Investigating Feasibility of QUIC on Constrained Devices

Mikhail Dyuldin

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Bachelor of Science in Computer Science

Supervisor: Stefan Weber

August 2021

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Mikhail Dyuldin

August 9, 2021

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Mikhail Dyuldin

August 9, 2021

Investigating Feasibility of QUIC on Constrained Devices

Mikhail Dyuldin, Bachelor of Science in Computer Science

University of Dublin, Trinity College, 2021

Supervisor: Stefan Weber

This project aims to test the speed and performance of different protocols so that a comparison can be made between them and the function of Google's Quick UDP Internet Connections (QUIC). The protocols will be tested on two Espressif's ESP32 DevKit Cs which are low-powered and constrained microcontrollers, as well as on non-constrained devices that have been set up on my laptop so that the maximum capabilities of the protocols can be reached and used as a baseline of comparison to the function of the microcontrollers.

The tests will have emphasis on measuring the speed at which the protocols perform stress tests. There will then be an evaluation of their performance based off of the background information given in the State of the Art section, where I will go into the reasons why the protocols run differently from a baseline comparison and how different factors effect the measurements taken.

Acknowledgments

I would like to thank Stefan Weber, my supervisor and mentor for providing me direction and guidance over the last six months.

Mikhail Dyuldin

University of Dublin, Trinity College
August 2021

Contents

List of Figures

Chapter 1

Introduction

From the invention of Hypertext Transfer Protocols (HTTP) in the late 1980's to the present, there has not been a change to the protocol as big as HTTP/3 since its culmination. HTTP/3 will greatly improve the speed at which we browse the internet by revolutionising the way data gets sent from server to server. Greatly improving the speed at which vital tasks are completed, such as watching our favourite streamers to bouncing important information back from autonomous robots working in a factory.

This paper will explore if microcontrollers have the power necessary to shoulder this improved efficiency and speed and will discuss the possibilities of this new technology functioning on cheap and accessible pieces of hardware.

1.1 Structure of the Dissertation

- ***Chapter 2 - State of the Art -*** The necessary background will be provided on past HTTP versions and the innovations of HTTP/3 in the context of constrained devices as well as a discussion on the previous work by Lars Eggert's "Quant" project.
- ***Chapter 3 - Design of Experiments -*** Will show an overview of all eight experiments performed and the technologies used to set them up.
- ***Chapter 4 - Implementation -*** How the experiments were implemented and what issues were encountered along the way.
- ***Chapter 5 - Results and Evaluation -*** The results and a discussion on why they may be different from what was expected.
- ***Chapter 6 - Conclusions and Future Work -*** Final thoughts on the project and suggestions for future work.

Chapter 2

State of the Art

This section will go through the microcontroller, its coding environment used and an introduction to previous HTTP versions so that it is easier to understand the progress that has been made in this field. Afterwards we will focus on Google's QUIC implementation and a previous implementation of QUIC on a constrained device by Lars Eggert.

2.1 Hardware and Environment Used

The ESP32 DevKit C, which from now on will be referred to as "ESP" or "ESP32", contains 2 central processing unit (CPU) cores that have an adjustable frequency of operation from 80 to 240 MHz (default of 160MHz used). 4MB of flash memory and 8MB of Pseudo Static Random Access Memory (PSRAM) and is micro-USB powered. The Microcontroller Unit (MCU) come installed with a module called "ESP-Wrover-E" which allows communication over WiFi and Bluetooth. This capability is the main reason why this MCU was chosen over other models.

Espressif has a native open source framework called "ESP-IDF" which is what will be used in getting the programs to work on the ESP. The reason this framework was used over others such as "Arduino" is because ESP-IDF has hundreds of different configurations that can be changed for the running of specific tasks on the MCU, such as changing the protocol behaviour, debugging options, partition tables for storage and many more. Whereas for the Arduino framework is also open source but is made to be able to work on many different MCU's which means that the functionality available on it will be limited to the most basic functions on any given MCU. It is also designed to be easy to use for people that know how to program but aren't familiar with the hardware side of things.

2.2 HTTP

HTTP was invented by Tim Berners-Lee at CERN in the late 80's to early 90's. Based off of requests and responses, it is the underlying communications protocol for the World Wide Web (WWW), the standards of which are regulated by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (WWWC). The original version of HTTP was version 0.9. It could only perform GET requests and included no header information that would allow the transfer of different data types such as images and audio files. The responses were only in hypertext and there was no error codes available. This is the bare minimum of how the internet as we know it should work.

The next version of HTTP which is 1.0 released in 1996 and it didn't take long after that for the next version again to be released in 1997, updates to which were made in 1999 made it become the standard that most sites use to this day. The best way to explain how HTTP/1.0 works is by comparing it with HTTP/1.1 to see what features it is missing from its more modern counterpart while at the same time learning about the technology that is widely used today. Since there are a lot of changes, most of which are out of scope for this paper, we will only focus on the changes that relate to its performance as this is in line with the topic of our constrained devices.

The main performance improvement can be seen from the addition of persistent connections. In HTTP/1.0, much like its predecessor, terminated the socket or connection at the completion of a request. This negatively impacts the speed of subsequent requests and their completions because the Transmission Control Protocol (TCP) needs to do a 3-way handshake at the start of each request to establish a connection and then a costly termination of said socket. The comparison between the two HTTP versions can be seen in the figure below. 2.1.

TCP also has a congestion control method called "slow start" that gradually increases the amount of data sent out in order to prevent network congestion, it stops increasing once a limit is found.

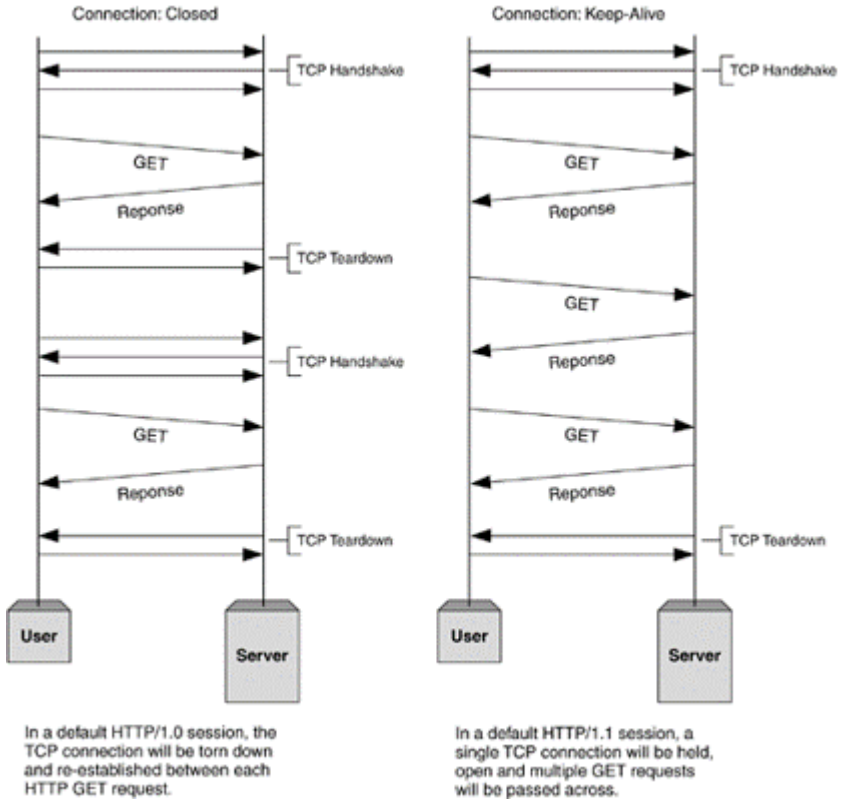


Figure 2.1: This shows a side by side comparison of how HTTP/1.0 and HTTP/1.1 both handle multiple connection consecutively. HTTP/1.1 handles many more requests in the same amount of time due to its "Keep Alive" header that tells the server that the connection is not finished yet. Credit to informit.com

Chunked Transfer Encoding is another improvement found in HTTP/1.1. This is useful because the server does not know the amount of data that it needs to send, so instead of buffering the data to find out how much there is to send, it just sends it instead. Delimiting it with a hexadecimal "\r\n" at the start of the chunk. Once there is no more data left to send, it delimits the chunk the same way at the end but this time with no data to follow it. This is called a "zero-length chunk"

The addition of pipelining is a major upgrade to HTTP/1.1, as it allows multiple requests to be sent without waiting for a response. This saves time because the client is not wasting time listening for a response. The server sends out the packets in the same order as it was requested, see figure below 2.2. If a packet is dropped then the server needs to wait for it to be re-transmitted before it does anything else. This is called a Head of Line (HOL) blocking.

