ESERCIZIO 1

```
def myOnMessageReceived(self, paho_mqtt, userdata, msg):
    print(f"{self.clientID}"+"Topic: "+msg.topic+" QoS: "+str(msg.qos)+" Message: "+str(msg.payload))
```

```
if __name__=="__main__":
    mosquitto_pub=MyPublisher("publisher", "test.mosquitto.org")
    mosquitto_pub.start()
    mosquitto_sub_1=MySubscriber("subscriber_1", "test.mosquitto.org", "tiot/6/topic1")
    mosquitto_sub_1.start()
    mosquitto_sub_2=MySubscriber("subscriber_2", "test.mosquitto.org", "tiot/6/topic2")
    mosquitto_sub_2.start()
    mosquitto_sub_3=MySubscriber("subscriber_3", "test.mosquitto.org", "tiot/+")
    mosquitto_sub_3.start()
    mosquitto_sub_4=MySubscriber("subscriber_4", "test.mosquitto.org", "tiot/#")
    mosquitto_sub_4.start()

    mosquitto_pub.myPublish("tiot/6/topic1")
    mosquitto_pub.myPublish("tiot/6/topic2")
    mosquitto_pub.myPublish("tiot/6")
    mosquitto_pub.myPublish("tiot/6/topic3")

    mosquitto_pub.stop()
    mosquitto_sub_1.stop()
    mosquitto_sub_2.stop()
    mosquitto_sub_3.stop()
    mosquitto_sub_4.stop()
```

Per esercizio, vogliamo usare le funzioni della libreria paho.mqtt.client per far comunicare un Publisher e tanti Subscriber attraverso il paradigma di comunicazione publish/subscribe, con un Message Broker interposto tra le due entità. Il Publisher invierà informazioni sotto un certo topic al Broker, lui li inoltrerà a tutti i Subscriber che si sono precedentemente iscritti a quel topic (un Subscriber può iscriversi a più topic usando le wildcards).
Per implementare MyPublisher e MySubscriber, sono state usati i codici standard visti a lezione, senza nessun cambiamento se non quello di stampare, nella funzione di callback *myOnMessageReceived*, il proprio clientID ogni volta che un Subscriber riceve un messaggio, per identificarsi (riportiamo solo il codice di quella funzione).
Nel main definisco il Publisher e 4 Subscribers, considerando un po' di combinazioni del topic sotto cui pubblico i dati, in modo che a ogni *publish()* certi Subscriber ricevano i contenuti a loro destinati.

ESERCIZIO 2

Obiettivo: realizzare un MQTT subscriber per ricevere i valori di temperatura inviati dall'Arduino.

```
mqtt.begin("test.mosquitto.org", 1883);
//SOTTOSCRIZIONE
message["deviceID"] = 123;
message["resources"] = "temperature";
message["end-points"] = "tiot/6/devices/temperature";
serializeJson(message, output);
mqtt.publish("tiot/6/device", output);


float temp = readTemp();
String message = senMLEncode("temperature", temp, "Cel");
mqtt.publish("tiot/6/devices/temperature", message);
```

Il codice di questo esercizio si sviluppa su più fronti:
1. Arduino: nella funzione *setup()* avviene la sottoscrizione al Catalog, tramite la funzione *mqtt.publish()* che prende come parametri il topic per l'iscrizione al Catalog e il messaggio comprendente il device da collegare.
Mentre nella *loop()* viene ripetutamente misurato il valore di temperatura corrente e pubblicato sul broker in un formato senML così come descritto nei laboratori precedenti (funzione *readTemp()* e *senMLEncode()*);

```
def myOnMessageReceived (self, paho_mqtt , userdata, msg):
    print ("Topic:'" + msg.topic+"', QoS: '"+str(msg.qos)+"' Message: '"+str(msg.payload) + "'")
    payload = json.loads(msg.payload)
    new_device = {
            "deviceID": payload["deviceID"],
            "end-points": msg.topic,
            "resources": payload["resources"],
            "timestamp": time.time()
        }
    DEVICES.append(new_device)
```

