Ejercicio

FUNDAMENTOS DE INTERNET DE LA COSAS

2023-2024

Universidad Carlos III de Madrid

Cuaderno 2- Ejercicio - 9

2023/2024

MQTT - ENVIO DE COMANDOS

Cuaderno 2 - Ejercicio 9

Universidad Carlos III de Madrid. Escuela Politécnica Superior

Javier García Guzmán

Sección 1

Objetivos

El propósito de este ejercicio consiste en:

- Afianzar los conceptos de publicación y subscripción que se manejan en el protocolo MQTT.
- Afianzar los conceptos de coreografía de servicios basados en docker compose.

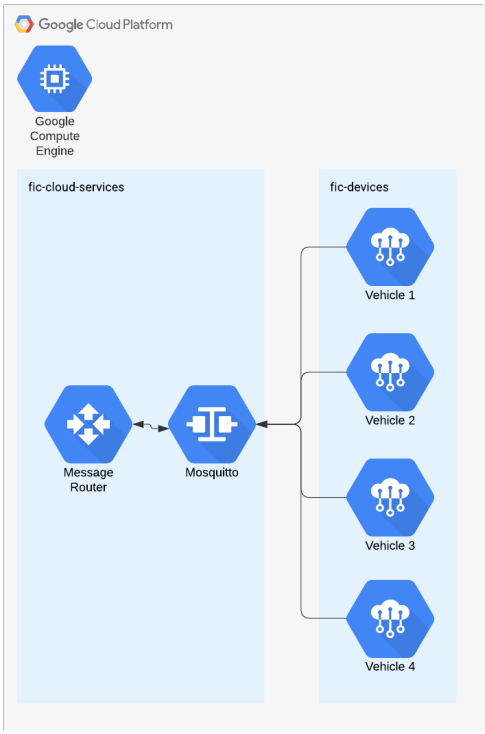
Sección 2

Introducción y pasos iniciales

Arquitectura y organización del proyecto

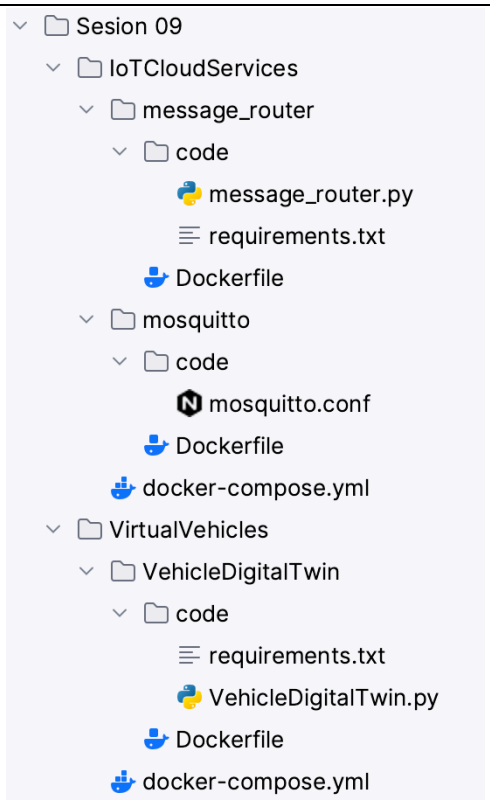
En el ámbito de este proyecto se partirá de la solución obtenida en la sesión 8.

La arquitectura del proyecto es la siguiente:



La arquitectura es la misma que la empleada en el ejercicio 8 porque el propósito de esta sesión consiste en ser capaces de enviar comandos desde el Message Router a los distintos vehículos.

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 *VirtualVehicles* (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 despligue 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 *VirtualVehicles* habrá una carpeta *VehicleDigitalTwin* 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 de cada gemelo digital.

Por último, en la carpeta *VirtualVehicles* se incluirá el fichero *docker-compose.yml* con la orquestación de todos los gemelos digitales a contemplar en la máquina *fic-devices*.

Preparación del proyecto

En primer lugar, es necesario clonar el proyecto en gitlab (https://teaching.sel.inf.uc3m.es) correspondiente a la sesión 9.

Se recomienda que, en este momento, se copien todos los ficheros de código y configuración desarrollados en la sesión 8 a la carpeta en la que se está desarrollando el código de la sesión 9, contemplando la estructura de carpetas presentada en el apartado anterior.

Sección 3

Actualización del servicio Message Router

Para desarrollar el servicio denominado Message Router será necesario, en primer lugar, desarrollar y probar el código de su funcionamiento en el entorno personal de desarrollo del estudiante y, posteriormente, utilizar el fichero *Dockerfile* y *docker-compose.yml* desarrollado en el ejercicio anterior para 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 actualizar el código incluido en el fichero denominado *message_router.py*.

Los elementos de código que se deben incluir en la versión actualizada de Message Router son los siguientes:

1. Cada 60 segundos, el message router le mandará una ruta a uno de los vehículos conectados.
    - Los posibles puntos de origen y destino se configurarán en una variable global en Message Router Serán los siguientes:

```
pois = ["Ayuntamiento de Leganes", "Ayuntamiento de Getafe",
"Ayuntamiento de Alcorcón", "Ayuntamiento de Móstoles",
"Universidad Carlos III de Madrid - Campus de Leganés",
"Universidad Carlos III de Madrid - Campus de Getafe",
"Universidad Carlos III de Madrid - Campus de Puerta de Toledo",
"Universidad Carlos III de Madrid - Campus de Colmenarejo",
"Ayuntamiento de Navalcarnero", "Ayuntamiento de Arroyomolinos",
"Ayuntamiento de Carranque", "Ayuntamiento de Alcalá de Henares",
"Ayuntamiento de Guadarrama", "Ayuntamiento de la Cabrera",
"Ayuntamiento de Aranjuez"]
```

    - Aleatoriamente, se seleccionará de los vehículos conectados aquel al que se enviará la ruta, siempre y cuando ese vehículo no tenga una ruta ya asignada.

