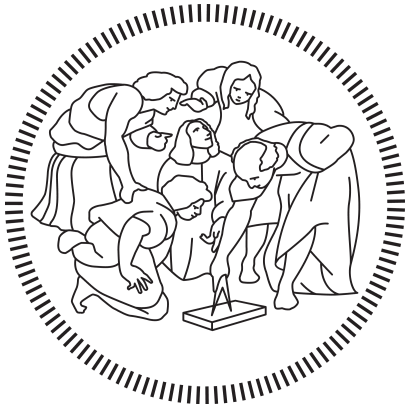
AY 2023/2024



Politecnico di Milano

Internet of Things Challenge - 2

Rishabh Tiwari - rishabh.tiwari@mail.polimi.it - 10987397
Marcos V. Firmino P. - marcosvinicius.firmino@mail.polimi.it - 10914211
Alexander Stephan - alexander.stephan@mail.polimi.it - 10932707

Professor
Redondi Alessandro Enrico Cesare

Version 1.1
April 3, 2024

1 Introduction

In our project for IoT Challenge 2, we dove into a dataset full of network conversations to uncover how devices are chatting using CoAP and MQTT—two languages of the Internet of Things. Armed with tools like Wireshark and some coding magic, we shifted through the digital back-and-forth to answer a set of questions about these conversations. We looked for patterns, like how often devices ask for updates or if any messages got lost along the way. All the technical jargon aside, our goal was to get the gist of how these IoT devices talk to each other and keep their connections smooth and snappy.

2 Traffic Analysis Exercise

Disclaimer: we also provide Python scripts for most exercises on Webeep.

1. **Analysis of CoAP Clients and GET Requests**

   (a) *How many different CoAP clients sent a GET request to a temperature resource (.../temperature)?*

   Identification of CoAP GET Requests:

   ```
   coap.code == 1 && coap.opt.uri_path contains "temperature"
   ```

   This filter is used to search for GET requests to resources containing "temperature". In fact, they all match `.../temperature`, as requested. Clients are identified by source port, and only one duplicate port (48049) was observed. Thus, we count **8** unique clients that sent a total of 9 requests.

   (b) *For each of the clients found in 1a), write the MID of the longest CoAP response (any response) received by the client*

   ```
   coap && coap.code >= 65 && coap.code <= 165 &&
   (udp.dstport == 55898 || udp.dstport == 33677 || udp.dstport ==
      51812 ||
    udp.dstport == 48049 || udp.dstport == 52276 || udp.dstport ==
       48645 ||
    udp.dstport == 52247 || udp.dstport == 41264)
   ```

   We filter by valid response codes and the ports previously observed in 1a). After filtering and sorting by length, the message ID **10589** was found to be the longest CoAP response.

2. **CoAP POST Requests to `coap.me` Server**

   (a) *How many CoAP POST requests directed to the "coap.me" server did NOT produce a successful result?*

   ```
   coap && ip.dst == 134.102.218.18 && coap.code == 2
   ```

   The IP address 134.102.218.18 was obtained inspecting the respective DNS reply in the traffic dump. We can't use utilities like `dig`, as the server IP might has changed in the meantime. Finally, tokens of POST requests to this IP were retrieved via Pyshark.

   In a second step, we go through the responses and register all valid responses with a matching token. (We assume that result means response here. We also assume that a NON request trivially matches this criteria, as there should be no response at all, we do not exclude them.) We subtract the no. of valid responses from the number of total requests. The number of POST requests without successful results was **17**, identified by filtering response codes in the range from 65 to 69 (see slides in Lab 3).

   (b) *How many requests from 2a) are directed to a "weird" resource? (resources like /weirdXX)?*

   We originally assumed that "weirdXX" means that there have to exactly two characters after the word "weird". However, this does not seem to the case according to Webeep. Therefore, we look for any resource containing "weird". We wrote a Python script that loops over the the packets with the following filter and checks whether they match the tokens from 2a).

```
coap && ip.dst == 134.102.218.18 && coap.code == 2
```

Then, we check whether the last segment in the URL starts with "weird". We find **7** requests that match the criteria.

3. **MQTT Publish Messages with QoS=2**

   (a) *How many MQTT Publish messages with qos=2 are RECEIVED by the clients running in the machine capturing the traffic?* As per common convention, we assume that the IP of the current machine is `127.0.0.1`.

   ```
   mqtt.msgtype == 3 && mqtt.qos == 2 && ip.dst == 127.0.0.1
   ```

   The query resulted in **30** publish messages received by the clients running on the local machine.

   (b) *How many clients are involved in the messages found in 3a)?* Again, we inspect the source ports. We observe 3 different source ports, consequently there are **3** clients.

   (c) Message IDs of Subscribe Requests: The MQTT Message IDs that enabled clients to receive these messages were **51523**, **44887**, **32965**, **51524**, and **51525**.

4. **MQTT Subscriptions with Wildcards**

   (a) *How many MQTT clients sent a subscribe message to a public broker using at least one wildcard?*

   Number of clients using wildcards in subscribe messages:

   ```
   mqtt.msgtype == 8 && (mqtt.topic contains "+" || mqtt.topic
       contains "#") && ip.src != 127.0.0.1
   ```

   By inspecting the source ports, we deduced: there were **4** clients sending subscribe messages with at least one wildcard.

