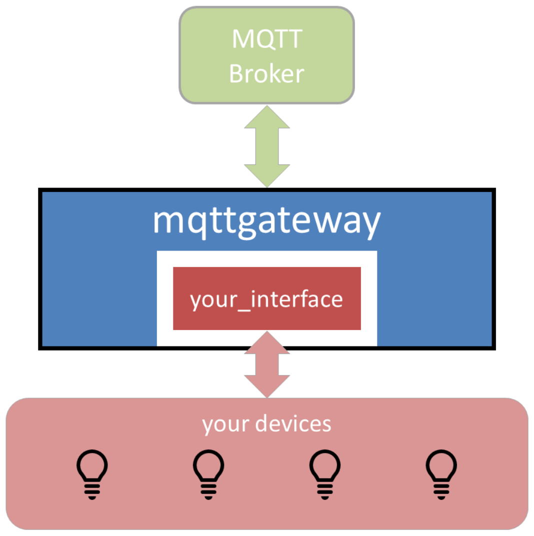
mqttgateway Documentation

Release 1.1.5

Paolo Taddonio

Jun 17, 2022

Contents



`mqttgateway` is a python framework to build consistent gateways to MQTT networks.

CHAPTER 1

What it does:

- it deals with all the boilerplate code to manage MQTT connections, load configuration and other data files, and create log handlers;
- it encapsulates the interface in a class that needs only 2 methods, an initialisation method (`__init__`) and a loop method (`loop` or `loop_start`);
- it creates an intuitive messaging abstraction layer between the wrapper and the interface;
- it isolates the syntax and keywords of the MQTT network from the interface.

CHAPTER 2

Who is it for:

Developers of MQTT networks in a domestic environment looking to adopt a definitive syntax for their MQTT messages and to build gateways with their devices that are not MQTT enabled.

CHAPTER 3

Available interfaces

Check the existing fully developped interfaces. Their names usually follows the pattern **<interface_name>2mqtt**, for example musiccast2mqtt.

This library comes with a **dummy** interface to test the installation and that can be used as a template.

CHAPTER 4

Links

- **Documentation** on readthedocs.
- **Source** on github.
- **Distribution** on pypi.

CHAPTER 5

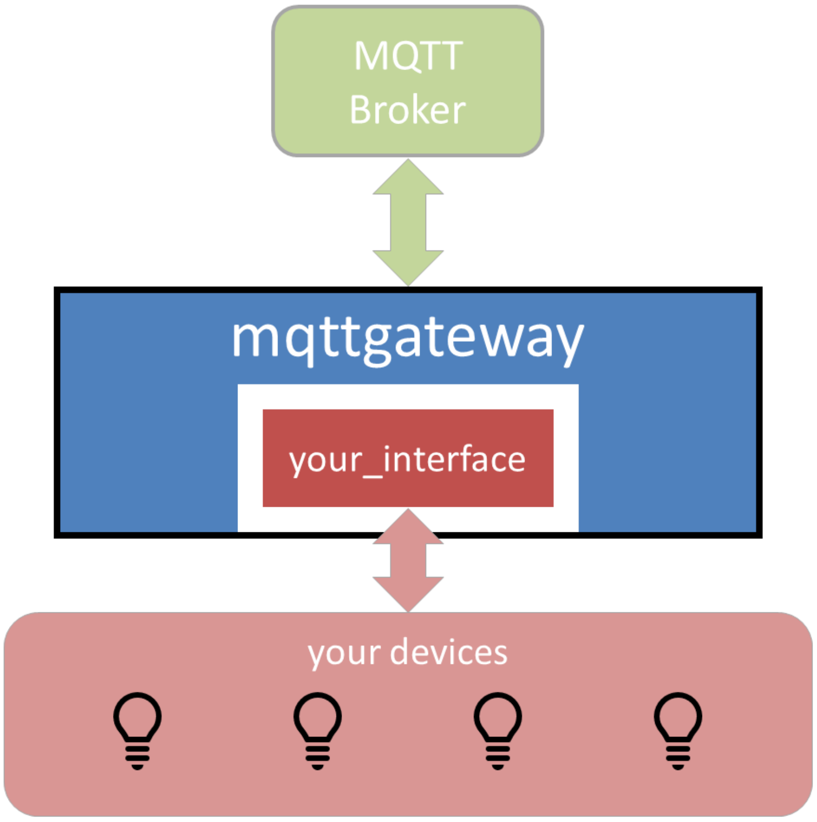
Contents

5.1 Overview

5.1.1 Objective

When setting up an IoT eco-system with a lot of different devices, it becomes quickly difficult to have them talking to each other. A few choices need to be made to solve this problem. This project assumes that one of those choices has been made: using MQTT as the messaging transport. This project then intends to help in the next set of choices to make: defining a messaging model and expressing it in an MQTT syntax to be shared by all devices.

This model is implemented as a python library aimed at facilitating coding the gateways between devices that do not support natively MQTT communication and the MQTT network. These gateways can then run as services on machines connected to these devices via whatever interface is available: serial, Bluetooth, TCP, or else.



5.1.2 Description

This project has two parts:

1. The definition of the messaging model. It is an abstraction layer that defines a message by a few characteristics, adapted to domestic IoT environments, that help resolving the destination and purpose of the message in a flexible and intuitive way.
2. The implementation of this model through a python library. The library takes care of formatting and translating back and forth the messages between their MQTT syntax and their internal representation, as well as managing the connection to the broker and various application necessities.

For more information, go to *Description*.

5.1.3 Usage

This project is provided with the core library, and an example interface (the **dummy** interface) that does not interface with anything but shows how the system works. Once installed, running the application `dummy2mqtt` allows to test the basic configuration and show how it is reacting to incoming MQTT messages, for example.

Developers can then write their own interface by using the **dummy** interface as a template, or following the tutorial alongside the theoretical interface **entry**.

End users will download already developed interfaces, for which this library will simply be a dependency.

For a complete guide on how to develop an interface, go to *Tutorial*.

5.1.4 Installation

The installation can be done with `pip`, on both Linux and Windows systems. The only dependency is the paho.mqtt library.

For the full installation guide, go to *Installation*.

5.2 Installation

5.2.1 Download

Get the library from the PyPi repository with the `pip` command, preferably using the `--user` option:

```
pip install --user mqttgateway
```

Alternatively use the *bare* `pip` command if you have administrator rights or if you are in a virtual environment.

```
pip install mqttgateway
```

Running `pip` also installs an executable file (`exe` in Windows or executable python script in Linux) called `dummy2mqtt`. It launches the demo interface **dummy** with the default configuration. Its location *should* be `%APPDATA%\Python\Scripts\dummy2mqtt.exe` on Windows and `~/.local/bin/dummy2mqtt` on Linux (*it probably depends on the distribution though. . .*). If not, please search for the file manually.

Also, those same locations *should* be already defined in the **PATH** environment variable and therefore the executable *should* launch from any working directory. If not, the variable will have to be updated manually, or the executable needs to be specified with its real path.

5.2.2 Configuration

A configuration file is needed for each interface. In the library, the default interface `dummy` has its own configuration file `dummy2mqtt.conf` inside the package folder.

The configuration file has a standard `INI` syntax, as used by the standard library `ConfigParser` with sections identified by `[SECTION]` and options within sections identified by `option:value`. Comments are identified with a starting character `#`.

There are four sections:

1. `[MQTT]` defines the MQTT parameters, most importantly the IP address of the broker under the option `host`. The address of the MQTT broker should be provided in the same format as expected by the **paho.mqtt** library, usually a raw IP address (`192.168.1.55` for example) or an address like `test.mosquitto.org`. The default port is 1883, if it is different it can also be indicated in the configuration file under the option `port`. Authentication is not available at this stage.
2. `[LOG]` defines the different logging options. The library can log to the console, to a file, send emails or just send the logs to the standard error output. By default it logs to the console.
3. `[INTERFACE]` is the section reserved to the actual interface using this library. Any number of options can be inserted here and will be made available to the interface code through a dictionary initialised with all the `option:value` pairs.

4. `[CONFIG]` is a section reserved to the library to store information about the configuration loading process. Even if it is not visible in the configuration file it is created at runtime.

For more details about the `.conf` file, its defaults and the command line arguments, go to *Configuration*.

5.2.3 Launch

If `pip` installed correctly the executable files, just launch it from anywhere:

```
dummy2mqtt
```

Launched without argument, the application looks for a configuration file in the same directory as the targeted script with the same name as the application, with a *.conf* extension. In this case, it finds the file `dummy2mqtt.conf` inside the package folder:

```
# Configuration file for dummy2mqtt. -------------------------------------------------
↪----------------------------
[CONFIG]
# Reserved section used by the loader to store the location where
#   the configuration settings are coming from, or to store
#   the error if there was one.

# -----------------------------------------------------------------------------
[INTERFACE]
# Section for whatever options are needed by the gateway interface
#   being developed. All these options will be written in a
#   dictionary and passed to the interface.

# Example (it can be checked in the logs):
port: /dev/tty

# -----------------------------------------------------------------------------
[MQTT]
# The parameters to connect to the MQTT broker
host: test.mosquitto.org
port: 1883
keepalive: 60

# The client id can be provided here; if left empty it defaults to the application␣
↪name
clientid:

# This is the timeout of the 'loop()' call in the MQTT library
timeout: 0.01

# Mapping option. By default it is off.
mapping: on

# Map file: there needs to be a mapping file if the <mapping> option is on.
#   If the <mapfilename> option is left blank, the mapping option is turned
#   off, whatever the value of the <mapping> option.
#   To use the default name and path, use a dot <.> for this option.
#   The default name used is <*application_name*.map>.
#   See below for other instructions on file names and paths.
mapfilename: dummy_map.json

# The 'root' keyword for all MQTT messages.
```

(continues on next page)

(continued from previous page)

```
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the keyword should then be found in the mapping file.
#   By default it is empty ("").
# Example:
#root: home
# Here we don't need it because we use a map.
root:

# The topics to subscribe to, separated by a comma.
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the topics should then be found in the mapping file.
# Example:
#topics: home/lighting/#, home/+/mygateway/#
# Here we don't need it because we use a map.
topics:

# ----------------------------------------------------------------------------
[LOG]
# Logs: all WARN level logs and above are sent to stderr or equivalent.
#   3 more log outputs can be set up: console, rotating files and email.
# Log levels: indicate what log levels are required for each log output.
#   Levels are indicated with the following strings (from the logging module):
#   CRITICAL, ERROR, WARN or WARNING, INFO and DEBUG; use NONE if unused.

# Console level: these are the logs directed to stdout.  Usually only used for␣
↪testing.
consolelevel: DEBUG

# Log file: file location if logs to file is required.
#   Leave this option blank to not enable a log file.
#   Use a dot <.> to use the default name and path.
#   The default name used is <*application_name*.log>.
#   Make sure the application will have the rights to write in this file.
#   See below for other instructions on file names and paths.
logfilename:

# File level: level for logs directed to the file named by the <logfilename> option.
#   If that option is blank, there is no file log whatever value is given to the␣
↪option
#   <filelevel> (there is no default file).
filelevel: NONE

# Number of files required for the rotating files. Default is 3.
#filenum:3

# Maximum size of each log file, in KB. Default is 50'000.
#filesize: 50000

# Email credentials; leave empty if not required.
#   All CRITICAL level logs are sent to this email, if defined.
#   For now there is no authentication.
emailhost:
# for example: emailhost: 127.0.0.1
emailport:
# for example: emailport: 25
emailaddress:
# for example: address: me@example.com
```

(continues on next page)

(continued from previous page)

```
#-------------------------------------------------------------------------------
# Note on file paths and names:
#   - the default name is 'application_name' +
#                          default extension (.log, .map, ... etc);
#   - the default directories are (1) the configuration file location, (2) the
#     current working directory, (3) the application directory, which
#     'should' be the location of the launching script;
#   - empty file paths have different meaning depending where they are used;
#     best to avoid;
#   - file paths can be directory only (ends with a '/') and are appended with
#     the default name;
#   - file paths can be absolute or relative; absolute start with a '/' and
#     relative are prepended with the default directory;
#   - file paths can be file only (no '/' whatsoever) and are prepended with
#     the default directory;
#   - use forward slashes '/' in any case, even for Windows systems, it should
#     work;
#   - however for Windows systems, use of the drive letter might be an issue
#     and has not been tested.
#-------------------------------------------------------------------------------
```

With the configuration provided, the application uses `test.mosquitto.org` as MQTT broker and will log messages from all levels only into the console.

