COMPUTER ENGINEERING PROJECT II

TUTORIAL 6: Python client for monitor and control

Last revision: March 8th, 2021

1 INTRODUCTION & MOTIVATION

In the previous tutorials, Zigbee2Mqtt (Z2M) was presented and you should now be able to monitor and control Zigbee devices with it. Yet, the interaction between these has to be orchestrated by another software that interacts with Z2M. MQTT Explorer and the Mosquitto clients were used to demonstrate how to interact with Z2M, but these clients are intended to be used by a developer for debugging purposes, not for production grade code. For this, we need a full stack programming language which will allow us to combine business code (algorithms, state machines, technical integrations).

In this tutorial a software application, Cep2App is presented. The application performs simple controller actions (roughly corresponding to use cases in an actual application), such as turn lights (and/or appliances) on or off, based on the events it receives from the Z2M monitoring sensors. Some specifics of the application are also addressed here: encoding and decoding JSON and threads in Python.

IMPORTANT: this document is just a part of the documentation of the Cep2App. The code is heavily commented and you should refer to it for more specific details of the implementation. Here, only a high level view of the application is presented. In another words, you should see this document as a map to start navigating in the code.

2 WHAT TO READ BEFORE THE TUTORIAL?

It is expected that you have a basic knowledge of the following topics:

- What Zigbee2Mqtt is and how do other services interact with it (addressed in the previous tutorials);
- The model-view-controller design pattern [1].

3 MATERIALS

You will need the following materials:

- Raspberry Pi with Z2M and the Mosquitto broker installed
- Python 3
- Cep2App on GitHub: https://github.com/jmiranda-au/cep2app

4 THE CEP2APP

The Cep2App is a Python demo application that processes events published by Z2M and communicates with a remote web service via HTTP. Specifically, when sensor events are received (for example, motion sensors), the application changes the state of the actuator devices (for example, a LED light or a power plug) and records these events on the remote web service (here named Cep2WebService).

This application was designed to be executed in the Raspberry Pi, along with Z2M – naturally, a MQTT broker is needed to intermediate these two clients. A simplified sequence diagram that represents this high level behavior is depicted in Figure 1.

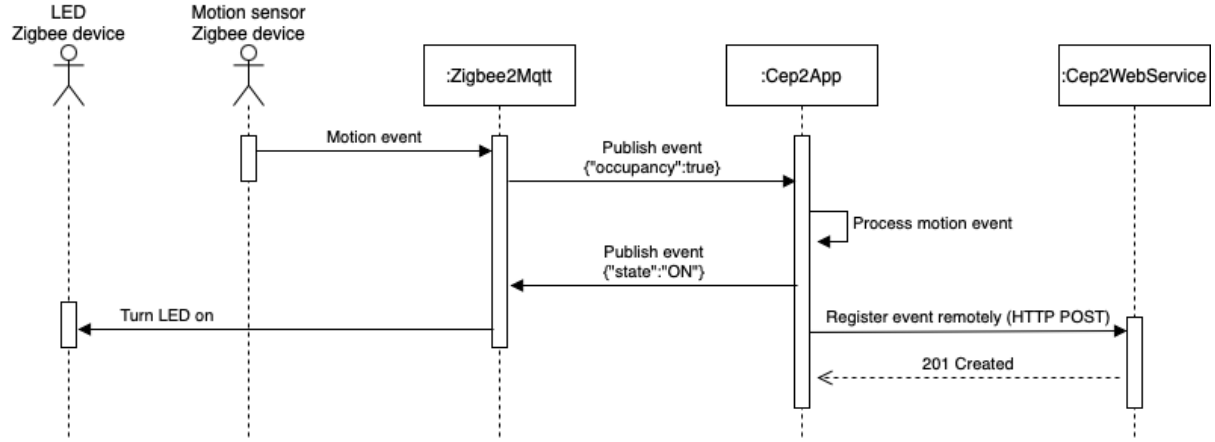

Figure 1. Sequence diagram representing the interaction of Cep2App with Z2M and a Cep2WebService.

