Internet of Things (IoT)

Lab 2: Z-Wave

October 2020

Introduction

The goal of this Lab is to set up and manage a Z-Wave-based sensor and actuator network. You shall:

- Set up a Z-Wave network composed of a controller (master) device node, and two slave device nodes: a multi-function sensor and a light dimmer. The multi-function sensor can measure several quantities, such as temperature, humidity, etc., and can detect motion. The dimmer is a multi-level switch that allows to control a light bulb's brightness level. Slave device nodes are connected (paired) to a Z-Wave controller (see Section **IoT infrastructure**).

- Complete a Python API module which reads data from the sensor devices and commands the light dimmers. This API module is the backend of a RESTful Python application server which will be deployed on a Raspberry Pi (Figure 1).

- Test the application with the provided command-line-based RESTful client.

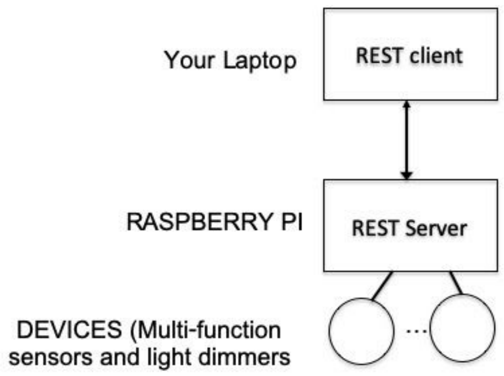

Figure 1 : Three-layer IoT infrastructure: IoT device layer, REST server layer (Raspberry Pi) and client application layer (your laptop)

Z-Wave basics

This lab uses the Z-Wave protocol to set up the IoT infrastructure described in Figure 1. A Z-Wave network is a *meshnet* that may include up to 232 nodes (extensible by *bridging*), and consists of two sets of nodes: controllers (masters) and slave devices. The Z-Wave protocol allows only one *primary* controller, but several *secondary* ones; we content ourselves with a simple flat mesh layout: one primary controller node and two slave non-controller nodes. A controller node handles its network by *including*, *excluding* and *querying/monitoring* its slave nodes. Data collection from slave nodes is made through the controller. Our controller of choice is the Z-Wave.Me “UZB” USB dongle (Figure 2): this device can be operated via API calls only.

Figure 2: the Z-Wave.Me “UZB” controller

How to connect a device to a Z-Wave network?

In order to add slave nodes to the network, the controller must be put into *inclusion* mode. Inclusion mode must be programmed through the *add_node* routine (see Annex 1). Before trying to connect a node, make sure that it is not already included: if in doubt, reset it.

Connecting sensors

Our sensor device of choice is the Aeotec Multisensor 6. Once plugged to a USB port, if its LED blinks green, it means that it is already paired to a controller. Please, reset it by pressing and holding its action button for 20s (Figure 3): its LED should glow continuously through a blend of colors.

Once the controller switches to inclusion mode (blue LED blinking on its casing), press once the action button of the node you want to add to the network. The sensor’s LED will start blinking green and the controller’s LED will stop blinking.

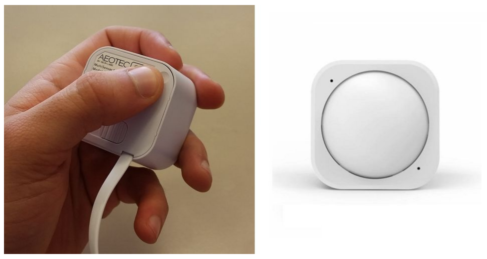

Figure 3: an Aeotec Multisensor 6.

Connecting dimmers

Our light bulb of choice is the Domitech dtA19-750-27. Start by plugging the dimmer in (bulb screwed in and AC cord connected to mains) and set its manual AC switch on: light should be on. To reset the dimmer, with the light on, operate its AC switch *off* and *on* 4 times within 4s: the dimmer will flash two times and the light should stay on at maximum brightness. Next, turn the dimmer off again.

With the controller into inclusion mode and the dimmer manually turned off, switch it on manually. If the inclusion is successful, the bulb will flash twice and stay on, while the controller LED stops blinking.

How to disconnect a device from a Z-Wave network?

In order to disconnect (remove) slave nodes from the network, the controller must be put into *exclusion* mode. Exclusion mode must be programmed through the *remove_node* routine (see Annex 1). Before removing a (recently added) node, make sure it has completed its “Query Stage” (see Section **“Node” class**); if in doubt, wait for a couple minutes.