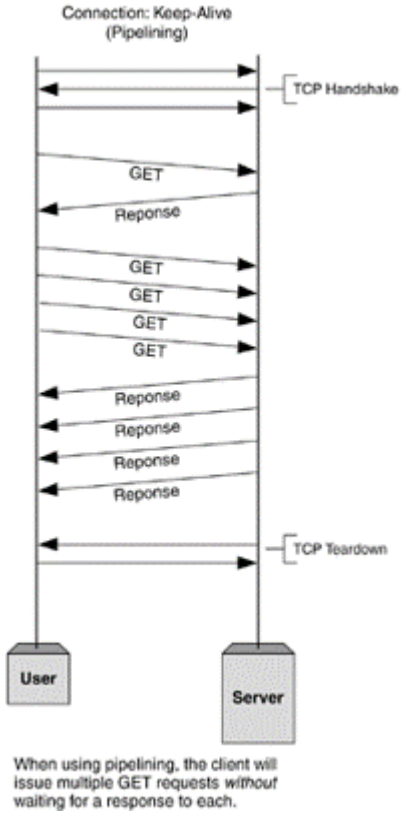


Figure 2.2: Http/1.1 Pipelining requests for greater efficiency. Credit to informit.com

2.3 HTTPS

The "S" in HTTPS stands for "Secure". Secure Socket Layer (SSL) is used to make HTTP secure and it is the reason why we can do things like online shopping and doctors appointments without all of our data being stolen. There has been such a push for every website to switch over to the secure method of browsing that Google actually increased a site's search ranking if it is secured [2]. While SSL does provide security, it also adds an extra second or so at establishment of a connection to verify the clients and server's certificates and there is extra processing required to encrypt and decrypt outgoing and incoming messages respectively. This adds up over time and a noticeable difference can be seen with larger and more numerous files. Although on non-constrained devices the difference is minute.

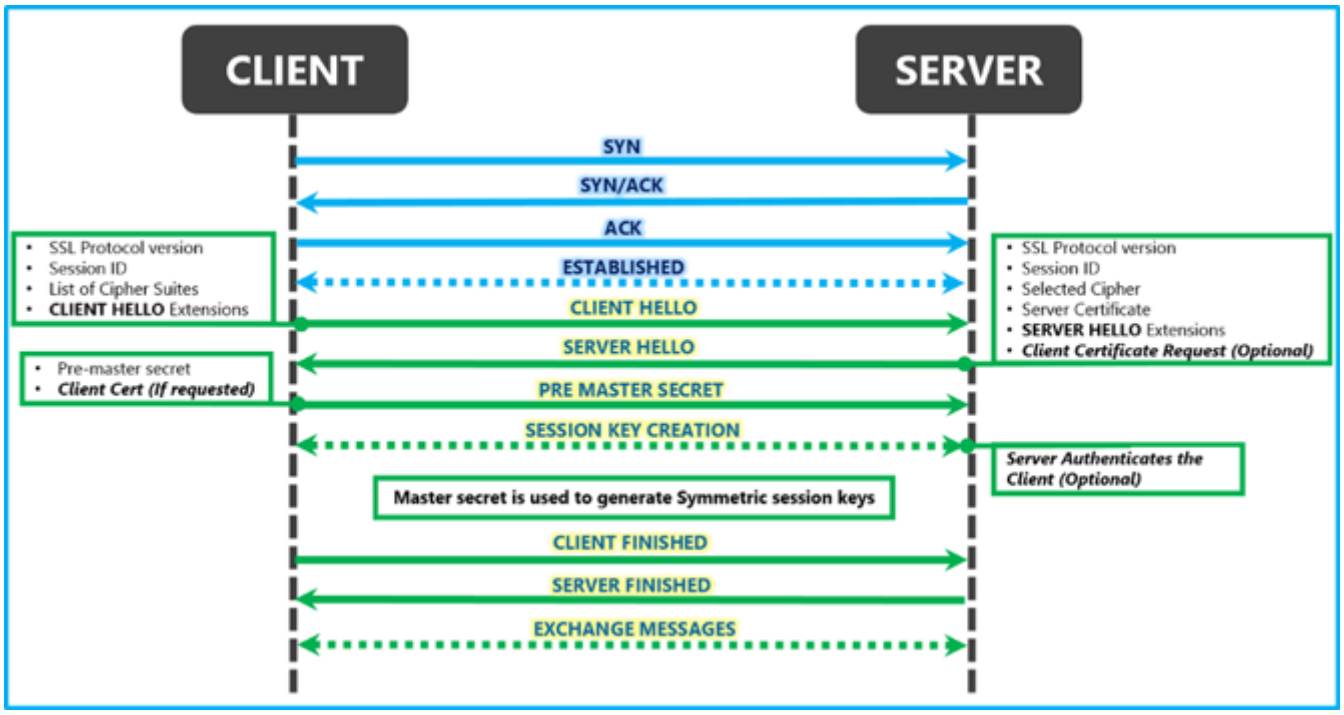


Figure 2.3: Here you can see the extra overhead that a secure connection adds to a TCP handshake. The blue arrows show a regular 3-way handshake and the arrows in green show the additional transmissions that need to be made in order to acquire a secure connection. Credit to msdn.microsoft.com

2.4 HTTP/2

HTTP/2 is a major iteration of the HTTP, in fact it was so big that they had to name it a whole number more as the new version includes a new binary framing level that is not backwards compatible with the previous version. Because of this packets sent across this protocol can be processed by a CPU much faster as it is in the CPU's native language. HTTP/2 will be the next standard used by most sites on the internet, at the moment it is already used by 45.9% of all the websites [3].

In HTTP/1.1, requests that are pipelined must be received in the same order so as not to get HOL blocked. In HTTP/2 this is improved upon by allowing multiple requests to be multiplexed together in a single stream instead of sending them out one by one. But the HOL blocking exists fundamentally in the TCP layer because it cannot assemble the request and act on them if say one packet out of six got dropped. That one packet would have to be re transmitted for the data to appear for the client. HTTP/1.1 can mimic this efficiency by opening multiple connections but this is at the cost of needing extra memory resources to keep those separate connections open.

There are many other improvements added besides that, header compression can lower

the overhead of packets which increases the throughput of a connection and a server can push essential data that the client will need in the future without it ever being requested in the first place. There are many more improvements but that is outside the scope of the paper.

2.5 Windowing

Taking the TCP/IP stack as an example, it places received packets into a buffer to be processed by the upper levels of the protocol and the same for packets that need to be sent out. Network buffers need to be approximately 1518 bytes each for it to fit Ethernet and TCP frame data, plus a header to describe the data that it holds. Originally the maximum frame size was defined by Robert Metcalfe in 1976 while working at Xerox, which is 1518 bytes [10].

TCP uses a flow control method called windowing. The capacity for a window to receive data is directly related to the hardware and processing ability. A value of zero for the window would halt the transmission and a window can have a maximum size of 65,535 byes (a 16-bit field).

Here I will reference the figure 2.4 when explaining the behaviour of windows.

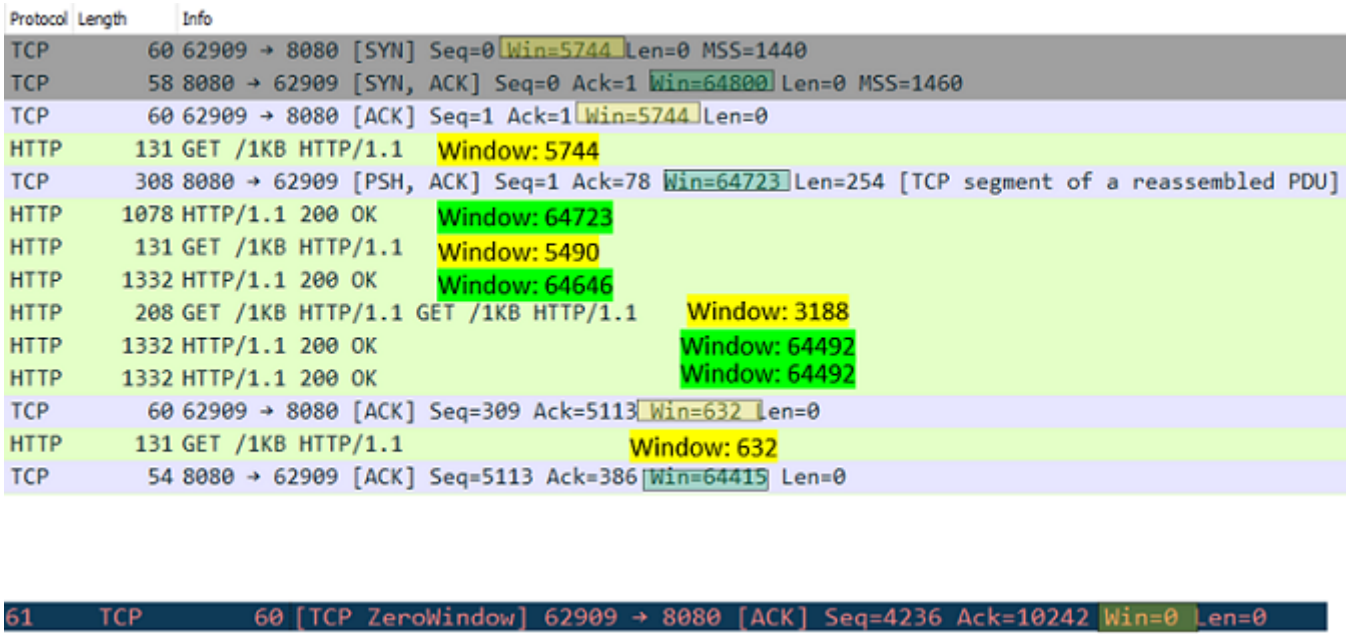


Figure 2.4: Here you will see communication between an ESP32 and an Nginx server on a desktop, the esp has a yellow highlighted window and the server has a green highlighted window. Followed by the first packet with a "ZeroWindow" warning that is dark in colour.

As you can see, Nginx has a max window size due to my laptop's superior memory,

while the ESP32 starts with 5744 bytes which is around 4 buffers worth ($4 * 1460 = 5840 bytes$) the missing bytes are the headers at 24 bytes per buffer.

