POLITECNICO
MILANO 1863

054323 - INTERNET OF THINGS

Challenge 2: Exercise

Student Names:

Kien Ninh Do (Erasmus) - 11075395
Dominik Leon Ucher (Erasmus) - 11073396

April 5, 2025

Contents

1 Exercise 1

1.1 CQ1

To answer the question we start by using Wireshark filtering to only get Confirmable PUT requests. This is done by the code below:

```
coap.code == 3 and coap.type == 0
```

We had to also make sure to only look at requests from the local server, which has an IP adress 127.0.0.1. After that we applied the filter and went through each packet row. In each packet there was a token and a response indication in the CoAP message. The token is used to find different responses, as CQ1 asked for different confirmable PUT requests. If the token is the same then the message is the same. The response indicated which number in the .pcap file has the response message. We then had to lookup the response packet and see if it was successful or unsuccessful. If it was successful it started with 2.XX. If it was unsuccessful it started with either 3.XX, 4.XX or 5.XX.

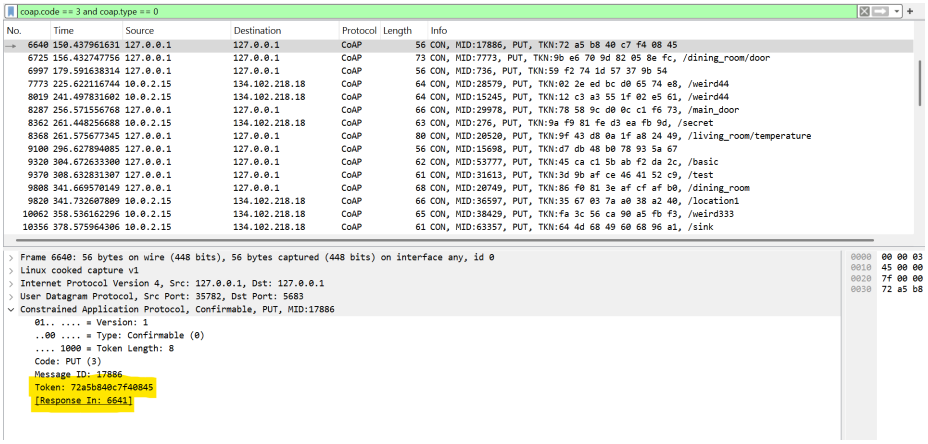

Figure 1: Analysis of unsuccessful Confirmable PUT responses

Below you can find each packet number with an unsuccessful response and their corresponding token.

Packet Number	Token
6640	72a5b840c7f40845
6725	9be6709d82058efc
6997	59f2741d57379b54
8287	78589cd00cc1f673
8368	9f43d80a1fa82449
9100	d7db48b078935a67
9320	45cac15babf2da2c
9370	3d9bafce464152c9
9808	86f0813eafcfafb0
10792	dd02502ce1ce9c96
11443	a9193d831f5f04b8
13156	90ba9c68c9110222
13637	24cd6982b5508e40
13675	570a2ecdb72f3f2b
13826	abe691f70fa19516
13840	2907148fb168abf8
13846	a298b93e59cd04a3
13850	3f2a9e16d0f49314
13883	392d043bec68bdf4
13893	d016db46b3510ca4
13895	ea642b05bffc5580
14048	4ae5b48a0b77de72

In total we found 22 unsuccessful requests.

1.2 CQ2

We started by finding coap.me IP address which was 134.102.218.18. We then went on to find all CoAP GET requests that were CON. Then we did the same with NON. The two codes can be seen below

```
coap.code == 1 and coap.type == 0 and ip.dst == 134.102.218.18
```

```
coap.code == 1 and coap.type == 1 and ip.dst == 134.102.218.18
```

We first went through the list and compared resources that were in both CON and NON. We then went through the shared resources and counted how many times the resource was used in CON and then how many times they were used in NON. The results are as follows.

Resources	CON	NON
/weird	1	3
/secret	1	1
/weird44	1	2
/sink	1	3
/weird33	3	1
/large	14	14
/4	2	1
/validate	1	1

In total we found $\underline{3}$ unique CON and NON requests.

