Systems of Systems (SYOSY)

M.Sc. In Critical Computing Systems Engineering

ISEP/IPP – 2021/22, 2nd semester

Assignment 1:

M2M Messaging Protocols

Pedro Santos

Outline

1. Introduction to M2M

[Part 1]

2. Review of MQTT
3. Inspecting QoS modes in MQTT

[Part 2]

4. Review of CoAP
5. Inspecting CoAP messages
6. Implementing a CoAP server in Python

[Part 3]

7. Implementing the OBSERVE option

Introduction to M2M Messaging Protocols

M2M Messaging Protocols

- Machine-to-Machine (M2M) communications are becoming more and more relevant, to enable inter-machine communication
- Dedicated messaging protocols have been proposed as middleware to this type of communication
- Examples
  - Message Queue Telemetry Transport (MQTT)
  - Constrained Application Protocol (CoAP)

Review of CoAP

CoAP Overview

Motivation

- Constrained Application Protocol
- COAP was developed for Machine-to-Machine scenarios, in which simple devices need to communicate.
- It is particularly targeted for small low power sensors, switches, valves and similar components that need to be controlled or supervised remotely.

REST-based

- CoAP protocol was developed in accordance with **Representational State Transfer (REST)** architectural style, which HTTP also follows.
- It was thought that having CoAP methods resemble HTTP method requests and responses would be beneficial to facilitate adoption and become interoperable.
- Unlike REST, it has a very light and simple packet structure that is designed to be as minimal as possible, with binary data representation for commands and data whenever possible.

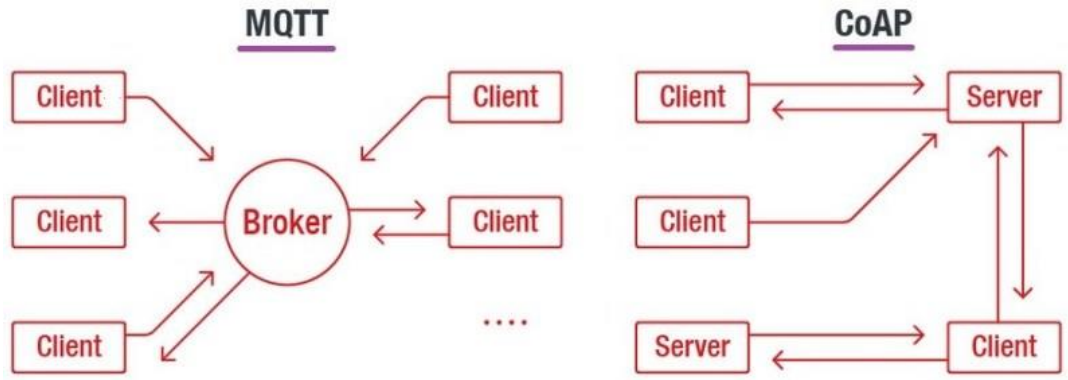
REST Architectural style

- Resources are identified by **URI – Uniform Resource Identifiers**
- Similarities to HTTP:
  - HTTP also defines URLs – *Uniform Resource Locator*
  - Methods are very similar to HTTP methods
  - Response codes are a subset of HTTP response codes
  - Options carry additional information (similar to HTTP header lines, but using a more compact encoding)
- CoAP Methods
  - **GET :** Retrieves information of an identified resource
  - **POST :** Creates a new resource under the requested URI
  - **PUT :** Updates the resource identified by an URI
  - **DELETE :** Deletes the resource identified by an URI
- Additional methods under CoAP
  - FETCH: This method provides a solution that spans the gap between the use of GET and POST. As with POST, the input to the FETCH operation is passed along within the payload of the request rather than as part of the request URI. Unlike POST, however, the semantics of the FETCH method are more specifically defined;
  - PATCH. Using PATCH avoids transferring all data associated with a resource in case of modifications, thereby not burdening the constrained communication medium.

Server-Client & Device Grouping

Server/Client one-to-one model

- CoAP follows a client-request paradigm, unlike MQTT that follows a publisher-subscriber paradigm
- Each client connects to the server and sends/requests data.
- The server doesn't manage many-client message routing.
- However, similarly to MQTT, the client can become an 'observer' to get frequent asynchronous updates to a topic of interest (unlike traditional REST)



Device Grouping

- It supports the ability to send a request directly to a group of devices, thereby implementing multicast.
- CoAP devices with limited resources can be grouped together or by **the same function of the devices** (taking light or temperature), or **by location** (taking a reading in a certain room, floor of the building)
- There are several ways to create a group:
  - The device can be pre-programmed for a certain group;
  - The device can be defined into the group through the resource directory;
  - The device can be defined in a group from a user device.