The class diagram of the Cep2App's code is depicted in Figure 2. The architecture of the application is loosely based in the Model-View-Controller (MVC) design pattern [1]. In short, the pattern is built around three classes: the **model** class structures and provides methods for managing the application's data; the **view** class provides an interface (for example, a GUI) for handling users inputs and outputs with the application; and the **controller** class mediates the interactions between the model and the view classes. The first design of the pattern expected that the model and view would communicate directly with each other; in more recent approaches, the view and model classes never interact directly with each other. The last is the approach used in Cep2App, i.e. interactions by external actors are received in the view and handed to the model by the controller. Naturally, outputs from the model are passed to the view by the controller. It is important to note that although these three classes are represented as three entities, they might be constituted by more than one class.

As depicted in Figure 2, the application has a controller (Cep2Controller) and a model (Cep2Model) classes. A view is not required, since the application does not have a user interface (this is further discussed in section 6). The Cep2Controller class interacts with Z2M using an instance of Cep2Zigbee2mqttClient and is also able to interact with the Cep2 web service using the Cep2WebClient.

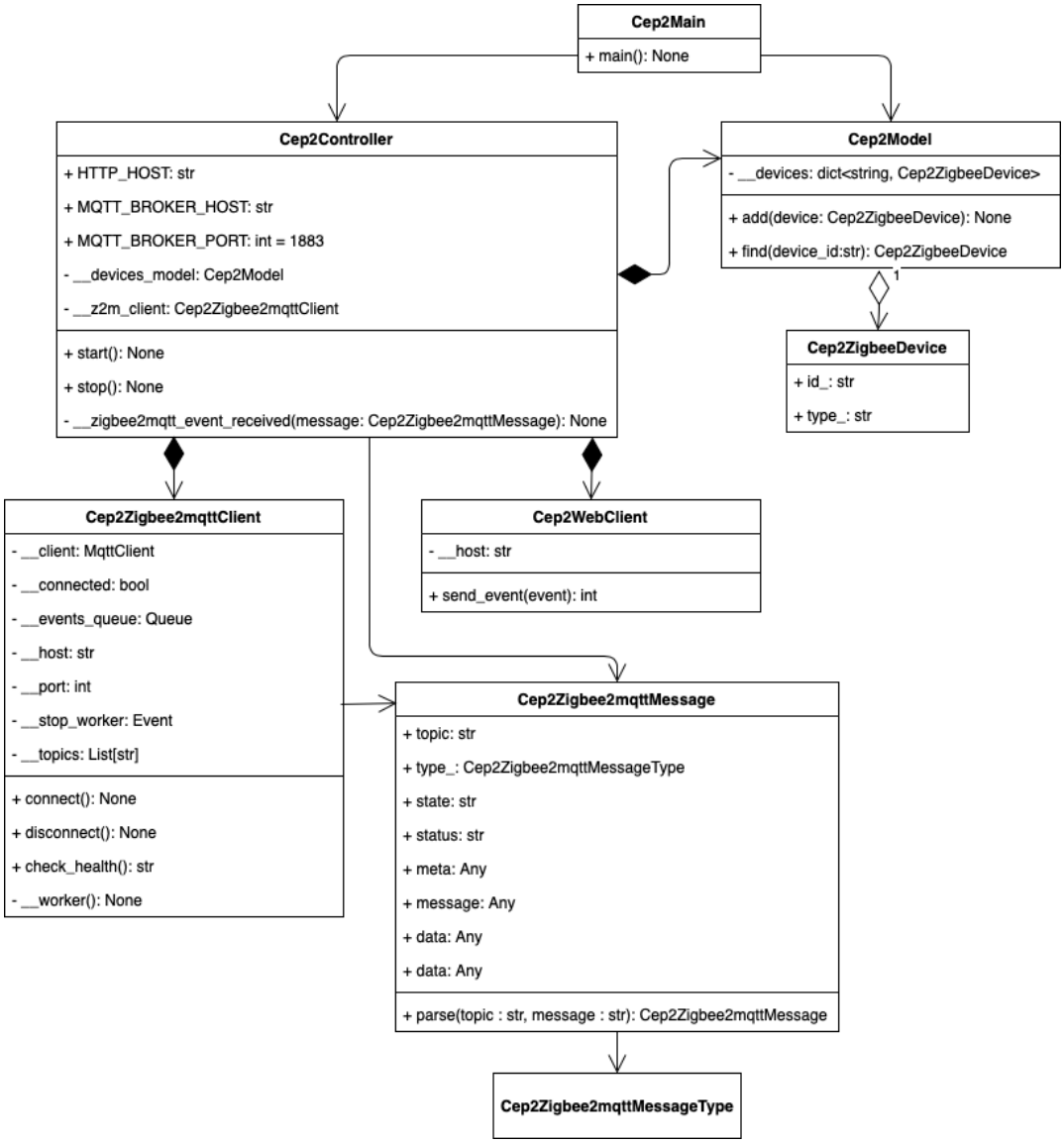

Figure 2. Class diagram for the Cep2App.

The Cep2Zigbee2mqttClient has a subscriber for Z2M events (it subscribes to the default topic `zigbee2mqtt/#`), and it also provides methods for publishing to Z2M. As written, this class is instantiated by the controller, which also assigns to it a callback method, `__zigbee2mqtt_event_received()`, that is called when events from Z2M are received by the subscriber. This callback is where the business logic of the application is handled. In the case of Cep2App, the Z2M sensor event is parsed, then it changes state of the actuators (if necessary) and registers an event in the web service. More logic can be added here, but caution should be taken, since blocking this call might lead to loosing messages due to, for example, high processing time of data, or blocking communication with remote services.

The Cep2Model class only structures data relative to the sensors. In the demo application, this class only contains instances of Cep2ZigbeeDevice that represent the Zigbee devices that

the application should accept events from, or actuate over. This model assumes that all devices are placed in the same physical room. This class can be extended to support database communication, or even communicate with remote data storage services.

The Cep2App model has one PIR (motion) sensor, and one LED and one power plug actuators. The application's business logic is very simple: when motion is detected by a motion sensor (Z2M sends either an occupancy event with true or false when motion is detected or not), the LED and power plugs are turned on (motion detected) or off (motion not detected).

