
University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

# Improving MQTT at the transport layer using QUIC

**Ivan Nikitin**

January 25, 2022

# Abstract

Every abstract follows a similar pattern. Motivate; set aims; describe work; explain results.

“XYZ is bad. This project investigated ABC to determine if it was better. ABC used XXX and YYY to implement ZZZ. This is particularly interesting as XXX and YYY have never been used together. It was found that ABC was 20% better than XYZ, though it caused rabies in half of subjects.”

# 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: 25 March 2022

# Contents

# 1 Introduction

# 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, the MQTT protocol and the Rust programming language.

## 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 in order 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 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.

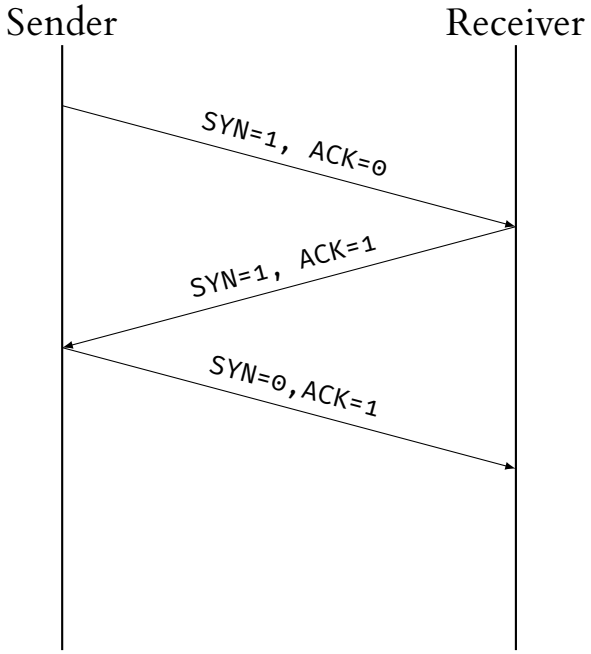


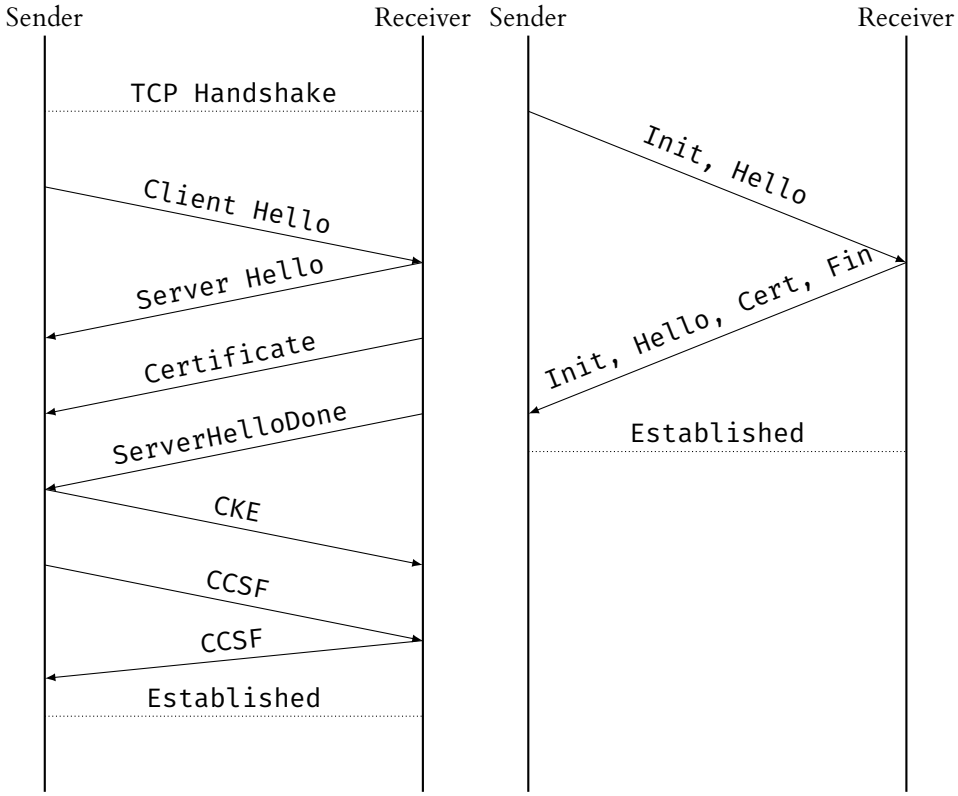

***Figure 2.1:*** *The TCP handshake needed for connection establishment. The values of the SYN and ACK fields being set indicate the kind of segment sent. For example, for the SYN-ACK stage both the SYN and ACK fields are set.*

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 its creation, QUIC is a relatively new general-purpose transport layer protocol 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). QUIC 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 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,




*(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.*

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

It is hard to give a set criterion or definition for which devices qualify as IoT devices. Generally, an IoT device is 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 also leave room for memory management issues, leading to vulnerabilities. When it comes to privacy, a 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, 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 just 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. We present a comparison of these results to the ones obtained through the evaluation that we have conducted in chapter 7.

