University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Level 4 Project Report Template

Ivan Nikitin

October 28, 2021

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 and systems programming spaces and identify how these apply to what we aim to achieve.

2.1 The evolution of the transport layer

The following section provides a background on transport layer protocols and recent advancements in the space applicable to the body of work conducted. Together, the protocols described comprise most of the traffic on the Internet. Hence, we do not consider more minor use-case protocols that exist.

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. TCP ensures that data is not lost, altered or duplicated and delivers it in the same order that it sent it. 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.

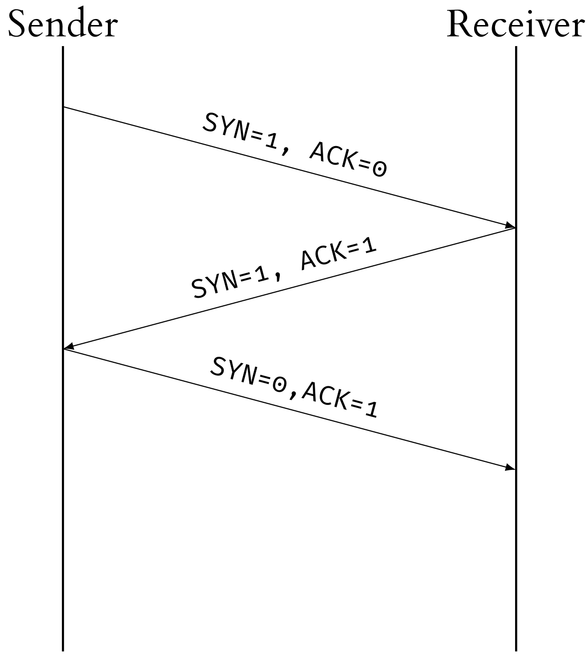


Figure 2.1: *The TCP handshake needed for connection establishment. The values of the SYN and ACK fields being set indicate the kind of segment sent. For example, for the SYN-ACK stage both the SYN and ACK fields are set.*

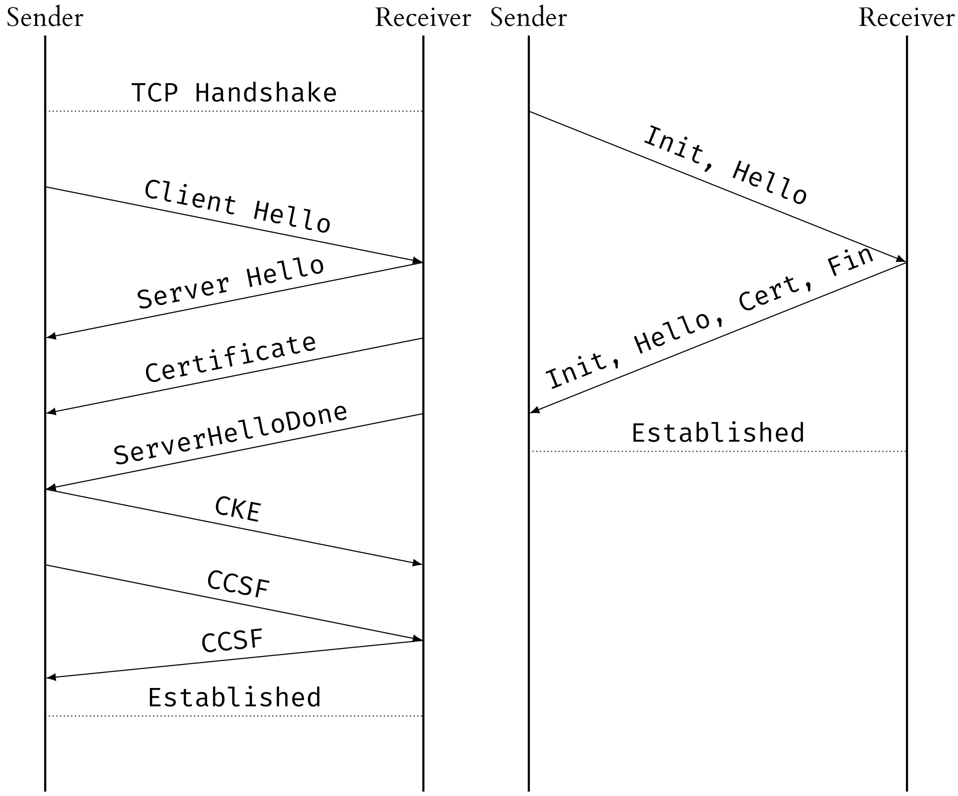
Due to the establishment of communication and properties guaranteed by TCP, this process lengthens communication latency. Hence, in use-cases where reliability and connection state is not required, the User Datagram Protocol (UDP) (Postel 1980) is preferred. UDP uses a connectionless communication model 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.