Transport & Session/States

Stateless and Sessionless

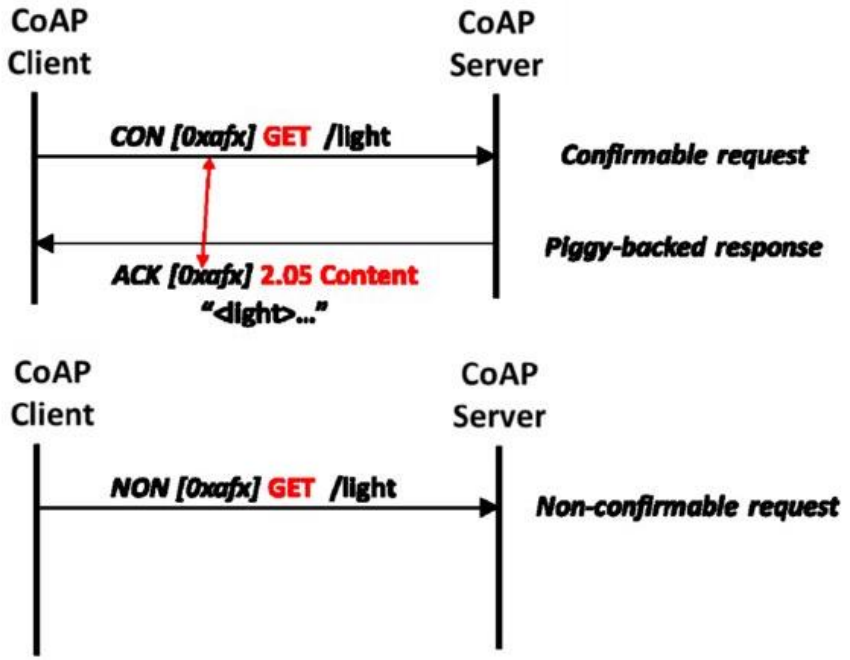
- HTTP/REST uses sessions but is stateless, you're expected to disconnect after data is transmitted. MQTT uses the idea of a ‘session’ or continuous connection (e.g. one socket, one session).
- CoAP is not only **stateless** (per connection), it's **sessionless**: data is sent and requested at *any* time, somewhat like if you had MQTT but without a connection state.
- That means you could run **CoAP on a transport like UDP, SMS, packet radio or satellite** where it’s hard to get immediate responses!

Transport

- The CoAP protocol uses **UDP (User Datagram Protocol)** as the default transport protocol.
- This reduces the size of the service data and increase efficiency. All messages are coded in binary form.
- Limitations:
    - Firewalls that often block individual ‘random’ packets and only permit outgoing TCP connections (you can run CoAP over TCP like Particle but it isn’t as common) but then you sort of lose the benefit of UDP.
    - Another downside is that since each message is stand-alone you cannot have any fragmentation - each message must fit in a single ZigBee, UDP, Sigfox, etc. packet.

Message Types

- The CoAP protocol has four types of messages:
  - **Confirmable:** some messages require an acknowledgement. These messages are called "Confirmable". When no packets are lost, each confirmable message elicits exactly one return message of type Acknowledgement or type Reset
  - **Non-confirmable:** Some other messages do not require an acknowledgement. This is particularly true for messages that are repeated regularly for application requirements, such as repeated readings from a sensor where eventual arrival is sufficient.
  - **Acknowledgment:** An Acknowledgement message acknowledges that a specific confirmable message (identified by its Message ID) arrived
  - **Reset:** A Reset message indicates that a specific message (confirmable or non-confirmable) was received, but some context is missing to properly process it. This condition is usually caused when the receiving node has rebooted and has forgotten some state that would be required to interpret the message.



CoAP Message Structure

- **Version (Ver):** 2-bit unsigned integer. Indicates the CoAP version number.
- **Type (T):** 2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-Confirmable (1), Acknowledgement (2) or Reset (3).
- **Option Count (OC):** Indicates the number of options after the header (0-14).
  - If set to 0, there are no options and the payload (if any) immediately follows the header.
  - If set to 15, then the number of options is unlimited, and an end-of-options marker is used to indicate no more options.
- **Code:** 8-bit unsigned integer. Indicates if the message carries a request (1-31) or a response (64-191), or is empty (0).
- **Message ID:** 16-bit unsigned integer. Used for the detection of message duplication, and to match messages of type Acknowledgement/Reset and messages of type Confirmable/Non-confirmable.

<table>
<tr><td>4
V T</td><td>0
TKL</td><td>02
Code</td><td>0000
Message ID</td></tr>
<tr><td colspan="4">b66576656e7473
Options: Uri-Path="events"</td></tr>
<tr><td>ff
Payload Marker</td><td colspan="3">7b22616363657373546f6b656e223a22645f
736b5f616263313233222c226e616d65223a
2274656d706572617475726522c22646174
61223a223231227d
Payload: {"accessToken":"d_sk_abc123","name":"temperature","data":"21"}</td></tr>
</table>

OBSERVE option

- The OBSERVE option is an extension of the CoAP GET method. More precisely, it is an optional field in the GET request header.
- **When a client queries the server with an Observe option, it basically asking for the current status of the alarm and also to be notified if it changes in the future**.
- Note that the server does not have to honour the observe request.
  - For example, if a CoAP resource doesn't support Observers or it has reached the maximum registered observers.
  - In this case, the Observe option will just be ignored and the request will default to a plain GET request.
- The server may also periodically send the current state of the resource to all registered observers.
  - If it doesn't hear anything back from any observer then that observer will be removed from the resource's registered observers list.
  - This is one mechanism the server uses to clean up the observers list in the event any client silently disappears.

