MQTT and CoAP implementations for Arduino

Alice Valentini*
* DISI, University of Bologna, Italy

alice.valentini7@studio.unibo.it

***Abstract*—Wireless Sensor Networks are becoming more and more employed. Since they are typically composed by a set of nodes constrained from the computational, storage and energetic point of view, new bandwidth-efficient and energy-efficient protocols have been implemented over the years. MQTT (Message Queue Telemetry Transport) and CoAP (Constrained Application Protocol) are two of the most popular protocols used in these scenarios. These two protocols have different features, while MQTT follows the publish-subscribe communication pattern, CoAP follows the Representational State Transfer (REST) architectural style and it is more similar to HTTP. In this report we will examine how this protocols can be implemented on an Arduino microcontroller board equipped with an ethernet shield and different sensors, using existing libraries and software.**

I. Introduction

Wireless sensor networks (WSNs) have been gaining increasing attention, both from commercial and technical point of views, because of their potential of enabling new and attractive solutions in areas such as industrial automation, asset management, environmental monitoring, transportation business, etc. They are typically composed by a set of sensor nodes and gateways equipped with limited amount of storage, energetical and processing capabilities. As a result, bandwidth-efficient and energy-efficient protocols for sensor data transmission are required in this scenarios, as traditional protocols resulted not suitable. MQTT (Message Queue Telemetry Transport) and CoAP (Constrained Application Protocol) have been proposed to specifically address the difficult requirements of real-world WSN deployment scenarios and now are two of the most used protocols designed for WSNs and machine-to-machine (M2M) communications.

MQTT is an open protocol designed by IBM, it uses a *topic-based publish-subscribe* communication pattern. In this architecture, a client needing data (known as subscriber) registers its interests with a server (also known as broker), the client producing data (known as publisher) sends it to a server which forwards it to the subscribers. Communications use TCP/IP protocol with different levels of Quality of Service (QoS).
One of the advantages of this approach is that sensor nodes need not know the identities of clients that are interested in their data and conversely, clients need not know the identities of sensor nodes generating the sensor data. This decoupling enables the architecture to be highly scalable [14]. MQTT brokers may require username and password authentication from clients to connect. To ensure privacy, the TCP connection may be encrypted with SSL/TLS.

CoAP, on the other hand, is a service layer protocol and it is very similar to HTTP. It follows Representational State Transfer (REST) architectural style and instead of TCP, uses UDP for transmitting data and Universal Resource Identifer (URI) instead of topics. CoAP does not provide different QoS mechanisms but can use ”confirmable” messages for reliability, this kind of messages require to be acknowledged. Because CoAP is built on top of UDP not TCP, SSL/TLS are not available to provide security, instead, DTLS (Datagram Transport Layer Security) can provide the same assurances as TLS but for transfers of data over UDP. The core of the protocol is specified in RFC 7252 [13].

II. Related works

MQTT and CoAP are certainly the most employed M2M protocols and many implementations for different devices have been developed over the years. MQTT has been employed in Facebook Messenger app since, according to Facebook engineers, its features permit to maintain a persistent connection to the servers and sending and receiving messages within hundred of milliseconds without draining too much the battery life [1].
MQTT is also employed in IBM MQ, a messaging middleware [4], in IBM IoT MessageSight and in many IoT solutions [9]. There are also many open source libraries and message brokers written in several languages, in the next chapter we will focus on *mosquitto*, a MQTT broker written in C and *PubSubClient*, a library for Arduino [8].
In addition, several websites offer the possibility to use a MQTT online broker by subscribing to a plan if users can't implement a broker by their own.

CoAP has not been employed widely like MQTT but libraries and implementations in many programming languages can still be found [2]. In the next chapter we will see *libcoap*, a C implementation of CoAP that can be used both on constrained devices and on a larger POSIX system, then *microcoap* [6] and *CoAP-simple-library* [3] implementations for Arduino. One of the most popular CoAP implementations is *Copper*, a Mozilla Firefox extension which can be used to send and receive CoAP messages using the browser.

III. Architecture

Here are shown and discussed the architecture's main aspects of both the implementations. The devices used are a Linux laptop, an Arduino UNO board equipped with an ethernet shield and different sensors and actuators and an Android smartphone. In both cases devices communicate with each other through a LAN.

