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

Introduction to M2M Messaging Protocols

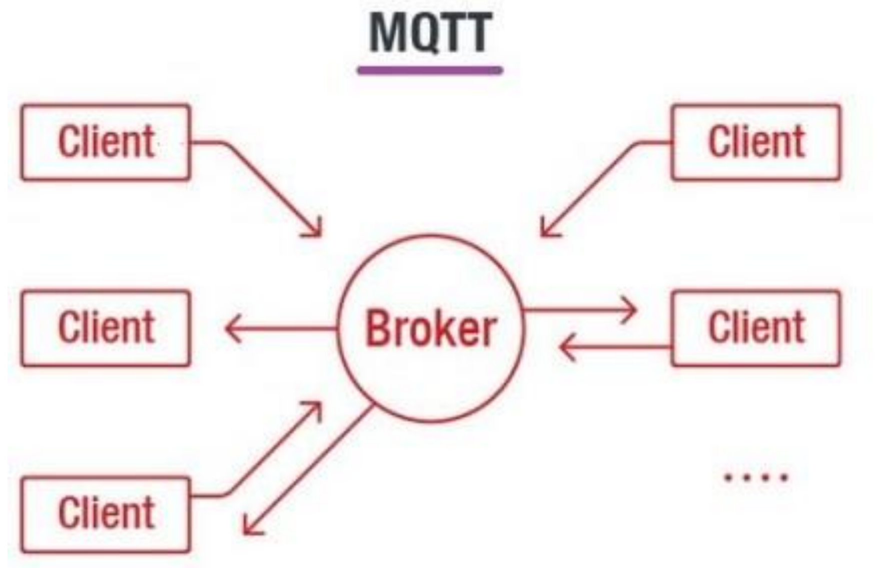
M2M Messaging Protocols

- 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 MQTT

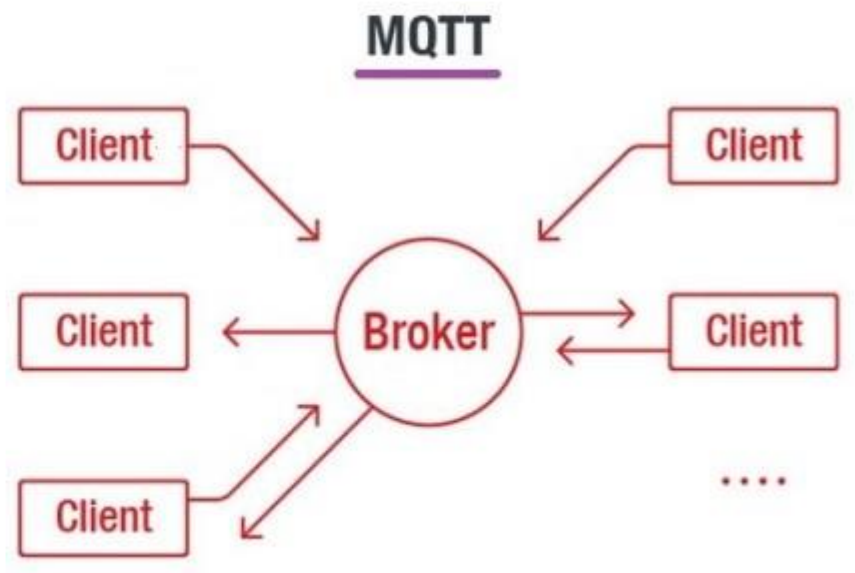
Message Queue Telemetry Transport (MQTT)

- MQTT is an OASIS standard messaging (and an ISO recommendation ISO/IEC 20922). protocol for the Internet of Things (IoT) and was designed by IBM.
- Lightweight machine-to-machine communication protocol for topic-based publish-subscribe architectures.
- Oriented for connecting remote devices with small code footprint and minimal network bandwidth.
- Publisher-subscriber paradigm:



Publisher-Subscriber Paradigm

- Clients do not have addresses like in email systems, and messages are not sent to clients.
- **Messages are published to a broker on a topic**. For example, a publisher might send a message temp: 22.5 on a topic heating/thermostat/living-room.
- **The job of an MQTT broker is to filter messages based on topic, and then distribute them to subscribers**.
- A client can receive these messages by subscribing to that topic on the same broker
- There is no direct connection between a publisher and subscriber.
- All clients can publish (broadcast) and subscribe (receive).
- MQTT brokers do not normally store messages.



Connections and Client IDs

Connections

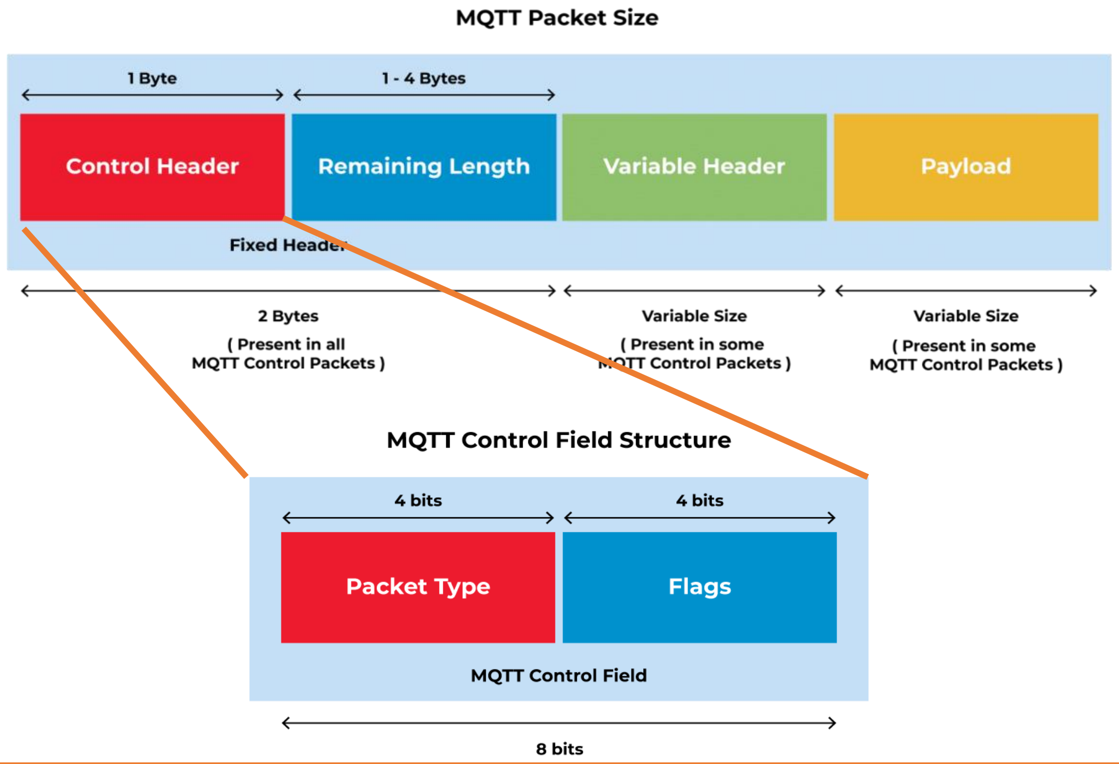
- Connections are acknowledged by the broker using a Connection acknowledgement message.
- ***You cannot publish or subscribe unless you are connected.***

