Part 56
-
MQTT

Version: 2021-02-06

What is MQTT?

MQTT stands for Message Queuing Telemetry Transport.
MQTT is a lightweight publish & subscribe messaging protocol designed for machine to machine (M2M) telemetry in low bandwidth environments.
It was designed by Andy Stanford-Clark and Arlen Nipper in 1999 for connecting Oil Pipeline telemetry systems over satellite. Although it started as a proprietary protocol it was released royalty free in 2010 and became an OASIS standard in 2014.

MQTT is fast becoming one of the main protocols for IOT (internet of things) deployments.

MQTT Versions
There are two different variants of MQTT and several versions.

- MQTT v3.1.1 : In Common Use
- MQTT v5 : Currently Limited use
- MQTT-SN : See notes later

The original MQTT has been in use for many years and is designed for TCP/IP networks. MQTT v3.1.1 is version in common use. There is very little difference between v3.10 and 3.1.1.
The latest MQTT version (v5), has now been approved in Jan 2018.
As of release 1.6 the mosquitto broker supports MQTT v5 in addition to MQTT v3.11.
You can continue to use older version 3.11 client with the latest broker.
The Paho Python client v 1.5 now supports for v5.

MQTT-SN which was specified in around 2013, and designed to work over UDP, ZigBee and other transports. MQTT-SN doesn't currently appear to be very popular. and the specification hasn't changed for several years, but I expect that to change as IOT deployments start.

MQTT Clients
Because MQTT clients don't have addresses like email addresses, phone numbers etc. you don't need to assign addresses to clients like you do with most messaging systems.
For MQTT v3.1.1 there is client software available in almost all programming languages and for the main operating systems Linux, Windows, Mac from the Eclipse Paho project.

- Paho Python client.
- Node.js MQTT Client

The Paho client v1.5.1 now supports MQTT v5.0

MQTT Brokers or Servers
The original term was broker but it has now been standardized as Server. You will see both terms used. There are many MQTT srvers available that you can use for testing and for real applications. There are free self hosted brokers, the most popular being Mosquitto and commercial ones like HiveMQ.
Mosquitto is a free open source MQTT server that runs on Windows and Linux.
If you don't want to install and manage your own broker you can use a cloud based broker.
Eclipse has a free public MQTT broker and COAP server that you can also use for testing.

MQTT Over WebSockets
Websockets allows you to receive MQTT data directly into a web browser.
This is important as the web browser may become the de-facto interface for displaying MQTT data. MQTT websocket support for web browsers is provided by the Javascript MQTT Client.

MQTT Security
MQTT supports various authentications and data security mechanisms.It is important to note that these security mechanisms are configured on the MQTT broker, and it is up to the client to comply with the mechanisms in place.

Basic information
If you are familiar with the web and email then you will probably find that MQTT is very different. There is some basic information

- The standard port is 1883.
- You can not use MQTT without a broker
- The standard version uses TCP/IP.
- You can use multiple clients publish to the same topic
- It is not possible to know the identity of the client that published a message, unless the client includes that information in the topic or payload.
- Messages that get published to topics that no one subscribes to are discarded by the broker.
- You can't find easily the topics that have been published as the broker doesn't seem to keep a list of published topics as they aren't permanent.
- You can subscribe to a topic that no one is publishing to
- Messages can be stored on the broker but only temporarily. Once they have been sent to all subscribers they are then discarded. When you publish a message you can have the broker store the last published message. This message will be the first message that new subscribers see when they subscribe to that topic. MQTT only retains 1 message.

MQTT Basics

Clean Sessions and Persistent Connections

When a client connects to a server it can connect using either a non-persistent connection (clean session) or a persistent connection.
With a non-persistent connection the server doesn't store any subscription information or undelivered messages for the client. This mode is ideal when the client only publishes messages.
It can also connect as a durable client using a persistent connection. In this mode the server/server will, depending on the QOS of the published messages and subscribing client, store messages for the client if it is disconnected.

Retained Messages

Normally if a publisher publishes a message to a topic, and no one is subscribed to that topic the message is simply discarded by the server.
However the publisher can tell the server to keep the last message on that topic by setting the retained message flag. This can be very useful, as for example, if you have sensor publishing its status with long time intervals

Last Will and Testament

The last will and testament message is used to notify subscribers of an unexpected shut down of the publisher. Each topic can have a last will and testament message stored on the server.

Keep Alives

MQTT uses a TCP/IP connection which is normally left open by the client so that is can send and receive data at any time.
If no data flows over an open connection for a certain time period then the client will generate a PINGREQ and expect to receive a PINGRESP from the server. If this fails then the server considers the connection broken and closes it.

Authentication mechanisms

MQTT supports various authentications and data security mechanisms.
It is important to note that these security mechanisms are configured on the MQTT server, and it is up to the client to comply with the mechanisms in place.

Websockets

Websockets allows you to receive MQTT data directly into a web browser. This is important as the web browser may become the DE-facto interface for displaying MQTT data.
MQTT websocket support for web browsers is provided by the Javascript MQTT Client.

MQTT Topics

MQTT topics are a form of addressing that allows MQTT clients to share information.
MQTT Topics are structured in a hierarchy similar to folders and files in a file system using the forward slash / as a delimiter.
Using this system you can create a user friendly and self descriptive naming structures of you own choosing. Topic names are case sensitive and use UTF-8 strings.
Must consist of at least one character to be valid.

Except for the $SYS topic there is no default or standard topic structure.
That is there are no topics created on a server by default, except for the $SYS topic.
All topics are created by a subscribing or publishing client, and they are not permanent.
A topic only exists if a client has subscribed to it, or a server has a retained or last will messages stored for that topic.

The $SYS topic

This is a reserved topic and is used by most MQTT servers to publish information about the server. They are read-only topics for the MQTT clients. There is no standard for this topic structure but there is a guideline that most server implementations seem to follow.
the $SYS topics in Node-Red are:

TOPIC	VALUE
$SYS/broker/version	mosquitto version 1.4.10
$SYS/broker/timestamp	Fri, 22 Dec 2017 08:19:25 +0000
$SYS/broker/uptime	2251623 seconds
$SYS/broker/clients/total	2
$SYS/broker/clients/inactive	0
$SYS/broker/clients/disconnected	0
$SYS/broker/clients/active	2
$SYS/broker/clients/connected	2
$SYS/broker/clients/expired	0
$SYS/broker/clients/maximum	3
$SYS/broker/messages/stored	61
$SYS/broker/messages/received	419773

Subscribing to Topics
A client can subscribe to individual or multiple topics.
When subscribing to multiple topics two wildcard characters can be used. They are

- # (hash character) - multi level wildcard
- + (plus character) - single level wildcard

Wildcards can only be used to denote a level or multi-levels i.e /house/# and not as part of the name to denote multiple characters e.g. hou# is not valid topic naming
Examples of valid topic subscriptions

- Single topic subscriptions
  - /
  - /house
  - house/room/main-light
  - house/room/side-light

- Using Wildcards
  - Subscribing to topic house/# covers
    - house/room1/main-light
    - house/room1/alarm
    - house/garage/main-light
    - house/main-door
    - etc
  - Subscribing to topic house/+/main-light covers
    - house/room1/main-light
    - house/room2/main-light
    - house/garage/main-light

    but doesn't cover
    - house/room1/side-light
    - house/room2/side-light
  - Invalid Topic Subscriptions
    - house+ – Reason- no topic level
    - house# – Reason- no topic level

Publishing to Topics
A client can only publish to an individual topic. That is, using wildcards when publishing is not allowed. Eg. To publish a message to two topics you need to publish the message twice

When are Topics Created
Topics are created dynamically when:
- Someone subscribes to a topic
- Someone publishes a message to a topic with the retained message set to True.

