**Installation**

The latest stable version is available in the Python Package Index (PyPi) and can be installed using

pip install paho-mqtt

Or with virtualenv:

virtualenv paho-mqtt

source paho-mqtt/bin/activate

pip install paho-mqtt

To obtain the full code, including examples and tests, you can clone the git repository:

git clone git://git.eclipse.org/gitroot/paho/org.eclipse.paho.mqtt.python.git

**Usage and API**

Detailed API documentation is available through **pydoc**. Samples are available in the **examples** directory.

The package provides two modules, a full client and a helper for simple publishing.

**Client**

You can use the client class as an instance, within a class or by subclassing. The general usage flow is as follows:

* Create a client instance
* Connect to a broker using one of the connect\*() functions
* Call one of the loop\*() functions to maintain network traffic flow with the broker
* Use subscribe() to subscribe to a topic and receive messages
* Use publish() to publish messages to the broker
* Use disconnect() to disconnect from the broker

Callbacks will be called to allow the application to process events as necessary. These callbacks are described below.

**Constructor / reinitialise**

**Client()**

Client(client\_id="", clean\_session=True, userdata=None, protocol=MQTTv31)

The Client() constructor takes the following arguments:

client\_id

the unique client id string used when connecting to the broker. If client\_id is zero length or None, then one will be randomly generated. In this case the clean\_session parameter must be True.

clean\_session

a boolean that determines the client type. If True, the broker will remove all information about this client when it disconnects. If False, the client is a durable client and subscription information and queued messages will be retained when the client disconnects.

Note that a client will never discard its own outgoing messages on disconnect. Calling connect() or reconnect() will cause the messages to be resent. Use reinitialise() to reset a client to its original state.

userdata

user defined data of any type that is passed as the userdata parameter to callbacks. It may be updated at a later point with the user\_data\_set() function.

protocol

the version of the MQTT protocol to use for this client. Can be either MQTTv31 or MQTTv311

**Example**

import paho.mqtt.client as mqtt

mqttc = mqtt.Client()

**reinitialise()**

reinitialise(client\_id="", clean\_session=True, userdata=None)

The reinitialise() function resets the client to its starting state as if it had just been created. It takes the same arguments as the Client() constructor.

**Example**

mqttc.reinitialise()

**Option functions**

These functions represent options that can be set on the client to modify its behaviour. In the majority of cases this must be done *before* connecting to a broker.

**max\_inflight\_messages\_set()**

max\_inflight\_messages\_set(self, inflight)

Set the maximum number of messages with QoS>0 that can be part way through their network flow at once.

Defaults to 20. Increasing this value will consume more memory but can increase throughput.

**message\_retry\_set()**

::

message\_retry\_set(retry)

Set the time in seconds before a message with QoS>0 is retried, if the broker does not respond.

This is set to 5 seconds by default and should not normally need changing.

**tls\_set()**

tls\_set(ca\_certs, certfile=None, keyfile=None, cert\_reqs=ssl.CERT\_REQUIRED,

tls\_version=ssl.PROTOCOL\_TLSv1, ciphers=None)

Configure network encryption and authentication options. Enables SSL/TLS support.

ca\_certs

a string path to the Certificate Authority certificate files that are to be treated as trusted by this client. If this is the only option given then the client will operate in a similar manner to a web browser. That is to say it will require the broker to have a certificate signed by the Certificate Authorities in ca\_certs and will communicate using TLS v1, but will not attempt any form of authentication. This provides basic network encryption but may not be sufficient depending on how the broker is configured.

certfile, keyfile

strings pointing to the PEM encoded client certificate and private keys respectively. If these arguments are not None then they will be used as client information for TLS based authentication. Support for this feature is broker dependent. Note that if either of these files in encrypted and needs a password to decrypt it, Python will ask for the password at the command line. It is not currently possible to define a callback to provide the password.

cert\_reqs

defines the certificate requirements that the client imposes on the broker. By default this is ssl.CERT\_REQUIRED, which means that the broker must provide a certificate. See the ssl pydoc for more information on this parameter.

tls\_version

specifies the version of the SSL/TLS protocol to be used. By default TLS v1 is used. Previous versions (all versions beginning with SSL) are possible but not recommended due to possible security problems.

ciphers

a string specifying which encryption ciphers are allowable for this connection, or None to use the defaults. See the ssl pydoc for more information.

