Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

Project report for course in Internet-of-Things protocols

Adam Temmel

Table of Contents

Abbreviations

IoT	Internet of Things
RTT	Round Trip Time

1 Introduction

As Moore's Law foretold[1], we have been steadily increasing the ratio between computing power and metric area unit used to contain it. Computers are no longer contained to a single room, as they nowadays are able to fit in the palm of our hands. Another side effect of Moore's Law is our newfound ability to design communication protocols not only with respect to the computer, but to instead cater to the needs of developers, who might appreciate working with a protocol which is more comprehensible for humans instead of machines. While this is greatly appreciated for most use cases, there are a few outliers where these protocols are not feasible to use. One such case is the field of Internet-of-Things[2]. These devices are more tightly constrained in terms of resources when compared to your average personal computer, suggesting that they, depending on usage circumstances might require communication protocols designed for computers first and humans second.

1.1 Background and problem motivation

As IoT devices have been present for quite some time now, several different protocols targeting low-performance device communication have been drafted. Two of these are CoAP[3] and MQTT[4]. CoAP uses a REST-like model for device communication, whereas MQTT operates using a publish-subscribe model, meaning that they differ slightly in terms of what is and is not a suitable usage case for the device. Depending on the scenario, it might even be advantageous to join these two protocols when designing a larger system, depending on the specific needs of the individual components. It is not unreasonable to suggest that a system with more moving parts might very well be more complicated to author than a system with less moving parts. As such, it could be of interest to reconstruct such a situation in an attempt to later dissect it and discuss the ease of implementation for such a project, which is what this study tries to accomplish.

1.2 Overall aim

This project aims to reconstruct a rather basic (but scaleable) scenario between several components using various different communication protocols. In total, **five** different components are present within the system, with **three** different protocols being used, depending on the context. These protocols are the two aforementioned CoAP and MQTT, as well as the WebSocket protocol.

1.3 Concrete and verifiable goals / Detailed problem statement

The concrete and verifiable goals present for this project are as follows:

1. A working system consisting of at least 5 different components (including the end client) and 3 different protocols.
2. Partial implementations of the MQTT, CoAP and WebSocket protocol for usage within the project.
3. Working interactions between the author's protocol implementations as well as given library counterparts.
4. A benchmark able to assure the quality of the system.

1.4 Scope

Due to resource and simplicity constraints, the system will only be present on a single machine, which will detract some authenticity from the project, as a more authentic scenario would distribute the different components to different machines, depending on the use case(s) of what they are attempting to mimic.

1.5 Outline

Chapter 1 presents a basic introduction to the project as well as some background information. Chapter 2 describes some underlying theory regarding the different communication protocols used. Chapter 3 describes an overarching vision of the system that the project aims to design. In chapter 4, details regarding the individual components and their purposes are presented, including a brief introduction to the WebSocket protocol. Chapter 5 showcases the resulting frontend that the project produces as well as some round-trip-time measurements. Chapter 6 presents some general discussions regarding the project, including ethical discussions and a few suggestions for future work.

1.6 Contributions

Certain components of different protocols required some external libraries to finalize. These are as follows:

- libcoap[5] - For creating a CoAP server.
- mqtt_cpp[6] - For creating different MQTT clients.

- openssl[7] - For computing the SHA-1 hash.

2 Theory

This chapter will present some underlying theory to understand the rest of the report.

2.1 TCP

The *Transmission Control Protocol* is a reliable protocol for data transmission over the internet[8]. The protocol features both checked and ordered transmission of data, leading it to being a widely used protocol for a variety of applications.

2.2 UDP

The *User Datagram Protocol* (sometimes referred to as the *Unreliable Datagram Protocol*) is a lightweight datagram-based protocol[9]. It is presented as an alternative to the TCP protocol for environments where the amount of available resources are too constrained for any potential usage of the TCP protocol. This is achieved by sacrificing the reliability features of TCP, meaning that UDP is less reliable as a consequence.

2.3 MQTT

The *MQTT* protocol is described as a lightweight publish/subscribe protocol designed with IoT devices in mind[4]. It is designed with a client-broker architecture in mind, meaning that clients with different agendas in mind all connect to a single broker which is responsible for distributing different messages. This distribution is handled by allowing clients to subscribe and publish messages to different topics, thusly achieving the aforementioned publish/subscribe functionality.