Client Name or Client ID

- All clients are required to have a client name or ID.
- The client name is used by the MQTT broker to track subscriptions etc.
- Client names must also be unique.
- If you attempt to connect to an MQTT broker with the same name as an existing client then the existing client connection is dropped.

MQTT Packet Format

- It requires a fixed header of 2 bytes.
- The first byte is the control header (where the first 4 bits are message type and the other 4 bits are control flags) and the packet length goes in the second byte, extending for 3 more bytes is necessary.



Message Type:

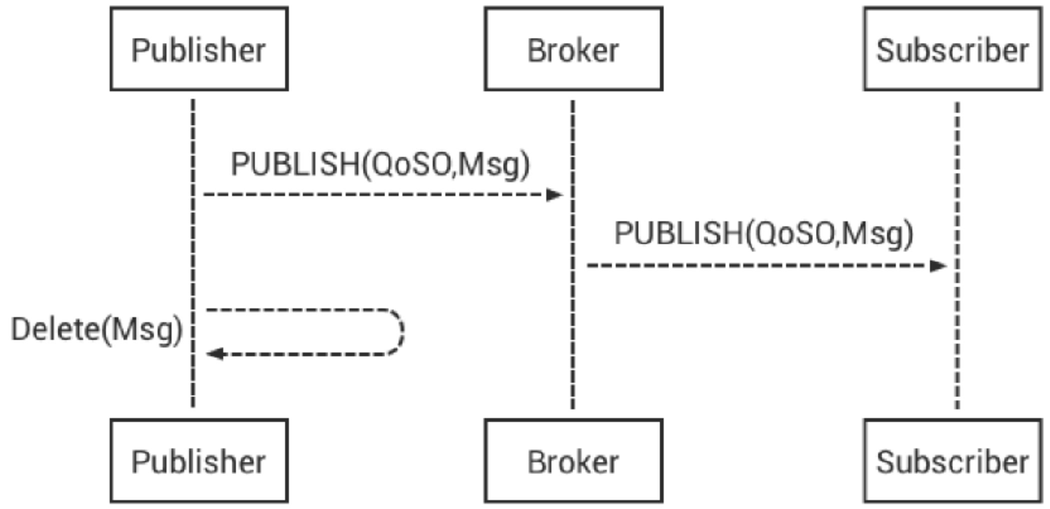
Message type	Value	Description	Fixed Header
Reserved	0	Reserved	Present
CONNECT	1	Client connect request to server or broker	Present
CONNACK	2	Connect request acknowledgment	Present
PUBLISH	3	Publish message	Present
PUBACK	4	Publish acknowledgment	Present
PUBREC	5	Publish receive	Present
PUBREL	6	Publish release	Present
PUBCOMP	7	Publish complete	Present
SUBSCRIBE	8	Client subscribe request	Present
SUBACK	9	Subscribe request acknowledgment	Present
UNSUBSCRIBE	10	Unsubscribe request	Present
UNSUBACK	11	Unsubscribe acknowledgment	Present
PINGREQ	12	PING request	Present
PINGRESP	13	PING response	Present
DISCONNECT	14	Client is disconnecting	Present
Reserved	15	Reserved	Present

Header Flags Structure in details:

3	2	1	0	bit
DUP	**QOS**	**QOS**	**RETAIN**	Header Flags

Reliability in MQTT: QoS modes

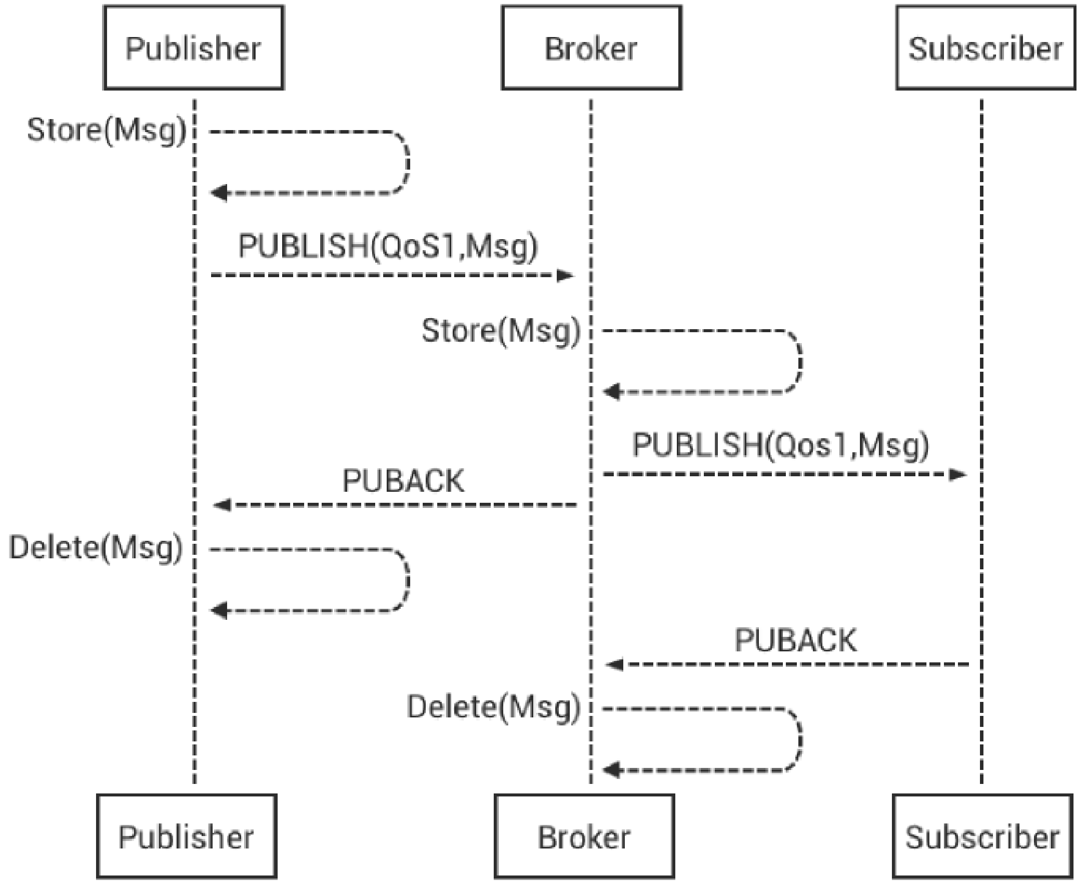
- MQTT offers three levels of Quality of Service (QoS) for reliable message delivery: QoS0, QoS1, and QoS2.
- QoS0 (commonly called Fire and Forget or At most once) offers only a best-effort delivery. There is no guarantee of delivery and the recovery effort is null or minimal.
- The recipient does not acknowledge receipt of the message and the message is not stored and re-transmitted by the sender. QoS level 0 is often called "fire and forget" and provides the same guarantee as the underlying TCP protocol.
- Use QoS 0 when …
  - **You have a completely or mostly stable connection between sender and receiver.** A classic use case for QoS 0 is connecting a test client or a front end application to an MQTT broker over a wired connection.
  - **You don't mind if a few messages are lost occasionally.** The loss of some messages can be acceptable if the data is not that important or when data is sent at short intervals
  - **You don't need message queuing.** Messages are only queued for disconnected clients if they have QoS 1 or 2 and a persistent session.



QoS0 – *Fire and Forget*

MQTT QoS1 mode

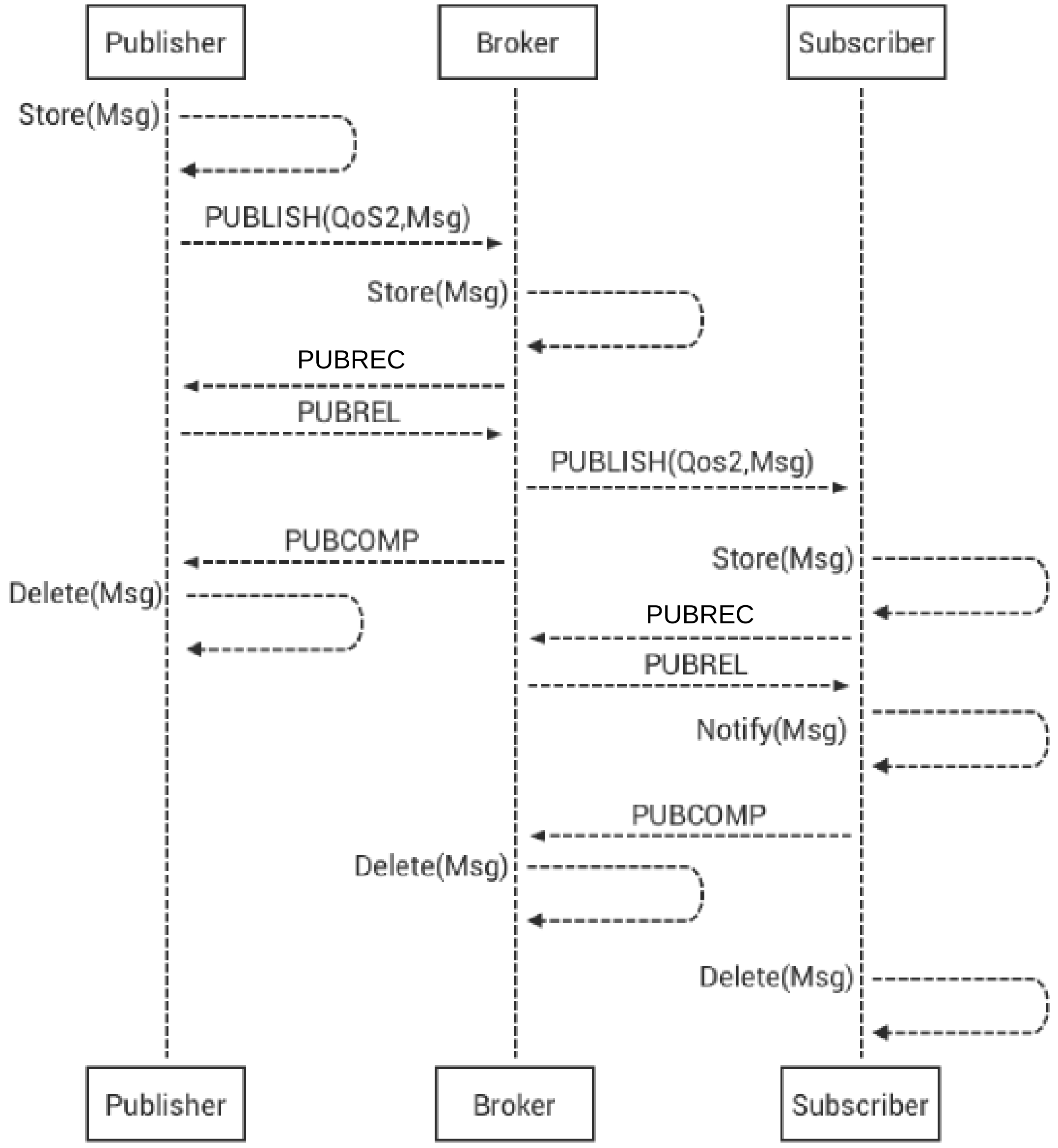
- **QoS1 (At least Once) guarantees that a message is delivered at least one time to the receiver**. The sender stores the message until it gets a PUBACK packet from the receiver that acknowledges the message.
  1. The sender uses the packet identifier in each packet to match the PUBLISH packet to the corresponding PUBACK packet. If the sender does not receive a PUBACK packet in a reasonable amount of time, the sender resends the PUBLISH packet.
  2. When a receiver gets a message with QoS 1, it can process it immediately. For example, if the receiver is a broker, the broker sends the message to all subscribing clients and then replies with a PUBACK packet.
  3. If the publishing client sends the message again it sets a duplicate (DUP) flag. In QoS 1, this DUP flag is only used for internal purposes and is not processed by broker or client. The receiver of the message sends a PUBACK, regardless of the DUP flag.
- Use QoS 1 when …
  - **You need to get every message and your use case can handle duplicates.** QoS level 1 is the most frequently used service level because it guarantees the message arrives at least once but allows for multiple deliveries. Of course, your application must tolerate duplicates and be able to process them accordingly.
  - You can't bear the overhead of QoS 2. **QoS 1 delivers messages much faster than QoS 2.**



