to be manufactured from relatively cheap components.

These constraints mean that IoT devices are limited in hardware resources. Hardware limitations come in three primary forms - CPU power, memory and storage. Storage in the form of flash memory provides the hardest to solve problems regarding secure data transfer. The keys required for protocols such as TLS are often large and need to be stored. For example, the *ESP*8266 controller, a widely used IoT chip, comes with 4Mb of flash memory. After installing the firmware and binaries needed, little to no memory may remain for additional storage.

The circumvention of these constraints at the cost of insecure firmware and communication, amongst other issues, is why IoT has become synonymous with security concerns. Efforts to classify the security issues in the IoT space (Alaba et al. 2017; Gupta and Lingareddy 2021; Swamy et al. 2017) and create a taxonomy have generally shown several main topics: insecure firmware level code, issues with privacy due to authentication and authorisation and general security concerns due to poor encryption at the transport layer.

Insecure firmware in IoT devices comes from issues with firmware updates and general vulnerabilities stemming from code. A primary reason is that most programmers opt to create software for IoT devices in memory unsafe languages. Languages such as C provide the needed efficiency to circumvent processing constraints; however, they also leave room for memory management issues, leading to vulnerabilities. When it comes to privacy, the primary goal is ensuring data integrity and confidentiality. Data sent via the network must not be tampered with nor snooped on by third parties during communication. Man in the middle attacks is a prime concern for protocols such as MQTT due to their lack of self-imposed encryption.

Hence, finding a way to circumvent the hardware constraints presented by IoT devices and still

provide secure data transfer is paramount to the safe adoption of IoT. Additionally, opting to create IoT firmware code in a memory safe language may prevent vulnerabilities that are present on IoT devices from the moment of deployment.

## 2.3 MQTT

Considering the constraints that apply to IoT devices, we will now look at one of the widely used application-level IoT protocols. MQTT (OASIS 2014), originally standing for Message Queuing Telemetry Transport, is designed to be a lightweight protocol to transport messages between devices using the publish-subscribe method. MQTT defines two types of participants - the broker and the client. The broker is a server that receives messages from all clients and then routes these messages to the appropriate destinations. On the other hand, a client is a device that runs an MQTT library and sends the broker messages.

The broker handles the routing of messages in MQTT via *topics*. When a client wishes to publish a message, it does so on a client, and the broker distributes this message to all other clients subscribed to this topic. The broker facilitating communication means that the publishing client does not need to keep track of other clients' locations to communicate with them.

Communication via MQTT can happen after the initial connection request from the client and the subsequent connection acknowledgement from the broker. The connection request can specify a quality of service, referred to as the QoS parameter, to indicate the nature of message delivery.

The possible QoS parameters are as follows:

- At most once (fire and forget) - a client sends a message once, and the broker takes no steps to acknowledge delivery.
- At least once (acknowledged deliver) - a message is re-sent until the broker acknowledges it has received it.
- Exactly once (assured delivery) - the client and broker have to participate in a two-way handshake to ensure that a single copy of a message is received.

We present an example of an MQTT connection with two clients and a broker with QoS of at most once in Figure 2.3. With each level of QoS measure increasing, so does the communication overhead. However, this measure only affects the MQTT part of the communication. The underlying transport layer protocol, such as TCP, will still act as intended. Hence, MQTT relies on an underlying transport-level protocol for data transmission. Additionally, MQTT sends all its connection credentials in plain text format; hence, the communication is vulnerable if the transport layer does not provide encryption. The most widely used suite for providing data transfer and encryption for MQTT is TCP/TLS; however, any protocol that provides lossless, bi-directional communication can be used.

As QUIC provides such a form of communication, we can see that we can use it as the transport level protocol for MQTT. The benefits of this are numerous. For example, perceived performance benefits from lower communication overhead and encryption at header and packet levels. QUIC also comes with the benefit of multiplexing. QUIC can open several streams on a single connection instead of opening several connections from one client. Multiplexing will later play an essential role in our implementation of MQTT/QUIC.

The first implementation of MQTT using QUIC presented by Kumar and Dezfouli (2019) uses the *ngtcp2* implementation of QUIC in the C programming language. The authors find that QUIC reduces the connection overhead time significantly and reduces the processor and memory usage.

Due to the authors' chosen underlying QUIC implementation, their API implementation requires its own message queue amongst other common functions such as key settlement. The QUIC

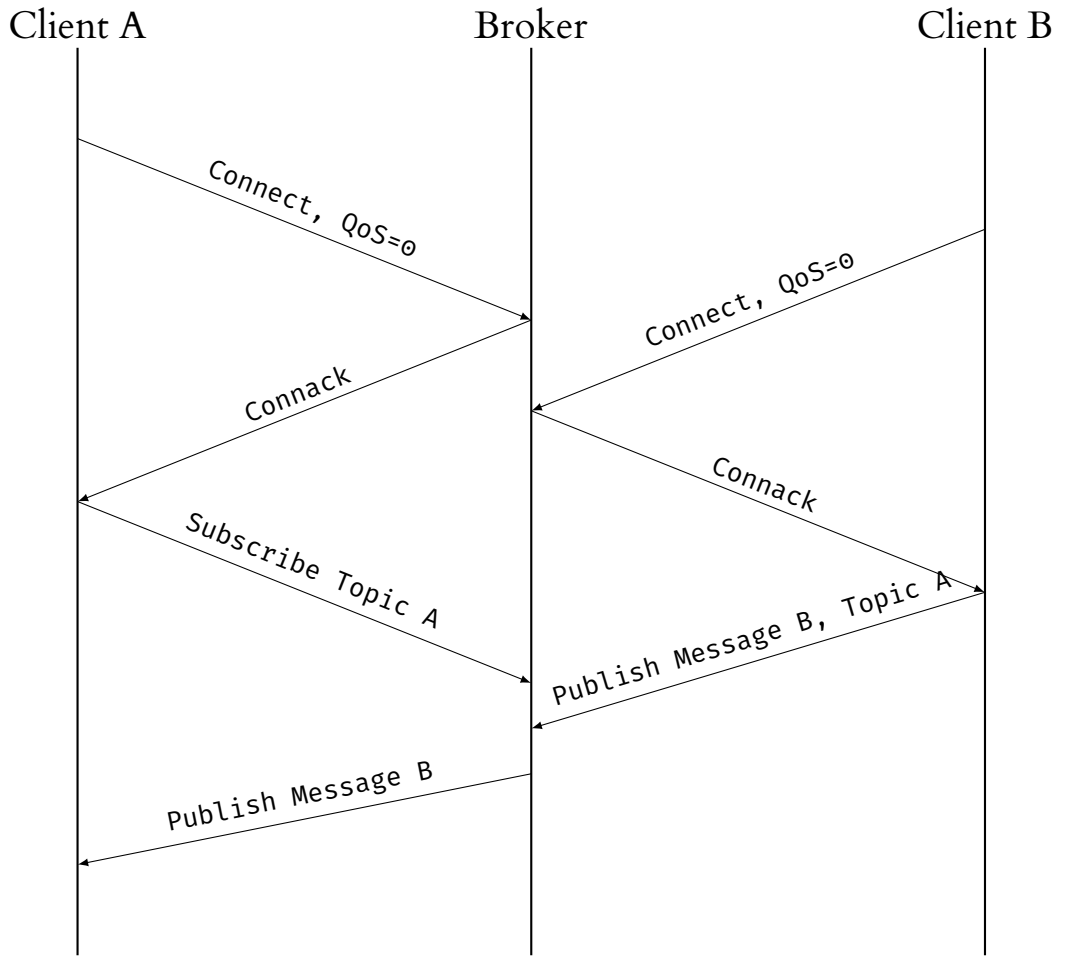



***Figure 2.3:*** *The MQTT connection, specifying the QoS measure, followed by the publishing of a message to a broker. It is important to note that the QoS measure does not impact the underlying data transmission provided by a protocol such as TCP. In this example, client B publishes message B to topic A after connecting, and the broker publishes this message to client A as it previously subscribed to topic A.*

implementation chosen for this project allows the library to handle some of this logic instead. We justify our choice of QUIC library and compare existing implementations in chapter 3.3.1.

## 2.4 Hardware constrained protocols

We now motivate our focus on MQTT using QUIC by comparing this approach to other popular IoT protocols.

The Constrained Restful Environments (CoRE) working group (CoRE 2016) develops a framework for REST applications that run on constrained devices. The primary protocol that CoRE works on is the Constrained Application Protocol (CoAP) (Shelby et al. 2014). CoAP is a specialised transfer protocol designed to work on constrained devices. CoAP emphasises a generic design that supports low overhead, machine-to-machine interaction, multicast and asynchronous messaging. As CoAP is seen as the successor to HTTP for hardware constrained devices, we do not focus on a comparison to HTTP. Other popular IoT protocols primarily consist of proprietary protocols that are phasing out due to the webs standardisation of REST. Hence, our primary

point of comparison to MQTT with QUIC is CoAP.

CoAP works by providing a usual request/response model between endpoints that mimics the style of HTTP requests while using UDP at the transport layer. In contrast to HTTP, CoAP relies on datagram-oriented UDP communication as its transport to support some of these features. The abstraction that CoAP provides is a UDP layer that deals with asynchrony and a request/response layer that specifies interactions. The header structure for CoAP and MQTT is very similar, with the significant difference being that MQTT needs an ordered reliable packet delivery mechanism. In the past, this restricted MQTT to the TCP/TLS stack; however, with the introduction of QUIC, we can see further similarities between CoAP and MQTT using QUIC with both protocols providing data transfer using UDP. A report on the active deployment of data transport protocols T-Mobile (2018) showed that 62.61% of deployments rely on MQTT as one of their protocols. In contrast, CoAP was the most widely used UDP based protocol, with 22.49% of deployments relying on it as one of their protocols. This is partly because the class of constrained devices that CoAP targets does not cover the entire range of constrained IoT devices. The authors show that MQTT is more suitable for less constrained devices with long-standing connections; however, it may not be suitable for relatively more constrained devices.

In this work, we have focused on providing MQTT with similar advantages that CoAP enjoys due to UDP and analysing if MQTT can be made more efficient to capture the range of more demanding hardware constrained devices. Capturing a more comprehensive range of constrained devices using MQTT would increase the number of deployments that can take advantage of its feature set. Additionally, our choice stems not only from MQTT's popularity but also due to the exciting prospect of solving its historical dependency on TCP/TLS. This is further reinforced by CoAP showing a similar approach and being widely deployed for hardware constrained devices.

## 2.5 Rust programming language

Rust is a modern systems programming language created at Mozilla designed to be highly safe and performant. It is a systems language that aims to maintain the performance that we expect from languages like C while also using a unique *ownership* system to maintain memory safety. Instead of garbage collection, Rust opts for a system managed through the resource acquisition is initialisation (RAII) principle (Rust 2021). All values have a unique owner, and their scope is tied to this owner. Hence, by design, Rust does not allow dangling pointers, null pointers, and data races as the compiler will not allow for a programmer to compile unsafe code without circumventing it using the *unsafe* keyword.

For example, consider the program written in C in Listing 2.1 compared to a similar application in Rust in Listing 2.2. The C program demonstrates a use after free bug. The pointer is freed and then used in the print statement, resulting in undefined behaviour. While this example is relatively trivial, these bugs are often tough to debug in a more extensive, more complex system, leading to security vulnerabilities. This issue does not only exist in codebases and organisations with low resources; at the BlueHat security conference, Microsoft researcher Miller (2019) presented that Microsoft targets roughly 70% of their yearly patches at fixing memory safety bugs. On the other hand, the analogous Rust code will not compile due to the ownership system, mitigating this issue altogether.

```c
void bar() {
   int *ptr = (Point *) malloc(sizeof(int));
   free (ptr);
   printf("%d", *ptr); // obvious use after free, however this will compile
}
```

***Listing 2.1:*** *An example of a use after free in C code. This is an incorrect use of dynamic memory management, however it compiles. In large, complex code bases, missing bugs such as these often happens and can cause exploitable security issues.*

```rust
fn bar() {
   let example = String::from("Example");
   let mut example_ref = &example;
   {
      let new_example = String::from("New Example");
      example_ref = &new_example;
   }
   println!("our string is {}", &example_ref); // causes a compiler error in
       Rust: error ‘new_example‘ does not live long enough
}
```

***Listing 2.2:*** *A similar application in Rust will not compile due to the safety guaranteed by the ownership system. A borrow occurs when* `example_ref` *is assigned to point to* `new_example`*, however the ownership system recognises that the borrowed value does not live long enough.*

When it comes to networked systems in the IoT space, Rust is a natural application due to its focus on concurrency and safe systems programming. Firstly, a memory-safe language may circumvent security-related bugs in IoT firmware and the supporting network stacks. As previously discussed, getting the firmware correct on the first try is essential in IoT due to the difficulty of providing updates. However, assessing the performance of Rust implementations of the QUIC stack is vital to solidifying Rust as a performant systems programming language for hardware constrained devices. If the binary sizes produced by the Rust implementations are larger than their C equivalents or if the code is not as performant, then the memory safety guarantees may not matter for IoT developers.

