**IOT communication protocols for extreme conditions at remote locations.**



Marijn Verschuren 05-05-2023

# Table of contents

[1. Table of contents 2](#_Toc1)

[2. Table of figures 2](#_Toc2)

[3. Introduction 3](#_Toc3)

[4. HTTP 4](#_Toc4)

[4.1. TCP 4](#_Toc5)

[5. CoAP 8](#_Toc6)

[5.1. UDP 8](#_Toc7)

[6. Implementation 9](#_Toc8)

[6.1. Measurement setup 9](#_Toc9)

[6.2. HTTP client 10](#_Toc10)

[6.3. CoAP client 12](#_Toc11)

[7. Bibliography 13](#_Toc12)

# 2. Table of figures

[Figure 1. OSI model 4](#_Toc1)

[Figure 2. IP packet 4](#_Toc2)

[Figure 3. TCP packet 5](#_Toc3)

[Figure 4. TCP handshake 6](#_Toc4)

[Figure 5. TCP data transfer 6](#_Toc5)

[Figure 6. TCP packet loss 7](#_Toc6)

[Figure 7. TCP packet timeout 7](#_Toc7)

[Figure 8. UDP packet 8](#_Toc8)

[Figure 9. Measurement setup 9](#_Toc9)

[Figure 10. External resistor 9](#_Toc10)

[Figure 11. Connect to WiFi HTTP 10](#_Toc11)

[Figure 12. Connect to HTTP server 10](#_Toc12)

[Figure 13. HTTP requests 10](#_Toc13)

[Figure 14. HTTP serial output 10](#_Toc14)

[Figure 15. HTTP network activity snapshot 11](#_Toc15)

[Figure 16. HTTP server response 11](#_Toc16)

[Figure 17. HTTP power consumption 11](#_Toc17)

[Figure 18. Connect to WiFi CoAP 12](#_Toc18)

[Figure 19. CoAP request 12](#_Toc19)

[Figure 20. CoAP network activity 12](#_Toc20)

# 3. Introduction

This paper aims to determine the best IOT protocol for remote glacier size monitoring to track the effects of climate change on water levels globally. The protocols that were chosen for this research are HTTP and CoAP because of their similarities and relative simplicity. Since the project revolves around a remote application low power operation is held in high regard, other important metrics include expandability and reliability, these points are crucial to a maintenance free deployment. This paper however will not speak on the physical construction and deployment of the sensors and will merely compare the chosen IOT protocols for this application.

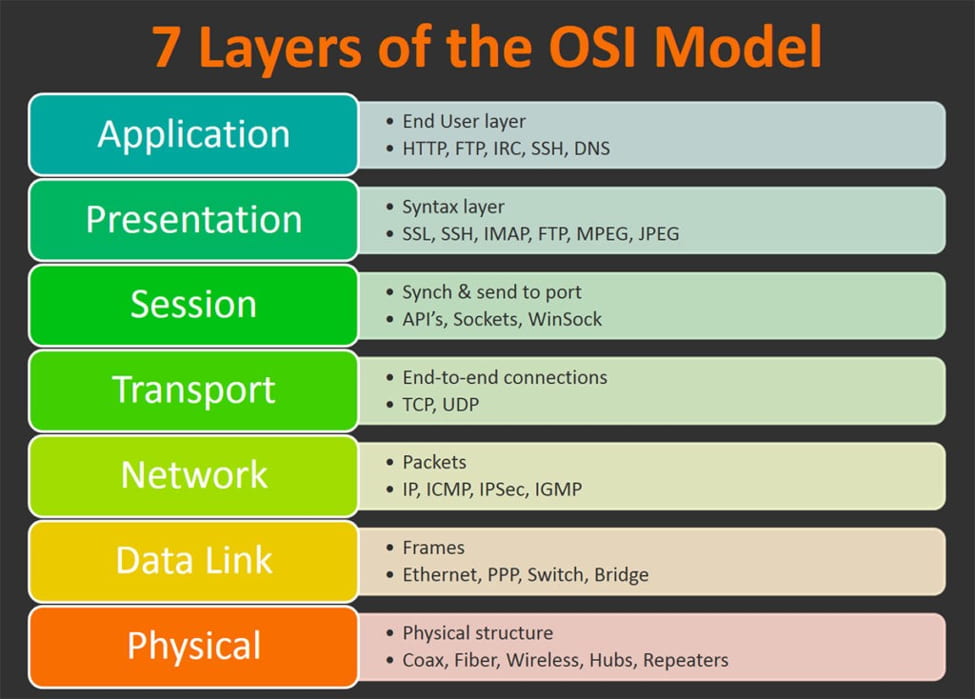
# 4. HTTP

HTTP (HyperText Transfer Protocol) is a widely used client-server internet protocol meant for hypertext, a form of text that can be used for things such as a web page or structuring of information. It features multiple request types that a client can issue the most used types are: GET, POST, DELETE and PUT. GET is obviously used to request resources from the server as DELETE is obvious to delete resources from it. POST and PUT are used to upload resources to the server but PUT is just a little different than POST because it is used to replace/update data. In return of a request the server will send a response with a response code ranging from 100 to 599, the first digit of this code represents the code type. The different code types are: Info (1xx), success (2xx), redirection (3xx), client error (4xx) and server error (5xx) *(MDN, 2023)*.

HTTP uses TCP (Transmission Control Protocol) as its transport layer which will ensure reliability and connection. In the image below the connection between HTTP and TCP is explained by showing what is built on top of what.

