Workshop - 1

Introduction to MQTT

Overview

Message Queueing Telemetry Transport(MQTT) is a lightweight, publish-subscribe modelled protocol that runs over TCP/IP stack, widely used in IoT networking.

In this lab, you will download, install and configure an MQTT broker on your computer and familiarise with its working using Command Line Interface (CLI) and a given Python program.

Objectives

After completing this lab, you will be able to:

- Install an MQTT broker on your local computer
- Interact with the broker using CLI
- Configure the listener
- Communicate with the local MQTT broker using a predefined Python program

---

Installing MQTT Broker

You might be familiar with Hyper Text Transfer Protocol (HTTP). A Client usually makes a request and the server responds. This mode of communication is usually considered as Request/Response model or Client/Server model.

In MQTT, a publisher publishes a message to the broker and any subscriber connected to the same broker having subscribed to the said topic will receive that message.

There are a number of implementations of MQTT broker that can be classified based on open source availability, additional functionality and community support. Some of the popular ones are

- HiveMQ
- Mosquitto
- RabbitMQ
- MQTT Route
- more...

In this module, we will use the Mosquitto broker.

Task 1 Download and Start the broker that suits your operating system from the following

Windows

- Download the 64-bit | 32-bit installer
- Make sure you select the option to install the broker as a Service
- You can Start, Stop or Restart the mosquitto broker service from the Task Manager *(Ctrl+Shift+Esc)*, then select the Services tab and find the mosquitto service

> You may not be able to execute mosquitto client instructions from the Command Prompt unless you navigate to the mosquitto installation directory *(C:\Program Files\mosquitto)*, so it is recommended that you add this path to your System Environment Variable. Guide

> Once the mosquitto directory is added as the Environment system variable, you can also Start, Stop and check the status of the broker using the following instruction `sc [start/stop/query] mosquitto` choose one of the three and remove the square brackets on either side

Linux

- **Ubuntu**
  Execute the follow on your Terminal

```
#Update the packages
sudo apt update

#Install the mosquitto broker as well as the clients
sudo apt install mosquitto mosquitto-clients

#Start the mosquitto service
sudo systemctl start mosquitto.service

#To stop the mosquitto service you can uncomment the following
instruction and use it
#sudo systemctl stop mosquitto.service
#To restart the mosquitto service you can uncomment the following
instruction and use it
#sudo systemctl restart mosquitto.service
```

- **Amazon Linux 2**
  Execute the following on your Terminal

```
sudo amazon-linux-extras install epel-release
sudo yum install mosquitto
sudo systemctl start mosquitto.service
```

Mac

Execute the following on your Terminal

```
brew install mosquitto
# if you do not have homebrew installed already, execute the
following to install it
```

```
# /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)'
# For more info on Homebrew, visit https://brew.sh/
# Execute the following instruction in the Terminal to start
Mosquitto service
/usr/local/sbin/mosquitto
```

Other Operating Systems

Check here

Interacting with MQTT Broker using CLI

Although the mosquitto broker might be installed in your computer, you still need to make sure that the service is running and listening in the configured port `Default: 1883`

Task 2: Check if the mosquitto service is listening. Open a Terminal window (say, Terminal 1) and execute the following instruction

Windows

```
netstat -an | findstr 1883
```

and you should see the response as follows

```
  Active Connections

    Proto  Local Address          Foreign Address
  State
    ...
    TCP    127.0.0.1:1883         DESKTOP-UTES7QV:0
  LISTENING
    ...
```

> The preceding `127.0.0.1` before `1883` means that the mosquitto service is listening to connections made to port 1883 only from the same computer. This is the default in Windows

Linux

```
netstat -ltn
```

and you should see the response as follows

```
  Active Internet connections (only servers)
  Proto Recv-Q Send-Q Local Address           Foreign
  Address         State
  ...
  tcp        0      0 0.0.0.0:1883              0.0.0.0:*
  LISTEN
  tcp6       0      0 :::1883                   :::*
  LISTEN
  ...
```

> The preceding `0.0.0.0` before `1883` means that the mosquitto service is listening to connections made to port 1883 from any device in any of the computer's networks. This is the default in Linux

Mac

`lsof -nP -iTCP:1883 | grep LISTEN`
and the response should be

```
mosquitto ... user  4u  IPv4 ...  0t0  TCP 127.0.0.1:1883
(LISTEN)

mosquitto ... user  5u  IPv6 ...  0t0  TCP [::1]:1883
(LISTEN)
```

> The preceding `127.0.0.1` before `1883` means that the mosquitto service is listening to connections made to port 1883 only from the same computer. This is the default in MacOS

Publish

Messages can be published to a topic named **topic_name** with the flag `-t` and the message **message** with the flag `-m`

`mosquitto_pub -t "topic_name" -m "message"`
By default, the *mosquitto_pub* command publishes to the `localhost` and the port `1883`. If we had to enforce this in the command, it would look as follows:

`mosquitto_pub -h "localhost" -p 1883 -t "topic_name" -m "message"`
Publishing to other hosts and ports (say, port `5000`)

```
#The following commands are only to demonstrate the usage of an
IP address or a domain name as the host
mosquitto_pub -h "1.2.3.4" -p 5000 -t "topic_name" -m "message"
mosquitto_pub -h "mqtt.gl.in" -p 5000 -t "topic_name" -m
"message"
```
Task 3: Publish the message `Testing publish` to the local broker with the topic `iot-learning/lab1`

`mosquitto_pub -t "iot-learning/lab1" -m "Testing publish"`
You may not be able to see this message elsewhere since there isn't any client subscribed to this topic yet.

Let's understand and create a subscriber next.