2.4 CoAP

The *Constrained Application Protocol* is described as a variant of HTTP suited for IoT devices[3]. All of the regular REST parameters are present within the protocol, with different resources being marked using URL paths to distinguish them. It can also operate using UDP as the underlying protocol, meaning that CoAP is well suited for IoT devices.

2.5 WebSockets

The *WebSocket* protocol was authored as alternative to HTPP for bidirectional communication between a server and client[10]. This includes instant messaging applications, games or other services which ideally operate without the need of reloading a webpage. It, like HTTP, uses TCP to provide a stable ground for the protocol.

3 Model

This chapter will discuss the model used to attain the goals of the project.

3.1 System overview

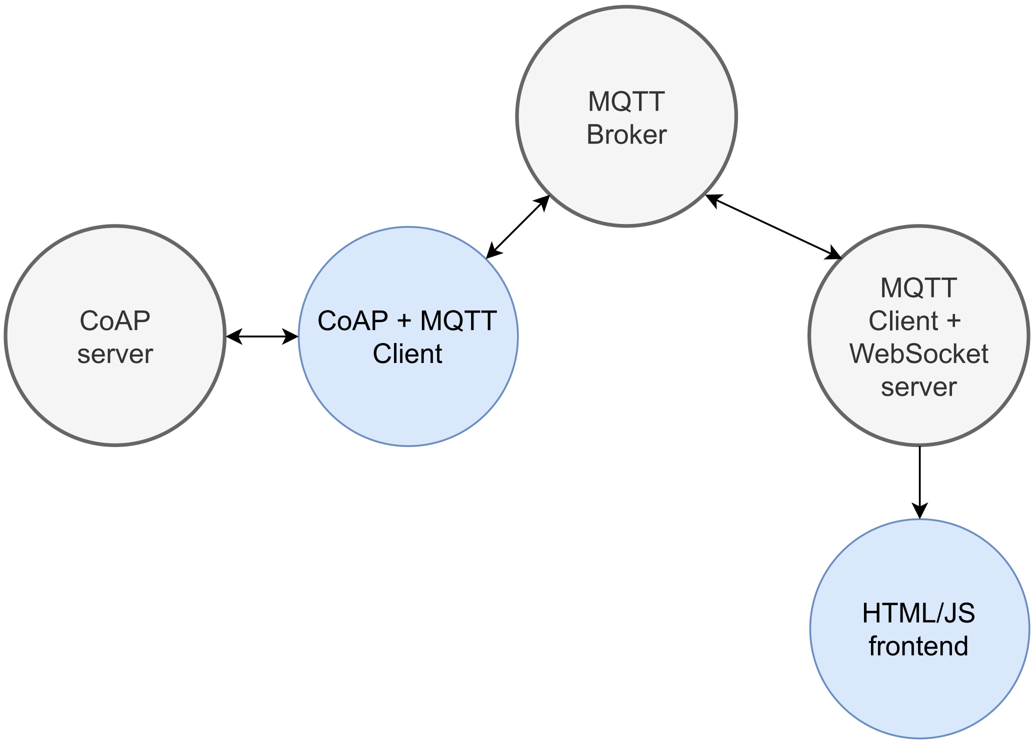

Figure 1: Overview of the system

Figure 1 shows a brief overview of the different components in the system. They are as follows:

3.1.1 CoAP server

A CoAP server responsible for providing current CPU and memory usage. This component should ideally symbolise some sort of IoT device which is of interest to monitor.

3.1.2 CoAP + MQTT Client

This component is responsible for picking up on MQTT based requests for current CPU/memory usage, then translating these requests into the

CoAP protocol and sending them to the CoAP server. Upon receiving a response from the CoAP server, this response is then translated back into a message able to be published to the MQTT broker.

3.1.3 MQTT Broker

The MQTT broker is responsible for managing all publications and subscriptions authored by the two MQTT clients. If either of the two clients wish to subscribe to a topic, this subscription will be registered and remembered by the broker. The next time a client wishes to publish something to this topic, this publication is retransmitted to all clients currently subscribed to this topic.

