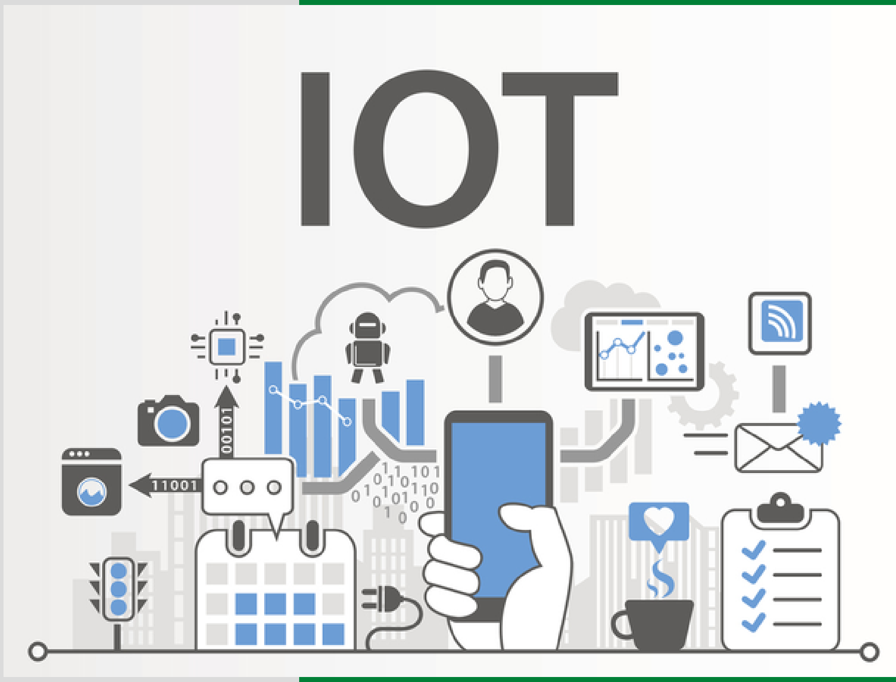
ALEXANDRE B BARRETO

IOT PROTOCOLS

MQTT FOUNDATIONS

AUTHOR
Alexandre B Barreto
barretoabb@tec.mx



GOAL

This exercise aims to introduce the basic concepts about how to implement a MQTT infrastructure.

REVIEW THE MQTT PROTOCOL

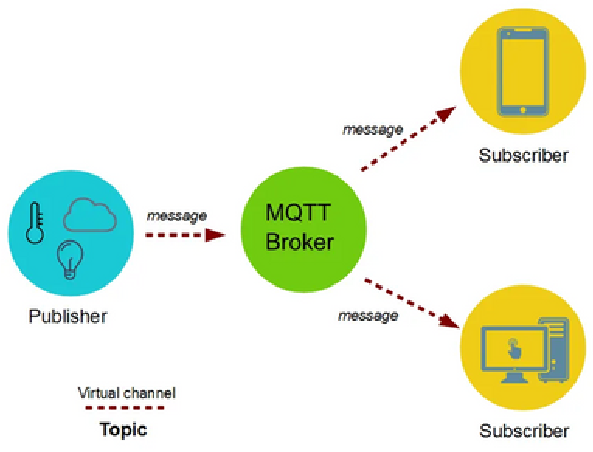
MQTT protocol is a Machine to Machine (M2M) protocol widely used in IoT (Internet of things). The MQTT protocol is a message-based, extraordinarily lightweight protocol, so it is adopted in IoT. Almost all IoT platforms support MQTT to send and receive data from smart objects.

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.



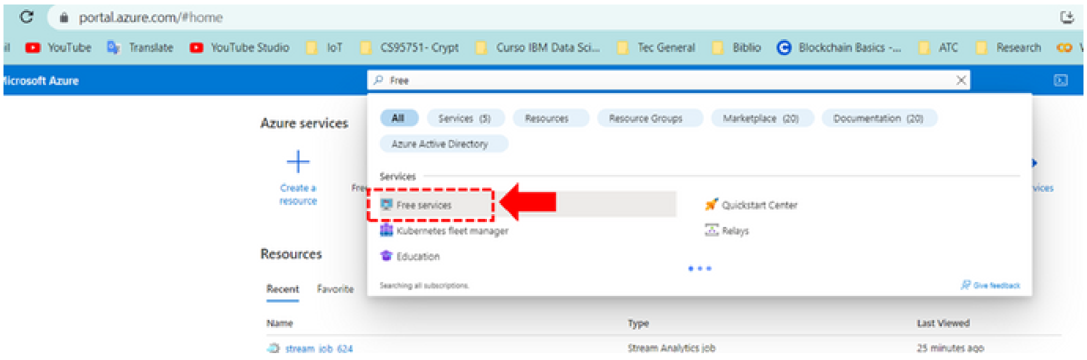
The critical component in MQTT is the **broker**. The main task of the MQTT broker is dispatching messages to the MQTT clients ("subscribers"). In other words, the **MQTT broker receives messages from the publisher and dispatches these messages to the subscribers**.

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").

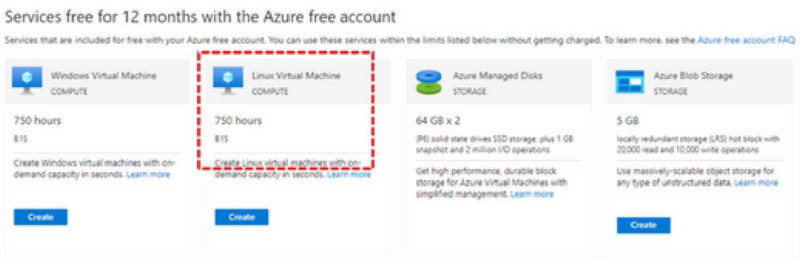
What is MQTT (MQ Telemetry Transport)?
https://www.youtube.com/watch?v=6XqGXw_5NhM

TASK 1 – CREATE AN UBUNTU MACHINE IN AZURE

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



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


Services free for 12 months with the Azure free account

Services that are included for free with your Azure free account. You can use these services within the limits listed below without getting charged. To learn more, see the Azure free account FAQ

Windows Virtual Machine
COMPUTE
750 hours
B1S
Create Windows virtual machines with on-demand capacity in seconds. Learn more
Create

Linux Virtual Machine
COMPUTE
750 hours
B1S
Create Linux virtual machines with on-demand capacity in seconds. Learn more
Create