QoS1 - At least once

MQTT QoS2 mode

- QoS2 offers the highest reliability - Exactly Once. This level guarantees that each message is received only once by the intended recipients.
- This guarantee is provided by at least two request/response flows (a four-part handshake) between the sender and the receiver.
  1. Receiver gets a QoS 2 PUBLISH packet from a sender, and replies with a PUBREC packet that acknowledges the PUBLISH packet. If the sender does not get a PUBREC packet from the receiver, it sends the PUBLISH packet again with a duplicate (DUP) flag.
  2. Once the sender receives a PUBREC packet from the receiver, the sender can discards the initial PUBLISH packet. The sender stores the PUBACK packet from the receiver and responds with a PUBREL packet.
  3. After the receiver gets the PUBREL packet, it can discard all stored states and answer with a PUBCOMP packet. After the sender receives the PUBCOMP packet, the packet identifier of the published message becomes available for reuse.
- Use QoS 2 when …
  - **It is critical to your application to receive all messages exactly once**. This is often the case if a duplicate delivery can harm application users or subscribing clients. Be aware of the overhead and that the QoS 2 interaction takes more time to complete.



QoS2 – Exactly Once

Persistency & Retained Messages

Persistence

- Using a persistent connection, **the broker will store subscription information, and undelivered messages for the client**.
- In order for the broker to store session information for a client, a client id must be used.
- When a client connects to broker it indicates whether the connection is **persistent** or not, by setting the 'clean-session' flag to FALSE or TRUE respectively, in the CONNECT packets.
  - **In command-line, clean session is the default**. In the *mosquitto_sub* command, **add flag `'–c'` (`'--disable-clean-session'`) for a persistent connection.**
- However it is important to realise that not all messages will be stored for delivery, as the quality of service, of the subscriber and publisher has an effect.

Retained Messages

- If a publisher publishes a message to a topic and no one is subscribed to that topic, the message is simply discarded by the broker.
- However the publisher can tell the broker to keep the last message on that topic by setting the "retained" message flag.
- This can be very useful, as for example, if you have sensor publishing its status only when changed e.g. Door sensor. What happens if a new subscriber subscribes to this status? Without retained messages the subscriber would have to wait for the status to change before it received a message.
- What is important to understand is that only one message is retained per topic. The next message published on that topic replaces the last retained message for that topic.

QoS modes in MQTT

Steps

1. Install MQTT Broker
2. Configure MQTT Broker
3. Connect to RPi for inspecting traffic
4. Test QoS modes

1./2. Install & Configure MQTT Broker

1. Follow instructions here:
   - https://appcodelabs.com/introduction-to-iot-build-an-mqtt-server-using-raspberry-pi
   - (note: to test publish & subscribe you must have two SSH connections)

2. Configure MQTT Broker:
   - In */etc/mosquitto/mosquitto.conf*, add
     - `allow_anonymous true`
     - `listener 1883`

3. Inspect different types of MQTT QoS

1. Connect to RPI via VNC (if in Windows)
2. Start Wireshark: `sudo wireshark`
3. Apply filter ‘MQTT’
4. Start

```
sudo systemctl stop mosquitto
sudo pkill mosquitto
sudo mosquitto -v –c /etc/mosquitto/mosquitto.conf
sudo tail –f /var/log/mosquitto/mosquitto.log
```

4. Test QoS modes

Common steps:

1. Initialise the client, requesting or not a ‘clean_session’
2. Subscribe to a topic with QoS set
3. Disconnect.
4. Publish to the topic that the client subscribed to with QOS set.
5. Reconnect client
6. Make note of any messages received

4. Test QoS modes

Test 1 – QoS=0 & no Persistence

- Publish / Subscribe QoS: 0
- clean_session=TRUE (‘-c’ flag is omitted, so default behaviour is non-persistent connection)
- Steps:
  1. Start Pub: `mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 1" -q 0`
  2. Start Sub: `mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0`
- Result: ?