Must be called before connect\*().

**tls\_insecure\_set()**

tls\_insecure\_set(value)

Configure verification of the server hostname in the server certificate.

If value is set to True, it is impossible to guarantee that the host you are connecting to is not impersonating your server. This can be useful in initial server testing, but makes it possible for a malicious third party to impersonate your server through DNS spoofing, for example.

Do not use this function in a real system. Setting value to True means there is no point using encryption.

Must be called before connect\*).

**username\_pw\_set()**

username\_pw\_set(username, password=None)

Set a username and optionally a password for broker authentication. Must be called before connect\*().

**user\_data\_set()**

::

user\_data\_set(userdata)

Set the private user data that will be passed to callbacks when events are generated. Use this for your own purpose to support your application.

**will\_set()**

::

will\_set(topic, payload=None, qos=0, retain=False)

Set a Will to be sent to the broker. If the client disconnects without calling disconnect(), the broker will publish the message on its behalf.

topic

the topic that the will message should be published on.

payload

the message to send as a will. If not given, or set to None a zero length message will be used as the will. Passing an int or float will result in the payload being converted to a string representing that number. If you wish to send a true int/float, use struct.pack() to create the payload you require.

qos

the quality of service level to use for the will.

retain

if set to True, the will message will be set as the "last known good"/retained message for the topic.

Raises a ValueError if qos is not 0, 1 or 2, or if topic is None or has zero string length.

**Connect / reconnect / disconnect**

**connect()**

connect(host, port=1883, keepalive=60, bind\_address="")

The connect() function connects the client to a broker. This is a blocking function. It takes the following arguments:

host

the hostname or IP address of the remote broker

port

the network port of the server host to connect to. Defaults to 1883. Note that the default port for MQTT over SSL/TLS is 8883 so if you are using tls\_set() the port may need providing manually

keepalive

maximum period in seconds allowed between communications with the broker. If no other messages are being exchanged, this controls the rate at which the client will send ping messages to the broker

bind\_address

the IP address of a local network interface to bind this client to, assuming multiple interfaces exist

**Callback**

When the client receives a CONNACK message from the broker in response to the connect it generates an on\_connect() callback.

**Example**

mqttc.connect("iot.eclipse.org")

**connect\_async()**

connect\_async(host, port=1883, keepalive=60, bind\_address="")

Identical to connect(), but non-blocking. The connection will not complete until one of the loop\*() functions is called.

**Callback**

When the client receives a CONNACK message from the broker in response to the connect it generates an on\_connect() callback.

**connect\_srv()**

connect\_srv(domain, keepalive=60, bind\_address="")

Connect to a broker using an SRV DNS lookup to obtain the broker address. Takes the following arguments:

domain

the DNS domain to search for SRV records. If None, try to determine the local domain name.

See connect() for a description of the keepalive and bind\_address arguments.

**Callback**

When the client receives a CONNACK message from the broker in response to the connect it generates an on\_connect() callback.

**Example**

mqttc.connect\_srv("eclipse.org")

**reconnect()**

reconnect()

Reconnect to a broker using the previously provided details. You must have called connect\*() before calling this function.

**Callback**

When the client receives a CONNACK message from the broker in response to the connect it generates an on\_connect() callback.

**disconnect()**

disconnect()

Disconnect from the broker cleanly. Using disconnect() will not result in a will message being sent by the broker.

**Callback**

When the client has sent the disconnect message it generates an on\_disconnect() callback.