Figure 1. OSI model

*The-Physical-Layer-in-OSI-Model-Explained-thumbnail.jpg (JPEG Image, 975 × 699 pixels). (n.d.). Retrieved May 5, 2023, from https://shardeum.org/blog/wp-content/uploads/2022/09/The-Physical-Layer-in-OSI-Model-Explained-thumbnail.jpg*

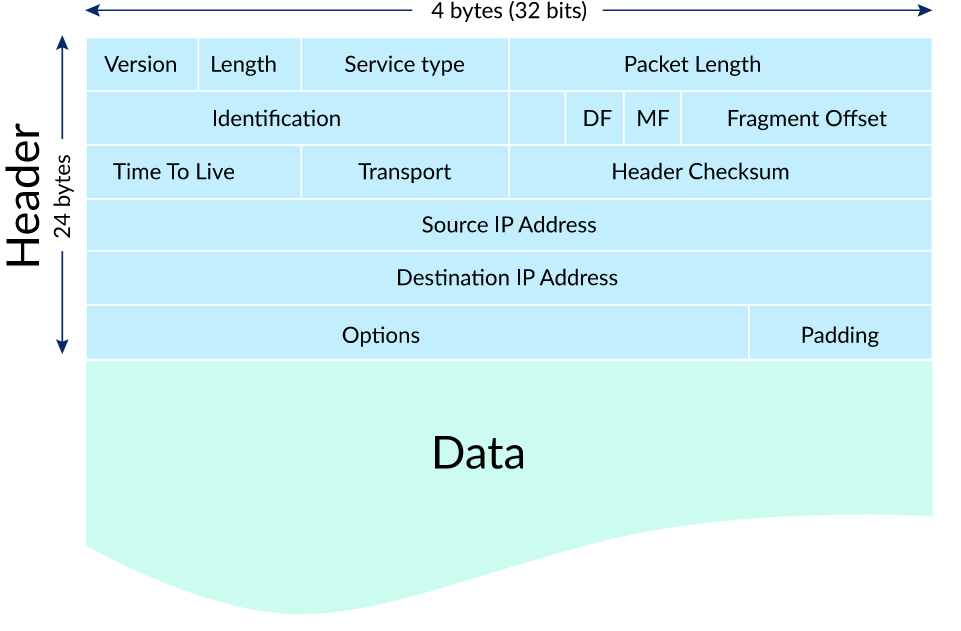


## 4.1. TCP

TCP is a transport protocol that is built on top of the IP network protocol (see Figure 1). The IP protocol works in packets that make sure that the packet is sent to the correct computer and that both parties know where the data came from, if it (the header) was received correctly and how large the data segment is among other things (see Figure 2 for an image of an IP packet).

Figure 2. IP packet

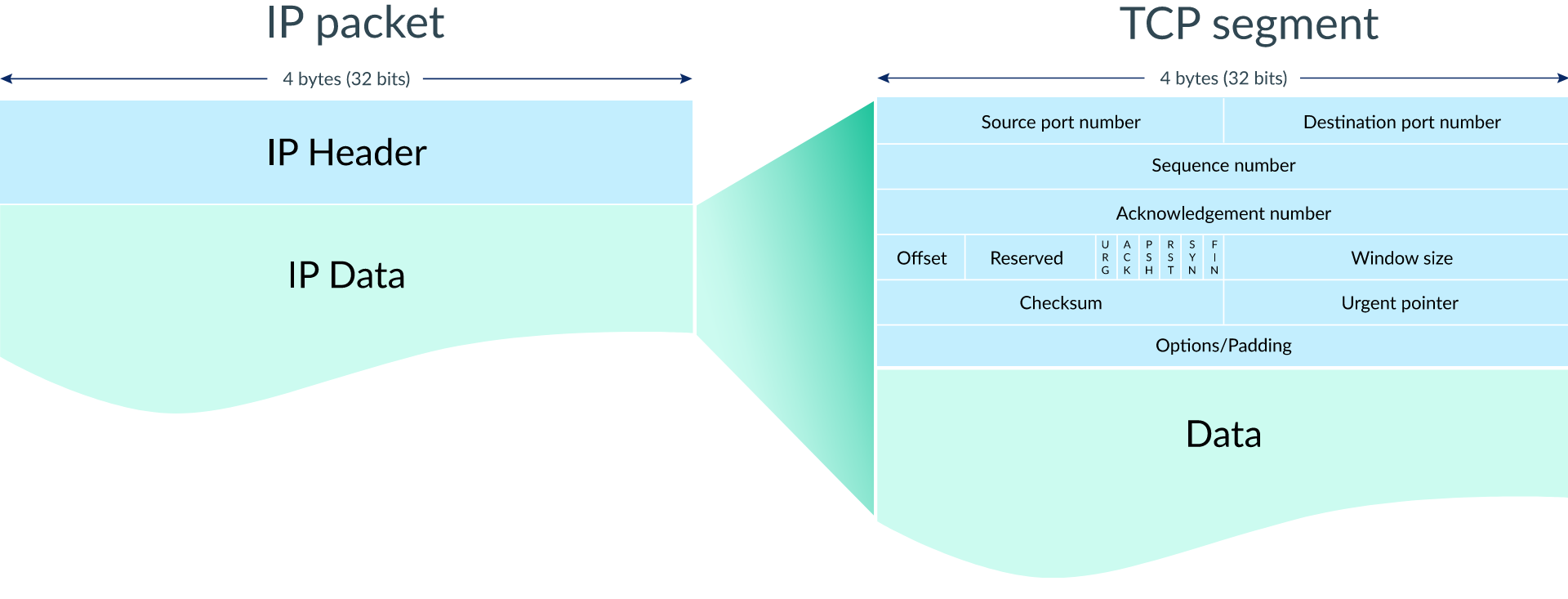
*IP packet*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/337190cba133e19ee9d8b5878453f915971a59cd.svg>



The TCP segment (header) is then placed inside the data section of the IP packet (along with the data) this can be seen in Figure 3. In the TCP segment you can find a checksum which is used to make sure the data is received correctly, there is also some additional data in the form of flags and sequence variables that are used to establish and hold a connection.