3.1.4 MQTT Client + WebSocket Server

This component works as a bridge between the MQTT broker and the end user frontend. It is meant to regularly publish requests for current CPU/memory utilization and subscribe to the topic in which the responses to these requests are published. Upon receiving a response, it is then translated into a WebSocket message which is sent to the frontend. This component should also be able to benchmark the RTT between publishing a request and receiving a response, to later enrich the message sent to the frontend with the measured time.

3.1.5 HTML/JS frontend

The frontend is a comparatively rather simple component, all things considered. It listens for published messages from the WebSocket server and updates three different graphs which plot CPU usage, memory usage and round trip time as a function of time. The frontend is not capable of sending messages in and of itself, but instead relies on the WebSocket server being able to pump out new information.

4 Design / Implementation

This chapter aims to discuss the implementation details of the system presented in *figure 1*.

4.1 CoAP server

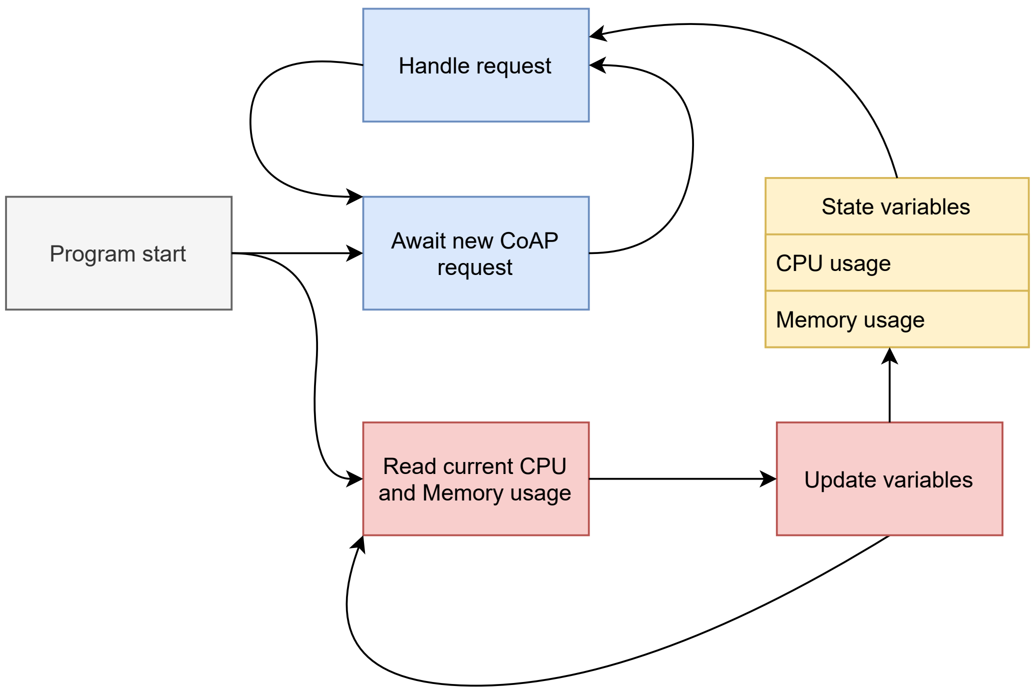

Figure 2: Flowchart depicting the duties and the program flow of the CoAP server.

Figure 2 shows the program flow of the CoAP server. The flow is separated into two threads marked as blue and red boxes in figure 2. There is also a yellow box, representing data shared between the threads. This data is locked behind mutex locks so as to not cause any race conditions between the red and blue thread.

The red thread is responsible for regularly reading the current CPU and memory usage from the system. It then tries to access the shared state variables in order to update them with the new information. Upon successfully updating the variables, the lock is released and the thread sleeps for one second before attempting to read the CPU and memory usage once more.

The blue thread is responsible for performing all server-related duties of the system component. It listens for potential requests to the urls `/cpu` and `/mem`. Upon recieving a request, it tries to access the corresponding shared state variable, embeds it into a response, and sends the response. It then goes back into listening mode, awaiting the next request.