- Para generar la ruta, se obtendrá aleatoriamente un origen y un destino de entre los POIs disponibles asegurando que el mismo POI no es seleccionado como origen y destino al mismo tiempo.
- Se generará un texto JSON con la ruta asignada al vehículo.

```
route = {"Origin": origin, "Destination": destination}
```

- Se enviará la ruta al vehículo utilizando como topic el identificado con su matrícula y dedicado exclusivamente al envío de rutas. A contiuación, se muestra un ejemplo de cómo se puede construir ese topic, teniendo en cuenta que connected vehicles es un array que en cada posición contiene los datos de cada vehículo conectado.

```
"/fic/vehicles/" + connected_vehicles[to_vehicle]["Plate"] +
"/routes"
```

La representación abstracta de la estructura de datos connected_vehicles es la siguiente:

0	**id**	**Plate**	**Route**	
	virtual_client_1	0001BBB	Origin	Ayuntamiento de Getafe
			Destination	Ayuntamiento de Móstoles
1	**id**	**Plate**	**Route**	
	virtual_client_2	0002BBB	Origin	None
			Destination	None
2	**id**	**Plate**	**Route**	
	virtual_client_3	0003BBB	Origin	None
			Destination	None
	…			
N-1	**id**	**Plate**	**Route**	
	virtual_client_n	0010BBB	Origin	None
			Destination	None

Un ejemplo de la sentencia de publicación de la ruta es la siguiente:

```
mqtt_broker_client.publish("/fic/vehicles/" +
connected_vehicles[to_vehicle]["Plate"] + "/routes",
payload=json.dumps(route), qos=1, retain=False)
```

- Por último, se actualiza la estructura de datos de connected vehicles para registrar la ruta que se acaba de asignar al vehículo.

```
connected_vehicles[to_vehicle]["Route"]["Origin"] = origin
connected_vehicles[to_vehicle]["Route"]["Destination"] =
destination
```

2. El método *on_message* se tendrá que modificar para que cuando se reciba un evento de ruta completada por parte de un vehículo, se proceda a la retirada de la ruta asignada de la estructura de datos Connected Vehicles

El formato del mensaje JSON que se recibirá es el siguiente:

```
{"Plate": "1234BBC", "Event": "Route Completed", "Timestamp": 1703593978}
```

Creación de la imagen con Dockerfile

El fichero Dockerfile que se utilizará para el Message Router en este ejercicio es igual al utilizado en el ejercicio anterior.

Orquestación de los servicios con Docker Compose

El fichero *docker-compose.yml* que se utilizará para la máquina *fic-cloud-services* en este ejercicio es igual al utilizado en el ejercicio anterior.

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

Actualización del gemelo digital del simulador del vehículo

Para incluir las capacidades de recepción de las rutas asignadas por Message Router y el envío de la información a éste de que la ruta se ha completado 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 *VirtualVehicles/VehicleDigitalTwin/code* se tiene que actualizar el fichero de código Python que se denomine *VehicleDigitalTwin.py*.

Las modificaciones a incluir son las siguientes:

1. Asegurar que el vehículo no asigna ninguna ruta cuando comienza su ejecución
2. Cuando el gemelo digital se ha conectado al mosquitto, suscribirse al topic por el cual se reciben las rutas asignadas.

```
ROUTES_TOPIC = "/fic/vehicles/" + get_host_name() + "/routes/"
```

3. Cuando el gemelo digital reciba un mensaje por el topic de las rutas, realizar la asignación de la misma al vehículo.

```
required_route = msg.payload.decode()
# print("Assigning a route to the vehicle number: {}".format(),
required_route)
routes_loader(required_route)
```

4. Cuando el gemelo digital finalice una ruta, mandar el evento correspondiente al Message Router.
   - Crear una variable global para registrar un evento

```
event_message = ""
```

- En el método *vehicle_controller*, cuando se haya finalizado la ejecución de la ruta, generar un nuevo mensaje de evento

```
event_message = "Route Completed"
```

- En el método *mqtt_communications*, en el loop general, en caso de que haya un mensaje de evento, realizar su publicación.

```
if event_message is not "":
    publish_event(client)
    event_message = ""
```

A continuación, se muestra un ejemplo de las instrucciones para enviar el evento

```
event_to_send = {"Plate": vehicle_plate, "Event":event_message,
"Timestamp":datetime.timestamp(datetime.now())}
client.publish(EVENTS_TOPIC, payload=event_to_send, qos=1,
retain=False)
```

Actualización de la imagen con Dockerfile

El fichero Dockerfile que se utilizará para el Vehicle Digital Twin en este ejercicio es igual al utilizado en el ejercicio anterior.

Configuración de la orquestación del servicio Mosquitto

El fichero *docker-compose.yml* que se utilizará para la máquina *fic-devices* en este ejercicio es igual al utilizado en el ejercicio anterior.

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

- Message Router – Envío de rutas – 4,5 puntos
- Message Router – Recepción de evento de finalización de la ruta – 1,5 puntos
- Gemelo Digital – Recepción y asignación de la ruta – 3 puntos
- Gemelo Digital – Evento de finalización de la ruta – 1 punto

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 Sesion09 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 09/05/2024 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.