Test 2 – QoS=0 & Persistence

- Publish / Subscribe QoS: 0
- “clean_session”=FALSE (‘-c’ flag indicates persistent connection)
- Steps:
  1. Start Pub: `mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 2" -q 0`
  2. Start Sub: `mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 –c`
- Result: ?

4. Test QoS modes

Test 3 – QoS=1/2 & Persistence

- Publish / Subscribe QoS: 1 or 2
- “clean_session”=FALSE
- Steps:
  1. Start Pub: `mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 3" -q 1/2`
  2. Start Sub: `mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 1/2 -c`
- Result: ?

Test 4 – Different Pub/Sub QoS

- Publish QoS: 1; Subscribe QoS: 0
- “clean_session”=FALSE
- Steps:
  1. Start Pub: `mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 4" -q 1`
  2. Start Sub: `mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 -c`
- Result: ?

4. Test QoS modes

Test 5 – Different Pub/Sub QoS

- Publish QoS: 0; Subscribe QoS: 1
- “clean_session”=FALSE
- Steps:
  1. Start Pub: `mosquitto_pub -h localhost -t "test/message" -m "MESSAGE 4" -q 1`
  2. Start Sub: `mosquitto_sub -h localhost -i Sub1 -t "test/message" -q 0 -c`
- Result: ?

End of Part 1

References

- Steve Cope. “MQTT Clean Sessions and QOS Examples”. *http://www.steves-internet-guide.com/mqtt-clean-sessions-example/*

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

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.

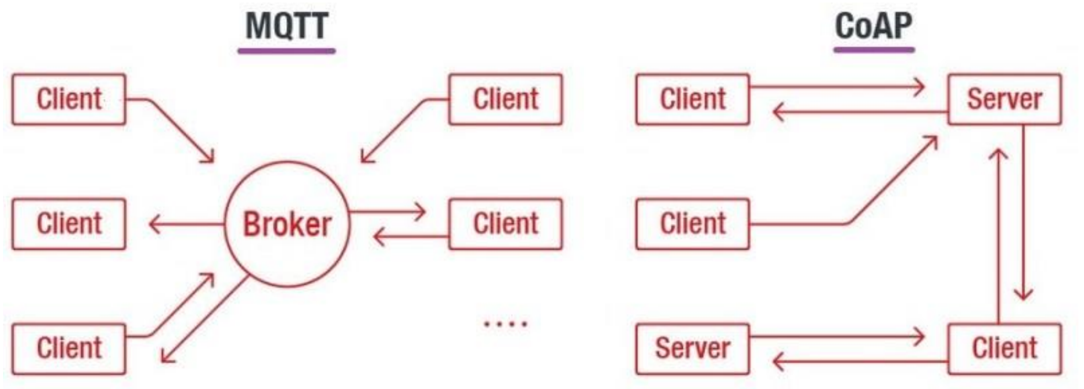
REST Architectural style

- 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

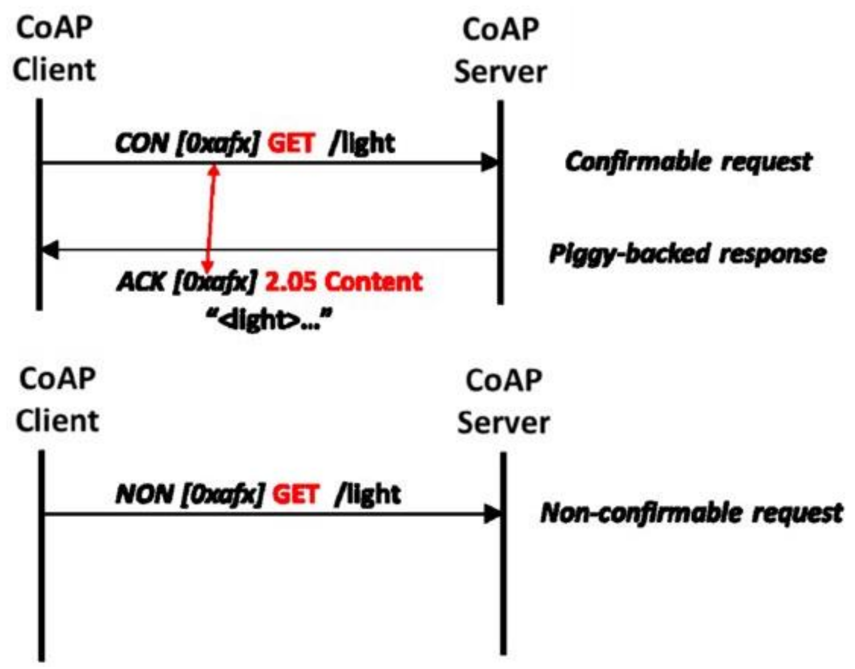
- HTTP/REST uses sessions but is stateless, you're expected to disconnect after data is transmitted. MQTT uses the idea of a ‘session’ or continuous connection (e.g. one socket, one session).
- CoAP is not only **stateless** (per connection), it's **sessionless**: data is sent and requested at *any* 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.