4.2 CoAP + MQTT Client

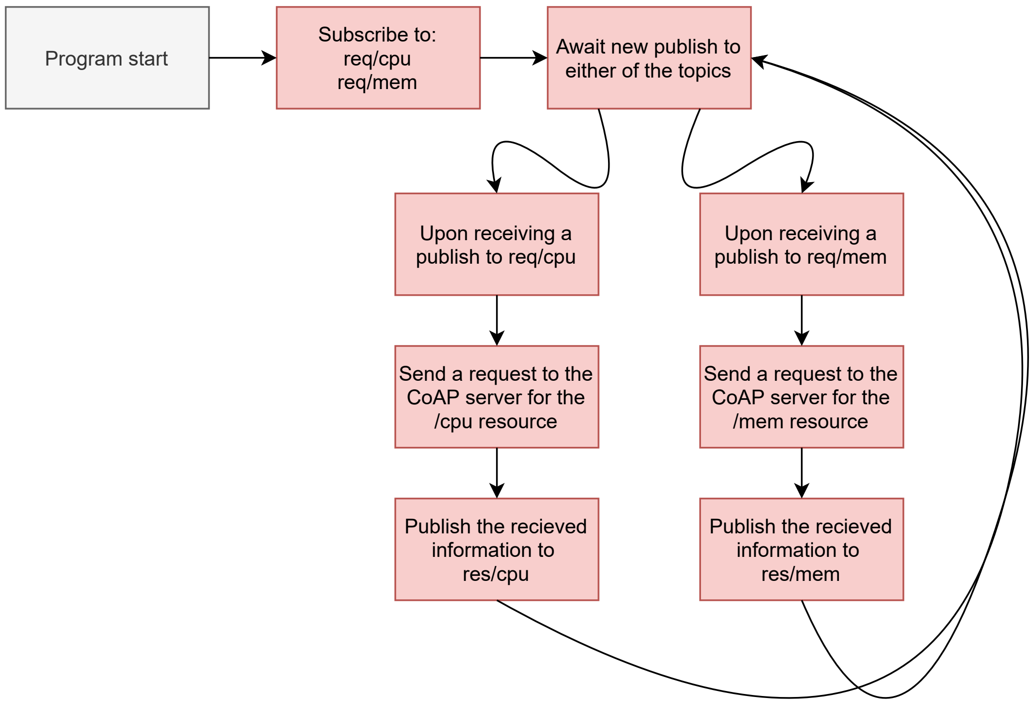
Program start
Subscribe to: req/cpu req/mem
Await new publish to either of the topics
Upon receiving a publish to req/cpu
Upon receiving a publish to req/mem
Send a request to the CoAP server for the /cpu resource
Send a request to the CoAP server for the /mem resource
Publish the recieved information to res/cpu
Publish the recieved information to res/mem

Figure 3: Flowchart depicting the program flow of the CoAP + MQTT Client component

The CoAP + MQTT Client, pictured in *figure 3* can be viewed as a sort of bridge between the two protocols. Its only purpose is to intercept publications to the `res/cpu` and `req/cpu` MQTT topics. Upon receiving a message from either of those two topics, a request to the CoAP server for either the `/cpu` or `/mem` resource is sent. Once the response to this message is received, the contents thereof is published to either the `res/cpu` or the `res/mem` topic. Once a publication has been properly handled, the bridge goes back into awaiting a new publication.