A. MQTT

MQTT uses a *topic-based publish-subscribe* architecture. This means that when a client publishes a message *M* to a particular topic *T*, then all the clients subscribed to the topic *T* will receive the message *M*. MQTT is designed in such a way that its implementation on the clients side is very simple, indeed, two MQTT devices do not interact with each other, but via a broker, which is primarily responsible for receiving all messages, filtering them, decide who is interested in it and then sending the message to all subscribed clients.
The communications use TCP/IP protocol on port 1883 or 8883 if SSL is in use.
MQTT provides 3 Quality of Service (QoS) levels: QoS level 0 means that a message is delivered at most once and no acknowledgement of reception is required. QoS level 1 means that every message is delivered at least once and confirmation of message reception is required. In QoS level 2, a four-way handshake mechanism is used for the delivery of a message exactly once [14].

Topics in MQTT are specified with a UTF-8 string. A topic can have more topic levels delimited by a slash (/) and are case sensitive. A client can subscribe to more topics at once using wilcards: "+" is a single level wildcard and can be used for substitute one topic level, "#" is a multi level wildcard and covers an arbitrary number of topic levels, it is required that the multi level wildcard is always the last character in the topic and it is preceded by a forward slash.

MQTT defines five different methods to indicate the desired action to be performed on the identified resource:

- Connect
- Disconnect
- Publish
- Subscribe
- Unsubscribe

MQTT packets consist in a fixed 2 byte header, a variable header (optional) and a payload (optional).

In our case, as can be seen in Fig. 1, the broker is installed on the laptop and the designated clients are Arduino linked to two different temperature sensors, DHT11 and Dallas DS18B20, which acts as the publisher while the subscribers are a client application installed on the same laptop and an app installed on the Android smartphone. The data is published at a regular interval.
The topics provided are three: *sensors/dht11/temp*, *sensors/dht11/humidity* and *sensors/dallas/temp*, as DHT11 sensors provides info about both temperature and humidity while Dallas DS18B20 only about temperature. Wildcards can be used in this case by, for example, gain temperature data from both sensors by using the wildcard "+": *sensors/+/temp*.

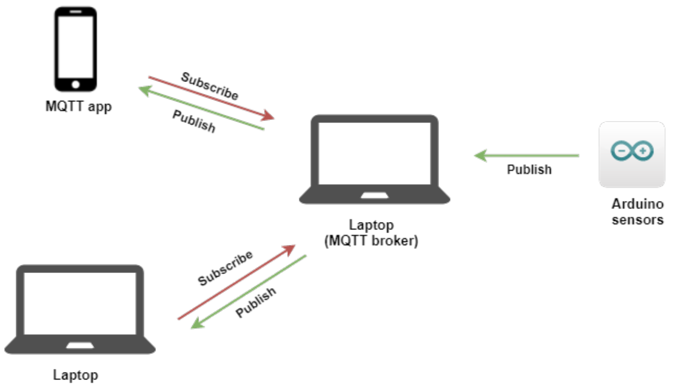


Fig. 1. MQTT implementation's architecture

B. CoAP

CoAP follows the REST architectural style and can be assimilated to the client-server one. As mentioned before, communications use UDP protocol on port 5683.
In our case, Arduino acts as the server providing data while the client (the laptop) can request them. The sensor used was only DHT11 but a multicolor led as actuator was added here. The client, as with HTTP, can make requests with the methods GET, POST, PUT and DELETE which are identified by a code in the request's packet. The resources located in Arduino are identified with a URI.

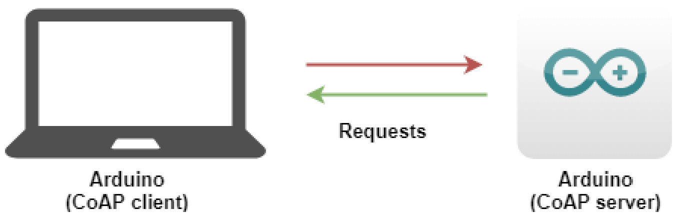


Fig. 2. CoAP implementation's architecture

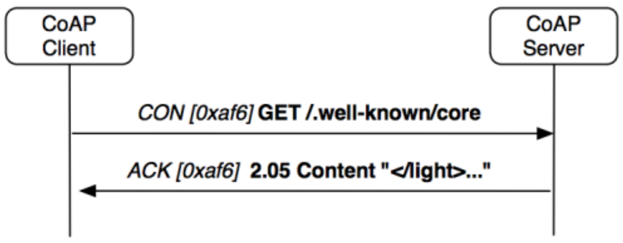

CoAP
Client
CoAP
Server
CON [0xaf6] GET /.well-known/core
ACK [0xaf6] 2.05 Content "</light>..."