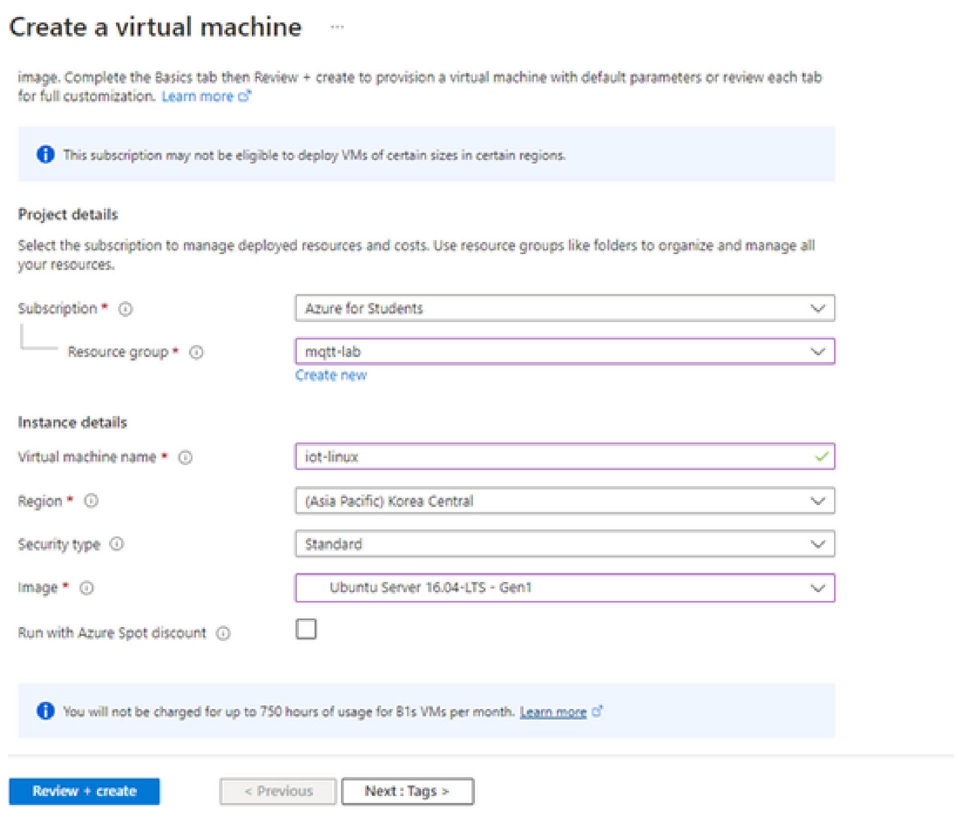
Azure Managed Disks
STORAGE
64 GB x 2
(P6) solid state drives SSD storage, plus 1 GB snapshot and 2 million I/O operations
Get high performance, durable block storage for Azure Virtual Machines with simplified management. Learn more

Azure Blob Storage
STORAGE
5 GB
locally redundant storage (LRS) hot block with 20,000 read and 10,000 write operations
Use massively-scalable object storage for any type of unstructured data. Learn more
Create

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

ATTENTION PLEASE

Remember that this configuration will not be charged for up to 750 hours of usage for B1s VM per month.



Create a virtual machine ···

image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. Learn more

This subscription may not be eligible to deploy VMs of certain sizes in certain regions.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Azure for Students
Resource group *	mqtt-lab
	Create new

Instance details

Virtual machine name *	iot-linux ✓
Region *	(Asia Pacific) Korea Central
Security type	Standard
Image *	Ubuntu Server 16.04-LTS - Gen1
Run with Azure Spot discount	☐

You will not be charged for up to 750 hours of usage for B1s VMs per month. Learn more

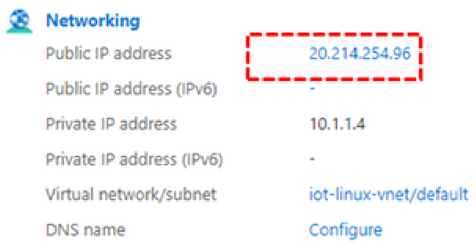
ATTENTION PLEASE

Now, Azure will create and start your virtual machine. Remember, when you are not using the device, shut it down, avoiding exceeding the monthly amount of time accessible to your type of account.

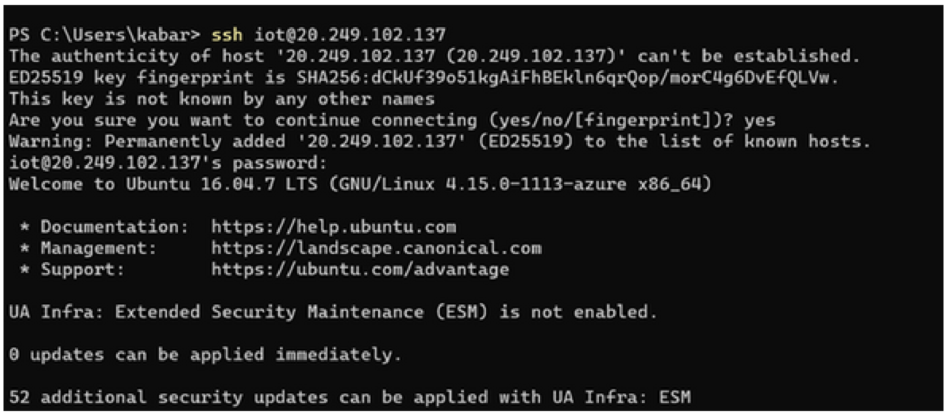
To have access to the server, open a **Windows Power Shell Terminal** (or **simple Linux / Mac terminal**) and run the following commands: **ssh iot@<address of VM>**. Validate the returned fingerprint. If you have never connected to this VM before you will be asked to verify the host's fingerprint. 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.

ATTENTION PLEASE

Before starting, the configuration, check if the server has the same address as before. In our license, every time you restart the server, the IP address will change.



Networking	
Public IP address	20.214.254.96
Public IP address (IPv6)	-
Private IP address	10.1.1.4
Private IP address (IPv6)	-
Virtual network/subnet	iot-linux-vnet/default
DNS name	Configure



```
PS C:\Users\kabar> ssh iot@20.249.102.137
The authenticity of host '20.249.102.137 (20.249.102.137)' can't be established.
ED25519 key fingerprint is SHA256:dCkUf39o51kgAiFhBEkln6qrQop/morC4g6DvEfQLVw.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '20.249.102.137' (ED25519) to the list of known hosts.
iot@20.249.102.137's password:
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.15.0-1113-azure x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

UA Infra: Extended Security Maintenance (ESM) is not enabled.

0 updates can be applied immediately.

52 additional security updates can be applied with UA Infra: ESM
```

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.

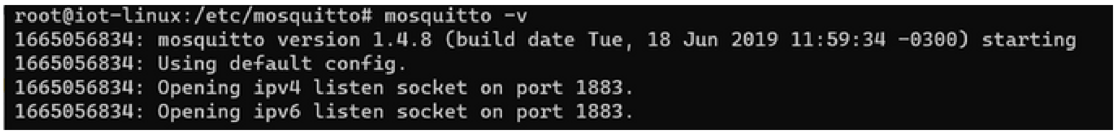
The MQTT protocol provides a lightweight method of carrying out messaging using a publish/subscribe model. This makes it suitable for Internet of Things messaging, such as with low-power sensors or mobile devices such as phones, embedded computers, or microcontrollers.