# 3 Requirements and design

The following section presents the identified objectives and the methodology we follow for the implementation of *MQuicTT*. We discuss the steps we have taken to analyse the existing QUIC and MQTT implementations to motivate our choice of libraries. Additionally, we describe the design choices taken.

## 3.1 Requirements

QUIC theoretically presents benefits in terms of faster connection times and resilience to packet loss. IoT devices that save energy by turning themselves off or disconnecting from the network when they are unused can benefit from faster reconnection times. Additionally, higher resilience to packet loss can be beneficial for IoT deployments where many devices on data links may cause congestion. However, due to the introduction of streams and other features that provide these performance guarantees, QUIC is also a relatively complex protocol. Hence, we must analyse if this complexity translates into challenges for hardware constrained devices.

As previously discussed, IoT devices often carry sensitive data; hence, we must also consider that communication should be safe and secure. This requirement leads us to two choices. Firstly, we only consider communication that happens over an encrypted channel, that is, a network connection secured by TLS. This also gives us a chance to test QUIC's claim of faster secure connections due to a less complex handshake. Secondly, we must consider the programming language that we use for the implementation. Modern systems programming is dominated by C; however, the weak type system of C leads to many vulnerabilities stemming from bad memory management and race conditions. This means that insecure firmware may be shipped from the manufacturer and deployed when an IoT system is installed. The issue of such bugs in systems languages has led to the creation of the Rust programming language, which uses an ownership memory management system to mitigate these bugs. However, Rust is a relatively new programming language and does not yet benefit from the years of optimisations and ecosystem support that C has. Rust also opts to use a dependency manager that packages external dependencies into the binary of the program, which may have an effect on the binary size of the implementation. Hence, it is still important to contrast Rust with other languages.

We focus on the MQTT protocol as it is the most popular application layer protocol in the IoT space (T-Mobile 2018). MQTT is also beneficial for the analysis due to it being lightweight and having a low code footprint. This means that the implementation of MQTT itself should not be a bottleneck for hardware constrained devices. Hence, with the high-level goal of analysing the viability of a safe implementation of QUIC as a transport layer protocol for IoT devices, we must create an implementation that can be used as a test bench for such an analysis. This implementation will be a QUIC-based implementation of the MQTT protocol in a safe programming language.

We must then analyse the resulting implementation in terms of network performance and the constraints of IoT devices. For network performance, we must evaluate if the QUIC-based implementation is at least as performant as its TCP counterpart. In terms of hardware constraints, we have chosen to focus on the storage constrained in IoT devices. We focus specifically on storage as we predict that this is where our implementation will see the biggest challenge. The

requirements of using TLS, as well as the relatively untested binary sizes produced by Rust, mean that the implementation could be too large to act as firmware for devices with extreme hardware constraints.

Hence, from these high-level requirements, to analyse QUIC as a transport layer protocol for IoT, we have identified the following specific requirements using a MoSCoW style analysis:

- Must:
  - We **must** create a test case for MQTT using QUIC - MQuicTT.
  - We **must** analyse the performance of MQuicTT compared to base MQTT implementations.
  - We **must** focus on realistic testing scenarios for evaluation.
- Should:
  - We **should** analyse a general methodology for reducing the hardware footprint of transport layer protocols for IoT.
  - We **should** analyse the hardware footprint and feature set of the chosen QUIC and TLS libraries.
- Could:
  - We **could** create a QUIC extension that uses a lightweight security protocol.

The creation of *MQuicTT* is not only important to be able to evaluate the usage of QUIC for IoT, but also to test the viability of porting an MQTT implementation to QUIC in Rust.

We treat the base version of MQTT as the standard for comparisons as this will be an industry used TCP implementation of the protocol. A comparison to this will be made in terms of network performance and storage use. We have chosen to focus on the storage constraint of IoT hardware as this is often the hardest requirement to fulfil. Storage also often dictates the other hardware specifications for the device as it usually takes up most physical space.

As previously discussed, we analyse the viability of creating a general methodology for reducing the QUIC stack for IoT devices following the methodology of Eggert (2020). To do this, we must also analyse the storage footprint of each feature of the transport layer stack. By features, we mean the features as described in the protocol's respective RFC. For QUIC, for example, this could be the 0-RTT or connection migration features. If any of these are heavy in terms of required storage space and not necessary for IoT devices, we could extract them from the implementation.

Finally, we hypothesise that TLS takes a lot of storage space and hardware, again, based on the analysis performed by Eggert (2020). Hence, it could be possible to create a QUIC extension with a lightweight security protocol such as KP-ABE. Such an extension, however, may be out of the scope of this work.

From these requirements, we have split the project into the following objectives:

- **O1**: Compare existing QUIC and MQTT implementations that can act as underlying libraries.
- **O2**: Create an abstraction layer for the QUIC specific logic.
- **O3**: Implement MQuicTT by changing the underlying MQTT implementation to use this abstraction layer.
- **O4**: Create a test bench for the network performance analysis of MQuicTT.
- **O5**: Devise a methodology for trimming down the QUIC stack in terms of binary size.
- **O6**: Analyse the components of the MQuicTT binary.

**O1** focuses on finding implementations that can be used as the underlying libraries for *MQuicTT*. As discussed in Section 3.3, we have opted to use existing implementations; hence we must

compare them based on our criteria of performant, safe implementations that fit IoT storage constraints.

We have also opted to create an intermediate layer to encapsulate the QUIC protocol logic in the implementation. Hence, **O2** focuses on creating this abstraction.

**O3** describes the core of the implementation of *MQuicTT*. Using the libraries identified in **O1** and the abstraction created in **O2** we must create a port of the chosen MQTT implementation using QUIC.

We then move to the evaluation stage of the project; hence, creating a test bench in **O4** will be crucial for the network performance portion of this evaluation.

In **O5** we then focus on the storage analysis of the resulting implementation and devise methods for trimming down the QUIC stack. We also analyse the components of the resulting binaries in **O6** to find where the contributions to the binary size come from exactly.

Hence, the rest of this chapter is split into sections corresponding to the design for the implementation centred objectives, that is, **O1**, **O2**, **O3**. The rest of the objectives are addressed in Chapter 5.

## 3.2 Design of the implementation

In this section, we present a comparison of existing QUIC and MQTT implementations that can be used as underlying libraries for MQuicTT.Additionally, we discuss the high-level design of MQuicTT and the approach we have taken when it comes to using existing libraries compared to creating new protocol implementations.

## 3.3 Overall MQuicTT design

We first analyse what is required to create *MQuicTT* based on the established requirements and objectives.

We opt to use existing implementations of QUIC and MQTT. An alternate approach to this would be creating implementations from scratch that could be optimised for our use case. However, we demonstrate an actual use case by opting to use existing libraries. These libraries have both been built for production use and are relatively widely deployed. Hence, we are more likely to get accurate results. In addition to this, the development time needed to create implementations for these protocols from scratch would far exceed this project's scope.

Secondly, we must create an intermediate library that will act as an abstraction from the QUIC logic - *QuicSocket*. By creating this intermediate library and ensuring that its API mimics a standard TCP socket, we lower the difficulty of porting *rumqtt* to QUIC. Additionally, this library can be used for any application that uses the QUIC protocol. The key consideration when creating this abstraction is how we want to use QUIC streams.

To recap, QUIC streams are an encapsulation around the data transfer portion of the QUIC protocol. After establishing a connection, QUIC opens streams in order to send and receive data. There can be an arbitrary number of streams on a single UDP connection using QUIC. Hence, QUIC provides advantages by removing head-of-line blocking as different logical separations of data can be sent on different streams. For example, we can opt to send an HTML file and a stylesheet for a website on different streams so that if a packet of the stylesheet gets lost, the HTML can still be rendered.

We can also use this strategy to improve the performance of *MQuicTT* by choosing an appropriate encapsulation. There are two main possibilities here. Firstly, we may opt to send a single MQTT

connection on a QUIC stream. Secondly, we may opt to treat every subscription to a broker as a separate QUIC stream. The specifics of our choice and a discussion on this topic can be found in Chapter 4.

Hence, the design for the implementation has been split into the following stages. Firstly, we must find appropriate implementations of QUIC and MQTT based on our criteria. Although we opt to use a Rust implementation, it is still important to consider other systems-language based implementations of QUIC. If the Rust implementations of the protocol stand out in terms of storage consumption compared to, for example, C++, this would indicate that Rust is perhaps less suitable for this application. Secondly, we must create an abstraction around the chosen QUIC implementation that acts as a socket for the QUIC protocol. This abstraction must have the functionality required to securely transfer data using QUIC. That is, we must be able to work with TLS keys, establish a secure connection and send and receive data using streams. This library must also have an API similar to that of the TCP sockets API for external consistency. This means that we must have functions that take in buffers, write or read from those buffers, and return the number of bytes that were read or written. Adhering to this API will also simplify the port. Lastly, we must use the resulting socket library to port the chosen MQTT implementation to QUIC.

Further discussion of specific technical choices that we have made during the implementation can be found in Chapter 4.

### 3.3.1 QUIC

We now compare the existing implementations of the QUIC protocol in the context of usability for IoT devices and present the reasoning behind our choice of QUIC library. We have considered the mainstream implementations in the C, C++, Rust and Go programming languages as these are the languages that can be considered systems languages and would thus be the most widely used ones for IoT devices. In addition to this, we did not consider implementations paired with a browser web engine as these would be impossible to use on hardware constrained devices. Hence, we did not include notable implementations such as the QUIC implementation of the chromium web engine (Chromium 2021) and *Neqo* (Mozilla 2022).

Importantly, to analyse the performance of underlying QUIC implementations, we have transferred the same file using each implementation several times and have taken the average time to do so. We repeated this while altering parameters as described in Section 5.1.1. The differences in transfer time were negligible and have hence not been presented as part of the comparison of libraries. Hence, we compare QUIC implementations in terms of their TLS use, programming language and the binary size produced by their client and server implementations.

First, we consider the general landscape of available QUIC implementations to contextualise the ones developed in Rust. Table 3.1 demonstrates the analysed QUIC implementations. In order to find the size of the binary, we have used the Linux *ls* utility. In the case of the *mvfst* implementation, we have taken further steps due to the reported binary size being far too large. Therefore, we had to remove the C++ debug symbols that were causing the binary size to be over 300 MiB. We have identified the following main methods by which QUIC implementations incorporate TLS:

- Use of an external library - the implementation uses an external TLS API either by using a package manager in the case of Rust or by relying on an installed implementation in the case of C and C++.
- Use of own implementation - the implementation packages its implementation of the TLS protocol alongside QUIC.

This different use of TLS presented a challenge in calculating the binary size of the QUIC servers and clients.

***Table 3.1:*** *The identified QUIC implementations and their binary size footprint. The footprint has been split into a client and server footprint using a minimal reproducible example for each. We used each implementation to create an example client and server capable of sending and receiving QUIC packets and analysed the binary size. Where provided, we compared this to the given examples to ensure as little implementation bias as possible. However, this is still not a perfect estimate, and some variance due to implementation details may be present.*

| Implementation | PL | Footprint (client) (MiB) | Footprint (server) (MiB) | TLS method |
|---|---|---|---|---|
| ngtcp2 | C | 3.4 | 4.3 | External |
| picoquic | C | 3.2 | 3.9 | External |
| msquic | C | 3.2 | 4.1 | External |
| quic-go | Go | 8.7 | 9.9 | External |
| Quinn | Rust | 9.1 | 9.5 | External |
| Quiche | Rust | 7.8 | 7.0 | Own |
| mvfst | C++ | 11.1 | 12.0 | Own |

Specifically, in each case, we have considered the external dependencies that a developer will have to install to run the QUIC implementation on a device. Hence, in the case of C implementations that require an external TLS library, such as OpenSSL, to be installed and linked on the system, we have opted to add the binary size produced by OpenSSL to the size of the QUIC binaries. Additionally, in the case of *picoquic*, we have added the size of the *picotls* dependency on top of the size of OpenSSL.