4.3 MQTT Broker

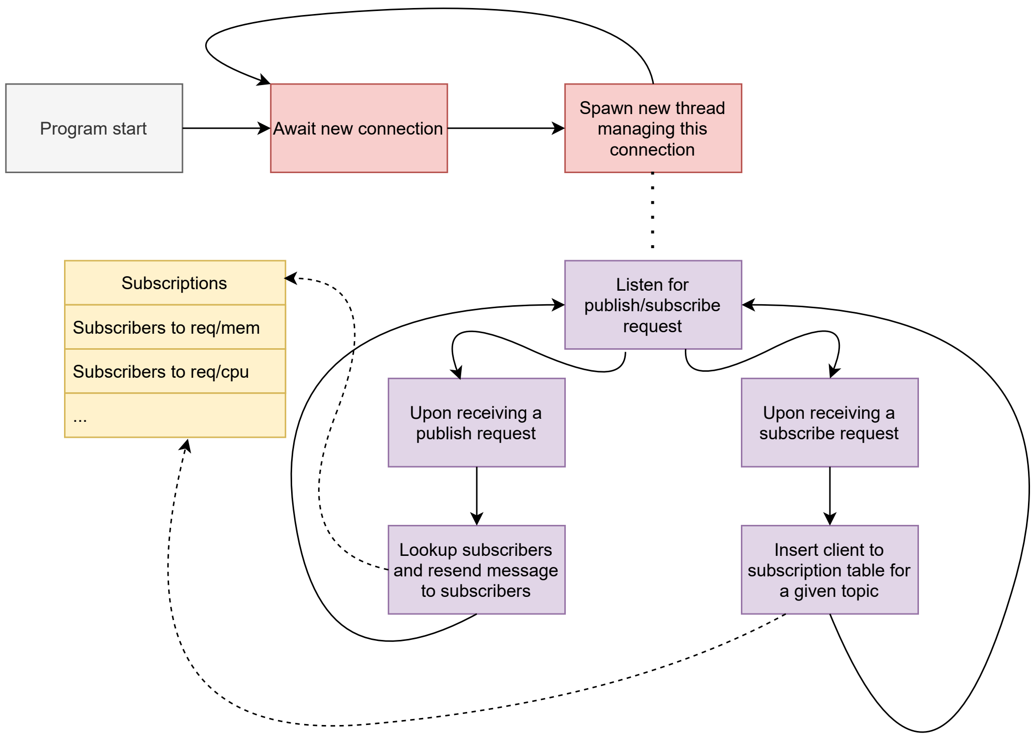
Program start
Await new connection
Spawn new thread managing this connection
Subscriptions
Subscribers to req/mem
Subscribers to req/cpu
...
Listen for publish/subscribe request
Upon receiving a publish request
Upon receiving a subscribe request
Lookup subscribers and resend message to subscribers
Insert client to subscription table for a given topic

Figure 4: Flowchart depicting the program flow of the MQTT broker

The MQTT broker, pictured in *figure 4*, is responsible for managing all MQTT topic subscriptions and publications within the system. On startup, it consists solely of one main thread which listens to new connection requests. Once a new connection request is received, a new thread is spawned which maintains this individual client's publish/subscribe messages. All subscriptions are kept track of in a table which maps topic names to a set of clients, meaning that when a new subscription is requested, the client authoring this subscription is inserted into the corresponding set. If a client sends a publish message, a map lookup is performed to find the appropriate set of subscribers, who will then receive a duplicate of the original message.

4.4 MQTT Client + WebSocket server

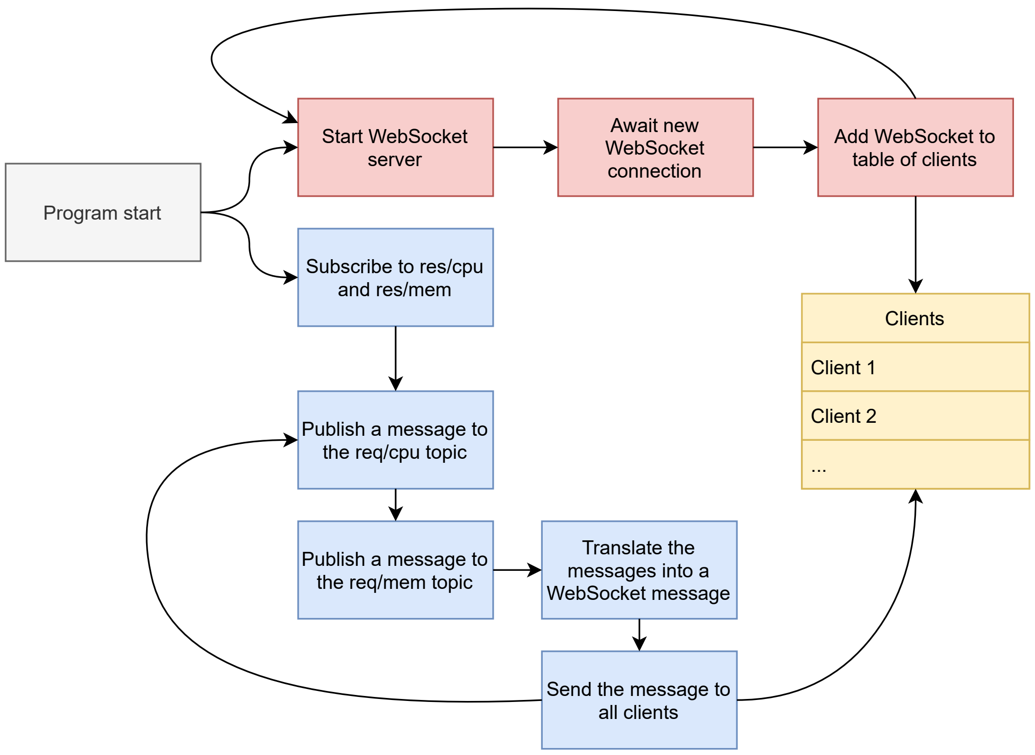
Start WebSocket server
Await new WebSocket connection
Add WebSocket to table of clients
Program start
Subscribe to res/cpu and res/mem
Clients
Client 1
Client 2
...
Publish a message to the req/cpu topic
Publish a message to the req/mem topic
Translate the messages into a WebSocket message
Send the message to all clients