Once started, the application logs a banner message and the full configuration used. Check here that all the options are as intended.

Then the log should show if the MQTT connection was successful and display the topics to which the application has subscribed.

After the start-up phase, the **dummy** interface logs any MQTT messages it receives. It also emits a unique message every 30 seconds.

Start your a MQTT monitor app (I use mqtt-spy). Connect to your MQTT broker (here it is `test.mosquitto.org`) and subscribe to the topic:

```
testmqttgtw/dummyfunction/#
```

You should see the messages arriving every 30 seconds in the MQTT monitor, as well as in the log.

As the application has subscribed as well to this same topic `testmqttgtw/dummyfunction/#`, it receives back from the broker the same message it just sent, as can be seen in the log.

Publish now a message from the MQTT monitor:

```
topic: testmqttgtw/dummyfunction//kitchen//me/C
payload: audio_on
```

You should see in the log that the MQTT message has been received by the gateway, and that it has also been processed correctly by the mapping processor: a first log indicates that the **MQTT** message has been received by the `mqttgateway` library, a second log indicates that the **internal** message has been received by the `dummy` interface, with the changed (*mapped*) values of the various characteristics.

Note: When the application sends a message with a topic it has subscribed to (as above), it receives it back from the broker, as seen before. Indeed a log showed that the MQTT message was received by the library. However, because of a feature that silences *echo* messages (via the `sender` characteristic), the library stops the message and does not send it to the `dummy` interface. That is why there is no second log in that case.

5.2.4 The mapping data

The mapping data is an optional feature that allows to map some or every keyword in the MQTT vocabulary into the equivalent keyword in the interface. This mapping is a very simple many-to-one relationship between MQTT and internal keywords for each characteristic, and its use is only to isolate the internal code from any changes in the MQTT vocabulary.

For the **dummy** interface, the mapping data is provided by the text file `dummy_map.json`. It's just there as an example and it is enabled in the configuration provided. If you send MQTT messages with MQTT keywords from the mapping file, you should see their *translation* in the logs.

Note that the map file also contains the *root* of the MQTT messages and the topics that the interface should subscribe to.

For more details on the mapping data, go to *Description*.

5.2.5 Deploying a gateway

The objective of developing a gateway is to ultimately deploy it in a production environment. To install a gateway as a service on a linux machine, go to *Configuration*.

5.3 Description

5.3.1 The message model

The primary use case for this project is a domestic environment with multiple *connected* devices of any type where MQTT has been selected as the communication transport.

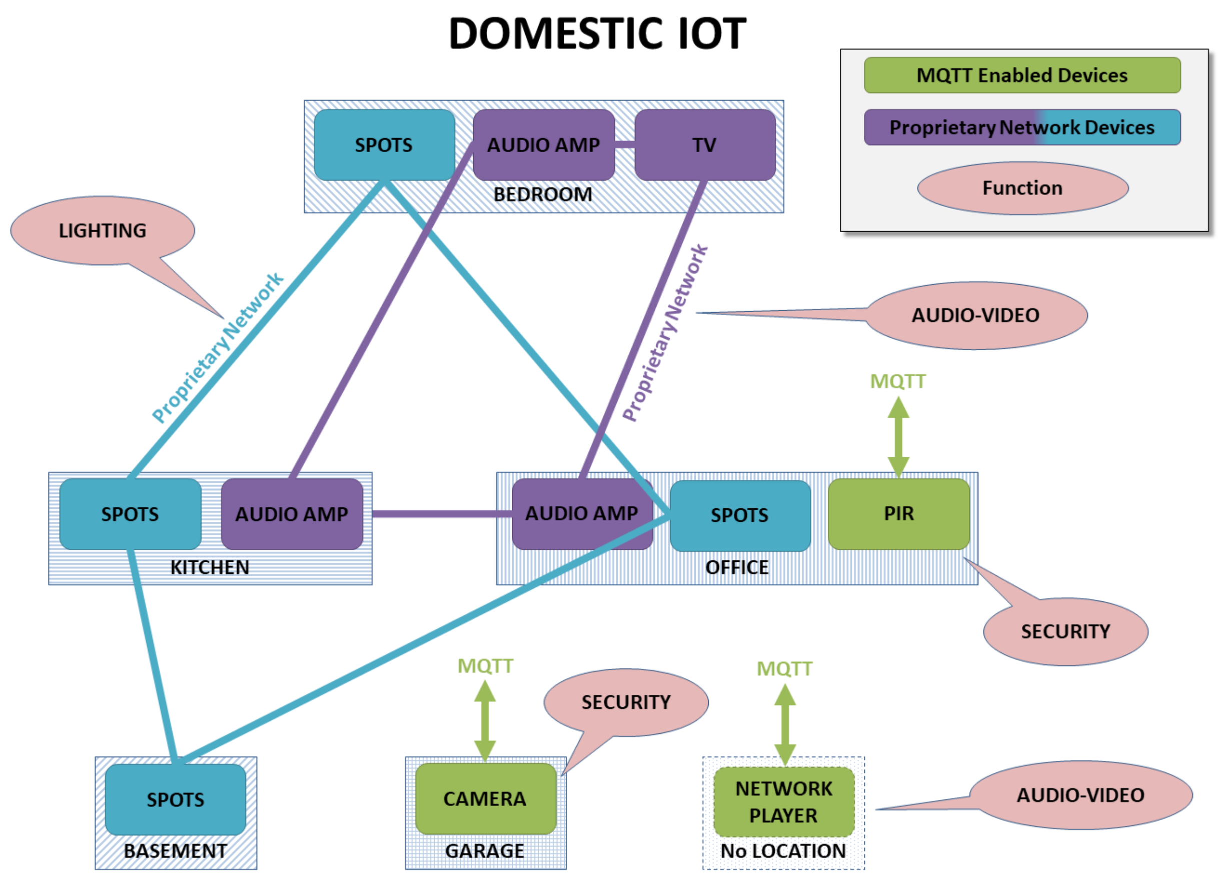
For the devices that communicate natively through MQTT, there is a need to agree on a syntax that makes the exchange of messages coherent.

For those devices that are not *MQTT enabled*, there is a need to develop ad-hoc gateways to *bridge* whatever interface they use natively (serial, Bluetooth, TCP. . .) to one that is MQTT based.

This library addresses both requirements.

Example

In the example below, a smart home has some lighting connected in four different rooms through a proprietary network, four audio-video devices connected through another proprietary network, and some other devices that are already MQTT-enabled, but that still need to speak a common language.



The first objective of this project is to define a common MQTT syntax, and make it as *intuitive* as possible. Ideally, a human should be able to write an MQTT message off-hand and operate successfully any device in the network.

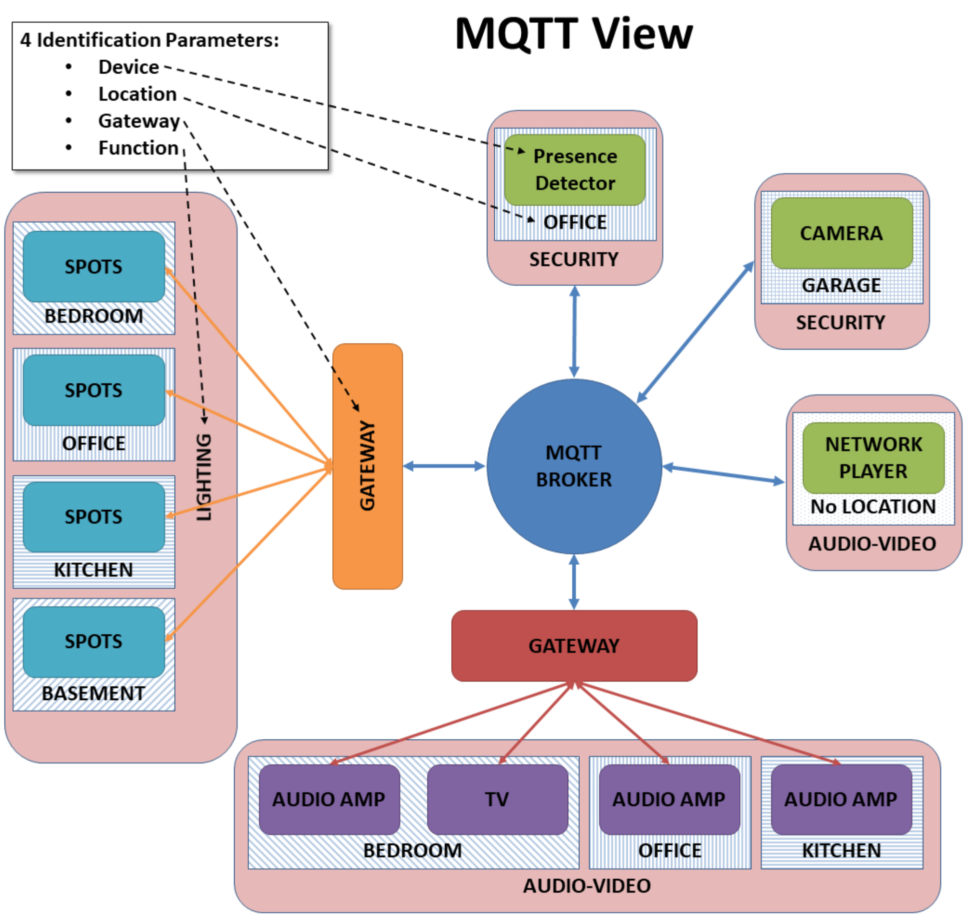
Message Addressing

The first step of any message is to define its destination. A flexible addressing model should allow for a heuristic approach based on a combination of characteristics of the recipient (for example its type and location), instead of a standard deterministic approach (for example a unique device id).

