MQTT with Paho in Python

Install Paho MQTT

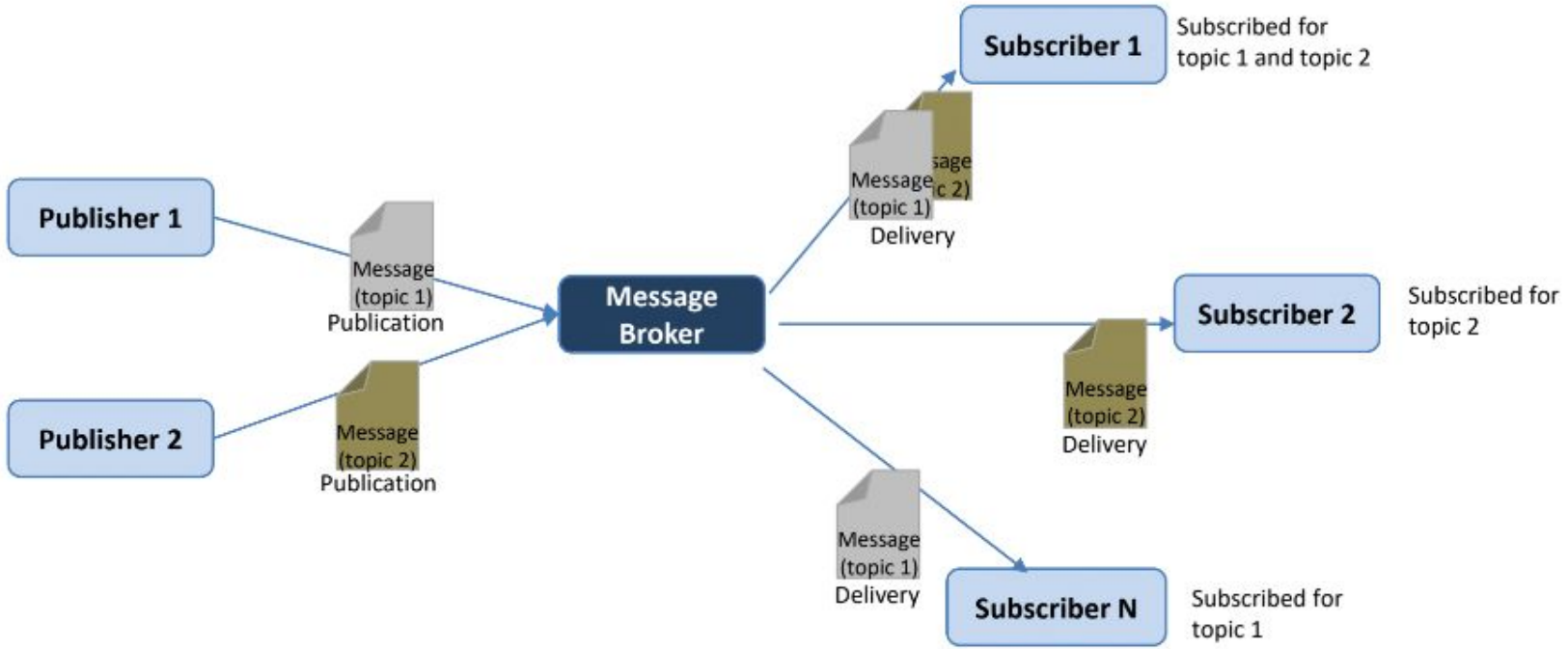
To install Paho MQTT, open a terminal and type `pip3 install paho-mqtt`

Polito giveth and Polito taketh

For all this exercises you will need to use a free MQTT broker. Unfortunately, if you are connected to the Polito WIFI you may **NOT** be able to reach every broker. A working one should be `mqtt.eclipseprojects.io`

In case a broker stops working (whether you're here or at home) you can choose another one from this website (use the one that does not require any authentication or apikey)

Intro - MQTT


Publisher 1
Publisher 2
Message (topic 1) Publication
Message (topic 2) Publication
Message Broker
Subscriber 1
Subscribed for topic 1 and topic 2
Message (topic 1) Delivery
Subscriber 2
Subscribed for topic 2
Message (topic 2) Delivery
Message (topic 1) Delivery
Subscriber N
Subscribed for topic 1

Intro - MQTT

We could briefly resume the structure of the MQTT communication paradigm with 3 main actors: *Publisher, Subscriber, Broker*

- The *Publisher* is the actor that wants to send messages tagged by a *topic*
- The *Subscriber* is the actor that wants to receive messages that belong to variable number of *topics*.
- The *Broker* is the actor in the middle: it receives the messages from all the publishers and forwards each of them to the subscriber according to the *topic* .