1.3 CQ3

We first start off by identifying HiveMQ IP addresses and their aliases, as this is the type of information given in the MQTT packet. We do that by using this code below:

```
dns.flags.response == 1
```

Next, we navigate to `Edit` → `Find Packet`, search for `"hivemq"` as a string in packet details, and examine the DNS answers. We find that HiveMQ has the following IP address aliases:

- `18.192.151.104`
- `35.158.43.69`
- `35.158.34.213`

Furthermore we need to find MQTT cleint that subscribe using a multi-level wildcard. MQTT subscribe message type is 8 is Wireshark. And cleints that subscribe using multi-level wildcard are the one that end in (`#`). Therefore we used the code below to filter the rows:

```
mqtt.msgtype == 8 and (ip.dst == 18.192.151.104 or ip.dst == 35.158.43.69 or
            ip.dst == 35.158.34.213) and mqtt.topic contains "#"
```

Then we get 6 rows and have to find which clients are the same and which are unique. We find this by using the IPv4 identification number in the packet. And as you can see below there are only 4 total different clients.

IPv4 Identification
0x61f7 (25079)
0xaf2b (44843)
0xd43a (54330)
0xaf33 (44851)
0x0906 (2310)
0xaf45 (44869)

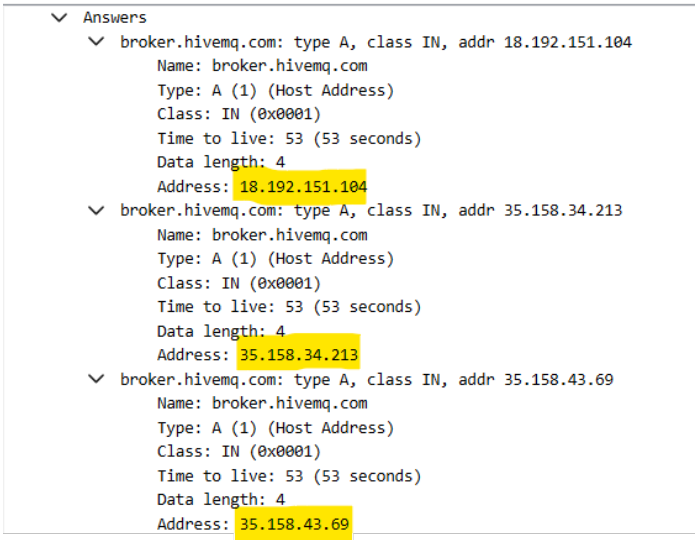

Figure 2: HiveMQ IP Aliases

In total we found 4 different clients

1.4 CQ4

We start by using Wireshark filtering and finding only MQTT packets that have a will topic that contains the string "university". We receive only one packet that fulfills our constrains. We then inspect the packet close and see that is in fact does contain "university" on the first level of the will topic.

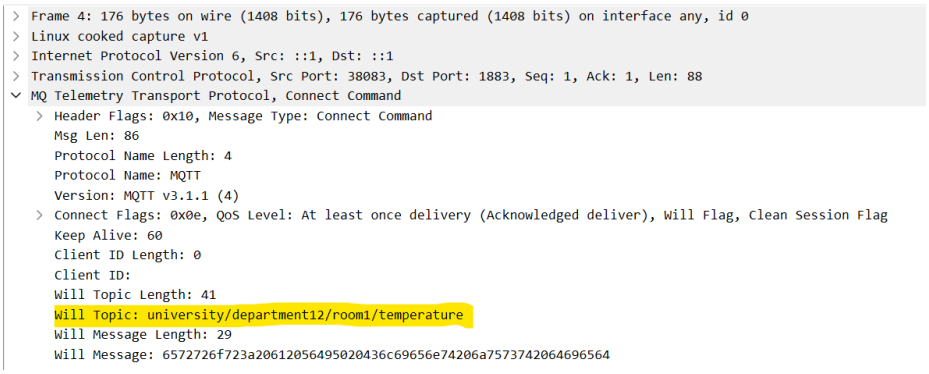

Figure 3: Will Topic containing "University"

In total we found 1 MQTT client

1.5 CQ5

We start by using Wireshark filtering and find only MQTT connect messages that contain a will message.

```
mqtt.msgtype == 1 and mqtt.willmsg
```

We find 4 connect messages with a will message. Their row packet number is seen below:

Last Will Messages
4 - Client ::1
192 - Client 10.0.2.15
352 - Client 10.0.2.15
557 - Cleint 10.0.2.15

We then have to do a trace back and see if their subscription contains any wildcards. Only the first Last Will message has subscriptions without wild cards. The other 3 Last Will messages come from subscriptions that contain a wild card.

