Ejercicio

FUNDAMENTOS DE INTERNET DE LA COSAS

2024-2025

Universidad Carlos III de Madrid

# Cuaderno 2- Ejercicio - 8

# 2024/2025

MQTT - MOSQUITTO - ENVIO DE TELEMETRIAS

# Cuaderno 2 - Ejercicio 8

---

Universidad Carlos III de Madrid. Escuela Politécnica Superior

*Javier García Guzmán*

Sección 1

## Objetivos

El propósito de este ejercicio consiste en:

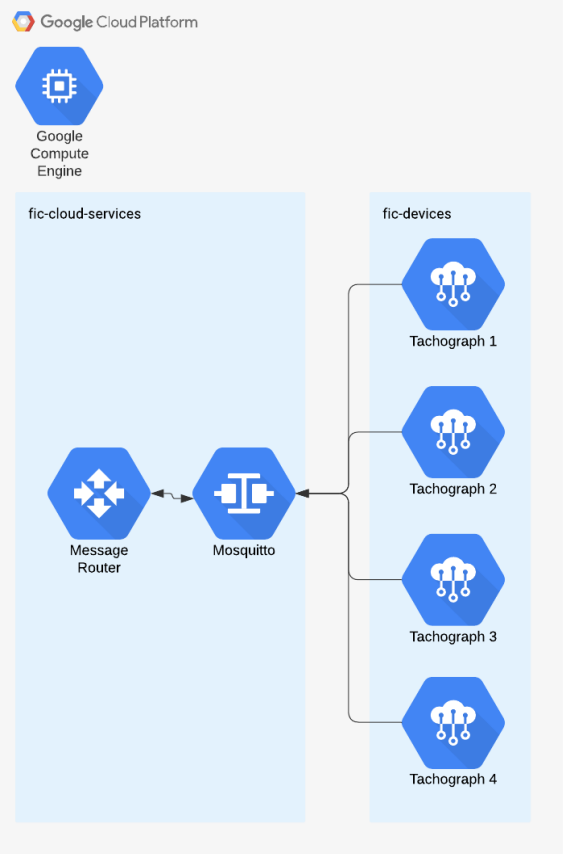
- Saber instalar un broker de mensajes MQTT (i.e. Mosquitto) utilizando contenedores.
- Poner en práctica los conceptos básicos para la conexión a un servicio de MQTT.
- Poner en práctica los conceptos de publicación y subscripción que se manejan en el protocolo MQTT.
- Afianzar los conceptos de coreografía de servicios basados en docker compose.

Sección 2

# Introducción y pasos iniciales

## Arquitectura y organización del proyecto

En el ámbito de este proyecto se partirá de la solución obtenida en la sesión 7. La arquitectura del proyecto es la siguiente:



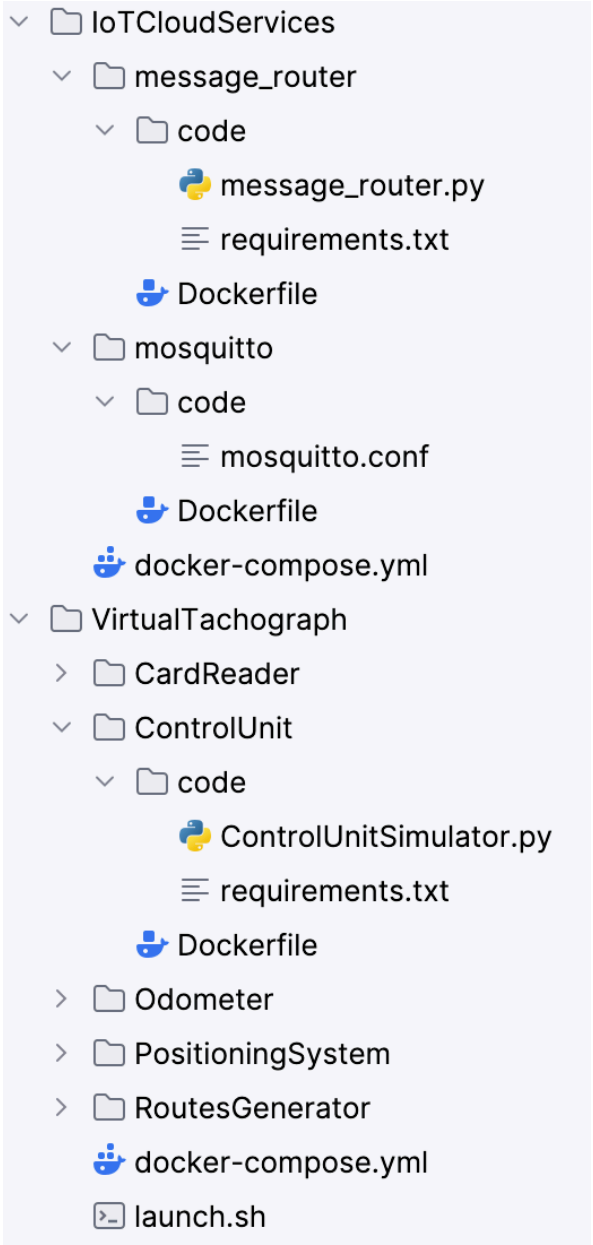

En primer lugar, se desplegará, en una máquina virtual que se denominará *fic-cloud-services*, un contenedor con la implementación de un broker MQTT, en este caso, Mosquitto.

En esa misma máquina, se desplegará otro contenedor, con el primer elemento que se desarrollará de la infraestructura cloud de la solución IoT que se está construyendo en los

cuadernos de la asignatura. Este componente tiene como propósito conectarse al Broker MQTT, suscribirse a los canales por los que los tacógrafos envían telemetría y recibir los mensajes con telemetría y almacenarlos localmente.

Por último, los gemelos digitales de un tacógrafo se desplegarán en contenedores independientes. Estos contendores se ejecutarán en una máquina virtual que se denominará *fic-devices*. Se modificarán los gemelos digitales desarrollados en la sesión 7 para que se puedan conectar Broker MQTT, publicar sus telemetrías a través de los canales configurados para ello y desconectarse del broker una vez que el vehículo se haya detenido.

La organización del código y los ficheros del proyecto será la siguiente:



El código se estructura en dos carpetas: *IoTCloudServices* su código se ejecutará en la máquina *fic-cloud-services*) y *VirtualTachograph* (su código se ejecutará en la máquina *fic-devices*)

- En *IoTCloudServices* habrá dos carpetas, una denominada *mosquitto* que contendrá el Dockerfile y los ficheros de configuración necesarios para el despliegue de esta solución MQTT de código libre en un contenedor.

  La segunda carpeta se denominará *message_router* y contendrá el código fuente en Python para la implementación de este servicio (este código se colocará en la subcarpeta *code* donde también existirá un fichero requirements.txt con las dependencias necesarias para el funcionamiento del código desarrollado). Además, esta carpeta tiene el fichero Dockerfile para el despliegue mediante un contenedor del componente Message Router.

  Por último, en la carpeta *IoTCloudServices* se incluirá el fichero *docker-compose.yml* con la orquestación de los contenedores que se despliegan en la máquina *fic-cloud-services*.

- En *VirtualTachograph* se encontrarán todos los ficheros provenientes de la sesión 07 y contendrá el código fuente en Python para la implementación del gemelo digital del tacógrafo. Para comenzar este ejercicio, se copiarán los ficheros que se obtuvieron como solución para la sesión 07.

## Preparación del proyecto

En primer lugar, es necesario clonar el proyecto en gitlab (https://teaching.sel.inf.uc3m.es) correspondiente a la sesión 8.

## Creación de la máquina virtual para los serivios cloud: *fic-cloud-services*

1. En la consola de Google Cloud, ve a la página **Instancias de VM**.
2. Selecciona el proyecto y haz clic en **Continuar**.
3. Haz clic en **Crear instancia**.
4. Especifica un **Nombre** para la VM. Se recomienda poner el siguiente nombre: fic-cloud-services.
5. Opcional: Cambia la **Zona** para esta VM. Compute Engine aleatoriza la lista de zonas dentro de cada región para fomentar el uso en varias zonas.
6. Selecciona una **Configuración de máquina** para la VM.
7. En la sección **Disco de arranque**, haz clic en **Cambiar** y, luego, haz lo siguiente:
   a) En la pestaña **Imágenes públicas**, elige lo siguiente:

- Sistema operativo (Seleccionar el SO que viene por defecto: Debian)
- Versión del SO
- Tipo de disco de arranque
- Tamaño de disco de arranque

b) Opcional: Para ver las opciones de configuración avanzadas, haz clic en Mostrar configuración avanzada.

c) Para confirmar las opciones del disco de arranque, haz clic en Seleccionar.

