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

In the following section we provide 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, smaller use-case protocols that do exist were not seen as applicable.

First described by Cerf and Kahn (1974), the Transmission Control Protocol (TCP) has been the main protocol of the Internet suite since its initial implementation. TCP provides a *reliable* and *ordered* delivery of bytes. That is, TCP ensures that data is not lost, altered or duplicated and is delivered in the same order that it was sent. This is achieved by assigning a sequence number to each transmitted packet and requiring an *acknowledgment* (commonly referred to as ACK) from the receiving side. If an ACK is not received, the data is re-transmitted. On the receiving side, the sequence numbers can also be used to order packets as intended by the sender.

As TCP is a connection based protocol, connection establishment must take place 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 acknowledgment - ACK, and the sequence number it will start its segment with - SYN. Hence, this step is often referred to as the SYN-ACK. In the third and final step, the client must acknowledge the response. At this point, the connection on which data can be transferred is established.

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 cipher suites, authenticate the server via its public key and certificate authority's signature, and generate a session key that can be used 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 cipher suites, compression methods and a random number. The server responds with a *ServerHello* message that contains the selected TLS version, cipher suite, compression method and its own 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, depending on the chosen cipher suite, may contain a public key. This is followed by the client sending a *ChangeCipherSpec* message indicating that all communication from this point is authenticated and encrypted along with a finished message. The server responds with the same message and the TLS connection is established.

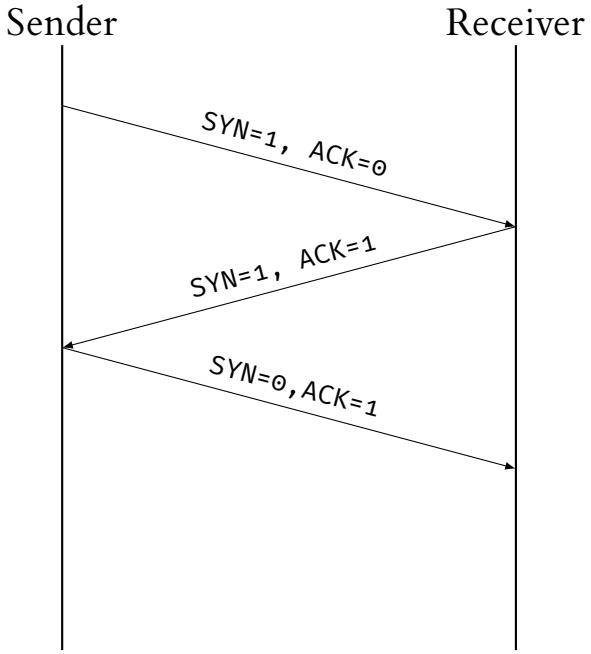


Figure 2.1: *The TCP handshake needed for connection establishment. The values of the SYN and ACK fields being set indicate the kind of segment sent. For example, for the SYN-ACK stage both the SYN and ACK fields are set.*

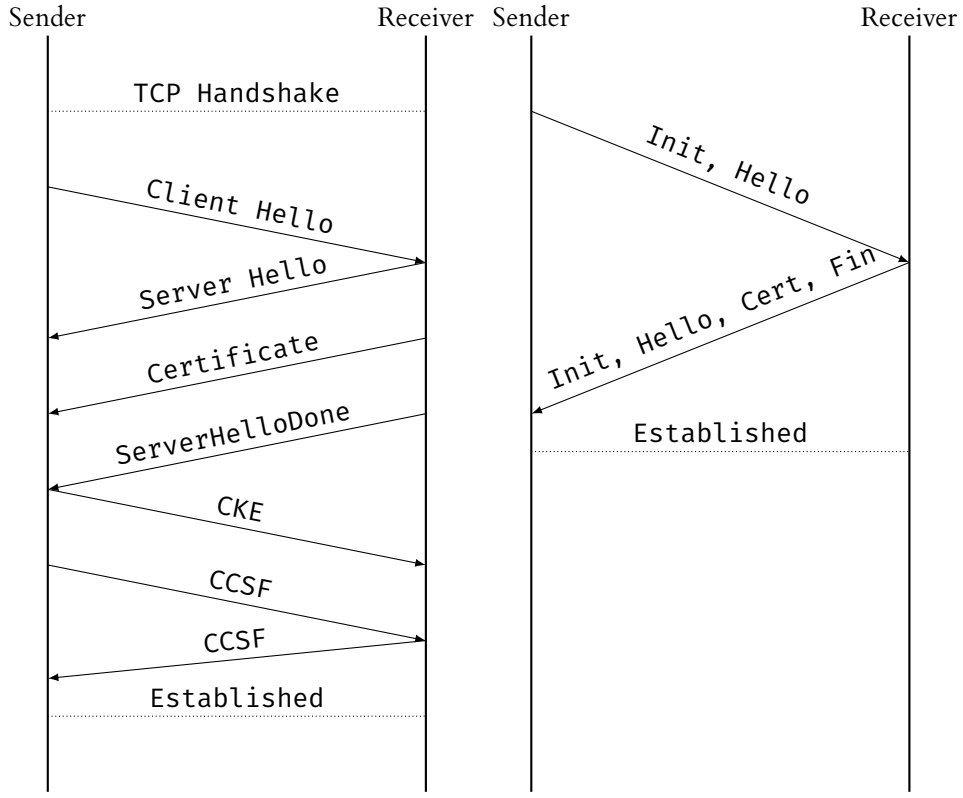
Due to the establishment of communication and properties guaranteed by TCP, such as reliability through retransmission, an inherent trade-off is created, and latency is lengthened. Hence, in use-cases where reliability and connection state is not required, the User Datagram Protocol (UDP) (Postel 1980) is preferred. UDP uses a connectionless communication model that aims to have a minimal number of semantics. The only mechanisms provided by UDP are port numbers and checksums in order to ensure data integrity. This is preferable for real-time systems as using TCP for these would cause an overhead to latency and retransmission of packets that are no longer needed by the application.