Figure 3. TCP packet

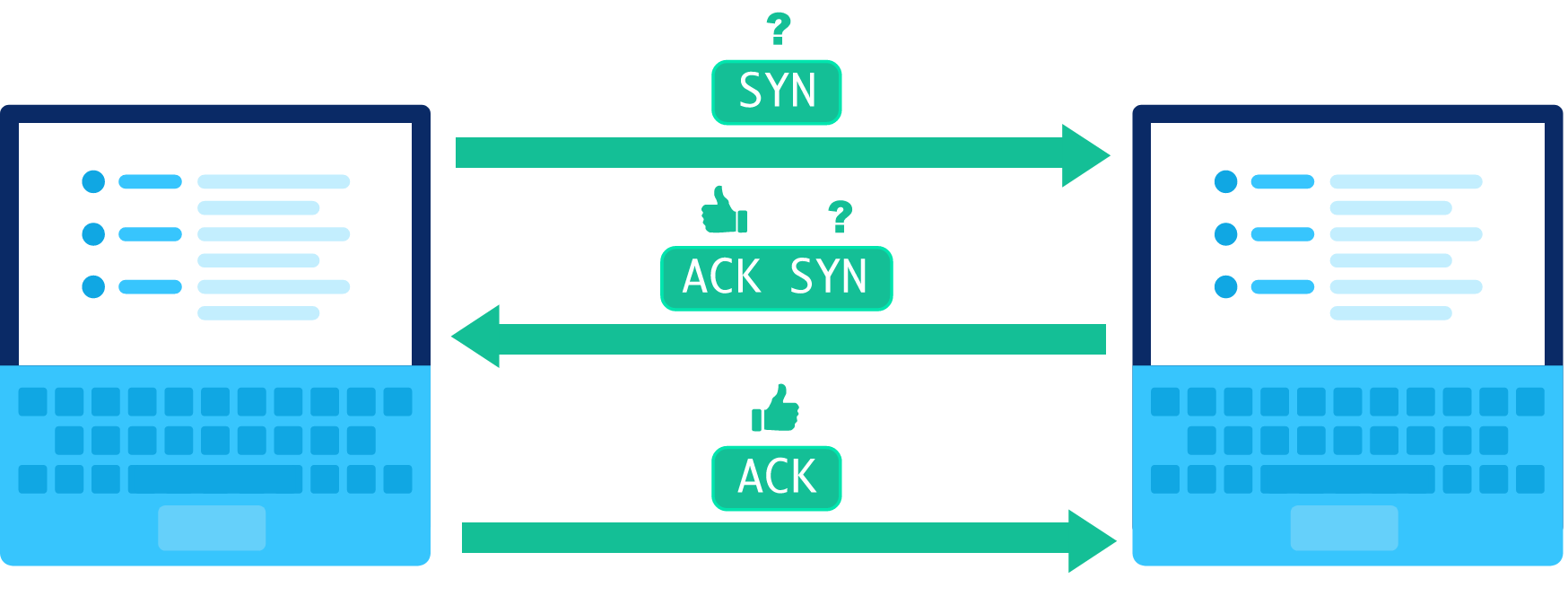
*TCP segment*. (n.d.). Retrieved May 6, 2023, from https://cdn.kastatic.org/ka-perseus-images/e5fdf560fdb40a1c0b3c3ce96f570e5f00fff161.svg



When initiating a connection the process shown in Figure 4 is used this is commonly known as the handshake. In this process the client will a sync request to the server (SYN bit set in header) to which the server will respond with a sync acknowledgment (SYN and ACK bit set in header), When this is received by the client an acknowledgment is sent and the connection is officially established. These handshake packets typically do not include data *(Khan Academy, n.d.).*

Figure 4. TCP handshake

*TCP handshake*. (n.d.). Retrieved May 6, 2023, from https://cdn.kastatic.org/ka-perseus-images/d09f9d37ff2a2deb21a8822f8c99ba6b86319f0b.svg



Now that the handshake is completed the data transfer can begin. As displayed in Figure 5 a sequence number is added to the header, this is set to the amount of bytes that were transferred before it so that the receiver can detect packet loss. Packet loss is detected by keeping track of the received bytes this is done by incrementing the acknowledgment number, now the incoming sequence number can be compared to the last acknowledgment number (these need to be equal). When packet loss is detected the packet is requested again by sending another acknowledgment with the expected sequence number as acknowledgment number (see Figure 6) *(Khan Academy, n.d.)*. Additionally a timeout is added so that when the sender does not receive an acknowledgment in time the packet is sent again (see Figure 7).

Figure 5. TCP data transfer

*TCP send packet*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/2cfc6b88b3b5c3a27386503d347524c2065a57d9.svg>

