Mittuniversitetet
MID SWEDEN UNIVERSITY

Implementing Internet of Things Protocols

Elias Berglin

MID SWEDEN UNIVERSITY
Department of Information Systems and Technology (IST)

Main field of study: Computer Engineering
Credits:
Semester, year: VT, 2022
Supervisor: Stefan Forsström
Examiner: Mikael Gidlund

Abstract

With the number of IoT devices increasing in the world different ways of communication has emerged. Two of these (MQTT and CoAP) have become really prominent. The problem with different communication standards is that sometimes different systems need to communicate with one another. This project aims to build an end to end system where the main data provider is a CoAP-server and the consumer uses an MQTT-client. The report results in a functioning system where a user application built using Flutter can show real-time data from a Raspberry Pi running a CoAP-server. Round trip time was measured to evaluate the system, and it was concluded that the project was a success but with such powerful devices a pure MQTT implementation would have been a better solution.

Keywords: CoAP, MQTT, Flutter, IoT

Table of Contents

Abbreviations

RTT	Round Trip Time
MQTT	Message Queue Telemetry Transport
CoAP	Constrained Application Protocol
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IoT	Internet of Things
UI	User Interface

1 Introduction

1.1 Background and problem motivation

IoT is a growing market in the world with MQTT and CoAP emerging as the dominant protocols for communication. Different IoT systems might need to communicate with one another event tho they might use different protocols. To do this a translator is needed to be able to translate CoAP to MQTT.

1.2 Overall aim

The aim for this project is to implement a system that translates responses from a CoAP server to a MQTT client running on a mobile application. The scenario is to monitor the CoAP server system usage and prepare a system for future integration with temperature sensors.

1.3 Concrete and verifiable goals

The goals for this project

- Combine a MQTT-client with a CoAP-client for translation.
- Implement a CoAP-server capable of serving CPU, memory usage and mock temperature sensors.
- Create a user application capable of displaying the information.
- Perform measurements on the system for evaluation.

1.4 Scope

This project will focus on implementing an end to end system from a CoAP-server to MQTT-client through a MQTT-broker. The only measurement that I will be conducting is the RTT from the client application to the CoAP-server from this the mean, min max and standard deviation will be calculated. From the total time for x amount of requests the request per second will be calculated.

1.5 Outline

Chapter 2 will go into theory about the technology used. Chapter 3 will motivate the technology choices as well as describe the system. Chapter 4 will go through the construction process for the system. Chapter 5 will present the result from the measurements. Chapter 6 will discuss the result as well as draw conclusion from this project.

1.6 Contributions

This project and report was done by me. External open-source packages was used to complete the project. All external packages are credited as citations.

2 Theory

2.1 CoAP

CaAP is a communication protocol over UDP aimed for lower powered computers. It is often used by small microcontrollers that only have limited storage and RAM available. CoAP uses the client server model. The server accepts requests and then responds. The server also provides methods for clients to discover resources on the server. CoAP was designed to be easily translated to HTTP and uses similar request to HTTP, for example GET and POST. [1]

2.2 MQTT

MQTT is a communication protocol over TCP. It is lightweight and provides an open communication channel between devices. The MQTT system consists of one broker and many clients. A client can subscribe and publish data to topics. The broker then makes sure to distribute the data to all subscribers. The protocol was designed in the 1990s for use in oil pipeline monitoring. Today it is more commonly used in IoT applications. [2]

2.3 Flutter

Flutter is a framework for building multiplatform applications. Flutter is open source and developed by Google. Flutter applications are written in dart and the framework can compile the dart code to machine code for both x86 and ARM. It can also combine the code to JavaScript for web compatibility. [3] As of version 2.0 of Flutter the following platforms are supported: [4]

- Android
- IOS
- Web
- Windows (beta)
- Windows UWP (alpha)
- Linux (beta)
- macOS (beta)

Flutter is built upon the concept of widgets. A widget can be everything from a container to a button. Flutter has two different types of widgets: Stateful and Stateless. The difference here is state. Stateful widgets can react and re-render the UI on state changes while Stateless is static.

3 Methodology

3.1 System overview

The structure of the IoT system including the planned devices can be found in figure 1.