Figure 3.1 further visualises the comparison of binary sizes between the various implementations. This is an important aspect due to the aforementioned hardware constraints. Notably, we can see that five out of the seven analysed implementations opt to use an external TLS library or engine. Out of these five, all C implementations supported OpenSSL, with the Go and Rust implementations opting to use a TLS library. In the case of *quic – go*, this was the *crypto/tls* package, and in the case of Rust, *rustls*. We can also see that the Rust implementations are not drastically different in binary footprint size.

Notably, although Go is described as memory safe, it does not opt for compile-time memory safety and instead uses the panic model. The panic model is another advantage of Rust compared to languages such as Go, as Rust provides these guarantees using a robust type system. Between the two Rust implementations - Quinn and Quiche, we chose Quinn due to issues with the Quiche library. On the other hand, Quiche opts to make the user create a *mio* event loop, which interfered with the *tokio* runtime environment used in our chosen MQTT implementation discussed in the next section. In addition to this, we found that the Quinn API is easier to work with when creating the intermediate library discussed in Chapter 4.1; for example, Quinn handles the QUIC handshake in the library and does not require the developer to create an event loop.

### 3.3.2 MQTT

Compared to QUIC, the choice of MQTT implementation was substantially simpler. The criteria for MQTT implementation were that it was developed in Rust, implemented both a client and a broker, had widespread use and adhered to MQTT version 5.0. Based on the above criteria we identified two possible implementations: Eclipse's (2018) *paho* and *rumqtt* (Bytebeam 2020). We opted to use *rumqtt* at it is a native Rust implementation, whereas *paho* provides a rust binding to an underlying C implementation. We chose to evaluate a fully Rust native MQTT/QUIC stack as this provides an opportunity for a valuable comparison to mainstream C implementations. Other available implementations supported only one side of the MQTT protocol or only supported version 3.1.1.

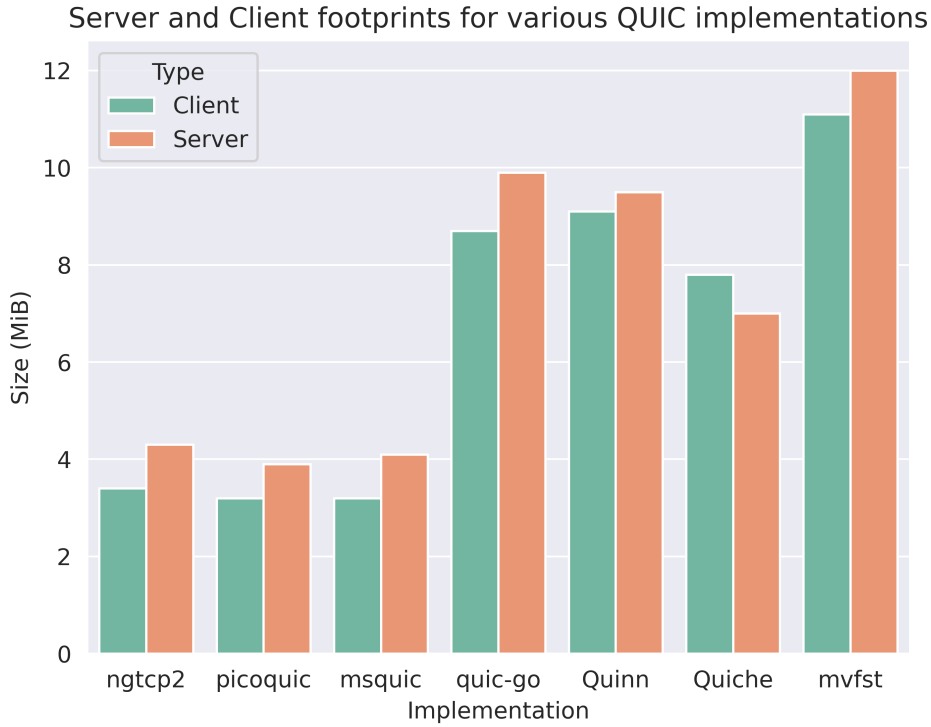



***Figure 3.1:*** *The sizes of the client and server footprints for the selected QUIC implementations. Notable, only Quiche produced a server binary with a smaller size than its corresponding client binary. It is hard to estimate the error margin for the data as this depends on implementation details in the example client and servers.*

*Rumqtt* provides to components for an MQTT application – *rumqttc* and *rumqttd*. The former can be used to create an MQTT client, and the latter a broker. However, the code base for these is somewhat similar, easing the incorporation of QUIC. Both components provide an interface for supporting asynchronous communication using a *tokio* runtime, which fits nicely into our choice of QUIC implementation as *Quinn* requires a *tokio* environment. By default, *rumqtt* uses TCP as its transport layer protocol and TLS through *rustls*. All underlying implementation assumptions remain equal because this is the same library as *Quinn* uses.

# 4 Implementation

We now present the implementation for the intermediate QUIC library that we created and MQuicTT. We discuss the specific implementation choices that we have made during development and present example uses of each implementation.

## 4.1 QuicSocket

This section will present *QuicSocket* [1] - the intermediate wrapper API that we developed using *Quinn* to facilitate establishing a QUIC connection, control streams, and send and receive data.

*QuicSocket* provides two main types to communicate with *Quinn* - a *QuicServer* and a *QuicClient*, and a trait that both of these implement - *QuicSocket*. The *QuicSocket* trait defines three function: *new*, *send* and *recieve*. When initialised using the *new* function, the server and the client attempt to open a QUIC connection. In the case of the server, it binds to a given port and starts listening for a connection request. The client binds to a random available port and attempts to connect to a provided URL. Additionally, in the *new* function, the server generates a new TLS certificate or reads it from a specified path. The client's *new* function must be supplied with the corresponding certificate's path. If either certificate is not present, the server will reject the connection attempt. Once a connection has been opened, data transfer can happen.

```
async fn send(&mut self, payload: Vec<u8>) -> Result<()> {
   let (mut send, _) = self
      .connection
      .open_bi()
      .await
      .map_err(|e| anyhow!("failed to open stream: {}", e))
      .unwrap();
   send.write_all(&payload)
      .await
      .map_err(|e| anyhow!("failed to send request: {}", e))?;
   send.finish()
      .await
      .map_err(|e| anyhow!("failed to shutdown stream: {}", e))?;
   Ok(())
}
```

***Listing 4.1:*** *The send function that the QuicClient and QuicServer use for sending data. Data transfer is facilitated by opening a bidirectional stream on the pre-existing connection.*

In order to send data, either side can use the *send* function demonstrated in Listing 4.1. We first open a bidirectional stream and then write the provided payload. Once the client completes writing the payload, it shuts down the stream. On the other side, the receive function in Listing 4.2

[1] QuicSocket - `https://github.com/Apolexian/QuicSocket`

```rust
async fn recv(&mut self, buf: &mut [u8]) -> Result<usize> {
    let (_, mut recv) = self.bi_streams.next().await.unwrap().unwrap();
    let len = recv
        .read(buf)
        .await
        .map_err(|e| anyhow!("failed to read response: {}", e))?;
    Ok(len.unwrap())
}
```

***Listing 4.2:*** *The recv function that the QuicClient and QuicServer use for sending data.*

processes the next available QUIC stream and reads the data on it into a supplied buffer. It then returns the number of bytes written to the buffer. Hence, to use *QuicSocket*, we must either provide or generate TLS certificates, open a QUIC connection and then use the send and receive functions. The developed API is analogous to how a TCP socket sends and receives data. An alternative approach that we considered was to make the *recv* function return the value that the client or server read from the stream. However, this approach does not integrate well into any protocol implementation due to the general preference for the buffer pattern.

Error handling is handled via the *anyhow* package throughout the library for idiomatic error handling. In addition to this, we wrap return values in Rust's *Result* construct. Hence, all errors are idiomatically mapped wherever a failure can occur.

An important design choice that we have made when developing *QuicSocket* is using QUIC streams. Every call to send and receive in the current implementation creates a bidirectional stream. An alternate choice was to open a stream at the start of the connection and use it until the connection closes. However, the latter approach runs into issues when it comes to reading from the stream. All data was read on one call to receive, which unnaturally coalesced MQTT packets.

Notably, all operations happen asynchronously. However, as of the implementation time, the Rust standard library does not support asynchronous traits. The workaround to this issue is using the *async – trait* crate. An unfortunate drawback of using this crate is that every function call results in a heap allocation due to the crate's implementation semantics. The extra heap allocations do not usually present an issue; however, in the case of hardware constrained devices, this may lead to a bottleneck depending on the usage of the library. The reasons for not supporting asynchronous traits in Rust are somewhat complex. For one, an asynchronous function in Rust returns an *implFuture*, meaning that an asynchronous trait would have to support returning *implTrait*, which it does not. However, the *async – trait* crate instead returns a *dynFuture*, the Rust implementation of dynamic dispatch, resulting in a heap allocation but allowing it to be used inside of traits. We will not further discuss the other reasons for this crate's existence; however, Matsakis (2019) created a comprehensive analysis of this issue. Once Rust incorporates asynchronous traits into the language, the heap allocation issue will be resolved, resulting in this problem no longer existing for hardware constrained devices.

In addition to the QUIC specific logic, our goal was also for all communication to happen securely. Hence, *QuicSocket* does not accept connection requests without correct TLS keys. To make working with this easier, we have created functions that handle the creation and reading of TLS keys.

```rust
#[allow(unused)]
#[allow(clippy::field_reassign_with_default)] //
    https://github.com/rust-lang/rust-clippy/issues/6527
pub fn gen_certificates() -> Result<(), Box<dyn Error>> {
   let cert =
       rcgen::generate_simple_self_signed(vec!["localhost".into()]).unwrap();
   let cert_der = cert.serialize_der().unwrap();
   fs::write("./cert.der".to_string(), &cert_der).unwrap();
   let priv_key = cert.serialize_private_key_der();
   fs::write("./key.der".to_string(), &priv_key).unwrap();
   let key = rustls::PrivateKey(priv_key);
   let cert = vec![rustls::Certificate(cert_der.clone())];
   let mut server_crypto = rustls::ServerConfig::builder()
      .with_safe_defaults()
      .with_no_client_auth()
      .with_single_cert(cert, key)?;
   server_crypto.alpn_protocols = ALPN_QUIC_HTTP.iter().map(|&x|
       x.into()).collect();
   Ok(())
}
```

***Listing 4.3:*** *The gen$_c$ertificates function that can be used to generate appropriate TLS certificates.*

Listing 4.3 shows the source code of the function that can be used to generate TLS certificates. The certificates are save to corresponding "key" and "certificate" files and can be distributed to the client and server. Both the client and server expect certificates to be present when they are run.

## 4.2 MQuicTT

This section describes how we used *QuicSocket* to create a QUIC port of *rumqtt*, the design decisions that we have taken, and possible alternate approaches.

As previously mentioned, *rumqtt* provides two APIs: *rumqttc* for creating MQTT clients and *rumqttd* for creating brokers. Hence, each of these APIs relies on an underlying TCP implementation to transmit data. A configuration file is also required for *rumqtdd* that specifies various parameters such as paths to TLS keys and the ports that MQTT will operate on. Additionally, *rumqtt* implements its own layer of TLS to ensure secure communication on top of the TLS layer provided by the underlying transport protocol.

To create *MQuicTT*, we have taken steps that can broadly be summarised in two phases: altering the default TLS code and changing the underlying transport protocol from TCP to QUIC.

As we are working in the domain of hardware constrained devices, providing multiple layers of encryption would be excessive, even for the goal of secure communication. We can afford for only the transport layer to handle encryption. Hence, the first step that we have taken is to remove all TLS functionality from both *rumqttd* and *rumqttc*.

After this, we identified that both APIs have a central interface that controls network communication. In the case of *rumqttc* this is the *network* struct inside *framed.rs* [2] and in the case of *rumqttd* it is the *network* struct inside *network.rs* [3]. Both of these interfaces use a *tcpstream* to send and receive data. The *tcpstream* is opened when the struct is initialised and closed when

[2] framed.rs - `https://github.com/bytebeamio/rumqtt/blob/master/rumqttc/src/framed.rs`
[3] network.rs - `https://github.com/bytebeamio/rumqtt/blob/master/rumqttd/src/network.rs`

the MQTT connection ends. As TCP is a stateful protocol, minimal connection handling occurs after opening the initial stream.

```
#[tokio::main(worker_threads = 1)]
async fn main() {
   ...
   let quic_client = QuicClient::new(None).await;
   let mut mqttoptions = MqttOptions::new("test-1", ip, 1883, addr, quic_client);
   mqttoptions.set_connection_timeout(10);
   mqttoptions.set_keep_alive(5);

   let (client, mut eventloop) = AsyncClient::new(mqttoptions, 100);
   task::spawn(async move {
      requests(client).await;
   });
}
```

***Listing 4.4:*** *An example of initialising an MQuicTT client. The QUIC connection is established by initialising a QuicClient and the resulting client is passed as an MQTT option.*

As we have modelled *QuicSocket* to have an API that closely resembles a typical *tcpstream* API, the challenge of this stage of the port was managing QUIC streams and the state of connection. Due to UDP, the protocol that underlyes QUIC, being stateless, we have handled the connection differently in brokers and clients.