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 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 3.1.

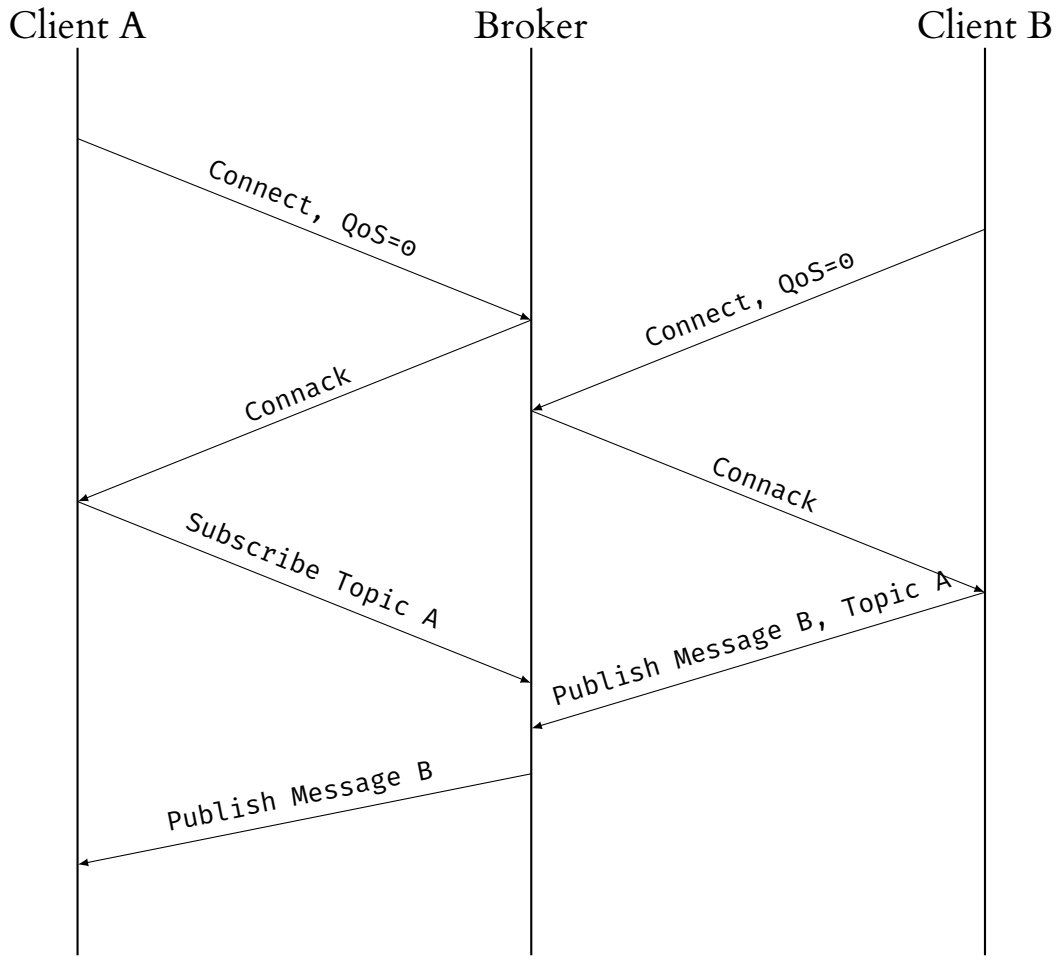



***Figure 2.3:*** *The MQTT connection, specifying the QoS measure, followed by the publishing of a message to a broker. It is important to note that the QoS measure does not impact the underlying data transmission provided by a protocol such as TCP. In this example, client B publishes message B to topic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