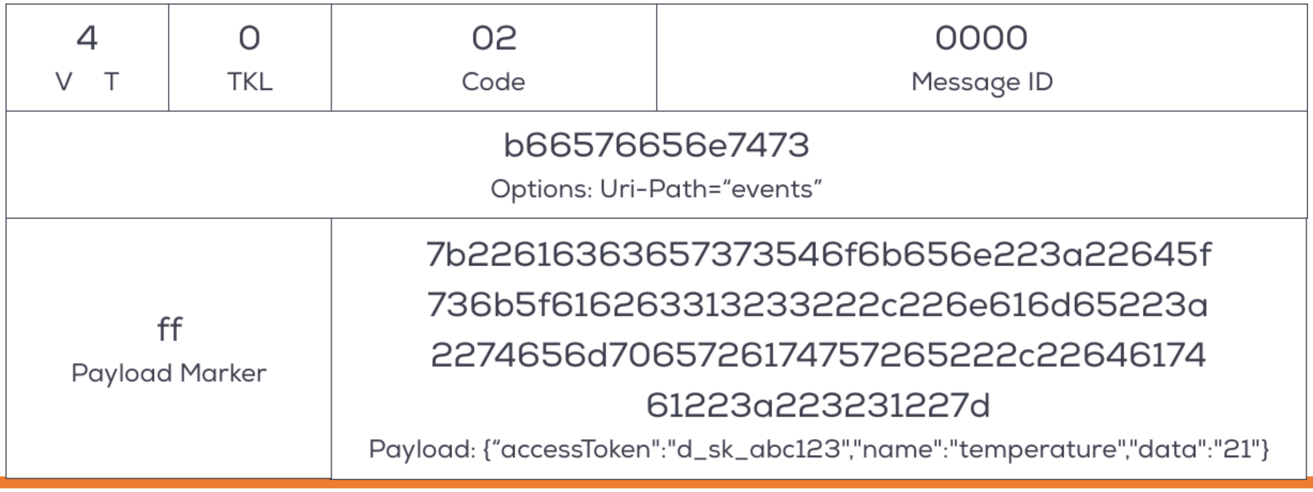
Message Types

- 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.



4 V T	0 TKL	02 Code	0000 Message ID
b66576656e7473 Options: Uri-Path="events"			
ff Payload Marker		7b226163636573735546f6b656e223a22645f 736b5f616263313233222c226e616d65223a 2274656d706572617475726522c22646174 61223a223231227d Payload: {"accessToken":"d_sk_abc123","name":"temperature","data":"21"}	

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 → class AlarmResource

Function → def __init__

Initial Alarm state → self.state = "OFF"

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

Handling PUT request arriving from client → render_put(self, request):

Server – Main

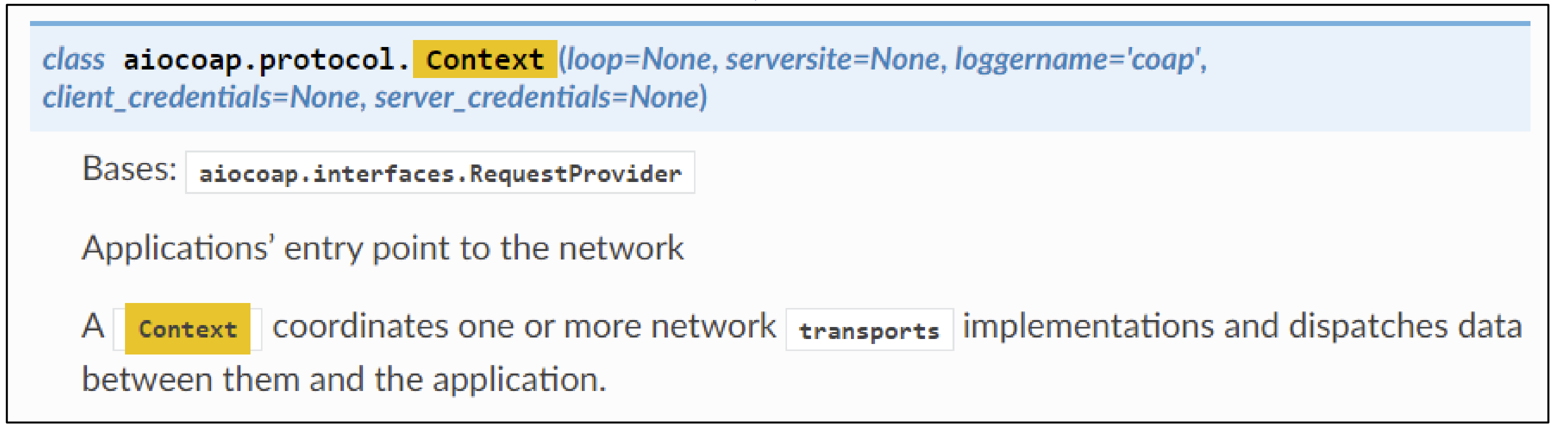
```
def main():
  # Resource tree creation
  root = resource.Site()
  root.add_resource(['alarm'], AlarmResource())

  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*

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

Implementing the Observe Option

Implementing CoAP Observe option

Revisiting the 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 asking for the current status of the alarm and also to be notified if it changes in the future.
- Let proceed with the implementation by updating the AlarmResource class.

1. Instead of inheriting from *aiocoap*’s Resource class, AlarmResource will now inherit from Aiocoap’s ObservableResource. This will provide the functionality to manage the observers, we just need to handle what to send and when to send it.
2. Let also update the handling of the PUT request so that when the status of the alarm is updated, it will set a flag to indicate the server to notify observers.
3. Add a notify_observers_check() method which continuously loop and check to see if the notify_observers flag is set or not. If it is set then the server will send an update to each observer by calling render_get().

Server OBSERVE – Main

```
import aiocoap.resource as resource
import aiocoap
import threading
import logging
import asyncio

class AlarmResource(resource.ObservableResource):
  ...