A combination of these four characteristics cover those requirements:

- the **function** of the device: lighting, security, audio-video, etc;
- its **location**;
- its **gateway**: which application is managing that device, if any;
- the name of the **device**.

In our example, the MQTT point of view shows how those four characteristics, or just a subset, can define all the devices in the network.



Some considerations about those four characteristics:

- not all four characteristics need to be provided to address succesfully a device;
- the **device** name can be generic (e.g. `spotlight`) or specific and unique within the network (e.g. `lightid1224`); if the generic case name is used, obviously other characteristics would be needed in the message to address the device.
- any device can *respond* to more than one value for some characteristics; for example a device could have more than one **function**: a connected light fitted with a presence sensor could have both `lighting` and `security` functions.
- the **gateway** and a unique **device** id are the most deterministic characteristics and should be the choices for fast and unambiguous messaging.
- the **location** is probably the most intuitive characteristic; it should represent the place where the device operates and not where it is physically located (e.g. an audio amplifier might be in the basement but if it powers speakers in the living room then that should be its the location); but the location might even not be defined, for example for a house-wide security system, or an audio network player that can broadcast to different rooms.
- the **function** is another important intuitive characteristic; not only it helps in addressing devices, but it also clarifies ambiguous commands (e.g. `POWER_ON` with `lighting` or with `audiovideo` means different things).

Those four characteristics should ensure that the messaging model is flexible enough to be heuristic or deterministic. A gateway will decide how flexible it wants to be. If it has enough processing bandwidth, it can decide to subscribe to all **lighting** messages for example, and then parse all messages received to check if they are actually addressed to it. Or it can subscribe only to messages addressed specifically to itself (through the gateway name for example), restricting access only to the senders that know the name of that gateway.

Message Content

The content of a message is modelled in a standard way with those 3 elements:

- a **type** with 2 possible values: *command* for messages that are requiring an action to be performed, or *status* for messages that only broadcast a state;
- an **action** that indicates what to do or what the status is referring to;
- a set of **arguments** that might complete the **action** characteristic.

The key characteristic here is the **action**, a string representing the *what* to do, with the optional **arguments** helping to define by *how much* for example. It can be `POWER_ON` and `POWER_OFF` on their own for example (no argument), or `SET_POWER` with the argument `power:ON` or `power:OFF`, or both. The interface decides what actions it recognises.

Message Source

The sender, which can be a single device if it has direct access to the MQTT network or a gateway, is another characteristic in this model. It can be very useful in answering status requests in a targeted way, for example.

5.3.2 Bridging MQTT and the interface

There are therefore a total of 8 characteristics in our message model:

- **function**,
- **gateway**,
- **location**,
- **device**,
- **type**,
- **action**,
- **argument** of action,
- **sender**.

They are all strings except **type** which can only have 2 predefined values. They are all the fields that can appear in a MQTT message, either in the topic or in the payload. They are all attributes of the internal message class that is used to exchange messages between the library and the interface being developed. They are all the characteristics available to the developer to code its interface.

The internal message class

The internal message class *`internalMsg`* defines the message objects stored in the lists that are shared by the library and the interface. There is a list for incoming messages and a list for outgoing messages. At its essence, the library simply parses MQTT messages into internal ones, and back. The library therefore defines the MQTT syntax by the way it converts the messages.

The conversion process

The conversion process happens inside the class `msgMap` with two methods to translate back and forth messages between MQTT and the internal message class.

These methods achieve 2 things:

- define the syntax of the MQTT messages in the way the various characteristics are positioned within the MQTT topic and payload;
- if mapping is enabled, map the keywords for every characteristic between the MQTT *vocabulary* and the internal one; this is done via dictionaries initialised by a *mapping file*.

The MQTT syntax

The library currently defines the MQTT syntax as follows.

The **topic** is structured like this:

```
root/function/gateway/location/device/sender/type
```

where `root` can be anything the developer wants (`home` for example) and `type` can be only `C` or `S`.

The **payload** is simply the action alone if there are no arguments:

```
action_name
```

or the action with the arguments all in a JSON string like this:

```
{"action":"action_name","arg1":"value1","arg2":"value2",...}
```

where the first `action` key is written as is and the other argument keys can be chosen by the developer and will be simply copied in the **argument** dictionary.

This syntax is defined within the 2 methods doing the conversions. The rest of the library is agnostic to the MQTT syntax. Therefore one only needs to change these 2 methods to change the syntax. However in that case, all the devices and other gateways obviously have to adopt the same new syntax.

The mapping data

By default, when the mapping option is disabled, the keywords used in the MQTT messages are simply copied in the internal class. So, for example, if the **function** in the MQTT message is `lighting`, then the attribute `function` in the class `internalMsg` will also be the string `lighting`. If for any reason a keyword has to change on the MQTT *side* (maybe because a new device is not flexible enough and imposes its own keywords), it would have to be reflected in the code, which is unfortunate. For example this new device, a connected bulb, uses `light` as function and not `lighting`, but `lighting` is now hard coded in the interface. In order for the interface to recognise this new keyword, a *mapping* can be introduced that links the keyword `light` in the MQTT messages to `lighting` in the internal representation of messages. This mapping is defined in a separate JSON file, and the code does not need to be modified.

The mapping option can be enabled (it is off by default) in the configuration file, in which case the location of the JSON file is required. All the keyword characteristics (except **type**) can (but do not have to) be mapped in that file: **function**, **gateway**, **location**, **device**, **sender**, **action**, **argument keys** and **argument values**.

Furthermore, to give more flexibility, there are 3 mapping options available for each characteristic that need to be specified:

- `none`: the keywords are left unchanged, so there is no need to provide the mapping data for that characteristic;

- `strict`: the conversion of the keywords go through the provided map, and any missing keyword raises an exception; the message with that keyword is probably ignored;
- `loose`: the conversion of the keywords go through the provided map, but missing keywords do not raise any error and are passed unchanged.

The mapping between internal keywords and MQTT ones is a one-to-many relationship for each characteristic. For each internal keyword there can be more than one MQTT keyword, even if there will have to be one which has *priority* in order to define without ambiguity the conversion from internal to MQTT keyword. In practice, this MQTT keyword will be the first one in the list provided in the mapping (see below) and the other keywords of that list can be considered *aliases*.

Going back to the example above, for the unique internal function keyword `lighting`, we would define a list of MQTT keywords as `["light", "lighting"]`, so that `lighting` in internal code gets converted to `light` in MQTT (as it is the new *priority* keyword) but `lighting` in MQTT is still accepted as a keyword that gets *converted* to `lighting` in internal messages.

The mapping data is provided by a JSON formatted file. The JSON schema `mqtt_map_schema.json` is available in the `gateway` package. New JSON mapping files can be tested against this schema (I use the online validation tool https://www.jsonschemavalidator.net/)

The mapping file also contains the topics to subscribe to and the root token for all the topics. These values override the ones found in the configuration file if the mapping feature is enabled.

5.3.3 Application structure

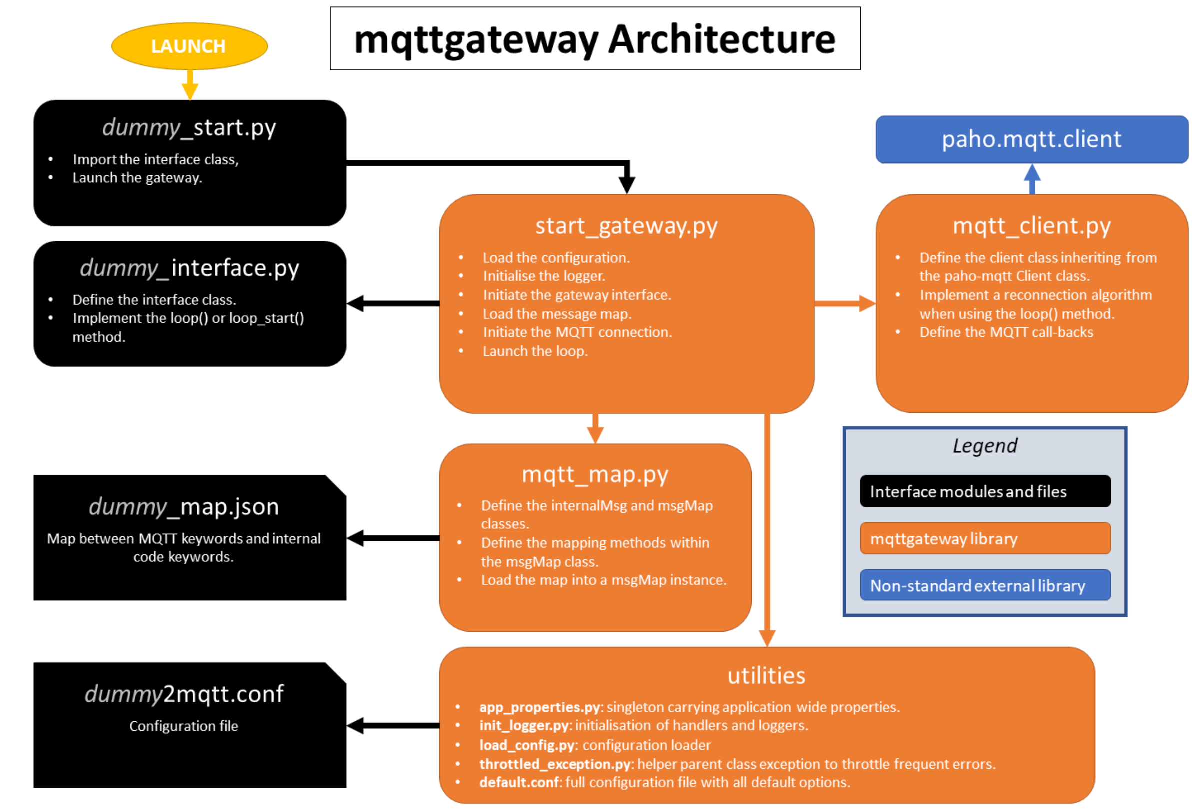
The `mqttgateway` package contains all the files needed to run a full application, in this case the `dummy2mqtt` application.