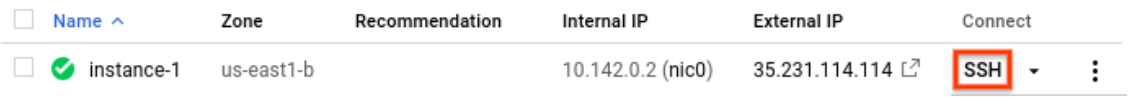
8. Para crear y, también, iniciar la VM, haz clic en **Crear**.

## Instalación de Docker Engine en *fic-cloud-services*

Una vez que hemos completado el paso anterior, es necesario que nos conectemos por SSH a la máquina en la GCP que se ha creado en la sección 2 de este ejercicio.

Para conectarte a las VMs mediante SSH en el navegador desde la consola de Google Cloud, haz lo siguiente:

1. En la consola de Google Cloud, ve a la página **Instancias de VM**.
2. En la lista de instancias de máquinas virtuales, haz clic en **SSH** en la fila de la instancia a la que deseas conectarte.




Como Docker Engine no está en la distribución por defecto del Debian que se ha utilizado para la creación de la máquina virtual, es necesario instalarlo completando los siguientes pasos:

https://docs.docker.com/engine/install/debian/

Como último paso, ejecutar el siguiente comando que permitirá que no sea necesario utilizar sudo para la ejecución de docker engine:

```
sudo usermod -aG docker $USER
```

Posteriormente, cerrar el terminal SSH y volverlo a abrir para que los cambios tengan efecto.

Asimismo, para este y futuros ejercicios, será necesario instalar docker compose. Para ello, es necesario ejecutar los siguientes pasos (consultar la opción *Install the plugin manually*):

https://docs.docker.com/compose/install/linux/

Una vez completado este paso, es necesario clonar el proyecto en gitlab (https://teaching.sel.inf.uc3m.es) correspondiente a la sesión 7.

Sección 3

# Creación de la imagen y despliegue del contenedor del Mosquitto

## Creación de la imagen con Dockerfile

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a `debian:buster`.
- Ejecutar `apt update && apt upgrade -y`
- Ejecutar `apt install mosquitto mosquitto-clients -y`
- Ejecutar `touch /etc/mosquitto/passwd`
- Ejecutar `mosquitto_passwd -b /etc/mosquitto/passwd fic_server fic_password`
- Lanzar mosquitto ejecutando el siguiente comando: `/usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf`

Si se necesita soporte acerca de los comandos para implementar los pasos anteriores en un Dockerfile, utilizar la hoja de resumen de comandos que se incluye en la sección 4 del enunciado del ejercicio 7.

Además, en la carpeta *IoTCloudServices/mosquitto/code* es necesario crear un fichero denominado *mosquitto.conf* con las opciones iniciales de la instancia de mosquitto.

El contenido de este fichero es el siguiente:

```
allow_anonymous false
password_file /etc/mosquitto/passwd
```

## Configuración de la orquestación del servicio Mosquitto

Para desplegar la imagen de mosquitto en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cloud-services*.

Para ello, en la carpeta *IoTCloudServices* se creará un fichero denominado *docker-compose.yml*. El contenido inicial de este fichero es el siguiente:

```yaml
version: '3'

services:
  mosquitto:
    build: ./mosquitto
    ports:
      - "1883:1883"
    volumes:
      - "./code/mosquitto.conf:/etc/mosquitto/mosquitto.conf"
```

Este contenido indica que se va a utilizar la versión 3 de la especificación de docker compose.

Esta orquestación contiene servicios (por ahora, solo uno).

El que se está configurando actualmente se denomina mosquitto y va a construir un contendor a partir del Dockerfile que se encuentra en la carpeta *IoTCloudServices/mosquitto*.

Este servicio publicará el puerto 1883 del contenedor a través del puerto 1883 de la máquina que lo alberga (máquina host).

Un resumen de los principales elementos que se pueden utilizar en docker compose se incluyen en la imágen que se muestra en la página siguiente.

Por último, este contenedor tendrá un volumen que permitirá copiar el fichero *./code/mosquitto.conf* de la máquina local sobre el fichero */etc/mosquitto/mosquitto.conf* del contenedor.

En este momento, se recomienda hacer *commit* y *push* del código del proyecto.