Disconnecting sensors

When the controller is in exclusion mode (blue LED blinking on its casing), press once the action button of the node you want to remove from the network. The sensor’s LED will stop blinking green and start glowing with different colors; the controller’s LED will stop blinking.

Disconnecting dimmers

Turn the dimmer off manually using the AC switch. With the controller in exclusion mode, turn the light on manually using the AC switch. The bulb will flash twice and stay on; the controller’s LED will stop blinking.

Building your own IoT infrastructure

The lab’s IoT infrastructure is composed of one sensor, one dimmer device, a UZB controller device, a Raspberry Pi 4B and your laptop. The Raspberry is already installed and configured with the latest OpenZWave (OZW) system libraries and with the latest Python wrapper libraries. If you wish to install the whole software stack on your laptop, see here: https://gitedu.hesge.ch/lsds/teaching/master/iot/smart-building#other-gnulinux-distribution-support.

Set up your system like this:

1. Plug the UZB controller into one of the Raspberry’s USB ports.
2. Plug the Multisensor into one of the Raspberry’s (or your laptop’s) USB ports: this is just needed to feed power to the sensor, as it comes without batteries.
3. Plug the dimmer into a wall AC outlet.

Your Raspberry is configured to automatically connect to the predefined Wi-Fi AP “IoT-Labo” which will be available in the classroom; ask the instructor for the AP passphrase. The WLAN IP address of your raspberry Pi is `192.168.1.2x`, where ‘X’ is your assigned kit number. Please, **do not** modify the related configuration! If you need to work outside the classroom, you’ll probably have to connect a keyboard and a monitor to your Raspberry and set up an additional different network connection. You may also use the provided Ethernet cable: the default IP address is `192.168.1.1x`.

Interacting with your IoT platform

The Python Open Z-Wave software stack

Python-openzwave (Py-OZW) (http://openzwave.github.io/python-openzwave/) is an open source library which provides an API for the interaction with the controller connected to the Raspberry Pi; it is a wrapper around the system’s OZW library. Through Py-OZW we can retrieve data or send commands to the nodes (sensors and dimmers) included in a Z-Wave network. Python Open Z-Wave (Py-OZW) builds a software representation of the network by mapping the components of the Z-Wave network to corresponding software objects (Figure 5).

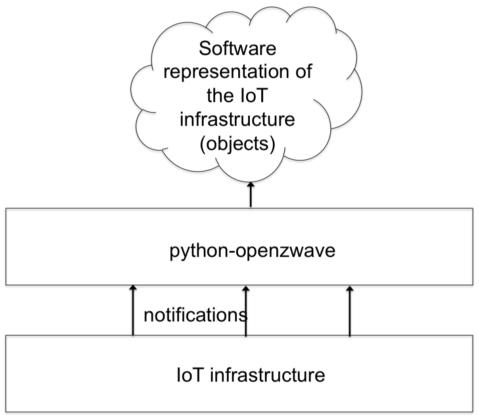

Figure 5: python-openzwave: Software (virtual) representation of the IoT infrastructure.

The software representation (objects) is updated through *notifications* from the *IoT infrastructure* layer which inform the *python-openzwave* layer of the occurrence of *events,* such as adding nodes, removing nodes, new nodes' readings, etc. (Figures 5-6). For each notification, Py-OZW sends signals to the end-user *Application* (Figure 6).

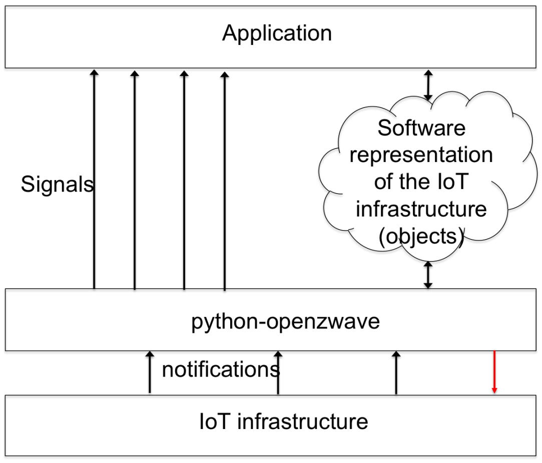

Figure 6: Signals are handled by the end user application

Here is the list of the most important classes in the software representation of the network.