Fig. 3. CoAP event diagram

There are two types of messages: requests and replies. Each message contains a Message ID used to detect duplicates and for optional reliability. Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (ACK) with the same Message ID, this is called a piggybacked response. A message that does not require reliable transmission can be sent as a Non-confirmable message (NON). These are not acknowledged, but still have a Message ID for duplicate detection. Multicast is supported by CoAP using multicast IP destination addresses.
Packets are much smaller than the HTTP ones, their structure is shown in Fig. 4. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. The payload usually contains the information requested or supplied by the client [13].

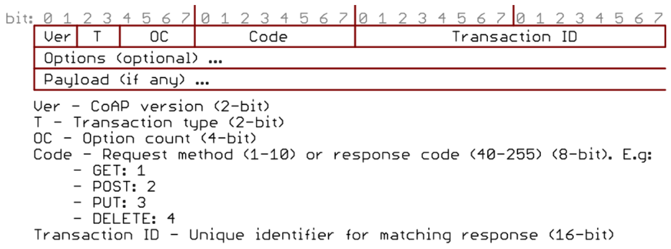

bit: 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7 | 0 1 2 3 4 5 6 7
Ver
T
OC
Code
Transaction ID
Options (optional) ...
Payload (if any) ...
Ver - CoAP version (2-bit)
T - Transaction type (2-bit)
OC - Option count (4-bit)
Code - Request method (1-10) or response code (40-255) (8-bit). E.g:
- GET: 1
- POST: 2
- PUT: 3
- DELETE: 4
Transaction ID - Unique identifier for matching response (16-bit)

Fig. 4. CoAP packet structure

The CoAP standard also provides a special URI for server's service discovery at <*./well-known/core*>, it provides the list of links in the application/link-format hosted by the server and it is returned along with the content type attribute as defined in RFC 6690. It allows clients to discover what resources are provided and what media types they are.

IV. Implementation

In this section the technologies used for implementing the systems are shown and discussed.

A. MQTT

In this system Arduino (the publisher) publishes temperature data collected from the two sensors DHT11 and Dallas DS18B20 at a regular interval, the broker installed on the laptop collects the data and sends it to another process installed in the laptop and the smartphone (the two subscribers).

1) Mosquitto: The broker used is *Mosquitto* [10], an open source message broker installed on the laptop. This software didn't need any additional configuration after the installation. The only thing that it is required is its persistent running in background. The status can be checked using this command from the linux command line:

```
$ sudo service mosquitto status
```

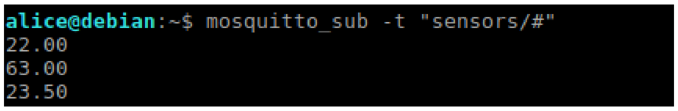
We will see later that this software can also be used as a client for publishing or subscribing to specific topics.

2) PubSubClient: PubSubClient [11] library was used for building the Arduino client as it offers functions for publishing data and manage the connection to the broker. This library can be easily installed in the Arduino IDE by searching it in the libraries section. We will see that with a few function calls it is possible to build a simple MQTT client for Arduino.
Firstly, the board must gain an IP address via the ethernet shield, after that, it can start to attempt the connection to the broker, knowing its IP address. Once the connection is established with the broker, the board can start to publish data.
As no values other than the server address are used in the function for connecting to the broker, the QoS value used is the default (0, no acknowledgement required).

3) The subscribers: As explained before, this system provides two clients: the laptop and the smartphone. On the laptop was used *Moquitto* again as it can act also as a client. Using the terminal it is necessary to send a simple command for subscribing to a given topic:

```
$ mosquitto_sub -t "sensors/*"
```

Once sent, the updates received are printed in the terminal as we can see from Fig. 5.



```
alice@debian:~$ mosquitto_sub -t "sensors/#"
22.00
63.00
23.50
```

Fig. 5. Subscribing with Mosquitto

On the smartphone the MQTT client used was *MQTT Dashboard* app available on Google Play Store. After setting the IP address and the port of the broker, the user can subscribe to any topic. Data can also be shown with the unit of measure next to it. The result is shown in Fig. 6.