Subscribe

The simplest instruction to subscribe to a topic named **topic_name** is

`mosquitto_sub -t "topic_name"`
By default, the *mosquitto_sub* command publishes to the `localhost` and the port `1883`. If we had to enforce this in the command, it would look as follows:

```
mosquitto_sub -h "localhost" -p 1883 -t "topic_name"
```

Executing an instruction with the *mosquitto_sub* command establishes a connection with the broker and keeps looking for any messages that are published to the given topic. Hence it is a blocking command and when it is running, the Terminal cannot be used for executing other instructions.

Task 4: Open another Terminal (say, Terminal 2) and execute an instruction to subscribe to the topic `iot-learning/lab1`

```
mosquitto_sub -t "iot-learning/lab1"
```

Go back to the Terminal 1 and execute the publish instruction and you should now see the message printed on the Terminal 2

MQTT Topics - Deep Dive

In MQTT, topics are strings (UTF-8) which can have several levels split with a separator `/`

For example:

```
level0/level1/level2/level3
devices/building1/floor3/room8/temperature
alpha/beta/gamma
```

Also a topic must have **atleast 1 character** and it is **case-sensitive**. This means that the topic *alpha/beta* is different from the topic *Alpha/Beta*

MQTT topics also support the usage of **Wildcards** `+`, `#` while subscribing to a topic. There are 2 types of wildcards

A **single-level wildcard** `+` - used between the levels For example:

```
# To receive messages containing the temperature values of
room8 in all the floors of building 1
devices/building1/+/room8/temperature
```

A **multi-level wildcard** `#` - used at the leaf level For example:

```
#To receive messages from all the devices in building 1
devices/building1/#
```

Subscribing to the wildcard `#` topic directly may result in a chaos for large systems where hundreds or more messages are published every second. Hence it is not recommended to subscribe to the wildcard, unless it is a test environment where the number of messages published per second is quite low.

Also, the topics starting with `$` are reserved for the internal statistics of the broker and they are not subscribed to when a subscriptions is made with the multi-level wildcard `#`. Commonly, `$SYS/` is used for all the following information, but broker implementations varies.

- $SYS/broker/clients/connected
- $SYS/broker/clients/disconnected
- $SYS/broker/clients/total
- $SYS/broker/messages/sent
- $SYS/broker/uptime

You can find more on the $SYS-topics at MQTT GitHub wiki.

> Since the topics follow UTF-8 string formatting where certain characters have existing definitions, it is recommended not to use any kind of white space or a forward slash in the topic names

Task 5: Subscribe to a topic `devices/+/single` in the Terminal 1

```
mosquitto_sub -t "devices/+/single"
```

In the Terminal 2, publish to different topics and look at the values showing up in Terminal 1

```
mosquitto_pub -t "devices/wild1/single" -m "Message published to devices/wild1/single"
mosquitto_pub -t "devices/wild2/single" -m "Message published to devices/wild2/single"
```

Task 6: Subscribe to a topic `devices/#` in the Terminal 1

```
mosquitto_sub -t "devices/#"
```

In the Terminal 2, publish to different topics and look at the values showing up in Terminal 1

```
mosquitto_pub -t "devices/value" -m "Message published to devices/value"
mosquitto_pub -t "devices/level1/level2" -m "Message published to devices/level1/level2"
```

Attributes

Publishing the contents of a file as the message To publish the contents of a file as the message, use the flag `-f` instead of `-m` as follows:

```
mosquitto_pub -h "localhost" -p 1883 -t "topic_name" -f file_path
```

Task 7: Create a file named `message.json`, insert the following contents to it and save the file

```
{
        "device_name":"IOTDev248",
        "device_type": "counter"
        "timestamp": "2021-01-14 13:13:13"
        "value": 45
}
```

Open a terminal and navigate to the directory where you saved the above file and publish the contents of this file to the topic `filepublish`

```
mosquitto_pub -t "filepublish" -f message.json
```

Debugging the messages One of the simplest ways of debugging the communication between the client and broker is by adding the flag `-d` while you publish or subscribe to the broker. This allows you to look at various packets exchanged between the client and the broker.

Task 8: Publishing with debug enabled Command

```
mosquitto_pub -d -t "topic_name" -m "{'keys':'values'}"
```

Response

```
1609685374: New connection from ::1:59928 on port 1883.
Client (null) sending CONNECT
1609685374: New client connected from ::1:59928 as auto-A49A14D9-
6030-B438-2599-42C62077CF16 (p2, c1, k60).
Client (null) received CONNACK (0)
Client (null) sending PUBLISH (d0, q0, r0, m1, 'topic_name', ...
(17 bytes))
Client (null) sending DISCONNECT
1609685374: Client auto-A49A14D9-6030-B438-2599-42C62077CF16
disconnected.
```

Task 9: Subscribing with debug enabled

Command

```
mosquitto_sub -d -t "topic_name"
```

Response

```
Client (null) sending CONNECT
Client (null) received CONNACK (0)
Client (null) sending SUBSCRIBE (Mid: 1, Topic: topic_name, QoS:
0, Options: 0x00)
Client (null) received SUBACK
Subscribed (mid: 1): 0

#And every 60 seconds, you can find the following messages
printed
Client (null) sending PINGREQ
Client (null) received PINGRESP
```

The reason why you see the term `null` in the above responses is because the client wasn't assigned with an ID

Configuring Listener

Configurations for the mosquitto broker is defined in the `mosquitto.conf`, usually located in the following path

Windows - *C:\Program Files\mosquitto* Linux - */etc/mosquitto/* Mac - */usr/local/etc/mosquitto/*

Initially, the `mosquitto.conf` file contains all the configurations in commented state. Create a backup of this and save it with another name, say `mosquitto.conf.bak` for future reference.