4.1 Running Cep2App

To execute the application, you can run it using the command `python3 Cep2Main.py`. The application assumes that an instance of Z2M is running and checks it when the application starts, specifically when Cep2Controller's start() is called. This method uses the Z2M's health check topic to check the current Z2M state.

The application also assumes that a web service is running. For test purposes, you can the command `python3 -m http.server 8000` to simulate a web service – the implemented request will return an HTTP 501 status code, since the server only supports GET requests and the Cep2WebClient does a POST. Also important: if you change the application port, don't forget to change the port in the command.

5 PYTHON TIPS AND TRICKS

In this section some Python specifics are addressed. These are also addressed throughout the code and you should refer to the its comments for further information. Here, they are presented in a generic form.

5.1 Decoding from and encoding to JSON

Encoding and decoding are important to understand, as a distributed system has to send data using a transport protocol (e.g. TCP/IP) between networked computers and programs who might not understand data in the same manner. Also, the byte stream used by TCP/IP and other protocols does not match well with the data structures used in most programming languages. Thus, encoding and decoding are part of the processes of "serializing" data structures into a byte stream and deserializing them back. The choice of an encoding format is highly relevant to cover already during the high-level design, as choosing a non-heterogenous format will make it hard to use other programming languages. With MQTT we often choose JSON encoding as this maps very well with lightweight type of clients often using MQTT.

In Python the decoding from and encoding to JSON can be made using the json module [2]. To decode a JSON string, the following code can be used:

```
import json

try:
       payload = '{"ocuppancy":true, "linkquality":51}'
       # Decode a JSON string to a Python object.
       json_obj = json.loads(payload)
       print(f"occupancy = {json_obj['occupancy']}")
except JSONDecodeError:
       # There was an error parsing the JSON message. Here, it is ignored.
       pass
except (KeyError, TypeError):
      # An exception occurred because either the occupancy key is not in the
      # object (KeyError) or the json_obj is not a dictionary (TypeError; the
      # object might be a list).
      print("Could not understand the JSON event")
```

To encode JSON, i.e. from a Python object to a JSON string, the following code can be used:

```
      # A Python dictionary with two keys
      payload = {"ocuppancy":true, "linkquality":51}
      # Encode to a JSON string
      json_str = json.dumps(payload)
      print(json_str)
```

5.2 Working with threads

In this section a very brief introduction on threads in Python is presented. For this, a simple implementation of the producer-consumer problem [3] is presented below, with one producer pushing messages into a queue and multiple consumers (running in threads) pulling messages from it.