In the next slides you can find the examples for the implementation of a publisher and a subscriber

MyPublisher

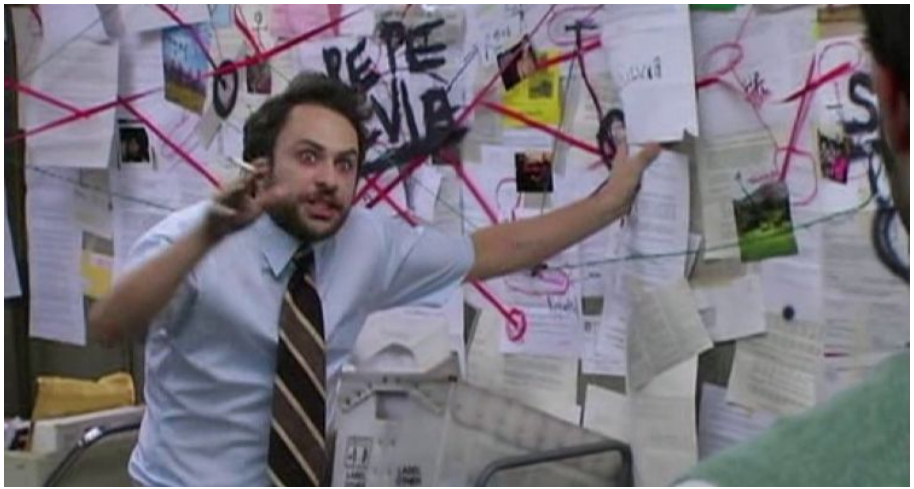
```
import paho.mqtt.client as PahoMQTT

class MyPublisher:
        def __init__(self, clientID,broker):
                self.clientID = clientID
                # create an instance of paho.mqtt.client
                self._paho_mqtt = PahoMQTT.Client(self.clientID, True)
                # register the callback
                self._paho_mqtt.on_connect = self.myOnConnect
                self.messageBroker = broker
        def start (self):
                #manage connection to broker
                self._paho_mqtt.connect(self.messageBroker, 1883)
                self._paho_mqtt.loop_start()
        def stop (self):
                self._paho_mqtt.loop_stop()
                self._paho_mqtt.disconnect()
        def myOnConnect (self, paho_mqtt, userdata, flags, rc):
                print ("Connected to %s with result code: %d" % (self.messageBroker, rc))
        def myPublish(self, topic, message):
                # publish a message with a certain topic
                self._paho_mqtt.publish(topic, message, 2)
```