TASK 2 - INSTALLATION AND BASIC TEST OF MQQT MOSQUITTO BROKER

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

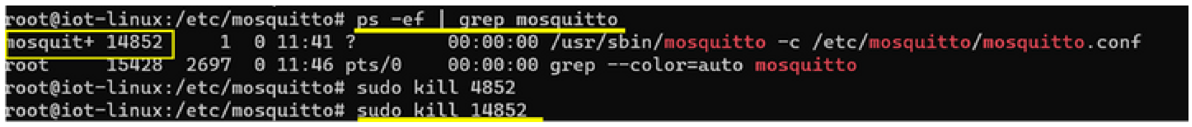
- **sudo apt-get install mosquitto**
- **sudo apt-get install mosquitto-clients**
- **sudo apt clean**

To test Mosquitto type this command: **mosquitto -v**. This command initializes Mosquitto in a verbose mode.



```
root@iot-linux:/etc/mosquitto# mosquitto -v
1665056834: mosquitto version 1.4.8 (build date Tue, 18 Jun 2019 11:59:34 -0300) starting
1665056834: Using default config.
1665056834: Opening ipv4 listen socket on port 1883.
1665056834: Opening ipv6 listen socket on port 1883.
```

If when you run the command, you receive this message: "Error address already in use", kill the process and run again the command.



```
root@iot-linux:/etc/mosquitto# ps -ef | grep mosquitto
mosquit+ 14852     1  0 11:41 ?        00:00:00 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf
root     15428  2697  0 11:46 pts/0    00:00:00 grep --color=auto mosquitto
root@iot-linux:/etc/mosquitto# sudo kill 4852
root@iot-linux:/etc/mosquitto# sudo kill 14852
```

TASK 3 - CONFIGURATION OF MQQT MOSQUITTO BROKER

You can configure the Mosquitto broker using a configuration file. The default configuration file is **mosquitto.conf**, 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 to add a line to logging destination is printed in the **stdout**.

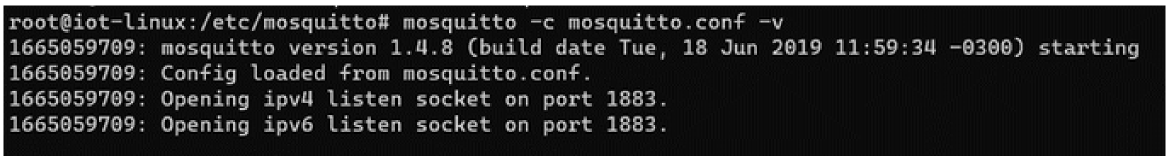
The Mosquitto project also provides a C library for implementing MQTT clients and the very popular **mosquitto_pub** and **mosquitto_sub** command line MQTT clients. To test the implementation, we will open 3 terminal tabs in the Power Shell, remote log in Azure server (using ssh), and become superuser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

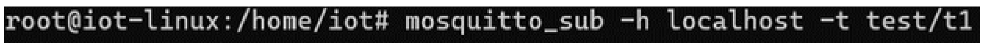
```
allow_anonymous true

pid_file /var/run/mosquitto.pid
persistence true
persistence_location /var/lib/mosquitto/
log_dest file /var/log/mosquitto/mosquitto.log
log_dest stdout

include_dir /etc/mosquitto/conf.d
```

Step 2 – Start the broker.



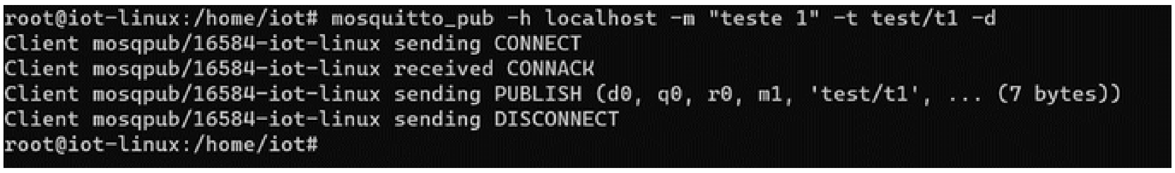
```
root@iot-linux:/etc/mosquitto# mosquitto -c mosquitto.conf -v
1665059709: mosquitto version 1.4.8 (build date Tue, 18 Jun 2019 11:59:34 -0300) starting
1665059709: Config loaded from mosquitto.conf.
1665059709: Opening ipv4 listen socket on port 1883.
1665059709: Opening ipv6 listen socket on port 1883.
```

Step 3 – Start a subscriber to a specific topic (test_broker/t1)



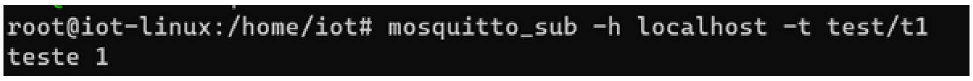
```
root@iot-linux:/home/iot# mosquitto_sub -h localhost -t test/t1
```

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



```
root@iot-linux:/home/iot# mosquitto_pub -h localhost -m "teste 1" -t test/t1 -d
Client mosqpub/16584-iot-linux sending CONNECT
Client mosqpub/16584-iot-linux received CONNACK
Client mosqpub/16584-iot-linux sending PUBLISH (d0, q0, r0, m1, 'test/t1', ... (7 bytes))
Client mosqpub/16584-iot-linux sending DISCONNECT
root@iot-linux:/home/iot#
```

Step 5 – Check if the subscriber receives an update.



```
root@iot-linux:/home/iot# mosquitto_sub -h localhost -t test/t1
teste 1
```

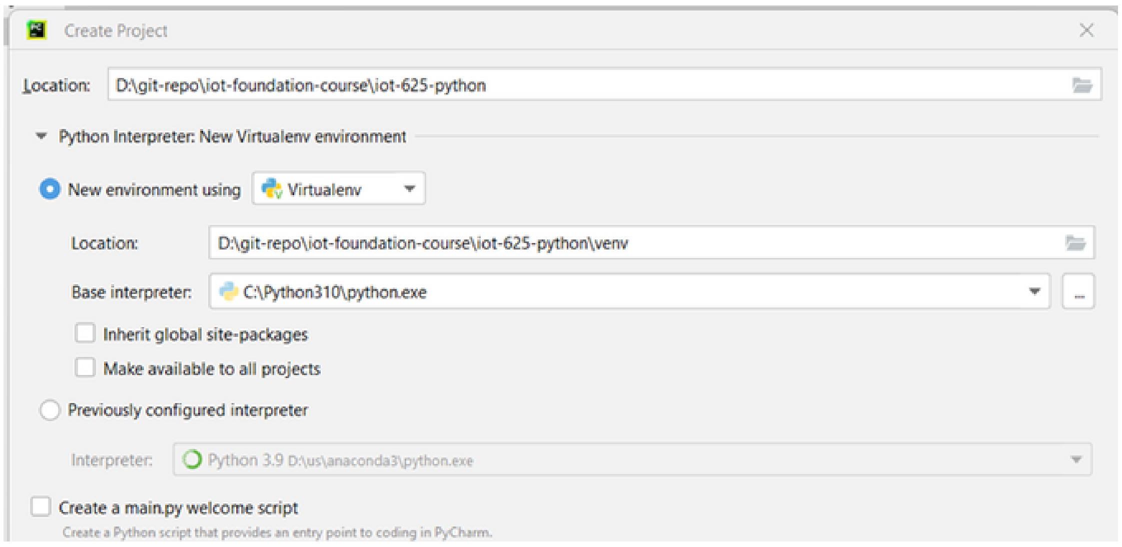
TASK 4 - CREATING AN MQTT PYTHON CLIENT

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.