# DOCKER COMPOSE CHEAT SHEET

## File

### structure

```
services:
   container1:
      properties: values

   container2:
      properties: values

networks:
   network:

volumes:
   volume:
```

## Types

### value

```
key: value
```

### array

```
key:
   - value
   - value
```

### dictionary

```
master:
   key: value
   key: value
```

## Properties

### build

build image from dockerfile in specified directory

```
container:
   build: ./path
   image: image-name
```

### image

use specified image

```
image: image-name
```

### container_name

define container name to access it later

```
container_name: name
```

### volumes

define container volumes to persist data

```
volumes:
   - /path:/path
```

### command

override start command for the container

```
command: execute
```

### environment

define env variables for the container

```
environment:
   KEY: VALUE
---
environment:
   - KEY=VALUE
```

### env_file

define a env file for the container to set and override env variables

```
env_file: .env
---
env_file:
 - .env
```

### restart

define restart rule (no, always, on-failure, unless-stopped)

```
expose:
    - "9999"
```

### networks

define all networks for the container

```
networks:
   - network-name
```

### ports

define ports to expose to other containers and host

```
ports:
    - "9999:9999"
```

### expose

define ports to expose only to other containers

```
expose:
    - "9999"
```

### network_mode

define network driver (bridge, host, none, etc.)

```
network_mode: host
```

### depends_on

define build, start and stop order of container

```
depends_on:
    - container-name
```

## Other

### idle container

send container to idle state > container will not stop

```
command: tail -f /dev/null
```

### named volumes

create volumes that can be used in the volumes property

```
services:
  container:
    image: image-name
    volumes:
      - data-
volume:/path/to/dir

volumes:
  data-volume:
```

### networks

create networks that can be used in the networks property

```
networks:
   frontend:
      driver: bridge
```

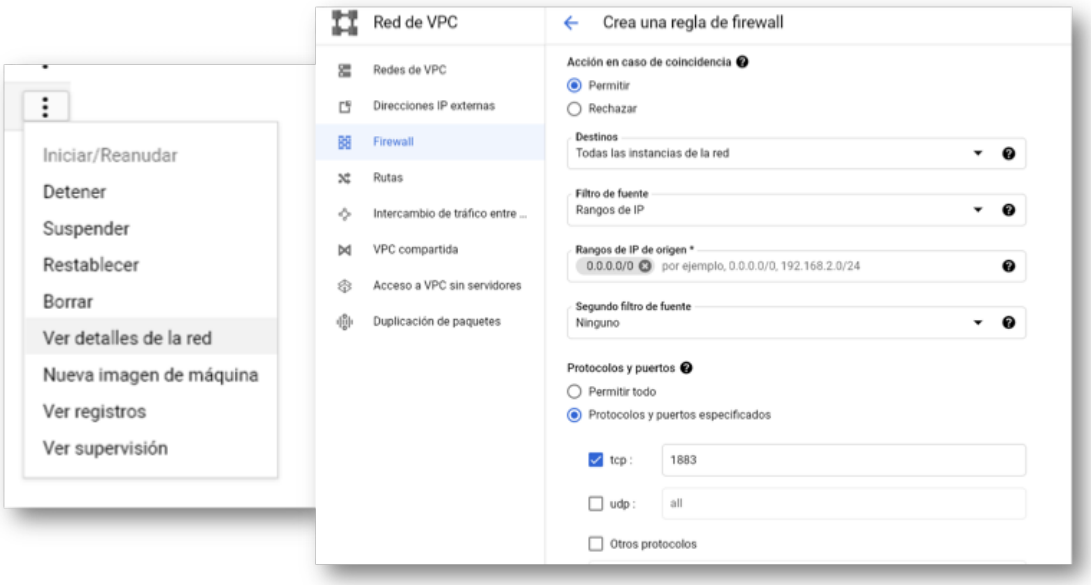
## Configurar el firewall de la máquina fic-cloud-services

Para que el servicio MQTT esté accesible en el puerto 1883 es necesario habilitarlo en el firewall disponible en la GCP para las máquinas virtuales del proyecto de la asignatura.

Para hacerlo, tendréis que ir Firewall y luego seleccionad Crear nueva regla de cortafuegos que se denominará: *allow-mqtt*.

La configuración utilizada para esta regla es la que se muestra en la imagen:

- Habilitar todas las IP en la red, para que pueda acceder a ese puerto desde cualquier IP
- Habilitar el puerto TCP 1883 que es el que usa el MQTT por defecto




## Despliegue del servicio Mosquitto

Para desplegar la imagen de mosquitto en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cloud-services*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-cloud-services* creada en la sección 2 de este enunciado.