When are Topics Removed from a Server
- When the last client that is subscribing to that server disconnects, and clean session is true.
- When a client connects with clean session set to True.

Republishing Topic Data
This is likely to be done when changing or combining naming schemes.
The idea is that a client would subscribe to a topic, e.g. hub1/sensor1 and republish the data using a new topic naming of house1/main-light.

MQTT Publish and Subscribe
In MQTT the process of sending messages is called publishing, and to receive messages an MQTT client must subscribe to an MQTT topic.

1. MQTT Publishing Basics

A client is free to publish on any topic it chooses. Currently there are no reserved topics. However servers can restrict access to topics. A client cannot publish a message to another client directly and doesn't know if any clients receive that message.
A client can only publish messages to a single topic, and cannot publish to a group of topics. However a message can be received by a group of clients if they subscribe to the same topic.

Message Flow and QOS on Published Messages
MQTT supports 3 QOS levels 0,1,2.
- QOS -0 – Default and doesn't guarantee message delivery.
- QOS -1 – Guarantees message delivery but could get duplicates.
- QOS -2 -Guarantees message delivery with no duplicates.

A message is published using one of these levels with QOS level 0 being the default.
If you want to try and ensure that the subscriber gets a message even though they might not be online then you need to publish with a quality of service of 1 or 2.
The schematic below shows the message flow between client and server for messages with QOS of 0, 1 and 2.

Messages published with a QOS of 1 and 2 are acknowledged by the server.
This results in several messages being sent.
Messages published with a QOS of 0 require only 1 message, and are not acknowledged by the server
Published messages with a QOS of 1 or 2 also have a Message ID number which can be used to track the message.

Publishing Messages and The Retain Flag
When a client publishes a message to a server it needs to send:
- The message topic
- The message QOS
- Whether the message should be retained.- Retain Flag

The retain Flag is normally set to False which means that the server doesn't keep the message. If you set the retain flag to True then the last message received by the server on that topic with the retained flag set will be kept.

The QOS of the published message has no effect on the retained message.
The main use of this is for sensors that don't change very much and publish their status infrequently. If you have a door sensor, for example, then it doesn't make much sense publishing it's status every second when it is almost always the same.
However if it only publishes it's status when it changes what happens when a subscriber subscribes to the sensor.
In this case if the last status was published without the retain flag set then the subscriber wouldn't know the status of the sensor until it published it again.

What Happens to Published Messages?

1 .What happens to the published message after the subscriber receives it?
2. What happens to the published message if there are no subscribers?
To answer these questions just think of a TV or radio broadcast.
If you aren't tuned into the broadcast you simply miss it!
So for question 1 and question 2 the answer is- The message is deleted from the server.
Explanation
When a client publishes a message on a topic then the server will distribute that message to any connected clients that have subscribed to that topic.
Once the message has been sent to those clients it is removed from the server
If no clients have subscribed to the topic or they aren't currently connected, then the message is removed from the server.
In general the server doesn't store messages.

Note**:** Retained messages, persistent connections and QOS levels can result in messages being stored temporarily on the server/server.

2. Subscribing To Topics

To receive messages on a topic you will need to subscribe to the topic or topics.
When you subscribe to a topics you also need to set the QOS of the topic subscription.
The QOS levels and their meaning are the same as those for the published messages.
When you subscribe to a topic or topics you are effectively telling the server to send you messages on that topic.
To send messages to a client the server uses the same publish mechanism as used by the client.
You can subscribe to multiple topics using two wildcard characters (+ and #)
All subscriptions are acknowledged by the server using a subscription acknowledge message that includes a packet identifier that can be used to verify the success of the subscription.

Mosquitto MQTT Server on Raspberry Pi

Installing The Mosquitto Server/Server

I am installing on Debian/Raspbian Buster

```
sudo apt -y update
sudo apt -y install mosquitto
sudo apt -y install mosquitto-clients
```

It should automatically start mosquitto. To stop and start the service use

```
sudo service mosquitto stop
sudo service mosquitto start
```

The stop/start scripts start the mosquitto server in the background and also use the default `mosquitto.conf` file in the `/etc/mosquitto/` folder.

```
# ls /etc/mosquitto/
ca_certificates  certs  conf.d  mosquitto.conf
```

The mosquitto binary is located in the `/usr/sbin` folder
If you want to see the control messages on the console then you need to start the mosquitto server manually from a command line.
You first need to to stop the server from running, and then type:

```
# mosquitto -v
1577992420: mosquitto version 1.5.7 starting
1577992420: Using default config.
1577992420: Opening ipv4 listen socket on port 1883.
1577992420: Opening ipv6 listen socket on port 1883.
```

Testing The Install

Note: you might need to install net-tools

```
apt -y install net-tools
```

To test it is running use command:

```
# netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:1883            0.0.0.0:*               LISTEN
tcp6       0      0 [::]:http               [::]:*                  LISTEN
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
tcp6       0      0 [::]:1883               [::]:*                  LISTEN
```

You should see the Mosquitto server running on port 1883 as shown in the screen shot above.

Starting Mosquitto Using a Configuration file
The configuration file `(mosquitto.conf)` that comes with the install only has a few lines for logging.
To start mosquitto using a configuration file use the following command:

```
mosquitto -c filename
```

You can find the `mosquitto.conf` template file in the `/etc/mosquitto/` folder.
It is a good idea to create a copy of this file before editing it.

Note**:** For testing it is easier to use a configuration file in your home directory rather than the `/etc/mosquitto` folder as you need root permissions to edit files in this folder.

Enabling Logging
This is useful for troubleshooting. Logging is already enabled in the default config file so all you need do is start mosquitto with this config file.

Important Note: on Linux although you stop mosquito from running using the command

```
sudo service mosquitto stop
```

to start it use:

```
mosquitto -c /etc/mosquitto/mosquitto.conf
```

You can also use a command line switch -v to enable logging:

```
mosquitto -v
```

Running Multiple Mosquitto Servers
You can configure a server to listen on several ports, but to create multiple servers with their own configurations then you will need to start multiple instances of mosquitto.

Examples:
Start mosquitto and listen on port 1883

```
mosquitto -p 1883
```

Start mosquitto as a daemon and listen on port 1884

```
mosquitto -p 1884 -d
```

Start mosquitto as a daemon and use the mosquitti-2.conf file.

```
mosquitto -c /etc/mosquitto/mosquitto-2.conf -d
```

Useful Linux Commands
To stop Mosquitto when running as a daemon:

```
pgrep mosquitto
kill -9 PID (that you get from above command)
```

Mosquitto Client Scripts
Once you have installed the clients tools using

```
sudo apt-get install mosquitto-clients
```

there is a simple subscriber client

```
mosquitto_sub
```

and a publisher client

```
mosquitto_pub
```

Use

```
mosquitto_sub --help
```

or

```
mosquitto_pub --help
```

Other Tools
MQTTlens is also very useful for troubleshooting and quick testing. It is an add-on for the chrome browser. It lets you publish and subscribe to topics using a web interface, and is much easier to use than the command line clients.

