IOT, CoAP-MQTT Project

Filip Wessman

MITTUNIVERSITETET
Department of Information Systems and Technology

Main field of study: Implementation av IOT-protokoll
Autor: Filip Wessman, fiwe1601@student.miun.se
Credits: 6hp
Semester/Year: HT 2021
Examiner: Stefan Forsström, stefan.forsstrom@miun.se
Course code: DT065G
Degree programme: Degree of Master of Science in Engineering: Computer Engineering

Table of Contents

1 Introduction

We are today becoming increasingly more and more connected to the internet and to everything and everyone around. A big part of this thanks to smartphones and advances in internet connectivity around the world.

By 2025, there's estimated to be around 60 billion connected devices around the world. A greater part of these are sensors and actuators, Internet of Things (IoT) devices, that is scattered everywhere which interact and communicated with each other.

1.1 Background and problem motivation

The main purpose of IoT is to facilitate and help us in our everyday life by attaching various of sensor and actuators to the things in our surroundings.

Since the sheer number connected devices and the data they generate are only expected to increase it will get harder from them to coexist on the existing network. Hence putting more load on the network. By this using lightweight transmission protocols, for instance CoAP and MQTT, could benefit and facilitate this process. Including having a platform which acts as a form of gateway for handling CoAP and MQTT messages.

1.2 Overall aim

The overall aim for this project is to design and implement a combined IoT system which handles the CoAP and MQTT protocols. A sort of working proof of concept were CoAP sensor generated values are transferred across a chain to a MQTT human friendly front-end interface application.

1.3 Concrete and verifiable goals

The goal with this work is to implement a proof-of-concept application which demonstrates this chain from emulated CoAP generated sensor values to a MQTT frontend interface system. With this creating a MQTT Broker, CoAP Client, and frontend interface from scratch. Also evaluating the implemented system by performing quantitatively measurements of various of message transmission times.

1.4 Scope

The work will be limited to on CoAP server program, which is acting as sensor. One program acting as the gateway which is running a CoAP client and an MQTT client. One program running the MQTT broker and a program running an MQTT client with a form of human friendly front-end interface. Ready-made libraries will be used for the CoAP Server and the MQTT clients, while the MQTT Broker, CoAP Client and the interface will be implemented from scratch. Everything will be implemented och run locally.

1.5 Outline

Chapter 2 presents all relevant and necessary background information. Chapter 3 describes the project's work process. Chapter 4 then presents the implementation, how the method was implemented. Chapter 5 presents all the project's results. Lastly chapter 6 discusses the results, summarizes and concludes the project, and provides suggestions for possible future improvements.

2 Teori

This chapter does in short go through and describes all necessary background information needed to further understand the rest of this work.

2.1 IoT

The concept of Internet of Things (IoT) does in short describe physical Objects which is connected to the internet. Devices that exchange data with other devices, interconnection, and systems over communications networks. These devices are often embedded with various of different sensors and actuators. A giant network of devices with the estimation of over 55 billion IoT devices connected in the year 2025. [1]

2.2 CoAP

Constrained Application Protocol (CoAP) is an application protocol used by constrained, IoT, devices. A lightweight and simple binary-based protocol which makes use of two message types for interaction and communication, requests, and responses. It's a REST-based protocol largely inspired by HTTP that uses the User Datagram Protocol (UDP) communication protocol. This is defined in RFC 7252, June 2014. [2]

2.3 MQTT

Message Queue Telemetry Transport (MQTT) is a publish-subscribe network messaging protocol that transports messages between devices. It's designed to be lightweight and bandwidth efficient where it uses the Transmission Control Protocol (TCP) communication protocol. Its main concept is highly centralized around a coordinating broker server. Devices subscribe to a certain topic and receive data about that topic when the server receives the corresponding publish message, at which it unicasts the data to the subscribers. [3]

2.4 Java

Java is a popular and widely used, robust, high-level, and object-oriented programming language. Developed by Sun Microsystems in the year 1995 as a general-purpose software platform which can be used to develop various of systems. Designed for its portability where it can run on all platforms which support Java, called *write once, run anywhere (WORA).* As writing of this report Java SE version 17 is the newest stable release. [4]