A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondinte a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar que el mosquitto se encuentra en ejecución, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Para probar el correcto funcionamiento del mosquitto desplegado, se recomienda ejecutar los siguientes comandos (desde el terminal ssh la máquina virtual fic-cloud-services o desde el terminal de cualquier otra máquina que tenga conexión a Internet).

```
sudo apt-get install mosquitto-clients

mosquitto_sub -h <dirección_ip_de_fic-cloud-services> -t
"fic/tachographs/1234BBC/telemetry" -v

mosquitto_pub -h <dirección_ip_de_fic-cloud-services> -t
"fic/tachographs/1234BBC/telemetry " -m "ON"
```

Sección 4

# Desarrollo del servicio Message Router

Para desarrollar el servicio denominado Message Router será necesario, en primer lugar, desarrollar y probar el código de su funcionamiento y, posteriormente, editar el fichero *Dockerfile* y *docker-compose.yml* que permitirá su despliegue y orquestación con el servicio de Mosquitto previamente desplegado y configurado.

## Desarrollo y prueba local del código de Message Router

En la carpeta *IoTCloudServices/message_router/code* se tiene que crear un fichero de código Python que se denomine *message_router.py*.

Los elementos de código que se deben incluir en este fichero son los siguientes:

1. Se tiene que importar la librería de paho-mqtt para poder utilizar las funciones de Python que implementan el protocolo mqtt.

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

2. El método main debe:
   - Contener las sentencias necesarias para establecer el usuario y la contraseña que se va a utilizar para conectarse al mosquitto.

```
client.username_pw_set(username="fic_server",
password="fic_password")
```

   - Establecer cuáles son los métodos de callback para los eventos on_connect y on_message.

```
client.on_connect = on_connect
client.on_message = on_message
```

   - Conectar al broker teniendo en cuenta que MQTT_SERVER y MQTT_PORT tendrán el valor de variables de entorno.

```
MQTT_SERVER = os.getenv("MQTT_SERVER_ADDRESS")
MQTT_PORT = int(os.getenv("MQTT_SERVER_PORT"))
client.connect(MQTT_SERVER, MQTT_PORT, 60)
```

- Configurar este componente para que constantemente esté recibiendo comunicaciones del Mosquitto.

```
client.loop_forever()
```

3. El método *on_connect* debe:

- Definirse de la siguiente manera:

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

- Imprimir por pantalla el mensaje recibido desde el broker.
- En caso de que el código recibido (rc) sea igual a 0, el message router se debe suscribir a los canales de telemetría del vehículo:

```
INIT_TOPIC = "/fic/tachographs/+/request_access/"
client.subscribe(INIT_TOPIC)
print("Subscribed to ", INIT_TOPIC)
```

4. Con respecto al método *on_message* se debe considerar lo siguiente:

- Se debe generar una variable global

```
connected_tachographs = []
```

- Se debe generar otra variable global

```
authorised_tachogrpaphs = ["tachograph_control_unit-1",
                           "tachograph_control_unit-2",
                           "tachograph_control_unit-3",
                           "tachograph_control_unit-4"]
```

- El método on_message debe definirse de la siguiente manera:

```
def on_message(client, userdata, msg):
```

El funcionamiento esperado de este método debe incluir lo siguiente:

- Imprimir por pantalla el mensaje recibido desde el broker.
- Se comprueba el topic del mensaje que se ha recibido incluye *request_access*.

```
topic = (msg.topic).split('/')
if "request_access" in msg.topic:
```

  - Se obtiene el mensaje recibido por el message router

```
str_received_request_access = msg.payload.decode()
received_request_access =
json.loads(str_received_request_access)
```

  - Se comprueba que el id que ha proporcionado el tacógrafo se encuentra entre los preautorizados:

```
if
find_authorised_tachograph(received_request_access["Tachograp
h_id"]):
```

La función find_authorised_tachograph que realiza la búsqueda, obtiene el tachograph_id de la petición y establece si está entre los ítems incluidos en el array authorised_tachogrpaphs.

- En caso de que el tachograph_id esté preautorizado, se comprueba que no se ha conectado ya:

```
found, position =
is_connected(received_request_access["Tachograph_id"])
if not found:
```

La función is_connected busca en el array de los tacógrafos conectados si el que está solicitando su conexión ya está previamente conectado.