**Network loop**

These functions are the driving force behind the client. If they are not called, incoming network data will not be processed and outgoing network data may not be sent in a timely fashion. There are four options for managing the network loop. Three are described here, the fourth in "External event loop support" below. Do not mix the different loop functions.

**loop()**

loop(timeout=1.0, max\_packets=1)

Call regularly to process network events. This call waits in select() until the network socket is available for reading or writing, if appropriate, then handles the incoming/outgoing data. This function blocks for up to timeout seconds. timeout must not exceed the keepalive value for the client or your client will be regularly disconnected by the broker.

The max\_packets argument is obsolete and should be left unset.

**Example**

run = True

while run:

mqttc.loop()

**loop\_start() / loop\_stop()**

loop\_start()

loop\_stop(force=False)

These functions implement a threaded interface to the network loop. Calling loop\_start() once, before or after connect\*(), runs a thread in the background to call loop() automatically. This frees up the main thread for other work that may be blocking. This call also handles reconnecting to the broker. Call loop\_stop() to stop the background thread. The force argument is currently ignored.

**Example**

mqttc.connect("iot.eclipse.org")

mqttc.loop\_start()

while True:

temperature = sensor.blocking\_read()

mqttc.publish("paho/temperature", temperature)

**loop\_forever()**

loop\_forever(timeout=1.0, max\_packets=1)

This is a blocking form of the network loop and will not return until the client calls disconnect(). It automatically handles reconnecting.

The timeout and max\_packets arguments are obsolete and should be left unset.

**Publishing**

Send a message from the client to the broker.

**publish()**

publish(topic, payload=None, qos=0, retain=False)

This causes a message to be sent to the broker and subsequently from the broker to any clients subscribing to matching topics. It takes the following arguments:

topic

the topic that the message should be published on

payload

the actual message to send. If not given, or set to None a zero length message will be used. Passing an int or float will result in the payload being converted to a string representing that number. If you wish to send a true int/float, use struct.pack() to create the payload you require

qos

the quality of service level to use

retain

if set to True, the message will be set as the "last known good"/retained message for the topic.

Returns a tuple (result, mid), where result is MQTT\_ERR\_SUCCESS to indicate success or MQTT\_ERR\_NO\_CONN if the client is not currently connected. mid is the message ID for the publish request. The mid value can be used to track the publish request by checking against the mid argument in the on\_publish() callback if it is defined.

A ValueError will be raised if topic is None, has zero length or is invalid (contains a wildcard), if qos is not one of 0, 1 or 2, or if the length of the payload is greater than 268435455 bytes.

**Callback**

When the message has been sent to the broker an on\_publish() callback will be generated.

**Subscribe / Unsubscribe**

**subscribe()**

subscribe(topic, qos=0)

Subscribe the client to one or more topics.

This function may be called in three different ways:

**Simple string and integer**

e.g. subscribe("my/topic", 2)

topic

a string specifying the subscription topic to subscribe to.

qos

the desired quality of service level for the subscription. Defaults to 0.

**String and integer tuple**

e.g. subscribe(("my/topic", 1))

topic

a tuple of (topic, qos). Both topic and qos must be present in the tuple.

qos

not used.

**List of string and integer tuples**

e.g. subscribe([("my/topic", 0), ("another/topic", 2)])

This allows multiple topic subscriptions in a single SUBSCRIPTION command, which is more efficient than using multiple calls to subscribe().

topic

a list of tuple of format (topic, qos). Both topic and qos must be present in all of the tuples.

qos

not used.

The function returns a tuple (result, mid), where result is MQTT\_ERR\_SUCCESS to indicate success or (MQTT\_ERR\_NO\_CONN, None) if the client is not currently connected. mid is the message ID for the subscribe request. The mid value can be used to track the subscribe request by checking against the mid argument in the on\_subscribe() callback if it is defined.

Raises a ValueError if qos is not 0, 1 or 2, or if topic is None or has zero string length, or if topic is not a string, tuple or list.

**Callback**

When the broker has acknowledged the subscription, an on\_subscribe() callback will be generated.