Although the first message has in total 3 subscriptions that do not contain a wild card, even though it was just 1 message. Therefore we get the following answer.

In total we found $\underline{\underline{3}}$ relevant client

1.6 CQ6

We use Wireshark filtering in this section as well. We split this question into 4 different filters that we put together to get the relevant MQTT packets. We start off by finding MQTT publish type messages:

```
mqtt.msgtype == 3
```

Then we find messages directed to Mosquitto IP address

```
ip.dst == 5.196.78.28
```

Then we find the MQTT packets that contain a MQTT Retain

```
mqtt.retain == 1
```

And we finish with finding MQTT packets that have QoS at most one

```
mqtt.qos == 0
```

We put these 4 constraints together to get the following filter code on Wireshark:

```
mqtt.msgtype == 3 and mqtt.qos == 0 and mqtt.retain == 1 and ip.dst == 5.196.78.28
```

We follow this up by exporting the selected packets into a CSV file. We then use Pandas in Python to count how many rows there are.

Python Script

```
import pandas as pd
import numpy as np

# Load the CSV file
df = pd.read_csv('Files/mosquitto.csv')

len(df)
```

In total we found 208 rows

1.7 CQ7

We started off by checking if there are any UDP packets going through port 1885, where we got no packets. Then we checked if there are any MQTT-SN packets in the .pcap file, where there were also none. Then we checked in there was any packets or any traffic at all at port 1885, where we also got no response.

In total we found 0 messages.

2 Exercise 2

2.1 Calculations

Transmission from sensor:

$$\frac{24 \times 60}{5} = 288 \text{ times} \tag{1}$$

Valve computation of average temperature:

$$24 \times 2 \times 2.4 \text{ mJ} = 115.2 \text{ mJ} \tag{2}$$

2.2 EQ1

2.2.1 a - CoAP

CoAP (Constrained Application Protocol) is designed to operate over UDP, which does not originally include message acknowledgment. While it is possible to implement acknowledgment mechanisms on top of UDP, doing so does not contribute significantly to energy efficiency.

In this system, the sensor is tasked with sending data, but a challenge arises when using PUT requests: the valve must constantly poll the sensor, consuming continuous energy. CoAP's Observe functionality addresses this issue by allowing the valve to subscribe to sensor data updates. Once a GET request is issued, the valve will only receive GET responses periodically, such as every five minutes, reducing the need for constant polling. This significantly improves the energy efficiency of the communication process.

By using GET + Observe, the system reduces unnecessary communication, allowing the valve to be energy-efficient while still receiving updates without needing to send repeated requests. This approach greatly reduces the overall energy consumption compared to traditional polling methods.

$$\textbf{GET request: sensor} \rightarrow \textbf{valve (TX):}\ 50\,\text{nJ/b} \cdot 60\,\text{B} \cdot 8 = 0.0240\,\text{mJ} \tag{3}$$

$$\textbf{GET request: valve} \rightarrow \textbf{sensor (RX):}\ 58\,\text{nJ/b} \cdot 60\,\text{B} \cdot 8 = 0.0278\,\text{mJ} \tag{4}$$

$$\textbf{GET response: valve} \rightarrow \textbf{sensor (TX):}\ 50\,\text{nJ/b} \cdot 55\,\text{B} \cdot 8 = 0.0220\,\text{mJ} \tag{5}$$

$$\textbf{GET response: sensor} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 55\,\text{B} \cdot 8 = 0.0255\,\text{mJ} \tag{6}$$

Thanks to Observe, the GET request is only sent once, while GET responses are received every five minutes. The total energy consumption can therefore be computed as:

$$E_{tot} = 288 \cdot E_{GET-response} + E_{GET-request} + 115.2 \tag{7}$$

$$E_{tot} = 288 \times 0.0475\,\text{mJ} + 0.0518\,\text{mJ} + 115.2\,\text{mJ} = \underline{\underline{128.93\,\text{mJ}}} \tag{8}$$

2.2.2 b - MQTT

MQTT follows a publisher-subscriber model where both the sensor and valve connect to a broker. The broker manages the message distribution, and both the sensor and valve initiate the connection by sending CONNECT requests. When QoS 0 is used, there is no acknowledgment required from the broker, meaning no PUBACK messages are sent. This reduces the communication overhead and energy consumption.