The connection request is sent before an MQTT client object is initialised in the client's case. Upon successful connection, the programmer must pass the connection to the initialisation function of the client. This means that a single QUIC connection underpins the client's entire MQTT connection, and streams are used for packets. Hence, when the client wishes to close the MQTT connection, it also closes the underlying QUIC connection. In order to do this we have modified the parameters needed to create an *rumqtt* MQTT client. An example of the initialisation of a QUIC connection and MQTT client using our implementation is demonstrated in Listing 4.4.

In the case of a broker, we must wait for a client to send a connection request. Hence, we may initialise the broker and wait for a QUIC connection, only allowing data transfer when any connection is established. This also means that the broker does not need to track which connection comes from which client.

```
fn main() {
   pretty_env_logger::init();
   let config: Config = confy::load_path("rumqttd.conf").unwrap();
   let mut broker = Broker::new(config);

   let mut tx = broker.link("localclient").unwrap();
   thread::spawn(move || {
      broker.start().unwrap();
   });
   ...
}
```

***Listing 4.5:*** *An example of initialising an MQuicTT broker. We can see that no operations with QuiSocket are required for this initialisation as all QUIC operations are handled internally.*

Hence, as we can see in Listing 4.5, the programmer is not required to use any *QuicSocket* code when creating an *MQuicTT* broker.

We considered an alternate approach to refactoring the *rumqtt* codebase to consolidate all network code into a shared internal package. This would mean that the API for creating a client and broker would be identical and perhaps more ergonomic. However, we decided not to go with this approach because *rumqttc* and *rumqttd* would no longer be disjoint, independent components. With the chosen approach, it is possible to use different underlying transport protocols for either side, which allows for flexibility in implementation.

Another avenue of discussion stems from *rumqtt* having two different client implementations: an asynchronous client and an asynchronous client. In either case, a *tokio* eventloop is used to handle events. As *QuicSocket* requires an asynchronous environment, the natural choice was to use the asynchronous client. It could, however, be possible to handle asynchronous events more efficiently by tying the QUIC eventloop to the MQTT eventloop as in the approach described by Kumar and Dezfouli (2019). In our case, we leave the comparison between these two approaches as an avenue for future work.

# 5 Analysis

This section presents the analysis results conducted using the previously described methodology. The analysis is broadly split into two stages: performance analysis and binary size analysis. The experiment design for the performance analysis can be found in Section 5.1.1, and the experiment design for the binary analysis can be found in Section 5.2.1. The performance analysis is further split into a connection time analysis and a general transmission time analysis.

## 5.1 Performance experiment

### 5.1.1 Network performance experiment design

We now discuss the analysis methodology for the performance portion of the implementation evaluation. That is, this section focuses on evaluating MQuicTT against the base *rumqtt* implementation.

The critical consideration for this design is the scenarios in which IoT devices are used. When evaluating the network performance of the implementations, we considered two options: using real IoT devices or using a network simulation tool. Due to technical limitations that came with using real devices, such as not being able to access the router of our network, we opted for simulation. In this section, we will discuss how we used Mininet (Lantz and Heller 2013), a realistic virtual network, in our evaluation.

Mininet is a tool that network developers and researchers can use to create software-defined networks (SDNs) using the *OpenFlow* standard.

Using the Python API provided, we created the network topology shown in Figure 5.1. The script takes several parameters to create the following three scenarios:

- A synthetic scenario testing the limits of implementations.
- A realistic scenario based on a smart-home use-case.
- A realistic scenario based on a 3D printer farm.

The topology for all three scenarios remains the same - a minimum spanning tree of a typical IoT mesh network. When designing this topology, we needed it to reflect various realistic IoT scenarios. This topology is designed to demonstrate network congestion and fits all three scenarios; hence we do not need to change it. We can imagine this as a singular room in a home with all of its smart appliances being a cluster in the smart home or a cluster of 3D printers in a section of a farm in the 3D printer farm. The critical feature of this topology is that many clusters are connected to a central broker that controls the topology, hence creating a dumbbell structure. The full definition of this topology can be found in Appendix B. The variables that the script changes between scenarios and simulations are the link's *bandwidth*, *delay* and the rate of *packet loss*.

The bandwidth of a link is the maximum rate of data transfer we can achieve. In contrast to bandwidth in signal processing, in computer networking, we measure bandwidth in bits per second rather than hertz. The delay of a link specifies the latency of the link, i.e. the time that a

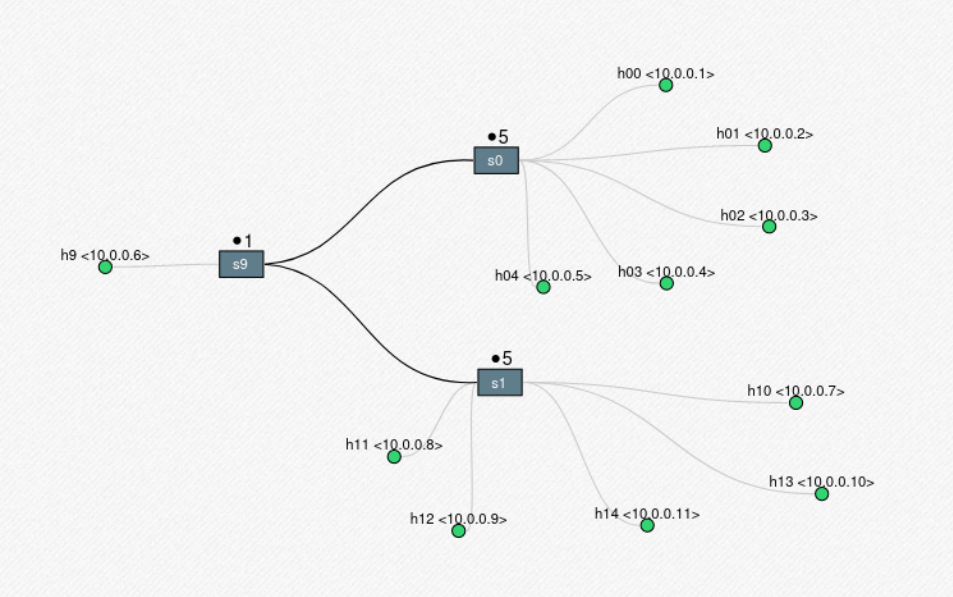



***Figure 5.1:*** *The resulting Dumbbell topology with the broker being h9 - the central node. The link between the central switch and the central host results in a congested network.*

bit of data takes to travel across a link. We measure this in milliseconds or seconds. Link delay corresponds to the geographical distance between the communicating parties; however, in the case of IoT, we can expect devices to be in local proximity. Lastly, the packet loss rate shows the percentage of corrupted or dropped packets in transit. Various protocols having to retransmit packets also adds to the delay of data transfer. Importantly, we have only considered the typical circumstances of packet loss and have not included scenarios such as interference or packet loss attacks.

The bandwidth and delay numbers correspond, as closely as possible, to various link types in a network. To do so, for the smart-home scenario, we have gathered data from the Ofcom (2021) report on UK broadband speeds. There were specific cases in which it was not possible to find this data in the report; hence it was augmented using a similar methodology in work conducted by Previdi et al. (2019) and in the case of ZigBee, the work by Alena et al. (2011).

***Table 5.1:*** *The parameters chosen for each link simulation in Mininet in the smart home scenario. The types of links were chosen as the most commonly occurring ones in IoT use cases.*

| Simulated Link Type | Link bandwidth (Mb/s) | Link delay (ms) | Packet loss rate (%) |
|---|---|---|---|
| Wi-Fi | 30 | 10 | 2 |
| ZigBee | 0.25 | 5 | 1 |
| 4G | 4 | 20 | 1.5 |
| 3G | 1 | 40 | 1.5 |
| 100Mb Ethernet | 100 | 1 | 0.2 |

Hence, we expect that the bandwidth and link delay numbers accurately represent a real-world scenario. However, it was complicated to find exact estimates for packet loss rates, with most sources describing approximations for a stable connection (SDU 2013) and not precise measurements. Hence, the data are best estimates, cross-validated through the different sources and are not exact values.

In the case of the smart home scenario, as presented in Table 5.1, we expect that our packet loss rates are accurate as these, as previously stated, do appear in the home broadband reports and other studies. In the case of the 3D printer, as presented in Table 5.2 farm scenario, due to the number of machines and the interference these cause in a 3D printer farm, we modelled this scenario to have more packet loss. We specifically chose this scenario to test the benefits that QUIC should receive in an environment with high packet loss. However, we found it difficult to find exact data on packet loss percentages in smart factories or workshops; hence, we expect a margin of error on some of the values.

In general, we can assume that the packet loss rates will increase by some constant factor across all links. Hence, we have also created a synthetic scenario with extreme packet loss as presented in Table 5.3. If the protocol performs well for an extreme scenario, we can expect it to also perform well for packet loss percentages up to that scenario.

***Table 5.2:*** *The parameters chosen for each link simulation in Mininet in the 3D printer farm scenario. The data assumes a typical IoT setup where most devices are within local geographical proximity. That is, the devices are communicating with each other within the range of one factory or site, with only the central node communicating with some server.*

| Simulated Link Type | Link bandwidth (Mb/s) | Link delay (ms) | Packet loss rate (%) |
|---|---|---|---|
| Wi-Fi | 30 | 10 | 5 |
| ZigBee | 0.25 | 5 | 3 |
| 4G | 4 | 20 | 2.5 |
| 3G | 1 | 40 | 2.5 |
| 100Mb Ethernet | 100 | 1 | 0.5 |

***Table 5.3:*** *The parameters chosen for each link simulation in Mininet in the synthetic scenario. Here we opt for extreme packet loss scenarios to test the boundaries of the protocols. We opted for 20 times the normal packet loss that the link can expect in each case. We have also tested this for higher values however, all protocol implementations greatly suffered in performance beyond this.*

| Simulated Link Type | Link bandwidth (Mb/s) | Link delay (ms) | Packet loss rate (%) |
|---|---|---|---|
| Wi-Fi | 30 | 10 | 15 |
| ZigBee | 0.25 | 5 | 15 |
| 4G | 4 | 20 | 20 |
| 3G | 1 | 40 | 20 |
| 100Mb Ethernet | 100 | 1 | 4 |

We evaluate MQuicTT's performance using the presented topologies and respective simulation parameters in the following way. Each protocol transmits 100 messages from a client to a broker for each data link.

During this, we measure the following:

- The time taken for the underlying transport protocol to establish a connection.
- The time taken to transmit the message to the broker.

This is repeated several times to ensure that variance does not affect results. This process is repeated for all three testing scenarios.

In general, MQTT allows for messages with a maximum size of approximately 260MB. However, this is a huge message, and most publicly deployed brokers reject it, so we created a representative message for each use case. Each topic in MQTT consists of a hierarchy of topic levels separated by a forward slash. For example, in a smart home scenario, we may have a

topic like *home/groundfloor/kitchen/temp* to control the temperature in the kitchen via a smart thermostat. A topic may also include a wildcard. The topic string *home/groundfloor/+/temp* includes a *single-level* wildcard that will match an arbitrary string. This would match the topic *home/groundfloor/lounge/temp*, but not match the topic *home/secondfloor/kitchen/temp*. If a client wishes to subscribe to multiple topics with the same prefix, a *multi-level* wildcard may be used. For example, the topic *home/secondfloor/kitchen/#* can be used to subscribe to all topics with a prefix matching the string before the hash character. In particular, brokers reserve topics for system messages starting with the $ character.

The chosen topic and the transmitted message for each scenario can be found in Appendix A. The messages were devised based on an arbitrary command that may be issued to an IoT device present in the scenario. For the synthetic scenario, we opted to use a message that is considerably longer to test the limits of each implementation. This message simply consisted of a payload with a field replicated thousands of times. It has been omitted from the Appendix due to being too long.

### 5.1.2 Connection time comparison

In this section of the analysis, we focus on connection time. By connection time, we mean the time it takes for the underlying transport protocol to establish a secure connection. For this analysis, we have not considered connection re-establishment.