Beginners Guide To The Python Paho MQTT Client

We look at the main client object, and it's methods.
We will then create a simple Python example script that subscribes to a topic and publishes messages on that topic. If all goes well we should see the published messages. The example scripts are kept simple, and I don't include any error checking. I use my own locally installed server

Installing The Client
You can install the MQTT client using PIP with the command:

```
pip install paho-mqtt
```

The Python MQTT Client
The core of the client library is the client class which provides all of the functions to publish messages and subscribe to topics.

The paho mqtt client class has several methods.The main ones are:

- `connect()` and `disconnect()`
- `subscribe()` and `unsubscribe()`
- `publish()`

Each of these methods is associated with a callback**.**

Importing The Client Class
To use the client class you need to import it. Use the following

```
import paho.mqtt.client as mqtt
```

Creating a Client Instance
The client constructor takes 4 optional parameters, as shown below but only the client_id is necessary, and should be unique.

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

To create a instance use:

```
client =mqtt.Client(client_name)
```

Connecting To a Server or Server
Before you can publish messages or subscribe to topics you need to establish a connection to a server. To do this use the connect method of the Python mqtt client.
The method can be called with 4 parameters. The connect method declaration is shown below with the default parameters.

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

Note**:** You only need to supply the host (server name or IP address).

The general syntax is

```
client.connect(host_name)
```

Publishing Messages
Once you have a connection you can start to publish messages. To do this we use the publish method. The publish method accepts 4 parameters. The parameters are shown below with their default values.

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

The only parameters you must supply are the topic, and the payload.
The payload is the message you want to publish.
The general syntax is:

```
client.publish("house/light","ON")
```

Example Python Script:
We are now in a position to create our first Python Script to publish a message.
The script below publishes the message OFF to topic house/main-light

```
import paho.mqtt.client as mqtt          #import the client1
server_address="192.168.1.184"
#server_address="iot.eclipse.org"        #use external server
client = mqtt.Client("P1")               #create new instance
client.connect(server_address)           #connect to server
client.publish("house/main-light","OFF") #publish
```

Subscribing To Topics
To subscribe to a topic you use the subscribe method of the Paho MQTT Class object.
The subscribe method accepts 2 parameters – a topic or topics and a QOS (Quality of Service) as shown below with their default values.

```
subscribe(topic, qos=0)
```

We will now subscribe to topics and in this example we will subscribe to the topic house/bulb1 which is also the same topic that I will publishing on.
Doing this lets us see the messages we are publishing but we will need to subscribe before we publish. So our script outline becomes.

1. Create new client instance
2. Connect to server
3. Subscribe to topic
4. Publish message

Our new example script is shown below, and I have inserted some print statements to keep track of what is being done.

```
import paho.mqtt.client as mqtt          #import the client1
server_address="192.168.1.184"
#server_address="iot.eclipse.org"
print("creating new instance")
client = mqtt.Client("P1")               #create new instance
print("connecting to server")
client.connect(server_address)           #connect to server
print("Subscribing to topic","house/bulbs/bulb1")
client.subscribe("house/bulbs/bulb1")
print("Publishing message to topic","house/bulbs/bulb1")
client.publish("house/bulbs/bulb1","OFF")
```

When a client subscribes to a topic it is basically telling the server to send messages to it that are sent to the server on that topic.
The server is, in effect, publishing messages on that topic.
When the client receives messages it generate the on_message callback.
To view those messages we need to activate and process the on_message callback.

Callbacks also depend on the client loop which is covered in below.
However at this stage it may be better to just except them and proceed with the script.
To process callbacks you need to:

1. Create callback functions to process any messages received
2. Start a loop to check for callback messages.

The client docs describe the on_message callback and the parameters it excepts.
Here is my callback function, which basically just prints the received messages:

```
def on_message(client, userdata, message):
    print("message received ",str(message.payload.decode("utf-8")))
    print("message topic=",message.topic)
    print("message qos=",message.qos)
    print("message retain flag=",message.retain)
```

Note**:** the message parameter is a message class with members topic, qos, payload, retain.
I.e message.topic will give you the topic.

Now we need to attach our callback function to our client object as follows:

```
client.on_message=on_message    #attach function to callback
```

and finally we need to run a loop otherwise we won't see the callbacks. The simplest method is to use `loop_start()` as follows.

```
client.loop_start()  #start the loop
```

We also need to stop the loop at the end of the script (`loop_stop()`), and in addition wait a little to give the script time to process the callback, which we accomplish using the `time.sleep(4)` function.
This what our completed example script now looks like:

```
import paho.mqtt.client as mqtt          #import the client
import time

########################################
def on_message(client, userdata, message):
  print("message received ",str(message.payload.decode("utf-8")))
  print("message topic=",message.topic)
  print("message qos=",message.qos)
  print("message retain flag=",message.retain)
########################################

server_address="192.168.1.184"
print("creating new instance")
client = mqtt.Client("P1")               #create new instance
client.on_message=on_message             #attach function to callback
print("connecting to server")
client.connect(server_address)           #connect to server
client.loop_start()                      #start the loop
print("Subscribing to topic","house/bulbs/bulb1")
client.subscribe("house/bulbs/bulb1")
print("Publishing message to topic","house/bulbs/bulb1")
client.publish("house/bulbs/bulb1","OFF")
time.sleep(4)                            # wait
```

```
client.loop_stop()                          #stop the loop
```

Note**:** logically you should be able to start the loop before you create a client connection, but it you do then you get unexpected results.

Troubleshoot using Logging
To help troubleshoot your applications you can use the built in client logging callback.
To use it you create a function to process the logging callback. My function is shown below and it simply prints the log message.

```
def on_log(client, userdata, level, buf):
    print("log: ",buf)
```

and then attach it to the callback:

```
client.on_log=on_log
```

Common Problems
1. Not seeing any messages or not seeing all expected messages.

- You haven't started a network loop or called the loop() function. Or you haven't registered or created the callback functions.
- You haven't subscribed to the correct topics or subscription has failed.
- Access restrictions are in place.

2. My messages don't appear in the order I expected?

- The callback functions are async functions which can be called at any time. Use a queue to store the messages and print in one place. I use the Python logging module.