Figure 5: Flowchart depicting the program flow of the MQTT client + WebSocket server

The component pictured in *figure 5* is responsible for managing the connection between the MQTT broker and the frontend. It achieves this by managing a WebSocket connection to the frontend as well as creating a MQTT client which subscribes to the same broker as the rest of the system. The WebSocket subset of this system is marked in red in *figure 5*, whereas the MQTT subset of the system is marked in blue. Both subsystems are run in separate threads, but the list of active clients is indirectly shared between the two subsystems. The WebSocket server thread only manages a set of active clients, appending them to a list as they connect. The MQTT client thread subscribes to the `res/cpu` and `res/mem` topics, before publishing an empty message to the `req/cpu` and `req/mem` topics (sequentially). Before the requests are sent, a timestamp is made. Another timestamp is made when the response is received. Upon receiving both responses, the difference between the timestamps is averaged before being placed into a JSON string which also contains the measured CPU usage and the measured memory usage; This string is embedded into a WebSocket message, which is distributed to all active clients.

4.4.1 The WebSocket protocol

In order to implement the above component, it was of interest to look into the WebSocket protocol specification and implement pieces of it. These pieces include:

- Parsing the client handshake request
- Creating a valid handshake response
- Encoding/decoding of data frames

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

Figure 6: An example of the client handshake request performed when a WebSocket client wishes to connect to a server[11].

Parsing the client handshake request (see *figure 6*) is a rather trivial problem. It closely resembles the HTTP header format in how several key-value pairs are presented as one big string, making the headers easy to read for humans. The important parts to extract is both the intent of using the WebSocket protocol, but also the `Sec-WebSocket-Accept`-header. The value associated with this header needs to be extracted in order to formulate a valid response header. After extracting the key, the correct accept key is generated by[11]:

1. Concatenate the key with the magic string:
   `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`.
2. Calculate the SHA-1 hash of this new string.
3. base64 encode the hash into a new string.
4. The encoded hash is the accept key.

After the accept key has been created, the handshake response can be created. An example of such a response is shown in *figure 7*.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Figure 7: An example of the server handshake response performed when a WebSocket server wishes to accept a client[11].

With the WebSocket handshake underway, all that is left is the ability to understand individual WebSocket messages, usually referred to as ”frames”[11].

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figure 8: A slimmed down explanation of how a WebSocket frame is constructed, courtesy of Mozilla Developer Network[11].

All of the information presented in *figure 8* is not necessary to digest for the project, as the WebSocket server only needs to be able to distribute messages, not necessarily receive them. The `opcode` field specifies what the type of the message is. It could be ”binary”, ”text” or a more specific message, such as ”close”[11]. The `Payload len` fields are partially optional, depending on the message length[11]. There is also a `MASK` bit present in the second byte, which specifies whether the payload is masked using a masking key[11]. The masking key is the 4 bytes between the end of the payload length field and the actual payload data. The mask is applied by applying the XOR operator on each key-byte and payload-byte pair, cyclically[11]. The client is required to set the mask bit, but the server does not need to mask its response(s), according to the specification[11].