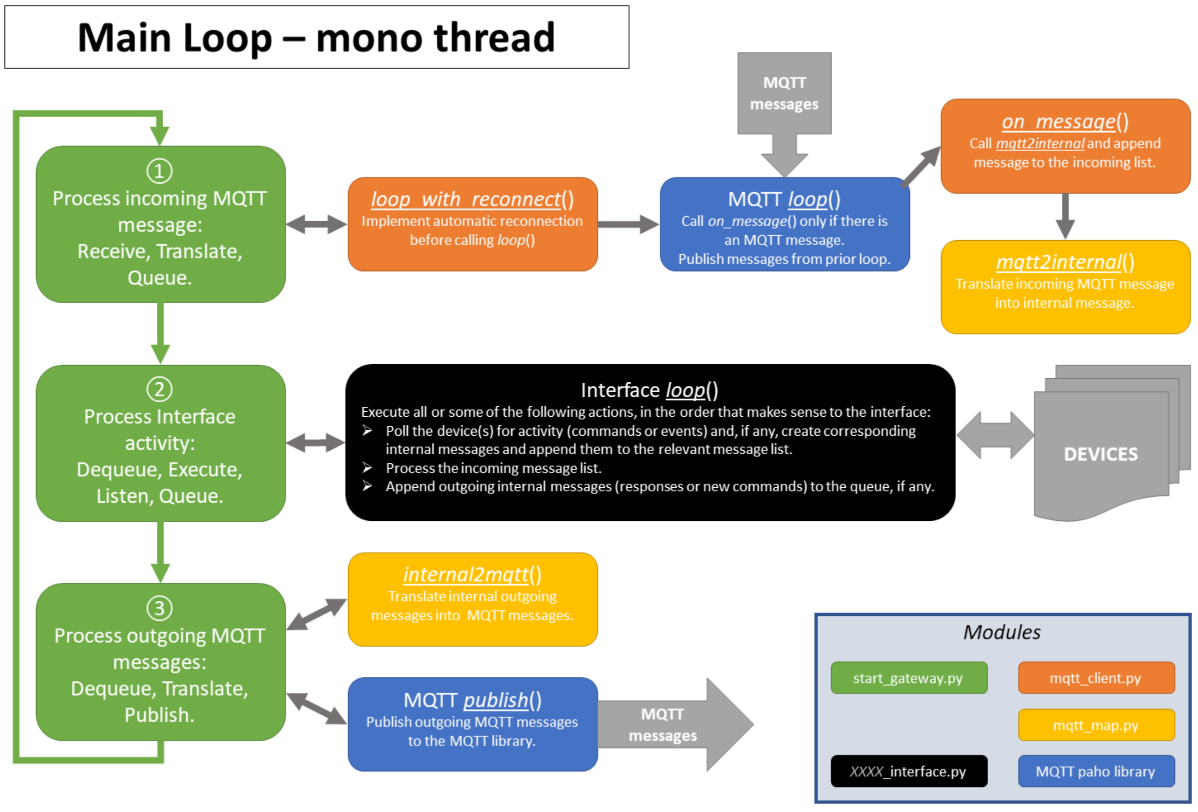
The files related to the implementation of the `dummy` interface are:

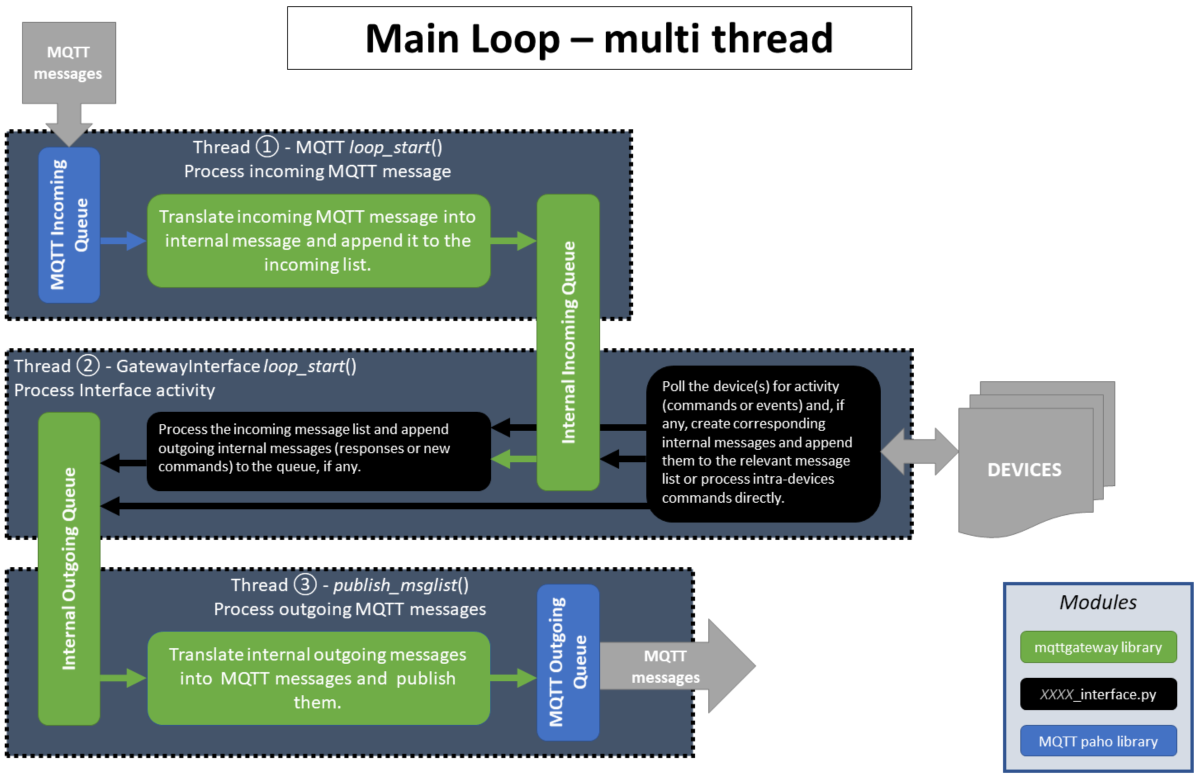
- `dummy_start.py`: the launcher script; call this script to launch the application.
- `dummy_interface.py`: the module that defines the class and methods called by the `mqttgateway` library to actually run the interface.
- `dummy2mqtt.conf`: the configuration file for the dummy interface, compulsory.
- `dummy_map.json`: the mapping file, optional.

The files exclusively related to the library are:

- `start_gateway.py`: the main module of the library, it configures the application, initialise the interface and the MQTT connection, and launches the loop(s).
- `mqtt_client.py`: the internal MQTT class, inherited from the `paho-mqtt` library, defines the algorithm to reconnect automatically when using the `loop()` method.
- `mqtt_map.py`: defines the internal message class, the maps and their methods, and loads the maps if any.
- `app_properties.py`: utility, defines a singleton with application-wide properties.
- `load_config.py`: utility, loads the configuration from a file.
- `init_logger.py`: utility, initialise handlers and loggers.
- `default.conf`: file with all the configuration options and their defaults.
- `mqtt_map_schema.json`: JSON schema to check the mapping files.


mqttgateway Architecture
LAUNCH
dummy_start.py
Import the interface class,
Launch the gateway.
dummy_interface.py
Define the interface class.
Implement the loop() or loop_start() method.
start_gateway.py
Load the configuration.
Initialise the logger.
Initiate the gateway interface.
Load the message map.
Initiate the MQTT connection.
Launch the loop.
paho.mqtt.client
mqtt_client.py
Define the client class inheriting from the paho-mqtt Client class.
Implement a reconnection algorithm when using the loop() method.
Define the MQTT call-backs
mqtt_map.py
Define the internalMsg and msgMap classes.
Define the mapping methods within the msgMap class.
Load the map into a msgMap instance.
dummy_map.json
Map between MQTT keywords and internal code keywords.
Legend
Interface modules and files
mqttgateway library
Non-standard external library
dummy2mqtt.conf
Configuration file
utilities
app_properties.py: singleton carrying application wide properties.
init_logger.py: initialisation of handlers and loggers.
load_config.py: configuration loader
throttled_exception.py: helper parent class exception to throttle frequent errors.
default.conf: full configuration file with all default options.


Main Loop – mono thread
① Process incoming MQTT message: Receive, Translate, Queue.
loop_with_reconnect()
Implement automatic reconnection before calling loop()
MQTT messages
MQTT loop()
Call on_message() only if there is an MQTT message.
Publish messages from prior loop.
on_message()
Call mqtt2internal and append message to the incoming list.
mqtt2internal()
Translate incoming MQTT message into internal message.
② Process Interface activity: Dequeue, Execute, Listen, Queue.
Interface loop()
Execute all or some of the following actions, in the order that makes sense to the interface:
Poll the device(s) for activity (commands or events) and, if any, create corresponding internal messages and append them to the relevant message list.
Process the incoming message list.
Append outgoing internal messages (responses or new commands) to the queue, if any.
DEVICES
③ Process outgoing MQTT messages: Dequeue, Translate, Publish.
internal2mqtt()
Translate internal outgoing messages into MQTT messages.
MQTT publish()
Publish outgoing MQTT messages to the MQTT library.
MQTT messages
Modules
start_gateway.py
mqtt_client.py
mqtt_map.py
XXXX_interface.py
MQTT paho library


Main Loop – multi thread
MQTT messages
Thread ① - MQTT loop_start()
Process incoming MQTT message
MQTT Incoming Queue
Translate incoming MQTT message into internal message and append it to the incoming list.
Internal Incoming Queue
Thread ② - GatewayInterface loop_start()
Process Interface activity
Process the incoming message list and append outgoing internal messages (responses or new commands) to the queue, if any.
Poll the device(s) for activity (commands or events) and, if any, create corresponding internal messages and append them to the relevant message list or process intra-devices commands directly.
DEVICES
Internal Outgoing Queue
Thread ③ - publish_msglist()
Process outgoing MQTT messages
Translate internal outgoing messages into MQTT messages and publish them.
MQTT Outgoing Queue
MQTT messages
Modules
mqttgateway library
XXXX_interface.py
MQTT paho library

5.4 Tutorial

Note: This tutorial refers to an early library. An update is in development.

Let's go through a practical example, with a very simple protocol.