QUIC is a relatively new general-purpose transport layer protocol originally designed by Roskind (2012) 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).

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

By doing so, QUIC also achieves secondary goal of lifting congestion control algorithms from the kernel space to the user space. Hence, congestion control algorithms can evolve without having to be 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 come cases, allowing to send data in the first packet.*

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

2.2 The Internet of Things

It is hard to give a set criteria or definition for which devices qualify as IoT devices. Generally, an IoT device is usually a device possessing some processing power that may have embedded censors. The key aspect of IoT devices is that they facilitate exchange of data 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 a multitude of applications in various fields including smart home automation, healthcare, consumer applications and others.

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

Short range wireless IoT technologies include bluetooth mesh networks, Z-wave, ZigBee and Wi-Fi, as well as other lesser used technologies. Due to the inherent advantages that come with 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. Long range networks, on the other hand, are quite 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 when compared to the previous categories, these were left out of the analysis.

In terms of wired technologies used by IoT devices, ethernet remains the dominant general purpose networking standard. Although wired technologies provide advantages in terms of data transfer speed, they do 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 of time 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. This adds an additional economical constraint to IoT devices – they need to be made from relatively cheap components.

These constraints mean that IoT devices are limited when it comes to hardware resources. Hardware limitations come in three main forms – CPU power, memory and storage. Storage in the form of flash memory provides the hardest to solve problems when it comes to 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 the installation of 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 come from both the issues with firmware updates and generally insecure code. Most software written for IoT devices is written 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, to ensure privacy and general security we must ensure data integrity and confidentiality. The data that is sent must not be tampered with, nor snooped on during communication. This 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.

```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.*

2.3 The Rust programming language

Rust is a programming language is a modern systems programming language originally created at Mozilla designed to be a 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* systems in order 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 also 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 unsafe code to be compiled, 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. That is, the pointer is freed and then used in the print statement, resulting in undefined behaviour. While this example is fairly trivial, in a larger, more complex system these bugs are often very hard to debug and lead to security vulnerabilities. This issue does not only exist in code bases and organisations with low resources, at the BlueHat security conference, Microsoft researcher Miller (2019) presented that roughly 70% of all Microsoft's yearly patches were targeted at fixing memory safety bugs. On the other hand, the analogous Rust code will not compile due to the ownership system, mitigating this issue all together.

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 towards Rust in this project comes from two main aspects. Firstly, a memory safe language may circumvent the issues with security related bugs in IoT firmware and the supporting network stacks. As previously discussed, getting the firmware correct on the first try is important in IoT due to the difficulty that comes with updates. Secondly, assessing the performance of Rust implementations of the QUIC stack is important to solidify 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 Network simulation

In order to evaluate network performance two main options were considered: 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, simulation was chosen. 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 can be used to created software-defined networks (SNDs) using the *OpenFlow* standard.

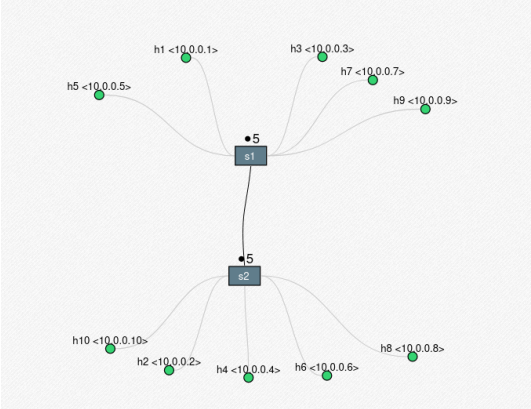


Figure 3.1: *The resulting Dumbbell topology with 5 hosts on either side of the switch. This topology simulates congestion on the link as the hosts have to share it for their data transfer.*

Using the Python API provided we created the network topology shown in Figure 3.1. The script used takes several parameters in order to create a simulation environment resembling a realistic scenario. The variables that the script changes between simulations to do so 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 on it. In contrast to bandwidth in signal processing, in computer networking bandwidth is typically measured in bits per second, rather than hertz. The delay of a link specifies the latency in terms of the time that a bit of data has to travel across a link and is typically measured in milliseconds. Link delay has a correspondence 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 rate of packet loss shows the percentage of packets that were corrupted or dropped in transit. Due to these packets being retransmitted in various protocols, this rate 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 the work conducted by Previdi et al. (2019) and in the case of ZigBee, the work by Alena et al. (2011).

Table 3.1: *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 between 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 especially difficult to find exact estimates for packet loss rates with most sources describing only what is a "good" rate for a stable connection (SDU 2013). Hence, the data are best estimates, cross-validated through the different sources and are not exact values.

Using the different links, a file of equal size was then transferred using the various QUIC implementations described in Chapter 4.

4 QUIC Implementations

5 Evaluation

5.1 Performance

5.2 Binary sizes

6 Conclusion

A | Data

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.

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.

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.

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.

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.