4.5 HTML/JS frontend

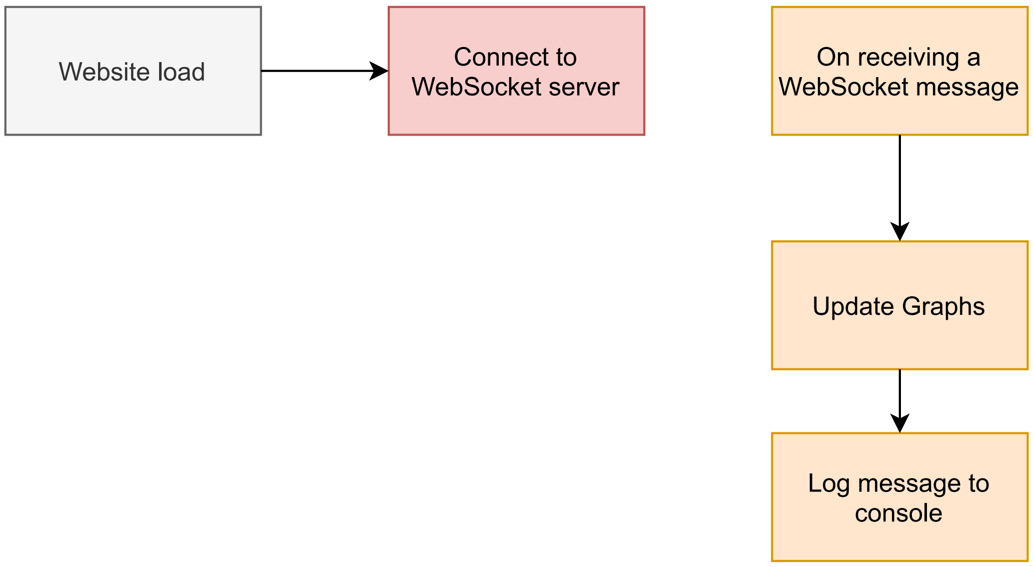

Figure 9: Flowchart depicting the "flow" of the frontend website

As pictured in *figure 9*, the website logic is not particularly interesting. On page load, it tries to establish a connection to the WebSocket server. From there on, it listens to any potential messages and once upon receiving a suitable message, updates the three graphs present on the website. The graphs are meant to present the information of interest, CPU usage, memory usage, and round trip time.

5 Results

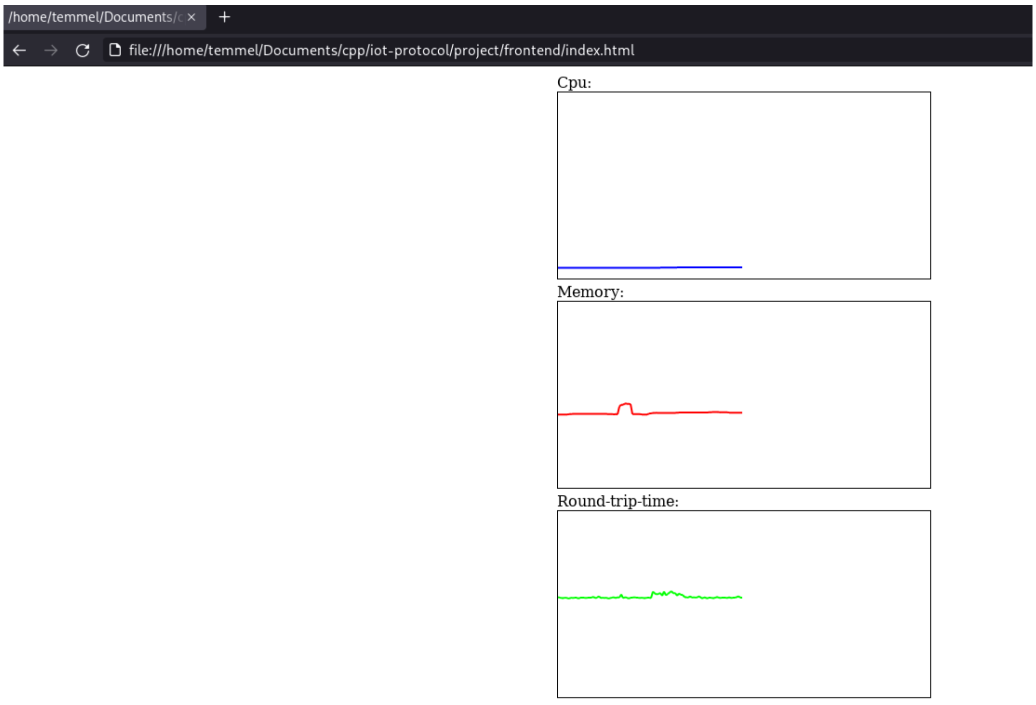
/home/temmel/Documents/c
file:///home/temmel/Documents/cpp/iot-protocol/project/frontend/index.html
Cpu:
Memory:
Round-trip-time:

Figure 10: Screenshot showcasing the appearance of the frontend.