2. Catalog: come già spiegato per l'esercizio 5 del laboratorio precedente, alla ricezione di un messaggio sul topic, col quale si è sottoscritto al message broker, il Catalog memorizza il device nella lista di dizionari DEVICES;

```
requests.put("http://localhost:8080/addService", json.dumps(payload))
response = requests.get("http://localhost:8080/deviceID/123")
our_device = response.json()
endpoint = our_device["end-points"]
print(endpoint)

#dato che è un subscriber, non può pubblicare -> per ricavare il broker
#richiesta GET
broker = requests.get("http://localhost:8080/broker")
our_broker = broker.json()
our_broker2 = our_broker["broker"]
print(our_broker2)

test2 = MySubscriber("MySubscriber", endpoint, our_broker2)

test2.start()
time.sleep(60)
```

3. MQTT subscriber: questo codice permette di invocare i servizi forniti dal web server al fine di:
- registrarsi, attraverso la *requests.put()*. Il Subscriber corrisponderà a un nuovo servizio, le cui informazioni vengono salvate sul Catalog;
- ricavare le informazioni riguardanti il message broker a cui far riferimento, attraverso la *request.get()*;
- ricavare l'endpoint sul quale sottoscriversi per ricevere le informazioni pubblicate dall'arduino Yun.

Tutte queste operazioni devono essere svolte utilizzando la parte RESTful del Catalog, perché per definizione un Subscriber MQTT non può mandare informazioni, ma solo riceverle, se si fa riferimento esclusivamente a questo paradigma. Per invocare i servizi forniti dal Catalog, come visto, si deve ricorrere a richieste http.
Ci si sottoscrive poi al broker, dopo aver ottenuto il suo identificativo, e si attende che la Yùn Arduino pubblichi i valori di temperatura misurati.

ESERCIZIO 3:
Vogliamo realizzare un MQTT publisher per inviare comandi di attuazione per cambiare lo stato del led gestito da Arduino.
Anche in questo caso il codice si sviluppa su più fronti:

```
mqtt.begin("test.mosquitto.org", 1883);
//SOTTOSCRIZIONE AL CATALOG
message["deviceID"] = 123;
message["resources"] = "led";
message["end-points"] = "tiot/6/device/led";
serializeJson(message, output);
mqtt.publish("tiot/6/device", output);
mqtt.subscribe("tiot/6/device/led", setLedValue);


if(doc_rec["e"][0]["n"] == "led"){
  if(doc_rec["e"][0]["v"] == 1 || doc_rec["e"][0]["v"] == 0){
    digitalWrite(LED_PIN, doc_rec["e"][0]["v"]);
  }
```

1. Arduino: la Yùn deve ricevere comandi di attuazione per gestire l'accensione o lo spegnimento del led. Per fare ciò si registra come nuovo device al Catalog tramite la funzione *mqtt.publish(),* dicendo che il led sarà la risorsa che gestisce. Dopodiché si sottoscrive al messageBroker su un altro topic, nel quale riceverà i valori per settare lo stato del led. Precisamente, alla ricezione di un messaggio viene invocata la funzione di callback *setLedValue()*, la quale controlla il formato del messaggio e, se corretto, modifica lo stato del led stesso;

```
def myOnMessageReceived (self, paho_mqtt , userdata, msg):
    print ("Topic:'" + msg.topic+"', QoS: '"+str(msg.qos)+"' Message: '"+str(msg.payload) + "'")
    payload = json.loads(msg.payload)
    new_device = {
            "deviceID": payload["deviceID"],
            "end-points": msg.topic,
            "resources": payload["resources"],
            "timestamp": time.time()
        }
    DEVICES.append(new_device)
```

2. Catalog: come già spiegato per l'esercizio 5 del laboratorio precedente, alla ricezione di un messaggio sul topic, col quale si è sottoscritto al message broker, il Catalog memorizza il device nella lista di dizionari DEVICES;

```
while True:
        ledState = not(ledState)
        senML = {
                "bn": "Yun",
                "e": [{
                    "n": "led",
                    "u": "",
                    "v": int(ledState)
                }]
            }


        test.myPublish(json.dumps(senML))
        time.sleep(60)
```

3. MQTT Publisher: il publisher permette di invocare i servizi forniti dal Catalog al fine di:
- registrarsi, attraverso la *requests.put()*;
- ricavare le informazioni riguardanti il Message Broker da usare, attraverso la *request.get()*;
- ricavare l'endpoint sul quale pubblicare i comandi di attuazione.

Per semplicità abbiamo deciso di spegnere/riaccendere il led periodicamente, ma si poteva anche attendere l'input da parte dell'utente per inviare il settaggio che l'utente stesso preferiva.

ESERCIZIO 4:
Si vuole realizzare uno Smart Home Controller remoto per implementare le stesse funzionalità dell'Esercizio 2.1 Lab Hardware – part 2. Per farlo dovrà lavorare da Publisher e da Subscriber. Anche questo codice si sviluppa su più fronti, vediamoli uno per volta:

```
if __name__=="__main__":

    #sottoscrizione al Catalog come nuovo servizio
    payload = {
        "serviceID": 789,
        "description": "Smart Home Controller"
        }

    requests.put("http://localhost:8080/addService", json.dumps(payload))

    #endpoint usato da Arduino per comunicare
    response = requests.get("http://localhost:8080/deviceID/123")
    our_device = response.json()
    endpoint = our_device["end-points"]
    print(endpoint)

    #dato che è un subscriber, non può pubblicare
    #per ricavare il broker facciamo una richiesta GET
    broker = requests.get("http://localhost:8080/broker")
    our_broker = broker.json()
    our_broker2 = our_broker["broker"]
    our_broker3 = our_broker["port"]
    print(our_broker2)

    endpoint2="tiot/6/device"
    test=MyMQTT("MySubscriber2", our_broker2, 1883)
    test.start()
    test.mySubscribe(endpoint2)
```

1. MQTT Publisher/Subscriber: per implementarlo abbiamo usato la classe MyMQTT discussa a lezione, che permetterà di ottenere entrambe le funzionalità del paradigma. Attraverso REST (funzioni della libreria requests) abbiamo aggiunto nel Catalog le informazioni del nuovo servizio, il Controller. Come visto più volte, senza riportare nuovamente il codice, il Catalog aggiungerà il servizio come nuovo item della lista di dizionari SERVICES.

Attraverso una richiesta GET abbiamo ricavato l'endpoint a cui ascolta la Yùn Arduino, il message Broker e il device iscritto al Catalog, con il formato e le modalità già viste in precedenza. Dal momento che "tiot/6/device" è, invece, l'endpoint a cui Arduino pubblica, allora mettiamo il Controller in ascolto in corrispondenza di esso.

```
while True:
    user_input = input("""Cosa vuoi fare?
    lcd: Stampa qualcosa nell'lcd
    temp: Per azionare il motore
    led: Settare il led in base alla temperatura\n""")
    senML = {
        "bn": "Yun",
        "e": [{
            "n": str(user_input),
            "u": "",
            "v": ""
                }]
            }

    if (user_input == "lcd"):
        senML["e"][0]["v"] = input("Cosa vuoi stampare?")
    if (user_input == "temp"):
        senML["e"][0]["v"] = input("Valore di temperatura: ")
    if (user_input == "led"):
        senML["e"][0]["v"] = input("Valore di temperatura: ")

    test.myPublish(endpoint, json.dumps(senML))
```