- En caso de que el tacógrafo ya esté conectado anteriormente, se rechazará una nueva conexión (no se permite la conexión de dos tacógrafos con el mismo id) y se informa al tacógrafo del rechazo de la conexión:

```
request_access_response = {"Tachograph_id":
received_request_access["Tachograph_id"],
                           "Authorization": "False",
                           "Timestamp":
datetime.datetime.timestamp(datetime.datetime.now()) * 1000}

client.publish("/fic/tachographs/" + topic[-3] + "/config/",
               payload=json.dumps(request_access_response),qos=1,
                   retain=False)
```

- En caso de que el tacógrafo NO esté conectado anteriormente, se aceptará la nueva conexión, se registra el tacógrafo como conectado, se informa al tacógrafo de la aceptación de la conexión y se realiza la suscripción al canal de telemetría por el que el tacógrafo enviará la información:

```
register_tachograph_connection(received_request_access["Tac
hograph_id"], topic[-3])
request_access_response = {"Tachograph_id":
received_request_access["Tachograph_id"],
              "Authorization": "True",
              "Timestamp":
datetime.datetime.timestamp(datetime.datetime.now()) *
1000}
client.publish("/fic/tachographs/" + topic[-3] + "/config/",
               payload=json.dumps(request_access_response),qos=1,
                   retain=False)
telemetry_topic = "/fic/tachographs/" + topic[-3]+
"/telemetry/"
client.subscribe(telemetry_topic)
print("Subscribed to ", telemetry_topic)
```

- En caso de que el tachograph_id NO esté preautorizado, se informa al tacógrafo que la conexión ha sido denegada:

```
request_access_response = {"Tachograph_id":
received_request_access["Tachograph_id"],
                           "Authorization": "False",
```

```
                                "Timestamp":
datetime.datetime.timestamp(datetime.datetime.now()) * 1000}

client.publish("/fic/tachographs/" + topic[-3] + "/config/",
               payload=json.dumps(request_access_response),qos=1,
                   retain=False)
```

- Se comprueba el topic del mensaje que se ha recibido incluye telemetry.

```
topic = (msg.topic).split('/')
if topic[-3] == "telemetry":
```

  - En este caso, se actualiza un array con la nueva telemetría recibida y se imprime por pantalla la información relativa a esta nueva telemetría recibida.

- Se comprueba el topic del mensaje que se ha recibido incluye events.

```
topic = (msg.topic).split('/')
if topic[-3] == "events":
```

  - En este caso, se actualiza un array con el nuevo evento recibido y se imprime por pantalla la información relativa la información recibida.

- Se comprueba el topic del mensaje que se ha recibido incluye session.

```
topic = (msg.topic).split('/')
if topic[-3] == "session":
```

  - En este caso, se tiene que retirar el tacógrafo de los actualmente conectados y se debe almacenar la información asociada en un fichero que tenga como nombre el id del tacógrafo conectado y la marca de tiempo en el que se realiza la desconexión.

El resumen de las principales funciones de la librería paho para el uso del protocolo mqtt son las siguientes:

1. Client()
2. loop_start()
3. loop_stop()
4. loop_forever(timeout, max_packets, retry_first_connection)
5. publish(topic, payload, qos, retain)
6. on_connect(client, userdata, flags, rc)
7. connect(host, port, keepalive, bind_address)
8. on_message(client, userdata, message)
9. subscribe(topic, qos)

## Creación de la imagen con Dockerfile

Para este propósito, en la carpeta *IoTCloudServices/message_router* se creará un fichero Dockerfile.

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.12
- Copiar el código que está en la carpeta *./code* a la carpeta */etc/usr/src/app*
- Establecer esta carpeta como directorio de trabajo.
- Instalar los paquetes necesarios especificados en requirements.txt. Los paquetes necesarios son: *requests* y *math,* para ello ejecutar pip install <paquetes necesarios>
- Ejecutar el código incluido en el fichero *message_router.py*

Las dependencias que se deben incluir en el fichero requirements.txt son: paho-mqtt

## Orquestación de los servicios con Docker Compose

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml*. El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio message_router de acuerdo con el siguiente código:

```
message_router:
  build: ./message_router
  environment:
    - MQTT_SERVER_ADDRESS=mosquitto
    - MQTT_SERVER_PORT=1883
    - PYTHONUNBUFFERED=1
  volumes:
    - "./message_router/code:/etc/usr/src/app"
  depends_on:
    - mosquitto
```

## Despliegue de la imagen con Docker Compose

Para desplegar el componente message_router es necesario conectarse por *ssh* a la máquina *fic-cloud-services* creada en la sección 2 de este enunciado.