My base Python MQTT framework

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import time
import os
import os.path
import sys
import atexit
import subprocess
import getmac

import paho.mqtt.client   as mqtt

from datetime             import datetime        as dt

################################################################################
# Global vars
################################################################################

strScriptName         = os.path.basename(sys.argv[0])
strScriptBase         = strScriptName.replace(".py","")
strMACaddress         = getmac.get_mac_address().replace(':','').upper()

intWaitTime           = 60 # seconds, time to wait before doing another round

#global for the mqtt connection
mqtt_server           = "192.168.1.42"
mqtt_port             = 1883
mqtt_login            = ""
mqtt_password         = ""
mqtt_clientID         = strScriptBase.lower() + "-" + strMACaddress
mqtt_topicbase        = strScriptBase.lower() + "s/" + strScriptBase.lower() + "-" + strMACaddress
mqtt_topiccmd         = mqtt_topicbase + "/command"
mqtt_topicsts         = mqtt_topicbase + "/status"
mqtt_waitConnect      = 300  # time in seconds to wait for connect

################################################################################
# Functions
################################################################################

def mqtt_on_connect(mqttc, userdata, flags, rc):

    global mqtt_topicsts, mqtt_topiccmd

    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")

    if rc == 0:
        strMessage = timestamp + " - Connected"
        print(strMessage)
        mqttc.connected_flag = True
        mqtt_publish(mqtt_topicsts, "online")
        mqtt_subscribe(mqtt_topiccmd)

    else:
        strMessage = timestamp + " - Connection Failed, Error: " + str(rc)
        print(strMessage)


def mqtt_on_subscribe(mqttc, userdata, mid, granted_qos):

    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Subscribed: " + str(mid) + " - QoS:" + str(granted_qos)
    print(strMessage)


def mqtt_on_publish(mqttc, userdata, mid):

    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Published Message '" + mid + "'"
    print(strMessage)
```

```
def mqtt_on_message(mqttc, userdata, msg):

    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    payload    = msg.payload.decode("utf-8")
    strMessage = timestamp + " - Received Message -> <" + payload  + "> on topic '" + msg.topic + "'
with QoS " + str(msg.qos)
    print(strMessage)

    # do whatever is needed based on the payload received


def mqtt_on_disconnect(mqttc, userdata, rc):

    if mqttc.connected_flag:

        mqttc.connected_flag = False

        dteNow     = dt.now()
        timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")

        if rc == 0:
            strMessage = timestamp + " - Disonnected "
            print(strMessage)

        else:
            strMessage = timestamp + " - Disonnected Failed - Error Code : " + str(rc)
            print(strMessage)
    else:
        time.sleep(15)
        mqtt_reconnect()


def mqtt_connect():

    global mqttc, mqtt_clientID, mqtt_topicsts
    global mqtt_login, mqtt_password, mqtt_waitConnect

    # connect to MQTT server/broker
    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Connecting ..."
    print(strMessage)
    mqttc = mqtt.Client(mqtt_clientID, clean_session=False, userdata=None)
    if mqtt_login != "":
        mqttc.username_pw_set(mqtt_login, mqtt_password)
    mqttc.will_set(mqtt_topicsts, payload="offline", qos=1, retain=True)
    # Assign event callbacks
    mqttc.on_connect     = mqtt_on_connect
    mqttc.on_publish     = mqtt_on_publish
    mqttc.on_subscribe   = mqtt_on_subscribe
    mqttc.on_message     = mqtt_on_message
    mqttc.on_disconnect  = mqtt_on_disconnect
    mqttc.connected_flag = False
    try:
        mqttc.connect(mqtt_server, port=mqtt_port, keepalive=360)
        time.sleep(3)
        #start the loop in another tread
        mqttc.loop_start()

        dteNow  = dt.now()
        dteLoop = dt.now() + timedelta(seconds = mqtt_waitConnect)
        while dteNow < dteLoop:
            if not mqttc.connected_flag:
                time.sleep(5)
            else:
                return True

    except Exception as e:
        dteNow       = dt.now()
        timestamp    = dteNow.strftime("%Y-%m-%d %H:%M:%S")
        strMessage   = timestamp + " -    Error in mttc_connect: " + str(e)
        print(strMessage)
        # if here connection failed
        os.execv(sys.executable, ['python'] + sys.argv)
        destroy()

    else:
        # if here connection failed
        os.execv(sys.executable, ['python'] + sys.argv)
```

```
        destroy()


def mqtt_reconnect():

    global mqttc, mqtt_clientID, mqtt_topicsts
    global mqtt_login, mqtt_password, mqtt_waitConnect

    # connect to MQTT server/broker
    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Reconnecting ..."
    print(strMessage)
    mqttc.loop_stop()
    mqttc.reinitialise(mqtt_clientID, clean_session=False, userdata=None)
    if mqtt_login != "":
        mqttc.username_pw_set(mqtt_login, mqtt_password)
    mqttc.will_set(mqtt_topicsts, payload="offline", qos=1, retain=True)
    # Assign event callbacks
    mqttc.on_connect     = mqtt_on_connect
    mqttc.on_publish     = mqtt_on_publish
    mqttc.on_subscribe   = mqtt_on_subscribe
    mqttc.on_message     = mqtt_on_message
    mqttc.on_disconnect  = mqtt_on_disconnect
    mqttc.connected_flag = False
    try:
        mqttc.connect(mqtt_server, port=mqtt_port, keepalive=360)
        time.sleep(3)
        #start the loop in another tread
        mqttc.loop_start()

        dteNow  = dt.now()
        dteLoop = dt.now() + timedelta(seconds = mqtt_waitConnect)
        while dteNow < dteLoop:
            if not mqttc.connected_flag:
                time.sleep(5)
            else:
                return True
    except Exception as e:
        dteNow       = dt.now()
        timestamp    = dteNow.strftime("%Y-%m-%d %H:%M:%S")
        strMessage   = timestamp + " -    Error in mttc_connect: " + str(e)
        print(strMessage)
        # if here connection failed
        os.execv(sys.executable, ['python'] + sys.argv)
        destroy()

    else:
        # if here connection failed
        os.execv(sys.executable, ['python'] + sys.argv)
        destroy()


def mqtt_subscribe(topic):

    global mqttc

    while not mqttc.connected_flag:
        mqtt_reconnect()

    mqttc.subscribe(topic, qos=1)
    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Subscribed to " + topic
    print(strMessage)

def mqtt_publish(topic, data):

    global mqttc, mqtt_topicsts

    while not mqttc.connected_flag:
        mqtt_reconnect()

    if topic != mqtt_topicsts:
        mqttc.publish(mqtt_topicsts, "online", qos=1, retain=True)

    payload = str(data)

    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
```

```
    strMessage = timestamp + " - Publishing data " + ("<" + payload +">").ljust(15) + " to topic <"
+ topic + ">"
    print(strMessage)
    mqttc.publish(topic, payload, qos=1, retain=True)
    dteNow     = dt.now()
    timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
    strMessage = timestamp + " - Published data  " + ("<" + payload +">").ljust(15) + " to topic <"
+ topic + ">"
    print(strMessage)


ef destroy():

    global  mqttc

    mqtt_publish(mqtt_topicsts, "offline")
    mqttc.loop_stop()

    sys.exit(0)


################################################################################
# Main program loop
################################################################################
if __name__ == "__main__":

    # clear console screen
    os.system('cls' if os.name == 'nt' else 'clear')

    # set exit procedure
    atexit.register(destroy)

    strMessage = timestamp + " - >>> Press Ctrl+C to exit <<<" + "\n"
    print(strMessage)

    mqtt_connect()

    try:

        # main endless loop
        while True:

            dteNextLoop = dt.now() + timedelta(seconds = intWaitTime)

            dteNow    = dt.now()
            timestamp = dteNow.strftime("%Y-%m-%d %H:%M:%S")
            savelog()