2.5 JavaFX

"JavaFX is an open source, next generation client application platform for desktop, mobile and embedded systems built on Java."[5] This platform is used to design, develop, and deploy rich client applications, Graphical User Interface (GUI). It is written in JavaFX Script and supports CSS styling. It features a language known as FXML, an XML-based declarative markup language, essentially like HTML. [5]

2.5.1 Scene Builder

Scene builder is a visual drag and drop design interface and tool which facilitates and lets users quickly design JavaFX GUI applications. It does this by generating the FXML markup code which then can be implmented in an IDE. [6]

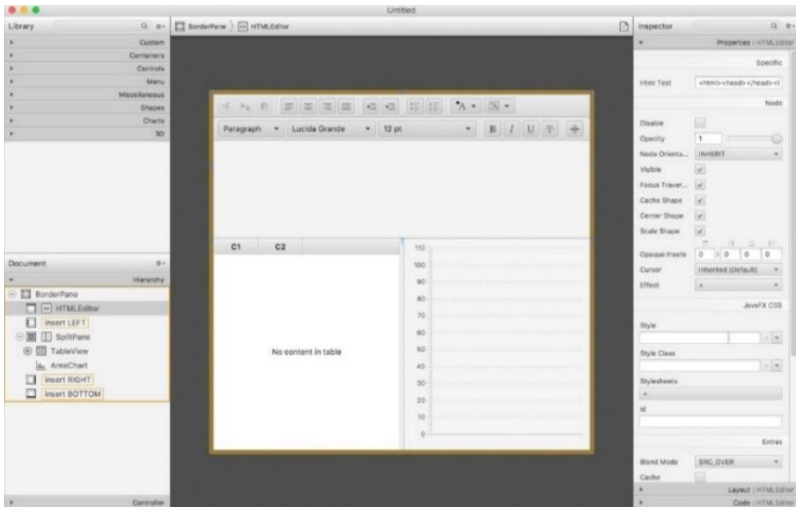


Figure 1: Scene Builder Interface [6]

2.6 MjCoAP

MjCoAP is a cross-platform, open-source, and lightweight Java CoAP library. It implements and supports both the client and server sides of the CoAP protocol. [7]

2.7 Paho MQTT Client

The Paho Java Client is an MQTT client library written in Java for developing applications that run Java compatible platforms. [8]

3 Methodology

This chapter describes and presents the project's overall workflow in the form of its methodology, used tools and methods. How the goals were achieved.

The most fundamental part before starting to implement all necessary code was to break down how the CoAP and MQTT messages are structured, it's format, byte for byte, bit for bit. A time-consuming task but which facilitated the phase of implementing the client and broker. The MQTT OASIS Standard Manual was essential for the understanding and latter implementing the MQTT broker. Similar for the CoAP client the RFC 7252 manual was examined. That hardest part when it came to this was understanding how the option header for the CoAP message functioned and when to use its extended format.

3.1 Tools

To be able to implement everything, several different development tools are used throughout the project, these are mentioned below.

3.1.1 Java

The Programming language which is used for the development is Java Development Kit (JDK) version 17. Including several of libraries, such as MjCoAP for the CoAP Server, Paho Java Client for the MQTT Client and JavaFX for the front-end implementation.

3.1.2 Software

Visual Studio Code is the integrated development environment (IDE) used with several needed extension to support the implementation. Scene Builder is used for designing the JavaFX GUI and MQTT.fx as a MQTT Client for testing purposes. The project is built with the project management and comprehension tool Apache Maven. This among other things facilitates the process of including the different libraries and dependencies.

3.1.3 Hardware

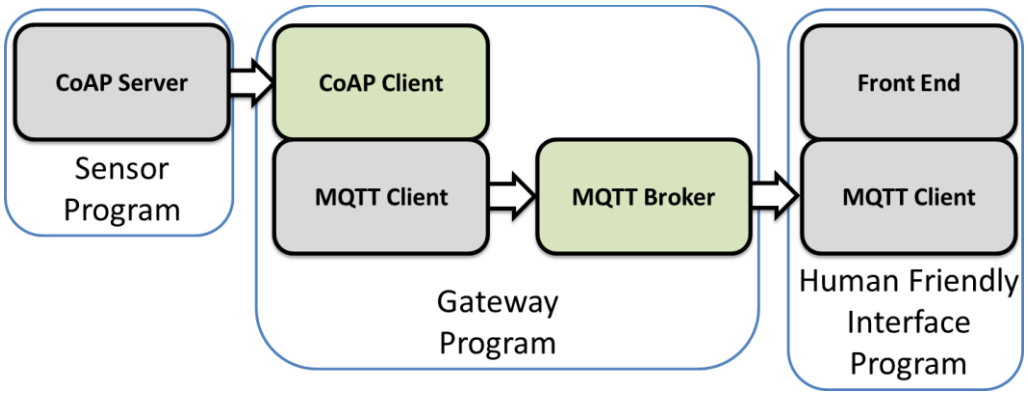
The project will be performed locally on a laptop running the operating system Microsoft Windows 11, 64-bit.

4 Implementation

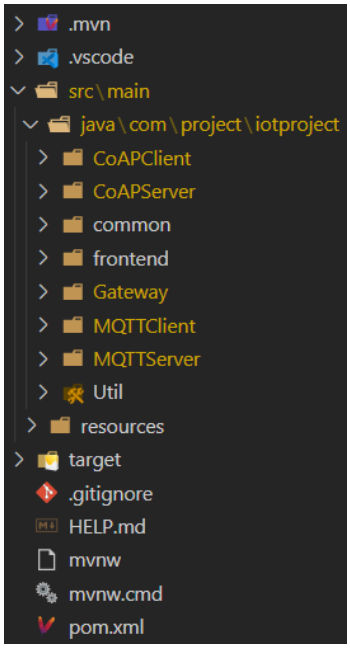
This chapter contains and describes all the practical details that involves the implementation process, based on the description in the method chapter 3.

4.1 Overall implementation details

Initially the most essential part was, as mentioned, examining the two manuals in detail, both before and under the implementation phase.



The image above illustrates how all project components will be divided and setup, explained in detail in the following subchapters.



The division of these components is well thought out which is reflected in the structure of the code implementation.

The *Util* folder contains functions which are required in many places in the rest of the code.

The IDE Visual Studio Code with Apache Maven made this implementation phase more efficient. More specifically the *pom.xml* file which contains information about the project and configuration details to build the project. Including all used dependencies, such as *mjcoap*, *paho* and *javafx*.

4.2 CoAP Server

This sensor program uses the *MjCoAP* library (included with the dependency in the image below). to implement the CoAP Server by extending *AbstractCoapServer*. Here *GET* requests with a message type on *NON* are handled with the built in *handleGetRequest* function which takes in *CoapRequest* method.

```
<dependency>
    <groupId>org.mjcoap</groupId>
    <artifactId>mjcoap</artifactId>
    <version>1.0</version>
    <scope>system</scope>
    <systemPath>${project.basedir}/src/main/java/com/project/iotproject/CoAPServer/mjcoap/lib/mjcoap.jar</systemPath>
</dependency>
```

When a message request has been acquired from a Client depending on its resource name, using *req.getRequestUriPath()*, different data responses are returned to the Client. These possible responses are, real CPU temp of the computer the program is running on, the local data and time, a random generated number, a random payload message or using a weather API, either the real current outside humidity or temperature in Sundsvall.

4.3 Gateway

The gateway program consists of and is divided into a CoAP Client, created from scratch, and a MQTT Client, implemented using the self-made Paho Java Client library (included with the dependency in the image below).

```
<dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
    <version>1.2.5</version>
</dependency>
```

All components and methods for the implemented CoAP Client were divided in a way which was recognizable from how the CoAP messages is structured and formatted. Which the CoAPMessage class then in-turn uses and implements. This message class is what latter is encoded to a byte array and sent to the Server, vice versa when a message is received.

The CoAP Client communicates with the CoAP server by using a *DatagramSocket* to send a *DatagramPacket* to a specified *InetAddress* Internet Protocol (IP) address. With the same methods applied when receiving CoAP messages.

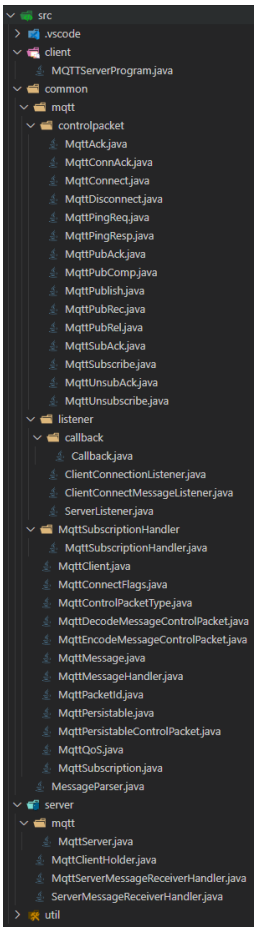
Using the Paho Java Client library a MQTT Client is created, by specifying the *serverURI* and a *clientId*, to communicate with a MQTT Server. These two parameters form a key which is used to store and reference messages while they are being delivered.

A thread is used to at a timed interval send different, selected at random, request messages by the CoAP client to the CoAP sensor server. These messages with set CoAP options correspond to what Uri-Path values the CoAP sensor server can handle. When the message response has been acquired the MQTT client then uses this data and forwards it to the MQTT Broker by publishing a message. This message contains data which corresponds to the CoAP server response payload as the MQTT publish message payload and URI Request Path as the MQTT publish message topic.

4.4 MQTT Broker

The MQTT broker server program with the most essential functions is created from scratch. It can receive, interpret, and send its correct corresponding messages to connected MQTT Clients. It's able to handle subscribe and unsubscribe requests, as well as handle clients that publish information to the subscribed topics.

All components and methods were implemented and divided in a way which was recognizable from how the MQTT messages is structured and formatted. Which the MqttMessage class then in-turn uses and implements. The message is divided into the two different headers, fixed and variable. This message class is what latter is encoded to a byte array and sent to and from the clients, vice versa when a message is received.



Every connected client which subscribes to a topic is stored in a map format on the server. This is later used when the broker receives a publish message at which it forwards it to all clients which have subscribed to this message corresponding topic.

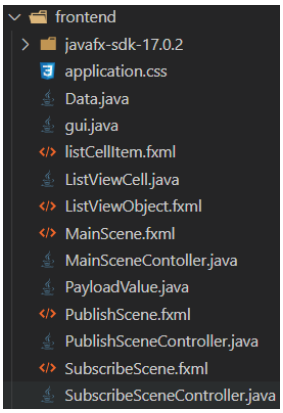
```
HashMap<String, ArrayList<MqttClient>> topicSubscriptionMap;
```

When this message has been processed and if accepted by the server broker the client is added to a container which holds all connected clients.

```
public class MqttClientHolder {
                                                    // Key, value.
    private HashMap<Integer, String> mqttClientIdMap;   // 1,2,3... , ClientId
    private HashMap<String, MqttClient> mqttClientMap;  // ClientId , MqttClient
```

A simple terminal console server-side application which displays the different executed tasks were implemented. The server runs locally on the default TCP/IP Port 1883.

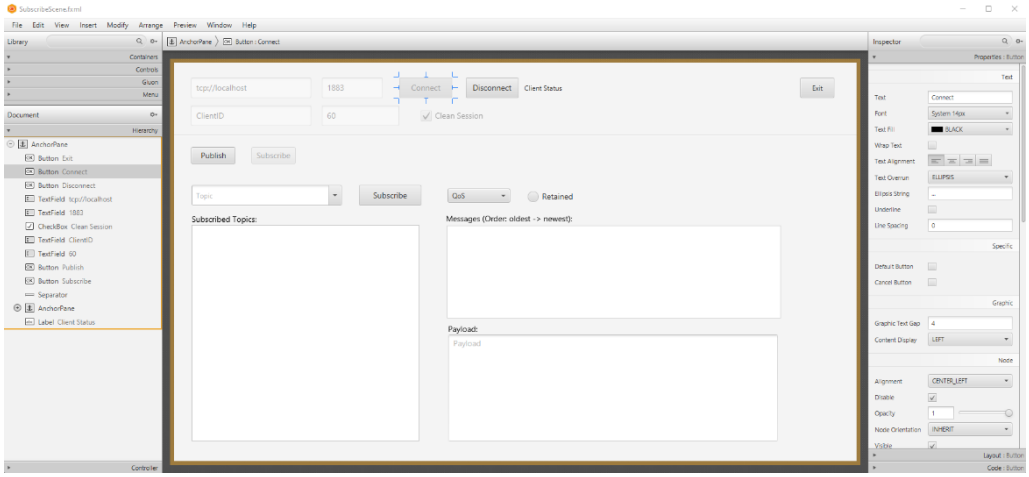
4.5 Front End



Here a human friendly JavaFX GUI which is using the Paho Java Client library to run a MQTT Client is created.

It's divided into 3 scenes corresponding to the different client method functions. The main scene functions as connecting to a broker. The subscribe scene for subscribing to a topic and viewing published messages relating to those topics. Lastly the publish scene which is used for publishing messages with a topic and payload to a MQTT broker. The methods here for the MQTT Client implementation is the same as described in chapter 4.3.

FXMLLoader is used to both switch between and load specific FXML scene documents. Methods are marked with a *@FXML* annotation to make them accessible to markup. Essentially deploying them as a module. By this a controller class can then be used to bind defined methods to actions in the FXML document. By in the controllers implementing *MqttCallback* enables the application to be notified when asynchronous events related to the client occur. Used by the methods, *connectionLost*, *messageArrived* and *deliveryComplete*. With messages are still received nonmatter what the current scenes is.



With the Scene Builder tool an AnchorPane is used as the root at which all the different components are added to.

4.6 Performance Measurements

Time measurements between different points in the code execution phase are taken and saved to a text document file. Based of these a mean value is then after 50 number of sent handled messages iterations calculated. The time is taken of when the CoAP sensor server sends a message to the point in which the GUI MQTT client receives that message. Also, the time for a the CoAP sensor server to send a message, CoAP client to send and receive a message, MQTT client to publish a message and for the MQTT Broker to publish a message to every subscriber to a certain topic.

5 Results

This chapter objectively presents all results of this project, divided into the four main components. The full chain from the CoAP server sending values to the MQTT Client GUI displaying these values is presented.

5.1 CoAP Server

The CoAP Server generated different option values based of the Uri-Path value. Presented as one of the values of the current local time in the following subchapter and finally displayed as a payload message in the MQTT Client GUI.

5.2 Gateway

The gateway holding the CoAP and MQTT Client prints information to the terminal containing and confirming the a publish message have been sent, using the *deliveryComplete* function.

```
optionValue: time
TimeElapsed: 14
Delivery complete callback: Publish Completed [time]
```

5.3 MQTT Broker

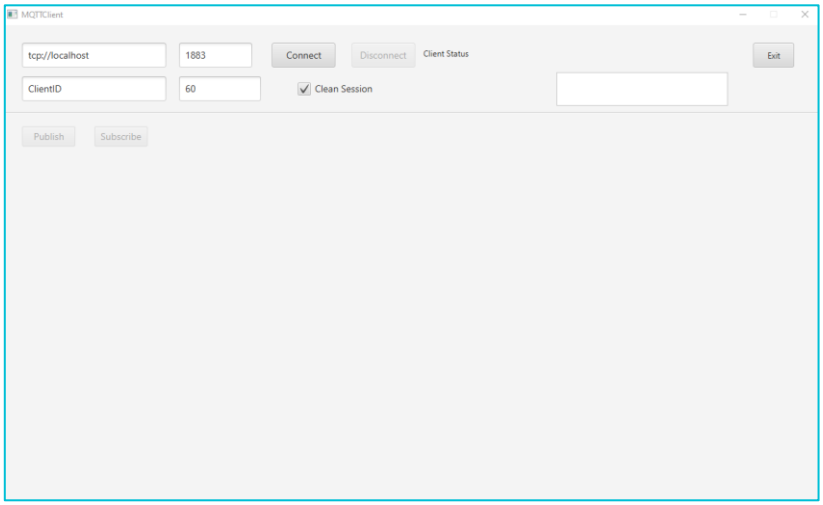
A simple terminal console server-side application which displays its different executed tasks were implemented, shown in the image to the right.