   (b) *Considering clients found in 4a), how many of them WOULD receive a publish message directed to the topic "metaverse/facility4/area0/light“*

   Out of the clients using wildcards, **2** would receive a publish message directed to the topic `metaverse/facility4/area0/light`, because they subscribed to `metaverse/+/area0/light` and `metaverse/facility4/+/light`.

5. *How many MQTT ACK messages in total are received by clients who connected to brokers specifying a client identifier shorter than 15 bytes and using MQTT version 3.1.1?*

   The task involved extracting a CSV of messages with an MQTT client identifier length less than 15 and using MQTT version 3.1.1 (`mqtt.ver == 4`).

   ```
   mqtt.clientid_len < 15 && mqtt.ver == 4
   ```

   We only see two connect commands from two different clients. Next, we scan for all types of ACK messages using the observed clients as destination IPs and ports. (We also observe the correct source 91.121.93.94.)

   ```
   ((ip.dst == 10.0.2.15  && tcp.dstport == 43949) || (ip.dst ==
       127.0.0.1 && tcp.dstport == 4887 ))
   && (mqtt.msgtype == 4 || mqtt.msgtype == 5 || mqtt.msgtype == 6 ||
       mqtt.msgtype == 7 || mqtt.msgtype == 9)
   ```

   Overall, we observe **18** such requests.

6. **Analysis of MQTT Subscribe Requests**

   (a) *How many MQTT subscribe requests with message ID=1 are directed to the HiveMQ broker?*

   Firstly, we discovered the HiveMQ IP by analyzing the DNS response for "broker.hivemq.com.". The HiveMQ IPs are the following:

   - 3.65.168.153
   - 3.66.35.116

With that information, we wrote the following command:

```
mqtt.msgtype == 8 && mqtt.msgid == 1 && (ip.dst == 3.65.168.153
    || ip.dst == 3.66.35.116)
```

The first part takes subscribe messages (type equals to 8); the second part looks for the correct message ID; the third filters for the correct destination IP.

The total number of subscribe messages sent were **3**, from different clients.

(b) *How many publish messages are received by the clients thanks to the subscribe requests found in 6a)* First we collect the corresponding IP, the specific port and topic of subscription of the 3 clients.

Let's enumerate them:

- Client 1:
  - IP: 10.0.2.15
  - Port: 38887
  - Topic: university/department2/floor5
- Client 2:
  - IP: 10.0.2.15
  - Port: 36707
  - Topic: house/kcbplh/section2
- Client 3:
  - IP: 10.0.2.15
  - Port: 59385
  - Topic: hospital/kcbplh/#

Now, we can filter for incoming messages from the HiveMQ directed to each specific client regarding this specific topic:

```
mqtt.msgtype == 3 && (ip.src == 3.65.168.153 || ip.src ==
    3.66.35.116 ) && (ip.dst == 10.0.2.15  && tcp.port ==
    CLIENT_PORT) && (mqtt.topic == SUBCRIPTION_TOPIC)
```

The first part filters for publish messages (type 3); the second part filters for the specific source IP (HiveMQ); The third part specifies the client (IP and Port combination); The fourth and final part filters for the specific topic of subscription.

Doing so for each client:

- Client 1
  - Code:

    ```
    mqtt.msgtype == 3 && (ip.src == 3.65.168.153 || ip.src ==
        3.66.35.116 ) && (ip.dst == 10.0.2.15  && tcp.port ==
       38887) && (mqtt.topic == "university/department2/floor5
       ")
    ```

  - Result: 0 tuples
- Client 2
  - Code:

    ```
    mqtt.msgtype == 3 && (ip.src == 3.65.168.153 || ip.src ==
        3.66.35.116 ) && (ip.dst == 10.0.2.15 && tcp.port ==
       36707) && (mqtt.topic == " house/kcbplh/section2")
    ```

  - Result: 0 tuples
- Client 3
  - Code:

    ```
    mqtt.msgtype == 3 && (ip.src == 3.65.168.153 || ip.src ==
        3.66.35.116 ) && (ip.dst == 10.0.2.15 && tcp.port ==
       59385) && (mqtt.topic contains "hospital/kcbplh")
    ```

  - Result: 0 tuples

With the results achieved, there were **0** publish messages received due to the subscription.

7. **MQTT-SN Publish Messages Analysis**

   (a) *How many MQTT-SN (on port 1885) publish messages sent after the hour 3.59PM (Milan Time) are directed to topic 6?*

   ```
   udp.dstport==1885 && mqttsn.topic.id == 6 && mqttsn.msg.type ==
       0x0c && mqttsn && !icmp
   ```

   The number of MQTT-SN publish messages sent after 3.59 PM (Milan Time) to topic 6 was **3**. We sorted by the UTC time and added +1.

   (b) *Explain possible reasons why messages in 7a) are not handled by the server*

   These messages were not handled by the server, possibly due to all publish messages being accompanied by an "ICMP 82 destination unreachable (port unreachable)" packet. This could indicate that the server is not listening on that port or has not even been started. Alternatively, there could be an issue with the network, so requests do not reach the targeted server.

3 Conclusion

After looking closely at how IoT devices talk to each other, we've learned a lot about their conversations. It turns out these devices are pretty chatty—they're always asking each other for updates. But like any chat, sometimes things don't go smoothly, and messages get lost or blocked. By figuring out the answers to our questions, we've also learned that even though these device networks are strong, they need careful watching and smart management. That way, we can keep the lines of communication open and make sure our devices can talk to each other without any hiccups.