            ''' should we sleep for a while '''
            dteNow = dt.now()
            if dteNow >= dteNextLoop:
                 # make sure payload is somewhere defined and valid
                topic = mqtt_topicbase + "/state"
                mqtt_publish(topic, payload)

            time.sleep(intWaitTime)

    except KeyboardInterrupt:
        dteNow     = dt.now()
        timestamp  = dteNow.strftime("%Y-%m-%d %H:%M:%S")
        strMessage = timestamp + " - Detected Ctrl-C interruption"
        print(("\r" + strMessage))
        savelog()

    destroy()
```

Appendix : Detailed information on Paho MQTT Client

Python Paho MQTT Client Objects

The main component of the Python Paho MQTT client library is the client class. The class provides all the necessary functions to connect to an MQTT server, publish messages, subscribe to topics and receive messages.
To create a new client object you first need to import the Paho MQTT client, and then create the object as shown below:

```
import paho.mqtt.client as mqtt
client1 = mqtt.Client()
```

The Client name
Although the client name is optional, it is only optional if clean sessions are True (default). However even if you don't provide a client name one is created for you by the client software. The screen shot below is taken from the MQTT server console and shows the result of the client connecting first without specifying a client id and then secondly supplying a client id. In both case the server sees a client id as the client will auto generate a random one.
The client name is used by the MQTT server to identify the client. This is necessary when using persistent connections as the server needs to store messages for the client when the client isn't connected.

Duplicate Client ids
If a client connects with a client id that is in use, and also currently connected then the existing connection is closed. Because the MQTT client may attempt to reconnect following a disconnect this can result in a loop of disconnect and connect. Therefore be careful when assigning client IDs.

Creating Unique Client ids
To create a unique client id you can use one of these popular schemes.

Client prefix + Serial number of device
Client prefix + MAC address of device
Client prefix + time+random number

Note**:** Use of a client prefix is recommended as servers can filter based on a client-id prefix.

Clean Session Flag
This flag tells the server to either remember subscriptions and store messages that the client has missed because it's offline-value is False or not to remember subscriptions and not to store messages that the client has missed because it's offline–valuec is True
By default it is set to True

Auxiliary Functions or Settings
There are several Client settings that may need to be changed before a connection is created. These settings are changed using auxiliary functions. Here is a list of the functions that are available:

`max_inflight_messages_set()`	– Affects Message throughput
`max_queued_messages_set()`	– Affects Message throughput
`message_retry_set(retry)`	– Affects Message throughput
`tls_set()`	– Used for SSL security
`tls_insecure_set(value)`	– Used for SSL security
`username_pw_set()`	– Used for username and passwords
`will_set()`	– Used for setting last will and testament

The most common used ones are `username_pw_set()`,`tls_set()`,and `will_set()`.

Websockets
websocket support is also built into the Paho MQTT client.
To Use Websockets with Python. Create the client object using the transport=websockets argument.

```
client= paho.Client("control",transport='websockets')
```

Simple Client Object Modifications
I usually add additional flags to the Client object and use them for detecting successful connections and subscribes. This I do before I create the client object. So often in my test scripts you will see an initialise client object function that looks like this.

```
import paho.mqtt.client as mqtt
def Initialise_clients(cname):
    #callback assignment
    client= mqtt.Client(cname,False)      #don't use clean session
    if mqttclient_log:                    #enable mqqt client logging
        client.on_log=on_log
    client.on_connect= on_connect         #attach function to callback
    client.on_message=on_message          #attach function to callback
    client.on_subscribe=on_subscribe
    #flags set
    client.topic_ack=[]
    client.run_flag=False
    client.running_loop=False
    client.subscribe_flag=False
    client.bad_connection_flag=False
    client.connected_flag=False
    client.disconnect_flag=False
    return client
```

or creating a sub class

```
import paho.mqtt.client as mqtt

class MQTTClient(mqtt.Client):

   def __init__(self,cname,**kwargs):
      super(MQTTClient, self).__init__(cname,**kwargs)
      self.last_pub_time=time.time()
      self.topic_ack=[]
      self.run_flag=True
      self.subscribe_flag=False
      self.bad_connection_flag=False
      self.connected_flag=True
      self.disconnect_flag=False
      self.disconnect_time=0.0
      self.pub_msg_count=0
      self.devices=[]
```

Python Paho MQTT Client Callbacks

Callbacks are functions that are called in response to an event.
The events and callbacks for the Paho MQTT client are as follows:

Event Connection acknowledged Triggers the on_connect callback
Event Disconnection acknowledged Triggers the on_disconnect callback
Event Subscription acknowledged Triggers the on_subscribe callback
Event Un-subscription acknowledged Triggers the on_unsubscribe callback
Event Publish acknowledged Triggers the on_publish callback
Event Message Received Triggers the on_message callback
Event Log information available Triggers the on_log callback

The Paho client is designed to only use the callback functions if they exist, and doesn't provide any default callback functions.
However it does provide a mechanism to set them.
To use a callback you need to do two things

1. Create the callback function
2. Assign the function to the callback.

Callback Function Example

As an example we will use the **on_connect** callback.
First I create my function. I've called it **on_connect** but I could have called it anything I wanted.
Here is the function.

```
def on_connect(client, userdata, flags, rc):
    logging.info("Connected flags"+str(flags)+"result code "\
    +str(rc)+"client1_id ")
    client.connected_flag=True
```

The function simply logs a message, and sets a flag.
Now to associate my function with the **On_connect callback** I use the following:

```
client.on_connect= on_connect
```

Note**:** If my function had been called myfunction then I would use the following:

```
client.on_connect= myfunction
```

Callbacks and the Client Loop

Callbacks are dependent on the client loop as without the loop the callbacks aren't triggered.
Below is the code of a simple python script that creates a connection and then waits in a loop for the on_connect callback to be triggered.
The on_connect callback then sets the flag that terminates the loop and the script ends.
This means that if the callback isn't processed then the while loop never ends as we shall see later.

```
import paho.mqtt.client as mqtt
import time

def on_connect (client, userdata, flags, rc}:
  global loop_flag
  printi" In on_connect callback ")
  loop_flag=0

broker_address="192.168.1.184"
#broker_address="iot.eclipse.org"
client = mqtt.Client()             #create new instance
client.on_connect=on_connect       #attach function to callback
```

```
client.connect (broker_address)     #connect to broker
client.loop_start()                 #start loop to process callbacks

loop_flag=1
counter=0
while loop_flag==1:
      print{"waiting for callback to occur ",counter)
      time.sleep{.01)               #pause 1/100 second
      counter+=1

client.disconnect{)
client.loop_stop{)
```

Here is the output generated by the script.
Notice how the loop executes waiting for the callback and stops once the callback gets processed

```
>>>
waiting for callback to occur  0
waiting for callback to occur  1
waiting for callback to occur  2
waiting for callback to occur  3
waiting for callback to occur  4
waiting for callback to occur  5
  In on_connect callback
```

In the script above the client.loop_start() method starts a loop to process callbacks.
So let's run the above script again, but this time I won't start the loop. To do that I simply comment out the client.start_loop() line.
Here is what happens when I run the script.

```
>>>
waiting for callback to occur  0
waiting for callback to occur  1
waiting for callback to occur  2
waiting for callback to occur  3
waiting for callback to occur  4
waiting for callback to occur  5
waiting for callback to occur  6
waiting for callback to occur  7
waiting for callback to occur  8
waiting for callback to occur  9
waiting for callback to occur  10
waiting for callback to occur  11
…
(interupted by hitting Control-C)
```

You can see that I had to manually terminate the script as it never processed the callback, and the callback is what terminates the while loop.

How Callbacks Work
If you dig into the Client code you will find the code for the Connect acknowledge message.
When the client receives a CONNACK message the callback is triggered if it exists.
Here is the relevant section of the code. (around line 276). Notice the if on_connect statement.
This is what checks for the callback, and if it exists it calls the callback- self.on_connect(). The self.on_connect function is the function we assigned earlier using.

```
client.on_connect= on_connect
```

You should see from the code that the function call passes several arguments, and so the function we create needs to be able to handle those arguments.
If you look back at the function I created you can see that it handles 4 arguments.

Asynchronous or Synchronous?
If you use the loop_start() or loop_forever functions then the loop runs in a separate thread, and it is the loop that processes the incoming and outgoing messages.
In this case the callbacks can occur any time in your script and are asynchronous.
However if you call the loop() function manually in the script then the callbacks are synchronous with your script as they can only occur when you call the loop() function.

Returning Values from Callbacks
Due to the way they operate It is not possible to return values from a callback using the standard return command.
Therefore if you need to get status information from a callback you need to use some form of global variable in the callback.
If you refer back to the on_connect call back example I used the connected_flag which is a property of the client object.
Because the client object is available throughout the script it can be accessed anywhere in the script.
In my scripts my general approach is extend the main mqtt client class with flags and other variable I need.
When I create client object it has these flags already assigned.
This is what it looks like:

```
import paho.mqtt.client as mqtt           #import mqqt client
### extend main class to include flags etc
mqtt.Client.bad_connection_flag=False
mqtt.Client.connected_flag=False
mqtt.Client.disconnect_flag=False
mqtt.Client.disconnect_time=0.0
mqtt.Client.pub_msg_count=0
##
client=mqtt.Client("myclient")            #create client object
###
client.client.connected_flag=True         #set flag to true in script
#####
```

Python Paho MQTT Client Connections

The MQTT client uses a TCP/IP connection to the server.
Once the connection is established the client can send data to the server, and the server can send data to the client as required.

The Connect Method

To establish a connection to an MQTT server using the Python client you use the connect method of the client object.
The method can be called with 4 parameters. The connect method declaration is shown below with the default parameters.

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

The only parameter you need to provide is the host name.
This can be the IP address or domain name of the server.

Note**:** you may need to setup other settings like passwords,last will and testament etc before connecting. The connect method is a blocking function which means that your script will stop while the connection is being established.

Was The Connection Attempt Successful?

When a client issues a connect request to a server that request should receive an acknowledgment.
The server acknowledgement will generate a callback (on_connect).
If you want to be sure that the connection attempt was successful then you will need to setup a function to handle this callback before you create the connection.
The function should receive 4 parameters, and can be called anything you want.
I have called mine on_connect().
Here is an example function definition:

```
def on_connect(client, userdata, flags, rc):
    if rc==0:
        print("connected OK Returned code=",rc)
    else:
        print("Bad connection Returned code=",rc)
```

The client is a client object.
rc (return code) is used for checking that the connection was established.

Note**:** It is also common to subscribe in the on_connect callback

Connection Return Codes

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.

Flags and userdata aren't normally used.

Processing The On_connect Callback

To process the callback you will need to run a loop.
Therefore the script generally looks like this.

1. Create Client object.
2. Create callback function on_connect()

3. Bind callback to callback function (on_connect())
4. Connect to Server.
5. Start a loop.

Because the callback function is asynchronous you don't know when it will be triggered.
What is sure however is that there is a time delay between the connection being created, and the callback being triggered.
It is important that your script doesn't proceed until the connection has been established.
For quick demo scripts I use time.sleep() to wait, and give the connection time to be established. However for working scripts I process the callback and use it to flag a successful or unsuccessful connection. So instead of this:

1. Create connection
2. Publish message
3. We have this:
4. Create connection
5. Verify successful connection or quit
6. Publish message and or subscribe

Here is some example code that uses time.sleep() to wait, and give the connection setup time to complete:

```
import paho.mqtt.client as mqtt                 #import client library
def on_connect(client, userdata, flags, rc):
   if rc==0
      print("connected ok")
client = mqtt.Client("python1")                 #create new instance
client.on_connect=onconnect                     #bind call back function
client.connect(server_address)                  #connect to server
client.loop_start()                             #Start loop
time.sleep(4)                                   # Wait for connection setup to
complete
… other code here
client.loop_stop()                              #Stop loop
```

Enhancing the Callback
To get better control of the connection I use a flag in the on_connect callback.
The flag I create as part of the client object so it is available throughout the script.
client.connected_flag=False. At the start of the script I set this flag (connected_flag) to False and toggle it to True when the Connection is successful, and back to False when we get a disconnect.

```
def on_connect(client, userdata, flags, rc):
    if rc==0:
        client.connected_flag=True              #set flag
        print("connected OK Returned code=",rc)
        #client.subscribe(topic)
    else:
        print("Bad connection Returned code= ",rc)
```

We can now use this flag to create a wait loop.

```
client.connect(server_address)                  #connect to server
while not client.connected_flag:                #wait in loop
     time.sleep(1)
```

Example Client Connection Script
The following script is a basic client connection script

```
#!python3
import paho.mqtt.client as mqtt                 #import the client1
import time

def on_connect(client, userdata, flags, rc):
    if rc==0:
        client.connected_flag=True              #set flag
        print("connected OK")
    else:
        print("Bad connection Returned code=",rc)

mqtt.Client.connected_flag=False                #create flag in class
server="192.168.1.184"
client = mqtt.Client("python1")                 #create new instance
client.on_connect=on_connect                    #bind call back function
client.loop_start()
print("Connecting to server ",server)
client.connect(server)                          #connect to server
while not client.connected_flag:                #wait in loop
    print("In wait loop")
    time.sleep(1)
print("in Main Loop")
client.loop_stop()                              #Stop loop
client.disconnect()                             #disconnect
```

If I run this script this is what I see:

```
Connecting to broker 192.168.1.184
In walt loop
In wait loop
connected OK
in main Loop
```

Failed Connection Examples
There are various conditions were the connection can fail to complete. They are:

Incorrect client settings e.g. bad password..
No network connection
Bad Network Connection parameters e.g. bad port number

It is important that these are detected and handled by the connection script.
We are going to look at a few of these and modify our connection code to detect them.

Note**:** For these examples I will use the Paho MQTT client and the Mosquitto server.

Connection Failures that Create an Exception
Trying to connect to a server using a bad IP address or port number will generate a socket error, and raise an exception.
This causes a Winsock error in Windows
In Python we can use a Try block to catch this so instead of

```
client.connect(server,port) #connect to server
```

We use

```
try:
    client1.connect(server,port) #connect to server
except:
    print("connection failed")
    exit(1) #Should quit or raise flag to quit or retry
```

When the connection attempt failed we would see:

```
creating client
connecting
connection failed
```

Connection Failures Detected Through Return Code
To determine if the connection was successful we need to examine the return code of the on_connect callback. A return code of 0 is successful, whereas other values indicate a failure. In the example below we will try to connect to a server without providing the required authentication.

