Simple Smart Home

Introduction

In this example, we'll design and implement a simple Smart Home setup. The main focus is on improving the designed communication channels and payloads over MQTT, and storage data structures on the edge server.

Imagine a home with multiple defined 'rooms' and a common edge server/router which can bidirectionally interact with each of the devices using an MQTT server running on the edge server. Each device can push to, and listen on, various topics on the MQTT server.

The underlying communication layer can be WiFi, bluetooth, etc. We obviously don't have to worry about that in this particular project, as you'll be running (simulating) all of them on your local machine. The edge server might also receive commands from consumers (you on your mobile app) which we'll simulate here by calling methods on the server from a main.py driver file. Please also assume that every component is trustworthy so there is no expectation of any authentication or access control.

Components

1. MQTT server: You should set up an MQTT server with no authentication, the ip and port could be specified in a config and read by all components, OR passed to all devices and edge server instances during initialization.

2. Rooms: Assume a predefined list of rooms in the home - Kitchen, BR1, BR2, Living. This list should be part of the server attributes and each device needs to specify mapping to one room, when it's created.

3. Device Type: Assume a predefined list of device types - LightDevice, ACDevice. This list should be part of the server attributes and each device needs to specify mapping to one type when it's created.

4. LightDevice: Multiple-intensity light (off/low/medium/high). It should be able to register with the server, change state to one of the four values based on command from the server, switch on/off, and respond to current status requests.

5. ACDevice: Simple AC with on/off switch, and temperature control (18 °C - 32 °C). It should be able to set any temperature directly based on command from the server keeping check on the bounds, it can send an error response on out of bounds requests.

It should also be able to register with the server, switch on/off, and respond to current status requests.

6. Edge server: It should be able to store the registered devices, accept new registrations, and send commands to the devices based on calls made to its external interface.

7. Driver: As this is a simulation, you should have a driver (main.py) which creates appropriate instances and initiates calls on the server to highlight the capabilities.

Program Organization

The zip file that you received has starter code available for device classes, edge server class and main.py to get you started. Please feel free to modify the variables, functions and calls as you prefer. You can add a README file to explain your basic structure and choices.

1. *main.py:* This is the driver code where you will be creating different devices and performing various operations related to the functionality of the devices. The main.py is covering all the cases mentioned in the problem statement.

2. *ACDevice.py:* This file will contain all the operations and methods relevant to the AC device_type. This will have the implementation of different methods such as registration, setting the temperature and fetching the temperature.

3. *LightDevice.py:* This file will contain all the operations and methods relevant to the Light_Device, device_type. This will have the implementation of different methods such as registration, setting the intensity of the light_device and fetching the current intensity status of the device.

4. *EdgeServer.py:* This is the main file which will get the command from the user to perform various operations for each of the devices. The different methods in EdgeServer class will help in making the communication with the devices to perform the task given by the user.

Lets understand the design of the topic topology for various types of communication in this project.

- Normally, you don't use the same topic to send information back-and-forth. Being publisher and subscriber on the same topic is not ideal due to unnecessary communication and lack of clarity.

- Imagine two extremes, one where each device has its own only one unique topic to publish and one to subscribe. In this case, logic to communicate is simple but the server gets loaded as for bulk calls, it has to write to, and read from, many topics.

- The other extreme is having only one topic to transmit and one to receive from the server. All devices can listen to that topic and only act on information that is for them, based on device id in the payload. This is very easy for the server but now each device is potentially getting a lot of unnecessary information to parse. It'll also hamper access control and make privacy separation among devices harder.

- We have implemented design topics in such a way that balances server and device load.

Housekeeping points

- This is a simulated minimal example and may not follow some standard practices.
- The focus is on the main flow, with relatively minimal error handling.

Problem Statement

We have implemented the demonstration of 3 lights and 2 ACs, spread across 3 rooms. Currently the communication related to the get operation is already implemented. Please demonstrate various calls to include all the command types below with different set input options.