5.4.1 The Need

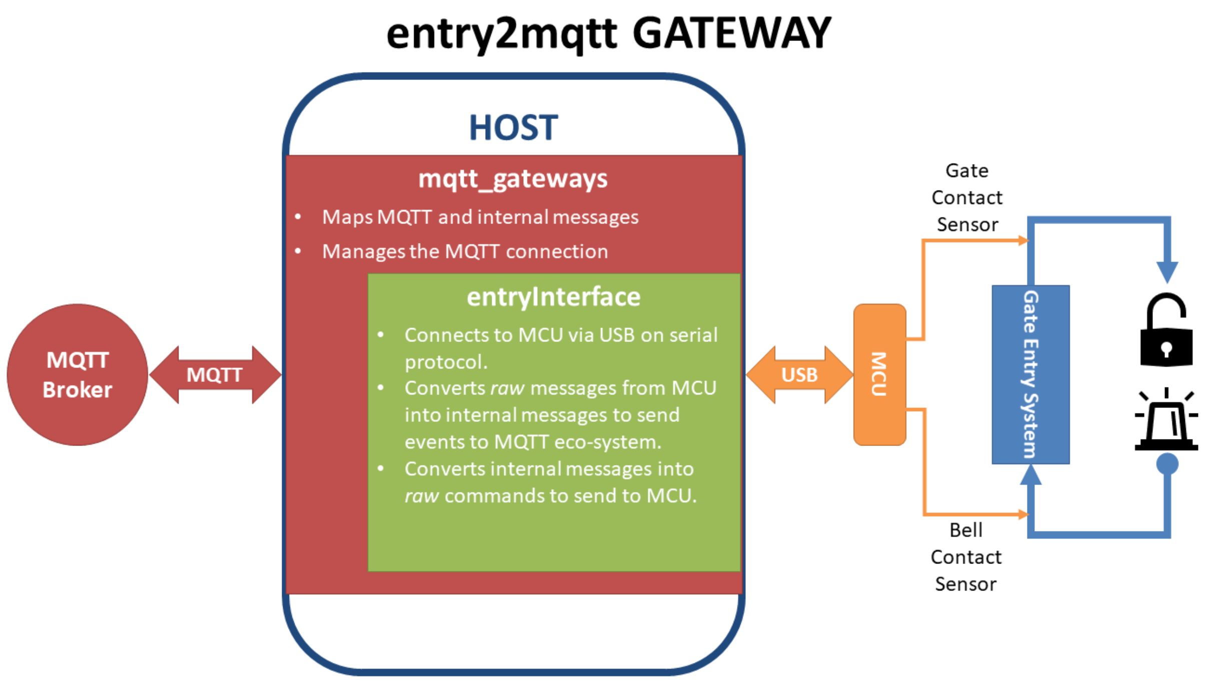
The gate of the house has an entry system, or intercom. Visitors push the bell button, and if all goes well after a brief conversation someone in the house let them in by pushing a gate release button. Residents have a code to let themselves in: they enter the code and the system releases the gate.

It would be nice to receive messages about these events, so that other events can be triggered (like switching on lights by night). It would also be nice to trigger the gate release independently of the entry system.

5.4.2 The Solution

We assume the entry system exposes the electrical contacts that operate the bell and the gate. A micro-controller (an Arduino for example), can sense the electrical contacts going HIGH or LOW and can communicate these levels to a computer through a serial port. The micro-controller can also be told to switch ON or OFF a relay to release the gate. We will call `Entry System` the combination of the actual entry system with the micro-controller physical interface.

Note: a computer with the right sensors like a Raspberry Pi could sense directly the electrical contacts without being shielded by another board. However this use-case suits the tutorial, and is probably more reliable in the long run.


entry2mqtt GATEWAY
HOST
mqtt_gateways
Maps MQTT and internal messages
Manages the MQTT connection
entryInterface
Connects to MCU via USB on serial protocol.
Converts raw messages from MCU into internal messages to send events to MQTT eco-system.
Converts internal messages into raw commands to send to MCU.
MQTT Broker
MQTT
USB
MCU
Gate Contact Sensor
Gate Entry System
Bell Contact Sensor

5.4.3 Implementation

The micro-controller is programmed to communicate with very simple messages for each event: each message is a pair of digits (in ASCII), the first indicating which contact the message is about and the second indicating its state. With 2 contacts (the bell and the gate), and 2 states (ON and OFF), there are only 4 messages to deal with: `10`, `11`, `20` and `21`. More precisely, the micro-controller:

- sends a message when a contact goes ON (`11` or `21`) and another one when it goes off (`10` or `20`);
- can also receive and process messages; in our case only the one triggering the gate release makes sense (let's say it is the *message* `21`); we will assume that the micro-controller turns the gate release OFF automatically after 3 seconds, for security, so there is no need to use the gate release OFF message (`20`); similarly, there is no need to process the messages `11` or `10` as there is no need to operate the bell via MQTT.

The next step is therefore to code the interface for the computer connected to the micro-controller. Let's call the interface **entry**. This will be the label used in all the names in the project (packages, modules, folders, class, configuration and mapping files).

5.4.4 The interface

The interface will be a Python package called `entry2mqtt`. Let's create it in a new folder `entry` with an empty module `__init__.py`. In order not to start from scratch, let's use the `dummy` interface as a template. Copy the files `dummy_start.py` and `dummy_interface.py` from the `dummy` package into the `entry` package, and change all the `dummy` instances into `entry` (in the name of the file as well as inside the file). The actual interface code has to be in the class `entryInterface` within the module `entry_interface.py`. It needs to have at least a constructor `__init__` and a method called `loop`.

The constructor

The constructor receives 3 arguments: a dictionary of parameters and two message lists, one for incoming messages and the other one for outgoing ones.

The dictionary of parameters is loaded with whatever we put in the configuration file in the `[INTERFACE]` section. It's up to us to decide what we put in there. Here we probably only need a *port* name in order to open the serial port. We will create the configuration file later, but for now we will assume that there will be an option `port:what_ever_it_is` in the `[INTERFACE]` section, so we can retrieve it in our code.

The constructor will generally need to keep the message lists locally so that the `loop` method can access them, so they will be assigned to local members.

Finally, the constructor will have to initialise the serial communication.

Starting from the template copied above, the only thing to add is the opening of the serial port. Add at the top of the module:

```
import serial
```

(you need to have the PySerial library in your environment), and add the following line inside the constructor:

```
self._ser = serial.Serial(port=port, baudrate=9600, timeout=0.01)
```

The `port` variable is already defined in the template (check the code). The `baudrate` has to be the same as the one set by the micro-controller. Finally the `timeout` is fundamental. It has to be short enough so that the main loop is not delayed too much. Without timeout, all the serial exchanges will be blocking, which can not work in our *mono-thread* process.

Note: It is obviously possible to use *natively* multiple threads for the library to avoid the blocking calls issues. Indeed, the paho library is already doing so for its part. However this is not the case for now even if it might be implemented in the future.

The `loop` method

This method is called periodically by the main loop to let our interface do whatever it needs to do.

The `loop` method should deal with the incoming messages first, process them, then *read* its own connected device for events, process them and stack in the outgoing list whatever message needs to be sent, if there are any.

Use the code in the template to read the incoming messages list and add the following code to deal with the case where the message is a command to open the gate:

```
if msg.action == 'GATE_OPEN':
    try:
        self._ser.write('21')
    except serial.SerialException:
        self._logger.info('Problem writing to the serial interface')
```

Always try to catch any exception that should not disrupt the whole application. Most of them should not be fatal.

Then read the serial interface to see if there are any events:

```
try:
    data = self._ser.read(2)
except serial.SerialException:
    self._logger.info('Problem reading the serial interface')
    return
if len(data) < 2:
    return
```

If there is an event, convert it into an internal message and add it to the outgoing message list:

```
if data[0] == '1':
    device = 'Bell'
    if data[1] == '0':
        action = 'BELL_OFF'
    elif data[1] == '1':
        action = 'BELL_ON'
    else:
        self._logger.info('Unexpected code from Entry System')
        return
elif data[0] == '2':
    device = 'Gate'
    if data[1] == '0':
        action = 'GATE_CLOSE'
    elif data[1] == '1':
        action = 'GATE_OPEN'
    else:
        self._logger.info('Unexpected code from Entry System')
        return
msg = internalMsg(iscmd=False, # it is a status message
                  function='Security',
                  gateway='entry2mqtt',
                  location='gate_entry',
                  device=device,
                  action=action)
self._msgl_out.append(msg)
```

Finally, let’s send a command to switch on the light in case the gate was opened:

```
if data == '21':
    msg = internalMsg(iscmd=True,
                      function='Lighting',
                      location='gate_entry',
                      action='LIGHT_ON')
    self._msgl_out.append(msg)
```

That’s it for the basic logic.

Other coding strategies

The resulting code is as simple as it can be. There are clearly more advanced *coding strategies* to make the code more *elegant* and ultimately easier to mantain and upgrade.

For example, the class can be defined as a subclass of the Serial class, as this would reflect well what it actually is, i.e. a higher level serial interface to a specific device.

Another possibility is to code the conversion of the messages from the serial interface into internal messages through lookup tables (dictionaries) instead of nested *ifs*.

There are always better ways to code, but it is important to note that, as the loop is supposed to run fast and is the piece of code that will run forever, it is worth investing some time on how to make that part more efficient.

The map file

The mapping feature is disabled by default. This means that all the keywords introduced earlier in the code (the commands `GATE_OPEN`, `GATE_CLOSE`, `BELL_ON` and `BELL_OFF`, as well as the location `gate_entry` and the function identifiers `Security``and ``Lighting`) will all be passed on **as is** to the MQTT messages, with exactly the same spelling and the same capitalised letters, if any. This might be acceptable if there are only a few devices and gateways in the MQTT network and the *vocabulary* stays quite small. But if the network grows and evolves, inevitably changes will happen and it becomes inpractical to have to change the code any time an identifier in the MQTT vocabulary had to change. That is where the mapping feature steps in.

The mapping feature can be enabled in the configuration file, in which case a file location for the map needs to be provided:

```
...
mapping: on
mapfilename: /the/path/to/your/mapfile/filename.json
```

The map file location option is subject to the various *rules* for file paths used in this library. In a nutshell, if the path is absolute there is no ambiguity, if it is relative the library will try the path starting from the configuration file directory first, then try the current working directory of the process, and finally the directory of the launching script.

The mapping file is a JSON formatted file with 2 objects (the `root` of the MQTT messages and a list of `topics` to subscribe to) and up to 8 dictionaries, 1 for each characteristic that can potentially be mapped. For each characteristic, a `maptype` needs to be provided (it can be either `none`, `loose` or `strict`) and then an actual `map`, if the `maptype` is not `none`.

For our interface, we assume we want to map all the data, as shown in the table:

Table 1: Data to map for the entry gateway

Characteritic	MQTT Keyword	Interface Keyword
function	security	Security
function	lighting	Lighting
gateway	entry2mqtt	entry2mqtt
location	frontgarden	gate_entry
device	gate	Gate
device	bell	Bell
action	gate_open	GATE_OPEN
action	bell_off	BELL_OFF
action	bell_on	BELL_ON
action	light_off	LIGHT_OFF
action	light_on	LIGHT_ON
action	gate_close	GATE_CLOSE

The map file would then look like this:

```
{
    "root": "home",
    "topics": ["home/security/+/frontgarden/+/+/C",
               "home/+/entry2mqtt/+/+/+/C",
               "home/+/+/+/entrysystem/+/C"],
    "function": {
        "map": {"security": "Security", "lighting": "Lighting"},
        "maptype": "strict"
    },
    "gateway": {
        "map": {"entry2mqtt": "entry2mqtt"},
        "maptype": "strict"
    },
    "location": {
        "map": {"frontgarden": "gate_entry"},
        "maptype": "strict"
    },
    "device": {
        "map": {"gate": "Gate", "bell": "Bell"},
        "maptype": "strict"
    },
    "sender": {"maptype": "none"},
    "action": {
        "map": {"gate_open": "GATE_OPEN",
                "bell_off": "BELL_OFF",
                "bell_on": "BELL_ON",
                "light_off": "LIGHT_OFF",
                "light_on": "LIGHT_ON",
                "gate_close": "GATE_CLOSE"
        },
        "maptype": "strict"
    },
    "argkey": {"maptype": "none"},
    "argvalue": {"maptype": "none"}
}
```

Save it in a file named `entry_map.json`.