There are a number of things that can be configured in this configuration file such as:

- the listener - bind port and bind interface
- message size limit
- max queued message
- maximum number of connections
- location of password file
- SSL/TLS support
- ... (refer the original mosquitto.conf file)

In this section, we will only configure the listener. Following is the format of configuring a listener

```
conf
listener [bind_port] [bind_interface]
```

Port binding

Task 11 Edit the `mosquitto.conf` file, empty it out and add the following line only

```
listener 5000
allow_anonymous true
```

and now restart the mosquitto broker

Windows Go to `Task Manager → Services → mosquitto`, right click and select `Restart`

Linux

```
sudo systemctl restart mosquitto.service
```

Mac

```
#Stop the broker
killall mosquitto
#Start the broker with specified configuration file in the
background
/usr/local/sbin/mosquitto -c
/usr/local/etc/mosquitto/mosquitto.conf -d
```

Now the broker will have started in the port 5000. Any further publish/subscribe you make to the broker explicitly needs to mention the port with the `-p 5000` flag

Interface binding

Task 12 Publish and subscribe messages with the newly configured port

```
#Publish example
mosquitto_pub -p 5000 -t "topic_name" -m "message"
#Subscribe example
mosquitto_sub -p 5000 -t "topic_name"
```

This was an example of how to configure a different port where the mosquitto broker listens. Similarly, we can restrict the network in which the broker will be available by binding it to one or more of your computer's network interfaces

To restrict the broker to function only for the localhost, the listener can be configured as

```
listener 1883 localhost
#or listener 1883 127.0.0.1
allow_anonymous true
```

If your computer is connected to a LAN and let's say your computer has the IP address `192.168.1.105`, then you could use this as the *bind_interface* to restrict the usage of your broker only to your LAN

```
listener 1883 192.168.1.105
allow_anonymous true
```

To open up your broker for all the network interfaces, you could use the bind_address as `0.0.0.0`

```
listener 1883 0.0.0.0
# This is the default configuration even if the bind_port
and bind_interface is not mentioned explicitly
allow_anonymous true
```

Task 13 Bind the broker connection to accept requests only from localhost, restart the broker and try publishing messages to the broker from another device in your computer's LAN

> You can use an MQTT application in your smartphone to publish messages to the broker. Make sure your smartphone and the computer are on the same LAN

Since the broker is configured to accept connections only from the localhost, other devices in your LAN should not be able to connect, publish or subscribe to the broker.

Now edit the configuration file and bind the computer's LAN IP address, restart the broker and verify if other devices on the LAN are able to connect, publish and subscribe to the broker.

Device Authentication

An MQTT can be configured to have any of the 3 types of authentications

Anonymous

This method allows any client/device with the valid hostname and the port number to get connected to the broker.

To enable this authentication method, the following line can be added to `mosquitto.conf` file

```
allow_anonymous true
```

> Defaults to false, unless no listeners there are no listeners defined in the configuration file, in which case it is set to true, but connections are only allowed from the the local machine

Task 1: Since there weren't any listeners configured in the earlier steps, we were able to establish connection to the broker. Let's enforce the broker not to allow anonymous connections and find out the result

Edit the `mosquitto.conf` to have the following line

```
listener 1883 0.0.0.0
allow_anonymous false
```

Save and restart the broker

Now when we subscribe to a topic from the Terminal

```
mosquitto_sub -t "test"
```

the response we receive is

```
Connection error: Connection Refused: not authorised.
```

Modify the `mosquitto.conf` to allow anonymous connections, save and restart the broker. Then verify if the clients are able to establish connection to the broker

Username and Password

Allowing anonymous connections is obviously not a good choice for IoT systems in production. Authenticating the clients with a username and password is one of the authentication methods supported by mosquitto. The usernames and passwords (in encrypted format) is stored in a password file and the same is referenced in the `mosquitto.conf` file

To enable this authentication method with a password file, the `mosquitto.conf` needs to have the following lines in it

```
listener 1883 0.0.0.0
allow_anonymous false
password_file <passwordfilepath>
```

Say, if you choose to have the password file name as `pwfile`, to be stored in the same directory as that of default `mosquitto.conf` location, the **<passwordfilepath>** will be

Windows: *C:\Program Files\mosquitto\pwfile*

Linux: */etc/mosquitto/pwfile*

Mac: */usr/local/etc/mosquitto/pwfile*

Task 2: Initially, the password file you configured might not exist, so to create a password file and add a user to it, execute the following instruction

```
#Add User by creating a new passwordfile
mosquitto_passwd -c <passwordfilepath> <username>
```

> Replace <username> with a string of your choice, and enter the desired password for this username when prompted

To add more users in the same password file, you can use the following instruction

```
#Add a new user in the existing users list (password file)
mosquitto_passwd <passwordfilepath> <username>
#Enter the password when prompted
```

When you open the password file with a text editor, you will see something similar to the following

```
roger:$6$clQ4Ocu312S0qWgl$Cv2wUxgEN73c6C6jlBkswqR4AkHsvDLWvtEXZZ8
sub_client:$6$U+qg0/32F0g2Fh+n$fBPSkq/rfNyEQ/TkEjRgwGTTVBpvNhKSyC
pub_client:$6$vxQ89y+7WrsnL2yn$fSPMmEZn9TSrC8s/jaPmxJ9NijWpkP2e7b
```

Task 3: Publish and subscribe with username and password

Once the `mosquitto.conf` file is modified as mentioned in Task 1 and restarted the broker to incorporate new configurations, then users are added to the password file as discussed in Task 2, connections to the broker can be established using the `-u` flag to add the username attribute and the `-P` flag to add the password attribute

Execute the following in Terminal 1

```
mosquitto_sub -u "<username>" -P "<password>" -t "IoTlearning"
```