B. CoAP

In this system a client, the laptop, makes a request to the CoAP server (Arduino) for gaining information about a multicolor led and a DHT11 temperature sensor.

Unfortunately, there aren't many implementations and libraries available on the Internet for building a CoAP server on Arduino, and the few ones, aren't really well documented. Two different implementations were tested: *CoAP-simple-library* [3] and *Microcoap* [6].

1) CoAP-simple-library: it is, indeed, very simple and can also be installed from the Arduino IDE, while on the GitHub page a few examples can be found. The project is composed

by three files: two CoAP library files (not necessary if the library has been installed from the IDE) and the Arduino sketch file. Firstly, it is necessary to set the endpoints in the setup section of the Arduino sketch file, that is, the URIs we want to configure.
In this implementation two URIs were provided: *coap://(address)/led* and *coap://(address)/dht11*. The first one accepts accepts GET and PUT requests: with a GET request the client gains information about the color the led is showing, while with a PUT request, the client can change it including the new value in the request's payload. In the second endpoint, only GET requests are admitted and temperature data is returned.

The configuration of *</.well-known/core>* endpoint has been implemented manually as it wasn't provided nor in the examples nor through functions.
Once the endpoints are set, it's needed to define their respective callback functions. They only check the request type (GET or POST) and include the value requested in the response's payload. The type check is ”manual” by getting the request code in the source packet.
The *coap.loop()* function in the loop section is responsible for sorting and manage the CoAP packets in input.

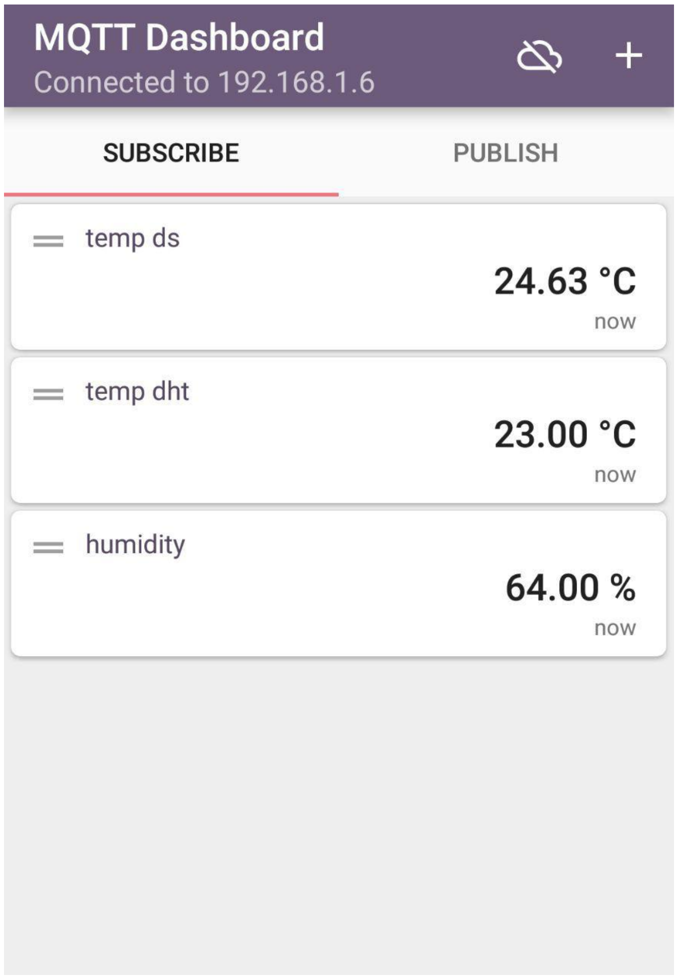


Fig. 6. MQTT Dashboard client

This library was very simple to use but supplies very few functions. Documentation is practically non existent.

2) Microcoap: The other implementation was built following a *microcoap* fork on GitHub [7] since the main version is not compatible with Arduino UNO (requires too much memory).
The documentation provided here is slightly better than *CoAP-simple-sibrary* as this repository is more popular.

This project is composed by four files: two CoAP library files, one C file for configuring the endpoints and the Arduino sketch file. The library files contain functions for manage CoAP packets and the connection to the clients, the Arduino sketch program only reads and handles the packets in input. So, only the endpoints file was changed by adding the path *coap://(address)/light* which accepts both GET and PUT requests, its functioning is similar to the one of the previous project. In the response request is also possible to add the content type contained in the payload.