A few comments on this *suggested* mapping:

- most of these keyword mappings only change the case or even nothing; this is for illustration purposes anyway, but in general it might still be good discipline to list all the keywords in a mapping to have in one view what the interface can deal with or not. Then if one day some MQTT keyword needs to change, everything is ready to do so.
- an important choice to make is the `maptype` for each characteristic. If it is set to `strict`, it will enable to filter messages quite early in the process and alleviate the code of further testing. In our example, even if the `gateway` map has only one item, which is even the same on both sides, setting the `maptype` to `strict` ensures that **only** that keyword is accepted, and any other one is discarded. This is obviously very different from setting the `maptype` to `none`, in which case that only keyword would still be accepted and left unchanged, but so would any other keyword.

5.4.5 Wrapping it all up

Once the interface is defined, all is left to do is to create the launch script and the configuration file. Those 2 steps are easy using the templates.

Copy the **dummy** project launch script `dummy_start.py` and paste it into the `entry` directory. Change every instance of `dummy` into `entry``. If all the naming steps have been respected, the script `entry_start.py` just created should work.

To create the configuration file, copy the configuration file `dummy2mqtt.conf` from the `dummy` package and paste it in the folder `entry` with the name `entry2mqtt.conf`. Edit the file and enter the `port` option under the `[INTERFACE]` section:

```
[INTERFACE]
port=/dev/ttyACM0
```

Obviously input whatever is the correct name of the port, the one shown is generally the one to use on a Raspberry Pi for the USB serial connection. If you are on Windows, your port should be something like `COM3`.

If you went through the *installation* process, then the MQTT parameters should already be set up, otherwise do so. Other parameters can be left as they are. Check the *configuration* guide for more details.

5.4.6 Launch

To launch the gateway, just run the launcher script directly from its directory:

```
python entry_start
```

Done!

5.5 Configuration

Note: In development

In the meantime, the default configuration, which is in the file `default.conf` inside the library package, is well documented and is a good starting point to understand the various options.

```
# Default configuration file for mqttgateway. -----------------------------------------
↪------------------------------------
[CONFIG]
# Reserved section used by the loader to store the location where
#   the configuration settings are coming from, or to store
#   the error if there was one.

# ----------------------------------------------------------------------------
[INTERFACE]
# Section for whatever options are needed by the gateway interface
#   being developed. All these options will be written in a
#   dictionary and passed to the interface.

# Example:
#port: /dev/tty

# ----------------------------------------------------------------------------
[MQTT]
# The parameters to connect to the MQTT broker
host: 127.0.0.1
port: 1883
```

(continues on next page)

(continued from previous page)

```
keepalive: 60

# The client id can be provided here; if left empty it defaults to the application␣
↪name
clientid:

# This is the timeout of the 'loop()' call in the MQTT library
timeout: 0.01

# Mapping option. By default it is off.
mapping: off

# Map file: there needs to be a mapping file if the <mapping> option is on.
#   If the <mapfilename> option is left blank, the mapping option is turned
#   off, whatever the value of the <mapping> option.
#   To use the default name and path, use a dot <.> for this option.
#   The default name used is <*application_name*.map>.
#   See below for other instructions on file names and paths.
mapfilename: .

# The 'root' keyword for all MQTT messages.
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the keyword should then be found in the mapping file.
#   By default it is empty ("").
# Example:
#root: home

root:

# The topics to subscribe to, separated by a comma.
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the topics should then be found in the mapping file.
# Example:
#topics: home/lighting/#, home/+/mygateway/#

topics:

# -----------------------------------------------------------------------------
[LOG]
# Logs: all WARN level logs and above are sent to stderr or equivalent.
#   3 more log outputs can be set up: console, rotating files and email.
# Log levels: indicate what log levels are required for each log output.
#   Levels are indicated with the following strings (from the logging module):
#   CRITICAL, ERROR, WARN or WARNING, INFO and DEBUG; use NONE if unused.

# Console level: these are the logs directed to stdout.  Usually only used for␣
↪testing.
consolelevel: NONE

# Log file: file location if logs to file is required.
#   Leave this option blank to not enable a log file.
#   Use a dot <.> to use the default name and path.
#   The default name used is <*application_name*.log>.
#   Make sure the application will have the rights to write in this file.
#   See below for other instructions on file names and paths.
logfilename:
```

(continues on next page)

(continued from previous page)

```
# File level: level for logs directed to the file named by the <logfilename> option.
#   If that option is blank, there is no file log whatever value is given to the␣
↪option
#   <filelevel> (there is no default file).
filelevel: INFO

# Number of files required for the rotating files. Default is 3.
filenum:3

# Maximum size of each log file, in KB. Default is 50'000.
filesize: 50000

# Email credentials; leave empty if not required.
#   All CRITICAL level logs are sent to this email, if defined.
#   For now there is no authentication.
emailhost:
# for example: emailhost: 127.0.0.1
emailport:
# for example: emailport: 25
emailaddress:
# for example: address: me@example.com

#-------------------------------------------------------------------------------
# Note on file paths and names:
#   - the default name is 'application_name' +
#                          default extension (.log, .map, ... etc);
#   - the default directories are (1) the configuration file location, (2) the
#     current working directory, (3) the application directory, which
#     'should' be the location of the launching script;
#   - empty file paths have different meaning depending where they are used;
#     best to avoid;
#   - file paths can be directory only (ends with a '/') and are appended with
#     the default name;
#   - file paths can be absolute or relative; absolute start with a '/' and
#     relative are prepended with the default directory;
#   - file paths can be file only (no '/' whatsoever) and are prepended with
#     the default directory;
#   - use forward slashes '/' in any case, even for Windows systems, it should
#     work;
#   - however for Windows systems, use of the drive letter might be an issue
#     and has not been tested.
#-------------------------------------------------------------------------------
```

5.6 mqttgateway package

5.6.1 Warning

As of 24 May 2018, most of the docstrings are obsolete. They will be updated gradually as soon as possible.

5.6.2 Package contents

The **mqttgateway** library helps in building gateways between connected devices and MQTT systems.

This package has 4 *groups* of files:

- the **core** of the library made of the modules:
  - `start_gateway.py` which contains the script for the application initialisation and main loop;
  - `mqtt_client.py` which defines the child of the MQTT Client class of the paho library, needed to implement a few extra features;
  - `mqtt_map.py` which defines the internal message class `internalMsg` and the mapping class `msgMap`.
- the **utilities** used by the core and that are really *application agnostic*; these are in the modules:
  - `app_properties.py`, a singleton that holds application wide data like name, directories where to look for files, configuration and log information;
  - `init_logger`, used by `app_properties` to initialise the loggers and handlers;
  - `load_config`, used by `app_properties` to load the configuration;
  - `throttled_exception`, an exception class that *mutes* events if they are too frequent, handy for connection problems happening in fast loops.
- the `dummy` interface, an empty interface to test the installation of the library and to be used as a template to write a new interface, and which is made of the modules:
  - `dummy_start`, the launcher script;
  - `dummy_interface`, the actual interface main class.
- various data files:
  - **`default.conf`, the file containing all the configuration options and their default** values;
  - `mqtt_map_schema.json`, the schema of the mapping files;
  - `dummy_map.json` and `dummy2mqtt.conf`, the map and configuration file of the `dummy` interface.

5.6.3 Modules

5.6.4 mqttgateway.app_properties module

Application wide properties.

This module defines a singleton that hides some application-wide properties. As any singleton, any instantiation returns always the same object. This one specifically re-routes the `__init__` method to ensure that all variables are only updated at the first instantiation and are never changed again. The attributes are only accessible with getters and there are no setters.

class `mqttgateway.app_properties.`**`AppProperties`**(**args*, ***kwargs*)

Bases: `object`

Singleton holding application properties.

`_init_properties`(*app_path*, *app_name=None*, *parse_dict=None*)

Initialisation of the properties.

This is effectively the constructor of the class. At first instantiation of this singleton, the `__init__` method points to this method. For the following instantiations, the `__init__` method points to the `_dummy` function.

Parameters

- **app_path** (*string*) – the full path of the launching script, including filename
- **app_name** (*string*) – the application name if different from the filename
- **parse_dict** (*dict*) – not implemented

get_path (*path_given*, *extension=None*, *dft_name=None*, *dft_dirs=None*)

Returns the absolute path of a file based on defined rules.

The rules are:

- the default name is `dft_name` + `extension`;
- the default directories are provided by the `dft_dirs` argument;
- file paths can be directory only (ends with a `/`) and are appended with the default name;
- file paths can be absolute or relative; absolute start with a `/` and relative are prepended with the default directory;
- file paths can be file only (no `/` whatsoever) and are prepended with the default directory;
- use forward slashes `/` in all cases, it should work even for Windows systems;
- however for Windows systems, use of the drive letter might be an issue and has not been tested.

Currently this method could return a path to a file that does not exist, if a file corresponding to the rules is not found.

Parameters

- **path_given** (*string*) – any type pf path; see rules
- **extension** (*string*) – the default extension of the file
- **dft_name** (*string*) – the default name to be used, usually the application name
- **dft_dirs** (*list of strings*) – the default directories where to look for the file

Returns the full path of the file found.

Return type string

get_file (*path_given*, *extension=None*, *dft_name=None*, *dft_dirs=None*)

Returns the content of the file defined by the arguments.

This method uses the `get_path()` to determine the file sought. All the usual exceptions are raised in case of problems. It is assumed the content of the file is text and that the size is small enough to be returned at once. The arguments are the same as `get_path()`.

Parameters

- **path_given** (*string*) – any type pf path; see rules
- **extension** (*string*) – the default extension of the file
- **dft_name** (*string*) – the default name to be used, usually the application name
- **dft_dirs** (*list of strings*) – the default directories where to look for the file

Returns the full content of the file.

Return type string

get_jsonfile (*path_given*, *extension=None*, *dft_name=None*, *dft_dirs=None*)

Returns a dictionary with the content of the JSON file defined by the arguments.

This method uses the `get_path()` to determine the file sought. All the usual exceptions are raised in case of problems. The arguments are the same as `get_path()`.

Parameters

- **`path_given`** (*`string`*) – any type pf path; see rules
- **`extension`** (*`string`*) – the default extension of the file
- **`dft_name`** (*`string`*) – the default name to be used, usually the application name
- **`dft_dirs`** (*`list of strings`*) – the default directories where to look for the file

Returns the content of the JSON file in dictionary format.

Return type dict

`load_config`(*cfg_dflt_string*, *cfg_filepath*)

See *`loadconfig`* for documentation.

`init_log_handlers`(*log_data*)

Creates new handlers from log_data and add the new ones to the log handlers list.

Also updates the registered loggers with the newly created handlers. See method *`initloghandlers`* for documentation on log_data format.