Execute the following in Terminal 2

```
mosquitto_pub -u "<username>" -P "<password>" -t "IoTlearning" -m
"publish message"
```

> Make sure to replace the `<username>` and `<password>` with the ones you used in Task 2

Other features of *mosquitto_passwd* command To delete/remove a user from the password file, you can use the following instruction

```
#Delete/Remove user
mosquitto_passwd -D <passwordfilepath> <username>
```

To add users to the password file by passing both the username and password as arguments, you can use the following instruction

```
mosquitto_passwd -b <passwordfilepath> <username> <password>
```

Say you have a list of usernames and passwords in plain text file for several users that are supposed to be added to the password file. In such case you can configure that file as a password file and update the passwords from text to hashed format

```
#Update a password file from text to hashed
mosquitto_passwd -U <passwordfilepath>
```

Setting Username and Password in paho.mqtt.client

To set the username and password for the `paho.mqtt.client` in python program, add the following instruction after instantiating the broker object

```
client.username_pw_set(username="xxx", password="yyy")
```

Task 4: Configure your MQTT broker to enable password based authentication, and modify the `publish.py` and `subscribe.py` to communicate with the broker using their designated authentication

Certificates

Just like how we have HTTPS, secure connection of HTTP, MQTTS is the secure connection of MQTT.

Defualt Ports	Insecure	Secured (SSL/TLS)
HTTP	80	443
MQTT	1883	8883

In MQTTS, an x509 certificate is used by the client and the broker to exchange the information in an encrypted format.

Obtaining and working with the SSL/TLS certificates works best in production environments where the broker host has a domain name associated with it.

You may find more information on configuring this here

Workshop - 2

Communicating with the broker using Python program

There are a number of Client libraries available to communicate with an MQTT broker such as:

- Eclise Paho Python
- gmqtt
- mqtttools
- ...

In this task, we will use the Eclise Paho Python (paho.mqtt.python) library.

Step 1:

Install the Eclipse Paho Python library in your computer by executeing the following instruction in your `Terminal` / `Command Prompt`

```
python -m pip install paho-mqtt
```

Code Walk-through

In the following section, let's walthrough the various steps which comes together as a complete python program as a simulation of IoT devices publishing data to the MQTT broker periodically.

Step 2: Create a python file in your computer

> Example: `publish.py`

The complete source code of this python program and the config file to define the broker settings and the device configuration is available in the companion notebook.

The Eclipse Paho module allows us to import the MQTT in any of the 3 modes

- Publisher
- Subscriber
- Client

A client sub-module of paho.mqtt allows us to publish and subscribe in addition to other functionalities.

Thus we will import the client submodule as mqtt and create an instance of the same

In [ ]:
```
import paho.mqtt.client as mqtt

#Instantiating an object with mqtt
client = mqtt.Client()
```

Before we connect the client to the broker, we will define the callback functions and link it to the client actions

The client actions could be

- conntecting to the broker
- publishing to the broker
- subscribing to the broker
- receiving a message from the broker on a subscribed topic

- disconecting from the broker, ...

In this program, we will work with 3 client actions

- on_connect
- on_message
- on_disconnect

and the 3 callback functions we will link to these actions will be with the same name, although it can vary

In [ ]:
```
# Callback function - executed when the program successfully connects to
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("test")

#Callback function - executed when the program gracefully disconnects fro
def on_disconnect(client, userdata, rc):
    print("Disconnected with result code "+str(rc))

#Callback function - executed whenever a message is published to the topi
#this program is subscribed to
def on_message(client, userdata, msg):
    print(msg.topic,str(msg.payload), "retain", msg.retain, "qos", msg.qo


#Setting callback functions for various client actions
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
```

The configuration of the broker and IoT devices are stored in a `config.json` file from which the settings are read and then processed.

Structure of this configuration file is as follows:

```
{
    'broker_host': <hostname/ip_address>,
    'broker_port': <port number>
    'devices':[
        {
            "type": <string>,
            "publish_frequency": <int, in_seconds>,
            "publish_topic": <string>,
            "std_val": <int>,
            "device_count": <int>
        },
        ...
    ]
}
```

Since the configuration is stored in json format, we import the contents of the file as json, then create a list object `device_config` where individual device settings are stored

In [ ]:
```
import json

# Reading the configuration file
f=open("config.json")
config = json.loads(f.read())
f.close()

# Initialising devices from the config.json file and assigning device_ids
device_config = []
for devices in config['devices']:
    for n in range(devices['device_count']):
        dev = {}
        dev['device_id'] = devices['type']+"_"+str(n)
        dev['device_type'] = devices['type']
        dev['publish_frequency'] = devices['publish_frequency']
        dev['std_val'] = devices['std_val']
        dev['publish_topic'] = devices['publish_topic']
        device_config.append(dev)
```

We then connect the client to the broker from the broker configuration read from the `config.json` file and initialize the client loop which keeps checking for any client actions and executes the appropriate callback function

In [ ]:
```
#Connecting to broker
client.connect(host=config["broker_host"], port=config["broker_port"], ke

'''
Start the MQTT client non-blocking loop to listen the broker for messages
in subscribed topics and other operations for which the callback function
are defined
'''
client.loop_start()
```

Assuming the publish frequency configured in `config.json` has a `1s` (one second) resolution, we initiate a while loop that executes once every second

In [ ]:
```
import time

clock = 0
while True:
    time.sleep(1)
    clock = clock+1
    #Perform publish action
```

Just like we started the client loop, it is also important to stop the client loop at the termination of the program. Else, the loop keeps running in the background until the Terminal is closed.