First Contact with CoAP

Steps

1. Install CoAP library *libcoap*
2. Test CoAP client and server
3. Connect to RPi for inspecting traffic

1./2.Install and setup libcoap

1. Setup
    1. `git clone https://github.com/obgm/libcoap.git`
    2. `./autogen.sh`
    3. `./configure --disable-documentation --disable-dtls`
    4. `./configure --disable-documentation --disable-shared --without-debug CFLAGS="-D COAP_DEBUG_FD=stderr"`
    5. `make`
    6. `make install`

2. Test client and server
    - In one terminal: `coap-server`
    - In another: `coap-client …`
        - `coap-client -m get coap://[::1]/time` `[Local host]`
        - `coap-client -m get coap://coap.me:5683` `[Cloud service]`

3. Inspect CoAP Traffic

1. Connect to RPI via VNC (if in Windows)
2. Start Wireshark: `sudo wireshark`
3. Apply filter ‘coap’
4. Try `coap-client -m get coap://coap.me:5683` . with:
    1. CONFIRMABLE
    2. NON-CONFIRMABLE

    https://libcoap.net/doc/reference/develop/man_coap-client.html
5. Try `coap-client -m get coap://[::1]/time` .
    1. What did you see?
    2. What about if you connect to another Raspberry?

Implementing a CoAP Server in Python

Create a CoAP server and Client

- We will use **aiocoap – the Python CoAP library**
- It is written in Python 3 using its native ***asyncio*** methods to facilitate concurrent operations while maintaining an easy-to-use interface.
- ***asyncio*** is used in multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.
- ***asyncio*** is a library to write concurrent code using the **async/await** syntax.
- **Features / Standards**
  - RFC7252 (CoAP): Supported for clients and servers. Multicast is supported on the server side, and partially for clients. DTLS is supported but experimental, and lacking some security properties. No caching is done inside the library.
  - RFC7641 (Observe): Basic support for clients and servers. Reordering, re-registration, and active cancellation are missing.
  - RFC7959 (Blockwise): Supported both for atomic and random access.
  - RFC8323 (TCP, WebSockets): Supports CoAP over TCP, TLS, and WebSockets (both over HTTP and HTTPS). The TLS parts are server-certificate only; preshared, raw public keys and client certificates are not supported yet.
  - RFC7967 (No-Response): Supported.
  - RFC8132 (PATCH/FETCH): Types and codes known, FETCH observation supported.

Preliminary Step: install aiocoap

1. Install python (if not already)
2. `pip install aiocoap`

Server – Function for Alarm Resource

```
# server.py
import aiocoap.resource as resource
import aiocoap
import asyncio

class AlarmResource(resource.Resource):
  """This resource supports the PUT method.
  PUT: Update state of alarm."""

  def __init__(self):
      super().__init__()
      self.state = "OFF"

  async def render_put(self, request):
      self.state = request.payload
      print('Update alarm state: %s' % self.state)

      return aiocoap.Message(code=aiocoap.CHANGED, payload=self.state)
```

Class

Function

Initial Alarm state

Handling PUT request arriving from client

Function, in this case *async* informs that can happen at any time

Server – Main

```
def main():
  # Resource tree creation
  root = resource.Site()
  root.add_resource(['alarm'], AlarmResource())

  asyncio.Task(aiocoap.Context.create_server_context(root, bind=('localhost', 5683)))
  asyncio.get_event_loop().run_forever()

if __name__ == "__main__":
  main()
```

Create resource (alarm) through call of function

Start server

Server IP

Server Port

class **aiocoap.protocol.** **Context** *(loop=None, serversite=None, loggername='coap', client_credentials=None, server_credentials=None)*

Bases: `aiocoap.interfaces.RequestProvider`

Applications' entry point to the network

A `Context` coordinates one or more network `transports` implementations and dispatches data between them and the application.

Client_PUT – Main

```
# client_put.py
import asyncio
import random

from aiocoap import *

async def main():
  context = await Context.create_client_context()
  alarm_state = random.choice([True, False])
  payload = b"OFF"

  if alarm_state:
      payload = b"ON"

  request = Message(code=PUT, payload=payload, uri="coap://localhost/alarm")

  response = await context.request(request).response
  print('Result: %s\n%r'%(response.code, response.payload))

if __name__ == "__main__":
  asyncio.get_event_loop().run_until_complete(main())
```

Wait for event

Prepare and send request

Run client

End of Part 2

References

- Chris Dihn. “Creating a Simple CoAP Server with Python”. *https://aniotodyssey.com/2021/06/12/creating-a-simple-coap-server-with-python*