QUIC is a relatively new general-purpose transport layer protocol designed by Roskind (2012) initially at Google as part of the Chromium web engine. In 2015 the first draft of the QUIC protocol was submitted to the IETF and was later standardised (Iyengar and Thomson 2021).

QUIC aims to improve upon, and eventually make obsolete, TCP by using the concept of multiplexing, which is a method of combining several signals or channels of communication over one shared medium. QUIC establishes multiplexed connections between the communicating endpoints using UDP.

QUIC facilitates 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. QUIC creates streams by either the sending or receiving side and can be unidirectional and bidirectional. Each side can send data concurrently on the stream and open any number of streams (specifically, QUIC provides a field for the maximum number of streams during the connection). Hence, QUIC allows an arbitrary number of streams to send arbitrary amounts of data on the UDP connection, subject to the constraints imposed by flow control.

By doing so, QUIC also achieves the secondary goal of lifting 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.



(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.*

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

It is hard to give a set criterion or definition for which devices qualify as IoT devices. Generally, an IoT device usually 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 many 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. Due to the inherent advantages of short-range wireless communication in IoT applications such as smart homes, this category of IoT technologies was the primary focus for the project, as discussed in later chapters. 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. 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.

Due to the uses of IoT, the form factor of these devices has to be physically small. Many of these devices have to run for long periods on a single lithium battery, hence needing to consume as least energy as possible. Additionally, many use cases of IoT devices require a large number of them connected in a 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 $ESP8266$ controller, a widely used IoT chip, comes with 4Mb of flash memory. After installing the firmware and binaries needed for the device to perform its function, little to no memory may remain for additional storage.

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: 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 generally insecure code. Most programmers opt to create software for IoT devices in the C programming language, which, while providing the needed efficiency, is also a source of insecure code that can lead to potential attacks, such as buffer overflows.

On the other hand, we must ensure data integrity and confidentiality to ensure privacy and general security. Data sent via the network must not be tampered with nor snooped on by third parties during communication. Hence, privacy requires secure methods for authentication, authorisation and transport-level 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.

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.