To ensure that the client gets disconnected from the broker and the loop is stopped, we add a Keyboard Interrupt exception which gets triggered when `Ctrl+C` is pressed and ensure the client disconnection and the stop of loop

In the try block, we now add the following code that iterates through the `device_config` list, building a message to be publish and then publishes to the

topic defined in the config

In [ ]:
```
clock = 0
while True:
    try:

        time.sleep(1)
        clock = clock+1
        #Perform publish action

    #Disconnect the client from MQTT broker and stop the loop gracefully
    except KeyboardInterrupt:
        client.disconnect()
        client.loop_stop()
        break
```

In [ ]:
```
import datetime


for devices in device_config:
    if clock%devices['publish_frequency']==0:
        print("Published to devices/"+devices["device_type"])

        #Initialize a dictionary to be sent as publish message
        message = {}

        #Generate timestamp in YYYY-MM-DD HH:MM:SS format
        message["timestamp"] = datetime.datetime.now().strftime("%Y-%m-%d
        message["device_id"] = devices["device_id"]
        message["device_type"] = devices["device_type"]
        message["value"] = devices["std_val"]

        #Publish the message
        client.publish(devices["publish_topic"], json.dumps(message))
```

Instead of publishing the configured standard value all the time, we will obtain a normally distributed value with mean as the standard value and standard deviation as 2, then rounding off to 2 decimal places

In [ ]:
```
import numpy as np


#Replacing message["value"] = devices["std_val"] with
message["value"] = round(np.random.normal(devices["std_val"],2),2)
```

This ensures the periodic publishing of the messages to the broker from this client program. Make sure you have a `config.json` file in the same directory as that of the python program.

Here is a sample `config.json` which will simulate

- **2 temperature sensor devices** publishing its values to the **devices/temp** topic every **5 seconds**
- **2 humidity sensor devices** publishing its values to the **devices/hum** topic every **10 seconds** and

- **1 co2 sensor device** publishing its values to the **devices/co2** every **5 seconds**

to the **local broker** hosted on port **1883**

Step 3: Create another file `config.json` in the same directory as that of your python program and copy-paste then save the following in it

```
{
    "broker_host": "localhost",
    "broker_port": 1883,
    "devices": [
        {
            "type": "temperature",
            "publish_frequency": 5,
            "publish_topic": "devices/temp",
            "std_val": 25,
            "device_count": 2
        },
        {
            "type": "humidity",
            "publish_frequency": 10,
            "publish_topic": "devices/hum",
            "std_val": 40,
            "device_count": 2
        },
        {
            "type": "co2",
            "publish_frequency": 5,
            "publish_topic": "devices/co2",
            "std_val": 20,
            "device_count": 1
        }
    ]
}
```

Feel free to modify, add and/or delete the devices in the given configuration

Step 4: Start the python program

```
python publish.py
```

Step 5: Open another Terminal and subscribe to a topic with the following command

```
mosquitto_sub -t "devices/[device_name]"
```

> Replace the `[device_name]` in above instruction with one of the `name` fields in `config.json` For example:

```
mosquitto_sub -t "devices/temp"
```

You should now see the python program publishing messages with the frequency mentioned in `config.json`

Step 6: Try subscribing to other topics and fetch those values

Step 7: Use **#** as a wilcard and subscribe to all topics under `devices/`

Introduction to Receiver Simulator

Overview

In this lab, you will work with the receiver simulator that collects the IoT device data and stores it in a MongoDB database.

We will also review some use cases of MQTT protocol in IoT systems.

In addition, we will look at authentication of devices, authorization to publish/subscribe in topics.

Objectives

After completing this lab, you will be able to:

- Fetch messages from MQTT broker and store it in a MongoDB database
- Review the use cases of MQTT broker in IoT systems
- Authenticate clients / devices
- Authorize the clients / devices to specific topics

Duration

This lab requires approximately **45 minutes** to complete.

In [ ]:
```
import paho.mqtt.client as mqtt

#Instantiating an object with mqtt
client = mqtt.Client()
```

Database Setup

Considering the semi-structured nature of data published by the IoT devices, it is generally preferable to use a NoSQL database. In this module, we will use one of the widely used NoSQL database engine, **MongoDB** (Community Edition).

You may follow the steps given at https://docs.mongodb.com/manual/administration/install-community/ to install the database engine on your computer. Instructions to start / stop the database engine is also described in the same link.

To ensure MongoDB is listening for connections, you can execute the following and confirm the response

Windows

```
netstat -an | findstr 27017
```
and you should see the response as follows

```
Active Connections

  Proto  Local Address          Foreign Address
State
  ...
  TCP    127.0.0.1:27017         DESKTOP-UTES7QV:0
LISTENING
  ...
```

Linux

`netstat -ltn`
and you should see the response as follows

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign
Address         State
...
tcp        0      0 127.0.0.1:27017
127.0.0.1:*               LISTEN
...
```

Mac

`lsof -nP -iTCP:27017 | grep LISTEN`
and the response should be

```
mongod  ... username    9u  IPv4 ...      0t0  TCP
127.0.0.1:27017 (LISTEN)
```

> The preceding `127.0.0.1` before `27017` means that the mongod service is listening to connections made to port 27017 only from the same computer.

Walkthrough of Receiver Simulator

Lab 1 introduced you to a Publish Simulator that keeps publishing MQTT messages as configured.

This section walks you through the Receiver Simulator that acts as an IoT client / device, receiving all the messages from the publishing devices / Publish Simulator and thus storing it on your MongoDB database.

This is normally done by a central server (on-prem or cloud), either directly communicating with the devices, or via an Edge server.

In order to communicate with the MongoDB service from the python program, we will use the `pymongo` client library. You may install it by executing the following instruction on your Terminal