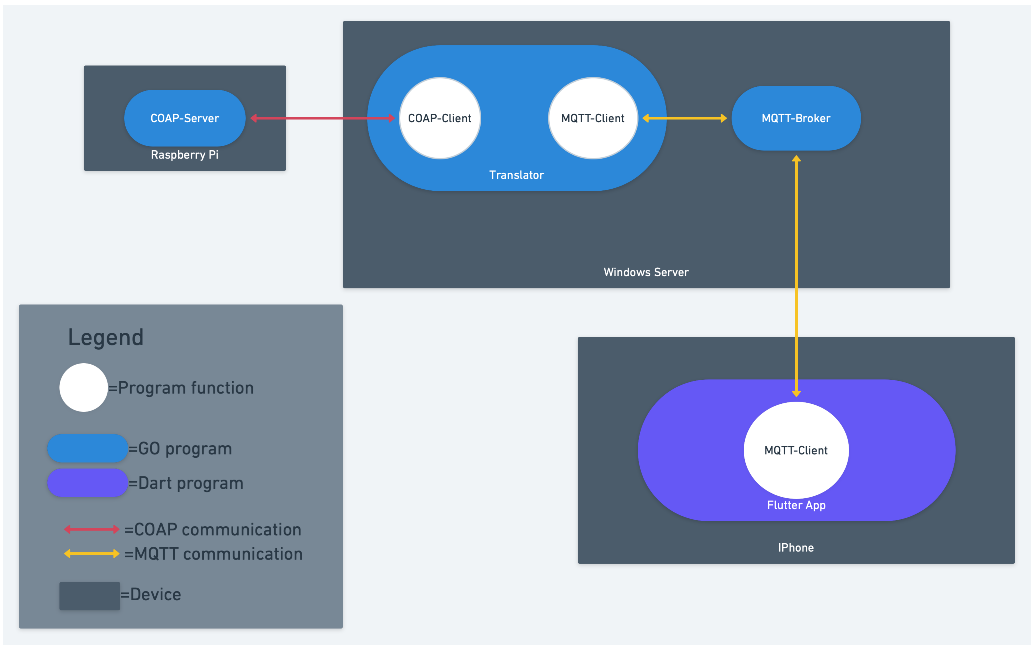


Figure 1: The planned communication scheme. See legend for explanation

3.2 Choice of technologies

Two components of the system was already developed as part of two previous projects. These are the CoAP-Client and the MQTT-broker. These where implemented in Go for its excellent concurrency. This does not affect the CoAP-client, but it is useful for the MQTT-broker for handling multiple connections. Another perk of Go is how easy it is to create the different connection types. Lastly Go has a built-in datatype for Bytes and that came in handy when reading and writing data from sockets.

Because of the Go implementation on the previous parts it was only natural to have the CoAP-server in Go. To create a server an external library was used [5]. The translator needs to combine both a CoAP-client and a MQTT-client. To be compatible with the created CoAP-Client the MQTT-client needs to be written in Go. For this I also used a Go library [6]. Lastly

Flutter will be used to create the user interface. Flutter was chosen for its capability to be compiled to multiple platforms. This allows for the app to be installed on multiple devices' whiteout needing to rewrite the application. To communicate with the MQTT-broker an external dart library was used [7].

3.3 System description

The primary purpose of the server is to collect system data more specifically CPU and RAM usage. A client will then be able to request this information. Additionally, the server can create mock temperature sensors. A sensor has a location, power status and a temperature. A client can request temperature, create a sensor, change power status and remove a sensor.

The translators have two purposes. It will collect data from the CoAP-server and publish using the MQTT-client. It will also react to published messages from other MQTT-clients and create CoAP-request accordingly. Ex. the Flutter app wants to create a sensor. The translator can then convert a publishing by the application to a CoAP POST-request.

The MQTT-broker is a standard MQTT-broker implemented using Golang. It will handle the subscription and publish messages, creating a communication channel between the translator and the Flutter application.

Lastly the Flutter application will be multi-platform application acting as the frontend of the IoT system. The app will have the following capabilities.

- Display CoAP-server system information.
- Display temperature sensor information.
- Create temperature sensors.
- Turn off specific temperature sensors.
- Delete a Temperature sensor.
- Run a benchmark.
- Display benchmark results.

The system requires one active CoAP-server, one active translator, one active MQTT broker and one or more Flutter applications. This implementation will put the CoAP-server on a Raspberry Pi running Raspbian and bundle the MQTT-broker and the Translator on a Server running Windows Server. The Flutter application will run on at least IOS but should be able to run on any operating system supported by Flutters build system.

3.4 Evaluation

To evaluate the system the RTT will be measured between the user Application and the CoAP server. This will be done using a benchmark function inside the Flutter application. Three benchmark runs will be done adding a new instance of the flutter application with each run to simulate different load on the system. The Flutter application will output the following metrics:

- Requests per second
- Average RTT
- Max RTT
- Min RTT

4 Implementation

4.1 CoAP-server

As mentioned in chapter 3.2 the CoAP-server is implemented in Go using an external library [5]. The library implements CoAP similarly to how Go normally handles HTTP. To start serving CoAP a router is created for routing request to different locations. A middleware was created to enable logging. The first rout added was the rout to get the mock temperature sensors. To create a rout an endpoint and a handler function needs to be provided to the router. In figure 2 we can see the code snipped to create the router, add the middleware and lastly creating the first route.

```go
r := mux.NewRouter()
r.Use(loggingMiddleware)
r.Handle("/temp", mux.HandlerFunc(handleTemp))
```