## 2.4 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

```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.*

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.

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 Foundational implementations

This chapter presents the rationale behind our choice of QUIC and MQTT implementations. In terms of QUIC, we used the *quinn* implementation in order to create the intermediate library discussed in Chapter 4. On the MQTT side, *rumqtt* was used as the starting point for our MQTT/QUIC implementation.

The choice criteria follow from the previously presented background and some general criteria. We analysed the available implementations based on their storage footprint, the programming language they were implemented in, their use of TLS, widespread adoption and availability of documentation.

## 3.1 QUIC

In this section, we 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).

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.

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.

***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 |

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.

In terms of using a memory-safe language, 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 was the primary reason for the choice of Rust instead of 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; for example, Quinn handles the QUIC handshake in the library and does not require the developer to create an event loop.

## 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.

*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

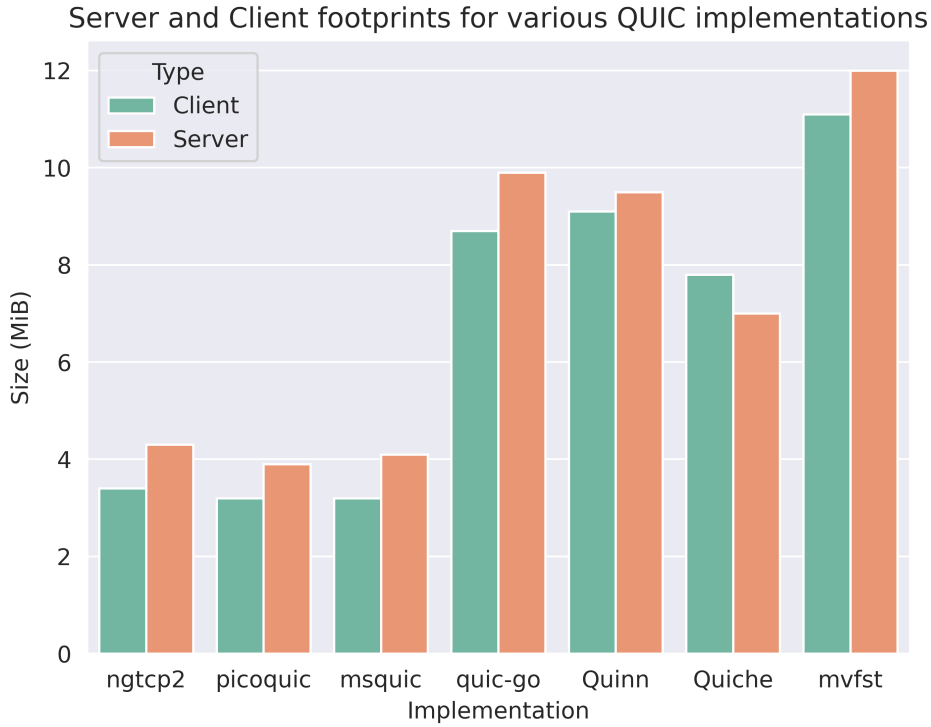



***Figure 3.1:*** *The sizes of the client and server footprints for the selected QUIC implementations. Notable, only Quiche produced a server binary with a smaller size than its corresponding client binary. It is hard to estimate the error margin for the data as this depends on implementation details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

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 QuicSocket

This chapter 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 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.

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

```
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:*** *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.*

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.

# 5 MQTT with QUIC at the transport layer

# 6 Network simulation

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 chapter, 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 (SNDs) using the *OpenFlow* standard.

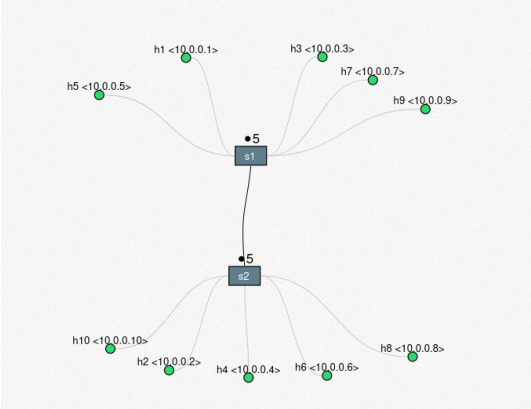



***Figure 6.1:*** *The resulting Dumbbell topology with 5 hosts on either side of the switch. This topology simulates congestion on the link as the hosts have to share it for their data transfer.*

Using the Python API provided, we created the network topology shown in Figure 6.1. The script takes several parameters to create a simulation environment resembling a realistic scenario. The variables that the script changes between 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, we measure bandwidth in bits per second rather than hertz in computer networking. The delay of a link specifies the latency of the link. It is the time that a bit of data takes to travel across a link. We measure this in milliseconds. 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, 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 6.1:*** *The parameters chosen for each link simulation in Mininet. The types of links were chosen as the most commonly occurring ones in IoT use cases. The data also 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 | 2 |
| ZigBee | 0.25 | 5 | 1 |
| 4G | 4 | 20 | 1.5 |
| 3G | 1 | 40 | 1.5 |
| 100Mb Ethernet | 100 | 1 | 0.2 |

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.

Using the different links, we then transferred a file of equal size using the various QUIC implementations described in Chapter 3.1 for the evaluation of QUIC implementations.

We evaluated the MQTT/QUIC implementation performance using the same topology and simulation parameters. In general, MQTT allows for messages with a maximum size of approximately 260MB. However, this is a huge message, and most publicly deployed brokers will reject it, so a general use-case was simulated.

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. Notably, brokers reserve topics for system messages starting with the $ character.

Taking this into account, we opted for a smart home scenario to simulate the MQTT communication. The data transmitted can be found in Appendix B.

# 7 Analysis

# 8 Conclusion

# A | Data

# B MQTT simulation messages

# 8 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`.

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

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/`.

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

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.

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.

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.

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.

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