|  |  |
| --- | --- |
| **Goal** | This lab aims to introduce the Eclipse MQTT Broker and how to use it. |

**Review the MQTT Protocol**

MQTT protocol is a Machine to Machine (M2M) protocol widely used in IoT (Internet of things). The MQTT protocol is a message-based, extraordinarily lightweight protocol, so it is adopted in IoT. Almost all IoT platforms support MQTT to send and receive data from smart objects.

The MQTT IoT protocol was developed around 1999. The main goal of this protocol was to create a very efficient protocol from the bandwidth point of view. Moreover, it is a very power-saving protocol. For all these reasons, it is suitable for IoT.

This uses the publish-subscribe paradigm in contrast to HTTP based on the request/response paradigm. It uses binary messages to exchange information with low overhead. It is straightforward to implement, and it is open. All these aspects contribute to its extensive adoption in IoT. Another exciting part is that the MQTT protocol uses a **TCP stack** as a transmission substrate.

As said before, the MQTT protocol implements a publish-subscribe paradigm. This paradigm decouples a client that publishes a message (“**publisher**”) to other clients that receive the message (“**subscribers**”). Moreover, MQTT is an **asynchronous protocol**, which means it does not block the client while waiting for the message.

In contrast to the HTTP protocol, which is a mainly asynchronous protocol. Another attractive property of the MQTT protocol is that it does not require that the client (“subscriber”) and the publisher are connected at the same time.

The critical component in MQTT is the **MQTT broker**. The main task of the MQTT broker is dispatching messages to the MQTT clients (“subscribers”). In other words, the **MQTT broker receives messages from the publisher and dispatches these messages to the subscribers.**

While it dispatches messages, the MQTT broker uses the **topic** to filter the MQTT clients that will receive the news. The topic is a string, and it is possible to combine the topics creating topic levels. A topic is a virtual channel that connects a publisher to its subscribers. MQTT broker manages this topic. Through this virtual channel, the publisher is decoupled from the subscribers, and the MQTT clients (publishers or subscribers) do not have to know each other to exchange data. This makes this protocol highly scalable without a direct dependency on the message producer (“publisher”) and the message consumer (“subscriber”).

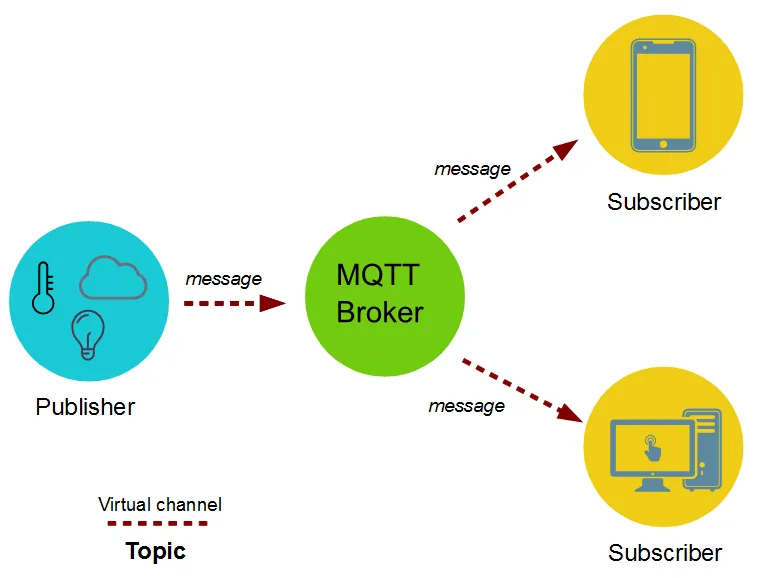
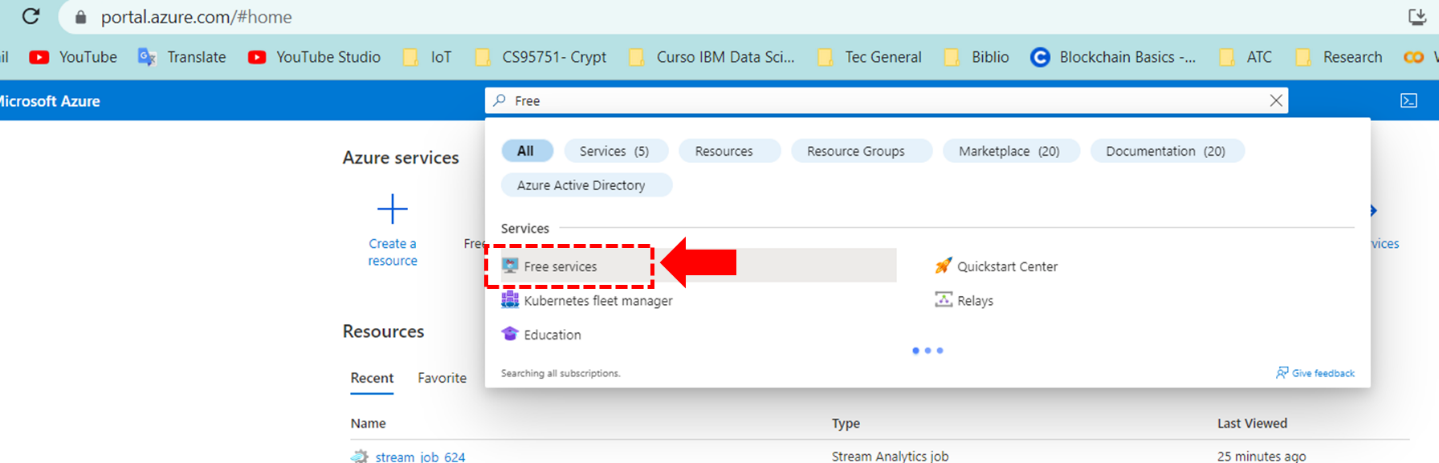


Figure - MQTT Architecture.

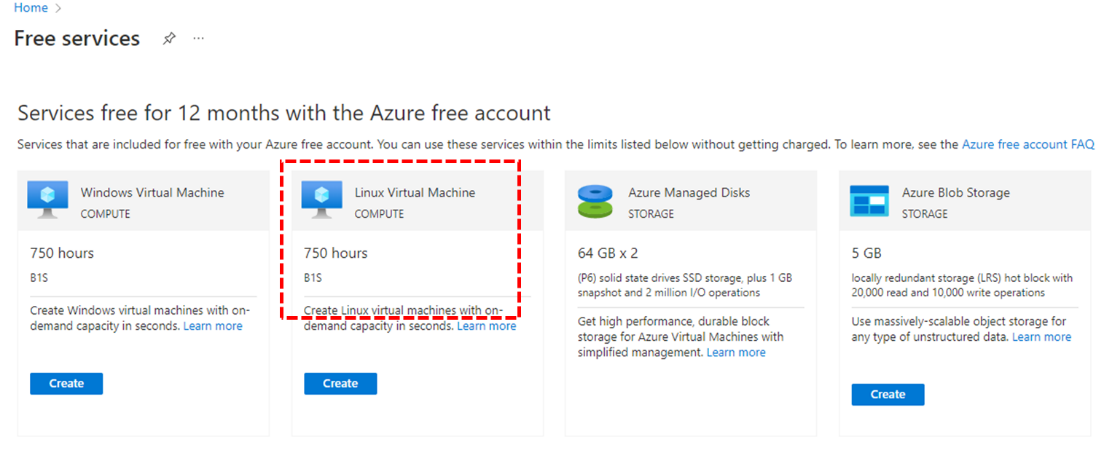
|  |  |
| --- | --- |
| WOW Deals - Sewell On The Go | **What is MQTT (MQ Telemetry Transport)?**  <https://www.youtube.com/watch?v=6XqGXw_5NhM> |

**Task 1 – Create an Ubuntu Machine in Azure**

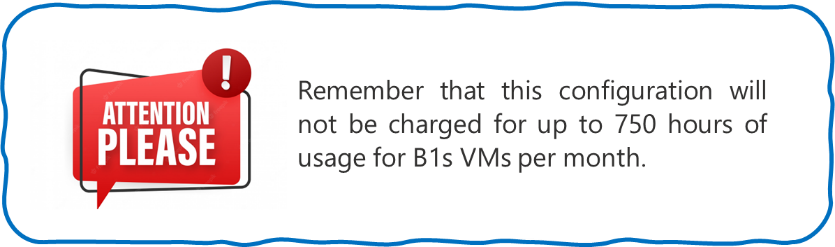
Log in to Azure and in the search bar, set the word **Free**. Azure will show some selections, click on **Free Services**.



Select a new Linux Virtual Machine and click on the **Create** button.

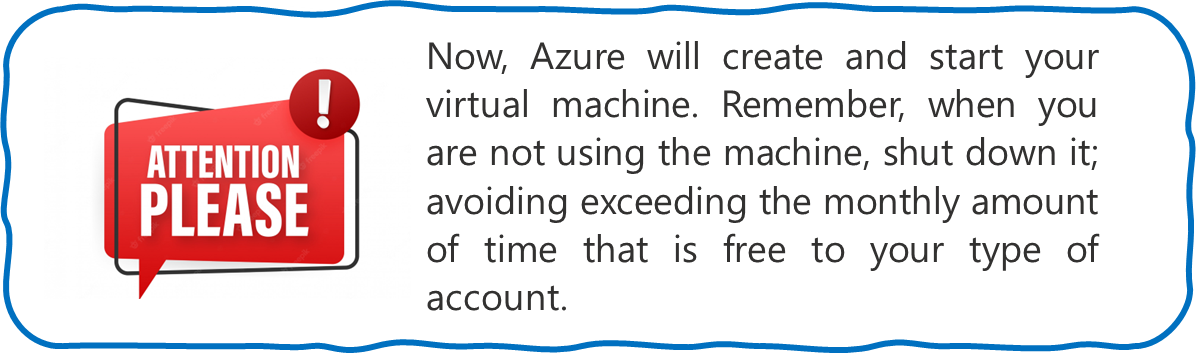


In the next screen, create a new resource group named **mqtt-lab**. Set virtual machine name as **iot-linux** and the image as **Ubuntu-Server 16.04 LTS - Gen 1**. In the authentication type, select **Password**, and the username set **iot** and the password **Iot123456789**. In the Public Inbound Ports, set **Allow selected ports**, and mark all options to the selected ports (**HTTP, HTTPS, SSH**). The other parameters keep the default value. Next will click on the **Review + Create** button.



Graphical user interface, text, application

Description automatically generated



To have access to the server, open a Windows Power Shell Terminal (or simple Linux / Mac terminal) and run the following commands: **ssh** [**iot@20.249.102.137**](mailto:iot@20.249.102.137). Validate the returned fingerprint. If you have never connected to this VM before you will be asked to verify the host’s fingerprint. It is tempting to simply accept the fingerprint presented; however, this exposes you to a possible person-in-the-middle attack. You should always validate the host’s fingerprint.

Text

Description automatically generated

Now, you need to ensure your Linux system is updated. To perform this task, you need to become a super-user and update the system, using these commands:

sudo su

apt-get update

apt-get upgrade

**Now, you are ready to use the Linux system.**

**What is Eclipse Mosquitto?**

Eclipse Mosquitto is an open-source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 5.0, 3.1.1, and 3.1. Mosquitto is lightweight and suitable for all devices, from low-power single-board computers to full servers.

The MQTT protocol provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for Internet of Things messaging, such as with low-power sensors or mobile devices such as phones, embedded computers, or microcontrollers.

**Task 2 - Installation and Basic Test of MQQT Mosquitto Broker**

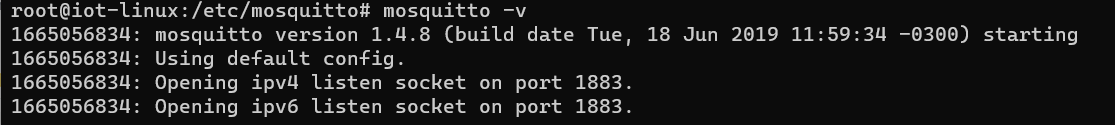
The installation of Mosquitto is quite simple, using the terminal, run the following commands:

sudo apt-get install mosquitto

sudo apt-get install mosquitto-clients

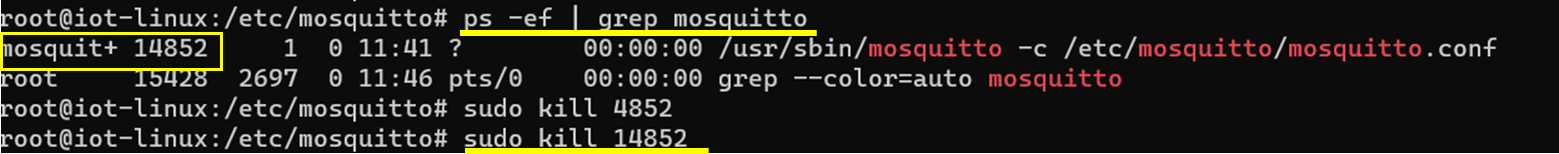
sudo apt clean

To test Mosquitto[[1]](#footnote-2), type this command: ***mosquitto -v***. This command initializes Mosquitto in a verbose mode.



Text

Description automatically generated with medium confidence



**Task 3 - Configuration of MQQT Mosquitto Broker**

You can configure the Mosquitto broker using a configuration file. The default configuration file is called **mosquitto.conf** and it is located at **/etc/mosquitto**. The Mosquitto broker supports client types MQTTv5 and MQTT v3.1.1; however, some configuration file settings will only affect MQTTv5 clients.

The default configuration uses a default listener, which listens on port **1883**, but you can create extra listeners.

First, we create a backup file of the original Mosquitto, using this command:

cd /etc/mosquitto

cp mosquitto.conf mosquitto.conf.orig

All settings have a default setting that is not set in the configuration file but is internal to Mosquitto. Settings in the configuration file override these default settings. However, default settings do not enable anonymous connection, as it only listens to the local host address. To change this configuration, add the ***allow\_anonymous*** line in the mosquitto.conf file.

Also, the default configuration file does not print any log in the terminal, to enable it add a line to log destination is printed in the **stdout**.

The Mosquitto project also provides a C library for implementing MQTT clients and the very popular **mosquitto\_pub** and **mosquitto\_sub** command line MQTT clients. To test the implementation, we will open 3 terminal tabs in the Power Shell, remote log in Azure server (using ssh), and become superuser (using sudo su).

Next, you need to run five different steps.

Step 1 – Change the config file enabling the anonymous connections and making the broker listen to the desired address.

allow\_anonymous true  
pid\_file /var/run/mosquitto.pid

persistence true

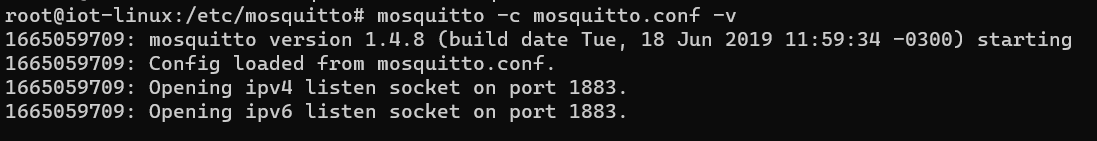
persistence\_location /var/lib/mosquitto/

log\_dest file /var/log/mosquitto/mosquitto.log

include\_dir /etc/mosquitto/conf.d

log\_dest stdout

Step 2 – Start the broker.



Step 3 – Start a subscriber to a specific topic (test\_broker/t1)



Step 4 - Publish a topic and a message in the topic.

*Text

Description automatically generated*

Step 5 – Check if the subscriber receives an update.

**

**Task 4 - Creating an MQTT Python Client**

To interact with Mosquitto, we will use a Python MQTT implementation named Paho, which implements versions 5.0, 3.1.1, and 3.1 of the MQTT protocol. This library provides a client class that enables applications to connect to an MQTT broker to publish messages, subscribe to topics, and receive published messages. It also provides some helper functions to make one-off posting messages to an MQTT server very straightforward. Also, it supports Python 2.7.9+ or 3.6+.

In our example code, we will have two files, one to publish (***mqqt\_publish.py***) and another to subscribe (***mqqt\_subscribe.py***). Both codes require importing the Paho library, defining a group of parameters, and connecting with the broker.

To create the project and install the required libraries, follow these steps:

Step 1 – Create in PyCharm the project **iot-625-python**.

Graphical user interface, text, application, email

Description automatically generated

Step 2 – Open the terminal in PyCharm and install the **paho.mqtt** library.

Text

Description automatically generated

Step 3 Create the publish file (***mqqt\_publish.py***).

import random  
import time  
from paho.mqtt import client as mqtt\_client  
  
broker = '20.249.102.137'  
port = 1883  
topic = "srv/temperature"  
client\_id = f'python-mqtt-{random.randint(0, 1000)}'  
  
def connect\_mqtt():  
 def on\_connect(client, userdata, flags, rc):  
 if rc == 0:  
 print("Connected to MQTT Broker!")  
 else:  
 print("Failed to connect, return code %d\n", rc)  
  
 client = mqtt\_client.Client(client\_id)  
 *# client.username\_pw\_set(username, password)* client.on\_connect = on\_connect  
 client.connect(broker, port)  
 return client  
  
def publish(client):  
 msg\_count = 0  
 while True:  
 time.sleep(1)  
 temperature = 20 + (random.randint(0, 100) \* 4)  
 msg = f"temperature: {temperature}"  
 result = client.publish(topic, msg)  
 *# result: [0, 1]* status = result[0]  
 if status == 0:  
 print(f"Send `{msg}` to topic `{topic}`")  
 else:  
 print(f"Failed to send message to topic {topic}")  
 msg\_count += 1  
  
def run():  
 client = connect\_mqtt()  
 client.loop\_start()  
 publish(client)  
  
run()

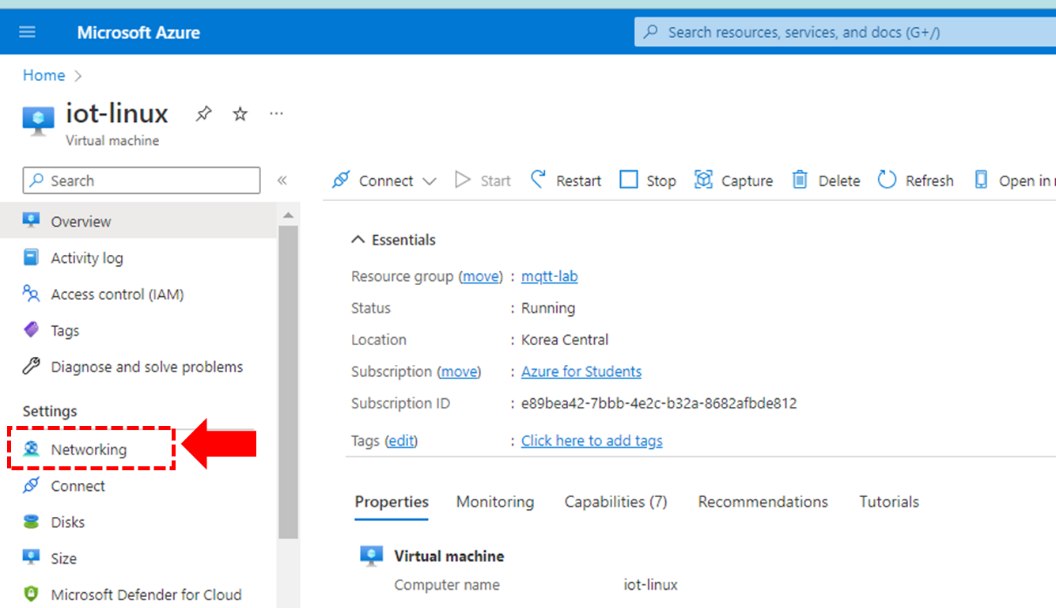
Step 4 – Create the subscribe file (***mqqt\_subscribe.py***) in PyCharm.

import random  
  
from paho.mqtt import client as mqtt\_client  
  
broker = '20.249.102.137'  
port = 1883  
topic = "srv/temperature"  
client\_id = f'python-mqtt-{random.randint(0, 100)}'  
  
def connect\_mqtt() -> mqtt\_client:  
 def on\_connect(client, userdata, flags, rc):  
 if rc == 0:  
 print("Connected to MQTT Broker!")  
 else:  
 print("Failed to connect, return code %d\n", rc)  
  
 client = mqtt\_client.Client(client\_id)  
 client.on\_connect = on\_connect  
 client.connect(broker, port)  
 return client  
  
def subscribe(client: mqtt\_client):  
 def on\_message(client, userdata, msg):  
 print(f"Received `{msg.payload.decode()}` from `{msg.topic}` topic")  
 client.subscribe(topic)  
 client.on\_message = on\_message  
  
def run():  
 client = connect\_mqtt()  
 subscribe(client)  
 client.loop\_forever()  
  
run()

Step 5 – Configure Azure to receive a connection from port 1883.

As you defined in the Linux VM installation, only a few ports are opened to be used in your server. To enable that server to accept connection from port 1883, you need to enable this port in the Azure Firewall.

Go to the Azure Server dashboard and click on Networking.



A new panel is opened and shown the firewall rules that you already have in your Azure server. Click on the **Add inbound port rule** button.

A screenshot of a computer

Description automatically generated with medium confidence

Now, we you configure a MQTT rule to enable Mosquitto to receive a connection from the Internet. The parameters are:

* Source: Any
* Source Port Range: \*
* Destination: Any
* Service: Custom
* Destination Port Range: 1883
* Protocol: TCP
* Action: Allow
* Name: MQTT

Graphical user interface, text, application, email

Description automatically generated

Step 6 – Run the Subscriber code.

Graphical user interface, text, application

Description automatically generated

Step 7 – Run the Publisher code.

Graphical user interface, text

Description automatically generated

Understanding the code

To import Paho, you need to define this line.

from paho.mqtt import client as mqtt\_client

Next, we need to define some variables to start the library.

broker = '192.168.15.12' # set the correct address of the broker  
port = 1883  
topic = "srv/temperature"  
*# generate client ID with pub prefix randomly*client\_id = f'python-mqtt-{random.randint(0, 1000)}'

The last standard part is the connection with a Broker, as you can see in the below code. It is essential that we need to realize an anonymous connection (the previous broker is not set to this feature, we will configure it late).

def connect\_mqtt():  
 def on\_connect(client, userdata, flags, rc):  
 if rc == 0:  
 print("Connected to MQTT Broker!")  
 else:  
 print("Failed to connect, return code %d\n", rc)  
  
 client = mqtt\_client.Client(client\_id)  
 client.on\_connect = on\_connect  
 client.connect(broker, port)  
 return client

An essential part of the code is the definition of the **on\_connect()** callback. There is a param **rc**, which stores a return code given by the MQTT broker and is used to check that the connection was established. The rc’s values are:

0: Connection successful

1: Connection refused – incorrect protocol version

2: Connection refused – invalid client identifier

3: Connection refused – server unavailable

4: Connection refused – Bad username or password

5: Connection refused – not authorized

6-255: Currently unused.

To process the callback, you need to run a loop (it will be explained soon in this lab). Therefore, the script generally looks like this.

1. Create a client object.
2. Create callback function on\_connect()
3. Bind callback to callback function (on\_connect())
4. Connect to Broker.
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. **Your script mustn’t proceed until the link has been established.**

Now we will define the publisher method in the ***mqqt\_publish.py***. The process is simple; you create a local loop that sends a simple message until the broker is closed.

def publish(client):  
 msg\_count = 0  
 while True:  
 time.sleep(1)  
 msg = f"messages: {msg\_count}"  
 result = client.publish(topic, msg)  
 *# result: [0, 1]* status = result[0]  
 if status == 0:  
 print(f"Send `{msg}` to topic `{topic}`")  
 else:  
 print(f"Failed to send message to topic {topic}")  
 msg\_count += 1

Finally, we need to understand the ***mqqt\_subscribe.py*** file. This file has another callback function: **on\_message()**, which creates a loop to wait for the messages published in the channel.

def subscribe(client: mqtt\_client):  
 def on\_message(client, userdata, msg):  
 print(f"Received `{msg.payload.decode()}` from `{msg.topic}` topic")  
  
 client.subscribe(topic)  
 client.on\_message = on\_message

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.

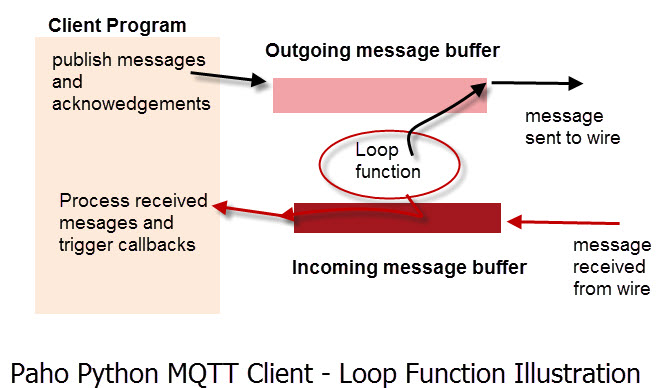


Figure - Loop Function.

You could manually program the client to read the receive buffers, but this would be tedious. The loop() function is a built-in function that reads the receive and sends buffers, and processes any messages it finds. On the receive side, it looks at the messages and will trigger the appropriate callback function depending on the message type.

The Paho Python client provides three methods:

* loop\_start(): starts a new thread that calls the loop method at regular intervals for you. It also handles re-connects automatically.
* loop\_forever(): blocks the program and is useful when the program must run indefinitely. This method function also handles automatic reconnects.
* loop(): call the loop manually, then you will need to create code to handle reconnects.

|  |  |
| --- | --- |
| WOW Deals - Sewell On The Go | **Python MQTT Client Connections– Working with Connections**  <http://www.steves-internet-guide.com/client-connections-python-mqtt/>  **Paho Python MQTT Client-Understanding The Loop**  <http://www.steves-internet-guide.com/loop-python-mqtt-client/> |

**Task 5 – Configuring Mosquitto to run in Bridge Mode**

Mosquitto has a bridging feature, which lets you connect two (or more) brokers together. They are generally used for sharing messages between systems. Typical usage is connected edge MQTT brokers to a central or remote MQTT network. The Mosquitto broker (server) can be configured to work as an MQTT bridge. Generally, the local edge bridge will only bridge a subset of the local MQTT traffic.

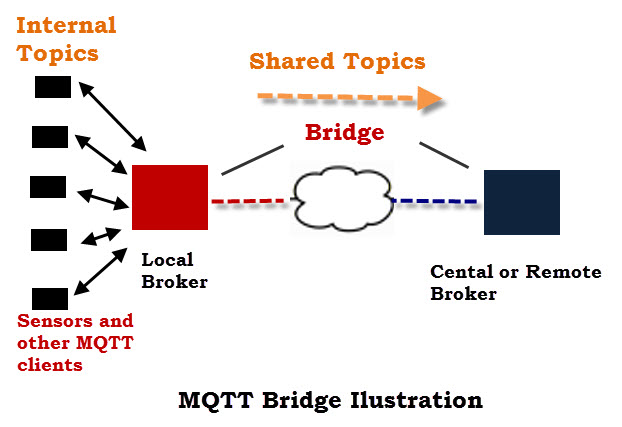


Figure - Bridge Mode.

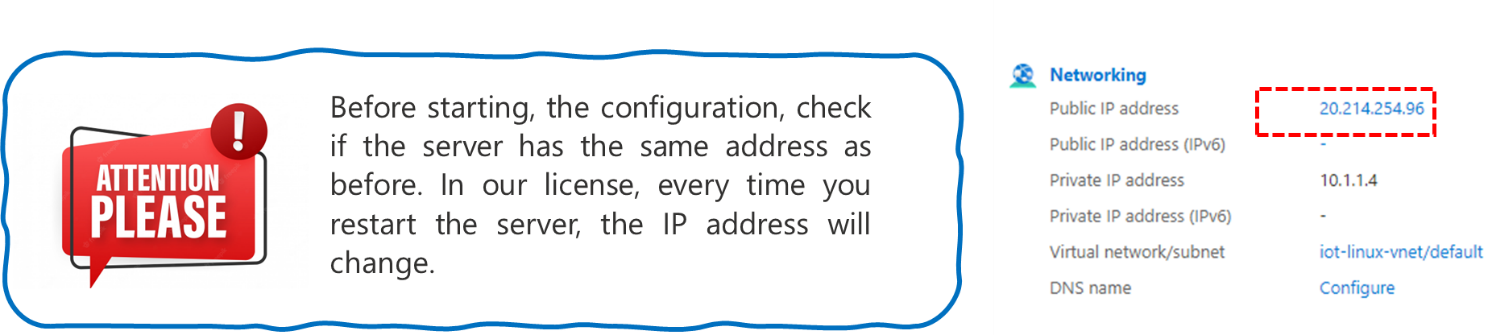
In this task, we will implement the architecture showed in the next Figure. On that we have two Mosquitto: a **local\_mosquitto** and a **central\_mosquitto**. The local redirects all messages received from the sensors to the central one. Because we run both servers on the same machine, central\_mosquitto uses port 1884. Also, it will authenticate all brokers that aim to send messages to it.

Graphical user interface, application

Description automatically generated

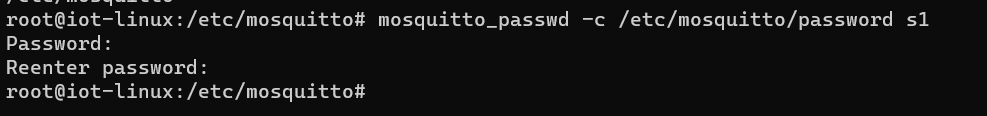
Following the above architecture, when S1\_PUB publish a message, local\_mosquitto will redirect it to centraL\_mosquitto. When central\_mosquitto receives the message, it sends it to all subscribers.

As we cited before the central\_mosquitto does not accept any anonymous connection, requiring a password file, where you define the users and password.

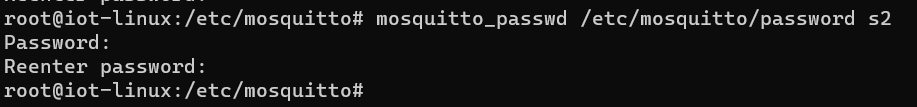


**Step 1 – Create the password file to central\_mosquitto.**

To create a password file, use the mosquitto\_passwd utility. You will be asked for the password. Note that -c means an existing file will be overwritten. To run this command, you define the user **s1** and password **s123456789**. The syntax of the command is: *mosquitto\_passwd -c <password file> <username>*



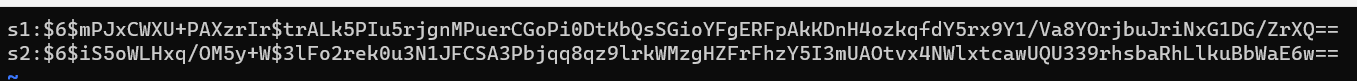
To add more users to an existing password file, or to change the password for an existing user, leave out the -c argument. Also, to remove a user from a password file, you need to use the **-D** option. In the next example, we will create a new user **s2** with **s2987654321** as a password.



Finally, we create an account to authenticate the broker that works like a bridge. The user will be **bridge1** and the password is **bridge123456789**.

You can see the password file using the vim command. The command saves the hash of the password using a SHA512 format.



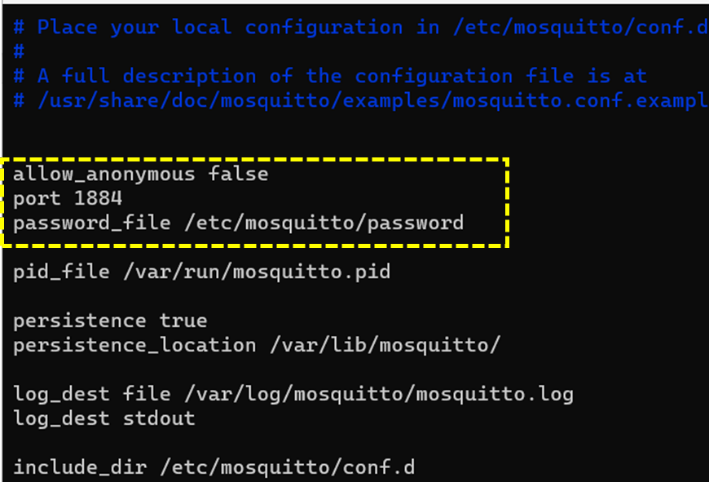


**Step 2 – Create the configuration file of central\_mosquitto**

In this step, we will create a copy of the original file and name it as **mosquito\_ central.conf**.



Using the **vim** command (or any other editor), we edit the configuration file to set the port to 1884, improve access to disable the anonymous connection, and define a password file to select the authorized users (can be more than one). The file will have this content:



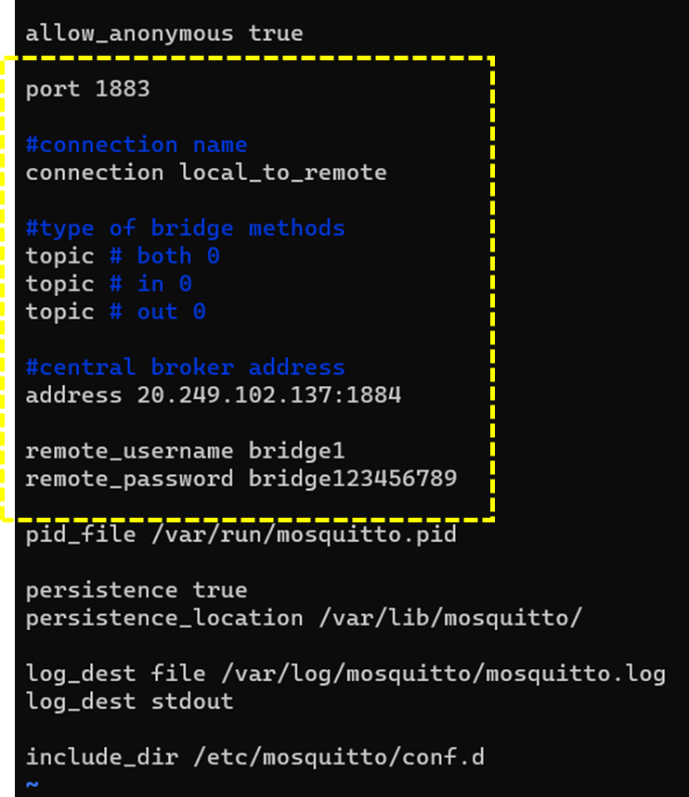
**Step 3 – Create the configuration file of local\_mosquitto**

In this step, we will create a copy of the original file and name it **local\_mosquito.conf**. Use the same process shown in the previous step. The content is quite simple and like the central one; however, you enable the anonymous connection (or not). The main difference is the definition of the connection name, the address of the remote broker, the direction of topics you bridge, and the remote authentication information.

The direction can be out, in, or both.

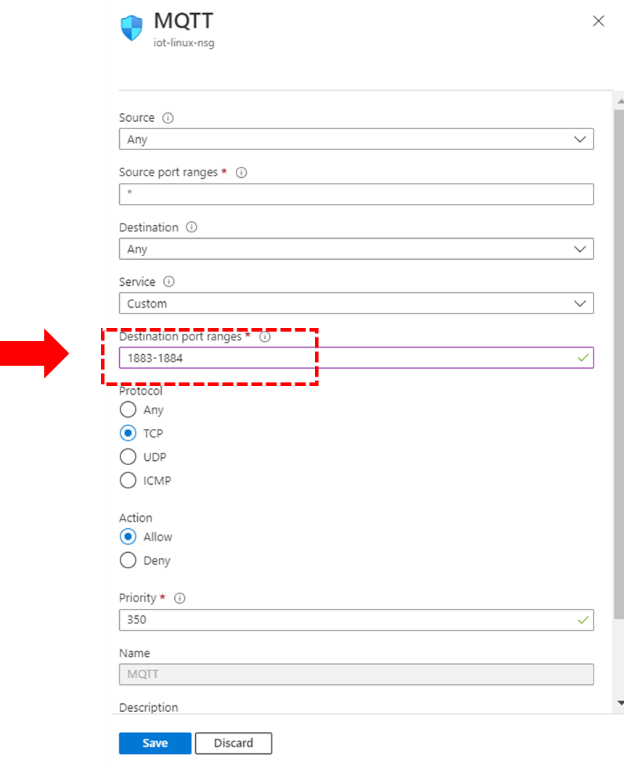
* out = publish from the broker
* in = receive from the remote broker
* both = publish and receive

The file content will be:



Step 4 – Configure Azure to receive a connection from port 1884 and 1883.

To add a new port, we use the same approach used. In the Networking, open the MQTT rule and change the Destination port range to 1883-1884, authorizing both ports in the Azure Linux.



Step 5 – Testing the Bridge

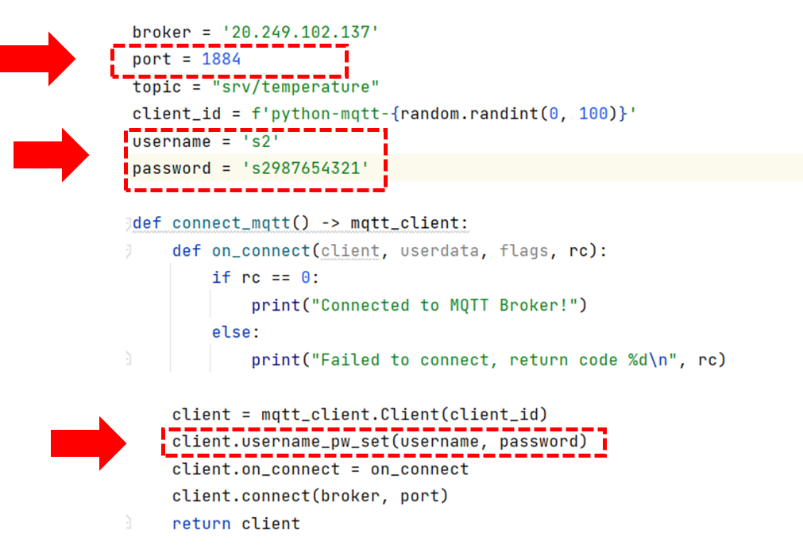
Graphical user interface, application

Description automatically generated

Now, we need to change the script files to reflect the scenario, where the publisher will use the mosquitto\_local (1883) and the subscriber the mosquitto\_central (1884). Also, the subscriber needs to set a username and password. The changes in the subscriber code are shown below.

Text

Description automatically generated



Also, we need to start two different instances of Mosquitto using a different shell. You need to start the central first because it will receive the bridge connections.

Text

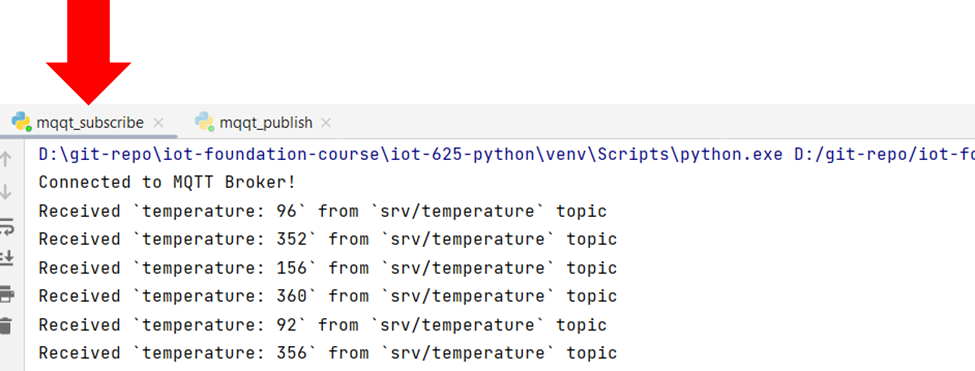
Description automatically generated

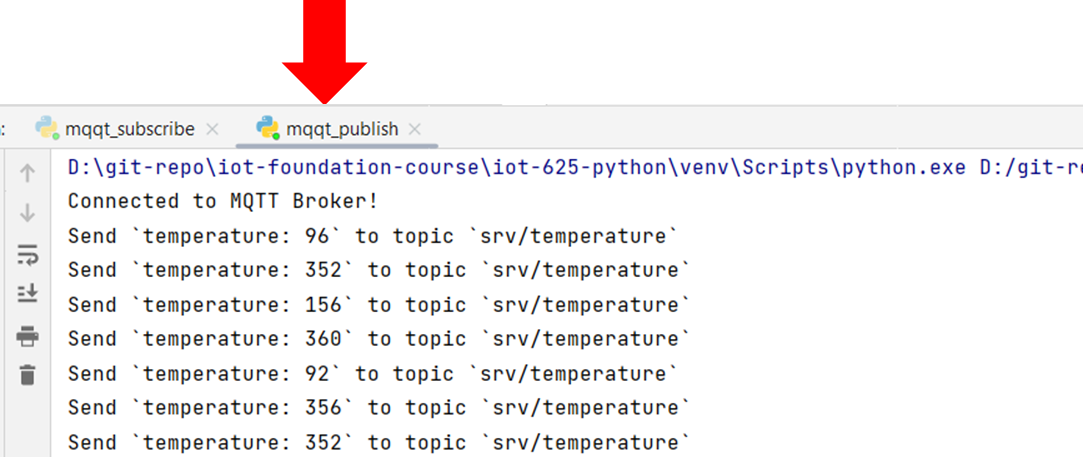
Next, you will start the bridges (if you have more than one).

Text

Description automatically generated

Now, you can run the subscriber and the publisher. They need to receive and send the messages.





Now, you just need to test and check that everything works.

1. To check how to configure Mosquitto as a Windows Service, check this link: <http://www.steves-internet-guide.com/install-mosquitto-broker/>. [↑](#footnote-ref-2)