A continuación, se debe obtener la última versión del código de esta sesión que se encuentra en el repositorio en esta máquina virtual mediante un comando *git pull*.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose stop

docker compose build

docker compose up -d
```

Para verificar que el Message Router se encuentra en ejecución, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor del message router>
```

Sección 5

# Modificación del gemelo digital del simulador del vehículo para que sea capaz de enviar telemetrías

Para incluir las capacidades de envío de telemetría en el gemelo digital del simulador del vehículo será necesario, en primer lugar, desarrollar y probar el código de su funcionamiento y, posteriormente, será necesario realizar modificaciones en relación con el fichero *Dockerfile* y se tendrá que actualizar la coreografía de servicios para la máquina *fic-devices* modificando el fichero *docker-compose.yml* que permitirá su despliegue.

## Actualización y prueba local del código del gemelo digital del simulador del vehículo

En la carpeta *VirtualTachograph/ControlUnit/code* se tiene que actualizar el fichero de código Python que se denomine *ControlUnitSimulator.py*.

Las modificaciones a incluir son las siguientes:

1. Se tiene que importar la librería de paho-mqtt para poder utilizar las funciones de Python que implementan el protocolo mqtt.

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

2. Se tiene que añadir un Daemon Thread independiente que gestione las comunicaciones.

```
t1 = threading.Thread(target=mqtt_communications, daemon=True)
t1.start()
t1.join()
```

3. En el método que implementa las comunicaciones debe implementar las siguientes funcionalidades:
   - Contener las sentencias necesarias para establecer el usuario y la contraseña que se va a utilizar para conectarse al mosquitto.

```
client.username_pw_set(username="fic_server",
password="fic_password")
```

   - Establecer cuáles son los métodos de callback para los eventos on_connect y on_message.

```
client.on_connect = on_connect
client.on_message = on_message
```

- Se debe generar una variable global con el ID del tacógrafo. Por ahora, este ID se generará aleatoriamente cuando la unidad de control comience su ejecución de acuerdo con el siguiente código:

```
tachograph_id = "tachograph_control_unit-" +
str(random.randint(1, 5))
```

- Configurar un mensaje que sea recibido por todos los suscriptores en caso de que se produzca una desconexión irregular por parte del vehículo.

```
SESSION_TOPIC = "/fic/tachographs/" + get_host_name() +
"/session/"
connection_dict = {"Tachograph_id": tachograph_id, "Status":
"Off - Unregulate Disconnection",
                   "Timestamp":
datetime.datetime.timestamp(datetime.datetime.now()) * 1000}
connection_str = json.dumps(connection_dict)
client.will_set(SESSION_TOPIC, connection_str)
```

- Conectar al broker teniendo en cuenta que MQTT_SERVER y MQTT_PORT tendrán el valor de variables de entorno.

```
MQTT_SERVER = os.getenv("MQTT_SERVER_ADDRESS")
MQTT_PORT = int(os.getenv("MQTT_SERVER_PORT"))
client.connect(MQTT_SERVER, MQTT_PORT, 60)
```

- Configurar este componente para que constantemente esté recibiendo comunicaciones del Mosquitto.

```
client.loop_forever()
```

5. El método *on_connect* debe:

- Definirse de la siguiente manera:

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

- Imprimir por pantalla el mensaje recibido desde el broker.
- En caso de que el código recibido (rc) sea igual a 0, el message router se debe suscribir a los canales de telemetría del vehículo:

```
REQUEST_ACCESS_TOPIC = "/fic/tachographs/" + get_host_name() +
"/request_access/"
request_access_message = {"Tachograph_id": tachograph_id,
                "Timestamp":
datetime.datetime.timestamp(datetime.datetime.now()) * 1000}
client.publish(REQUEST_ACCESS_TOPIC,
payload=json.dumps(request_access_message), qos=1,
                retain=False)
CONFIG_TOPIC = "/fic/tachographs/" + get_host_name() +
"/config/"
client.subscribe(CONFIG_TOPIC)
```

6. Con respecto al método *on_message* se debe considerar lo siguiente:

- El método on_message debe definirse de la siguiente manera:

```
def on_message(client, userdata, msg):
```

El funcionamiento esperado de este método debe incluir lo siguiente:

- Imprimir por pantalla el mensaje recibido desde el broker.
- Se comprueba el topic del mensaje que se ha recibido incluye *config*.

```
topic = (msg.topic).split('/')
if "config" in msg.topic:
```

  - En este caso se procede a verificar si el acceso a la red de tacógrafos ha sido concedido

```
if json_config_received["Tachograph_id"] == tachograph_id and
json_config_received["Authorization"] == "True":
```

    - En caso afirmativo, se actualiza el valor la variable que informa de esta situación a verdadero.

```
connection_granted = True
```

    - En caso negativo, se procede a la desconexión de la unidad de control de la red y, potencialmente, al apagado del sistema de tacógrafo.