```
>>> RESTART
creating client Success
connecting
connected
on_connect function—return code= 0
>>> RESTART

>>>
creating client
connecting
connected
on_connect function—return code= 5
on_connect function—return code= 5
on_connect function—return code= 5
on_connect function—return code= 5
on_connect function—return code= 5
on connect function—return code= 5
```

Notice the connection fails and returns a return code of 5 which indicates authentication failure. You should also notice that because I am using the loop_start() function the client will try to reconnect, but this is pointless as the result will be the same.

So our code should :

1. Stop the loop
2. Stop the script

We can stop the loop in the on_connect callback. However to stop the main script we need to set a flag that we can use to exit. I prefer to use a flag and stop the loop as part of the main script. Here is what the modified on_connect callback looks like:

```
def on_connect(client, userdata, flags, rc):
    if rc==0:
        client.connected_flag=True #set flag
        print("connected OK")
    else:
        print("Bad connection Returned code=",rc)
        client.bad_connection_flag=True
```

Here is the main script modifications to quit.

```
mqtt.Client.bad_connection_flag=False #
while not client.connected_flag and not client.bad_connection_flag: #wait in
loop
    print("In wait loop")
    time.sleep(1)
if client.bad_connection_flag:
    client.loop_stop()    #Stop loop
    sys.exit()
```

Using Authentication

If the server requires username and password authentication then you need to set this before connecting. This you do using the `username_pw_set()` helper function. e.g

```
client.username_pw_set(username="steve",password="password")
# now can connect
```

Connecting Using Websockets

Normally the python client will connect using MQTT but it can also connect using MQTT over websockets. To tell the client to use websockets instead of MQTT use the command

```
client= paho.Client("cname",transport='websockets')
```

instead of simply

```
client= paho.Client("cname")
```

You will also need to change the port.Websockets generally uses port 9001**.**

Handling Disconnects and Reconnects
A client can disconnect gracefully, if it has no more data to send by sending a disconnect message. The Paho client provides the disconnect method for this. It can also get disconnected due to a bad network connection. If the connection fails for some reason then you will need to decide whether or not you should try to reconnect.

A disconnect triggers the **on_disconnect** callback which you will need to examine.
This callback takes 3 parameters:

1. Client - Client object that disconnected
2. Userdata - user defined data not often used
3. Return Code (rc) - Indication of disconnect reason. 0 is normal all other values indicate abnormal disconnection

Here is the **on_disconnect()** code I use:

```
def on_disconnect(client, userdata, rc):
    logging.info("disconnecting reason  "  +str(rc))
    client.connected_flag=False
    client.disconnect_flag=True
```

You can see that I simply log it, and then set flags that can be used by the main program to detect the disconnect.

Note**:** You will need to be calling, or running a loop to trigger the callback.

Reconnecting
Generally you will need to reconnect as soon as possible.
If you run a network loop using loop_start() or loop_forever() then re-connections are automatically handled for you.
A new connection attempt is made automatically in the background every 3 to 6 seconds.
If you call the loop() function manually then you will need to handle the re-connection attempts yourself.
You can do this by using a connection flag that is toggled by the on_connect and on_disconnect callbacks.

Client Connection Summary
Taking into account the above our client connection code should.

1. Connect to server
2. Examine connection status and proceed if good
3. If connection status is bad attempt retry and or quit.
4. Handle disconnects and reconnects

Python Paho MQTT Client Loop

When writing code using the Paho Python client you would have had to use the loop() function.
When new messages arrive at the Python MQTT client they are placed in a receive buffer.
The messages sit in this receive buffer waiting to be read by the client program.
You could program the client to manually read the receive buffers but this would be tedious.
The loop() function is a built in function that will read the receive and send buffers, and process any messages it finds.
On the receive side it looks at the messages, and depending on the message type, it will trigger the appropriate callback function.
For example if it sees a CONNACK message it triggers the on_connect() callback.
Now instead of manually reading the receive buffer you just need to process the callbacks.
Outgoing messages and message acknowledgements are placed in the send buffer.
The loop function will read this buffer and send any messages it finds.

Calling the Loop Function

The Paho Python client provides three methods:

- loop_start()
- loop_forever() and
- loop().

The loop_start() starts a new thread, that calls the loop method at regular intervals for you. It also handles re-connects automatically.
To stop the loop use the loop_stop() method.
The loop_forever() method blocks the program, and is useful when the program must run indefinitely.
The loop_forever() function also handles automatic reconnects. The loop can be stopped by calling loop.stop().
You should stop the loop before you exit the script. You can also manually call the loop() method in your program. If you do this you must remember to call it regularly.
That is it must be in a loop. e.g pseudo code below:

```
while..
    some code
    client.loop(.1) #blocks for 100ms
    some code
```

Because the loop is a blocking function I call it with a timeout the default timeout is 1 second.
If you call the loop manually then you will need to create code to handle reconnects.
Important! If your client script has more than one client connection then you must call or start a loop for each client connection.
For example, if I create two clients client 1 and client2 in a script, then you would expect to see client1.loop() and client2.loop() in the script.

Implementation Note

I have experienced strange behaviour when starting a loop before creating a connection . So

```
client= mqtt.Client(cname)
client.connect(server,port))
client.loop_start()
Works Ok but
client= mqtt.Client(cname)
client.loop_start()
client.connect(server,port))
```

sometimes gives strange results.

Stopping the loop automatically
If you are using the loop_start() function then you will probably need to stop the loop automatically if the connection fails.
The easiest way of doing this is using the on_disconnect callback.

```
def on_disconnect(client, userdata,rc=0):
    logging.debug("DisConnected result code "+str(rc))
    client.loop_stop()
```

However you should only stop the loop if you are completely finished, and are going to exit.
Stopping the loop will stop auto reconnects unless you take steps to call the loop manually.

Loop_start vs Loop_forever
Loop_start starts a loop in another thread and lets the main thread continue if you need to do other things in the main thread then it is important that it doesn't end.
To accomplish this you need to use your own wait loop.
The loop_forever call blocks the main thread and so it will never terminate.
The loop_forever call must be placed at the very end of the main script code as it doesn't progress beyond it.
To terminate the script you will need to use a callback function to disconnect the client.

Handling Multiple Clients
If your script connects using multiple clients then each client will need a loop.
Therefore if you are using the loop_start() method then you will need to call it for each client connection.
Therefore if you have 2 client connections you will need two loops which equals two additional threads.
For 2 clients it isn't really a problem, but what if you have several hundred client connections then you will need several hundred additional threads.
In this situation it is better to manually call the loop for each client and use thread pooling.
I will cover this is another tutorial at a later date.
However for up to around 20 client connections then is is easier to use the inbuilt loop_start and stop functions.
To make it simpler add the clients to a list and loop through the list to start the loops and the same to stop e.g