"Network" class

The class `ZWaveNetwork` models the whole Z-Wave network. Here are some attributes and methods:

- `home_id_str` (type: string): unique identifier for the Z-Wave network.
- `is_ready` (type: bool): indicates if the network is ready or not. As soon as the network becomes ready, a `SIGNAL_NETWORK_READY` signal is generated by Py-OZW.
- `nodes_count` (type: int): number of nodes (including the controller) in the network.

- `controller` (type: ZWaveController): See Controller class.
- `nodes` (type: dict() of ZWaveNode objects): a dictionary of objects representing the nodes connected to the network (including the controller).
- `start()` (return type: None): this method creates the software representation of the Z-Wave network.
- `stop()` (return type: None): this method deletes the software representation of the Z-Wave network.

"Controller" class

The class `ZWaveController` models the network's controller. Here are few attributes and methods:

- `name` (type: string): controller's name.
- `device` (type: string): path of the device's driver (e.g. `/dev/ttyACM0`)
- add_node() (return type: bool): gets the controller in inclusion mode.
- `begin_command_add_device()` (return type: bool): gets the controller in exclusion mode.
- `cancel_command()` (return type: None): cancels any in-progress command, such inclusion and exclusion, running on the controller.

"Node" class

This `ZWaveNode` class models a node: sensor, actuator or controller. Hera are some node object's attributes and methods:

- `node_id` (type: int): unique identifier (ID) of the node in the network -- the controller node has ID = "1".
- `product_name` (type: str): commercial product's name. Aeotec sensor's product name can be "Multisensor 6 ..." and a dimmer's product name can be "ZE27". But often appears as "Unknown: type=..., id=...".
- `name` (type: str): name of the node. Usually empty at inclusion time, can be set by direct assignment.
- `location` (type: str): location of the node. Usually empty at inclusion time, can be set by direct assignment.
- `is_ready` (type: bool): tells if a node is ready to operate (QueryStage Completed).
- query_stage (type: str): indicates the state of a node. Once the Query Stage is at "Complete", the node object is ready to be used. More details: http://www.openzwave.com/dev/classOpenZWave_1_1Node.html#a8c740c962a9230d840984ca644e1f220
- `neighbors` (type: set() of int) : a set of the neighbouring nodes' IDs. Neighbours are nodes that can be reached by the current node.
- `values` (type: dict() of ZWaveValue): a dictionary of "value" objects keyed by unique integer IDs. A value object represents a measure or a configuration parameter of a node. You can use the method `get_values()` to retrieve only specific values.
- `set_config_param()` (return type: bool) : sets the value of a configurable parameter in a device.
- `get_battery_level()` (return type: int): retrieves the battery level of the node.

- `get_values (class_id, genre, type...)` (return type: set() of ZWaveValue): retrieves measures and/or parameters of a node from the dictionary `values` mentioned earlier. Parameters act as filters:
    - `class_id`: the class of the value (temperature, presence, battery level, etc.). The classes and their IDs are fixed by the Z-Wave specifications. For example, you would use `0x31` (COMMAND_CLASS_SENSOR_BINARY) to retrieve temperature, luminance and humidity, and `0x30` (COMMAND_CLASS_SENSOR_BINARY) for motion. To keep things simple, you can use "All" to retrieve values from all classes.
    - `genre:` discriminates between readings ("User") and setup values ("Config").
    - `type:` is the type of value, such as "String", "Int", etc. Note: these are libopenzwave.PyValueTypes, not the plain Python's ones.
    - `readonly, writeonly:` filter by read- or write-only values.

Note: Setting any parameter to "All" ignores the corresponding filter. Parameters can also be passed by keyword.

Example: to retrieve all sensor's readings (measurements):

```
get_values("All", "User", "All", True, False)
```

Example: to retrieve all Configuration and System parameters:

```
get_values(
        class_id="All", genre="All", type="All",
        readonly=False, writeonly="All"
)
```

"Value" class

This `ZWaveValue` class represents the values (readings, system and configuration parameters) that can be retrieved from a node. Here are few attributes and methods:

- `label` (type: str): the label of the value. E.g., "Temperature" for the readings of a sensor node or "Level" for a dimmer's brightness.
- `data` (type: depends on the type of the value): the reading value (measurement).
- `units` (type: str) : unit of the value. E.g. 'C' for Celsius temperature readings.
- index (type: int) : identifies a configuration parameter value.
- `genre` (type: str) : genre of the value (Basic, User, Config, System).
- `type` (type: str) : type of the value as in libopenzwave.PyValueTypes.
- `command_class` (type: int) : class_id of the value (hexadecimal code).