7. En el método que gestiona el hilo para la ejecución de comunicaciones mqtt, se tiene que publicar la telemetría del tacógrafo. A continuación, se muestra un posible ejemplo de la estructura de cómo se debería realizar la publicación de la telemetría:

```
while not disconnected:
    if connection_granted:
        publish_telemetry(client)
        publish_events(client)
    else:
        time.sleep(10)
```

La publicación de los mensajes de telemetría debe realizarse de una manera parecida al siguiente ejemplo:

```
STATE_TOPIC = "/fic/tachographs/" + get_host_name() + "/telemetry/"
while number_telemetries_sent < len(logs):
    client.publish(STATE_TOPIC, payload=json.dumps(logs[number_telemetries_sent]),
                   qos=1, retain=False)
    number_telemetries_sent += 1
```

## Actualización de la imagen con Dockerfile

Las dependencias que se deben incluir en el fichero requirements.txt son: paho-mqtt

## Configuración de la orquestación del servicio Mosquitto

Para desplegar la imagen de mosquitto en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cloud-services*.

Para ello, en la carpeta *IoTCloudServices* se creará un fichero denominado *docker-compose.yml*. El contenido inicial de este fichero es el siguiente:

```
services:
  tachograph_control_unit:
    build: ./ControlUnit
    image: tachograph_control_unit
    environment:
      - MQTT_SERVER_ADDRESS=<dirección ip de la vm fic-cloud-services>
      - MQTT_SERVER_PORT=1883
      - PYTHONUNBUFFERED=1
      - PORT=65431

    volumes:
      - ./VirtualVehicles/code:/etc/usr/src/code
```

Este contenido indica que se va a utilizar la versión 3 de la especificación de docker compose.

Esta orquestación contiene servicios (por ahora, solo uno).

El que se está configurando actualmente se denomina mosquitto y va a construir un contendor a partir del Dockerfile que se encuentra en la carpeta *IoTCloudServices/mosquitto*.

Este servicio publicará el puerto 1883 del contenedor a través del puerto 1883 de la máquina que lo alberga (máquina host).

## Despliegue del servicio Mosquitto

Para desplegar las versiones actualizadas del gemelo digital del simulador de vehículos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-devices*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-devices* creada en la sección 2 de este enunciado. A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *VirtualVehicles* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar el despliegue, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Sección 6

# Criterios de evaluación y procedimiento de entrega

## Criterios de Evaluación

- Configuración correcta de Mosquitto – 2 puntos
- Message Router – Conexión y recepción de telemetrías – 4 puntos
- Gemelo Digital – Conexión, envío de telemetrías y desconexión – 4 puntos


## Entrega de la solución del ejercicio guiado

La entrega se realizará de dos maneras:

1) **Código Fuente**. En el repositorio de código de la asignatura tendrá en el proyecto Sesion08 correspondiente al grupo de prácticas los ficheros de código con la organización en directorios y nomenclatura de los ficheros que se han indicado a lo largo de este guion.
2) **Video demostrativo.** En una tarea Kaltura Capture Video habilitada para este ejercicio se entregará un vídeo que demuestre el correcto funcionamiento de la solución elaborada.

   Para este programa, se debe proporcionar una breve explicación del código generado (código en Python, dockerfile y docker compose) para Mosquitto, Message Router y Gemelo Digital, mostrar el lanzamiento de la ejecución de la solución en la GCP y la visualización de las trazas con los resultados.

**La fecha límite para la entrega del ejercicio es 08/05/2025 a las 23:59 h.**

De acuerdo con las normas de evaluación continua en esta asignatura, si un estudiante no envía la solución de un ejercicio antes de la fecha límite, el ejercicio será evaluado con 0 puntos.


Cada estudiante debe guardar una copia de la solución entregada hasta la publicación de las calificaciones finales de la asignatura.