Here the Broker have received a Publish packets and decoded the byte array to a readable MqttMessage format.

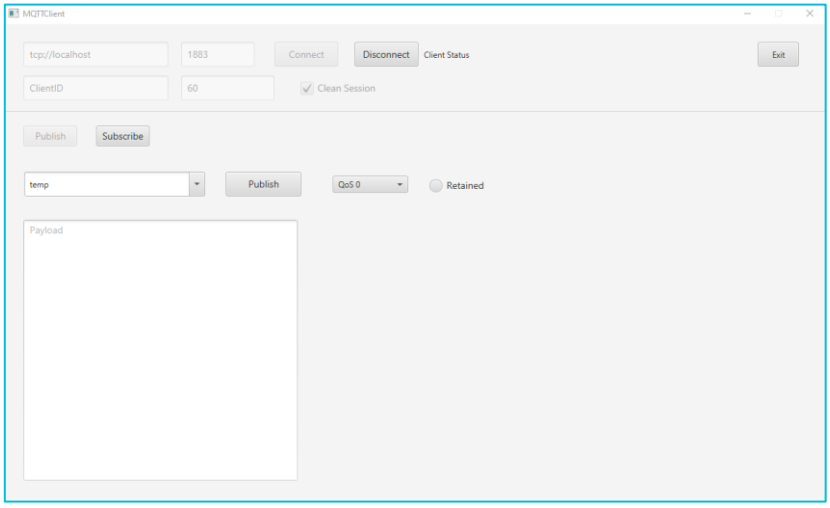
```
Packet Received:
ControlPacket Type: PUBLISH
Remaining Length: 63

---Mqtt Publish Control Packet---
DUPFlag: false
Retain: false
QoS Level: ATMOSTONCE
Packet Identifier: 0
Payload: Current Time(CET): 15:57:13
Time: 2022-02-22 15:57:13.539
Topic Name: time


Bytes:
Byte 0: 00110000
Byte 1: 00111111
Byte 2: 00000000
Byte 3: 00000100
```

5.4 Front End

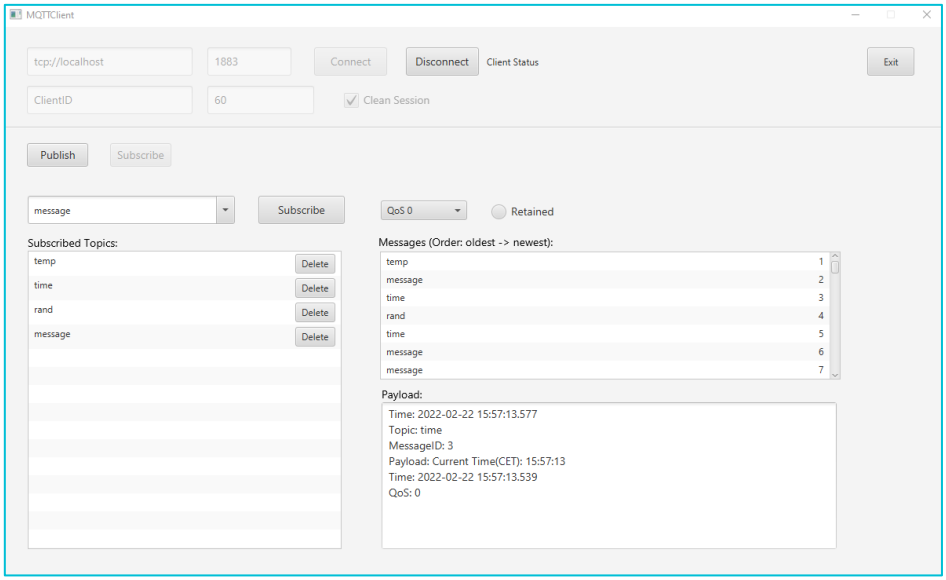
Here are the results presented of the front end MQTT Client GUI.