Figure 10 showcases a screenshot of the resulting frontend. Three graphs are present in this screenshot, one for the CPU usage, one for the memory usage and one for the round-trip-times collected when gathering this data. The CPU and memory graphs are normalized from a scale of 0 to 100, meaning that the roof of the graph indicates 100% usage, whereas the bottom of the graph indicates 0% usage. The round-trip-time was treated a bit differently, with the middle of the graph being 0 milliseconds. The graph line is placed one pixel upwards for each millisecond measured as round-trip-time.

As these measurements are rather difficult to study in detail, the website also logs each individual message to the console. In doing so, one can later reconstruct a mean value for the round-trip-time. The mean value during this run was measured to be 7.572 milliseconds, with a standard deviation of 1.67248126 milliseconds. These values were extracted from debug builds of all components, minus the frontend.

6 Discussion

This project indeed shows that creating a system consisting of several components using different protocols is completely doable. In doing so, the first goal of the project is fulfilled. Likewise, the CoAP client, the MQTT broker and the WebSocket server were all implemented for use within the project, whereas the CoAP server and the MQTT client code were pulled in as external dependencies, meaning that the second goal was also fulfilled. All of these connections could operate with each other as planned, resulting in the fulfillment of the third goal. The fourth goal was also achieved, as could be seen in *chapter 5*. The measurements show that the round-trip-time is rather large for a system that is run on a single computer, but one might argue that this could be due to the components being compiled in debug mode. Had more compiler optimizations been turned on, the graph might have ended up looking a little bit differently.

6.1 Ethical and Societal Discussion

Something to consider with these protocols is that IoT devices will most likely stay active for several years. All of these years of activity could be summed up into a great amount of energy spent, depending on the situation. It could therefore be of interest to vary the protocol(s) used in order to optimize the energy usage of the system, which in turn could make the entire system "greener". This is of course largely situational depending on the project one wishes to solve, but the proof alone that one can weave different protocols together and still end up with a comprehensible system shows that this is something worth to consider when designing an IoT-centered system.

6.2 Future Work

A clear prospect of a future work for this project would be to implement true bi-directional communication for WebSockets. As such, the frontend would be able to request the WebSocket server for the current CPU and memory usage stats, thusly being able to measure the round-trip-time without relying on the WebSocket server to do so. A consequence of this is that the ″true″ round trip time would be measured, instead of just measuring the internal round trip time between the CoAP server and the WebSocket server. As the WebSocket protocol has a somewhat involved decoding step for extracting the payload of the message, it could very well be so that this protocol introduces the largest overhead out of the three protocols used within the project.

References

[1] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics Magazine* (Apr. 19, 1965).

[2] Madakam S., R. Ramasway, and Tripathi S. "Internet of Things (IoT): A Literature Review". In: *Journal of Computer and Communications* (2015).

[3] *CoAP Homepage*. URL: `http://coap.technology/` (visited on 01/06/2022).

[4] *Mqtt Homepage*. URL: `https://mqtt.org/` (visited on 01/06/2022).

[5] *libcoap homepage*. URL: `https://libcoap.net/` (visited on 01/08/2022).

[6] redboltz. *mqtt_cpp repository*. URL: `https://github.com/redboltz/mqtt_cpp` (visited on 01/08/2022).

[7] *OpenSSL homepage*. URL: `https://www.openssl.org/` (visited on 01/08/2022).

[8] Vinton G. Cerf and Robert E. Kahn. "A Protocol for Packet Network Intercommunication". In: *IEEE Transactions on Communications* (May 1974).

[9] *User Datagram Protocol*. URL: `https://www.ipv6.com/general/udp-user-datagram-protocol/` (visited on 01/06/2022).

[10] *The WebSocket Protocol draft*. URL: `https://datatracker.ietf.org/doc/html/rfc6455` (visited on 01/06/2022).

[11] MDN Web Docs. *Writing WebSocket servers*. URL: `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers` (visited on 01/07/2022).