**unsubscribe()**

unsubscribe(topic)

Unsubscribe the client from one or more topics.

topic

a single string, or list of strings that are the subscription topics to unsubscribe from.

Returns a tuple (result, mid), where result is MQTT\_ERR\_SUCCESS to indicate success, or (MQTT\_ERR\_NO\_CONN, None) if the client is not currently connected. mid is the message ID for the unsubscribe request. The mid value can be used to track the unsubscribe request by checking against the mid argument in the on\_unsubscribe() callback if it is defined.

Raises a ValueError if topic is None or has zero string length, or is not a string or list.

**Callback**

When the broker has acknowledged the unsubscribe, an on\_unsubscribe() callback will be generated.

**Callbacks**

**on\_connect()**

on\_connect(client, userdata, rc)

Called when the broker responds to our connection request.

client

the client instance for this callback

userdata

the private user data as set in Client() or userdata\_set()

rc

the connection result

The value of rc indicates success or not:

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.

**Example**

def on\_connect(client, userdata, rc):

print("Connection returned result: "+connack\_string(rc))

mqttc.on\_connect = on\_connect

...

**on\_disconnect()**

on\_disconnect(client, userdata, rc)

Called when the client disconnects from the broker.

client

the client instance for this callback

userdata

the private user data as set in Client() or userdata\_set()

rc

the disconnection result

The rc parameter indicates the disconnection state. If MQTT\_ERR\_SUCCESS (0), the callback was called in response to a disconnect() call. If any other value the disconnection was unexpected, such as might be caused by a network error.

**Example**

def on\_disconnect(client, userdata, rc):

if rc != 0:

print("Unexpected disconnection.")

mqttc.on\_disconnect = on\_disconnect

...

**on\_message()**

on\_message(client, userdata, message)

Called when a message has been received on a topic that the client subscribes to.

client

the client instance for this callback

userdata

the private user data as set in Client() or userdata\_set()

message

an instance of MQTTMessage. This is a class with members topic, payload, qos, retain.

**Example**

def on\_message(client, userdata, message):

print("Received message '" + str(message.payload) + "' on topic '"

+ message.topic + "' with QoS " + str(message.qos))

mqttc.on\_message = on\_message

...

**on\_publish()**

on\_publish(client, userdata, mid)

Called when a message that was to be sent using the publish() call has completed transmission to the broker. For messages with QoS levels 1 and 2, this means that the appropriate handshakes have completed. For QoS 0, this simply means that the message has left the client. The mid variable matches the mid variable returned from the corresponding publish() call, to allow outgoing messages to be tracked.

This callback is important because even if the publish() call returns success, it does not always mean that the message has been sent.

**on\_subscribe()**

on\_subscribe(client, userdata, mid, granted\_qos)

Called when the broker responds to a subscribe request. The mid variable matches the mid variable returned from the corresponding subscribe() call. The granted\_qos variable is a list of integers that give the QoS level the broker has granted for each of the different subscription requests.

**on\_unsubscribe()**

on\_unsubscribe(client, userdata, mid)

Called when the broker responds to an unsubscribe request. The mid variable matches the mid variable returned from the corresponding unsubscribe() call.

**on\_log()**

on\_log(client, userdata, level, buf)

Called when the client has log information. Define to allow debugging. The level variable gives the severity of the message and will be one of MQTT\_LOG\_INFO, MQTT\_LOG\_NOTICE, MQTT\_LOG\_WARNING, MQTT\_LOG\_ERR, and MQTT\_LOG\_DEBUG. The message itself is in buf.

**External event loop support**

**loop\_read()**

loop\_read(max\_packets=1)

Call when the socket is ready for reading. max\_packets is obsolete and should be left unset.

**loop\_write()**

loop\_write(max\_packets=1)

Call when the socket is ready for writing. max\_packets is obsolete and should be left unset.

**loop\_misc()**

loop\_misc()

Call every few seconds to handle message retrying and pings.

**socket()**

socket()

Returns the socket object in use in the client to allow interfacing with other event loops.