```
python -m pip install pymongo
```

Step 1: Create a python file in your computer

> Example: `subscribe.py`

We will import the client submodule as mqtt and create an instance of the same

In [ ]:
```
# Callback function - executed when the program successfully connects to
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("devices/#")

#Callback function - executed when the program gracefully disconnects fro
def on_disconnect(client, userdata, rc):
    print("Disconnected with result code "+str(rc))

#Callback function - executed whenever a message is published to the topi
#this program is subscribed to
def on_message(client, userdata, msg):
    item = {"topic":msg.topic, "payload":msg.payload}
    dbt.insert_one(item)
    print("Received a messsage on " + msg.topic + " and inserted it to th


#Setting callback functions for various client actions
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
```

Unlike the callback functions in Lab 1, you might realize a couple of changes here.

In the `on_connect` function, we are subscribing to a partial multilevel wildcard topic **devices/#**. This will ensure all the topics from the simulator program in Lab 1 are subscribed to, in this program.

Also, for every message that the Lab 1 simulator program publishes, the `on_message` function in this program will insert the topic and the message into the MongoDB database.

Similar to the Lab 1, we are fetching the broker and database details from a configuration file `config.json`

```
{
    "broker_host": "localhost",
    "broker_port": 1883,

    "db_host": "localhost",
    "db_port": 27017,
    "db_name": "iot-db",
    "db_collection":"iot-sensors-data"
}
```

Since the configuration is stored in json format, we import the contents of the file as json and further initialize the database connection

In [ ]:
```
import json
import pymongo
```

```
#Reading the configuration file
f=open("config.json")
config = json.loads(f.read())
f.close()

#Initializing connection to the database
dbclient = pymongo.MongoClient(config["db_host"], config["db_port"])
db = dbclient[config["db_name"]]
dbt = db[config["db_collection"]]
```

In this lab tutorial, all the messages coming in the "devices/#" topic are stores in a single MongoDB database collection.

However this may not be a preference in the production environment. A desired way of storing data would be to store the messages in a separate collection for a given topic or a category. There could be more structured collections derived from these dump/log collections for performing analytics.

We then connect the client to the broker from the broker configuration read from the `config.json` file and initialize the client loop which keeps checking for any client actions and executes the appropriate callback function

In [ ]:
```
#Connecting to broker
client.connect(host=config["broker_host"], port=config["broker_port"], ke

'''
Start the MQTT client non-blocking loop to listen the broker for messages
in subscribed topics and other operations for which the callback function
are defined
'''
client.loop_start()
```

The start of loop will take care of executing relevant callback functions to receive the message and insert it to the database. Hence the program is only expected to keep running until there is a Keyboard Interrupt, during which the client will be disconnected and the loop will be stopped.

This is achieved with the combination of an infinite while loop and an exception block to detect the Keyboard Interrupt

In [ ]:
```
while True:
    try:
        #Do nothing
        pass

    #Disconnect the client from MQTT broker and stop the loop gracefully
    # Keyboard interrupt (Ctrl+C)
    except KeyboardInterrupt:
        client.disconnect()
        client.loop_stop()
        break
```

Step 2: If you haven't done already, create the configuration file `config.json` in the same directory as that of your python program and copy-paste then save the following in it

```
{
    "broker_host": "localhost",
    "broker_port": 1883,

    "db_host": "localhost",
    "db_port": 27017,
    "db_name": "iot-db",
    "db_collection":"iot-sensors-data"
}
```

Step 3: Start the python program

```
python subscribe.py
```

Step 4: Open another Terminal and navigate to the directory where you have stored `publish.py` from the Lab 1 and start the publish simulator

```
python publish.py
```

You should now see the relevant debug messages printed on the appropriate terminal windows.

Step 5: Let both the `publish` and `subscribe` simulators run for a few minutes so the database can collect some data and then stop the program `Ctrl+C`

Optional: You can view the data inserted into the database by performing a query as follows

In [ ]:
```
import pymongo
import json

#Reading the configuration file
f=open("config.json")
config = json.loads(f.read())
f.close()

#Initializing connection to the database
dbclient = pymongo.MongoClient(config["db_host"], config["db_port"])
db = dbclient[config["db_name"]]
dbt = db[config["db_collection"]]

#Querying for the messages that were published to the `devices/temp` topi
entries = dbt.find({"topic":"devices/temp"})

#Print the entries
for entry in entries:
    print(entry)
```

This program should display the entries inserted into the `db-collection` as follows

```
{'_id': ObjectId('5ffe5db4715dd2b8e9e348ab'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:10:52",
"value": 25.11, "device_type": "temperature", "device_id":
"temperature_0"}'}
```

```
{'_id': ObjectId('5ffe5db5715dd2b8e9e348ac'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:10:52",
"value": 24.02, "device_type": "temperature", "device_id":
"temperature_1"}'}
{'_id': ObjectId('5ffe5db9715dd2b8e9e348ae'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:10:57",
"value": 23.1, "device_type": "temperature", "device_id":
"temperature_0"}'}
{'_id': ObjectId('5ffe5dba715dd2b8e9e348af'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:10:57",
"value": 20.68, "device_type": "temperature", "device_id":
"temperature_1"}'}
{'_id': ObjectId('5ffe5dbe715dd2b8e9e348b3'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:11:02",
"value": 26.67, "device_type": "temperature", "device_id":
"temperature_0"}'}
{'_id': ObjectId('5ffe5dbf715dd2b8e9e348b4'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:11:02",
"value": 25.63, "device_type": "temperature", "device_id":
"temperature_1"}'}
{'_id': ObjectId('5ffe5dc4715dd2b8e9e348b6'), 'topic':
'devices/temp', 'payload': b'{"timestamp": "2021-01-13 08:11:08",
"value": 26.55, "device_type": "temperature", "device_id":
"temperature_0"}'}
```


> Note: In the `subscribe.py` program, since we did not specify a unique id with which each object in the collection can be identified, MongoDB inserts it with the key `_id`

Optional: Modify `subscribe.py` to subscribe to the topics published by the co2 sensors and hence only those messages getting inserted into the database

---

Config file `config.json`

In [ ]:

```
{
    "broker_host": "localhost",
    "broker_port": 1883,
    "devices": [
        {
            "type": "temperature",
            "publish_frequency": 5,
            "publish_topic": "devices/temp",
            "std_val": 25,
            "device_count": 2
        },
        {
            "type": "humidity",
            "publish_frequency": 10,
            "publish_topic": "devices/hum",
            "std_val": 40,
            "device_count": 2
        },
        {
            "type": "co2",
            "publish_frequency": 5,
```

```
            "publish_topic": "devices/co2",
            "std_val": 20,
            "device_count": 1
        }
    ]
}
```

Source code `publish_simulator.py`

In [ ]:
```
import paho.mqtt.client as mqtt
import time
import json
import numpy as np
import datetime

# Callback function - executed when the program successfully connects to
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("test")

#Callback function - executed when the program gracefully disconnects fro
def on_disconnect(client, userdata, rc):
    print("Disconnected with result code "+str(rc))

#Callback function - executed whenever a message is published to the topi
#this program is subscribed to
def on_message(client, userdata, msg):
    print(msg.topic,str(msg.payload), "retain", msg.retain, "qos", msg.qo

#Defining an MQTT client object
client = mqtt.Client()

#Setting callback functions for various client operations
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect

#Reading the configuration file
f=open("config.json")
config = json.loads(f.read())
f.close()

# Initialising devices from the config.json file and assigning device_ids
device_config = []
for devices in config['devices']:
    for n in range(devices['device_count']):
        dev = {}
        dev['device_id'] = devices['type']+"_"+str(n)
        dev['device_type'] = devices['type']
        dev['publish_frequency'] = devices['publish_frequency']
        dev['std_val'] = devices['std_val']
        dev['publish_topic'] = devices['publish_topic']
        device_config.append(dev)


#Connecting to broker
client.connect(host=config["broker_host"], port=config["broker_port"], ke

'''
```

```
Start the MQTT client non-blocking loop to listen the broker for messages
in subscribed topics and other operations for which the callback function
are defined
'''
client.loop_start()

clock=0
while True:
    try:
        # Iterating through the items in device configuration dictionary,
        time.sleep(1)
        clock = clock+1
        for devices in device_config:
            if clock%devices['publish_frequency']==0:
                print("Published to devices/"+devices["device_type"])
                #Initialize a dictionary to be sent as publish message
                message = {}
                #Generate timestamp in YYYY-MM-DD HH:MM:SS format
                message["timestamp"] = datetime.datetime.now().strftime("
                message["device_id"] = devices["device_id"]
                message["device_type"] = devices["device_type"]
                #Generate a random value using normal distribution functi
                # configured standard value for the given device type
                message["value"] = round(np.random.normal(devices["std_va
                #Publish the message
                client.publish(devices["publish_topic"], json.dumps(messa
    #Disconnect the client from MQTT broker and stop the loop gracefully
    # Keyboard interrupt (Ctrl+C)
    except KeyboardInterrupt:
        client.disconnect()
        client.loop_stop()
        break
```

Workshop - 3

Introduction to Node Red

Node-RED is a powerful tool for building Internet of Things (IoT) applications with a focus on simplifying the ‘wiring together’ of code blocks to carry out tasks.

It uses a visual programming approach that allows developers to connect predefined code blocks, known as ‘nodes’, together to perform a task.

The connected nodes, usually a combination of input nodes, processing nodes and output nodes, when wired together, make up a ‘flows’.

Although Node-RED was originally designed to work with the Internet of Things, i.e. devices that interact and control the real world, as it has evolved, it has become useful for a range of applications.

Installation & Setup

Node red is an open source software available for various devices. You can visit the official node red website.

Visit the website and choose the appropriate device. Official link is mentioned below:

https://nodered.org/docs/getting-started/

Choose the appropriate OS version, you need to install the relevant dependency/pre-requisite softwares mentioned on the website.

Start the node-red installation process.

Once node-red is installed on your system you can open it by running the appropriate command.

An empty node-red window will look like this:

Basic example

In this section, we will implement some very basic nodes to understand the data flow in node-red. Further we will enhance the example by using one of the features of node-red `function` to create random set of data.

We will also see how we can tune the interval based on our requirement.

We will also introduce another feature of node-red named as `switch` to do the anomaly detection.

Lets begin with the first step of the example implementation:

- Open node red by giving the correct command from terminal

Now we will pick nodes available in left pane. You can simply perform choose nodes from the left pane and drop it on the chart.

To create our first flow we will use following nodes:

- inject - 1 instance
- debug - 2 instance
- function - 1 instance
- switch - 1 instance

Flow of this example is as following:

1. By using inject, which is renamed as `timestamp` , we can trigger the execution.
2. Then we will edit the `function` node to write simple javascript code to generate random temperature data.
3. Once payload is created we will perform two tasks from the output point of `function` : (i) data printing and (ii) performing anomaly detection and printing the detected data.

Once you have all the necessary nodes as mentioned in the above image, we will connect these nodes based on the given flow of the example.

We will also see some of the basic features of each of these nodes. Once these nodes are connected, updated image will look as mentioned below:

Let's edit each of these nodes to perform the execution:

1. Click on `timestamp` node and edit it out as mentioned in the following image.

While editing this node we have provided interval as 1 second. You can also keep it as `None`. If it is None then in order to execute the flow you need to trigger the execution by clicking on the button.

2. Click on function and provide name and write appropriate javascript code to generate the random data.
3. Click on switch and provide the condition to do anomaly detection.
4. Click on `msg.payload` and provide the appropriate name for each of them as mentioned below:
5. Once all the editing is done, the update nodes and their connection will look as the following mage.

All the nodes are appearing with blue circle on them, it shows that there are some unsaved changes on these nodes. To save the flow click on `Deploy` button available in the top right corner.

You should see a successful deployed message on your screen.

You can also toggle on the debug button to enable and disable the messages available in the debugging section in right pane as mentioned below:

In [ ]:
```
# Javascript code used in the function node.
# You can edit this code to add and delete more data points