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
  # Resource tree creation
  root = resource.Site()
  alarmResource = AlarmResource()
  root.add_resource(['alarm'], alarmResource)
  asyncio.Task(aiocoap.Context.create_server_context(root, bind=('localhost', 5683)))

  # Spawn a daemon to notify observers when alarm status changes
  observers_notifier = threading.Thread(target=alarmResource.notify_observers_check)
  observers_notifier.daemon = True
  observers_notifier.start()

  asyncio.get_event_loop().run_forever()

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

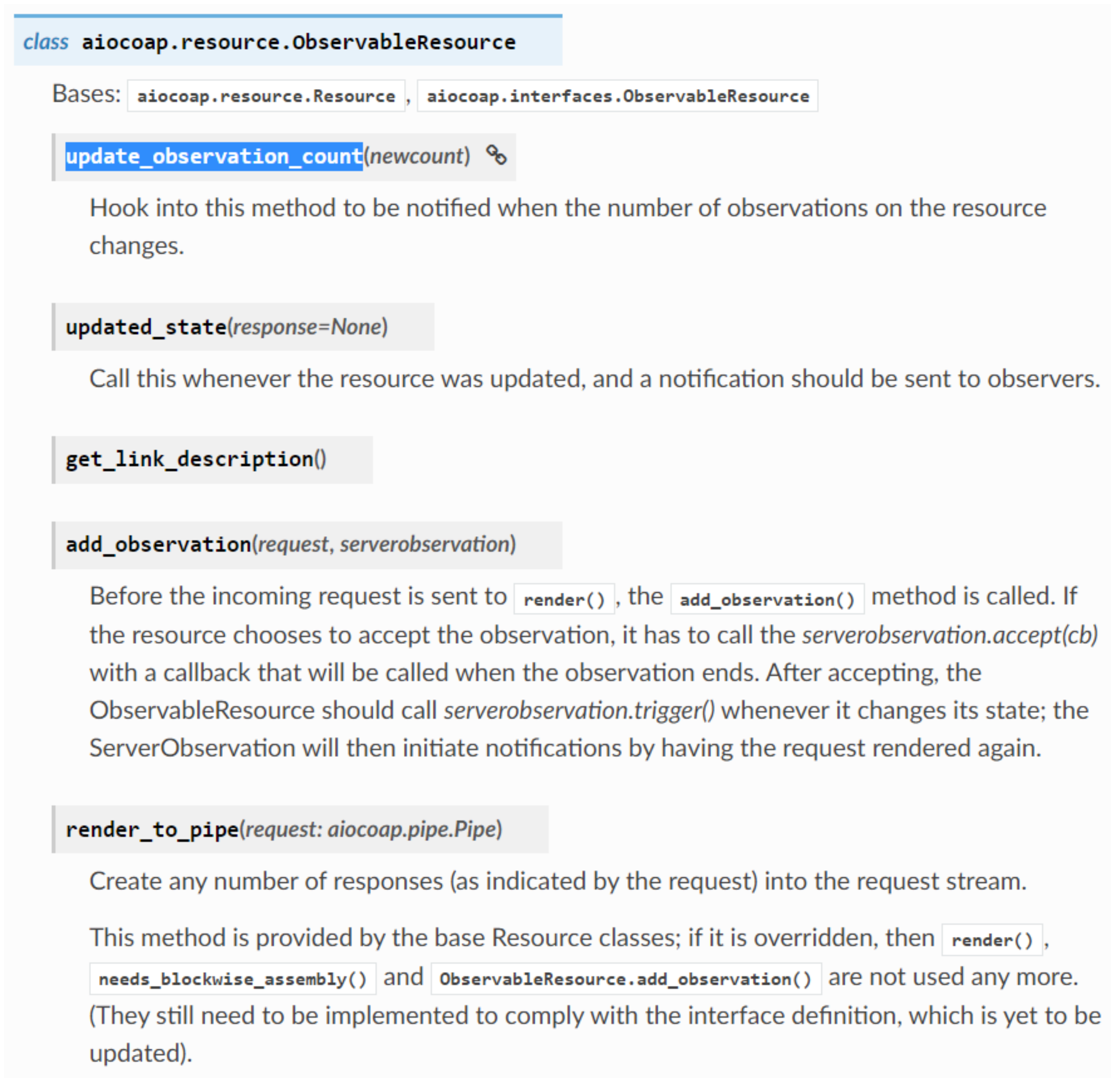
1. Instead of inheriting from Aiocoap's Resource class, AlarmResource will now inherit from Aiocoap's ObservableResource. This will provide the functionality to manage the observers, we just need to handle what to send and when to send it.

Server IP

Server Port

Start server

ObservableResource



class **aiocoap.resource.ObservableResource**

Bases: `aiocoap.resource.Resource`, `aiocoap.interfaces.ObservableResource`

update_observation_count(*newcount*)

Hook into this method to be notified when the number of observations on the resource changes.

updated_state(*response=None*)

Call this whenever the resource was updated, and a notification should be sent to observers.

get_link_description()

add_observation(*request*, *serverobservation*)

Before the incoming request is sent to `render()`, the `add_observation()` method is called. If the resource chooses to accept the observation, it has to call the *serverobservation.accept(cb)* with a callback that will be called when the observation ends. After accepting, the ObservableResource should call *serverobservation.trigger()* whenever it changes its state; the ServerObservation will then initiate notifications by having the request rendered again.

render_to_pipe(*request: aiocoap.pipe.Pipe*)

Create any number of responses (as indicated by the request) into the request stream.

This method is provided by the base Resource classes; if it is overridden, then `render()`, `needs_blockwise_assembly()` and `ObservableResource.add_observation()` are not used any more. (They still need to be implemented to comply with the interface definition, which is yet to be updated).

https://aiocoap.readthedocs.io/en/latest/module/aiocoap.resource.html

Server OBSERVE – Main

```
import aiocoap.resource as resource
import aiocoap
import threading
import logging
import asyncio

class AlarmResource(resource.ObservableResource):
   ...

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
   # Resource tree creation
   root = resource.Site()
   alarmResource = AlarmResource()
   root.add_resource(['alarm'], alarmResource)
   asyncio.Task(aiocoap.Context.create_server_context(root, bind=('localhost', 5683)))

   # Spawn a daemon to notify observers when alarm status changes
   observers_notifier = threading.Thread(target=alarmResource.notify_observers_check)
   observers_notifier.daemon = True
   observers_notifier.start()

   asyncio.get_event_loop().run_forever()

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

Separate thread to notify observers when resource state changes

Server OBSERVE – *AlarmResource* Structure

```python
class AlarmResource(resource.ObservableResource):
  """This resource supports the GET and PUT methods and is observable.
  GET: Return current state of alarm
  PUT: Update state of alarm and notify registered observers
  """

  def __init__(self):
    ...