The code is commented and you should refer to these as your documentation to understand the program. The most important parts of the implementation are marked in bold, specifically the thread creation (instantiate a Thread object) and the queue push (put) and pull (get). For more information on threads and queues in Python refer, respectively, to [4] and [5][1].

Also important is the use of an event [4] (`stop_consumer`) to indicate that the program should stop. This is similar to a boolean variable, in the sense that it can be set (true) or not (false), but it implements concurrency mechanisms so that no concurrency issues occur when multiple threads try to change flag's state. After all messages have been pushed into the queue, the producer sets the event to indicate that the consumer function must stop.

```
from queue import Empty, Queue
from threading import Event, Thread, current_thread
from time import sleep

# Queue where the messages will be pushed. This object already implements internal
# concurrency mechanisms (for example, a mutex).
message_queue = Queue()
# Event that indicates the consumer to stop pulling messages from the queue.
stop_consumer = Event()
```

[1] in [5], another implementation of a producer consumer is also presented.

```
def consumer() -> None:
    # Run while the event is not set. This will be set after the producer stops
    # pushing messages to the queue.
    while not stop_consumer.is_set():
        # Pull message from the queue. After 1 second the get method times out and
        # raises an Empty exception. In this case, the exception is ignored.
        try:
            message = message_queue.get(timeout=1)
            # Get the current thread's name, only for display purposes.
            thread_name = current_thread().getName()
            # Display the pulled message and the name of the current thread.
            print(f"{thread_name} {message}")
        except Empty:
            # Ignore the exception, since the function will keep trying to
            # get new mesages from the queue.
            pass
    print(f"exit {current_thread().getName()}")

if __name__ == "__main__":
    threads = []

    # Create three threads. Each thread will run consumer().
    for i in range(0, 3):
        threads.append(Thread(target=consumer, daemon=True, name=f"t{i}"))
        threads[i].start()

    # Push messages to a queue every second. This is the only producer.
    for i in range(0, 10):
        message_queue.put(f"message number {i}")
        sleep(1)

    # Set the event so that the worker loop stops, exiting the function.
    stop_consumer.set()

    # Safely wait for the threads to finish.
    for i in range(0, 3):
        threads[i].join()
```

6 NEXT STEPS

The presented code and text presents some of the topics that you can address in your project and final report. Yet, by no means it is an extensive description of the software or the architecture it is part of.

The first limitation is the class diagram of Figure 2, which only presents the classes of the software without any placement on a layered architecture. Where could the presented classes be placed on this architecture type (presentation, business logic, model/service)?

Another diagram that is not presented is a deployment diagram, although a description of where Cep2App is expected to run, as well as Z2M and the web service, is given. This is an important diagram since another developer that might use your code needs to know where the different parts of the system are executed.

Finally, the HTTP request is a minimalistic (and incomplete) approach to a REST (Representational State Transfer) architecture. This is not addressed in this tutorial, since it is out of scope. But, if you implement a web service, this is a possible approach to it. At this point you can do some initial research to understand how Cep2App fits on such architecture and is communication on such architecture.

7 FURTHER READING

You can read more about Python threads and events in [6] and [7] and about JSON encoding in [8]. In [9] several Python non-basic topics are covered, which might be useful when more advanced features are presented to you.

8 REFERENCES

[1] The Evolution of MVC: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc

[2] Python json module: https://docs.python.org/3/library/json.html

[3] Producer-consumer problem: https://en.wikipedia.org/wiki/Producer–consumer_problem

[4] Python threading module: https://docs.python.org/3/library/threading.html

[5] Python queue module: https://docs.python.org/3/library/queue.html

[6] An Intro to Threading in Python: https://realpython.com/intro-to-python-threading/

[7] How To Make Python Wait: https://blog.miguelgrinberg.com/post/how-to-make-python-wait

[8] Working With JSON Data in Python: https://realpython.com/python-json

[9] The Hitchhiker's Guide to Python!: https://docs.python-guide.org