MySubscriber

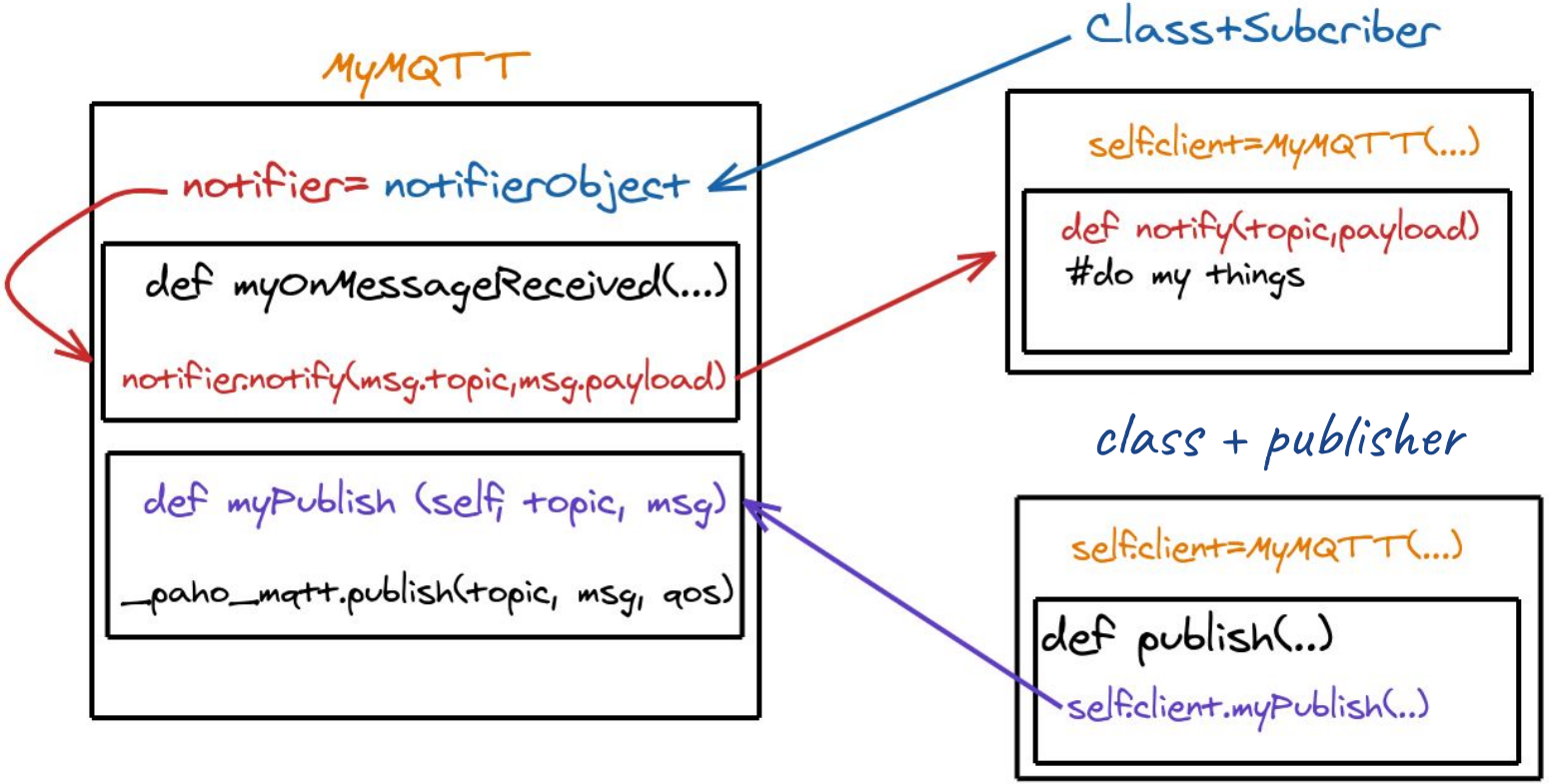
```
class MySubscriber:
        def __init__(self, clientID,topic,broker):
                self.clientID = clientID
                # create an instance of paho.mqtt.client
                self._paho_mqtt = PahoMQTT.Client(clientID, True)
                # register the callback
                self._paho_mqtt.on_connect = self.myOnConnect
                self._paho_mqtt.on_message = self.myOnMessageReceived
                self.topic = topic
                self.messageBroker = broker
        def start (self):
                #manage connection to broker
                self._paho_mqtt.connect(self.messageBroker, 1883)
                self._paho_mqtt.loop_start()
                # subscribe for a topic
                self._paho_mqtt.subscribe(self.topic, 2)
        def stop (self):
                self._paho_mqtt.unsubscribe(self.topic)
                self._paho_mqtt.loop_stop()
                self._paho_mqtt.disconnect()
        def myOnConnect (self, paho_mqtt, userdata, flags, rc):
                print ("Connected to %s with result code: %d" % (self.messageBroker, rc))
        def myOnMessageReceived (self, paho_mqtt , userdata, msg):
                # A new message is received
                print ("Topic:'" + msg.topic+"', QoS: '"+str(msg.qos)+"' Message: '"+str(msg.payload) + "'")
```

General MQTT Client: Why?

Considering the implementations above, everytime we want to add MQTT capabilities to a class, we need to write the same code to define all the functions that an MQTT client needs. We would like to have a smarter way to do that: we would like to have a *General purpose MQTT* client that we can always reuse without the need to copy-paste code.