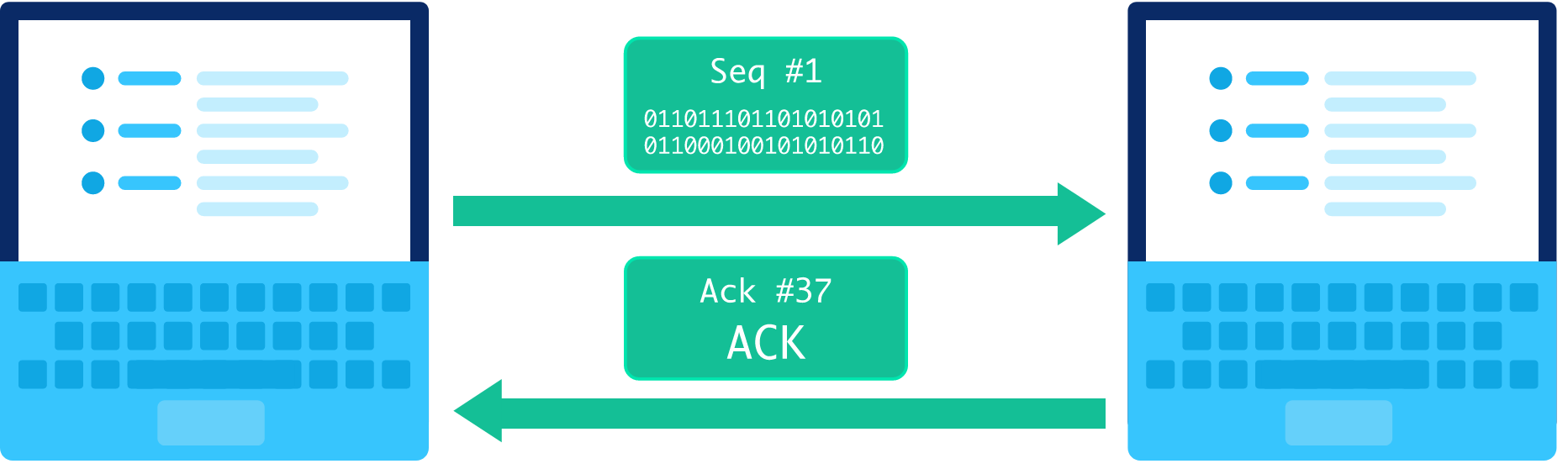


Figure 6. TCP packet loss

*TCP out of order packet*. (n.d.). Retrieved May 9, 2023, from <https://cdn.kastatic.org/ka-perseus-images/27f4fa1915c98689623e0ee224416c5290afc65a.svg>

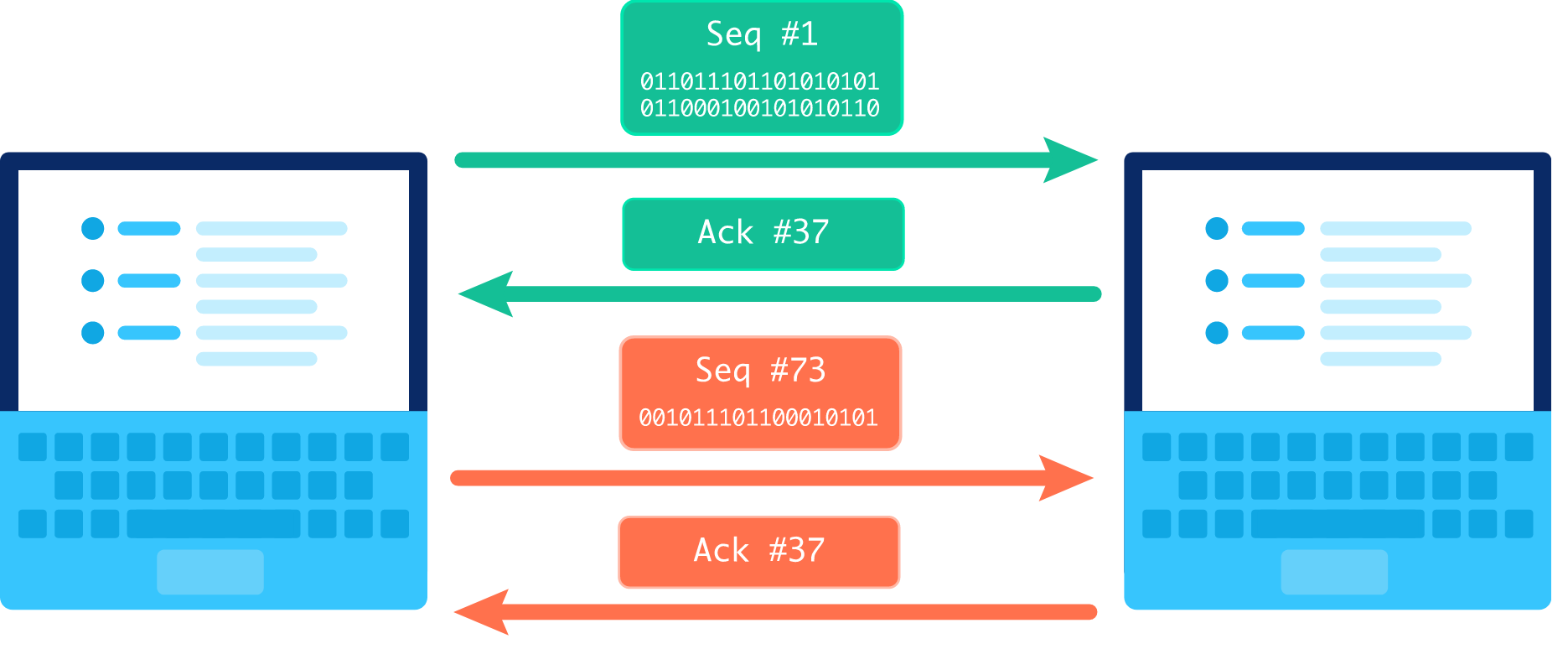
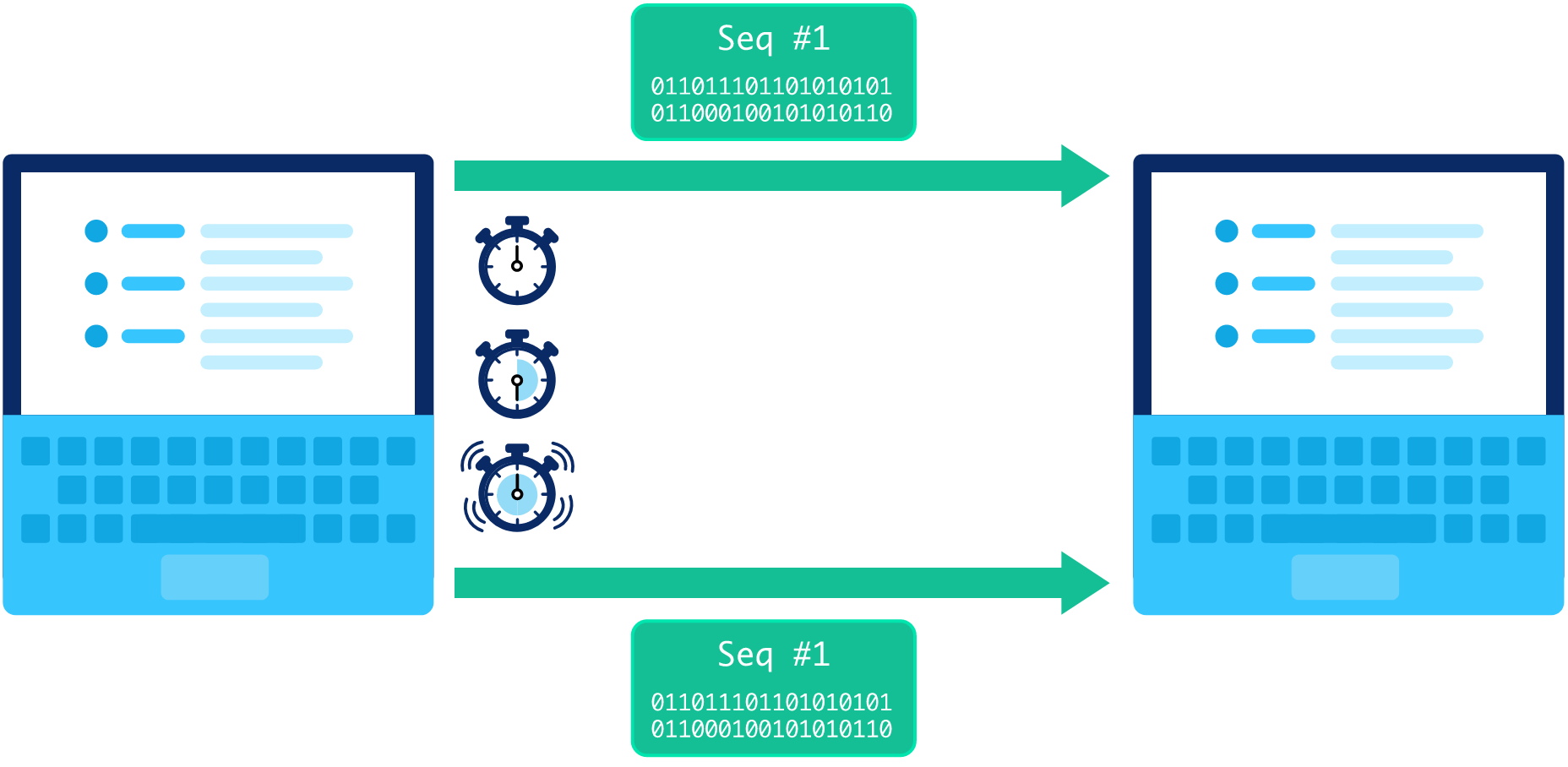