  # Handles PUT request
  async def render_put(self, request):
    ...

  # Handles GET request or observer notify
  async def render_get(self, request):
    ...

  # Observers change event callback
  def update_observation_count(self, count):
    ...

  # Ensure observers are notified if required
  def notify_observers_check(self):
    ...
```

Server OBSERVE – *render_put*

```
class AlarmResource(resource.ObservableResource):
  """This resource supports the GET and PUT methods and is observable.
  GET: Return current state of alarm
  PUT: Update state of alarm and notify registered observers
  """
  def __init__(self):
    super().__init__()
    self.status = "OFF“
    self.has_observers = False
    self.notify_observers = False

  # Handles PUT request
  async def render_put(self, request):
    self.status = request.payload.decode('ascii')
    print('Update alarm state: %s' % self.status)
    self.notify_observers = True
    return aiocoap.Message(code=aiocoap.CHANGED, payload=b'%s' % self.status.encode('ascii’))

  # Handles GET request or observer notify
  async def render_get(self, request):
  # Observers change event callback
  def update_observation_count(self, count):
  # Ensure observers are notified if required
  def notify_observers_check(self):
```

Two new flags

2. Let also update the handling of the PUT request so that when the status of the alarm is updated, it will set a flag to indicate the server to notify observers.

Server OBSERVE – *render_get*

```
class AlarmResource(resource.ObservableResource):
  """This resource supports the GET and PUT methods and is observable.
  GET: Return current state of alarm
  PUT: Update state of alarm and notify registered observers
  """
  def __init__(self):

  # Handles PUT request
  async def render_put(self, request):

  # Handles GET request or observer notify
  async def render_get(self, request):
    print('Return alarm state: %s' % self.status)
    payload = b'%s' % self.status.encode('ascii')

    return aiocoap.Message(payload=payload)

  # Observers change event callback
  def update_observation_count(self, count):

  # Ensure observers are notified if required
  def notify_observers_check(self):
```

Server OBSERVE – *update_observation_count*

```
class AlarmResource(resource.ObservableResource):
  """This resource supports the GET and PUT methods and is observable.
  GET: Return current state of alarm
  PUT: Update state of alarm and notify registered observers
  """
  def __init__(self):

  # Handles PUT request
  async def render_put(self, request):

  # Handles GET request or observer notify
  async def render_get(self, request):

  # Observers change event callback
  def update_observation_count(self, count):
    if count:
      self.has_observers = True
    else:
      self.has_observers = False

  # Ensure observers are notify if required
  def notify_observers_check(self):
```

A method of ObservableResource

Server OBSERVE – *notify_observers_check*

```
class AlarmResource(resource.ObservableResource):
  """This resource supports the GET and PUT methods and is observable.
  GET: Return current state of alarm
  PUT: Update state of alarm and notify registered observers
  """
  def __init__(self):

  # Handles PUT request
  async def render_put(self, request):

  # Handles GET request or observer notify
  async def render_get(self, request):

  # Observers change event callback
  def update_observation_count(self, count):

  # Ensure observers are notify if required
  def notify_observers_check(self):
    while True:
      if self.has_observers and self.notify_observers:
        print('Notifying observers')
        self.updated_state()
        self.notify_observers = False
```

3. Add a notify_observers_check() method which continuously loop and check to see if the notify_observers flag is set or not. If it is set then the server will send an update to each observer by calling render_get().

A method of ObservableResource

Client OBSERVE

```
import logging
import asyncio
from aiocoap import *

logging.basicConfig(level=logging.INFO)

def observe_callback(response):
  ...

async def main():
  context = await Context.create_client_context()

  request = Message(code=GET)
  request.set_request_uri('coap://localhost/alarm')
  request.opt.observe = 0
  observation_is_over = asyncio.Future()

  try:
    context_request = context.request(request)
    context_request.observation.register_callback(observe_callback)
    response = await context_request.response
    exit_reason = await observation_is_over
    print('Observation is over: %r' % exit_reason)
  finally:
    if not context_request.response.done():
      context_request.response.cancel()
    if not context_request.observation.cancelled:
      context_request.observation.cancel()

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

Identification of resource to observe

0 (register) adds the entry to the list;
1 (deregister) removes the entry from the list.

Client OBSERVE

```
import logging
import asyncio
from aiocoap import *

logging.basicConfig(level=logging.INFO)

def observe_callback(response):
  ...

async def main():
  context = await Context.create_client_context()

  request = Message(code=GET)
  request.set_request_uri('coap://localhost/alarm')
  request.opt.observe = 0
  observation_is_over = asyncio.Future()

  try:
    context_request = context.request(request)
    context_request.observation.register_callback(observe_callback)
    response = await context_request.response
    exit_reason = await observation_is_over
    print('Observation is over: %r' % exit_reason)
  finally:
    if not context_request.response.done():
      context_request.response.cancel()
    if not context_request.observation.cancelled:
      context_request.observation.cancel()

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

Recall that OBSERVE is an option of GET

Identification of resource to observe

0 (register) adds the entry to the list;
1 (deregister) removes the entry from the list.

Send request

Register which function handles the response

Await for response

Client OBSERVE

```
import logging
import asyncio
from aiocoap import *

logging.basicConfig(level=logging.INFO)

def observe_callback(response):
  if response.code.is_successful():
    print("Alarm status: %s" % (response.payload.decode('ascii')))
  else:
    print('Error code %s' % response.code)

async def main():
  ...

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

Testing the Observe Option

- You will need three terminals
    - Term. 1: server_observe.py
    - Term. 2: client_put.py
    - Term. 3: client_server.py

End of Assignment 1

References

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