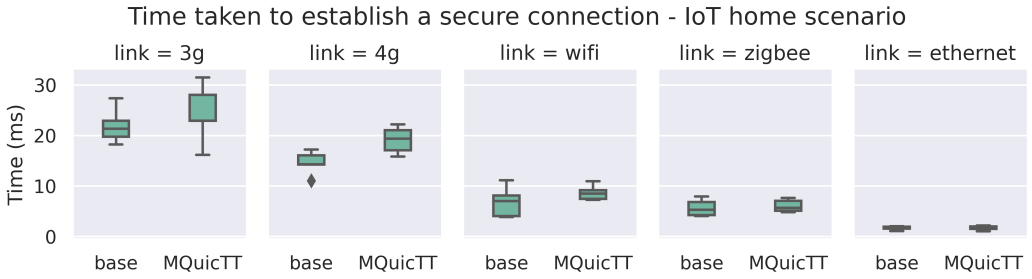
Connection time is essential in IoT devices as many do not keep a connection alive while idling. A device may opt not to be connected at all times to reduce energy consumption and computational power. The device will then establish a connection and transmit data whenever required. However, this means that some efficiency is lost if the connection has to be constantly re-established. Hence, MQTT defines the period to keep the connection alive as the *keep alive interval*. In an MQTT/TCP implementation, the standard way to extend the keep alive interval is to periodically send a *ping* packet, forcing the connection to remain open. However, it has been shown that this method can lead to security vulnerabilities (Vaccari et al. 2020; Mileva et al. 2021).

To measure the connection time, we have followed the previously described methodology. That is, we began a communication via MQuicTT and the other MQTT implementations and captured it using *tcpdump*. We have then extracted the time between the start of the QUIC or TCP communication and the handshake completion.

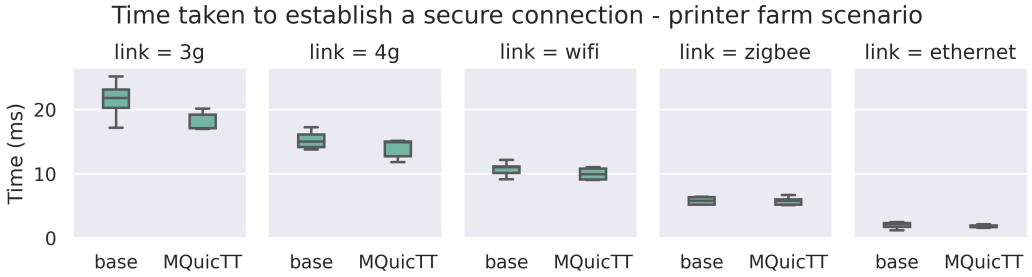
Figure 5.2a shows the results for the IoT home scenario. We can see that *MQuicTT* connects fractionally slower than the base implementation across the links on average, with the least overlap being on 3g and 4g links. However, when factoring in the variance, both implementations perform equally well. Notably, *MQuicTT* shows a higher variance in connection time, while the TCP implementation is consistent. This result can be attributed to the fact that QUIC's advantage comes in scenarios with packets loss, and the IoT home scenario does not implement high packet loss. QUIC should still theoretically connect faster due to its simpler handshake; however, the implementation may impose overhead that outweighs this.

On the other hand, in the printer farm scenario, as shown in Figure 5.2b, we can see that *MQuicTT*, on average, establishes a connection faster than the base implementation. When accounting for variance, we can see that the performance advantage is approximately on the scale of a single round trip time. This is consistent with the assumption that QUIC can handle packet loss better than the base TCP implementation. That is, the QUIC implementation likely did not need to retransmit packets as much as the base implementation, hence establishing a quicker connection.

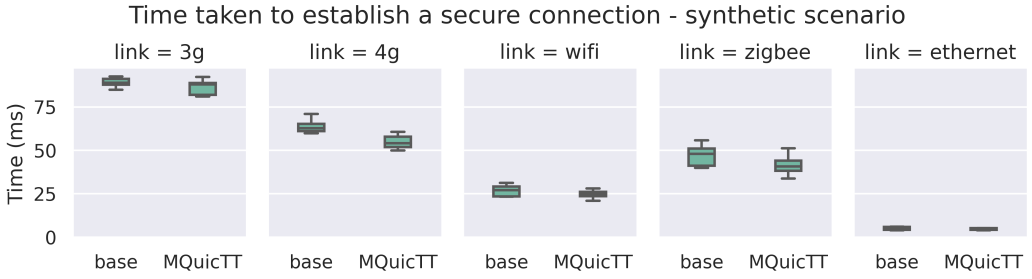
This result is supported by the data obtained from the synthetic scenario shown in Figure 5.2c. We can see that *MQuicTT* again, on average, establishes a connection quicker than the base




*(a) The time it took for the implementations to establish a secure connection in the IoT home scenario for each link.*




*(b) The time it took for the implementations to establish a secure connection in the printer farm scenario for each link.*




*(c) The time it took for the implementations to establish a secure connection in the printer farm scenario for each link.*

***Figure 5.2:*** *The time taken to establish a secure connection in their respective scenarios. We can see that MQuicTT on average performs better than the base TCP implementation in scenarios with high packet loss, however, it also experiences higher variance.*

implementation. This is expected as the packet loss here is extreme; hence, QUIC should perform better on average.

Overall, *MQuicTT* presents an advantage in terms of time needed to establish a connection in environments with high packet loss, however, is not advantageous for the IoT home scenario. These results support some of the theoretical advantages of QUIC; however, they also show that the benefit may not be as high as reported in the literature.

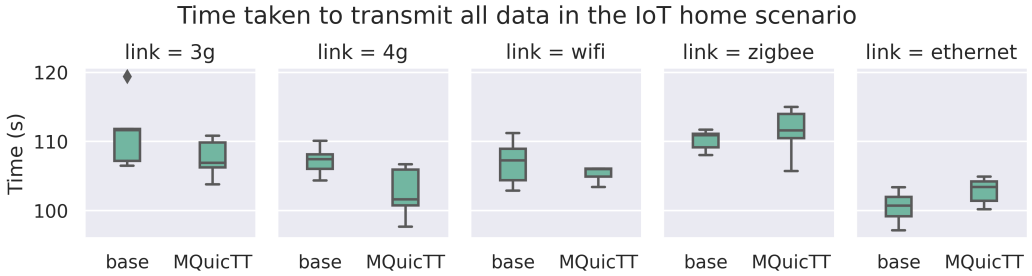
### 5.1.3 Data transmission comparison

We next evaluate the time it took both implementations to transmit the aforementioned MQTT messages. To do so, we have used the previously described methodology. That is, we have transmitted the MQTT messages shown in A for their respective scenario and have measured the total time taken for the transmission to occur.

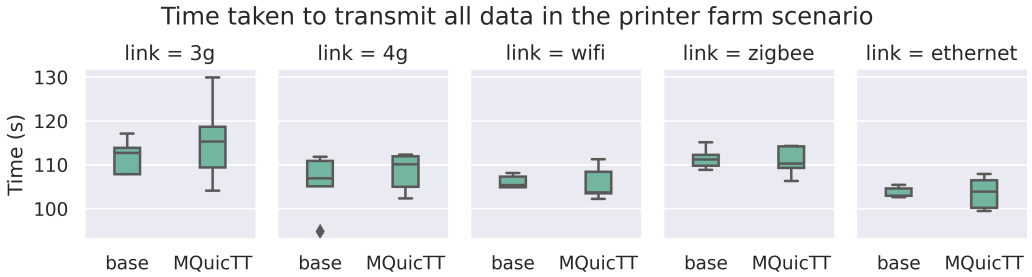
Figure 5.3a shows the results for the data transmission in the IoT home scenario. We can see that MQuicTT performs comparably to the base *rumqtt* implementation. *MQuicTT* performs better in the cases of 3g, 4g and WiFi and loses out to the base implementation in Zigbee and ethernet. The difference, however, is not substantial, and we can see from the variance that the implementations are comparable in performance. Notably, *MQuicTT* experiences higher variance, implying that it is more affected by changes in the simulation environment between tests, while the base implementation is more consistent. Overall, in this scenario, *MQuicTT* performs on par with the TCP implementation; however, it also presents no advantage in terms of performance.

Figure 5.3b shows the results for the printer farm scenario. As previously discussed, this scenario presents higher packet loss to simulate the kind of congestion a link may experience in an industrial IoT application. From the results, we can see that *MQuicTT* again performs on par with the base implementation. We can also see that *MQuicTT* experiences more variance in this scenario too. The theoretical advantage that the QUIC protocol may provide in environments with high packet loss does not seem to be demonstrated in this scenario. This is perhaps due to the packet loss not being extreme enough for QUIC to show an advantage. Overall, the implementations again perform comparably.

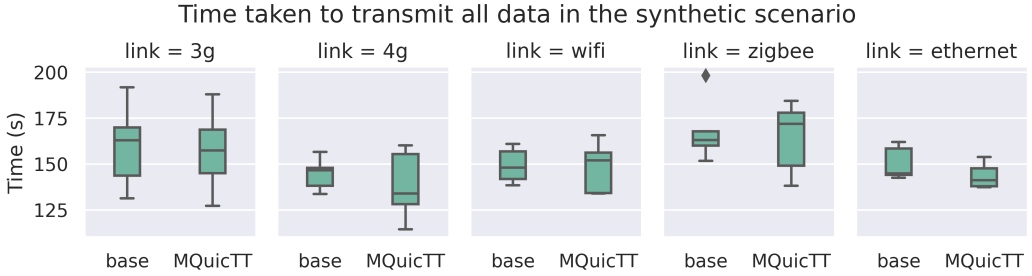
We now look at Figure 5.3c which shows the results for the synthetic scenario. That is, the scenario with extreme packet loss. The results of this scenario are similar to the previous ones. *MQuicTT* performs as well as the base *rumqtt*, with the results for ethernet showing a performance increase in favour of the QUIC implementation. Notably, we can see that in this scenario, both implementations experienced greater variance in measurements. Overall, *MQuicTT* performed as well as the base implementation in each scenario; however, it also presented no performance improvement. This conclusion is analogous to that of the data for connection time.




*(a) The time it took for the implementations to transmit the MQTT messages in the IoT home scenario for all links.*




*(b) The time it took for the implementations to transmit the MQTT messages in the printer farm scenario for all links. We can see that the results are similar to the IoT home scenario.*




*(c) The time it took for the implementations to transmit the MQTT messages in the printer farm scenario for all links.*

***Figure 5.3:*** *The time taken to transmit all MQTT messages in their respective scenarios. We can see that MQuicTT on average performs comparably to the base TCP implementation in scenarios with high packet loss, however, it also experiences higher variance.*

## 5.2 Binary size experiment

### 5.2.1 Binary size experiment design

This section describes how we have conducted further analysis on the binary size of the QUIC protocol that underlines MQuicTT. The focus of this section is to see if the QUIC stack contributes a significant overhead to MQuicTT's binary size and how we can reduce this overhead. We expect that the QUIC stack will contribute to the majority of the size of MQuicTT as MQTT itself is designed to have a low code size overhead. Additionally, we have conducted a similar analysis for the TLS library used and expect that it also contributes a significant amount to the resulting binary size. Hence, the first step of this part of the analysis will be to determine how much QUIC contributes to the binary size.

In order to get a breakdown of the binary we have used the *cargo – bloat* [1] utility. The utility analyses the binary using custom ELF, DWARF and Mach-O parsers and disassembles the binary to look for references and links to anonymous data. Doing so creates a map of the binary that shows where every byte has a label attached to it.

This utility provides the composition of a Rust binary. However, it is not perfect and results in some margin of error. Unfortunately, this margin of error is also not easily measurable. By comparing the total size of the binary as reported by *cargo – bloat* to the size reported by the operating system, we have deduced that the total error margin is within 1% with good precision. This should mean that we can get a somewhat accurate error margin on the components. However, it is also possible that the internal calculations are inaccurate despite the overall size being accurate.

The next step in this stage will be analysing methods for trimming down the QUIC stack binary size. Hence, in this stage, we shift our focus to the binary produced by *QuicSocket*. To reduce the binary size, we opt to use the method established by Eggert (2020) as recreating these steps may show a general framework for reducing binary sizes for hardware constrained devices. Notably, our application already handles client and server code separately; the MQTT broker requires a different binary to the MQTT client.

Hence, the steps we take are as follows:

- Compile the binary for a 32-bit target by setting the *target* flag in cargo to *i*686*–unknown–linux – gnu*.
- Remove any error handling code beyond what is needed for the binary to compile.
- Remove any code that writes to standard output.