Figure 7. TCP packet timeout



*TCP timeout*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/b1017461d232cd46fa5b445f80e75568bf31c57c.svg>

Finally the connection has to be closed this is the exact same process as the handshake but with the FIN flag set instead of SYN.

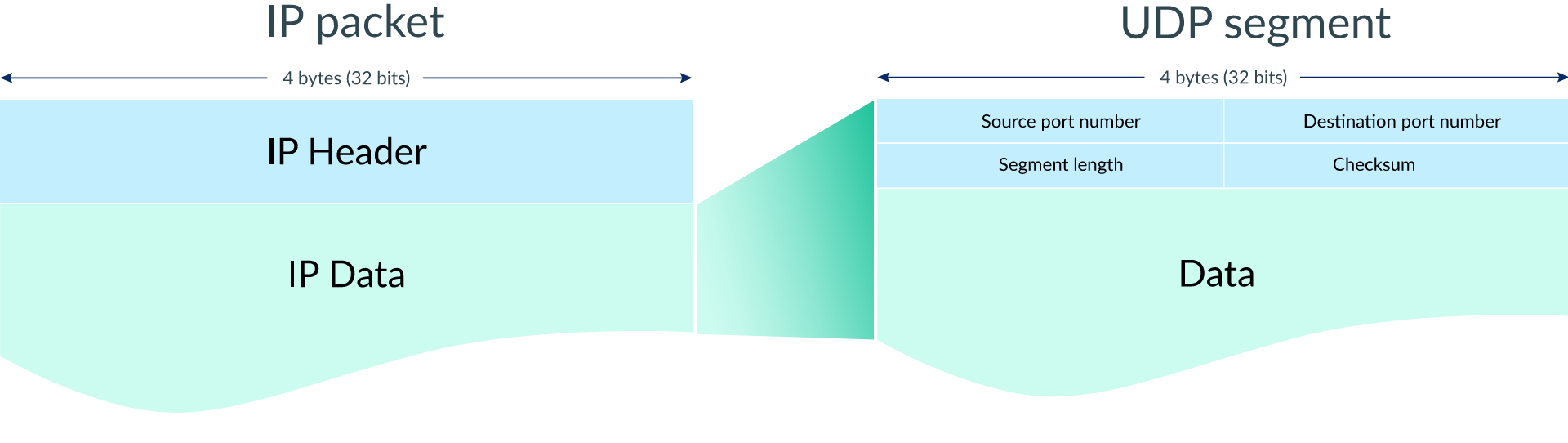
# 5. CoAP

CoAP (Constrained Application Protocol) is a specialized internet protocol for lossy or low power networks that is designed to resemble HTTP. It also has request types and response codes like HTTP with some small differences. The main difference between CoAP and HTTP is that It uses UDP as its transport layer instead of TCP *(CoAP Protocol, n.d.).* UDP has some advantages compared to TCP like the ability to multicast, increased speed and simplicity but these come at the cost of reliability.