**want\_write()**

want\_write()

Returns true if there is data waiting to be written, to allow interfacing the client with other event loops.

**Global helper functions**

The client module also offers some global helper functions.

topic\_matches\_sub(sub, topic) can be used to check whether a topic matches a subscription.

For example:

the topic foo/bar would match the subscription foo/# or +/bar

the topic non/matching would not match the subscription non/+/+

connack\_string(connack\_code) returns the error string associated with a CONNACK result.

error\_string(mqtt\_errno) returns the error string associated with a Paho MQTT error number.

**Publish**

This module provides some helper functions to allow straightforward publishing of messages in a one-shot manner. In other words, they are useful for the situation where you have a single/multiple messages you want to publish to a broker, then disconnect with nothing else required.

The two functions provided are single() and multiple().

**Single**

Publish a single message to a broker, then disconnect cleanly.

single(topic, payload=None, qos=0, retain=False, hostname="localhost",

port=1883, client\_id="", keepalive=60, will=None, auth=None, tls=None,

protocol=mqtt.MQTTv31)

**Function arguments**

topic

the only required argument must be the topic string to which the payload will be published.

payload

the payload to be published. If "" or None, a zero length payload will be published.

qos

the qos to use when publishing, default to 0.

retain

set the message to be retained (True) or not (False).

hostname

a string containing the address of the broker to connect to. Defaults to localhost.

port

the port to connect to the broker on. Defaults to 1883.

client\_id

the MQTT client id to use. If "" or None, the Paho library will generate a client id automatically.

keepalive

the keepalive timeout value for the client. Defaults to 60 seconds.

will

a dict containing will parameters for the client:

will = {'topic': "<topic>", 'payload':"<payload">, 'qos':<qos>, 'retain':<retain>}.

Topic is required, all other parameters are optional and will default to None, 0 and False respectively.

Defaults to None, which indicates no will should be used.

auth

a dict containing authentication parameters for the client:

auth = {'username':"<username>", 'password':"<password>"}

Username is required, password is optional and will default to None if not provided.

Defaults to None, which indicates no authentication is to be used.

tls

a dict containing TLS configuration parameters for the client:

dict = {'ca\_certs':"<ca\_certs>", 'certfile':"<certfile>", 'keyfile':"<keyfile>", 'tls\_version':"<tls\_version>", 'ciphers':"<ciphers">}

ca\_certs is required, all other parameters are optional and will default to None if not provided, which results in the client using the default behaviour - see the paho.mqtt.client documentation.

Defaults to None, which indicates that TLS should not be used.

protocol

choose the version of the MQTT protocol to use. Use either MQTTv31 or MQTTv311.

**Example**

import paho.mqtt.publish as publish

publish.single("paho/test/single", "payload", hostname="iot.eclipse.org")

**Multiple**

Publish multiple messages to a broker, then disconnect cleanly.

multiple(msgs, hostname="localhost", port=1883, client\_id="", keepalive=60,

will=None, auth=None, tls=None, protocol=mqtt.MQTTv31)

**Function arguments**

msgs

a list of messages to publish. Each message is either a dict or a tuple.

If a dict, only the topic must be present. Default values will be used for any missing arguments. The dict must be of the form:

msg = {'topic':"<topic>", 'payload':"<payload>", 'qos':<qos>, 'retain':<retain>}

topic must be present and may not be empty. If payload is "", None or not present then a zero length payload will be published. If qos is not present, the default of 0 is used. If retain is not present, the default of False is used.

If a tuple, then it must be of the form:

("<topic>", "<payload>", qos, retain)

See single() for the description of hostname, port, client\_id, keepalive, will, auth, tls, protocol.

**Example**

import paho.mqtt.publish as publish

msgs = [{'topic':"paho/test/multiple", 'payload':"multiple 1"},

("paho/test/multiple", "multiple 2", 0, False)]

publish.multiple(msgs, hostname="iot.eclipse.org")