```
import paho.mqtt.client as mqtt
clients=[]
nclients=20
mqtt.Client.connected_flag=False
#create clients
for i  in range(nclients):
   cname="Client"+str(i)
   client= mqtt.Client(cname)
   clients.append(client)
for client in clients:
  client.connect(server)
  client.loop_start()
```

Python Paho MQTT Client Authentication

You can restrict access to a server, and how you can protect your data using various security mechanisms. It is important to note that these security mechanisms are initiated by the server, and it s up to the client to comply with the mechanisms in place.
It is also important to realise that when planning security for your implementation that you must consider the capabilities of your MQTT clients as well as your server.
This tutorial will use the free Open source Mosquitto server, and the Paho Python MQTT client to illustrate these mechanisms.

Client Authentication

There are three ways that a Mosquitto server can verify the identity of an MQTT client:

- Client ids
- Usernames and passwords.
- Client Certificates

Client ids

All MQTT clients must provide a client id. When a client subscribes to a topic/topics the client id links the topic to the client and to the TCP connection.
With persistent connections the server remembers the client id and the subscribed topics.
When configuring an MQTT client you will need to assign a name/id to the client generally that name is unimportant as long as it is unique. However the Mosquitto Server allows you to impose client id prefix restrictions on the client name, and this provides some basic client security. You could, for example, choose a prefix of C1- for your client ids and so a client with client id of C1-python1 would be allowed but a client with id of python2 would not be allowed.
You will find this setting in the security settings section of the mosquitto.conf file.

```
clientid_prefixes C1-
```

Username and Password

An MQTT server can require a valid username and password from a client before a connection is permitted. The username/password combination is transmitted in clear text and is not secure without some form of transport encryption. However it does provide an easy way of restricting access to a server and is probably the most common form of identification used.
The username used for authentication can also used in restricting access to topics.
On the Mosquitto server you need to configure two settings for this to work. Again you will find these settings in the security section of the mosquitto.conf file.
They are allow_anonymous and password_file. To require username/password then allow_anonymous should be false and password_file should contain a valid passwords file.
Example settings

```
allow_anonymous false
password_file /home/pi/mosquitto/passwords.txt
```

To create the passwords you will need to use the mosquiito_passwd utility that comes with the Mosquitto server.

x509 Client Certificates

This is the most secure method of client authentication but also the most difficult to implement because you will need to deploy and manage certificates on many clients.
This form of authentication is really only suited to a small number of clients that need a high level of security.

Restricting Access to topics
You can control which clients are able to subscribe and publish to topics.
The main control mechanism is the username. (note: password not required), but you can also use the client id. Unless you are running an open server then this type of restriction will be common.

Securing Data
To protect the contents of your MQTT messages you can use:

- TLS or SSL Security
- Payload encryption

TLS Security
TLS security or as it is more commonly known SSL security is the technology that is used on the web. This security is part of the TCP/IP protocol and not MQTT. TLS security will provide an encrypted pipe down which your MQTT messages can flow. This will protect all parts of the MQTT message, and not just the message payload. The problem with this is that it requires client support, and it is unlikely to available on simple clients.

Payload Encryption
This is done at the application level and not by the server. This means that you can have encrypted data without having to configure the server. It also means that data is encrypted end to end and not just between the server and the client. MQTT is after all a messaging protocol. However this type of encryption doesn't protect passwords (if used) on the connection itself. Because it doesn't involve any server configuration or support this is likely to be a very popular method of protecting data.

Paho Python MQTT Client Changes for MQTTv5 Support

The Paho Python client version 1.5.1 included support for MQTTv5. Because of the new capabilities of MQTTv5 there have been changes to many of the common functions like connect,subscribe and publish etc.

MQTT Protocol Version

If you want to use the new MQTTv5 capabilities then is important to specify the protocol version you want to use. Supported versions are:

- MQTTv31 = 3
- MQTTv311 = 4
- MQTTv5 = 5

Therefore when creating the client for MQTTv5 use:

```
client = mqtt.Client("mqtt5_client",protocol=MQTTv5)
```

Connection Function

MQTT v3.1.1

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

MQTT v5

```
connect(self, host, port=1883, keepalive=60, bind_address="", bind_port=0,
clean_start=MQTT_CLEAN_START_FIRST_ONLY, properties=None)
```

The most important addition is the properties object. The `properties object` is present in other function calls, and so it is important to understand how to use it and we cover this next.

Properties Object

The properties object varies according to what function you will use it in. To use it in the connection function you first need to import the properties class and then create a properties object using the following syntax:

```
properties=Properties(PacketType)
```

The following are the available packet types (taken from the code:

```
CONNECT, CONNACK, PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE,
SUBACK, UNSUBSCRIBE, UNSUBACK, PINGREQ, PINGRESP, DISCONNECT, AUTH
```

In addition you will need to `set property attributes`. In the example code below we set the `maximum packet size` the client can receive.

```
from paho.mqtt.properties import Properties
from paho.mqtt.packettypes import PacketTypes
properties=Properties(PacketTypes.CONNECT)
properties.MaximumPacketSize=20
client.connect(host,port,properties=properties)
```

To make your code work with both MQTTv5 and MQTTv3.1.1 you need to set the properties object to None when using MQTTv3.1.1. So Instead of

```
properties=Properties(PacketTypes.CONNECT)
```

you would use

```
properties=None
```

Note: MQTTv5 Properties by Message Type has list of property fields that are available in the connection message and other message types.

Connect Acknowledge Callback Function
The server can return lots of information in the properties field of the `CONNACK message` and so the callback function must include it. Below shows `CONNACK` callbacks for v3.1.1 and v5 as defined in the latest client code. Old signature for MQTT v3.1 and v3.1.1 is:

```
on_connect(client, userdata, flags, rc)
```

signature for MQTT v5.0 client:

```
on_connect(client, userdata, flags, reasonCode, properties=None)
```

Notice the extra properties field. However existing code written for v3.1.1 used the following callback signature:

```
connect(client, userdata, flags, rc)
```

This will still work with MQTTv3.1.1 but you will get an error with MQTTv5. To make the callback work with v3.1.1 and v5 connections use:

```
on_connect(client, userdata, flags, rc, properties=None)
```

The following functions have also changed and you need to use the same logic as described above.

Subscribe and Subscribe Callback
MQTT v3.1.1 and v3.1

```
subscribe(topic, qos=0)
```

and for MQTT v5.0:

```
subscribe(self, topic, qos=0, options=None, properties=None):
```

Subscribe Callback
Old signature for MQTT v3.1.1 and v3.1 is:

```
on_ subscribe(client, userdata, mid, granted_qos)
```

and for the MQTT v5.0 client:

```
on_subscribe(client, userdata, mid, granted_qos, properties=None)
```

UnSubscribe Callback
Old signature for MQTT v3.1.1 and v3.1 is:

```
unsubscribe_callback(client, userdata, mid)
```

and for the MQTT v5.0 client:

```
unsubscribe_callback(client, userdata, mid)
```

Publish Message
Old signature for MQTT v3.1.1 and v3.1 is:

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

and for the MQTT v5.0 client:

```
publish(self, topic, payload=None, qos=0, retain=False, properties=None
```