Python Client - documentation

Contents

- Installation
- **Usage and API**
  - **Client**
    - Constructor / reinitialise
    - Option functions
    - Connect / reconnect / disconnect
    - Network loop
    - Publishing
    - Subscribe / Unsubscribe
    - Callbacks
    - External event loop support
    - Global helper functions
  - **Publish**
    - Single
    - Multiple
  - **Subscribe**
    - Simple
    - Using Callback
- Reporting bugs
- More information

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 https://github.com/eclipse/paho.mqtt.python
```

Once you have the code, it can be installed from your repository as well:

```
cd paho.mqtt.python
python setup.py install
```

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.

Getting Started

Here is a very simple example that subscribes to the broker $SYS topic tree and prints out the resulting messages:

```
import paho.mqtt.client as mqtt

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("$SYS/#")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("iot.eclipse.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

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=MQTTv311, transport="tcp")
```

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`
transport
set to "websockets" to send MQTT over WebSockets. Leave at the default of "tcp" to use raw TCP.

Constructor 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.

Reinitialise 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.

max_queued_messages_set()

```
max_queued_messages_set(self, queue_size)
```

Set the maximum number of outgoing messages with QoS>0 that can be pending in the outgoing message queue.

Defaults to 0. 0 means unlimited. When the queue is full, any further outgoing messages would be dropped.

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.

ws_set_options()

```
ws_set_options(self, path="/mqtt", headers=None)
```

Set websocket connection options. These options will only be used if `transport="websockets"` was passed into the `Client()` constructor.

path
The mqtt path to use on the broker.
headers
Either a dictionary specifying a list of extra headers which should be appended to the standard websocket headers, or a callable that takes the normal websocket headers and returns a new dictionary with a set of headers to connect to the broker.

Must be called before `connect*()`. An example of how this can be used with the AWS IoT platform is in the **examples** folder.

tls_set()

```
tls_set(ca_certs=None, certfile=None, keyfile=None, cert_reqs=ssl.CERT_REQUIRED,
    tls_version=ssl.PROTOCOL_TLS, 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. By default, on Python 2.7.9+ or 3.4+, the default certification authority of the system is used. On older Python version this parameter is mandatory.

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 (if the python version supports it) the highest TLS version is detected. If unavailable, 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_set_context()

```
tls_set_context(context=None)
```

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

context
an ssl.SSLContext object. By default, this is given by `ssl.create_default_context()`, if available (added in Python 3.4).

If you're unsure about using this method, then either use the default context, or use the `tls_set` method. See the ssl module documentation section about security considerations (https://docs.python.org/3/library/ssl.html#ssl-security) 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*()` and after `tls_set()` or `tls_set_context()`.

enable_logger()

```
enable_logger(logger=None)
```

Enable logging using the standard python logging package (See PEP 282). This may be used at the same time as the on_log callback method.

If `logger` is specified, then that `logging.Logger` object will be used, otherwise one will be created automatically.

Paho logging levels are converted to standard ones according to the following mapping:

Paho	logging
`MQTT_LOG_ERR`	`logging.ERROR`
`MQTT_LOG_WARNING`	`logging.WARNING`
`MQTT_LOG_NOTICE`	`logging.INFO` *(no direct equivalent)*
`MQTT_LOG_INFO`	`logging.INFO`
`MQTT_LOG_DEBUG`	`logging.DEBUG`

disable_logger()

```
disable_logger()
```

Disable logging using standard python logging package. This has no effect on the on_log callback.

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.

reconnect_delay_set

```
reconnect_delay_set(min_delay=1, max_delay=120)
```

The client will automatically retry connection. Between each attempt it will wait a number of seconds between `min_delay` and `max_delay`.

When the connection is lost, initially the reconnection attempt is delayed of `min_delay` seconds. It's doubled between subsequent attempt up to `max_delay`.

The delay is reset to `min_delay` when the connection complete (e.g. the CONNACK is received, not just the TCP connection is established).

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()` or `tls_set_context()`, 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.

Connect Example

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

connect_async()

```
connect_async(host, port=1883, keepalive=60, bind_address="")
```

Use in conjunction with `loop_start()` to connect in a non-blocking manner. The connection will not complete until `loop_start()` is called.

Callback (connect)

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 (connect_srv)

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

SRV Connect 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 (reconnect)

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.

Disconnect will not wait for all queued message to be sent, to ensure all messages are delivered, `wait_for_publish()` from `MQTTMessageInfo` should be used. See `publish()` for details.

Callback (disconnect)

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.

Loop 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.

Loop Start/Stop 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, retry_first_connection=False)
```

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

Except for the first connection attempt when using connect_async, use `retry_first_connection=True` to make it retry the first connection. Warning: This might lead to situations where the client keeps connecting to an non existing host without failing.

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 MQTTMessageInfo which expose the following attributes and methods:

- `rc`, the result of the publishing. It could be `MQTT_ERR_SUCCESS` to indicate success, `MQTT_ERR_NO_CONN` if the client is not currently connected, or `MQTT_ERR_QUEUE_SIZE` when `max_queued_messages_set` is used to indicate that message is neither queued nor sent.
- `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. `wait_for_publish` may be easier depending on your use-case.
- `wait_for_publish()` will block until the message is published. It will raise ValueError if the message is not queued (rc == `MQTT_ERR_QUEUE_SIZE`).
- `is_published` returns True if the message has been published. It will raise ValueError if the message is not queued (rc == `MQTT_ERR_QUEUE_SIZE`).

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 (publish)

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 (subscribe)

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 (unsubscribe)

When the broker has acknowledged the unsubscribe, an `on_unsubscribe()` callback will be generated.

Callbacks

on_connect()

```
on_connect(client, userdata, flags, 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 `user_data_set()`
flags
response flags sent by the broker
rc
the connection result
flags is a dict that contains response flags from the broker:
flags['session present'] - this flag is useful for clients that are
using clean session set to 0 only. If a client with clean session=0, that reconnects to a broker that it has previously connected to, this flag indicates whether the broker still has the session information for the client. If 1, the session still exists.

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.

On Connect Example

```
def on_connect(client, userdata, flags, 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 `user_data_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.

On Disconnect 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 and the message does not match an existing topic filter callback. Use `message_callback_add()` to define a callback that will be called for specific topic filters. `on_message` will serve as fallback when none matched.

client
the client instance for this callback
userdata
the private user data as set in `Client()` or `user_data_set()`
message
an instance of MQTTMessage. This is a class with members `topic`, `payload`, `qos`, `retain`.

On Message 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
...
```

message_callback_add()

This function allows you to define callbacks that handle incoming messages for specific subscription filters, including with wildcards. This lets you, for example, subscribe to `sensors/#` and have one callback to handle `sensors/temperature` and another to handle `sensors/humidity`.

```
message_callback_add(sub, callback)
```

sub
the subscription filter to match against for this callback. Only one callback may be defined per literal sub string
callback
the callback to be used. Takes the same form as the `on_message` callback.

If using `message_callback_add()` and `on_message`, only messages that do not match a subscription specific filter will be passed to the `on_message` callback.

If multiple sub match a topic, each callback will be called (e.g. sub `sensors/#` and sub `+/humidity` both match a message with a topic `sensors/humidity`, so both callbacks will handle this message).

message_callback_remove()

Remove a topic/subscription specific callback previously registered using `message_callback_add()`.

```
message_callback_remove(sub)
```

sub
the subscription filter to remove

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`.

This may be used at the same time as the standard Python logging, which can be enabled via the `enable_logger` method.

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.MQTTv311, transport="tcp")
```

Publish Single 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`.

transport
set to "websockets" to send MQTT over WebSockets. Leave at the default of "tcp" to use raw TCP.

Publish Single 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.MQTTv311, transport="tcp")
```

Publish Multiple 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`, `transport`.

Publish Multiple 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")
```

Subscribe

This module provides some helper functions to allow straightforward subscribing and processing of messages.

The two functions provided are `simple()` and `callback()`.

Simple

Subscribe to a set of topics and return the messages received. This is a blocking function.

```
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)
```

Simple Subscribe Function arguments

topics
the only required argument is the topic string to which the client will subscribe. This can either be a string or a list of strings if multiple topics should be subscribed to.
qos
the qos to use when subscribing, defaults to 0.
msg_count
the number of messages to retrieve from the broker. Defaults to 1. If 1, a single MQTTMessage object will be returned. If >1, a list of MQTTMessages will be returned.
retained
set to True to consider retained messages, set to False to ignore messages with the retained flag set.
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`.

Simple Example

```
import paho.mqtt.subscribe as subscribe

msg = subscribe.simple("paho/test/simple", hostname="iot.eclipse.org")
print("%s %s" % (msg.topic, msg.payload))
```

Using Callback

Subscribe to a set of topics and process the messages received using a user provided callback.

```
callback(callback, topics, qos=0, userdata=None, hostname="localhost",
    port=1883, client_id="", keepalive=60, will=None, auth=None, tls=None,
    protocol=mqtt.MQTTv311)
```

Callback Subscribe Function arguments

callback
an "on_message" callback that will be used for each message received, and of the form

> def on_message(client, userdata, message)

topics
the topic string to which the client will subscribe. This can either be a string or a list of strings if multiple topics should be subscribed to.
qos
the qos to use when subscribing, defaults to 0.
userdata
a user provided object that will be passed to the on_message callback when a message is received.

See `simple()` for the description of `hostname`, `port`, `client_id`, `keepalive`, `will`, `auth`, `tls`, `protocol`.

Callback Example

```
import paho.mqtt.subscribe as subscribe

def on_message_print(client, userdata, message):
    print("%s %s" % (message.topic, message.payload))

subscribe.callback(on_message_print, "paho/test/callback", hostname="iot.eclipse.org")
```

Reporting bugs

Please report bugs in the issues tracker at https://github.com/eclipse/paho.mqtt.python/issues (https://github.com/eclipse/paho.mqtt.python/issues).

More information

Discussion of the Paho clients takes place on the Eclipse paho-dev mailing list (https://dev.eclipse.org/mailman/listinfo/paho-dev).

General questions about the MQTT protocol are discussed in the MQTT Google Group (https://groups.google.com/forum/?fromgroups#!forum/mqtt).

There is much more information available via the MQTT community site (http://mqtt.org/).