General MQTT client: the idea



General MQTT Client: Code MyMQTT

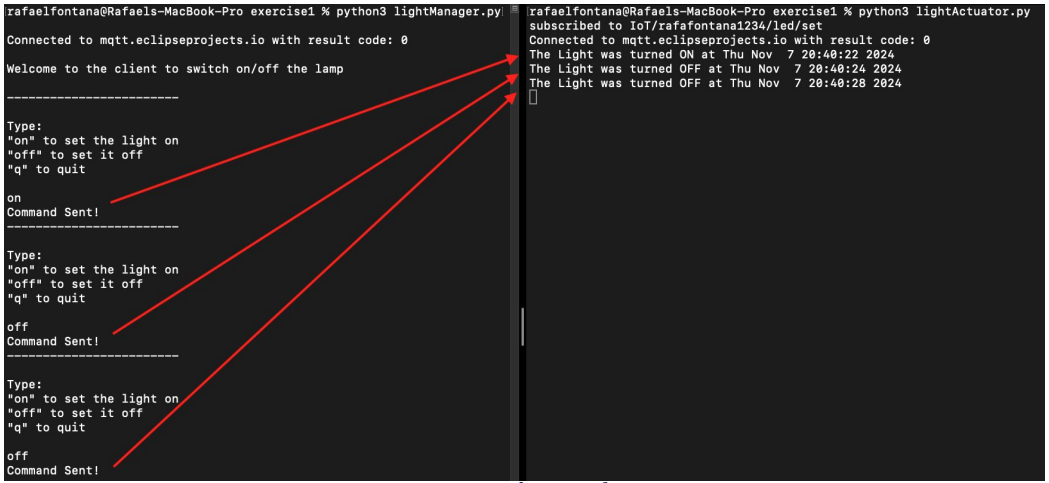
```python
import paho.mqtt.client as PahoMQTT

class MyMQTT:
    def __init__(self, clientID, broker, port, notifier=None):
        self.broker = broker
        self.port = port
        self.notifier = notifier
        self.clientID = clientID
        self._topic = ""
        self._isSubscriber = False
        # create an instance of paho.mqtt.client
        self._paho_mqtt = PahoMQTT.Client(clientID, False)
        # register the callback
        self._paho_mqtt.on_connect = self.myOnConnect
        self._paho_mqtt.on_message = self.myOnMessageReceived
    def myOnConnect (self, paho_mqtt, userdata, flags, rc):
        print ("Connected to %s with result code: %d" % (self.broker, rc))
    def myOnMessageReceived (self, paho_mqtt , userdata, msg):
        # A new message is received
        self.notifier.notify (msg.topic, msg.payload)
```

```python
def myPublish (self, topic, msg):
    # if needed, you can do some computation or error-check before publishing
    print ("publishing '%s' with topic '%s'" % (msg, topic))
    # publish a message with a certain topic
    self._paho_mqtt.publish(topic, json.dumps(msg), 2)
def mySubscribe (self, topic):
    # if needed, you can do some computation or error-check before subscribing
    print ("subscribing to %s" % (topic))
    # subscribe for a topic
    self._paho_mqtt.subscribe(topic, 2)
    # just to remember that it works also as a subscriber
    self._isSubscriber = True
    self._topic = topic
def start(self):
    #manage connection to broker
    self._paho_mqtt.connect(self.broker , self.port)
    self._paho_mqtt.loop_start()
def stop (self):
    if (self._isSubscriber):
        # remember to unsuscribe if it is working also as subscriber
        self._paho_mqtt.unsubscribe(self._topic)
        self._paho_mqtt.loop_stop()
        self._paho_mqtt.disconnect()
```

Exercise 1

Create a script (`lightActuator.py`)that mimics a light which is an MQTT subscriber for the topic *`IoT/<your-name>/led/set`* . The light status can be **on/off**. Then, create a client (`lightManager.py`) that uses MQTT to set the status of the light from the terminal. Use the SenML format for the MQTT payload.

In this case you will need to run both the script to make it work properly. In case you're connected to the Polito's network you need to run in on the same pc, otherwise this restriction does no apply.



Exercise 2

Extend Exercise 1 in such a way that `lightActuator` MUST check and compare the status of the light and the received MQTT command. If the status of the light and the received command are the same (e.g. command to set light ON, but light was already ON), the `lightActuator` will send an MQTT alert under the topic *`IoT/<your-name>/led/alert`* to the `lightManager`. The `lightManager` must be able to receive this alert (MQTT subscriber) and print it.

Therefore, both `lightActuator` and `lightManager` will work both as MQTT publishers and subscribers.

Exercise 3

Try to improve the previous exercise by creating a REST client to set the status of the light. You can use the file *index.html* as page for the GET request. When you will click on the button the page will execute a PUT request where the uri indicates the status we want to set (i.e. http://localhost:8080/on).

So, we need to create a web service able to handle a GET request and a PUT request:

- GET should return the `index.html`
- PUT should send an MQTT message to the proper topic with the status indicated in the URI of the request

Exercise 4

Create a client that collects the data coming from a group of temperature and humidity sensors that are on a building ***"IoT Project"*** .The simulated data is published using the script `sensors.py` . The building has 5 floors (from 0 to 4), with 3 rooms on each floor and one sensor in each room (5 x 3 -> 15 sensors in total). Each sensor publishes data using a topic as follows:

```
<yourName>/buildingID/floorID/roomID/sensorID
```

For example, the sensor on room 2 in the 4th floor will publish on the following topic:

```
rfontana/IoT_project/4/2/DH_311
```

We want to create a client that give the possibility to choose how to retrieve data according to three options:

- Data from all the sensors of the building
- Data from all the sensors on a single floor
- Data from a sensor in a particular room

If you feel confident enough, you can try to give the user the possibility to change his idea and change what he wants to monitor *on the fly*.

It may be needed to reduce the number of simulated floors and rooms since the broker could prevent too many clients connecting from the same PC.