After the connection, the valve subscribes to the broker. When all this is done, the sensor can publish to the broker that publishes further to the valve 288 times in the span of 24 hours.

$$\textbf{CONNECT: valve} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 54\,\text{B} \cdot 8 = 0.0216\,\text{mJ} \tag{9}$$

$$\textbf{CONNECT: sensor} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 54\,\text{B} \cdot 8 = 0.0216\,\text{mJ} \tag{10}$$

$$\textbf{CONNACK: broker} \rightarrow \textbf{sensor (RX):}\ 58\,\text{nJ/b} \cdot 47\,\text{B} \cdot 8 = 0.0218\,\text{mJ} \tag{11}$$

$$\textbf{CONNACK: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 47\,\text{B} \cdot 8 = 0.0218\,\text{mJ} \tag{12}$$

$$\textbf{SUBSCRIBE: valve} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 58\,\text{B} \cdot 8 = 0.0232\,\text{mJ} \tag{13}$$

$$\textbf{SUBACK: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 52\,\text{B} \cdot 8 = 0.0241\,\text{mJ} \tag{14}$$

$$\textbf{PUBLISH: sensor} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 68\,\text{B} \cdot 8 = 0.0272\,\text{mJ} \tag{15}$$

$$\textbf{PUBLISH: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 68\,\text{B} \cdot 8 = 0.0316\,\text{mJ} \tag{16}$$

$$E_{tot} = 2 \cdot (0.0216 + 0.0218) + (0.0232 + 0.0241) + 288 \cdot (0.0272 + 0.0316) + 115.2 = \underline{\underline{132.27\,\text{mJ}}} \tag{17}$$

2.3 EQ2

A solution to decrease the energy consumption when passing the Raspberry PI as a broker, is by using MQTT-SN instead of standard MQTT. MQTT-SN works very similar to the standard MQTT with a broker and a publish-subscribe-logic, but have several differences such as UDP-communication instead of TCP, more QoS-levels (-1 in addition) and the possibility to reduce the topic length (topic ID) to 2 bytes when using QoS -1 [1]. The cited thesis does also suggest that QoS -1 does not require that the publisher (sensor) connects, something that also will reduce the total energy.

In the original task, the topic length is 10 bytes, and have been calculated into the table. Thus, we can reduce all the message lengths by 8 bytes. The calculations will then be:

$$\textbf{CONNECT: valve} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 46\,\text{B} \cdot 8 = 0.0184\,\text{mJ} \tag{18}$$

$$\textbf{CONNACK: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 39\,\text{B} \cdot 8 = 0.0181\,\text{mJ} \tag{19}$$

$$\textbf{SUBSCRIBE: valve} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 50\,\text{B} \cdot 8 = 0.0200\,\text{mJ} \tag{20}$$

$$\textbf{SUBACK: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 44\,\text{B} \cdot 8 = 0.02416\,\text{mJ} \tag{21}$$

$$\textbf{PUBLISH: sensor} \rightarrow \textbf{broker (TX):}\ 50\,\text{nJ/b} \cdot 60\,\text{B} \cdot 8 = 0.0240\,\text{mJ} \tag{22}$$

$$\textbf{PUBLISH: broker} \rightarrow \textbf{valve (RX):}\ 58\,\text{nJ/b} \cdot 60\,\text{B} \cdot 8 = 0.0278\,\text{mJ} \tag{23}$$

$$E_{tot} = (0.0184 + 0.0181) + (0.0200 + 0.02416) + 288 \cdot (0.0240 + 0.0278) + 115.2 = \underline{\underline{130.20\,\text{mJ}}} \tag{24}$$

References

[1] Federico Ruzzier. “Adaptive Quality of Service for MQTT-SN”. Pages 19-22, Accessed: 2025-03-28. MA thesis. Politecnico di Milano, 2020-2021. URL: https://www.politesi.polimi.it/retrieve/94e0760e-3c66-4d92-abde-dc841732f716/Adaptable_QoS.pdf.