1. Implement status and switch on/off commands
    a. Implement a get status command on the devices
        i. Complete the get_status method exposed into the main.py driver to initiate a status command. This is called based on any of the following - device id, device type (all devices of that type), room type (all devices in that room), 'all' (all devices in the home). This is already implemented.
        ii. The server is calling all relevant devices and getting status back from them. It can return to main.py after calling them. Data can be printed on the console directly once it receives statuses from various devices.
        iii. In each of the device classes, we have ***_get_switch_status, _get_temperature, _get_light_intensity*** method. Implement these methods so that data (the status of their dynamic values - light state, switch, temperature, etc.) can be published back and available in the main server.

    b. Implement commands to switch off/on devices, on the server and devices
        i. The ***set_status*** method that is exposed to the main.py driver to initiate switch on/off commands. This can be based on any of the following - device id, device type (all devices of that type), room type (all devices in that room), 'all' (all devices in the home)

ii. The server should appropriately call all relevant devices methods and get command status back from them. Relevant device class should have a method for getting the status. It can return to main.py after calling them. Data can be printed on the console directly once it receives statuses from various devices.

iii. Implement the set methods (***_set_switch_status (LightDevice), _set_switch_status (ACDevice)***) in each of the device classes to switch off/on method(s) to change their status, returning success or failure accordingly. Of course, 'off' here is equivalent to reducing electricity consumption but they'll continue to listen to new commands.

2. Implement control commands for both devices
    a. Implement light intensity control command - change to a specific value (off/low/medium/high)
        i. Implement the set method exposed to the main.py driver to initiate a light control command. This can be based on any of the following - device id, room type (all light devices in that room), 'all' (all light devices in the home)
        ii. The server should appropriately call all relevant devices and get command status back from them. Relevant device class should have a method for getting the status. It can return to main.py after calling them. Data can be printed on the console directly once it receives statuses from various devices.
        iii. Implement the given control command method ***_set_light_intensity*** in Light_Device class, implementing the change and returning success or failure accordingly.

    b. Implement AC control command - change the temperature between (18 °C - 32 °C)
        i. Implement the set method exposed to the main.py driver to initiate an AC control command. This can be based on any of the following - device id, room type (all AC devices in that room), all (all AC devices in the home)
        ii. The server should appropriately call all relevant devices and get command status back from them. Relevant device class should have a method for getting the status. It can return to main.py after calling them. Data can be printed on the console directly once it receives statuses from various devices.
        iii. Implement the given control command method ***_set_temperature*** in AC_Device class, implementing the change and returning success or failure accordingly.

Evaluation Rubric

Total Project Points: **240**

- Basic compilation without errors (**10%**) : **24 Points**
- Correctness:
    - Problem statement - 1.a (**20%**) : **48 Points**
    - Problem statement - 1.b (**30%**) : **72 Points**
    - Problem statement - 2.a (**20%**) : **48 Points**
    - Problem statement - 2.b (**20%**) : **48 Points**

Note

- **Minimum Requirements**: The final submission that you upload needs to have a successful compilation, at the least.
    - The code files, with initial shell code, related to ACDevice, LightDevice and EdgeServer are provided. You can go ahead and start implementing them after understanding the problem statement. Also, If you want to start from scratch you can do that.
    - You are supposed to write and submit code that should mimic the behavior of a smart home. Here devices can be registered on the server and communicate with it for various command executions. User or main can send commands on different parameter based devices, device type, room and home.
    - It is very important to ensure that you are using MQTT to perform the communication between server and devices.

- **Expected Submission Files:**
    - **ACDevice.py:** This file will contain the implementation for ACDevice type. It will contain methods to listen on various topics and then it will have implementation to perform get and set operations based on the received command.
    - **LightDevice.py:** This file will contain the implementation for LightDevice type. It will contain methods to listen on various topics and then it will have implementation to perform get and set operations based on the received command.
    - **EdgeServer.py:** This is a class that will invoke and have registration processing, get and set methods that will publish the requests for processing topics based on received commands.
    - **main.py:** This will be the driver code that should be invoked to perform all the commands related to registration and command execution.

- Expected sample output file is also provided in the zip folder. The file has multiple test cases that will help you in designing and understanding the different output for different commands. **It is not required to match the print statements provided in the sample**

output. Your goal should be to ensure that your code is covering at least these test cases.

Program Instructions

1. Download the zipped folder named **C03-Project-01-Simple-Smart-home.zip**, unzip it on your local machine, and save it. Go into the directory named **C03-Project-01-Simple-Smart-home**.
2. You will see different python programs as mentioned in the Program organization section. Edit and modify the files to solve the tasks. You will be frequently making modifications in server and devices class along with driver code (main.py).
3. Please include a README specifying how it should be run.
4. Make sure that you have Python 3.6, or higher, installed. At your command prompt, run:

```
$ python --version
Python 3.7.3
```

   If not installed, install the latest available version of Python 3.