The *coap://(address)/.well-known/core* endpoint was already implemented in the example found on the GitHub page and the response is built automatically at setup.
The reason why the DHT11 temperature sensor was not included in this project is because the DHT11 library is written in C++ and can't be included in the endpoints file written in C.

3) Libcoap: In both the implementations, the library used for making the requests from the laptop is *libcoap* [5]. Once installed, it doesn't need any further configuration. Requests can be made from the terminal using these commands:

```
$ coap-client -m get coap://(arduino
   address)/light
$ coap-client -m put -e 2 coap://(arduino
    address)/light
```

For sending a non-confirmable message is necessary to add the flag *-N* to the command.

V. Performance evaluation

With the current equipment it was not possible to evaluate accurately the performances of both the systems but researches has been made to find out how these two protocols behave in different network scenarios. In detail, D. Thanganvel et al. [14] compared the performances of the two protocols via a common middleware using a Wide Are Network emulator. The performance of the protocols was measured in terms of delay and total data (bytes) transferred per message with different rates of packet loss. The results showed that MQTT messages experienced lower delays than CoAP for lower packet loss and higher delays than CoAP for higher packet loss, moreover, when the message size is small and the loss rate is equal to or less than 25%, CoAP generates less extra

traffic than MQTT to ensure reliable transmission. Also, as the message size increases, the probability that UDP loses the message is higher than TCP, which causes CoAP to retransmit the whole message more often than MQTT.
These differences resides mainly in the characteristics of TCP and UDP protocols.

It was made another similar research along with other M2M protocols [12] where latency and bandwidth were considered. Even here the differences between the two protocols are due to the features of UDP and TCP protocols. MQTT consumes more bandwidth due its packet retransmission mechanism to guarantee packet delivery.

VI. Conclusion

In this report we have seen possible implementations for these protocols on an Arduino board.
MQTT libraries and softwares have good documentation, many examples and a bigger community behind, so developing the project was easy and enjoyable. CoAP libraries instead, have very poorly commented examples and very poor documentation which lead to more difficulties in developing the projects.

The two protocols have different features and could be use in different scenarios. MQTT is more scalable but needs a broker, while CoAP is decentrilized. CoAP generates less bandwidth but performs worse when the payload is big and is less reliable when network conditions are bad, so it's preferable to use it when reliability is not a strong requirement. MQTT doesn't offer service discovery and there isn't indication on the data type received, also it is not possible to change or delete data residing on the client.
This experiment has limitations regarding the evaluation of the performances as it was only possible to rely on other researches.

Future improvements could include implementing a new CoAP library taking the existing ones as a reference, adding more functions and providing a clearer documentation. Also addition to the Arduino implementations by including other sensors and methods could be made.

References

[1] Building facebook messenger. https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920. Visited: 2017-09-07.

[2] Coap implementations. http://coap.technology/impls.html. Visited: 2017-09-07.

[3] Coap-simple-library. https://github.com/hirotakaster/CoAP-simple-library. Visited: 2017-09-07.

[4] Ibm mq. https://www.ibm.com/us-en/marketplace/secure-messaging. Visited: 2017-09-07.

[5] libcoap. https://libcoap.net/. Visited: 2017-09-10.

[6] microcoap. https://github.com/1248/microcoap. Visited: 2017-09-07.

[7] microcoap fork. https://github.com/cache91/microcoap. Visited: 2017-09-10.

[8] Mqtt libraries. https://github.com/mqtt/mqtt.github.io/wiki/libraries. Visited: 2017-09-07.

[9] Mqtt products that are "things". https://github.com/mqtt/mqtt.github.io/wiki/things. Visited: 2017-09-07.

[10] An open source mqtt v3.1 broker. https://mosquitto.org/. Visited: 2017-09-12.

[11] Pubsubclient library. https://github.com/knolleary/pubsubclient. Visited: 2017-09-07.

[12] Yuang Chen and Thomas Kunz. Performance evaluation of iot protocols under a constrained wireless access network. In *Selected Topics in Mobile & Wireless Networking (MoWNeT), 2016 International Conference on*, pages 1–7. IEEE, 2016.

[13] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap). 2014.

[14] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of mqtt and coap via a common middleware. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.