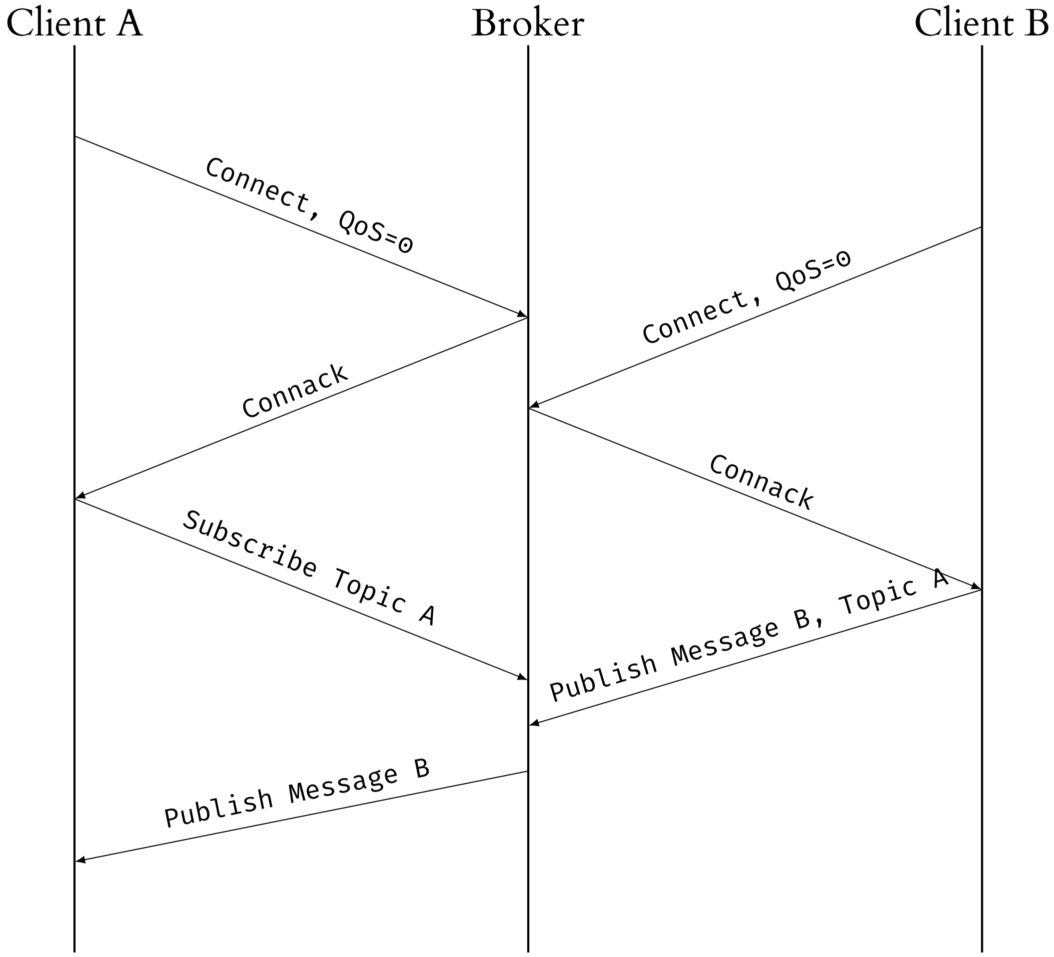


__Figure 2.3:__ The MQTT connection, specifying the QoS measure, followed by the publishing of a message to a broker. It is important to note that the QoS measure does not impact the underlying data transmission provided by a protocol such as TCP. In this example, client B publishes message B to topic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

2.4 The 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 2021b). 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 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.

Additionally, due to the focus on concurrency and safe systems programming, Rust has a natural use in networked systems. However, while Rust aims to be as fast as traditional systems languages, this remains to be seen with even the language's developers saying that the matter of performance is hard to assess Rust (2021a). Therefore, the interest in Rust in this project comes from two main aspects. 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 updates. Secondly, assessing the performance of Rust implementations of the QUIC stack is vital to solidifying Rust as a performant systems programming language. 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 would not matter.

3 Foundational implementations

3.1 QUIC

In this chapter, 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 consider mainstream implementations mainly 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 do not consider implementations paired with a browser web engine as these would be impossible to use on hardware constrained devices. Hence, we do not include notable implementations such as the QUIC implementation of the chromium web engine (Chromium 2021) and *Neqo* (Mozilla 2022).

Table 3.1 demonstrates the analysed QUIC implementations. We consider the implementations based on the programming language, the footprint of the produced binary and how the implementation incorporates TLS. 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 have 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

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.

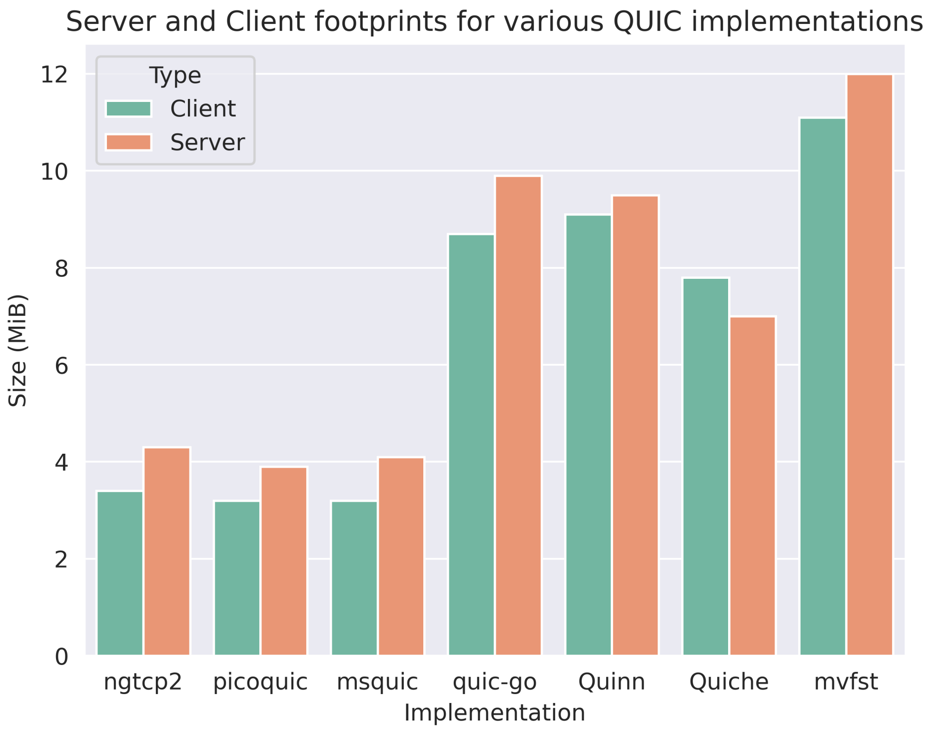


Figure 3.1: *The sizes of the client and server footprints for the selected QUIC implementations. Notable, only Quiche produced a server binary with a smaller size than its corresponding client binary. It is hard to estimate the error margin for the data as this depends on implementation details in the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

Figure 3.1 further visualises the comparison of binary sizes between the various implementations. We can see that the implementations in the C language have the lowest binary footprints, approximately 30% less than their counterparts. Notably, we can also 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, $rustls$.

Considering that we opted to use a memory-safe language, this left us choosing between the two Rust implementations. Although Go is described as memory safe, it does not opt for compile-time memory safety and instead uses the panic model. Between the two, Quinn handles the QUIC handshake in the library and does not require the developer to create an event loop. 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. In addition to this, we found that the Quinn API is easier to work with when creating our intermediate library.

3.2 MQTT

4 QuicSocket

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.

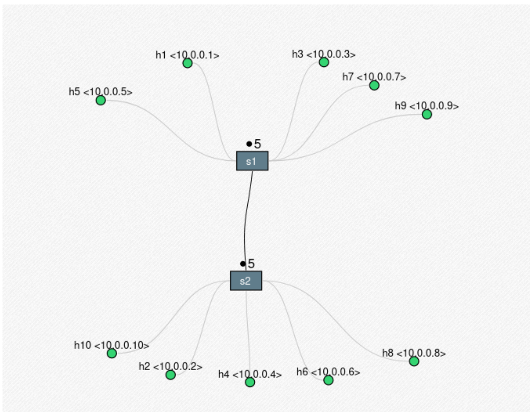


Figure 6.1: *The resulting Dumbbell topology with 5 hosts on either side of the switch. This topology simulates congestion on the link as the hosts have to share it for their data transfer.*

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 Evaluation

7.1 Performance

7.2 Binary size breakdown

7.3 TLS and possible alternatives

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.

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`.

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/`.

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. The Rust Language FAQ, 2021a. URL https://doc.rust-lang.org/1.0.0/complement-lang-faq.html#how-fast-is-rust?

Rust. RAII - Rust By Example, 2021b. 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.