Parameters **`log_data`** (*`string`*) – see related doc for more info.

Returns a message indicating what has been done, to be potentially logged.

Return type string

`register_logger`(*logger*)

Register the logger and add the existing handlers to it.

Call this method to add the logger in the registry held in *`AppProperties`*. Doing so, the logger will inherit all the pre-defined handlers.

Parameters **`logger`** (*logging.Logger* object) – the logger to register

`get_name`()

Name getter.

Returns the name of the application.

Return type string

`get_directories`()

Directories getter.

The relevant directories of the application are computed and stored at once at the launch of the application. If they have been deleted, moved or their name is changed while the application is running, they will not be valid anymore.

The relevant directories are the current working directory, the directory of the launching script and the directory where the configuration file was found (which could be different from the first 2 because of the option to provide it in the command line).

Returns a list of full paths of relevant directories for the application.

Return type list

`get_cmdline_args`()

Command line arguments getter.

Returns the dictionary returned by `parser.parse_args()`.

Return type dict

get_config()

Configuration getter.

Returns the dictionary returned by `ConfigParser.RawConfigParser()`

Return type dict

5.6.5 mqttgateway.dummy_interface module

The **dummy** interface class definition. Use it as a template.

This module defines the class *`dummyInterface`* that will be instantiated by the module *`start_gateway`*.

class `mqttgateway.dummy_interface.`**dummyInterface**(*params*, *msglist_in*, *msglist_out*)

Bases: `object`

An interface that doesn't do anything but allows to test the installation.

The minimum requirement for the interface class is to define 2 out of 3 possible public methods:

- the constructor `__init__`,
- either the `loop` method or the `loop_start` method.

Parameters

- **params** (*`dictionary of strings`*) – contains all the options from the configuration file This dictionary is initialised by the `[INTERFACE]` section in the configuration file. All the options in that section generate an entry in the dictionary. Use this to pass parameters from the configuration file to the interface, for example the name of a port, or the speed of a serial communication.
- **msglist_in** (*`MsgList`* object) – list of incoming messages in their internal representation.
- **msglist_out** (*`MsgList`* object) – list of outgoing messages in their internal representation.

loop()

The method called periodically by the main loop.

Place here your code to interact with your system.

5.6.6 mqttgateway.dummy_start module

Launcher script for the **dummy** gateway.

This is an *empty* gateway to test the installation setup. It allows to test the loading of the configuration files, the log setup and the basic operation of the core application.

One can also use this as a template. If the name conventions have been respected, just change all occurrences of `dummy` into the name of your interface.

`mqttgateway.dummy_start.`**main**()

The entry point for the application

5.6.7 mqttgateway.init_logger module

Functions to create pre-defined handlers and add them to a logger.

Reminder of attributes for formatting log records (from https://docs.python.org/2/library/logging.html#logrecord-attributes):

Attribute name	Format	Description
asctime	%(asctime)s	**Human-readable time when the LogRecord was crea** By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	%(created)f	**Time when the LogRecord was created (as returned** time.time()).
filename	%(filename)s	**Filename portion of pathname. mynote: = module +** python scripts.
funcName	%(funcName)s	Name of function containing the logging call.
levelname	%(levelname)s	**Text logging level for the message ('DEBUG', 'INFO** 'WARNING', 'ERROR', 'CRITICAL').
levelno	%(levelno)s	**Numeric logging level for the message (DEBUG, INF** WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	**Source line number where the logging call was issue** (if available).
module	%(module)s	Module (name portion of filename).
msecs	%(msecs)d	**Millisecond portion of the time when the LogRecord** created.
message	%(message)s	**The logged message, computed as msg % args.** This is set when Formatter.format() is invoked.
name	%(name)s	Name of the logger used to log the call.
pathname	%(pathname)s	**Full pathname of the source file where the logging ca** was issued (if available).
process	%(process)d	Process ID (if available).
processName	%(processName)s	Process name (if available). mynote: is always "MainProcess".
relativeCreated	%(relativeCreated)d	**Time in milliseconds when the LogRecord was create** relative to the time the logging module was loaded.
thread	%(thread)d	Thread ID (if available).
threadName	%(threadName)s	Thread name (if available).

The argument datefmt to the Formatter class is of the form: “%Y-%m-%d %H:%M:%S”. The complete set of fields can be found here: time.strftime.

The *default* (ISO8601) formatter seems to be:

```
logging.Formatter(fmt='%(asctime)s.%(msecs)03d',datefmt='%Y-%m-%d,%H:%M:%S')
```

The strings to use to identify the logging levels are defined in the constant `_LEVELNAMES`.

`mqttgateway.init_logger.`**`_LEVELNAMES = {'CRITICAL': 50, 'DEBUG': 10, 'ERROR': 40, 'INFO': 2`**
value in the logging library}

Type Dictionary {“level as string”

`mqttgateway.init_logger.`**`initlogger`**(*logger*, *log_data=None*)
Configures a logger with pre-defined handlers based on user-defined configuration data.

Uses `initloghandlers()` to create the handler, check documentation there for more details on the format of `log_data`.

The logger level is forced to `DEBUG` here.

Parameters

- **`logger`** – the actual logger object to be initialised;
- **`log_data`** (*`dict`*) – dictionary of configuration data.

Returns A string made of various lines of messages relating to what handler has been added to the logger. This string can then be logged by the caller or silenced as desired.

`mqttgateway.init_logger.`**`initloghandlers`**(*log_data*)
Returns a list of handlers based on user-defined configuration data.

The `log_data` has to be in the form of:

```
{
 'console':
    {'level': xxxx },
 'file':
    {'level': yyyy,
     'path': zzzz,
     'number': xxxx,
     'size': yyyy},
 'email':
    {'host': xxxx,
     'port': yyyy,
     'address': zzzz,
     'subject': xxxx }
}
```

The following handlers are created and appended to the list returned:

- the standard ‘Stream’ handler, which will always log level WARN and above to stderr;
- a console log handler;
- a rotating file handler that requires a log level, a file path (used as is), the maximum size of the file and the desired number of rotating files;
- an email handler with the level set to CRITICAL.

The functionality is provided by the standard `logging` library. Check the documentation for more information on the various parameters.

Parameters **`log_data`** (*`dict`*) – dictionary of configuration data

Returns A pair made of the list of handlers and a message (string), to be logged by the caller or silenced as desired.

5.6.8 mqttgateway.load_config module

Function to facilitate the loading of configuration parameters.

Based on the standard library ConfigParser.

`mqttgateway.load_config.`**`loadconfig`**(*cfg_dflt_path*, *cfg_filepath*)

Load the configuration from a file based on a default one.

This function uses the standard library `ConfigParser`.

The objective of this function is to provide a mechanism to ensure that all the options that are expected by the application are defined and will not need to be checked throughout the code. To achieve that, a default configuration needs to be provided, represented by the string `cfg_dflt_path` passed as argument. This string is expected to have all the necessary sections and options that the application will need, with their default values. All options need to be listed there, even the ones that HAVE to be updated and have no default value.

The function loads this default configuration, then checks if the configuration file is available, and if found it grabs only the values from the file that are also present in the default configuration. Anything else in the file is not considered, except for the `[INTERFACE]` section (see below). The result is a configuration object with all the necessary fields, updated by the values in the configuration file, if present, or with the default values if not. The application can therefore call all fields without checking for their existence.

The exception to the above process in the `[INTERFACE]` section, which is the section reserved for the developper of the gateway to define its own specific options. The options of this section will be loaded 'as is' in the Config object. These options will be sent to the interface through a dedicated dictionary. It is then up to the developper of the interface to check their velidity or provide its own defaults for these options.

Finally, the function updates the option `location` in the section `[CONFIG]` with the full path of the configuration file used, so that it can be checked or logged if desired later. This allows to make sure which file has been loaded in case there is some ambiguity. The function also 'logs' the error in the 'error' option of the same section, if any OS exception occurred while opening or reading the configuration file.

Parameters

- **`cfg_dflt_path`** (*`string`*) – represents the default configuration.
- **`cfg_filepath`** (*`string`*) – the path of the configuration file; it is used 'as is' and if it is relative there is no guarantee of where it will actually point. It is preferrable to send a valid absolute path.

Returns `configparser.ConfigParser` object loaded with the options of the configuration.

Return type dict

5.6.9 mqttgateway.mqtt_client module

This is a child class of the MQTT client class of the PAHO library.

It includes the management of reconnection when using only the `loop()` method, which is not included natively in the current PAHO library.

Notes on MQTT behaviour:

- if not connected, the `loop` and `publish` methods will not do anything, but raise no errors either.

- the `loop` method handles always only one message per call.

exception `mqttgateway.mqtt_client.`**`connectionError`**(*msg=None*)

Bases: *`mqttgateway.throttled_exception.ThrottledException`*

Base Exception class for this module, inherits from ThrottledException

`mqttgateway.mqtt_client.`**`mqttmsg_str`**(*mqttmsg*)

Returns a string representing the MQTT message object.

As a reminder, the topic is unicode and the payload is binary.

`mqttgateway.mqtt_client.`**`_on_connect`**(*client*, *userdata*, *flags*, *return_code*)

The MQTT callback when a connection is established.

It sets to True the `connected` attribute and subscribes to the topics available in the message map.

As a reminder, the `flags` argument is a dictionary with at least the key `session present` (with a space!) which will be 1 if the session is already present.

`mqttgateway.mqtt_client.`**`_on_subscribe`**(*client*, *userdata*, *mid*, *granted_qos*)

The MQTT callback when a subscription is completed.

Only implemented for debug purposes.

`mqttgateway.mqtt_client.`**`_on_disconnect`**(*client*, *userdata*, *return_code*)

The MQTT callback when a disconnection occurs.

It sets to False the `mg_connected` attribute.

`mqttgateway.mqtt_client.`**`_on_message`**(*client*, *userdata*, *mqtt_msg*)

The MQTT callback when a message is received from the MQTT broker.

The message (topic and payload) is mapped into its internal representation and then appended to the incoming message list for the gateway interface to execute it later.

class `mqttgateway.mqtt_client.`**`mgClient`**(*host='localhost'*, *port=1883*, *keepalive=60*, *client_id="*, *on_msg_func=None*, *topics=None*, *userdata=None*)

Bases: `paho.mqtt.client.Client`

Class representing the MQTT connection. `mg` means `MqttGateway`.

Inheritance issues: The MQTT paho library sets quite a few attributes in the Client class. They all start with an underscore and have *standard* names (`_host`, `_port`,. . .). Also, some high level methods are used extensively in the paho library itself, in particular the `connect()` method. Overriding them is dangerous. That is why all the homonymous attributes and methods here have an `mg_` prepended to avoid these problems.

Parameters

- **host** (*string*) – a valid host address for the MQTT broker (excluding port)
- **port** (*int*) – a valid port for the MQTT broker
- **keepalive** (*int*) – see PAHO documentation
- **client_id** (*string*) – the name (usually the application name) to send to the MQTT broker
- **on_msg_func** (*function*) – function to call during on_message()
- **topics** (*list of strings*) – e.g.['home/audiovideo/#', 'home/lighting/#']
- **userdata** (*object*) – any object that will be passed to the call-backs

lag_end()

Method to inhibit the connection test during the lag.

One of the feature added by this class over the standard PAHO class is the possibility to reconnect when disconnected while using only the `loop()` method. In order to achieve this, the connection is checked regularly. At the very beginning of the connection though, there is the possibility of a race condition when testing the connection state too soon after requesting it. This happens if the `on_connect` call-back is not called fast enough by the PAHO library and the main loop tests the connection state before that call-back has had the time to set the state to `connected`. As a consequence the automatic reconnection feature gets triggered while a connection is already under way, and the connection process gets jammed with the broker. That's why we need to leave a little lag before testing the connection. This is done with the function variable `lag_test`, which is assigned to this function (`lag_end`) at connection, and switched to a *dummy* lambda after the lag has passed.

lag_reset()

Resets the lag feature for a new connection request.

mg_connect()

Sets up the *lag* feature on top of the parent `connect` method.

See `lag_end` for more information on the *lag feature*.

mg_reconnect()

Sets up the *lag* feature on top of the parent method.

loop_with_reconnect(*timeout*)

Implements automatic reconnection on top of the parent loop method.

The use of the method/attribute `lag_test()` is to avoid having to test the lag forever once the connection is established. Once the lag is finished, this method gets replaced by a simple lambda, which hopefully is much faster than calling the time library and doing a comparison.

5.6.10 mqttgateway.mqtt_map module

This module is the bridge between the internal and the MQTT representation of messages.

As a reminder, we define the MQTT syntax as follows:

- topic:

```
root/function/gateway/location/device/sender/type-{C or S}
```

- payload: action or status, in plain text or in a json string like `{key1:value1,key2:value2,..}`

class mqttgateway.mqtt_map.**internalMsg**(*iscmd=False*, *function=None*, *gateway=None*, *location=None*, *device=None*, *sender=None*, *action=None*, *arguments=None*)

Bases: `object`

Defines all the characteristics of an internal message.

Note about the behaviour of None: a characteristic set to `None` and one set to an empty string are considered the same, and they both mean a non existent or missing value. It could be interesting to differentiate between then at a later stage as, for example, an empty string could still be mapped to an existing internal value, as if it was a default, but that is not the case here. Therefore `None` values are always converted to empty strings.

TODO: implement smart retrieval of arguments with key checks

Parameters

- **iscmd** (*bool*) – Indicates if the message is a command (True) or a status (False)
- **function** (*string*) – internal representation of function
- **gateway** (*string*) – internal representation of gateway
- **location** (*string*) – internal representation of location
- **device** (*string*) – internal representation of device
- **sender** (*string*) – internal representation of sender
- **action** (*string*) – internal representation of action
- **arguments** (*dictionary of strings*) – all values should be assumed to be strings

clear()
Clears all content of the message.

copy()
Creates a copy of the message.

argument(*arg*, *raises=False*, *default=None*)
Return the argument if found in the arguments dictionary.

reply(*response*, *reason*)
Formats the message to be sent as a reply to an existing command

This method is supposed to be used with an existing message that has been received. Using this method for all replies guarantees a consistent syntax for replies.

Parameters

- **response** (*string*) – code or abbreviation for response, e.g. `OK```or ``ERROR`
- **reason** (*string*) – longer description of the responses

class mqttgateway.mqtt_map.**MsgList**
Bases: `Queue.Queue`, `object`

Message list to communicate between the library and the interface.

Defined as a Queue list in case the library is used in multi-threading mode.

The methods are called `push` and `pull` in order to differentiate them from the *usual* names (put, get, append, pop, . . .).

push(*item*, *block=True*, *timeout=None*)
Pushes the item at the end of the list.

Equivalent to append or put in other list implementations. The `block` and `timeout` arguments have the same meaning as in the `Queue` library.

Parameters

- **item** (*object*) – the object to push in the list
- **block** (*boolean*) – in case the list is full
- **timeout** (*float*) – wait time if block == True

pull(*block=False*, *timeout=None*)
Pull the first item from the list.

Equivalent to pop or get in other list implementations. The `block` and `timeout` arguments have the same meaning as in the `Queue` library.

Parameters

- **block** (*boolean*) – in case the list is empty
- **timeout** (*float*) – wait time if block == True

class mqttgateway.mqtt_map.**mappedTokens** (*function*, *gateway*, *location*, *device*, *sender*, *action*, *argkey*, *argvalue*)

Bases: `tuple`

Tokens representing a message that can be mapped.

_asdict ()

Return a new OrderedDict which maps field names to their values

_fields = ('function', 'gateway', 'location', 'device', 'sender', 'action', 'argkey',

classmethod _make (*iterable*, *new=<built-in method __new__ of type object>*, *len=<built-in function len>*)

Make a new mappedTokens object from a sequence or iterable

_replace (***kwds*)

Return a new mappedTokens object replacing specified fields with new values

action

Alias for field number 5

argkey

Alias for field number 6

argvalue

Alias for field number 7

device

Alias for field number 3

function

Alias for field number 0

gateway

Alias for field number 1

location

Alias for field number 2

sender

Alias for field number 4

mqttgateway.mqtt_map.**NO_MAP = {'action': {'maptype': 'none'}, 'argkey': {'maptype': 'no**

Default map, with no mapping at all.

class mqttgateway.mqtt_map.**msgMap** (*jsondict=None*)

Bases: `object`

Contains the mapping data and the conversion methods.

The mapping data is read from a JSON style dictionary. To access the maps use:

```
mqtt_token = maps.*field*.i2m(internal_token)
```

Example:

```
mqtt_token = maps.gateway.i2m(internal_token)
```

Parameters **jsondict** (*dictionary*) – contains the map data in the agreed format; if None, the NO_MAP structure is used.

class tokenMap(*maptype*, *mapdict=None*)

Bases: `object`

Represents the mapping for a given token or characteristic.

Each instantiation of this class represent the mapping for a given token, and contains the type of mapping, the mapping dictionary if available, and the methods to convert the keywords back and forth between MQTT and internal representation.

The mapping dictionary passed as argument has the internal keywords as keys and as value a list of corresponding MQTT keywords. Only the first of the list will be used for the reverse dictionary, the other MQTT keywords being 'aliases'.

Parameters

- **maptype** (*string*) – type of map, should be either 'strict'. 'loose' or 'none'
- **mapdict** (*dictionary*) – dictionary representing the mapping

m2i(*mqtt_token*)

Generic method converting an MQTT token into an internal characteristic.

i2m(*internal_token*)

Generic method converting an internal characteristic into an MQTT token.

static _mapnone(*token*, *dico*)

Returns the argument unchanged.

Parameters

- **token** (*string*) – the token to convert
- **dico** (*dictionary*) – the mapping dictionary to use for the conversion, if needed

Returns converted token

Return type string

static _maploose(*token*, *dico*)

Returns the argument converted if in dictionary, unchanged otherwise.

If `token` is None, it is always converted in an empty string.

Parameters

- **token** (*string*) – the token to convert
- **dico** (*dictionary*) – the mapping dictionary to use for the conversion, if needed

Returns converted token

Return type string

static _mapstrict(*token*, *dico*)

Returns the argument converted if in dictionary, raises exception otherwise.

If `token` is None, it is always converted in an empty string. An empty string is kept as an empty string, even if not in the dictionary.

Parameters

- **token** (*string*) – the token to convert
- **dico** (*dictionary*) – the mapping dictionary to use for the conversion, if needed

Returns converted token

Return type string

sender()

Getter for the `_sender` attribute.

mqtt2internal(*mqtt_msg*)

Converts the MQTT message into an internal one.

Parameters **mqtt_msg** (*mqtt.MQTTMessage*) – a MQTT message.

Returns the conversion of the MQTT message

Return type internalMsg object

Raises `ValueError` – in case of bad MQTT syntax or unrecognised map elements

`internal2mqtt`(*internal_msg*)
Converts an internal message into a MQTT one.

Parameters **`internal_msg`** (*`internalMsg`*) – the message to convert

Returns a full MQTT message where topic syntax is `root/function/gateway/location/device/sender/{C or S}` and payload syntax is either a plain action or a JSON string.

Return type a MQTTMessage object

Raises `ValueError` – in case a token conversion fails

5.6.11 mqttgateway.start_gateway module

Defines the function that starts the gateway.

`mqttgateway.start_gateway.`**`startgateway`**(*gateway_interface*)
Entry point.

`mqttgateway.start_gateway.`**`_startgateway`**(*gateway_interface*)
Initialisation of the application and main loop.

Initialises the configuration and the log, starts the interface, starts the MQTT communication then starts the main loop. The loop can start in mono or multi threading mode. If the `loop` method is defined in the `gateway_interface` class, then the loop will operate in a single thread, and this function will actually *loop* forever, calling every time the `loop` method of the interface, as well as the `loop` method of the MQTT library. If the `loop` method is not defined in the `gateway_interface` class, then it is assumed that the `loop_start` method is defined and it will be launched in a separate thread. The priority given to the mono thread option is for backward compatibility.

The data files are:

- the configuration file (compulsory), which is necessary at least to define the MQTT broker; a path to it can be provided as first argument of the command line, or the default path will be used;
- the map file (optional), if the mapping option is enabled.

The rules for providing paths of files are available in the configuration file template as a comment. The same rules apply to the command line argument and to the paths provided in the configuration file.

Parameters **`gateway_interface`** (*`class`*) – the interface class (not an instance of it!)

5.6.12 mqttgateway.throttled_exception module

An exception class that throttles events in case an error is triggered too often.

`exception` `mqttgateway.throttled_exception.`**`ThrottledException`**(*msg=None*, *throttlelag=10*, *module_name=None*)
Bases: `exceptions.Exception`

Exception class base to throttle events

This exception can be used as a base class instead of `Exception`. It adds a counter and a timer that allow to silence the error for a while if desired. Only after a given period a trigger is set to `True` to indicate that a number of errors have happened and it is time to report them.

It defines 2 members:

- `trigger` is a boolean set to True after the requested lag;
- `report` is a string giving some more information on top of the latest message.

The code using these exceptions can test the member `trigger` and decide to silence the error until it is True. At any point one can still decide to use these exceptions as normal ones, ignore the `trigger` and `report` members and just raise the exception as *normal*.

Usage:

```
try:
    #some statements that might raise your own exception derived from␣
↪ThrottledException
except YourExceptionError as err:
    if err.trigger:
        log(err.report)
```

Parameters

- **msg** (*string*) – the error message, as for usual exceptions, optional
- **throttlelag** (*int*) – the lag time in seconds while errors should be throttled, defaults to 10 seconds
- **module_name** (*string*) – the calling module to give extra information, optional

_count = 0

_timer = 1655500849.557323

5.7 Indices and tables

- genindex
- modindex

Python Module Index

Index