A questo punto siamo pronti a creare il nostro Controller, chiamato qui test, che si sottoscrive al topic a cui Arduino manderà i dati che processa; permettiamo allora all'utente di decidere quale funzionalità fornita da Arduino voglia realizzare, attraverso un semplice comando user-friendly, e creiamo un dizionario in locale per mandare ad Arduino l'input in conformità al formato che si aspetta di ricevere; l'utente fornisce i dati, noi definiamo il formato e lo inviamo ad Arduino come documento JSON. Questo comando richiamerà la funzione desiderata, definita nello sketch Arduino.

```
  //SOTTOSCRIZIONE AL CATALOG
  message.clear();
  message["deviceID"] = 123;
  message["resources"] = "led";
  message["end-points"] = "tiot/6/device/sensors";
  String output;
 serializeJson(message, output);
 Serial.println(output);

  mqtt.publish("tiot/6/device", output);
  mqtt.subscribe("tiot/6/device/sensors", setValue);
  analogWrite(FAN_PIN, (int)current_speed);
  analogWrite(RLED_PIN, (int)brightness);
  attachInterrupt(digitalPinToInterrupt(PIR_PIN), checkPresence, CHANGE);
}
void setValue(const String& topic, const String& subtopic, const String& message){

  DeserializationError err = deserializeJson(doc_snd, message);
  if(err){
    Serial.println(F("DeserializedJson() failed with code "));
    Serial.println(err.c_str());
  }

  if(doc_snd["e"][0]["n"] == "led"){
      led(doc_snd["e"][0]["v"]); //doc_rec = val di temperatura
    }
  if(doc_snd["e"][0]["n"] == "lcd"){
    lcd.clear();
    lcd.print(doc_snd["e"][0]["v"].as<String>());
    }

   if(doc_snd["e"][0]["n"] == "temp"){
    dcMotor(doc_snd["e"][0]["v"]);
    }

 }

void loop() {
  //SENSORE DI TEMPERATURA
  int sig = analogRead(TEMP_PIN);
  float R = ((1023.0 / (float)sig) - 1.0) * R0;
  float log_sig = log(R / R0);
  float T = 1 / ((log_sig / B) + (1 / 298.15));
  float temp = T - 273.15;
  Serial.println("Temperature now: " + String(temp)); //stampa di verifica
  String message = senMLEncode("temperature", temp, "Cel");
  mqtt.publish("tiot/6/device", message);
  Serial.println(message);


  //SENSORE DI RUMORE
  int value = digitalRead(SND_PIN);
  if (value == LOW) {
      String message = senMLEncode("presence", 1, "");
      mqtt.publish("tiot/6/device", message);
  }
delay(2000);
}
```

2. Arduino: la nostra Yùn dovrà innanzitutto iscriversi come nuovo servizio al Catalog; per farlo, nella *setup()* definiamo il dizionario nel solito formato, inserendo i dati relativi alla sottoscrizione come nuovo device, tra cui l'endpoint a cui la Yùn è messa in ascolto, e subito il Catalog lo conosce ed è in grado di fornirlo al Controller, quando lo richiede.
Pubblicata la sottoscrizione al Catalog, Arduino fungerà anche da Subscriber, in quanto aspetta di ricevere un comando per fornire le attuazioni della Smart Home, il comando inviato dal Controller secondo i desideri dell'utente che abbiamo definito poco prima. Quando la Yùn riceverà un'informazione via MQTT, farà partire la funzione di callback *setValue().*

Lei, ottenuto il documento JSON in arrivo dal Controllore, lo deserializza e controlla quale input ha dichiarato l'utente; in base a esso richiama una delle tre funzioni interne, di cui per semplicità non si riporta il codice, già definite nei laboratori HW precedenti, che richiedono i comandi di attuazione ricevuti via MQTT ed eseguono le azioni corrispondenti.
Come richiesto dal testo, inoltre, la Yùn pubblica periodicamente i valori di temperatura rilevati e informazioni sulla presenza di persone tramite il sensore di rumore, sempre al topic al quale il Controllore è in ascolto.

SMART HOME CONTROLLER: EMBEDDED vs. HOME CLOUD:

Abbiamo implementato, in due diversi laboratori, la versione embedded e quella remota del nostro Smart Home Controller; entrambe sono valide e funzionanti, ma ai fini pratici la soluzione embedded può rivelarsi più macchinosa e meno efficiente della home cloud.
La prima, infatti, richiede che inviamo comandi di attuazione ad Arduino e riceviamo da lui informazioni senza un mediatore esterno, ma in modo diretto, mentre la home cloud organizza l'ambiente IoT della nostra abitazione secondo precise gerarchie: usiamo il paradigma di comunicazione MQTT per avere un'entità centrale, il Message Broker, attraverso cui passano tutti i comandi di attuazione e le informazioni di temperatura e presenza di persone processate da Arduino.
Questo potrebbe rivelarsi uno svantaggio nel caso in cui dovessero esserci malfunzionamenti della rete locale, ma se in buona approssimazione possiamo considerarli trascurabili, ci rendiamo conto che la soluzione home cloud porta molti vantaggi, soprattutto nel caso in cui il nostro environment IoT sia composto da molte Yùn Arduino, magari una per stanza. In una situazione del genere, molto probabile, una infrastruttura come quella cloud ci permette di raccogliere in un punto centrale il controllo dell'attuazione e la gestione e il processamento delle informazioni, permettendoci facilità di gestione della nostra Smart Home, che potrà essere controllata anche da remoto, semplicemente collegando a internet il flusso di dati, facendolo passare per il nostro gateway.