## 5.1. UDP

UDP like TCP is a transport protocol that is built on top of the IP protocol (see TCP for an explanation of the IP protocol) it also works with packets but does not require a connection to be established, this makes UDP more suitable for lossy networks which may mess up things like a handshake, even though some data will be lost with UDP in this case. Another feature of connectionless protocols is that they can send to multiple receivers at once, this is called multicasting the only downside is that data transfer is less reliable without a connection. In Figure 8 you can see that the packets are a lot less complex than the TCP packet. This and the fact that there are no acknowledgments makes UDP a lot faster than TCP *(Khan Academy, n.d.).*

Figure 8. UDP packet



*UDP segment*. (n.d.). Retrieved May 10, 2023, from <https://cdn.kastatic.org/ka-perseus-images/9d185d3d44c7ef1e2cd61655e47befb4d383e907.svg>

# 6. Implementation

For the client implementation we have chosen the ESP32 as our microcontroller because it can connect to WiFi and because code for both HTTP and CoAP was already available for it. In addition to the clients server applications were made to test the output of the client and check whether they stay connected or not. Though there was a problem with the CoAP server because the library we used could not send data.

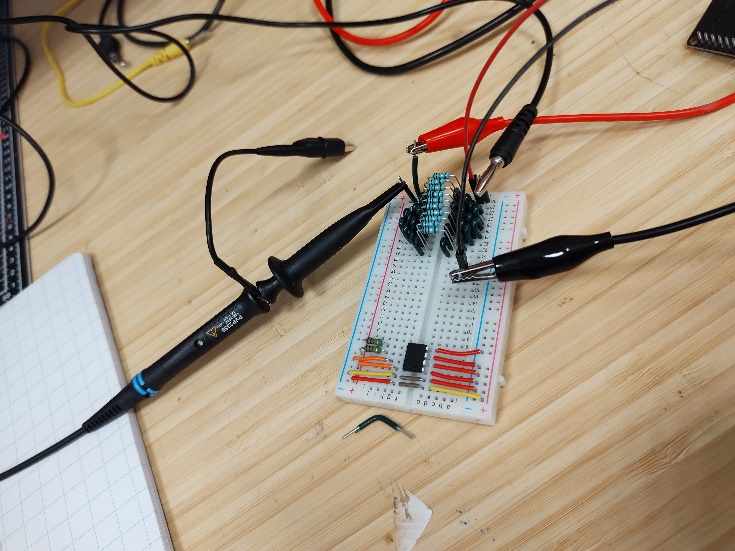
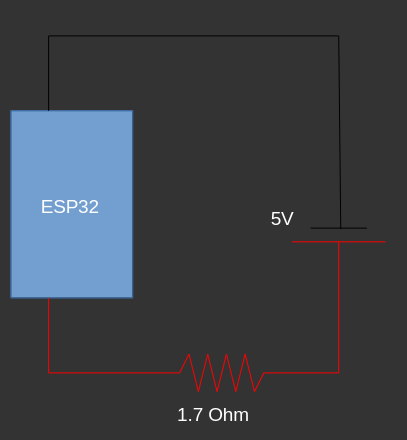
## 6.1. Measurement setup

To measure the power draw of the clients the voltage across an external resistor is measured see Figure 9 for an image of this setup. The external resistor has to be as small as possible to prevent large voltage drops that influence the client, so that is what we tried to create in Figure 10 by connecting 10x1 Ohm resistors in series. These resistors turned out to have significant errors which in combination with the sub optimal setup created a 1.7 Ohm resistor which we ended up using for one measurement for the CoAP client (Figure 22).

This setup is actually not what we used to measure power consumption because we got access to a *Nordic Power Profiler Kit II* which gave us way more accurate data. We made this setup because we originally did not have access to the power profiler kit.

Figure 9. Measurement setup

Figure 10. External resistor



## 6.2. HTTP client

The client code starts off by connecting to WiFi this can be seen in Figure 11. Then the client connects to the server by providing its IP address and port, this can be seen in Figure 12. Now that the client is connected a request can be made, in Figure 13 you can see that a GET request is sent and the returned data is displayed on the serial monitor (Figure 14). The library we used for HTTP was already built into the Arduino environment for the ESP32.

Figure 11. Connect to WiFi HTTP



Figure 12. Connect to HTTP server



Figure 13. HTTP requests

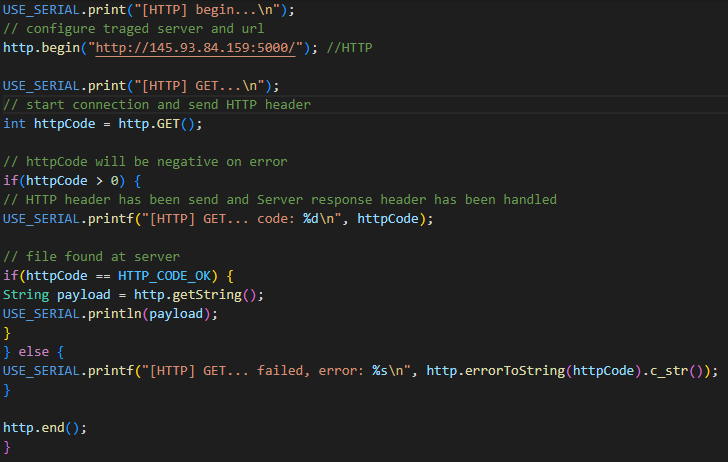
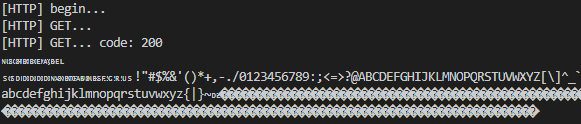


Figure 14. HTTP serial output



When looking at the network activity the expected transaction can be seen in Figure 15 (the steps are explained under HTTP). And in Figure 16 the response from the server can be seen, this matches up perfectly with the serial output of the client (Figure 14).

Figure 15. HTTP network activity snapshot

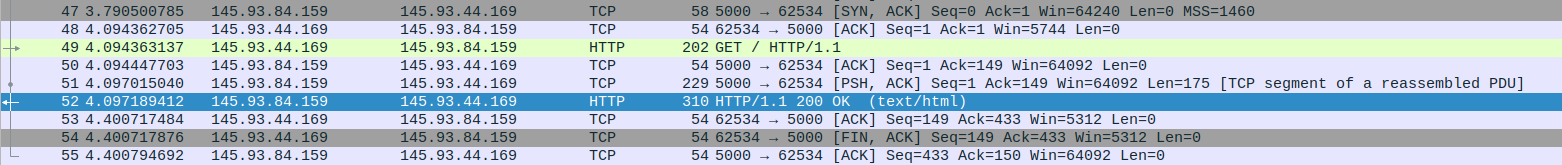
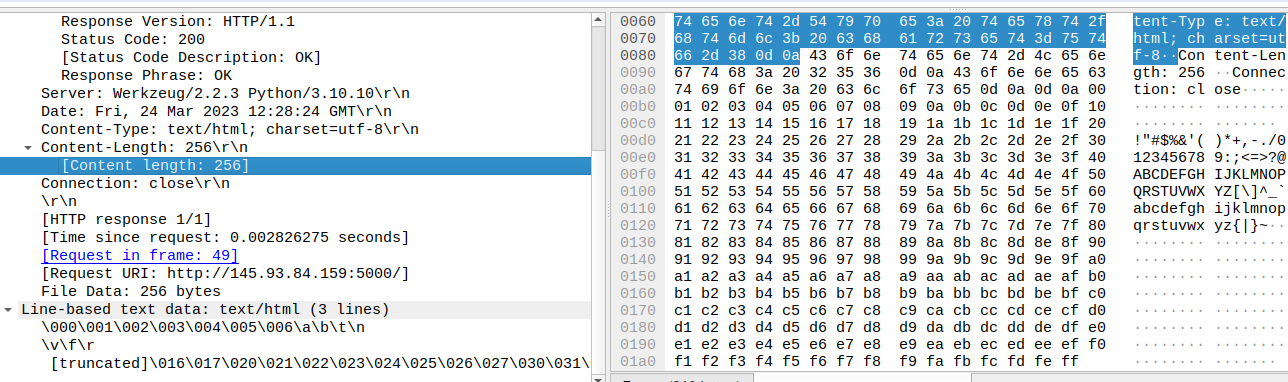
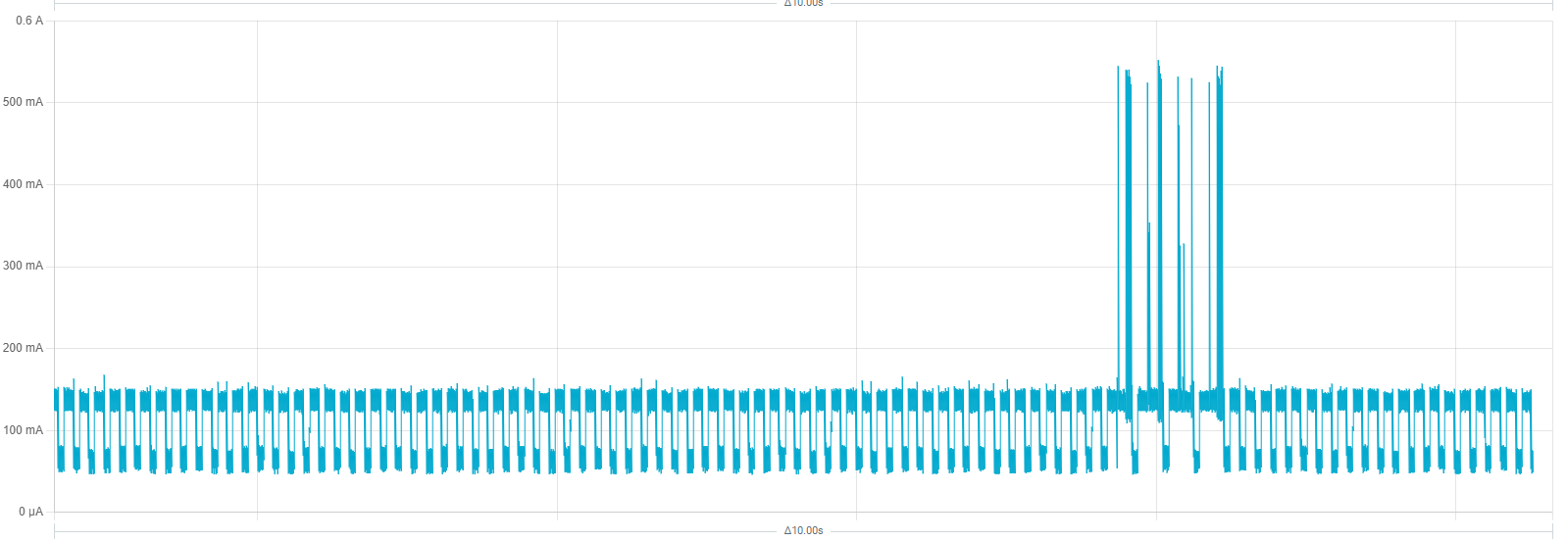


Figure 16. HTTP server response



And now the most important part, the power measurements. You can see in Figure 17 that the ESP32 has an idle power draw of around 100 mA on average, but when a HTTP request is made that power draw quickly rises to 500 mA for a small amount of time. We think looking at the graph it is safe to say that the average power draw during the HTTP communication is arround 200 mA.

Figure 17. HTTP power consumption



## 6.3. CoAP client

Much like the HTTP client the CoAP client starts off by connecting to the wifi (Figure 18). Now that we have an internet connection we can start sending request this is done in Figure 19, a big difference with the HTTP code is that we enter a receive loop after we send a request this is done because the library we used doesn't use the second core like HTTP does. The library we used the *CoAP simple-library by* (hirotakaster, 2023).

Figure 18. Connect to WiFi CoAP

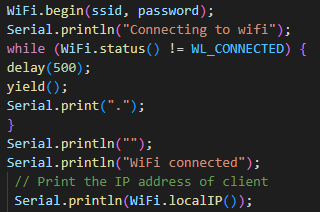
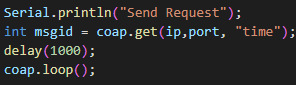


Figure 19. CoAP request



The client sends a request every second (keep in mind that the server does not respond due to issues with the code). The expected network activity can be seen in Figure 20. In Figure 21 the data received by the server can be seen.