Exercise: a Z-Wave-based IoT application

A Z-Wave-based IoT application has been partially developed and shall be completed. The source code is available at https://gitedu.hesge.ch/lsds/teaching/master/iot/Smart-Building

The application is made of three main components (Figure 7):

- `flask-main.py:` a front-end RESTful server module. You don't need to modify this file.
- `backend.py:` the incomplete backend module which implements the methods called by the RESTful server. This module uses the Py-OZW library; you shall complete it.
- The Py-OZW library, which is already locally installed for the user `pi` on your Raspberry Pi.

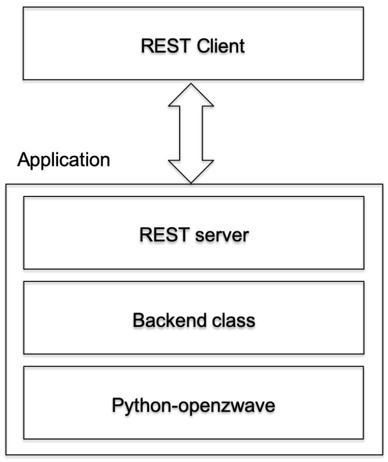

Figure 7: Architecture of the application

The other relevant files in the Git repository are:

- `configpi.py`: the application's configuration file which should work out-of-the-box in most cases.
- `post_client.py`: a command-line RESTful client script to test the application.

Now it is time to install the incomplete application on your Raspberry Pi. Log in to your Raspberry Pi via SSH (replace 'X' with your assigned kit number — the instructor will provide you with the password):

```
your-computer$ ssh pi@192.168.1.2x
```

First thing to do, **change** user pi's password:

```
rbpiX$ passwd
```

Then, in the home directory (`/home/pi/`), download the source code of the Smart Building project:

```
rbpiX$ cd IoT
rbpiX$ git clone https://gitedu.hesge.ch/lsds/teaching/master/iot/Smart-Building
```

The resulting directory `Smart-Building` contains the application's incomplete source codes. Please, refer to the project's `README.md` file for the details of what shall be modified and how to run the application. You can test it by launching the server:

```
rbpiX$ cd Smart-Building
rbpiX$ python flask-main.py
```

In the application's backend module (`backend.py`) some methods are already implemented; you shall implement the missing ones, as listed in Annex 1. The frontend module (`flask-main.py`) provides all the documented calls to the corresponding backend methods. The full REST API documentation is available as a Web page `doc/index.html` (apidoc) from the Git project and can be accessed from a either a local clone URL:

```
file:///.../Smart-Building/doc/index.html
```

or, from the application running on your Raspberry Pi:

```
http://192.168.1.2x:5000
```

Summary of the work to do

The goal of this lab is to set up a minimal Z-Wave-based IoT platform and complete the RESTful application:

1. Set up the Z-Wave network as explained above.
2. Implement, in the file `backend.py`, the methods listed in Annex 1 according to the specifications given in the API documentation, and *without* modifying the file `flask-main.py`.
3. Test the application with the help of the REST client `post_client.py`:
    - help: `$ post_client.py --help`
    - manual: `$ post_client.py --manual`

Deliverable: your `backend.py` file with your implemented methods.

Annex 1: List of backend methods

Available

1. `get_node_location`
2. `get_nodes_configuration`
3. `get_sensor_temperature`
4. `hard_reset`
5. `soft_reset`
6. `start`
7. `stop`

To implement

8. `add_node`
9. `get_dimmer_level`
10. `get_dimmers_list`
11. `get_neighbours_list`
12. `get_node_name`
13. `get_node_parameter`
14. `get_nodes_list`
15. `get_sensor_battery`
16. `get_sensor_humidity`
17. `get_sensor_luminance`
18. `get_sensor_motion`
19. `get_sensor_readings`
20. `get_sensor_ultraviolet`
21. `get_sensors_list`
22. `network_info`
23. `remove_node`
24. `set_dimmer_level`
25. `set_node_location`
26. `set_node_name`
27. `set_node_parameter`
28. `set_sensors_parameter`