Every time the server receives a GET request from the client, its window goes down accordingly by 77 bytes which is a small amount, holding only the TCP payload (pictured in figure 2.5 below). The window is freed when an acknowledgement is received from the client.

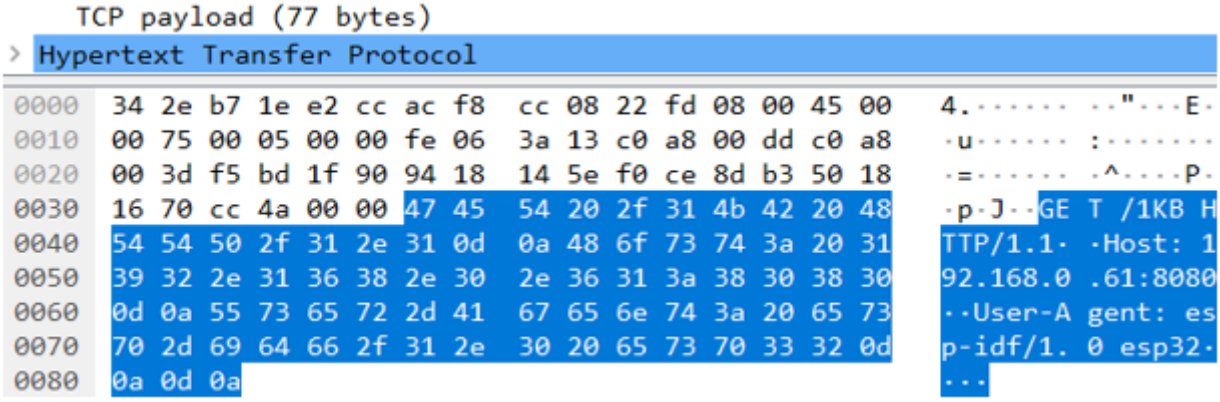


Figure 2.5: The highlighted area is the payload of that request, as seen through Wireshark.

The client on the other hand needs to work with a limited window size while receiving large packets that a pending processing by the upper levels of the protocol. This struggle with the client side window continues until less than 30 packets later the window runs out and transmission is temporarily halted, forcing a costly waiting time for the existing packets to be freed from the window by its processor.

The same type of flow control can be implemented for UDP but it would have to be programmed into the application layer where it would need to take care of undelivered packets.

2.6 What is HTTP3/QUIC?

HTTP/3 uses QUIC on its transport layer which is a transport protocol that is encrypted, multiplexed and low latency. It aims to significantly reduce the Round Trip Times (RTT) required in setting up a connection while also providing encryption on the packets being sent out and it solves the HOL blocking problem by multiplexing multiple streams of data on a single connection.

The protocol seems to have made improvements in every aspect of the HTTP but such leaps and bounds in progress do not come without its own cost and the cost of these improvements come from making changes directly to the transport layer, which has not

been done since the invention of HTTP. These changes are so low level that HTTP/2 cannot be directly integrated with QUIC because of the underlying frame mapping from application level to transport level is incompatible. Below in figure 2.6 you will see how QUIC fits into the traditional HTTPS stack and you will see how it uniquely positions itself to accomplish multiple functionalities within the stack.

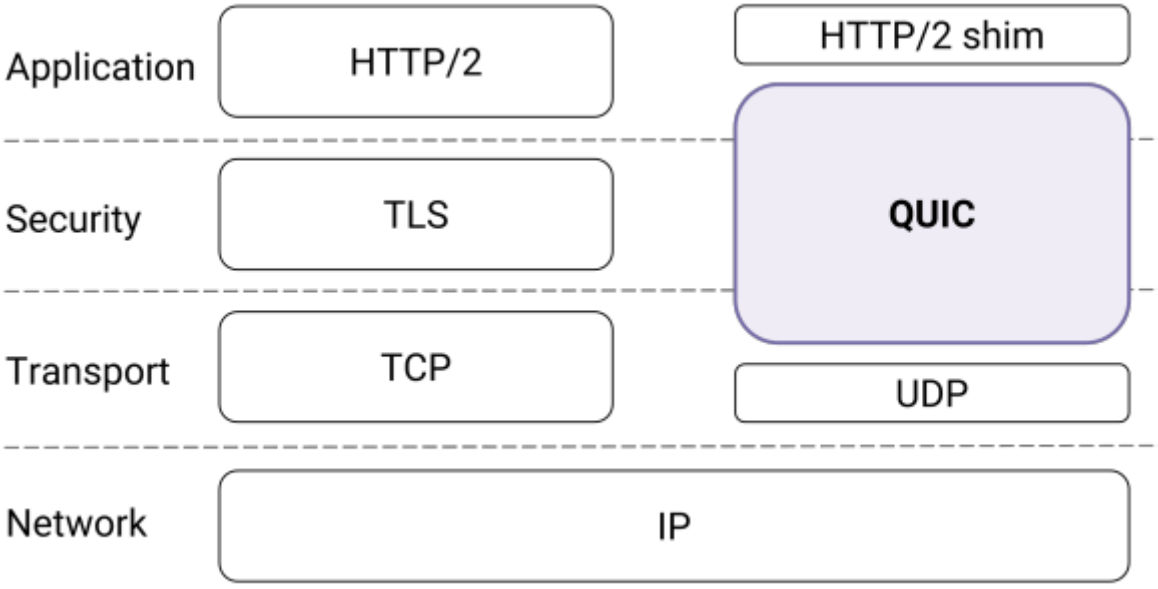


Figure 2.6: QUIC's position on the traditional HTTPS stack. Credit to QUIC team at Google

2.7 QUIC Multiplexing

TCP delivers its packets sequentially even with the improvement of multiplexing multiple requests into a stream. QUIC on the other hand is able to multiplex multiple streams onto a single connection. These streams are a lightweight abstraction that provide a reliable bidirectional bytestream that can transfer up to 2^{64} bytes on a single stream [7].

Each stream is identified by IDs. There can be either odd or even IDs, the odd IDs are assigned to streams that are initiated by clients and even IDs are assigned to those that are initiated by servers to avoid collisions. The initiation of a stream is implied when a new stream does not have any bytes sent across it yet and the closing of a stream is denoted by sending a "FIN" bit on the last stream frame [7].

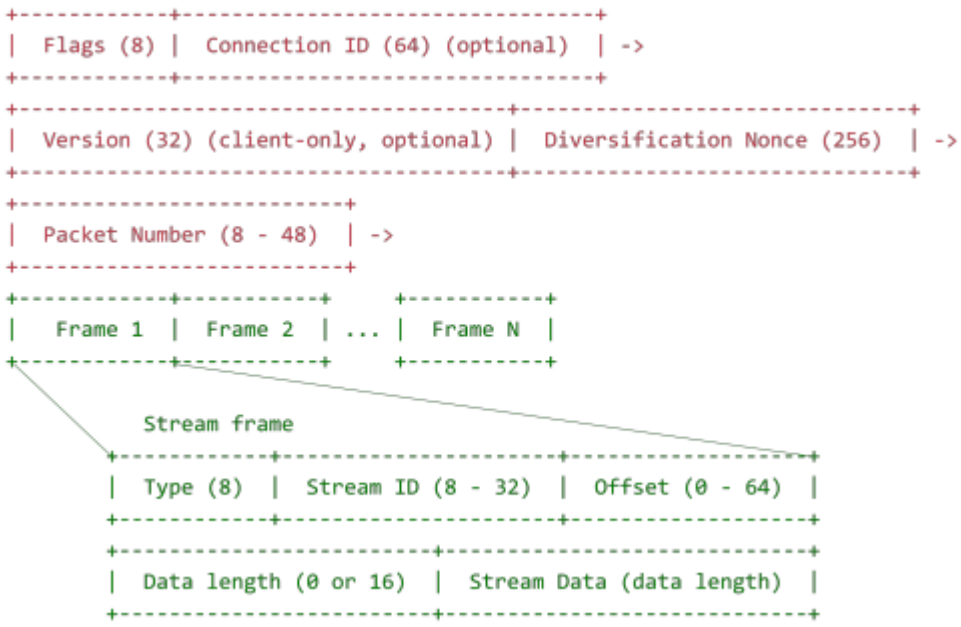


Figure 2.7: A QUIC Packet structure as of the 2017 version. Red is the unencrypted but authenticated header and green is the encrypted body of the packet. Credit to QUIC team at Google

2.8 QUIC Encryption

QUIC packets are fully authenticated and mostly encrypted. The figure 2.7 above illustrates the structure of a QUIC packet. The parts that are not encrypted in a typical packet are the parts that are needed for routing or for decrypting the packet itself, those are highlighted in red in figure 2.7, the description of which as a direct quotation from Google's QUIC paper:

"Flags encode the presence of the Connection ID field and length of the Packet Number field, and must be visible to read subsequent fields. The Connection ID serves routing and identification purposes; it is used by load balancers to direct the connection's traffic to the right server and by the server to locate connection state. The version number and diversification nonce fields are only present in early packets. The server generates the diversification nonce and sends it to the client in the SHLO packet to add entropy into key generation. Both endpoints use the packet number as a per-packet nonce, which is necessary to authenticate and decrypt packets. The packet number is placed outside of encryption cover to support decryption of packets received out of order, similar to Datagram Transport Layer Security (DTLS)"[11][7]

2.9 QUIC Flow Control

When a client is unable to release their receive buffers fast enough from the data that comes in different streams from the sender, then a type of HOL blocking occurs. To prevent one stream that slowly releases its buffer from taking up the whole receive buffer for that connection and blocking the other streams from receiving anything, QUIC employs connection-level flow control which limits buffer size for the whole connection on the receiver side and also stream-level flow control which limits the size of the buffers that any individual stream can use up. If a stream receives and processes data fast enough that the stream does not get clogged, then QUIC on the receiver side periodically sends out window update frames that tell the server that more data can be sent across that stream [7]. This is similar to the function of HTTP/2 and can be referred to as a credit-based flow control [4].