Figure 20. CoAP network activity



Figure 21. CoAP request

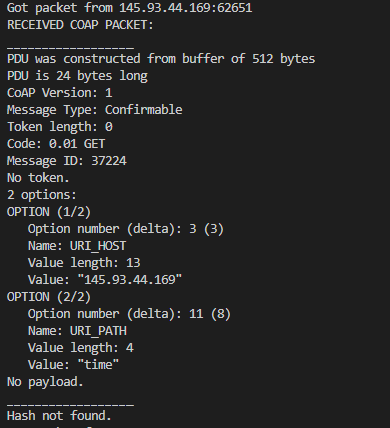
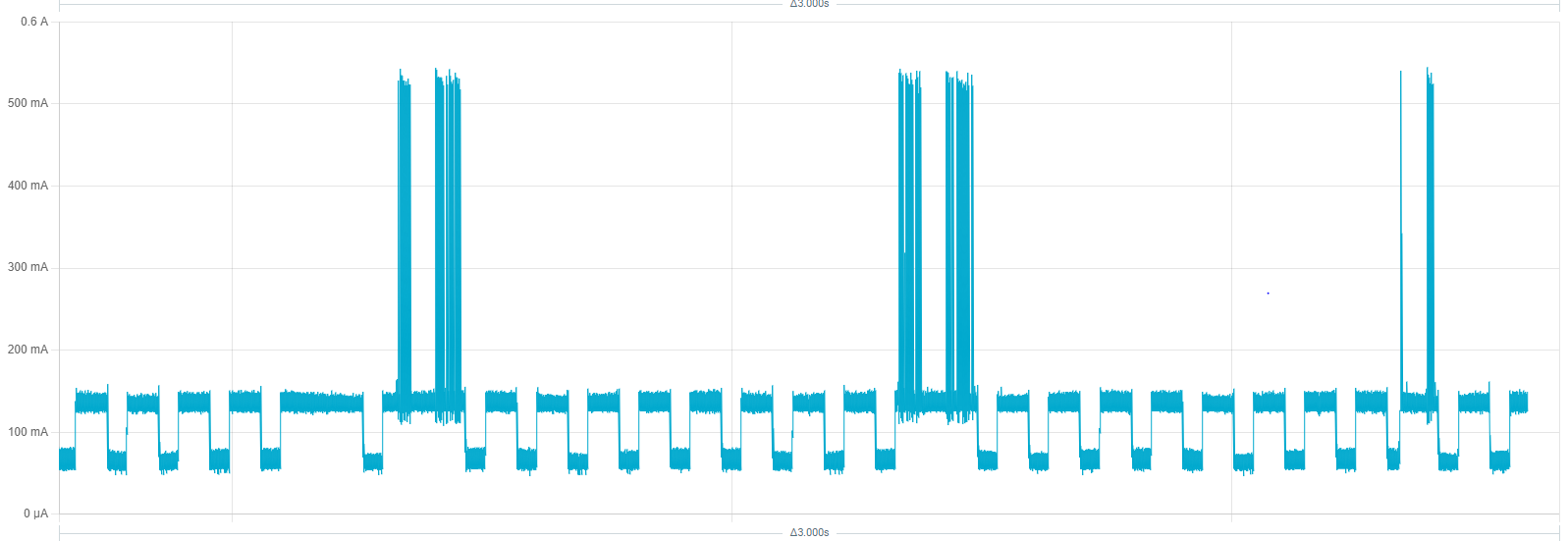


Figure 22. CoAP power measurement using external resistor



Figure 23. CoAP power measurement



# 7. Bibliography

*The-Physical-Layer-in-OSI-Model-Explained-thumbnail.jpg (JPEG Image, 975 × 699 pixels)*. (n.d.). Retrieved May 5, 2023, from <https://shardeum.org/blog/wp-content/uploads/2022/09/The-Physical-Layer-in-OSI-Model-Explained-thumbnail.jpg>

*TCP handshake*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/d09f9d37ff2a2deb21a8822f8c99ba6b86319f0b.svg>

*TCP segment*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/e5fdf560fdb40a1c0b3c3ce96f570e5f00fff161.svg>

*Transmission Control Protocol (TCP) (article) | Khan Academy*. (n.d.). Retrieved May 6, 2023, from <https://www.khanacademy.org/_render>

*TCP header only copy*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/ec71832edb1f2ff1d2ad12da494033ce2b25aafa.svg>

*TCP send packet*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/2cfc6b88b3b5c3a27386503d347524c2065a57d9.svg>

*TCP timeout*. (n.d.). Retrieved May 6, 2023, from <https://cdn.kastatic.org/ka-perseus-images/b1017461d232cd46fa5b445f80e75568bf31c57c.svg>

*HTTP response status codes - HTTP | MDN*. (2023, April 10). <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

*TCP out of order packet*. (n.d.). Retrieved May 9, 2023, from <https://cdn.kastatic.org/ka-perseus-images/27f4fa1915c98689623e0ee224416c5290afc65a.svg>

*UDP segment*. (n.d.). Retrieved May 10, 2023, from <https://cdn.kastatic.org/ka-perseus-images/9d185d3d44c7ef1e2cd61655e47befb4d383e907.svg>

*CoAP Protocol: What is Meaning, Architecture and Function❓*. (n.d.). Retrieved March 3, 2023, from <https://www.wallarm.com/what/coap-protocol-definition>

*User Datagram Protocol (UDP) (article) | Khan Academy*. (n.d.). Retrieved May 10, 2023, from <https://www.khanacademy.org/_render>

hirotakaster. (2023). *CoAP client, server library for Arduino.* https://github.com/hirotakaster/CoAP-simple-library (Original work published 2016)