Create Project

Location: D:\git-repo\iot-foundation-course\iot-625-python

Python Interpreter: New Virtualenv environment

[✓ New environment using] Virtualenv

Location: D:\git-repo\iot-foundation-course\iot-625-python\venv

Base interpreter: C:\Python310\python.exe

☐ Inherit global site-packages

☐ Make available to all projects

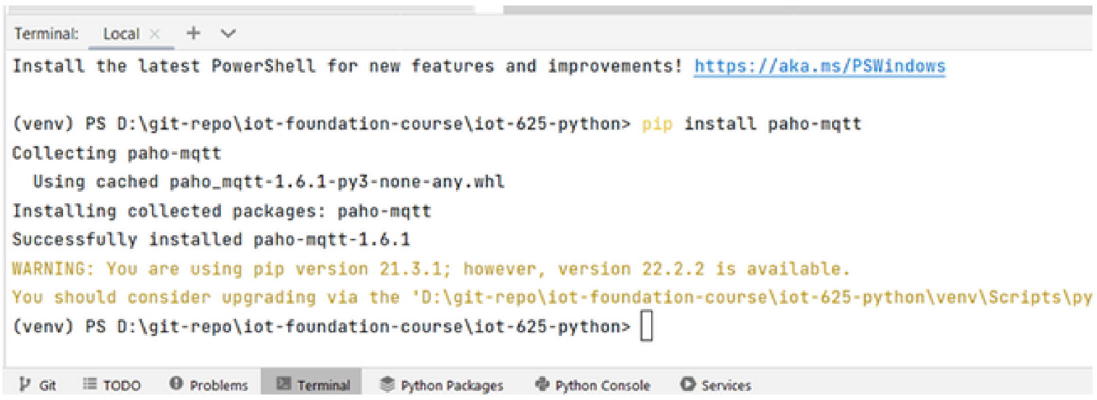
Previously configured interpreter

Interpreter: Python 3.9 D:\us\anaconda3\python.exe

☐ Create a main.py welcome script

Create a Python script that provides an entry point to coding in PyCharm.

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



```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

(venv) PS D:\git-repo\iot-foundation-course\iot-625-python> pip install paho-mqtt
Collecting paho-mqtt
  Using cached paho_mqtt-1.6.1-py3-none-any.whl
Installing collected packages: paho-mqtt
Successfully installed paho-mqtt-1.6.1
WARNING: You are using pip version 21.3.1; however, version 22.2.2 is available.
You should consider upgrading via the 'D:\git-repo\iot-foundation-course\iot-625-python\venv\Scripts\py
(venv) PS D:\git-repo\iot-foundation-course\iot-625-python> 
```

Step 3 Create the publish file (mqqt_publish.py).

[1] The code is available at https://github.com/kabartsjc/iot-foundation-course/tree/main/iot-625-python

```python
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.

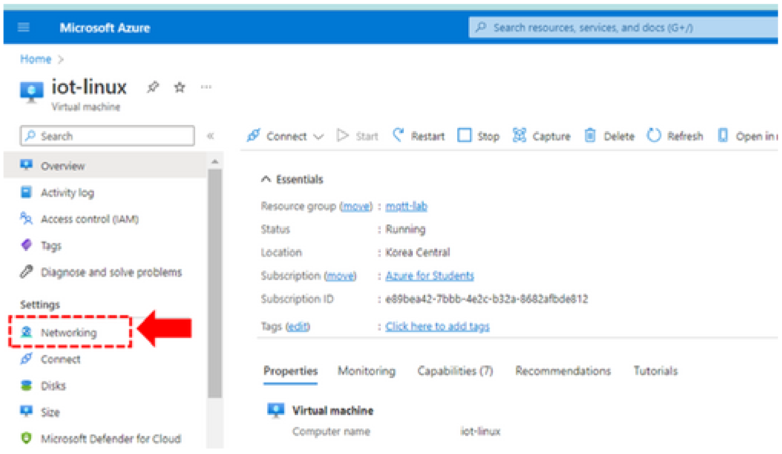
[1] The code is available at https://github.com/kabartsjc/iot-foundation-course/tree/main/iot-625-python

```
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. So, to enable that server to accept connection from port 1883, you need to allow this port in the Azure Firewall.

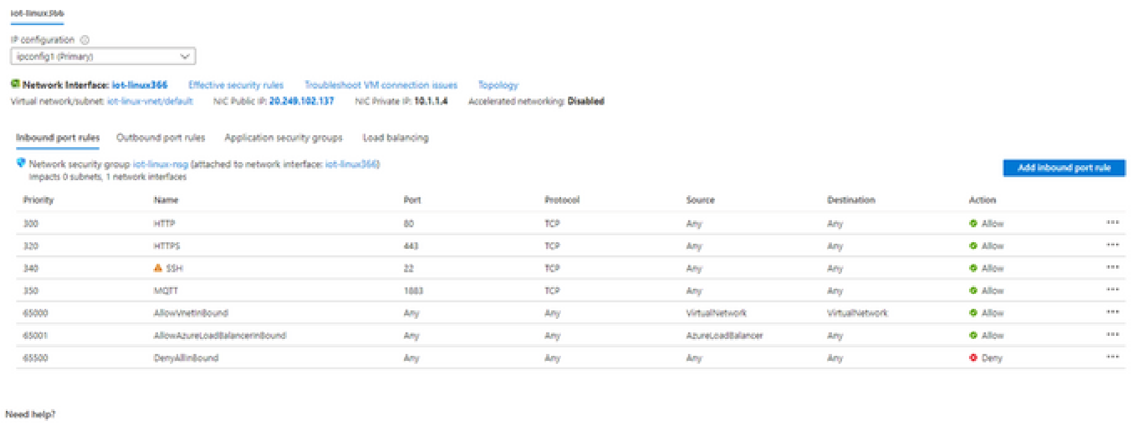
Go to the Azure Server dashboard and click on Networking.



Resource group (move) : mqtt-lab
Status : Running
Location : Korea Central
Subscription (move) : Azure for Students
Subscription ID : e89bea42-7bbb-4e2c-b32a-8682afbde812
Tags (edit) : Click here to add tags

Computer name iot-linux

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.



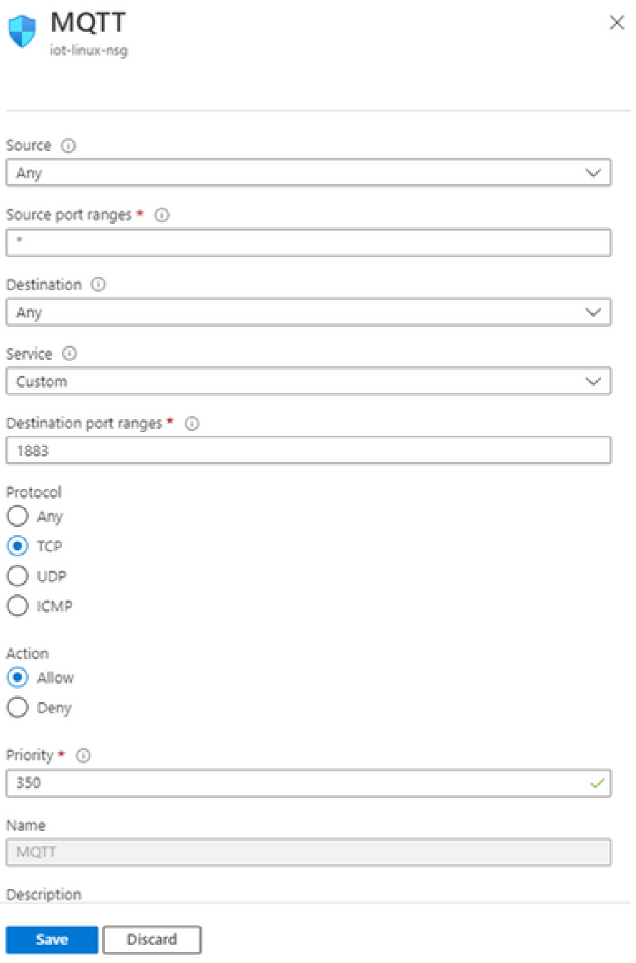
Network Interface: iot-linux366 Effective security rules Troubleshoot VM connection issues Topology
Virtual network/subnet: iot-linux-vnet/default NIC Public IP: 20.249.102.137 NIC Private IP: 10.1.1.4 Accelerated networking: Disabled

Network security group iot-linux-nsg (attached to network interface: iot-linux366)
Impacts 0 subnets, 1 network interfaces

Priority	Name	Port	Protocol	Source	Destination	Action
300	HTTP	80	TCP	Any	Any	Allow
320	HTTPS	443	TCP	Any	Any	Allow
340	SSH	22	TCP	Any	Any	Allow
350	MQTT	1883	TCP	Any	Any	Allow
65000	AllowVnetInBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowAzureLoadBalancerInBound	Any	Any	AzureLoadBalancer	Any	Allow
65500	DenyAllInBound	Any	Any	Any	Any	Deny

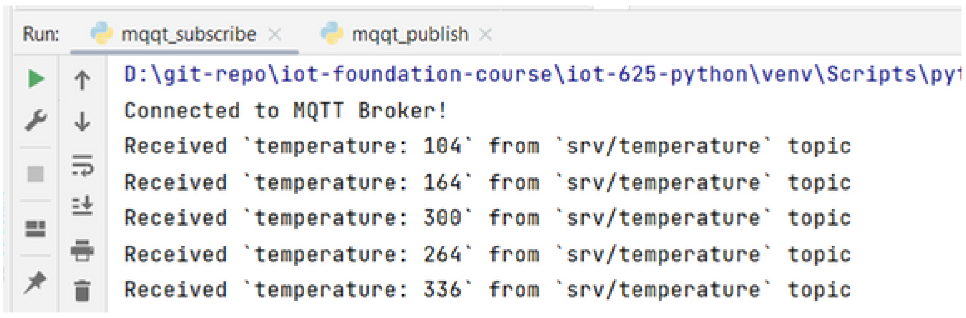
Now, we have configured an MQTT rule to enable Mosquitto to receive a connection from the Internet. The parameters are:

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


MQTT
iot-linux-nsg

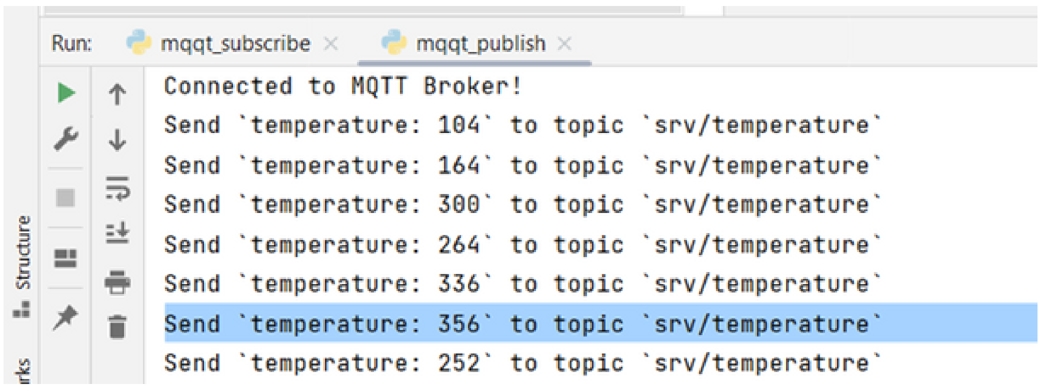
Source: Any
Source port ranges *: *
Destination: Any
Service: Custom
Destination port ranges *: 1883
Protocol: Any [✓ TCP] UDP ICMP
Action: [✓ Allow] Deny
Priority *: 350
Name: MQTT
Description

Step 6 – Run the Subscriber code.



```
D:\git-repo\iot-foundation-course\iot-625-python\venv\Scripts\pyt
Connected to MQTT Broker!
Received `temperature: 104` from `srv/temperature` topic
Received `temperature: 164` from `srv/temperature` topic
Received `temperature: 300` from `srv/temperature` topic
Received `temperature: 264` from `srv/temperature` topic
Received `temperature: 336` from `srv/temperature` topic
```

Step 7 – Run the Publisher code.



```
Connected to MQTT Broker!
Send `temperature: 104` to topic `srv/temperature`
Send `temperature: 164` to topic `srv/temperature`
Send `temperature: 300` to topic `srv/temperature`
Send `temperature: 264` to topic `srv/temperature`
Send `temperature: 336` to topic `srv/temperature`
Send `temperature: 356` to topic `srv/temperature`
Send `temperature: 252` to topic `srv/temperature`
```

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. Again, 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**. A param **rc** 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 the 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 is triggered. Therefore, your **script mustn't proceed until the link has been established.**

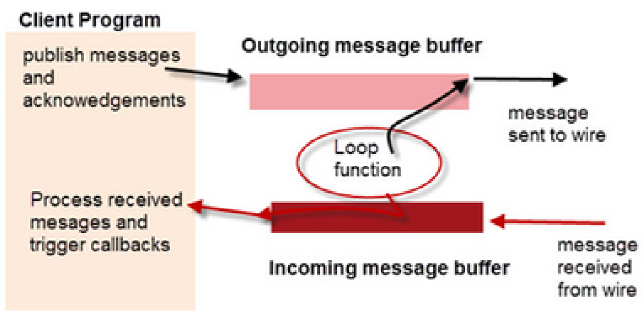
Now we will define the publisher method in the **mqqt_publish.py. Again, the** process is simple; you create a local loop that sends a simple message until the broker is closed.

```
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.


Client Program
publish messages and acknowedgements
Process received mesages and trigger callbacks
Outgoing message buffer
Loop function
message sent to wire
Incoming message buffer
message received from wire

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:

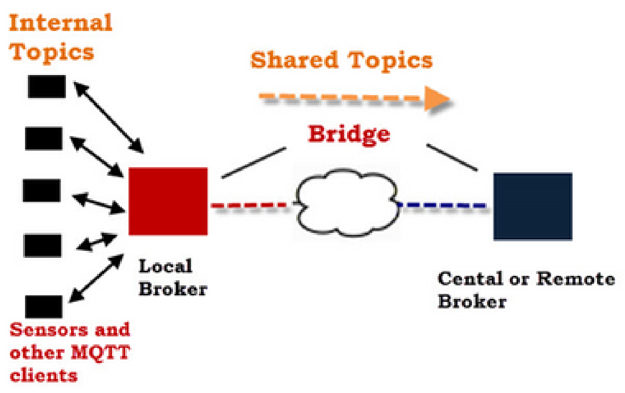
- loop_start(): starts a new thread that calls the loop method at regular intervals for you. It also handles re-connects automatically.
- loop_forever(): blocks the program and is useful when the program must run indefinitely. This method function also handles automatic reconnects.
- loop(): call the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

Want to know more?

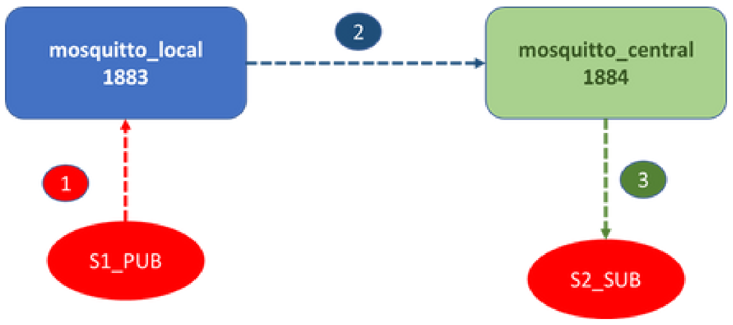
Python MQTT Client Connections– Working with Connections
http://www.steves-internet-guide.com/client-connections-python-mqtt/
Paho Python MQTT Client-Understanding The Loop
http://www.steves-internet-guide.com/loop-python-mqtt-client/

TASK 5 – CONFIGURING MOSQUITTO TO RUN IN BRIDGE MODE

Mosquitto has a bridging feature that lets you connect two (or more) brokers. 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.



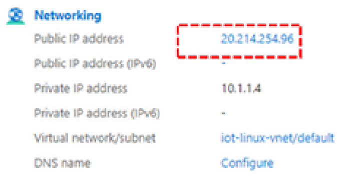
We will implement the architecture in this task in the following Figure. On that, we have two Mosquitto: a **local_mosquitto** and a **central_mosquitto**. The local redirects all messages received from the sensors to the central one. Because we run both servers on the same machine, central_mosquitto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



Following the above architecture, when S1_PUB publishes 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.

ATTENTION PLEASE

Before starting, the configuration, check if the server has the same address as before. In our license, every time you restart the server, the IP address will change.

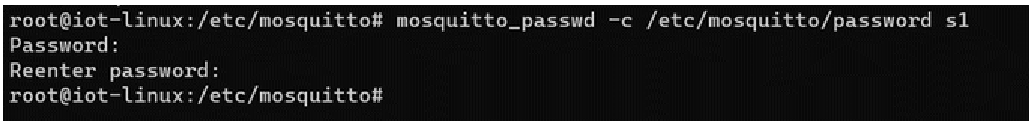


Networking

Public IP address	20.214.254.96
Public IP address (IPv6)	-
Private IP address	10.1.1.4
Private IP address (IPv6)	-
Virtual network/subnet	iot-linux-vnet/default
DNS name	Configure

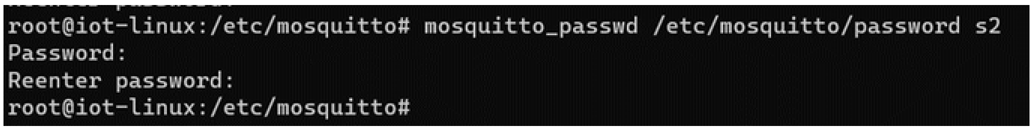
Step 1 – Create the password file to central_mosquitto.

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>



```
root@iot-linux:/etc/mosquitto# mosquitto_passwd -c /etc/mosquitto/password s1
Password:
Reenter password:
root@iot-linux:/etc/mosquitto#
```

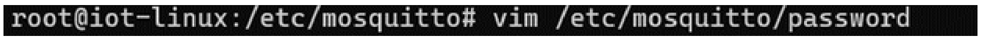
To add more users to an existing password file, or to change the password for an existing user, leave out the -c argument. Also, to remove a user from a password file, you need to use the **-D option**. In the next example, we will create a new user **s2** with **s2987654321** as a password.



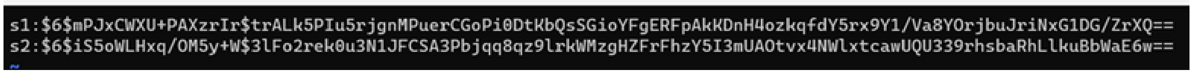
```
root@iot-linux:/etc/mosquitto# mosquitto_passwd /etc/mosquitto/password s2
Password:
Reenter password:
root@iot-linux:/etc/mosquitto#
```

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.



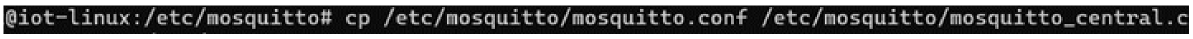
```
root@iot-linux:/etc/mosquitto# vim /etc/mosquitto/password
```



```
s1:$6$mPJxCWXU+PAXzrIr$trALk5PIu5rjgnMPuerCGoPi0DtKbQsSGioYFgERFpAkKDnH4ozkqfdY5rx9Y1/Va8YOrjbuJriNxG1DG/ZrXQ==
s2:$6$iS5oWLHxq/OM5y+W$3lFo2rek0u3N1JFCSA3Pbjqq8qz9lrkWMzgHZFrFhzY5I3mUAOtvx4NWlxtcawUQU339rhsbaRhLlkuBbWaE6w==
~
```

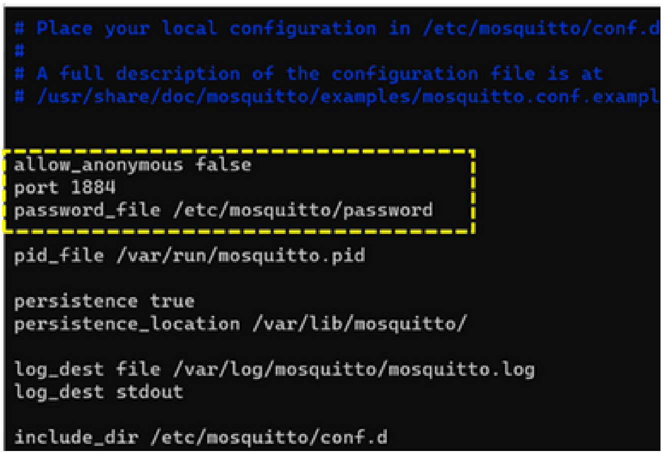
Step 2 – Create the configuration file of central_mosquitto

In this step, we will create a copy of the original file and name it mosquito_ central.conf.



```
@iot-linux:/etc/mosquitto# cp /etc/mosquitto/mosquitto.conf /etc/mosquitto/mosquitto_central.c
```

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:



```
# Place your local configuration in /etc/mosquitto/conf.d
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.exampl

allow_anonymous false
port 1884
password_file /etc/mosquitto/password

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log
log_dest stdout

include_dir /etc/mosquitto/conf.d
```

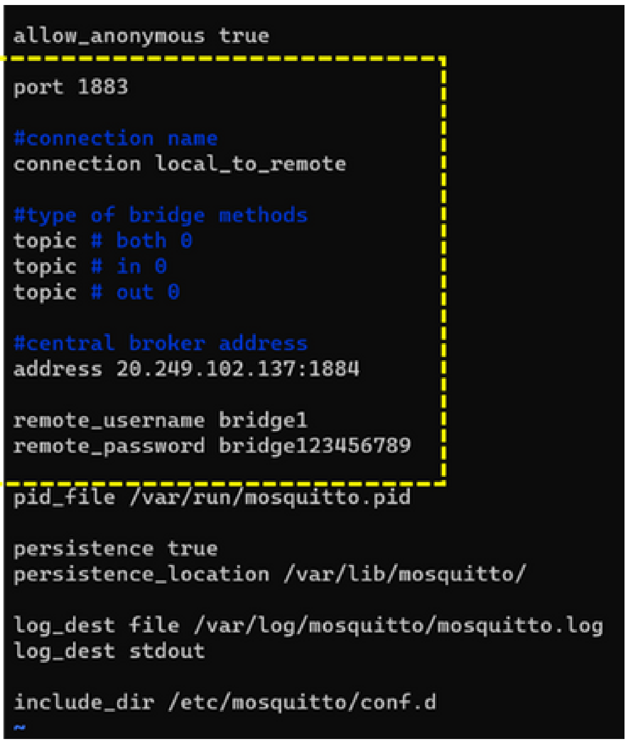
Step 3 – Create the configuration file of local_mosquitto

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

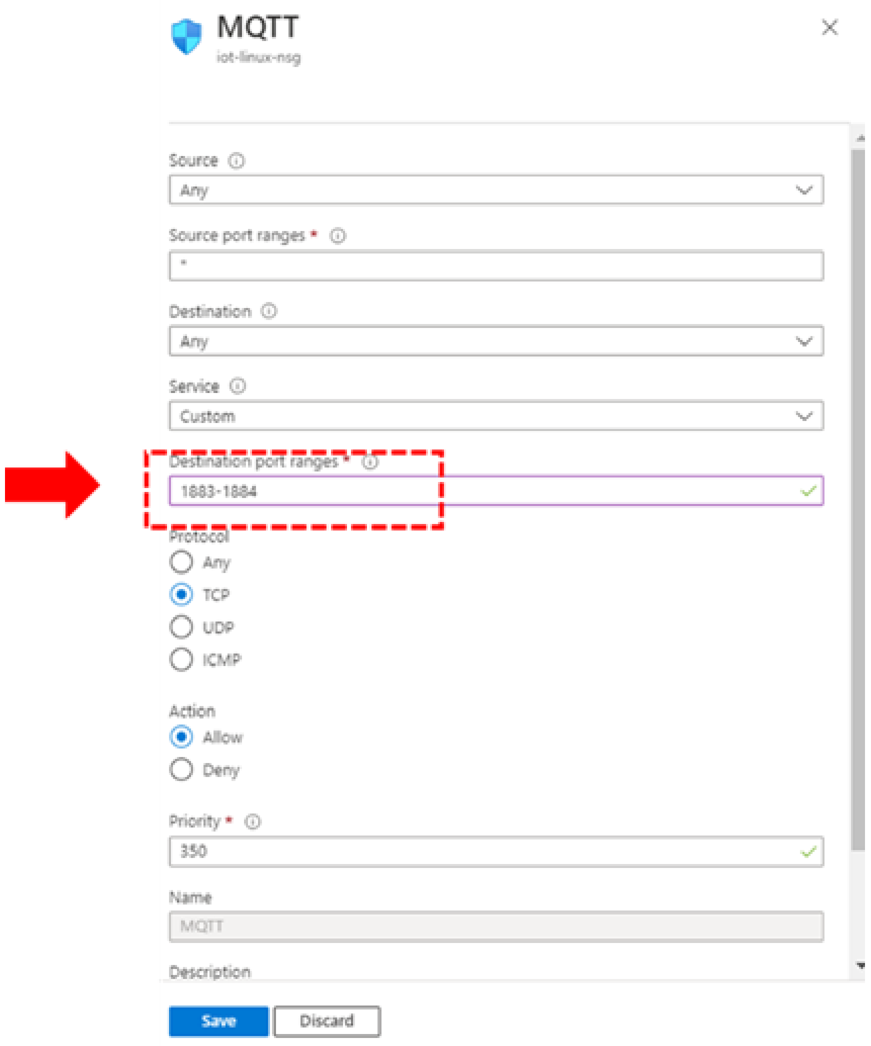
The file content will be:



```
allow_anonymous true

port 1883

#connection name
connection local_to_remote

#type of bridge methods
topic # both 0
topic # in 0
topic # out 0

#central broker address
address 20.249.102.137:1884

remote_username bridge1
remote_password bridge123456789

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log
log_dest stdout

include_dir /etc/mosquitto/conf.d
~
```

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.



MQTT

iot-linux-nsg

Source: Any

Source port ranges *: *

Destination: Any

Service: Custom

Destination port ranges *: 1883-1884

Protocol: Any [✓ TCP] UDP ICMP

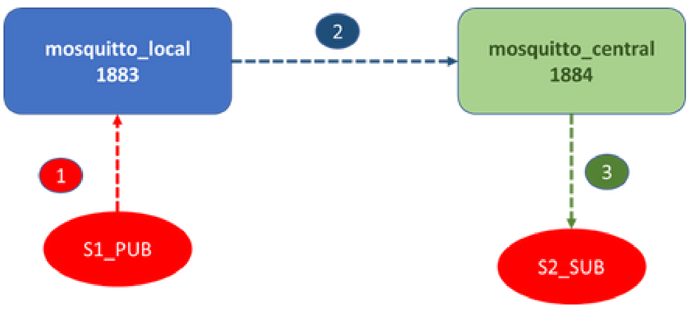
Action: [✓ Allow] Deny

Priority *: 350

Name: MQTT

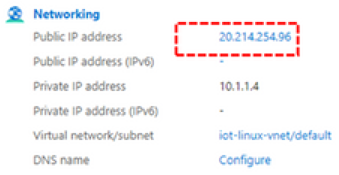
Description

Step 5 – Testing the Bridge

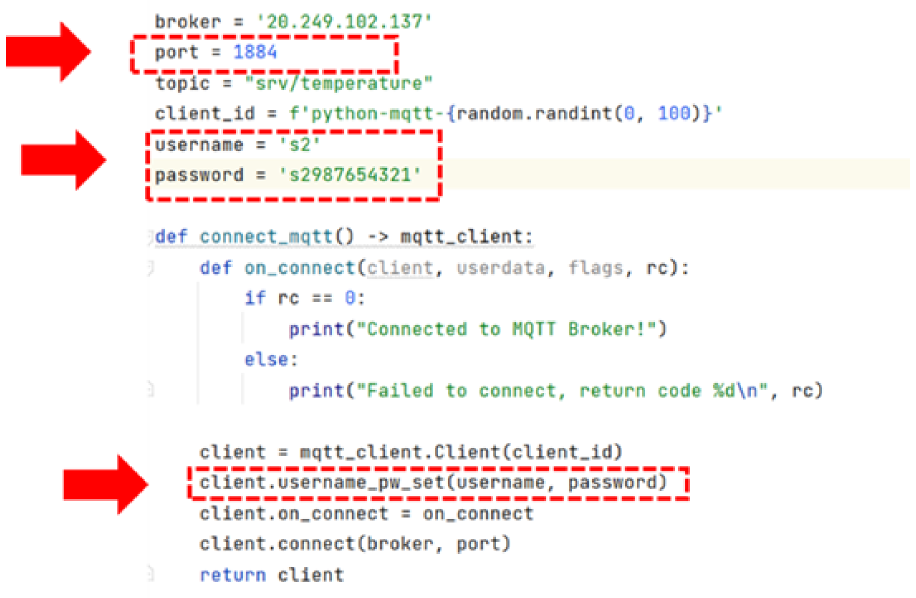


Now, we need to change the script files to reflect the scenario, where the publisher will use the mosquitto_local (1883) and the subscriber the mosquitto_central (1884). Also, the subscriber needs to set a username and password. The changes in the subscriber code are shown below.

Before starting, the configuration, check if the server has the same address as before. In our license, every time you restart the server, the IP address will change.

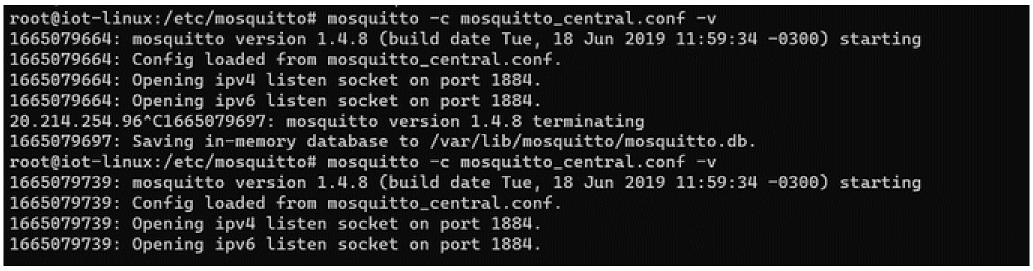


Networking	
Public IP address	20.214.254.96
Public IP address (IPv6)	-
Private IP address	10.1.1.4
Private IP address (IPv6)	-
Virtual network/subnet	iot-linux-vnet/default
DNS name	Configure



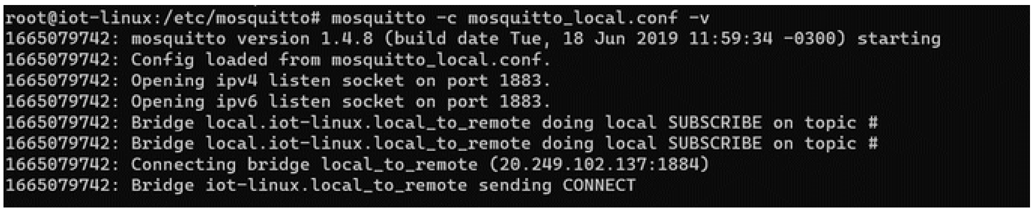
```
broker = '20.249.102.137'
port = 1884
topic = "srv/temperature"
client_id = f'python-mqtt-{random.randint(0, 100)}'
username = 's2'
password = 's2987654321'