Figure 2: Go code snipped showing creating the CoAP router, adding a logging middleware and creating the handler for temperature

The temperature handler decodes the incoming CoAP message to figure out what type of request it was. It also checks if the message has a payload. All the request types to this endpoint requires a payload. Figure 3 shows the code snippet for the initial handling of a request to the temperature endpoint. If the body does not exist it ends the request there by responding with bad request.

```go
method := r.Code.String()
body := make([]byte, 128)
if r.Body == nil {
    w.SetResponse(codes.BadRequest, message.TextPlain, bytes.
       NewReader([]byte(string("No body included"))))
    return
}
n, _ := r.Body.Read(body)
body = body[:n]
```

Figure 3: Code showing initial handling of the temperature request

The handler can handle 4 different request types: GET, POST, PUT, DELETE.

GET responds with the current temperature of the processor if it exists otherwise it returns not found. Because the sensor is simulated, after the

GET request it either increase or decrease the temperature of the sensor by a random amount between 1-3°C. Figure 4 shows the structure for a temperature sensor.

```
type TemperatureSensor struct {
    Status      bool
    Temperature int
    Location    string
}
```

Figure 4: Structure for a mock temperature sensor

POST creates a new temperature sensor. The request requires the payload to contain both the location of the sensor and the initial power status. When the request is received it creates a new instance of a temperature sensor. It assigns the values received and randomizes a temperature between -20-30°C. If the request is successful it responded with status code created.

PUT request are used to switch power status. The request requires the body to contain the new power status and the name of the sensor to be changed. If the sensor exists it changes its power status and responds with status Changed.

DELETE request removes a sensor if the payload includes the sensor name. If the request is successful it responds with Deleted.

For clients to be able to discover what temperature sensors is available the /all endpoint was created. This endpoint only accept GET request and uses Go's built in JSON library to convert the internal map of sensors to a JSON string to include in the response. The code implementation can be found in figure 5.

```
r.Handle("/all", mux.HandlerFunc(func(w mux.ResponseWriter, r
    *mux.Message) {
    data, _ := json.Marshal(thermostats)
    w.SetResponse(codes.Content, message.TextPlain, bytes.
       NewReader(data))
}))
```

Figure 5: Creation of the /all endpoint

The last endpoint is /pi which enables clients to gather information about the CoAP-server's processor and memory utilization. The server collects this data through the Linux proc files. CPU information is gathered through the proc/stat file while the memory is gathered through the proc/meminfo file. I used a library to help width parsing these files[8]. A separate thread reads the proc files every second. For CPU usage it uses the last calculated value to derive the current usage in percentage. For the memory the proc/meminfo contains both free memory and total memory. From this the function derives the current usage in the format used/total.

4.2 Translator

The translator builds on a previous implemented CoAP client in Go. To translate from CoAP to MQTT a MQTT library was added to the project[6]. The translator will poll data from the CoAP and then publish it to the MQTT-broker. It will also subscribe to certain MQTT messages to be able to convert them to CoAP request. The response from these messages will then be published.

The first thing the Translator does is send a request to the CoAP-server to retrieve the available sensor. This is to keep an internal map of what sensor are online to prevent the Translator to pull unnecessary data. After it retrieves the information about the temperature sensors it starts the connection to the MQTT broker and creates some constant subscriptions. When subscribing the MQTT library requires callback functions that will fire when a message is received to that topic. The connection code can be found in figure 6

```go
populateSensors()
var broker = "localhost"
var port = 1883
opts := mqtt.NewClientOptions()
opts.AddBroker(fmt.Sprintf("tcp://%s:%d", broker, port))
opts.SetClientID("Coap Translator")

opts.SetDefaultPublishHandler(messagePubHandler)
opts.OnConnect = connectHandler
opts.OnConnectionLost = connectLostHandler
opts.SetPingTimeout(time.Second * 60)

client := mqtt.NewClient(opts)

if token := client.Connect(); token.Wait() && token.Error()
    != nil {
    panic(token.Error())
}
```

Figure 6: Connection code for the Go MQTT client

The constant subscription are: all, home/add, home/delete and home-

/change. The all topic is used for communication to the MQTT network the available temperature sensors. If an MQTT client publish a message with the structure GET:*id*. The Translator will get the information using the CoAP client and then forward the response to MQTT network via a publish message to the topic all/*id*. The function for parsing all the publish messages can be found in figure 7

```go
func allCallback(c mqtt.Client, m mqtt.Message) {
    message := string(m.Payload())
    fmt.Println(message)

    if strings.Contains(message, "GET") {
        split := strings.Split(message, ":")
        if len(split) != 2 {
            return
        }
        COAPmsg := createGet("all", "")
        payload := string(sendCreatedCoap(COAPmsg))
        fmt.Println(payload)
        tok := c.Publish("all/"+split[1], 0, false, payload)
        tok.Wait()
    }
}
```

Figure 7: Function for parsing the /all endpoint

The home/add subscription is used to create a sensor. The Translator listens for publish messages and then relays the information to the CoAP-server. Figure 8 shows a code for the callback function to the home/add subscribe.

```go
func addCallback(c mqtt.Client, m mqtt.Message) {
 message := string(m.Payload())
 msg := createPost("temp", message)
 payload := string(sendCreatedCoap(msg))
 fmt.Print(payload)
 populateSensors()
}
```

Figure 8: Function for parsing the home/add publish

The home/change and home/delete functions similarly to home/add. home/change is used to change the power status of a temperature sensor and the home/delete is used to delete a temperature sensor. For all of these function the Translator repopulates the internal map to be up-to-date on what sensors exist and their power status

The main loop of the Translator is to first pull the temperature sensors if they are online. It then publishes this temperature to home/*id*. After it has taken care of the temperature it moves on to the CPU and memory information. It creates a request to the CoAP-server and then splits the information in to two publishes pi/cpu and pi/mem. When it has published the data it sleeps for 10 seconds and then starts from the top. The code for the main loop can be found in figure 9

```go
for {
    for k, online := range sensors {
        if online {
            msg := createGet("temp", k)
            p := sendCreatedCoap(msg)
            tok = client.Publish("home/"+k, 0, true, p)
            tok.Wait()
        }
    }
    msg := createGet("pi", "")
    p := sendCreatedCoap(msg)
    //fmt.Println(string(p))
    tok = client.Publish("pi/cpu", 0, true, strings.Split(
       string(p), ":")[0])
    tok.Wait()
    tok = client.Publish("pi/mem", 0, true, strings.Split(
       string(p), ":")[1])
    tok.Wait()
    time.Sleep(time.Second * 10)
}
```

Figure 9: The main loop of the Translator

4.3 Flutter mobile application

The frontend application was created using the Flutter framwork. To handle the IoT connection an external library [7] was used. Dart is a single threaded language but uses a similar system to JavaScript to handle asynchronous code. This enables the application to do different things at the same time. The MQTT library requires a callback function to be passed handle all the publish messages. This differs from the way the Go library handles it due to it using different callbacks for different topics.

The app has one screen that displays the information it receives from the MQTT broker. The screen consist of two grids. One for displaying the CoAP-server system information and the other for displaying all the temperature sensors. A sensor is represented with a tile that the user can interact with. The user can delete a sensor by touching and holding on the tile. To change power state a switch is present in the tile. Figure 10 shows the application home screen and figure 10 shows the code for one tile.

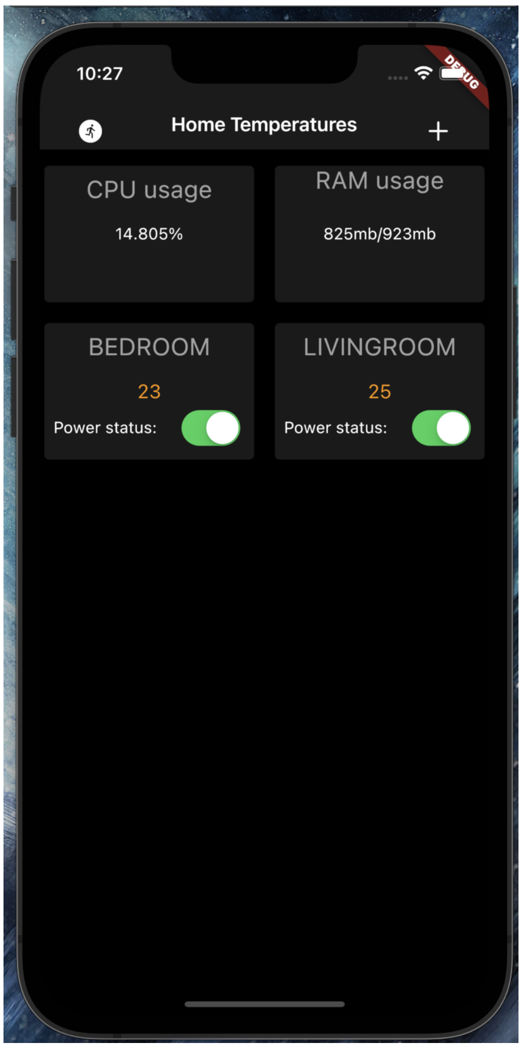


Figure 10: Flutter application main screen

```
GestureDetector(
onLongPress: () {
    openDialog(context, i);
},
child: Card(
    color: Colors.white10,
    child: Padding(
    padding: const EdgeInsets.all(8.0),
    child: GridTile(
        header: Text(
        sensorData.sensors[i].Location.toUpperCase(),
        style: const TextStyle(color: Colors.white60),
        textScaleFactor: 1.5,
        textAlign: TextAlign.center,
        ),
        child: GridTileBar(
        subtitle: sensorData.sensors[i].Status
            ? TemperatureText(
                sensorData.sensors[i].Temperature)
            : const Text(
                "Offline",
                textAlign: TextAlign.center,
                ),
        ),
        footer: Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
            const Text(
            "Power status: ",
            textScaleFactor: 1,
            style: TextStyle(color: Colors.white),
            ),
            CupertinoSwitch(
            value: sensorData.sensors[i].Status,
            onChanged: (newVal) =>
                sensorData.changePowerStatus(newVal, i),
            ),
        ],
        ),
    ),
    ),
),
);
```

Figure 11: The Dart code for creating a tile

To help with keeping a global state a package[9] was added. This package enables widgets to listen for changes in class variables and function outputs and then updates the UI accordingly. I used this package to manage a global state of temperature sensors and system usage of the CoAP-server.

Whenever new data is available the class calls notifyListeners() functions that tells every listener to update the UI.

To enable the user to add a new temperature sensor a dialog popup was created. The popup requires the user to input a name and the initial power status for the temperature sensor. A dialog in Flutter does not have the same state capabilities as a normal Widget. For user input to be validated state is needed. I solved this by using a special widget called StatefulBuilder. This widget creates its own inner state and instead of taking children it takes a function that returns a widget. Inside this function it has its own state management to re-render the UI as needed. The code for the dialog can be found in figure 12

```
StatefulBuilder(
builder: (context, setState) {
    return CupertinoAlertDialog(
    title: const Text("Add sensor"),
    content: Column(
        children: [
        CupertinoTextField(
            placeholder: "location",
            keyboardType: TextInputType.name,
            controller: _locationController,
        ),
        const SizedBox(
            height: 10,
        ),
        Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween
                ,
            children: [
            const Text(
                "Power status: ",
                textScaleFactor: 1.5,
            ),
            CupertinoSwitch(
                value: initialState,
                onChanged: (val) => setState(
                () {
                    initialState = !initialState;
                },
                ),
            ),
            ],
        ),
        ],
    ),
    actions: [
        CupertinoDialogAction(
        child: const Text("Close"),
        onPressed: () => Navigator.of(context).pop(),
        isDestructiveAction: true,
        ),
        CupertinoDialogAction(
        child: const Text("Add"),
        onPressed: submit,
        ),
    ],
    );
},
);
```

Figure 12: The Dart code for creating the add dialog

The last component to complete the app is the capability to run benchmark. To make it easier to create metrics a package[10] was added. The package can calculated a number of different metrics given an array of data. For this project we are only interested in mean, standard deviation, min and max.

To get the benchmark results a function is run to calculate the RTT. The app publishes to /all. This publish message is then distributed through the MQTT-broker and picked up by the Translator. The Translator then sends a request to the CoAP-server. When the Translator gets a response from the CoAP-server it the publishes to the MQTT-broker which then distributes the message back to the Flutter application. When the application receives the message it can then derive the round trip time.

To ensure only one message is sent at a time another package[11] containing mutex locks was added. Even tho Dart is single thread code can still occur asynchronous. Before a new message is sent from the application it needs to acquire the mutex lock. The lock is then released upon receiving the response. This way only one request can be active at a time. The code for the main benchmark function can be found in figure 13 and a code snippet showing how the stats package is used can be found in figure 14

```dart
Future<List<Duration>> benchmark(int runs) async {
    runningBenchmark = true;
    if (bench.isLocked) bench.release();
    b.clear();
    final benchStart = DateTime.now();
    for (var i = 0; i < runs; i++) {
        await bench.acquire();
        print("run: $i");
        lastStartTime = DateTime.now();
        final builder = MqttClientPayloadBuilder();
        builder.addString("GET:$devId");
        client.publishMessage("all", MqttQos.atMostOnce,
            builder.payload!);
    }
    await bench.acquire();
    runningBenchmark = false;
    totalTime = DateTime.now().difference(benchStart).
        inMilliseconds;
    bench.release();

    return b;
}
```

Figure 13: The Dart code for running the benchmark

```
final stat = Stats.fromData(l.map((e) => e.inMilliseconds)
        .toList()).withPrecision(3);
final reqPerSec = l.length /
(Provider.of<SensorProvider>(context, listen: false).
   totalTime / 1000);
```

Figure 14: The Dart code for calculating statistics

The benchmark results are displayed in a dialog popup. While the benchmark is running it displays running benchmark otherwise it displays the results. Examples of this can be found in figure 15 and figure 16 respectively.

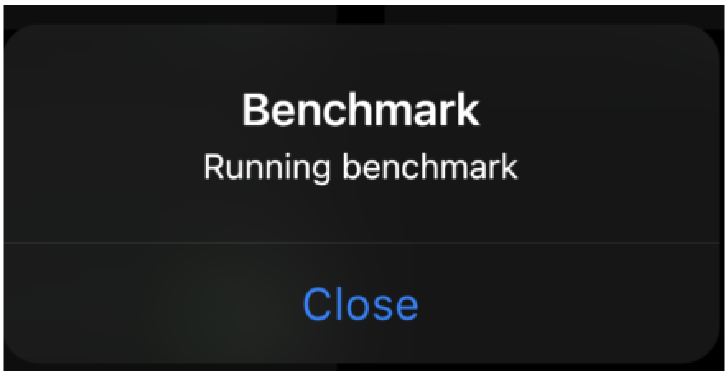
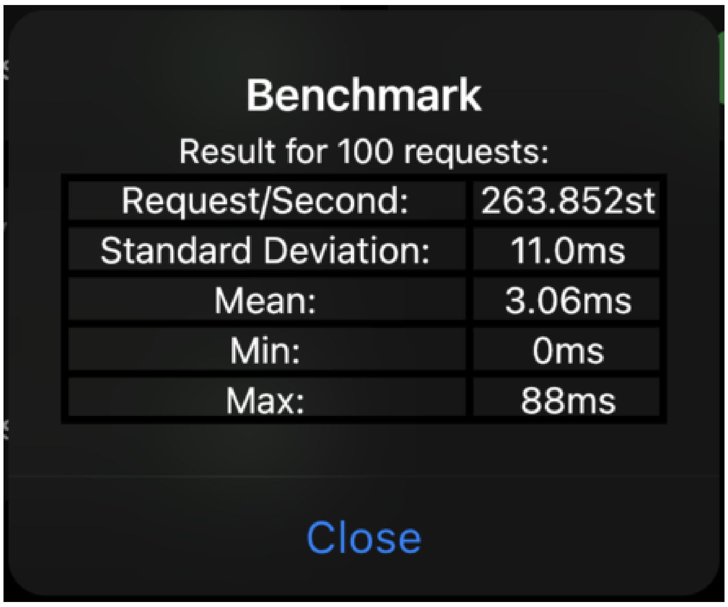


Figure 15: Dialog while benchmark is running



Result for 100 requests:

Request/Second:	263.852st
Standard Deviation:	11.0ms
Mean:	3.06ms
Min:	0ms
Max:	88ms

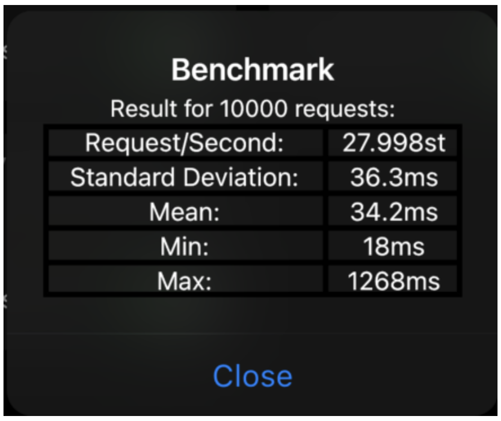
Figure 16: Dialog when benchmark is done

5 Results

The following results was collected with three separate runs. Each run added an instance of the application on different platform adding the number of clients making requests. All the platforms was run using the same MacBook Pro 14" with M1 Pro (8 core). The following platforms was used

- Device 1: iPhone 13 pro emulator (iOS 15.2)
- Device 2: macOS application (macOS 12.1)
- Device 3: Android Emulator (Android 12)

Figure 17, figure 18 and figure 19 shows the result for the three different benchmark runs.



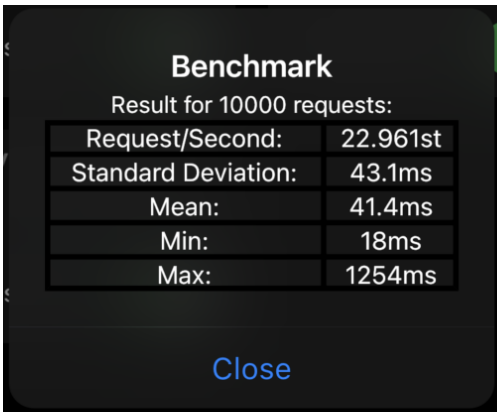
Benchmark

Result for 10000 requests:

Request/Second:	27.998st
Standard Deviation:	36.3ms
Mean:	34.2ms
Min:	18ms
Max:	1268ms

(a) iPhone 13 Pro

Figure 17: Benchmark run 1 with 1 devices

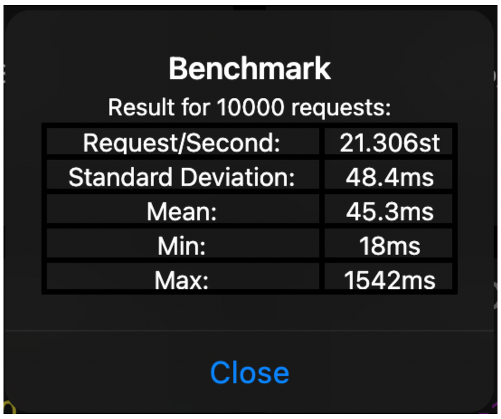


Benchmark

Result for 10000 requests:

Request/Second:	22.961st
Standard Deviation:	43.1ms
Mean:	41.4ms
Min:	18ms
Max:	1254ms

(a) *iPhone 13 Pro*



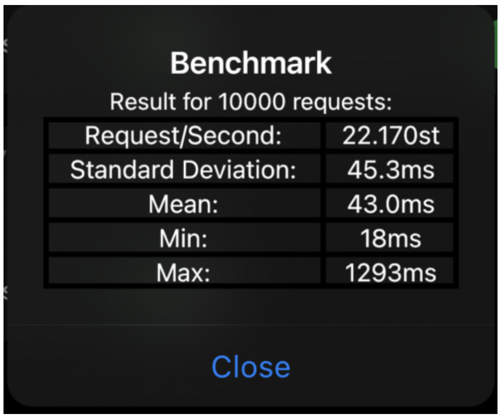
Benchmark

Result for 10000 requests:

Request/Second:	21.306st
Standard Deviation:	48.4ms
Mean:	45.3ms
Min:	18ms
Max:	1542ms

(b) *macOS application*

Figure 18: Benchmark run 2 with 2 devices

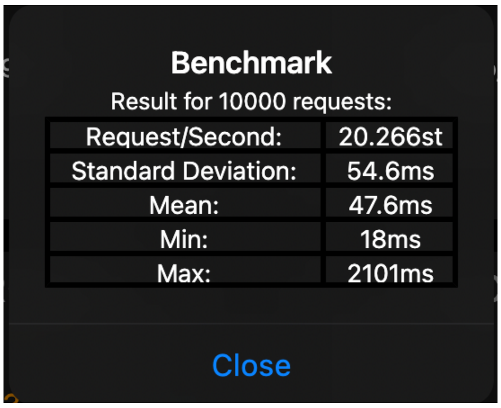


Benchmark

Result for 10000 requests:

Request/Second:	22.170st
Standard Deviation:	45.3ms
Mean:	43.0ms
Min:	18ms
Max:	1293ms

(a) *iPhone 13 Pro*

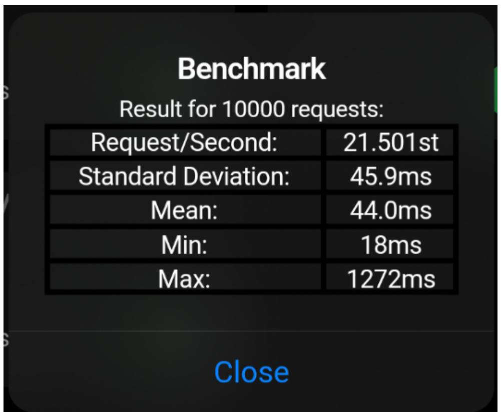


Benchmark

Result for 10000 requests:

Request/Second:	20.266st
Standard Deviation:	54.6ms
Mean:	47.6ms
Min:	18ms
Max:	2101ms

(b) *macOS application*



Benchmark

Result for 10000 requests:

Request/Second:	21.501st
Standard Deviation:	45.9ms
Mean:	44.0ms
Min:	18ms
Max:	1272ms

(c) *Android emulator*

Figure 19: Benchmark run 3 with 3 devices

6 Discussion

6.1 Implementation

A complete system was implemented with working communication from a MQTT client in a user application to a CoAP-server running. There is one bug in the system that I have not found a solution for. Sometimes the Dart version of the MQTT-client sends a connect-message with identity 0 which is a reserved message ID. As I have not created it was hard to find the cause for this bug and I was not able to create a successful workaround. Luckily this does not happen often.

Another problem was that I was not able to fully utilize Flutters multiplatform support. The MQTT Dart library does support all platforms, but it requires different client objects for web applications. Dart does provide a check if the application is running on the web but including the package for the web version of the MQTT-client also adds a package that is not supported by the other platforms thus breaking the application for other platforms.

By using Go for building everything is set up for concurrency, but I did not utilize it fully. Both the CoAP-server and the translator could be multithreaded using Go's concurrency features. This could theoretically decrease the latency.

6.1.1 Other alternatives

My implementation is not the only way to implement an end to end system for showing information from a remote device. Other alternatives could be using HTTP REST, WebSockets, pure CoAP and pure MQTT.

Both MQTT and WebSockets offer an open bidirectional channel for communication between client and servers. The difference here is that MQTT has a structure for how clients send and receive messages. WebSockets on the other hand only is a standard for an open communication channel. When a client opens a WebSocket connection it can send and receive data. Underlying message structure is decided by the developer[12].

CoAP and HTTP REST work similarly. CoAP being designed to mimic the HTTP REST system. CoAP being more lightweight and is thus more suitable for IoT purposes. [13] Periodic data information this could be an alternative but for my application that have sensors that push data more frequently using one of these technologies would mean sending more requests periodical instead of getting the information when it is published.

If I were to build this again I would aim for a purely MQTT system and remove CoAP. If I was dealing with more sparse update on the IoT data, then CoAP would be better suited due to the system not updating as often.

6.2 Results

In the figure 18 and figure 19. We can see that when the application is compiled for macOS it has overall the worst latency in terms of mean, max and standard deviation. It is expected for the result to be different depending on what applications request reach the broker first. But I also think that due to Flutter not being able to compile to ARM binaries for macOS and thus the macOS version of the application running on a compatibility layer could affect the results.

Looking at all the benchmark runs side by side the mean and max values does not change drastically except for the macOS case. Comparing figure 17a, figure 18a and figure 19a we can see that from benchmark run 1 to run 2 the number of request per second decrease from about 28 to about 23. But from run 2 to run 3 we only see a change from about 23 to 22. Similar changes can be seen in the other metrics as well, a bigger change between run 1 and run 2 but a smaller change between run 3 and 4.

To get a better understanding on how the system scales I would like to write a Go program that can generate x amount of Clients in separate threads to emulate more clients sending and receiving data. Due to the time limitation of this project I was not able to create this. Lastly as I mentioned in chapter 6.1 multi threading is possible on both the Translator and CoAP-server. It would be interesting to see how this changes the RTT.

6.3 Ethical and Societal Discussion

When talking about IoT as a whole I think it is important to talk about privacy concerns. A lot of IoT systems for consumers are built on cloud technologies for a fluid user experience, and one thing I keep in mind is if the cloud service the company uses for their IoT devices is free then the user data is the product.

With this project I am now able to implement IoT Protocols and if I want to avoid my data being collected I could just implement my own smart home. The problem here is we cannot expect the average consumer to do this. Thus, the consumer must be aware of what product collects data and how it is handled by the producer.

6.4 Future Work

First improvement I would make is to enable multithreading on both the Translator and the CoAP-server. This would theoretically improve the RTT due to both the Windows Server running the Translator and the Raspberry Pi running the CoAP-server having multiple cores to process requests on.

To better evaluate the system I would like to implement a program in Go that uses multiple threads to simulate more clients. With the multiple clients in the same program it would be easier to calculate metrics based on all clients RTT instead of each client on its own.

In the end I would ideily remove CoAP from the system completely because I do not think it contributes much to the system. For now the system runs on "stable" hardware with every component except the Flutter application running on hardwired Ethernet.

References

[1] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252. URL: https://rfc-editor.org/rfc/rfc7252.txt.

[2] Carsten Bormann. *CoAP*. Dec. 9, 2021. URL: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/ (visited on 01/04/2022).

[3] Google. *Flutter*. URL: https://flutter.dev (visited on 01/03/2022).

[4] Google. *Multi-Platform*. URL: https://flutter.dev/multi-platform (visited on 01/03/2022).

[5] plgd. *go-coap*. Dec. 16, 2021. URL: https://github.com/plgd-dev/go-coap (visited on 01/02/2022).

[6] Eclipse Foundation. *Eclipse Paho MQTT Go client*. Dec. 28, 2021. URL: https://github.com/eclipse/paho.mqtt.golang (visited on 01/02/2022).

[7] Steve Hamblett. *mqtt_client*. Jan. 2, 2022. URL: https://github.com/shamblett/mqtt_client (visited on 01/02/2022).

[8] Yo-An Lin. *goprocinfo*. Jan. 30, 2021. URL: https://github.com/c9s/goprocinfo (visited on 01/04/2022).

[9] Remi Rousselet. *Provider*. 2016. URL: https://github.com/rrousselGit/provider (visited on 01/06/2022).

[10] Kevin Moore. *stats*. Oct. 27, 2021. URL: https://github.com/kevmoo/stats (visited on 01/05/2022).

[11] Hoylen Sue. *dartmutex*. Mar. 15, 2021. URL: https://github.com/hoylen/dart-mutex (visited on 01/05/2022).

[12] Alexey Melnikov and Ian Fette. *The WebSocket Protocol*. RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://rfc-editor.org/rfc/rfc6455.txt.

[13] Michael Yuan. *Getting to know MQTT*. Dec. 9, 2021. URL: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/ (visited on 01/04/2022).

A Source Code