var message = {}
let current = new Date();
let cDate = current.getFullYear() + '-' + (current.getMonth() + 1) + '-'
let cTime = current.getHours() + ":" + current.getMinutes() + ":" + curre
let dateTime = cDate + ' ' + cTime;
message["timestamp"] = dateTime;
message["deviceid"] = "sensor101";
message["device_type"] = "temperature";
message["value"] = Math.floor((Math.random() * 18) + 18);
return {payload : message};
```

MQTT Setup & data publishing

In this section, we will be utilising the MQTT services available under node-red. To implement this example we will be using the `publish.py` script we have provided in

Week-01 of IoT module.

Flow of this example is mentioned below:

1. Create MQTT nodes to subscribe and publish to a paricular topic. In this example we are using the `publish.py` to publish the data, hence our main focus will be to subscribe to the topic.
2. Once we have subscribed to the topic by providing the detail related to the MQTT broker and port number we will see the raw data generated by the script.
3. We will also use `function` to create a JSON object before printing the data or performing any other operation.
4. On the output payload of the function we can create different switches to detect the anomaly for temperature, co2 and humidity. Alternatively, you can write a consoldiated function to handle all these three switches.
5. The detected anomaly can be printed using debug nodes.
6. We can also save the data in the database, to do this task we will be using mongodb.
7. We can also view the data stored in mongodb.

Look at the following image which shows the complete flow of MQTT example. Please find the image below:

To implement this example you should create the following nodes and connect the nodes based on the above given explanation:

- mqtt in - 1 instance
- mqtt out - 1 instance
- function - 1 instance
- switch - 4 instance
- debug - 5 instance
- mongodb in - 1 instance
- mongodb out - 2 instance

Lets complete the MQTT setup on the MQTT nodes. On the left pane under network you can find MQTT services.

1. Choose MQTT In and double click on the node to complete the MQTT setup. Click on server and provide the necessary details as mentioned in the following image.

2. Once server is created, provide the topic name to receive message from the particular topic. Remeber this topic name should be same that we have provided in the config i.e. `devices/#` to subscibe to all three data points that are `temp`, `hum` and `co2`

3. In this example we will be running the MQTT broker locally. Make sure that you are providing the correct Port number along with other related details.

4. Once MQTT broker service is running on your system, you should run the python script `publish.py`. If the details related to MQTT is correct then you will be

able to view the data.

- Published data is in string format hence we wrote a simple function to convert the data entry in JSON object.

Anomaly detection

In this section we will do anomaly detection based on the data points we are receiving. In a single switch we are first identifying the type of device. Look at the followig image for the reference:

Once we have identified the device_type now for each device we need to emloy different switches to perfrm anomaly detection. Look at the following image for temperaure anomaly detection. In this we are looking for any temperature>27 and less than 23 as anomaly.

You can edit these nodes to decide your own anomaly range.

Similar anomaly detection/switches are in place for humidity and co2 as well.

Data storage

Once we have implemented the anomaly detection, deploy the flow and ensure that you are receiving the raw data as well as anomaly data.

To utilize the mongodb features, mongodb service installation is required on the node red. To complete this installation follow the below mentioned steps:

- Click on the inequality sign on the top right corner beside deploy as mentioned in the following image.
- Choose `Manage palette` in the new window choose the install tab and search for `node-red-node-mongodb`. Click on on install and complete the installation process. Once it is successfully installed mongo related services will be availble under storage in left pane.

Now, the next part is to store the raw data as well as anomaly data in the database. To enable this feature we have already installed the mongodb library in the node-red.

Under the storage section look for `mongodb` in and `mongodb out`.

We need to create the relevant database and table in the mongodb to store the data.

```
* Run the mongodb server on your system.
* Open mogodb compass and connect to the server.
* Create a new database.
* After creating the database, create collection to store
raw_data and anomaly_data separately.
```

Database and collection created in the previous step will be used in node-red.

1. Double click on mongodb node to edit and perform the setup.
2. Click on server and enter the details such as Databse name which is created using mongodb compass.
3. Now provide the collection name and choose insert as an operation. Click on only store msg.payload object. Assign a name to the node.
4. Perform step 1, 2, and 3 on another mongodb node to store raw data, Make sure that you are giving correct collection name. No need to provide the server details because we have created the server while creating the first setup.
5. Now we will create some additional nodes to view the stored data in the DB. Double click on inject button and provide the details as mentioned in the following image :
6. Doublce click on created `mongodb in` node. We already have the server from previous step. Provide the collection name to perform the find operation.
7. Create a debug button and double click on it to edit the name and other informaion.

Once the complete setup is done. The flow setup will look as mentioned below:

Click on deploy. Make sure that MQTT broker service is running on the same port and server which is provided in the above step along `publish.py` script.

Once the flow is deployed it will appear as the following image.

You can enable and disable each of the debug buttons to view the recevied data for different anomaly detection switches.

Data in mongodb will be stored in the following manner. Please find the relevant image below: