Beginner’s Guide To Using Paho-MQTT, A Python MQTT Client

mntolia.com/mqtt-python-with-paho-mqtt-client/

By Maulin Tolia August 14, 2018

Paho-MQTT is an open-source **Python MQTT client** developed by the Eclipse Foundation. Paho-MQTT can run on any device that supports Python. In this tutorial, we will build an MQTT client with Paho. I will add each feature of the library to the client program and explain how it works. At the end of the tutorial you will have a basic understanding on how the library works.

If you are new to MQTT it’s better to learn the fundamentals first:
Fundamentals of MQTT

To run a broker on your PC, you can install mosquitto:
Installing Mosquitto On Windows and Ubuntu/Debian

What’s going on?

1. We import the paho library and set the broker address as `iot.eclipse.org` and the port number as `1883`.

> `1883` is the default port number in MQTT for all *unencrypted* connections.

2. We create an MQTT client object and call it `client`. We will see more about the paho client object in the next section.

3. Next we call the `connect()` function with the address & port number of the broker.

If the connection is successful, the `connect()` function will return 0.

Let us break down the client object:

1.1 The Client Object

The `client()` object creates an MQTT client. It takes 4 parameters which are optional:

`client(client_id="", clean_session=True, userdata=None, protocol=MQTTv311, transport="tcp")`

.

`client_id` is a unique string given by the client when connecting to a broker. If you don't provide a client id the broker will assign one to the client.

> Each client must have a unique client id. The broker uses the client id to uniquely identify each user. If you connect a second client with the same client id, the first client will get disconnected.

`clean_session` is a boolean set to *True* by default.

If set to *False* the broker stores information about the client.
If set to *True*, the broker will remove all stored information about the client.

To understand clean sessions see: Clean Sessions In MQTT Explained

`userdata` is data that you can send as a parameter to callbacks. More about callbacks in section 4.

`protocol` can be either `MQTTv31` or `MQTTv311` depending on which version you want to use.

`transport` defaults to `tcp`. If you want to send messages over WebSockets then set to `websockets`.

To disconnect from the broker cleanly we can use `disconnect()` function.

2. Publishing A Message

To publish a message we use the `publish()` function. The function takes 4 parameters:

`topic` is a string containing the topic name.

> The topic name is case sensitive.

`payload` is a string containing the message that will be published to the topic. Any client subscribed to the topic will see the payload message.

These are the optional parameters:

`qos` is either 0, 1 or 2. It defaults to 0. Quality of Service is the level of guarantee that the message will get received.
To understand the different MQTT Quality of Service Levels see this post:
MQTT QoS Levels Explained

`retain` is a boolean value which defaults to *False*. If set to True, then it tells the broker to store that message on the topic as the “last good message”.
To understand message retention in MQTT see this: MQTT Retained Messages

Adding publish function to our code:

To check if the message has been successfully published to a topic we need a client subscribed to that topic.

3. Subscribing To Topics

To subscribe to a topic we need the `subscribe()` function. The function takes 2 parameters

`topic` is a string containing the topic name.

> Note: The topic name is case sensitive.

`qos` is either 0, 1 or 2. It detaults to 0.

Let us modify our program for subscribing to the topic that we are publishing to:

> Note: If you want the client to subscribe to multiple topics then you can put them in a list of tuples.
>
> Example: client.subscribe([(‘topicName1’, 1),(‘topicName2’, 1)])
>
> The format of the tuple is [(Topic Name, QoS Level)]

When you execute this program you will notice that the published message still does not get displayed on the console/terminal.

Subscribing to a topic tells the broker to send you the messages that are published to that topic. We have subscribed to the topic but we need a callback function to process those messages.

4. Callbacks

Callbacks are functions that are executed when an event occurs. In paho these events are connect, disconnect, subscribe, unsubscribe, publish, message received, logging.

Before we implement callbacks to our program, we need to first understand how these callbacks can be called by the program. For this we will use the different loop functions available in paho.

4.1 The Loop Functions

loop is a function of the client object. When a message is received by the client, the message is stored in the receive buffer. When a message is to be sent from the client to the broker, it is stored in the send buffer. The loop functions are made to process any messages in the buffer and call a respective callback function. They will also attempt to reconnect to the broker on disconnection.

Most of the loop functions run asynchronously, which means when the loop function is called it will run on a separate thread.

There are 3 types of loop functions:

4.1.1 loop_forever(): This is a **blocking** loop function. Which means the loop function will keep running and you cannot execute any other line after you call this function. Use this function if you want to run the program indefinitely and if you have a single client contructor in your program.

4.1.2 loop_start()/loop_stop(): This is a **non-blocking** loop function which means you can call this function and still continue executing code after the function call. As the name suggests loop_start() is used to start the loop function and loop_stop() is used to stop the loop function. loop_start() can be used if you need to create more than 1 client object in the same program.

4.1.3 loop(): This is a blocking loop function. The difference between loop() and loop_forever() is that if you call the loop() function you have to handle reconnect manually unlike the latter. The loop_forever() & loop_start() function will automatically try to reconnect to the broker when it disconnects. It is not recommended to use loop() unless in special circumstances.

Now getting back to the callbacks. In the next section we will implement each callback into our program.

4.2 on_connect

The `on_connect()` callback is called each time the client connects/reconnects to the broker. Lets add the callback to our program.

You must assign the callback to the client object. If not the callback will not execute

What's Going on?

1. We create the on_connect callback function. It takes 4 parameters: the client object, userdata, flags, rc.
2. The `client` object.

3. `userdata` is custom data declared in the client constructor. You will need this if you want to pass custom data into the callback.
4. `flags` is a dictionary object that is used to check if you have set clean session to True or False for the client object.
5. `rc` which is the **r**esult **c**ode, is used to check the connection status. The different result codes are:

> 0: Connection successful 1: Connection refused – incorrect protocol version 2: Connection refused – invalid client identifier 3: Connection refused – server unavailable 4: Connection refused – bad username or password 5: Connection refused – not authorised 6-255: Currently unused.

The output of the program is:

```
Connected With Result Code 0
```

Which means the connection is successful.

4.3 on_disconnect

The `on_disconnect()` callback is called when the client disconnects from the broker.

Don't forget to assign the callback function to the client object!
`client.on_disconnect = on_disconnect`

Run the program, disconnect your internet and see if the message "Client Got Disconnected" gets printed. Also, reconnect to the internet to see if the client reconnects to the broker.

> You can attach multiple client objects to a single callback function. This is useful when your program connects to multiple brokers.

4.4 on_message

Getting back to the problem of messages not being displayed despite subscribing to the topic: The `on_message()` callback is used to process messages that are published to a subscribed topic.

In our program, we need to do 3 things:
1. Subscribe to the topic that we are publishing to.
2. Process the published message using the callback. In our program, we will simply print the message.
3. Assign the callback function to the `on_message` attribute of the client object.

Let us create another mqtt client. This new program will publish messages to the topic that our existing program is subscribed to.

The on_message() callback has 3 parameters: `client` , `userdata` & `message` . I already explained the first two in the `on_connect()` section.
The `message` object has 4 attributes: `topic` , `payload` , `qos` , `retain` .

If you notice each attribute of the `message` object is also a parameter we use in the client's `publish()` function.

First run `sub.py` and then run `pub.py` . The output will show the message and the topic.

If the message doesn't get printed, check if:

- You have added the function `on_message()` to the client object's attribute.
- If you have subscribed to the topic in the `on_connect()` callback in `sub.py` .
- If you have entered the topic names correctly in the `subscribe()` and `publish()` functions in `sub.py` and `pub.py` respectively.
- If you are printing the message in the `on_message()` function.

How to organize & process the messages?

If you have multiple messages being received from different topics and need to process each topic's message in a different way then we have to sort the messages.

One way to do this is by using the `message.topic` attribute to check which topic is the message published to. Then you can create if conditions to process the messages accordingly.

The better way to do this is to use the `message_callback_add(sub, callback)` function of the `client` object. This creates multiple callbacks to process messages from different topics.
The `sub` parameter takes the topic name and the `callback` parameter takes the name of the callback that will process messages of that topic.

Example:

Make sure you have subscribed to both the topics before adding callbacks to it.

Messages from other topics will get processed by the on_message() by default

You can try the above program by publishing to "KitchenTopic" & "BedroomTopic" and seeing if it processes it separately.

4.5 on_publish

The `on_publish()` function gets called when the `publish()` function is executed. This will return a tuple `(result, mid)` .

`result` is the error code. An error code of 0 means the message is published successfully.

`mid` stands for message id. It is an integer that is a unique message identifier assigned by the client. If you use QoS levels 1 or 2 then the client loop will use the `mid` to identify messages that have not been sent.

4.6 on_subscribe()/on_unsubscribe()

The `on_subscribe()` callback is called when a client subscribes to a topic.
Example: `on_subscribe(client, userdata, mid, granted_qos)`

The `on_unsubscribe()` callback is called when a client unsubscribes to a topic.
Example: `on_unsubscribe(client, userdata, mid)`

`mid` is the message id as discussed in the `on_publish()` function.

`granted_qos` is the qos level for that topic when subscribing. You can try it in your program if you want.

5. Other Useful Paho-MQTT Functions

Paho also has some useful functions that can be used in the program to execute functions without having to create a client constructor and going through the hassle of creating callbacks.

5.1 Single() / Multiple() Publish

The single and multiple functions are used to publish a single message or multiple messages to a topic on a broker without having to create a client object.

`topic` is the only **necessary** parameter. This is a string with the topic name.

`auth` is a dictionary with the username and password if the broker requires it. (The eclipse broker does not require authentication).
Example: `auth = {'username':"username", 'password':""}`

`tls` is required if we are using TLS/SSL encryption.
Example: `dict = {'ca_certs':"", 'certfile':"", 'keyfile':"", 'tls_version':"", 'ciphers':"<ciphers">}`

`transport` is accepts 2 values: `websockets` and `tcp`. Set it to `tcp` if you don't want to use websockets.

The other parameters are similar to the ones in the client object.

`msgs` is a **necessary** parameter. It contains a list of messages to publish. Each message can be dictionary or tuple.

The dictionary must be in this format:
`msg = {'topic':"< topicname >", 'payload':"", 'qos':'0', 'retain':'False'}`

The tuple must be in this format:
`("< topicname >", "< payload >", qos, retain)`

In both these formats a `topic` name has to be present.

5.2 Simple() / Callback()

simple is a blocking function subscribes to a topic or a list of topics and returns the messages published to that topic.

`simple(topics, qos=0, msg_count=1, retained=False, hostname="localhost", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311)`

`topics` is the only necessary parameter. This can a string for a single topic or a list for multiple topics.

`msg_count` is the number of messages that the must be returned before disconnecting from the broker. For msg_count > 1, the function will return a list of messages.

The other parameters are already explained in the single() / multiple() section.

`callback` is similar to `simple`, the only difference it takes an extra parameter namely callback. The simple function simply returns the messages from the topic, the callback function sends the returned messages to any function for processing.
`callback(callback, topics, qos=0, userdata=None, hostname="iot.eclipse.org", port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None, protocol=mqtt.MQTTv311)`

6. Will_set()

This is a very useful function. When the client connects to the broker it tells the broker that if it disconnects it must publish a message to a topic. It is an attribute of the client constructor.

`client.will_set(topic, payload=None, qos=0, retain=False)`

To understand how & when to use Last Will & Testament in MQTT see this post:
MQTT Last Will And Testament (Explained with Example)

Your feedback is very important to me. If this tutorial helped you or, if there are some parts of the article that are wrong or not explained properly, please let me know in the comments below.