The image above shows the connecting to client scene. This is what first is presented to the user when the GUI is initiated. Here it possible to change different values before connecting. The Text Area at the top right corner can display possible error messages related to written connection parameters. The "Client Status" label displays status related to connecting to a broker and other necessary related information.



This is the publish scene where a user can send publish messages, to a MQTT broker, with a selected topic, QoS, retained and payload message.



The image above shows the subscribe scene. Here the user can subscribe to a selected topic, QoS and retained value. Including receiving messages of to that topic and viewing its content. The image below shows what the MQTT Client receives in the implemented *messageArrive* function which is then printed in the GUI.

```
Total time elapsed: 23
Time: 2022-02-22 15:57:13.577    Topic:time    Message: Current Time(CET): 15:57:13
Time: 2022-02-22 15:57:13.539    QoS: 0    Retained: false    id: 0
```

5.5 Performance Measurements

Below follows a table of the calculated averages from 50 individual time measurements. Here 1 ms is the lowest time that for this project and used methods could be measured.

	CoAP Server	**CoAP Client (part of Gateway)**	**MQTT Client (part of Gateway)**	**MQTT Broker**	**CoAP Server -> MQTT Client GUI**
Average time in Milliseconds	1 ms	41 ms	1 ms	1 ms	22 ms

6 Conclusions/discussions

This chapter analyzes and discusses the project method and the results.

The presented results displayed the implement proof-of-concept application. A combined IoT system which handles both the CoAP and MQTT protocols. Where the developed programs were able to have the CoAP sensor generate values and then transferred across a chain to a MQTT human friendly front-end interface application. By this the goals for this project have been reached.

The most time-consuming task was for the CoAP Client to send and receive a response from the server. This time is very dependent on the execution time of the server. Hence the computing time varies depending on the task the CoAP server sensor performs. Where for instance fetching data from an API is more time consuming compared to generating a random number. From what could be seen based of all these quantitative measurements the results were good and the systems functioned very well. But since no comparison or real knowledge of what the expected values should be it hard to make a more realistic conclusion based of this. Including since this was also only preformed locally. By this it could be good to in the future deploy this in a more open environment with for instance a more realistic IoT device.

Afterwards all made choices for this project had worked very well with no major drawbacks. Where the most important and best choice was thoroughly studying the protocols corresponding manuals.

Based on the describe background and problem motivation contra the acquired results this developed software could have a positive impact for transferring data between IoT devices. It most important contribution is the combination of CoAP and MQTT. A platform which acts as a form of gateway for handling both these protocols. Including being portable in the sense of easy to deploy on almost any system.

Since this is no complete product there are a lot of improvements and various of optimizations which could be made for a future development.

References

[1] En.wikipedia.org. 2022. Internet of things - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Internet_of_things> [Accessed 21 February 2022].

[2] Datatracker.ietf.org. 2022. RFC 7252 - The Constrained Application Protocol (CoAP). [online] Available at: <https://datatracker.ietf.org/doc/html/rfc7252> [Accessed 21 February 2022].

[3] Docs.oasis-open.org. 2022. MQTT Version 3.1.1. [online] Available at: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html> [Accessed 21 February 2022].

[4] 2022. [online] Available at: <https://java.com/en/download/help/whatis_java.html> [Accessed 21 February 2022].

[5] Openjfx.io. 2022. JavaFX. [online] Available at: <https://openjfx.io/> [Accessed 21 February 2022].

[6] 2022. [online] Available at: <https://gluonhq.com/products/scene-builder/]> [Accessed 21 February 2022].

[7] 2022. [online] Available at: <http://netsec.unipr.it/project/mjcoap/]> [Accessed 21 February 2022].

[8] 2022. [online] Available at: <https://www.eclipse.org/paho/index.php?page=clients/java/index.php> [Accessed 21 February 2022].