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.username_pw_set(username, password)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client
```

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



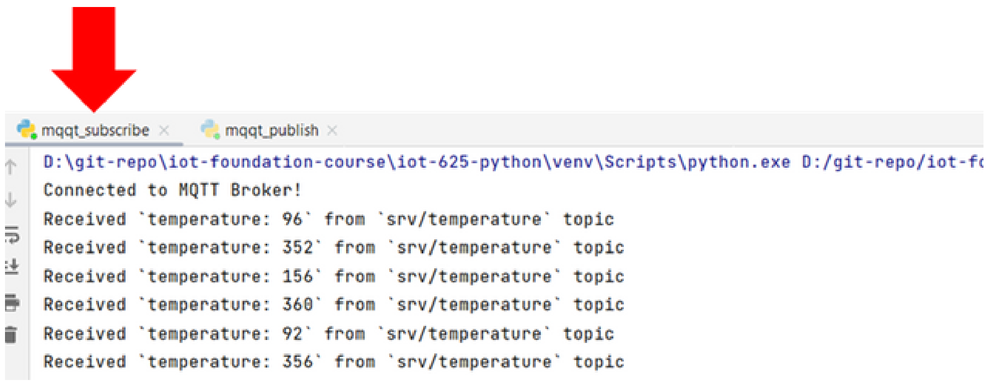
```
root@iot-linux:/etc/mosquitto# mosquitto -c mosquitto_central.conf -v
1665079664: mosquitto version 1.4.8 (build date Tue, 18 Jun 2019 11:59:34 -0300) starting
1665079664: Config loaded from mosquitto_central.conf.
1665079664: Opening ipv4 listen socket on port 1884.
1665079664: Opening ipv6 listen socket on port 1884.
20.214.254.96^C1665079697: mosquitto version 1.4.8 terminating
1665079697: Saving in-memory database to /var/lib/mosquitto/mosquitto.db.
root@iot-linux:/etc/mosquitto# mosquitto -c mosquitto_central.conf -v
1665079739: mosquitto version 1.4.8 (build date Tue, 18 Jun 2019 11:59:34 -0300) starting
1665079739: Config loaded from mosquitto_central.conf.
1665079739: Opening ipv4 listen socket on port 1884.
1665079739: Opening ipv6 listen socket on port 1884.
```

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

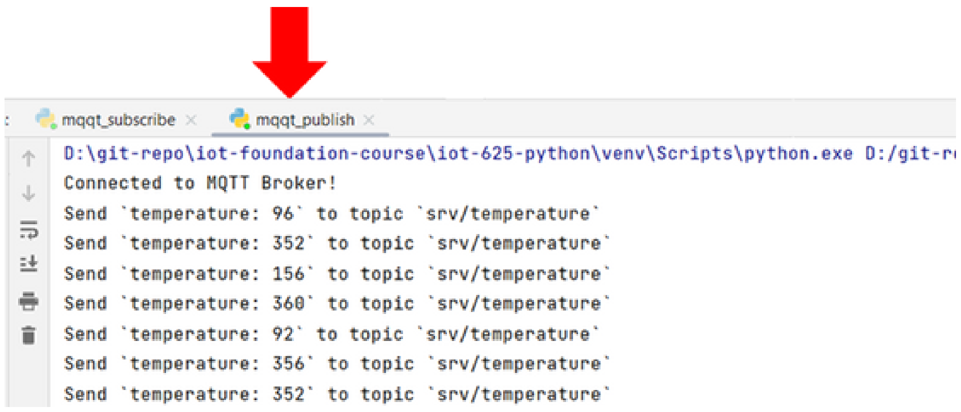


```
root@iot-linux:/etc/mosquitto# mosquitto -c mosquitto_local.conf -v
1665079742: mosquitto version 1.4.8 (build date Tue, 18 Jun 2019 11:59:34 -0300) starting
1665079742: Config loaded from mosquitto_local.conf.
1665079742: Opening ipv4 listen socket on port 1883.
1665079742: Opening ipv6 listen socket on port 1883.
1665079742: Bridge local.iot-linux.local_to_remote doing local SUBSCRIBE on topic #
1665079742: Bridge local.iot-linux.local_to_remote doing local SUBSCRIBE on topic #
1665079742: Connecting bridge local_to_remote (20.249.102.137:1884)
1665079742: Bridge iot-linux.local_to_remote sending CONNECT
```

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



```
D:\git-repo\iot-foundation-course\iot-625-python\venv\Scripts\python.exe D:/git-repo/iot-fo
Connected to MQTT Broker!
Received `temperature: 96` from `srv/temperature` topic
Received `temperature: 352` from `srv/temperature` topic
Received `temperature: 156` from `srv/temperature` topic
Received `temperature: 360` from `srv/temperature` topic
Received `temperature: 92` from `srv/temperature` topic
Received `temperature: 356` from `srv/temperature` topic
```



```
D:\git-repo\iot-foundation-course\iot-625-python\venv\Scripts\python.exe D:/git-re
Connected to MQTT Broker!
Send `temperature: 96` to topic `srv/temperature`
Send `temperature: 352` to topic `srv/temperature`
Send `temperature: 156` to topic `srv/temperature`
Send `temperature: 360` to topic `srv/temperature`
Send `temperature: 92` to topic `srv/temperature`
Send `temperature: 356` to topic `srv/temperature`
Send `temperature: 352` to topic `srv/temperature`
```

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