After every step, we record the difference in binary size made by the change using the same methodology.

Once this step is completed, we further analyse the size of *Quinn* and *Rustls* using a by-function binary size breakdown. Using the *cargo – bloat* utility, we can get a list of the contribution of each function to the binary size and then assign each function to its respective protocol feature.

### 5.2.2 MQuicTT binary composition

We first look at an in-depth breakdown of the MQuicTT binary and analyse the contribution to the QUIC stack. Figure 5.4 shows the result of the binary breakdown by crate.

When analysing the QUIC stack contribution to the binary size, we can see that *quinn* contributes to 17.8% of the binary size and *rustls* contributes a further 12.6%. Additionally, we can see that the *ring* crate contributes another 5.8%. This crate is a Rust binding for BoringSSL primitives;

[1] cargo-bloat - `https://lib.rs/crates/cargo-bloat`

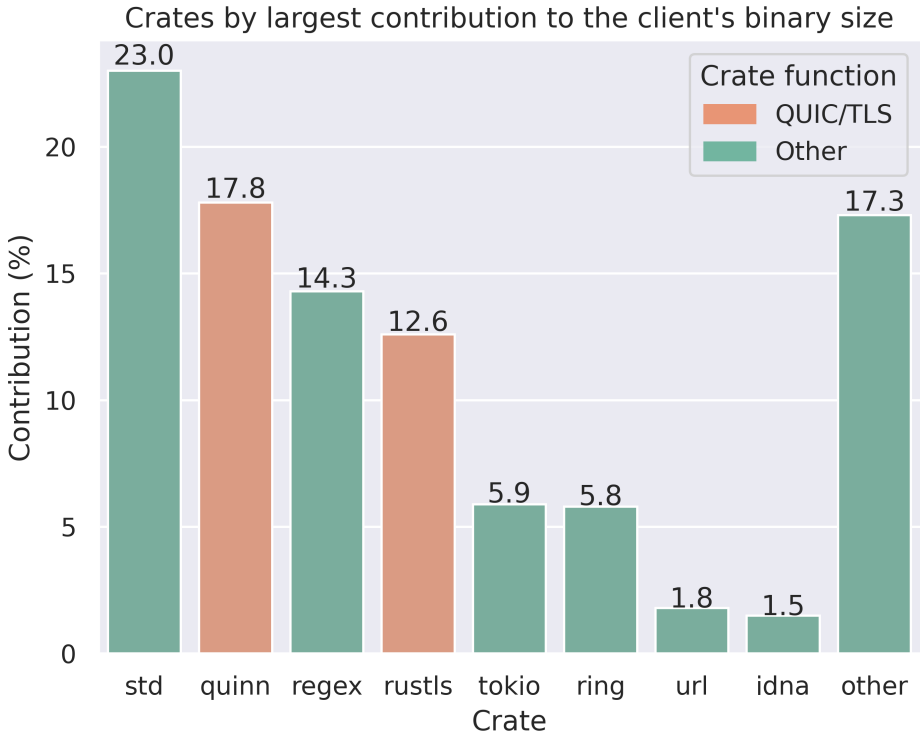



***Figure 5.4:*** *Top contributions to the binary size of the MQuicTT client by crate. That is, which libraries contribute the largest footprint.*

hence it is part of the TLS implementation. Overall, this means that the QUIC stack constitutes 36.2% of MQuicTT's client binary size. This result is in line with our prediction that the QUIC stack will contribute the most to the binary size.

Unexpectedly, however, the *regex* crate contributes 14.3% to the size of the client binary exceeding even the contribution made by the TLS implementation. This can partially be attributed to Rust's support for Unicode in strings. Specifically, a *char* type in Rust represents a Unicode scalar value, a Unicode version agnostic type. While providing many wanted features, Rust's native support for Unicode also increases the used resources. A possible workaround would be to add optional support for the *unicase* :: *Ascii* type to the *regex* crate. Another possible reason for the size of this crate is the source codes extensive use of the *inline*(*always*) annotation. This annotation tells the compiler to inline functions across crate boundaries, possibly contributing to the binary size. Overall, we can see that this anomaly can be attributed to the choice of programming language, which presents an insight into Rust as a language for hardware constrained devices.

We can also see that despite other crates not contributing a considerable amount towards the binary size, cumulatively, their size adds up to 17.3% of the binary size due to the large number of them. We have found over 40 crates that contributed to this, including data structure implementations, lower-level network interfaces, abstractions on byte buffers and logging utilities. Notably, we have found that logging utilities and error reporting do not largely contribute to the size of the client's binary. This comes in contrast to one of the steps in reducing the binary size taken by Eggert (2020).

Figure 5.5 shows similar results for MQuicTT's broker. We can see that in the case of the broker,

the QUIC stack constitutes 32.9% of the binary size. Similarly to the client, the *regex* crate again shows a large binary size footprint, and the lesser crates contribute to a sizeable cumulative amount. Additionally, similarly to the results presented for the client, we have found that the logging and error checking crates have not contributed largely to the binary size.

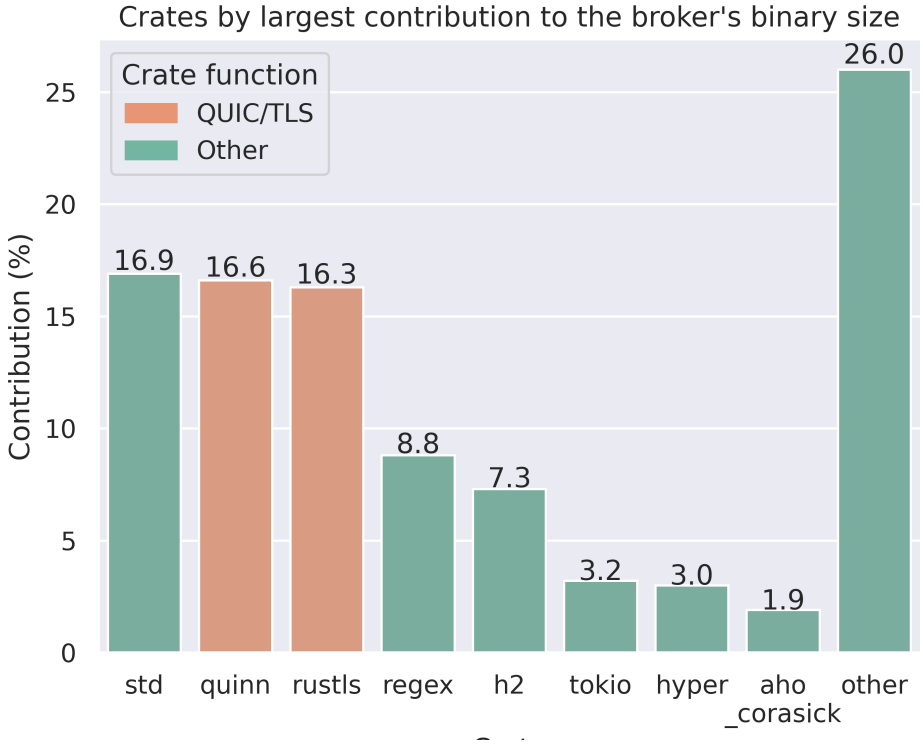


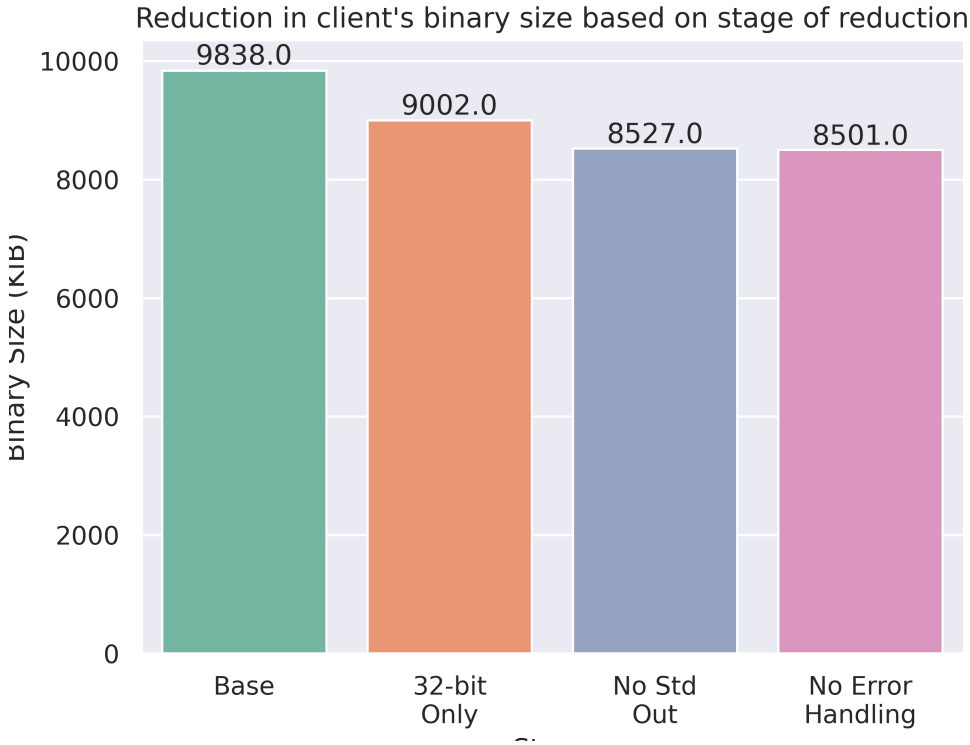

***Figure 5.5:*** *Top contributions to the binary size of the MQuicTT broker by crate. That is, which libraries contribute the largest footprint.*

We have further analysed the binary to see which methods contribute to the size for the client and broker. Notably, in both cases, the *quinn_proto* module of *quinn* contains several methods such as *connection* :: *Connection* :: *process_payload* and *endpoint* :: *Endpoint* :: *handle* that each contribute around 0.8%. Notably, however, the *regex* :: *exec* :: *ExecBuilder* :: *build* method from the *regex* crate contributes 0.7%, which is a larger contribution than *quinn's* methods for decrypting packets and polling for connections.

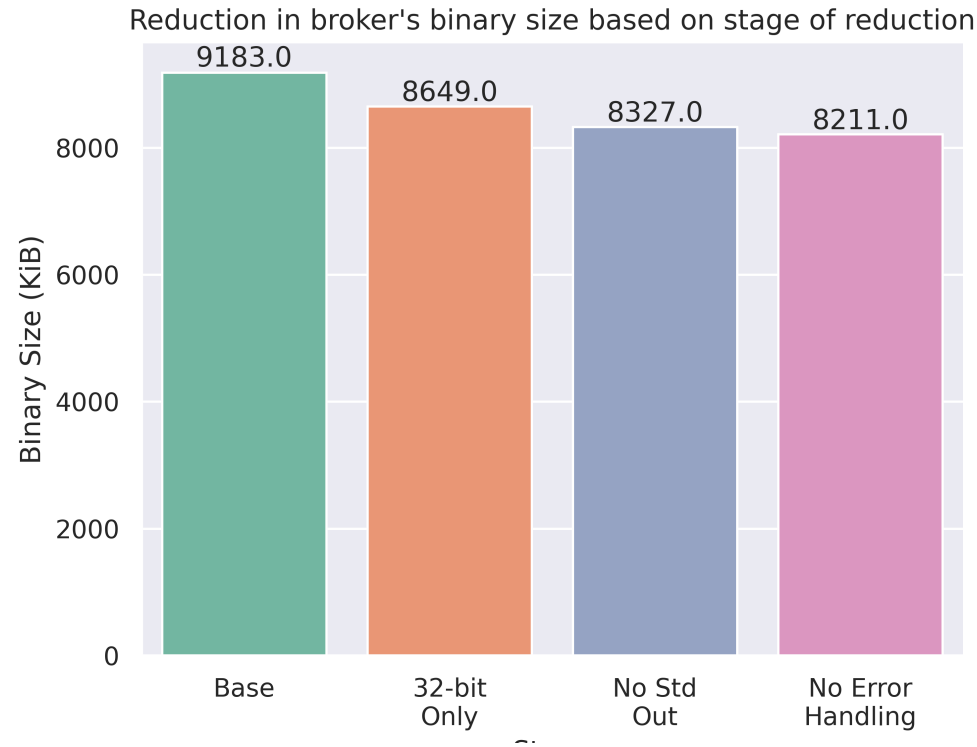
### 5.2.3 Reducing the QUIC stack

As we can see from the composition of the binaries of MQuicTT, the QUIC stack is around a third of the size of the implementation. Hence, we now analyse methods for reducing QUIC's contribution following the previously described methodology.