2.10 QUIC CPU Performance

Initially, QUIC was not written with a focus on CPU efficiency. Instead it was written to be easy to debug and to have rapid feature development [7]. When traffic was first served over QUIC for the Youtube platform it was found that the CPU-utilisation of QUIC was 3.5 times higher than that of Transport Layer Security (TLS)/TCP [7]. The main CPU load was used for cryptography, sending and receiving UDP packets and maintaining internal QUIC state [7]. These aspects received various optimisations to reduce the load they have on the CPU with a successful reduction from 3.5 times down to 2 times the CPU load of TLS/TCP [7]. Google's QUIC team expects the CPU load of QUIC to always be higher than that of TCP but they are confident that more reductions can be made to the load in the future.

But as it stands, microcontrollers are bottlenecked by their CPUs which stops protocols from running nearly as efficiently on systems with good CPUs. Will increasing the CPU load by twice the amount lead to the same requests completing over twice the amount of time? It might not be as simple as that but it is not far off the mark. The QUIC team at google have mentioned a few cases where the protocol is limited and one of the cases is very similar to the way my experiments were set up.

In the case of high bandwidth, low latency, and low-loss networks there are cases where the QUIC protocol performs worse than TCP and those cases are as a direct result of client side limitations of the CPU.

To quote Google's QUIC paper: "When used over a very high-bandwidth (over 100 Mbps) and/or very low RTT connection (a few milliseconds), QUIC may perform worse

than TCP. We believe that this limitation is due to client CPU limits and/or client-side scheduler inefficiencies in the OS or application. While these ranges are outside typical Internet conditions, we are actively looking into mitigations in these cases."[7] Unfortunately these are the exact conditions that my experiments operated under. This is exactly the reason that QUIC's performance gains are only marginal on mobile phones when compared to desktops[7], it is because when you compare the two, the CPU on the phone will seem constrained to the ones found on desktops. But when comparing most mobile phones' CPUs to that of any microcontroller, they are not nearly as close to being as constrained as that of a mobile phone; that gives you a sense of how much optimisation QUIC still needs to go through to get benefits from its use on MCUs.

2.11 Quant & My Project Before its Pivot

Originally my project involved studying Lars Eggert's (Current Chair of IETF) and Christian Huitema's (President of Private Octopus Inc.) projects called Quant and Picoquic respectively and then I was to choose which one of the projects I wanted to port over to ESP32's native environment ESP-IDF. Both projects were lacking in documentation which made understanding what they did very difficult, so I based my decision off of which project had less files and folders so that I would be able to wrap my head around them easier. In the end I chose Quant.

Quant also has a paper related to it that was published by Lars Eggert [6], it detailed the work that went into getting Quant to run on a Particle Argon and ESP32-DevKitC V4 microcontrollers, they ran Device-OS and RIOT-OS respectively. The paper mostly dealt with the storage, compute, memory and energy requirements of the Quant QUIC stack on those two platforms and found that a minimal standards-compliant QUIC client required approximately 58 to 63 KB of flash and could retrieve 5 KB of data in 4.2 to 5.1 seconds[6].

Out of curiosity I replicated the 5KB transfer experiment but running with TCP on a secure connection to a server that is running on my laptop on the same Wireless Local-Area Network (WLAN) which is the same description as from the paper[6]. My result was a transfer that on average completed in around 0.7 seconds, this is not including the "FIN" acknowledgment which takes exactly 5 seconds to appear after the data has already been received and decrypted by the client. This shows that the TCP still outperforms QUIC by a large margin, although this is a comparison made to the minimal Quant client which probably effects the speed negatively.

Getting back to what my project was like before the pivot; I had some stretch goals planned for when I was able to implement Quant, such as having multiple connections

open and attempting to implement encryption. After studying the project for many weeks and failing to compile it on Windows, I had an email exchange with Lars, who said that he never compiled it on Windows. So after installing Linux on my machine and failing to have it compile again, Stefan then attempted to compile it on MAC and Linux without success.

I knew from seeing an email exchange with a previous student who attempted this project but did not complete it, that Quant and Picoquic had picotls as a dependency but that wasn't available on ESP-IDF, instead there was a module called mbedtls which was similar to picotls so porting wouldn't be impossible. But when Stefan was trying to compile it on his end, he found out that Quant also has a picoquic dependency which is slightly bizarre as that is the other project that I was looking at and it also had an OpenSSL dependency which would be much more difficult to port over to ESP-IDF. That is when we decided that the project is not viable for my time frame so a discussion was had about pivoting the project to a different direction.

Chapter 3

Design Of Experiments

My experiments were designed to test the speed of transferring different sized files over HTTP/1.1/TCP and UDP by running batch tests that sent 1KB, 10KB, and 100KB files 55 times consecutively from server to client. HTTPS was also tested on the TCP experiments in order to get an idea of how costly the SSL is on the CPU but unfortunately was not able to get Datagram Transport Layer Security (DTLS) to work on the UDP experiments due to it being a stretch goal that I did not have time to complete, along with testing HTTP/2 as well.

3.1 Overview of the Approach

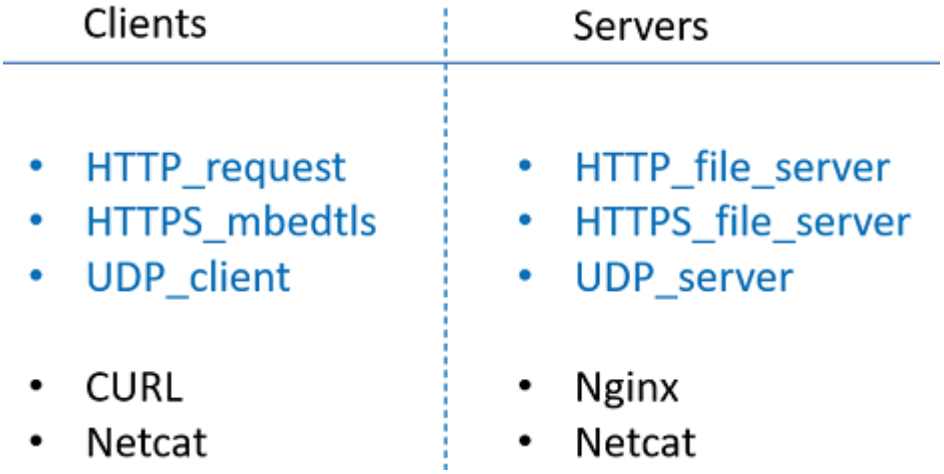


Figure 3.1: Here is a list of all the programs that I used in order to conduct my experiments. Programs highlighted in blue run on an ESP32 device while the ones that are in black are run off of my laptop.

The ESP programs all run on ESP-IDF and for the most part they were launched

through the ESP-IDF command line which allowed me to flash and monitor my programs on the ESP32s. I modified their code through VScode with the Espressif IDF extension installed. Once set up properly it allowed me to explore the code base easily which was useful for debugging, (although the set up often broke on my Windows machine due to updates to Windows and even though it was properly installed it barely operated as it was supposed to with certain bugs that forced me to use the command line instead towards the end of the project). The programs were configured through an accompanying GUI configuration tool which allowed me to search for various options.

Then I had to set up the non-constrained servers and clients. For the server I used Nginx which was set up through Docker, where I set up the unsecured and secured servers in separate images. The secured server was difficult to set up for me as it had to do with moving the config file from the Linux environment onto my Windows machine through Docker, and then to copy back over after making some changes to it, along with a key pair that I created. There was hardly any tutorials online on how to use Docker like this so in the end Stefan helped me configure the server through a pair programming session.

I used Netcat which is a networking utility that can be used directly through a command line or accessed through scripts. It is able to listen and write to network ports on a given Internet Protocol (IP) in TCP or UDP. It was used only for the UDP experiments as clients and servers.

And Finally I used Client URL (CURL) which is a command line tool that can make requests to servers through different protocols. I used it as my non-constrained client for my TCP experiments.

Chapter 4

Implementation

This sections will mostly go through the problems that I encountered during the set up of my experiments, such as bugs, network problems and certain caveats that need to be taken into consideration when viewing the results section of the paper.

4.1 HTTPS Clients for ESP32

When setting up the HTTPS client and running some test connections, I encountered a problem with not being able to set up a secure connection to secure server. This was because my server was operating under self-signed certificates, these are certificates that are made on the machine through a command line interface. The issue lies in the way that the HTTPS client authenticates the certificates, it cross checks the certificate with a list of Certificate Authorities (CA). These are organisations that are seen as trusted entities and only their certificates will be accepted by a client and a secure connection will then be allowed to open. I could have modified the list of CAs on my client and included my own self signed certificate onto its list of trusted certificate providers, or I could have gotten a free certificate from LetsEncrypt which is a CA that hands out certificates that last 90 days but needs to be set up on the server for automatic renewal[1]. But while looking for an alternate solution, I found a different example program that used the mbedtls library which gave me the ability to allow unsecured connections. This doesn't change the speed of acquiring a connection when compared to secured servers.

4.2 ESP32 Wireshark Visibility Issue

There was an issue with the programs running on the ESPs that they communicated with each other only which meant that I could not see any packets being sent from one to the

other. This issue was not present when connecting an ESP to a non-constrained server. So a modification had to be made to the mainline of each program, the modification changed the default IP communications mask, which tells the ESP what range of IPs it is allowed to communicate with. This piece of code is visible in listing 4.1 below.

```
void app_main(void)
{
    ESP_ERROR_CHECK( nvs_flash_init() );
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    ESP_ERROR_CHECK(example_connect());

    esp_netif_t* netif =
    esp_netif_get_handle_from_ifkey("WIFI_STA_DEF");

    if (netif) {
        esp_netif_ip_info_t ip_info;
        esp_netif_get_ip_info(netif, &ip_info);
        esp_netif_dhcpc_stop(netif);
        esp_netif_set_ip4_addr(&ip_info.netmask,255,255,255,255);
        esp_netif_set_ip_info(netif, &ip_info);
    }

    ESP_LOGI(TAG, "IP4 address set successfuly");
    xTaskCreate(&http_get_task, "http_get_task",
    4096, NULL, 5, NULL);
}
```

Listing 4.1: Changing the default IPv4 mask to force communication on the the IP specified in the configuration instead of having the ESPs communicate directly. The code that made the change is the line above the if statement and the if statement itself.

By changing the IPv4 mask from 255,255,255,0 to 255,255,255,255 it tells the ESP32 that it cannot communicate in any range and instead it can only communicate with the IP specified in the application layer through WiFi. After making the change and connecting my ESP32s acting as client and server to my laptop's hotspot; the client ESP sends a request to the server ESP and the server receives it, the hotspot will re-transmit the

same message again to the destination and it will show up as a duplicate packet then periodically the server sends the packet back to who sent it first and tell the sender that the destination is a straight shot to the recipient. Basically trying to force the two ESPs to talk to each other directly and not through the hotspot. This tells us that the hotspot is processing the information before sending it forward, increasing latency.

This problem was seemingly so novel that I could not find any resources online to explain it, let alone code a solution for it. After some pair programming sessions with Stefan and a helpful StackOverflow user called "Tarmo", I was able to get this solution to work.

4.3 Hotspot-added Latency

When I had the hotspot turned on, I noticed my ping/latency being double what it is normally and that prompted me to investigate the cause. Below in figure 4.1 I make a comparison of downloading files over the hotspot to downloading files directly from one ESP to another in both cases.

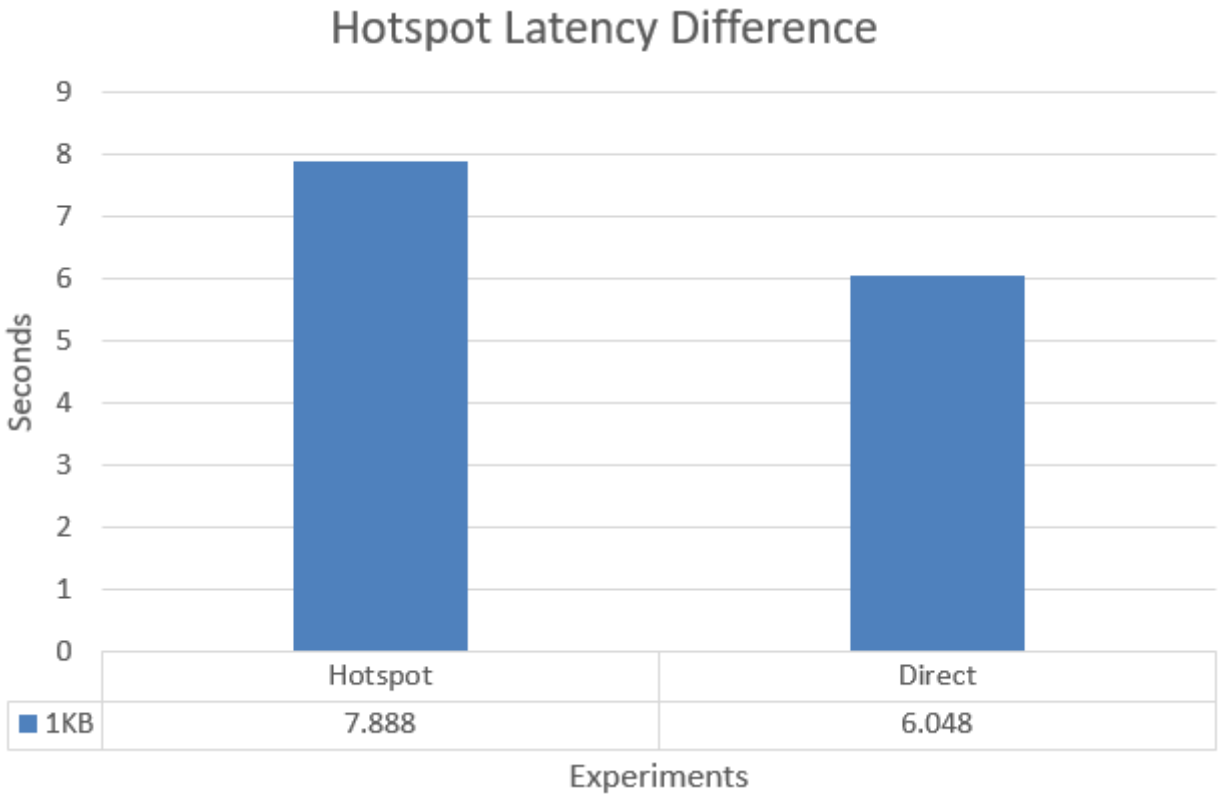


Figure 4.1: Here is 1KB file downloaded from one ESP to another on different networks. The average value from five readings was used to plot this bar chart.

I have come to the conclusion that when the hotspot is activated on my laptop's network card; every second clock tick is used to provide the function of giving a hotspot connection and every odd tick is used to provide my laptop with internet access. You can clearly see that the speed at which information gets delivered is negatively impacted by this as the time it took to transfer one batch of 55 files sized 1KB each reduced down to just over six seconds on the direct line, a difference of around 1.8 seconds. This ratio of slower transfer speed can be expected on every other experiment performed except for the baseline experiments that deal with non-constrained client to non-constrained servers.

4.4 Making the HTTPS File Server

The http_file_server came with the rest of the example programs provided. It allowed an ESP32 to act as a server to which files could be uploaded to and downloaded from. It used Serial Peripheral Interface Flash File System (SPIFFS) which is a very simple file storage system for flash memory. However there was no complimentary file server that could be run over a secure connection, instead I had to code that myself by adding self signed certificates to the program and replacing the HTTP library with the HTTPS one and making sure all the relevant functions calls the correct library. The HTTPS functionality also had to be enabled in the configuration file which I did not know for the longest time when trying implementing it, but with the help of Stefan's keen eye I got it to work eventually.

4.5 HTTP/1.0 Attempt at Measuring

I intended to test every TCP connection with its performance on the batch tests through HTTP/1.0 as well as HTTP/1.1. But after collecting the data on 1.0 I noticed that the responses were in HTTP/1.1. I thought it would be enough to change the HTTP version on the request but that didn't seem to be the case. After doing some searching around, I found that this behaviour was in line with Request for Comment (RFC) 2145 which is a publication body of the standards that have been set by everyone involved with the making of the internet, the most prominent of which is the IETF. I found out that the server is supposed to send out a more up to date response to the client as a way of telling them that the server is capable of communicating in a more up to date protocol. The response is supposed to be modified in such a way that a client with only HTTP/1.0 capabilities should still be able to read it without throwing any errors.

But after looking at my experiment where ESP requests another ESP through HTTP/1.0,

I noticed that the connection does not break down at the end of every response, which is not the type of behaviour you would expect from HTTP/1.0. When I attempted to run a similar experiment but this time the ESP requests from an Nginx server from my laptop, the server was not able to provide a proper response and the ESP received no data. This tells me that more configuration needs to be done to both the ESP and the Nginx server, but as I was busy with getting the other experiments to work at the time, I weighed the usefulness of these experiments and decided to abandon them as they wouldn't be worth the time and effort to get working.

Chapter 5

Results and Evaluation

Here are the graphs from the obtained results of the experiments; the time it takes for 55 packets of various sizes to complete are measured in seconds. Each graph includes observations on the results where an attempt is made to explain the numbers that were obtained.

5.1 100KB TCP Batch Tests

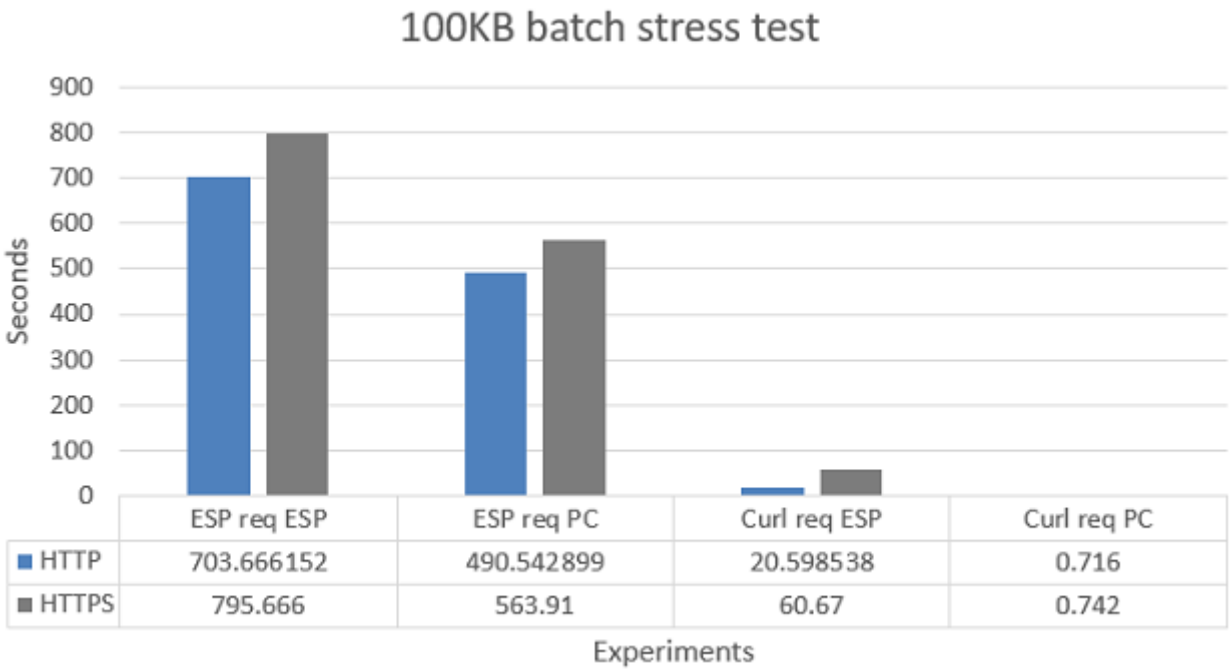


Figure 5.1: The results for the 100KB batch test over HTTP/1.1 with and without SSL

From looking at the results obtained in figure 5.1 we can see that in the experiment

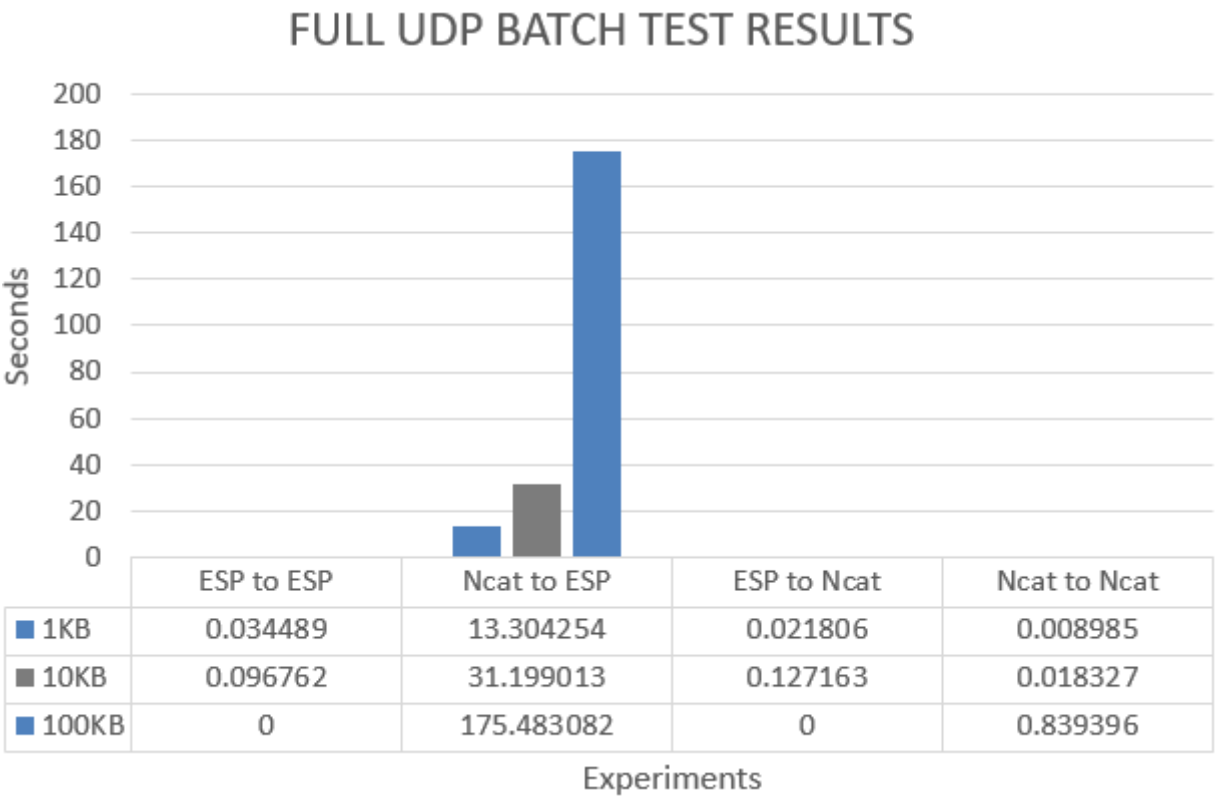
"ESP req ESP" the time it took for the batch test to complete is by far the longest out of them all and that is to be expected from two constrained devices trying to send data.

When ESP requests Nginx, the performance is much better than the last one since the speed of transmission is better but we are still limited by the processing of the receive buffer of the ESP device.

For the CURL requesting the ESP experiment, it performs surprisingly fast, showing us that the ESP performs well with transmitting data to a non-constrained server. This tells us that there is less processing power requirement from sending data rather than receiving it.

Curl requesting Nginx acts as a baseline of comparison, this is ultimately what the microcontrollers should strive to get although realistically they will never be as fast. Each experiment also shows a slight increase in the time it takes for the HTTPS counterpart to complete, this increase is in line with what is expected from HTTPS connections[9].

5.2 All UDP Batch Tests



	ESP to ESP	Ncat to ESP	ESP to Ncat	Ncat to Ncat
1KB	0.034489	13.304254	0.021806	0.008985
10KB	0.096762	31.199013	0.127163	0.018327
100KB	0	175.483082	0	0.839396

Figure 5.2: The results of "Ncat to ESP" are an outlier compared to the other test results, however the results of the other experiments can be derived from the table of figures below the bar chart.

Notice how the title of each of the UDP experiments is different than their TCP counterparts in figure 5.2? This is because UDP sends out a datagram without receiving a request first as this is one of the things that fundamentally differs from TCP. All of these experiments except for "Ncat to ESP" show that the speed of transmission and receiving is very high in comparison to TCP. "ESP to ESP" gets surprisingly close to the baseline speed of "Ncat to Ncat". However with "ESP to ESP" and "ESP to Ncat" the ESP failed in both cases to complete the 100KB batch tests because they both threw an error saying that they ran out of memory. The reason this didn't happen to the TCP experiments is because TCP has more robust flow control methods in place, which stops the transmission when its buffer is full, whereas the UDP seems to have a minimalistic flow control where is simply terminates the transmission when it reaches the same problem. More configuration is needed and could be a good candidate for future work that succeed this project.

As for "Ncat to ESP", I could not find a reason as to why it was taking so long to complete its batch tests. There was no communication back and forth to negotiate the speed of transmission. The UDP packets for 10KB and 100KB tests were fragmented into packets sized 1514 bytes but the same happened for "ESP to ESP" and "ESP to Ncat" and those delivered much faster.

5.3 ESP Requests ESP: TCP Tests

The results below in figure 5.3 are as expected. The time elapsed rises exponentially with the exponential rise in the size of the data transmitted, with HTTPS taking slightly longer as usual.

5.4 ESP Requests PC: TCP Tests

The results in figure 5.4 below are slightly faster than the previous experiment as explained in the "100KB TCP Batch tests" section.

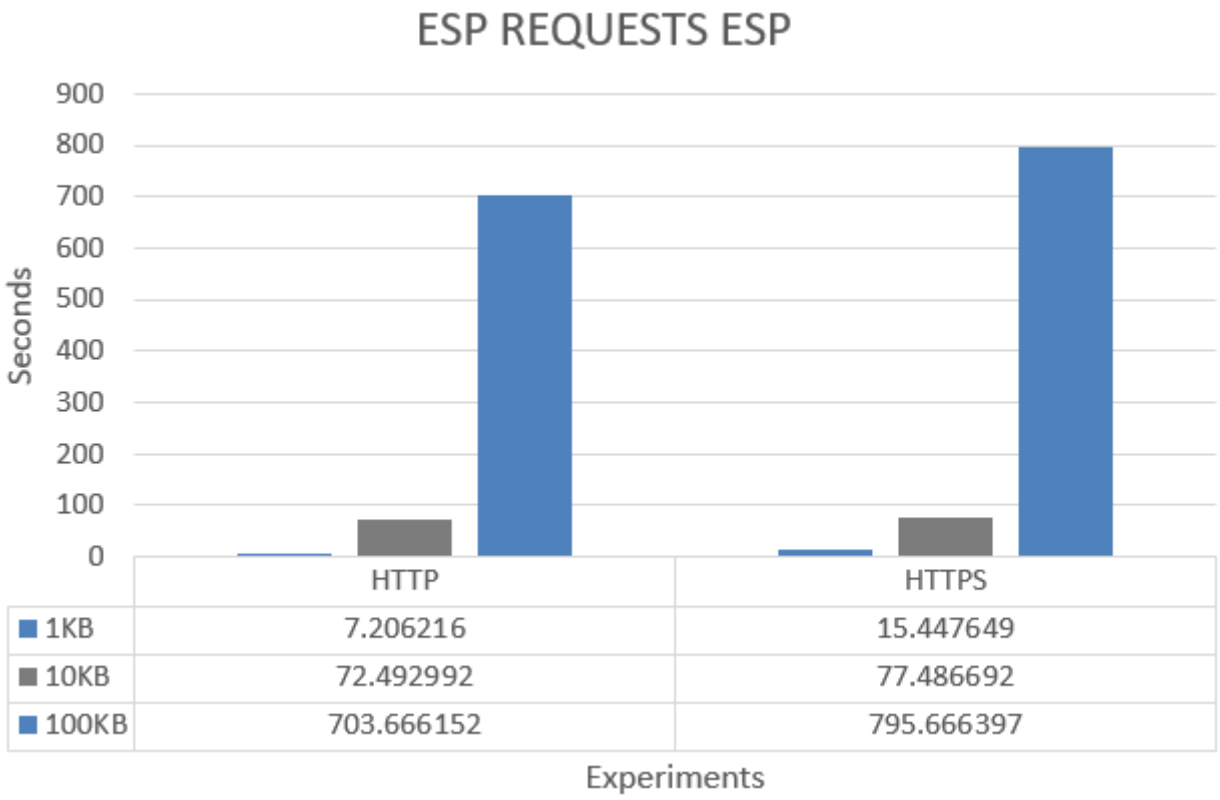


Figure 5.3: Here are all the batch test results done for TCP from one ESP to another.

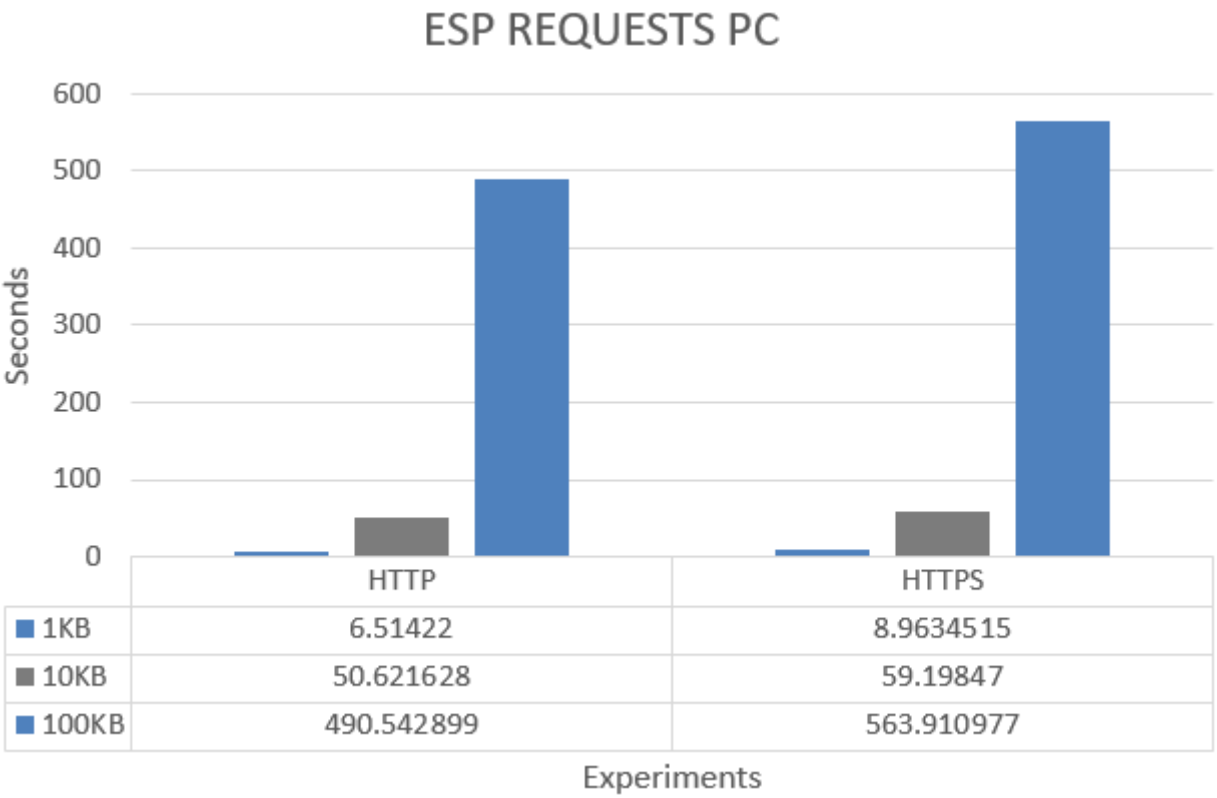


Figure 5.4: Here Are the full test results for every batch test for when ESP requests a server on the PC

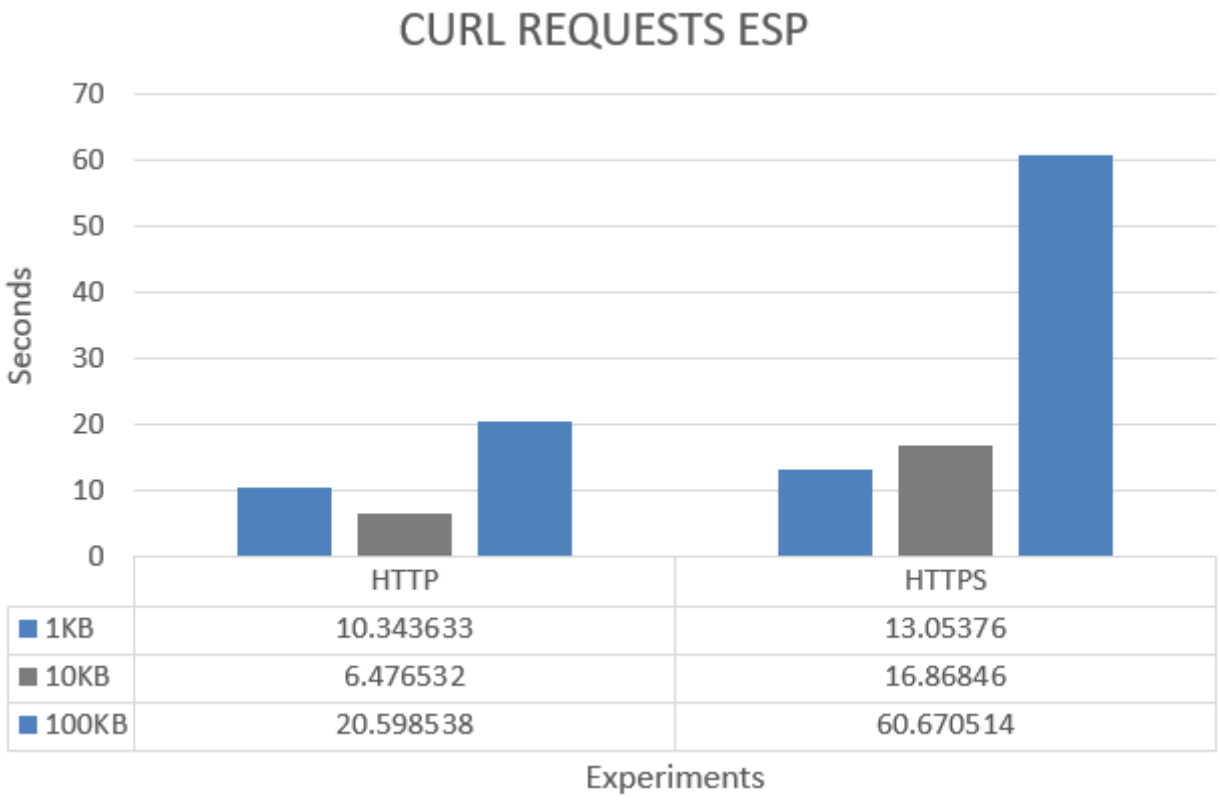


Figure 5.5: Here are all the batch test results for the experiment where CURL requests an ESP

5.5 CURL Requests ESP: TCP Tests

It takes longer than expected to transmit the 1KB file in figure 5.5 above. I suspect that it is because of the TCP slow start. It is less prominent of a difference with the HTTPS because the slow start is used for the HTTPS handshake. I am not sure why this behaviour was not present in the last experiment, perhaps the slow start functionality was not configured properly on the ESP requests? Or more likely is that it is present and it struggles enough with the 10KB tests that it is not noticeable as the ratio of difference in times between the 1KB and 10KB times is very high in "ESP requests PC" and the difference is not so high in this one.

5.6 CURL Requests PC: TCP Tests

The non-constrained client and server seem to operate as expected for these batch tests in figure 5.6 below. However the rise in the time taken for the 100KB batch test is much more than exponential and the only explanation I can think of is the buffer fills up more than the processor can handle and thus the transmission of the files is throttled.

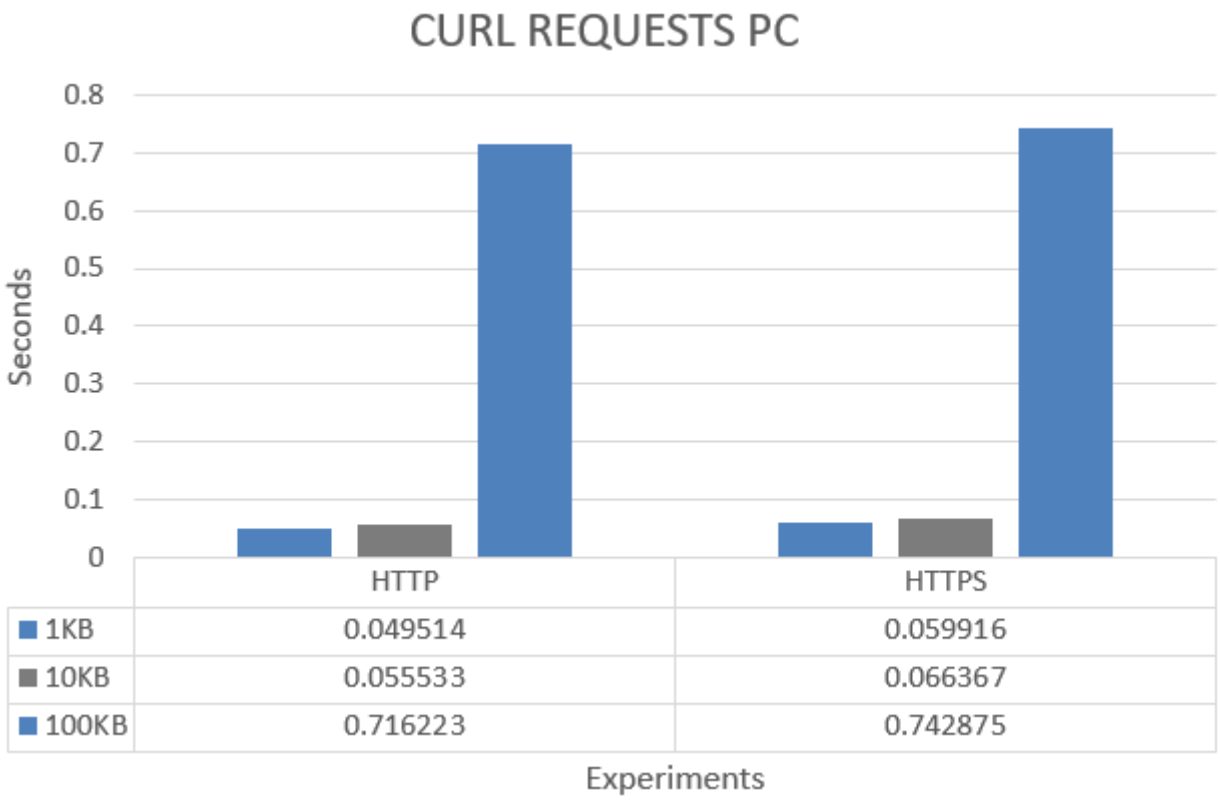


Figure 5.6: This is the final result and it shows you the baseline of what you can expect from a non-constrained client and server.

Chapter 6

Conclusions & Future Work

In conclusion, the advancements made to HTTP/2 and 3 solve a lot of the performance issues found with HTTP/1.1, such as HOL blocking disappearing with QUIC due to its multiplexed streams and the compression of headers which reduce the load on receive buffers. However from studying Google's QUIC paper and Lars Eggert's implementation, we see that the processing load of QUIC is still at least double that of TCP, hindering the transmission of information so much that it takes around five times the amount of time to transfer a 5KB file. The biggest issue with microcontrollers is still not solved, and that is that they simply do not have the CPU power or RAM availability to efficiently run QUIC. Networks nowadays are very efficient and they do not drop packets as much as they used to, however this technology could be useful for aircraft communication. The packet loss rate from ground to air is between 50%-70% packet loss [8], this is quite a large amount. A "good" loss rate for streaming music would be considered 1% and poor at 2.5%-5% [12], therefore aircraft could definitely benefit from QUIC but they would not be using constrained devices for their form of communication due to safety concerns and therefore would not experience the only major downside of QUIC.

As the technology stands today, I would not recommend running QUIC on microcontrollers that rely on speed of transferring information. If that is not the goal of an MCU's and instead a reliable form of communication is desired on a lossy network, then it could find a use since, for example, websites would load faster on a very lossy network with QUIC instead of TCP due to the lack of HOL blocking. However that is only taking the technology at its current state. Google's QUIC team is confident that more improvements will be made to the efficiency of QUIC's CPU usage, and while QUIC may never use less CPU than TCP, it could reach a point in the future where it is only a bit more costly than TCP, and then there would be many more uses on an MCU than there are currently; Such as swarms of quad-copter drones all communicating together through a massive lattice of

QUIC's multiplexed streams with multiple connections to complete one single task such as all of them working together to lift large objects or survey cities, or smart home features that respond almost instantly, allowing an orchestra of microcontrollers operating together to make magic a reality. With QUIC you would not need expensive wires to achieve the same result if everything communicates over the air almost instantly.

Think about the time it takes for a robot to see a frame through its camera, then that frame needs to get processed by its CPU to perform object recognition and then it needs to calculate its own state in relation to whatever it saw through its camera; it needs to do many things before it even sends the data to a server. Now imagine a swarm of robots all doing the same thing and sending their information to other robots around them, they need to react to that information to avoid collisions and to complete tasks together, a packet loss could send them crashing into each other but with QUIC you get reduced latency and much less impact from packet losses, which allows any robots that we make to have a reduced margin of error in their movements. This is exactly the type of technology that will enable self driving cars to drive without traffic lights which leads to a world that we would not recognise today because the possibilities are endless.

6.1 Future Work

Since QUIC is not there yet in terms of processor load on MCUs, and most microcontrollers are already operating on a network that is not very lossy; it would be wise to ignore the benefits of HOL blocking and instead attempt to use encrypted UDP communication over the existing HTTP/2 standard. The main CPU expense is the multiplexed streams with header compression. But if someone was to remove that and instead focus on fast packet transfer with added flow control to account for the occasional lost packets, then a very useful protocol can be made that would have many uses in the present day. microcontrollers could have an ability to get close to what I described above, just not quite all the way there. If the future work is done on an ESP32 device, then it would be worth attempting to expand the buffer size for that protocol by utilising the 8MB of Psuedo Static RAM (PSRAM) that is already present on them.

Chapter 7

Working in the Pandemic

This dissertation was carried out and written during a lockdown while residing in Ireland, who had one of the most stringent lockdowns in Europe [5]. The toll it has had on my mental health is hard to explain. there were times when I thought that I was going insane every second week, to certain months where I seemed content living my antisocial life, but it is safe to say that it has had an overall negative impact on my mental and physical health. The variety of people that I have seen over almost two years has been the lowest in my whole life to the point that I have also noticed my social abilities declining, not by a lot but the difference is there!

Fully remote work has enabled me to become very "lazy", it is hard to keep a healthy routine when everyone has been greatly encouraged to stay inside for almost two years. It would benefit me to talk to my peers in college to know that my struggle is not mine alone, a sense of solidarity can go a long way when you are faced with a monumental task. College is about meeting people from different walks of life and working together to make something amazing that we otherwise couldn't do alone. Remote work is here to stay in the professional world but it shouldn't stay for the college world; it is a place where you learn more than just the subject you are studying and that is an important part that shouldn't be lost in the post-pandemic world.

With that said, I cannot wait for things to have some semblance of normalcy again and my heart goes out to all the first year students who had to start college during the lockdown; I cannot image just how isolating and painful that is for them to move away from home only to not be able to meet the people you are working with.

Bibliography

[1] Why ninety-day lifetimes for certificates?

[2] Https as a ranking signal, Aug 2014.

[3] Usage statistics of http/2 for websites, 2021.

[4] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext transfer protocol version 2 (http/2), 2015.

[5] Paul Cullen. Ireland had eu's most stringent lockdown this year, analysis finds, May 2021.

[6] Lars Eggert. Towards securing the internet of things with quic. In *Workshop on Decentralized IoT Systems and Security (DISS). https://doi. org/10.14722/diss*, 2020.

[7] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.

[8] Ying Loong Lee. Packet loss rate, 2015.

[9] Lori Macvittie. Stop. just stop. https is not faster than http.

[10] Robert M Metcalfe and David R Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[11] Eric Rescorla and Nagendra Modadugu. Datagram transport layer security version 1.2. 2012.

[12] ICTP Science Dissemination Unit. Ictp-sdu home page.

Appendix A

.1 Link to Code

Code, captures, and results: https://github.com/MuxDed/FinalYearProject.git

Appendix B

.1 Abbreviations

- ***UDP - User Datagram Protocol***
- ***QUIC - Quick UDP Internet Connections***
- ***HTTP - Hypertext Transfer Protocol***
- ***CPU - Central Processing Unit***
- ***RAM - Random Access Memory***
- ***PSRAM - Psuedo Static RAM***
- ***MCU - Microcontroller Unit***
- ***IETF - Internet Engineering Task Force***
- ***WWWC - World Wide Web Consortium***
- ***TCP - Transmission Control Protocol***
- ***SSL - Secure Socket Layer***
- ***DTLS - Datagram Transport Layer Security***
- ***RTT - Round Trip Times***
- ***HOL - Head of Line***
- ***WLAN - Wireless Local Area Network***
- ***IP - Internet Protocol***
- ***CURL - Client URL***
- ***CA - Certificate Authority***

- ***SPIFFS - Serial Peripheral Interface Flash File System***
- ***RFC - Request for Comment***