Figure 5.6 demonstrates the reduction in binary size at each stage described in the methodology. Notably, compiling the binary to a 32-bit only target contributes significantly to the reduction of the binary size. The 32-bit only version of the broker is 5.8% smaller, and the 32-bit only client is 8.5% smaller. However, the two remaining stages did not contribute significantly to the reduction of the binary sizes. This can perhaps be attributed to implementation details as the reduction due to error handling and standard output use would be directly proportional to their use in the codebase. This is further reinforced by the client being reduced by a higher




*(a) Client binary size reduction.*




*(b) Broker binary size reduction.*

***Figure 5.6:*** *The stages of the reduction and the sizes of the QUIC binary at the corresponding stages. Subfigure 5.6a shows the reduction in the binary size of the client and Subfigure 5.6b shows similar results for the binary size of the broker. Compiling in 32-bit only bode results in the highest decrease in binary size.*

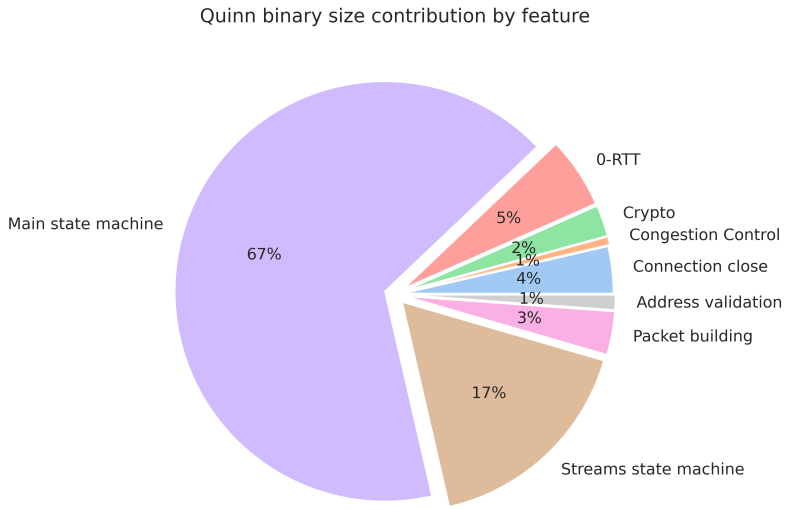



***Figure 5.7:*** *A by-feature breakdown of the binary size contribution of Quinn - the QUIC library used for MQuicTT. We can see that Quinn heavily leverages the state machine, making the binary size reduction operation quite difficult.*

percentage from removing error handling due to the client's source code having more of this feature.

Hence, from the binary analysis conducted, we can see that the QUIC stack contributes a significant amount to the overall binary size of MQuicTT. We can also see that reducing the size of the QUIC stack is not trivial, with the best results coming from compiling to a 32-bit only target.

Another avenue to reduce the binary size is to look at which features of QUIC and TLS contribute most to the binary size. We have performed a by feature analysis of the *Quinn* and *Rustls* binary contributions using the previously described methodology.

Figure 5.7 shows the by feature breakdown of the *Quinn* contribution to the binary size. Notably, we can see that *Quinn* heavily leverages a single state machine that handles a lot of logic that may be best abstracted away. For example, version negotiation does not have its own function and is rather just part of the general connection handling. Connection migration also does not appear to have its own contribution.

As a result, it is difficult to say how much exactly these features contribute to the overall size of the *MQuicTT* binary. There are two possible ways to estimate this that we considered. Firstly, we could look at the number of lines of code and draw some conclusions from this. However, lines of code do not correspond to binary size well at all; hence, this option was rejected. Secondly, we could re-write the *Quinn* implementation with better abstractions, isolating each feature. However, this is the ideal solution was unfortunately out of scope for this project.

From the features that were extracted successfully, we can see that 0-RTT contributes 5% or 12.2*Kb*. 0-RTT connection re-establishment, however, as discussed previously, can be seen as a favourable feature for IoT devices due to its ability to reduce connection time. In addition to this,

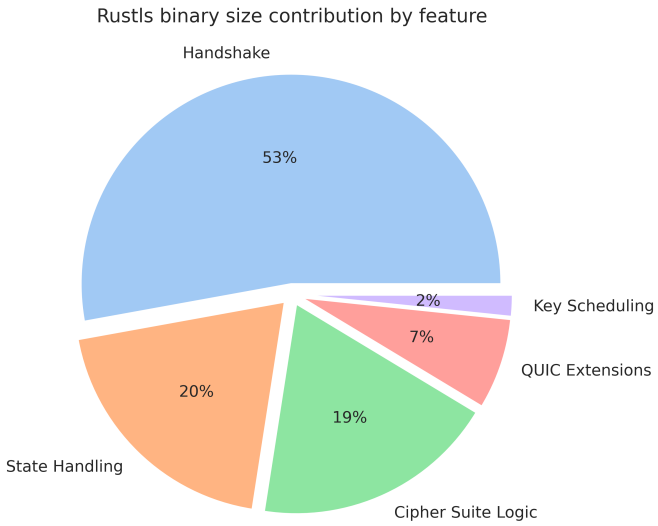



***Figure 5.8:*** *A by-feature breakdown of the binary size contribution of Rustls - the TLS library used for secure communication. Notably, the new QUIC extensions added to TLS contribute significantly to the binary size contribution.*

removing this feature would not surmount a substantial reduction of *MQuicTT*'s binary size.

Overall, from these results, we can see that the complexity of *QUIC* in its state machine and handling of streams is the key reason for *Quinn*'s 16% contribution to the size of *MQuicTT*'s binary.

Figure 5.8 shows a similar breakdown for *Rustls*. As expected, the handshake logic contributes most to the size of the binary. The functions that contribute most to the handshake are the client and server hello handlers. As QUIC does not use the traditional TLS handshake, it is possible to save space by stripping away the non-QUIC handshake semantics. It is hard to estimate how much exactly this would remove without making the changes; however, we can approximate this by removing handshake functions that QUIC would definitely not use. In our approximation, the contribution from the handshake could be reduced from 53% to approximately 44%.

*Rustls* also seems to have complex logic for interfacing with and picking cypher suites. The handling for this is a fifth of *Rustls*'s total binary size contribution. Some of this logic can perhaps be simplified if the deployment pre-chooses a cypher suite to use. That is, the library can be changed to interface only with that singular cypher.

Notably, QUIC extensions to *Rustls* such as in-place encryption and QUIC key generation only contribute 7%. This again shows that if non-QUIC features were to be removed, then we could save on binary size. However, this is also shows quite a large overhead for applications that do not use QUIC.

In regards to further reductions to the binary size, the contributions made by the standard library include over a thousand functions of approximately the same size. As we have already removed features such as backtracing, demangling and error reporting, removing these would be risky to the core functionality and would need to be investigated manually. The most frequently

appearing function is *stdcore* :: *ptr* :: *drop_in_place*, which is used by the compiler to execute the destructor of a pointed-to-value.

Overall, the most significant effect to the reduction of the binary size of *MQuicTT* came from compiling the binary in 32-bit only mode, removing error reporting and handling, and any standard output. A surprising result is the contribution of the regex library, which can be attributed to the fact that Rust opts to use $utf-8$ instead of *ascii* for strings. Reducing the binary size by removing QUIC features may not be the best approach due to limited gains. However, stripping away non-QUIC features from TLS may reduce the binary size considerably.

## 5.3 Summary

When discussing the possibility of using a Rust QUIC implementation as the transport layer protocol for network firmware in IoT devices, we established that the QUIC implementation must perform at least as well as the baseline and have a binary size that can fit onto widely used IoT devices.

From the above analysis, we can conclude that *MQuicTT* performs at least as well as the baseline TCP implementation of *rumqtt*. When it comes to connection time, *MQuicTT* performs marginally better, and when it comes to total transmission time, the implementations are on par. Hence, this requirement is satisfied.

In terms of binary size, we have managed to reduce it to approximately $8Mb$ from a starting point of $9.1Mb$, a reduction of 12.1%. We have also shown that the methods developed by Eggert (2020) do translate to our implementation, implying that this could be a general methodology. This size of binary would easily be installable on popular IoT devices such as the Raspberry Pi 3 Model B, Beagle Board or the Arduino Due. However, this size of binary would not support industrially-wide used chips such as the esp32. We have analysed the possible avenues of further reducing the binary size of *MQuicTT* by addressing issues with the regex library and by analysing *Quinn* and *Rustls* by feature. Hence, overall, we can say that we have achieved creating an implementation that can be used on a large number of IoT devices, but not on ones with stricter hardware constraints.

# 6 Conclusion

We conclude with a brief summary of the work conducted, the developed implementation and results found. Additionally, we provide a discussion on avenues for future work.

This work has looked at recent developments in the spaces of transport layer protocols and programming languages. We have aimed to analyse if the technologies, QUIC and Rust, set to succeed current industry standards can be used for IoT network firmware. The context of IoT provides interesting constraints in terms of the balance of performance, security and hardware constraints. We have discussed MQTT - the most widely used IoT application layer protocol. MQTT presents an HTTP like lightweight, publish-subscribe communication system. We have also shown that MQTT relies on the transport layer to provide transport semantics and secure communication.

By analysing the currently available QUIC and MQTT implementations, we identified libraries that were used as the basis for our implementation. Both of the selected libraries were developed in the Rust programming language. Using *Quinn*, we developed an intermediate API - *QuicSocket* that provides an API for connecting, sending and receiving using QUIC. Further, we have changed the base *rumqtt* implementation to use QUIC using this API, resulting in *MQuicTT* - a QUIC based MQTT implementation.

Using a mininet test bench, we have analysed the performance of the the resulting implementation and have shown that it is comparable to *rumqtt* in terms of connection time and total transmission time.

We have analysed the binary sizes of the broker and client of *MQuicTT* and identified the dependencies that contribute the most to the size. We have shown that Rust's use of $utf-8$ strings can result in size bloat. Additionally, we have further analysed the contributions made by the QUIC and TLS implementations to devise a methodology for trimming down the QUIC stack.

Overall, we have found that while *MQuicTT* performed comparably to *rumqtt*, we could not trim down the binary size enough for it to be used on truly hardware constrained devices. However, the resulting implementation can be used on devices of the Raspberry Pi class.

## 6.1 Future work

This section will address the limitations of the work that we have done and discuss future work that would improve or expand upon the topic. Each section will present a direction for further work and a discussion of it.

### 6.1.1 QUIC implementations

The number of available QUIC implementations has already far surpassed the number of major TCP implementations that are used in deployments. Even during the course of this work being produced, a new Rust QUIC implementation was made public - Amazon's *s2n – quic* [1]. The

[1] `https://github.com/aws/s2n-quic`

number of QUIC implementations and the difference in direction compared to TCP has several implications for this work.

Firstly, a new, more efficient implementation of Rust could be developed that would be a better fit as the base implementation for *MQuicTT*. Hence, new implementations would need to be analysed.

Secondly, each new implementation means that many servers will have to interface with differing implementations, often with different implementation semantics. An analysis of how different QUIC implementations perform in conjunction with each other could provide different efficiency statistics to the ones presented in our analysis. That is, both our broker and clients used *Quinn*, and it would be interesting to analyse if using, for example, *Quinn* for the broker and $s2n - quic$ for the client impacts performance.

Additionally, with so many implementations having to interface with each other, it would be worthwhile to analyse the number of bugs in QUIC deployments compared to TCP ones.

### 6.1.2 Binary analysis

As discussed, all binary analysis for this work was conducted manually. This task is rather tedious and error-prone. While it is possible to create a dependency analyser and a decompiler to analyse the binaries, it is currently impossible to connect this automatically with different features of a protocol.

For this to be possible, we would need to augment protocol implementations with some sort of indication of a feature. In the Rust programming language, this could be done via a macro. For example, a function with the macro $\#[feature(VersionNegotation)]$ could be used to indicate Quic's version negotiation feature. This method, however, can be viewed as quite restrictive as it essentially enforces a by-feature abstraction. As we have seen in *Quinn*, this is not always the way developers want to write their code.

Hence, to be able to conduct this sort of analysis, we would need to find a non-intrusive way of augmenting implementations with such details.

### 6.1.3 Comparison analysis to CoAP

We have previously discussed the similarities between using QUIC for MQTT and using CoAP. Both methods utilise UDP and an HTTP style messaging approach. Additionally, both provide retalitevly low code overhead, which is important for hardware constrained devices.

While this work did not focus on the comparison of *MQuicTT* to CoAP, such an analysis would be crucial to understanding the performance of QUIC in IoT. An additional avenue of future work is extracting the core features that both these approaches share and analysing if they could be employed to make other protocols better for hardware constrained devices. The result of such work could create a general approach to creating protocols for IoT devices.

### 6.1.4 Improved test bench

There are several avenues that could be taken for improving the analysis test bench that we used. Firstly, a remote controller could be implemented for the mininet topology to be able to use an actual mesh topology, with cycles, instead of the minimum spanning tree one deployed for this work. This would also mean that different load balancing and routing options could be tested for the scenarios to better replicate a realistic deployment. This would also mean that network failures, such as switches going offline, could be simulated programmatically. Naturally, a hardware test bench analysis would also improve any further work.

# A MQTT simulation messages

```
Topic
------

GENERICO-CORPO/JSON/0/1/ADDITIVE-MANUFACTURING-13/PRINT/1

Payload
-------

"print" = {
   "Timestamp"="20201001 171035",
   "BedTemp"="75",
   "ExtruderTemp"="205",
   "Adhesion"="Skirt",
   "File"="/home/prints/part.gcode",
   "Speed"="50 mm/s",
   "Layer Height"="0.12 mm",
   "Retraction"="6mm at 25mm/s",
   "Infill"="20%",
   "Initial layer speed"="20 mm/s",
   "Initial fan speed"="0%"
}
```

***Listing A.1:*** *The message payload and topic used to transmit for the printer farm scenario.*

```
Topic
------

HOME/JSON/0/1/SET-TEMP

Payload
-------

"req" = {
   "Room"="Bedroom",
   "Device"="AC",
   "Temp"="17",
}
```

***Listing A.2:*** *The message payload and topic used to transmit for the printer home scenario.*

# B | Mininet Topology Definition

The full mininet topology definition via the *dump* and *links* command. The Python API definition can also be found on GitHub [1].

```
Links:

h00-eth1<->h01-eth1 (OK OK) h00-eth2<->h02-eth1 (OK OK)
h00-eth3<->h03-eth1 (OK OK) h00-eth4<->h04-eth1 (OK OK)
h01-eth2<->h02-eth2 (OK OK) h01-eth3<->h03-eth2 (OK OK)
h01-eth4<->h04-eth2 (OK OK) h02-eth3<->h03-eth3 (OK OK)
h02-eth4<->h04-eth3 (OK OK) h03-eth4<->h04-eth4 (OK OK)
h10-eth1<->h11-eth1 (OK OK) h10-eth2<->h12-eth1 (OK OK)
h10-eth3<->h13-eth1 (OK OK) h10-eth4<->h14-eth1 (OK OK)
h11-eth2<->h12-eth2 (OK OK) h11-eth3<->h13-eth2 (OK OK)
h11-eth4<->h14-eth2 (OK OK) h12-eth3<->h13-eth3 (OK OK)
h12-eth4<->h14-eth3 (OK OK) h13-eth4<->h14-eth4 (OK OK)
s0-eth1<->h00-eth0 (OK OK) s0-eth2<->h01-eth0 (OK OK)
s0-eth3<->h02-eth0 (OK OK) s0-eth4<->h03-eth0 (OK OK)
s0-eth5<->h04-eth0 (OK OK) s0-eth6<->s9-eth2 (OK OK)
s1-eth1<->h10-eth0 (OK OK) s1-eth2<->h11-eth0 (OK OK)
s1-eth3<->h12-eth0 (OK OK) s1-eth4<->h13-eth0 (OK OK)
s1-eth5<->h14-eth0 (OK OK) s1-eth6<->s9-eth3 (OK OK)
s9-eth1<->h9-eth0 (OK OK)
```

***Listing B.1:*** *Mininet links on the used topology.*

```
Dump:

<Host h00:
    h00-eth0:10.0.0.1,h00-eth1:None,h00-eth2:None,h00-eth3:None,h00-eth4:None
    pid=17658>
<Host h01:
    h01-eth0:10.0.0.2,h01-eth1:None,h01-eth2:None,h01-eth3:None,h01-eth4:None
    pid=17660>
<Host h02:
    h02-eth0:10.0.0.3,h02-eth1:None,h02-eth2:None,h02-eth3:None,h02-eth4:None
    pid=17662>
<Host h03:
    h03-eth0:10.0.0.4,h03-eth1:None,h03-eth2:None,h03-eth3:None,h03-eth4:None
    pid=17664>
```

[1] Topology - https://github.com/Apolexian/level4-scripts/blob/master/mesh_topo/mesh.py

```
<Host h04:
    h04-eth0:10.0.0.5,h04-eth1:None,h04-eth2:None,h04-eth3:None,h04-eth4:None
    pid=17666>
<Host h9: h9-eth0:10.0.0.6 pid=17668>
<Host h10:
    h10-eth0:10.0.0.7,h10-eth1:None,h10-eth2:None,h10-eth3:None,h10-eth4:None
    pid=17670>
<Host h11:
    h11-eth0:10.0.0.8,h11-eth1:None,h11-eth2:None,h11-eth3:None,h11-eth4:None
    pid=17672>
<Host h12:
    h12-eth0:10.0.0.9,h12-eth1:None,h12-eth2:None,h12-eth3:None,h12-eth4:None
    pid=17674>
<Host h13:
    h13-eth0:10.0.0.10,h13-eth1:None,h13-eth2:None,h13-eth3:None,h13-eth4:None
    pid=17676>
<Host h14:
    h14-eth0:10.0.0.11,h14-eth1:None,h14-eth2:None,h14-eth3:None,h14-eth4:None
    pid=17678>
<OVSSwitch s0:
    lo:127.0.0.1,s0-eth1:None,s0-eth2:None,s0-eth3:None,s0-eth4:None,s0-eth5:None,
   s0-eth6:None pid=17683>
<OVSSwitch s1:
    lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None,s1-eth5:None,
   s1-eth6:None pid=17686>
<OVSSwitch s9: lo:127.0.0.1,s9-eth1:None,s9-eth2:None,s9-eth3:None pid=17689>
<Controller c0: 127.0.0.1:6653 pid=17651>
```

***Listing B.2:*** *Mininet* **dump** *on the used topology.*

# 6 Bibliography

F. A. Alaba, M. Othman, I. A. T. Hashem, and F. Alotaibi. Internet of Things security: A survey. *J. Netw. Comput. Appl.*, 2017. doi: 10.1016/j.jnca.2017.04.002.

R. Alena, R. Gilstrap, J. Baldwin, T. Stone, and P. Wilson. Fault tolerance in ZigBee wireless sensor networks. In *2011 Aerospace Conference*, pages 1–15, Mar. 2011. doi: 10.1109/AERO.2011.5747474. ISSN: 1095-323X.

Bytebeam. rumqtt, Mar. 2020. URL `https://github.com/bytebeamio/rumqtt`. original-date: 2019-10-15T07:49:25Z.

V. Cerf and R. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, May 1974. ISSN 1558-0857. doi: 10.1109/TCOM.1974.1092259.

Chromium. QUIC, a multiplexed transport over UDP - The Chromium Projects, 2021. URL `https://www.chromium.org/quic`.

CoRE. Core working group | ietf core wg, 2016. URL `https://core-wg.github.io/`.

Eclipse. Eclipse Paho MQTT Rust Client Library, Dec. 2018. URL `https://github.com/eclipse/paho.mqtt.rust`. original-date: 2017-10-03T19:57:53Z.

L. Eggert. Towards Securing the Internet of Things with QUIC. 2020. doi: 10.14722/diss.2020.23001.

Ericsson. IoT & Smart manufacturing - Mobility Report, June 2018. URL `https://www.ericsson.com/en/reports-and-papers/mobility-report/articles/realizing-smart-manufact-iot`.

S. Gupta and N. Lingareddy. Security Threats and Their Mitigations in IoT Devices. 2021. doi: 10.1007/978-3-030-69921-5_42.

J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. Request for Comments RFC 9000, Internet Engineering Task Force, May 2021. URL `https://datatracker.ietf.org/doc/rfc9000`. Num Pages: 151.

P. Kumar and B. Dezfouli. Implementation and Analysis of QUIC for MQTT. *arXiv:1810.07730 [cs]*, Jan. 2019. URL `http://arxiv.org/abs/1810.07730`. arXiv: 1810.07730.

B. Lantz and B. Heller. Mininet: An Instant Virtual Network on Your Laptop (or Other PC) - Mininet, 2013. URL `http://mininet.org/`.

G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong. The 2015 ukraine blackout: Implications for false data injection attacks. *IEEE Transactions on Power Systems*, 32, 2017. ISSN 08858950. doi: 10.1109/TPWRS.2016.2631891.

Z. Ling, K. Liu, Y. Xu, C. Gao, Y. Jin, C. Zou, X. Fu, and W. Zhao. IoT Security: An End-to-End View and Case Study. *ArXiv*, 2018.

N. Matsakis. Baby Steps, Oct. 2019. URL https://smallcultfollowing.com/babysteps/blog/2019/10/26/async-fn-in-traits-are-hard/.

A. Mileva, A. Velinov, L. Hartmann, S. Wendzel, and W. Mazurczyk. Comprehensive analysis of MQTT 5.0 susceptibility to network covert channels. *Computers & Security*, 104: 102207, May 2021. ISSN 0167-4048. doi: 10.1016/j.cose.2021.102207. URL https://www.sciencedirect.com/science/article/pii/S0167404821000316.

M. Miller. MSRC-Security-Research/2019_02 - BlueHatIL - Trends, challenge, and shifts in software vulnerability mitigation.pdf at master · microsoft/MSRC-Security-Research, 2019. URL https://github.com/microsoft/MSRC-Security-Research.

Mozilla. Neqo, an Implementation of QUIC written in Rust, Jan. 2022. URL https://github.com/mozilla/neqo. original-date: 2019-02-18T19:20:20Z.

OASIS. MQTT - The Standard for IoT Messaging, 2014. URL https://mqtt.org/.

Ofcom. UK home broadband performance, measurement period March 2021, Sept. 2021. URL https://www.ofcom.org.uk/research-and-data/telecoms-research/broadband-research/broadband-speeds/uk-home-broadband-performance-march-2021.

J. Postel. User Datagram Protocol. Request for Comments RFC 768, Internet Engineering Task Force, Aug. 1980. URL https://datatracker.ietf.org/doc/rfc768.

S. Previdi, L. Ginsberg, C. Filsfils, A. Bashandy, H. Gredler, and B. Decraene. IS-IS Extensions for Segment Routing. Request for Comments RFC 8667, Internet Engineering Task Force, Dec. 2019. URL https://datatracker.ietf.org/doc/rfc8667.

E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Request for Comments RFC 8446, Internet Engineering Task Force, Aug. 2018. URL https://datatracker.ietf.org/doc/rfc8446. Num Pages: 160.

J. Roskind. QUIC: Design Document and Specification Rationale - Google Docs, 2012. URL https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit.

Rust. RAII - Rust By Example, 2021. URL https://doc.rust-lang.org/rust-by-example/scope/raii.html.

SDU. ICTP-SDU: about PingER, Oct. 2013. URL https://web.archive.org/web/20131010010244/http://sdu.ictp.it/pinger/pinger.html.

Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). *Rfc 7252*, 2014. ISSN 2070-1721. URL https://datatracker.ietf.org/doc/html/rfc7252.

Statista. Number of IoT devices 2015-2025, 2016. URL https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

S. N. Swamy, D. Jadhav, and N. Kulkarni. Security threats in the application layer in IOT applications. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 477–480, Feb. 2017. doi: 10.1109/I-SMAC.2017.8058395.

T-Mobile. Narrowband iot solution developer protocols guide, 2018. URL https://www.t-mobile.com/content/dam/tfb/pdf/IoT-Solution-Developer-Protocols-Guide.pdf?icid=TFB_TMO_P_19IOT_E3UOLES40BRO6VPW19260.

M. Thomson and S. Turner. Using TLS to Secure QUIC. Request for Comments RFC 9001, Internet Engineering Task Force, May 2021. URL https://datatracker.ietf.org/doc/rfc9001. Num Pages: 52.

I. Vaccari, M. Aiello, and E. Cambiaso. SlowTT: A Slow Denial of Service against IoT Networks. *Information*, 11(9):452, Sept. 2020. ISSN 2078-2489. doi: 10.3390/info11090452. URL https://www.mdpi.com/2078-2489/11/9/452.

M. Weiser. The computer for the 21st century. 1991. doi: 10.1038/SCIENTIFICAMERICAN0991-94.