



FUNDAMENTOS DE INTERNET DE LA COSAS

2023-2024

Universidad Carlos III de Madrid

Cuaderno 2- Ejercicio - 11

2023/2024

MICROSERVICIOS PARA IOT - ORDENAR RUTAS

Cuaderno 2 - Ejercicio 11

Universidad Carlos III de Madrid. Escuela Politécnica Superior

Objetivos

El propósito de este ejercicio consiste en:

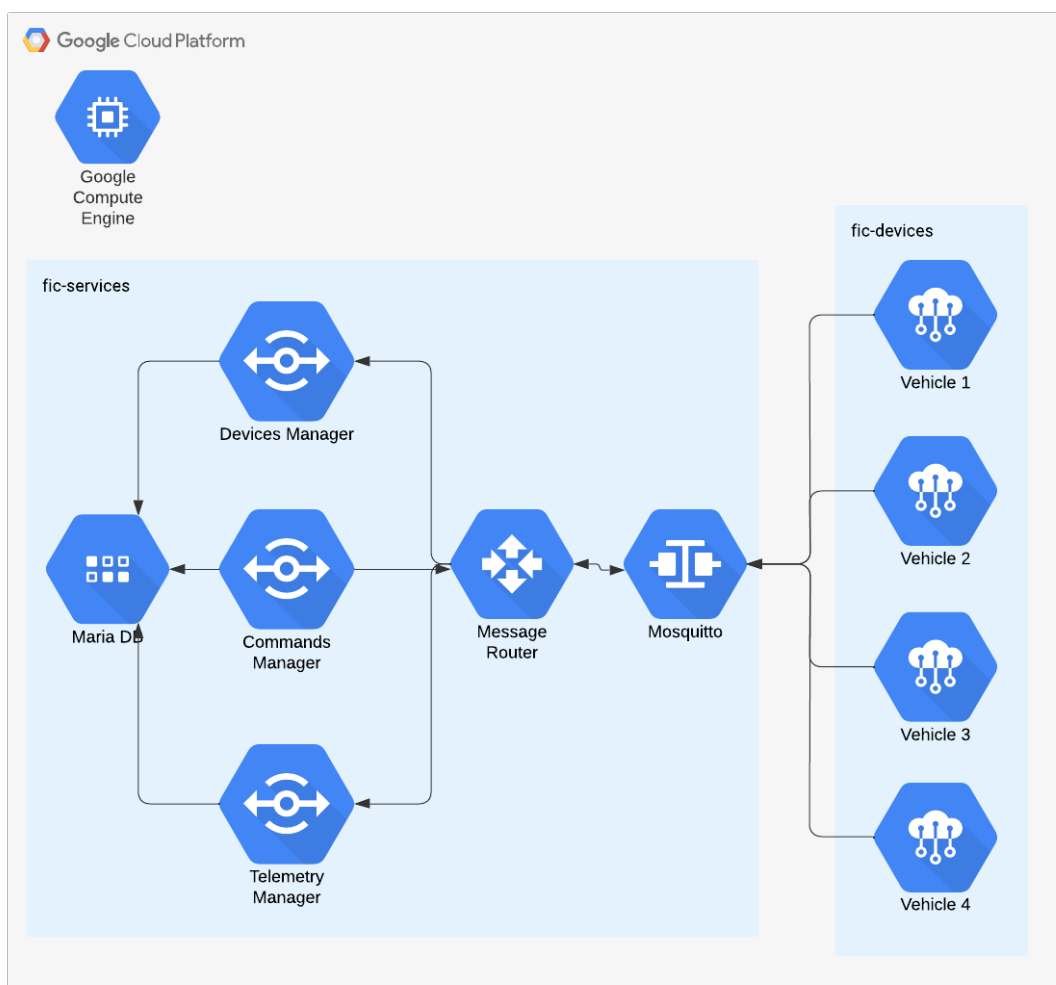
- Poner en práctica los conceptos necesarios para desarrollar microservicios que implementen los componentes de nube en IoT.
- Modificar el componente de Message Router para que funcione completamente como microservicio, incluyendo una interfaz API REST.

Introducción y pasos iniciales












































En el ámbito de este proyecto se partirá de la solución obtenida en la sesión 10.

Arquitectura y organización del proyecto

La arquitectura del proyecto es la siguiente:



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

- ✓  Sesion 11
 - ✓  IoTCloudServices
 - ✓  dbService
 -  Dockerfile
 -  initial_script.sql
 - ✓  message_router
 - ✓  code
 -  message_router.py
 -  requirements.txt
 -  telemetry_register_interface.py
 -  vehicle_register_interface.py
 -  Dockerfile
 - ✓  microservices
 - ✓  routes_microservice
 - ✓  code
 -  requirements.txt
 -  routes_db_manager.py
 -  routes_manager_api.py
 -  Dockerfile
 - ✓  telemetry_microservice
 - ✓  code
 -  requirements.txt
 -  telemetry_db_manager.py
 -  telemetry_manager_api.py
 -  Dockerfile
 - ✓  vehicles_microservice
 - ✓  code
 -  requirements.txt
 -  vehicles_db_manager.py
 -  vehicles_manager_api.py
 -  Dockerfile
 - ✓  mosquitto
 - ✓  code
 -  mosquitto.conf
 -  Dockerfile
 -  docker-compose.yml
 - ✓  VirtualVehicles
 - ✓  VehicleDigitalTwin
 - ✓  code
 -  requirements.txt
 -  VehicleDigitalTwin.py
 -  Dockerfile
 -  docker-compose.yml

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 cuatro carpetas:
 - *mosquitto* contendrá el Dockerfile y los ficheros de configuración necesarios para el despliegue de esta solución MQTT de código libre en un contenedor.
 - *message_router* 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.
 - *dbService* contendrá el Dockerfile para la creación del servicio de base de datos que se utilizará para almacenar los datos operativos de la red de vehículos IoT. También contendrá un fichero con el script para la creación de la BD con el modelo definido para este caso práctico.
 - *microservices* contendrá el código de los microservicios que se desarrollarán en el ejercicio. En el ámbito de este ejercicio, los microservicios a desarrollar son:
 - *vehicles_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*vehicles_manager_api.py* y *vehicles_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.
 - *telemetry_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*telemetry_manager_api.py* y *telemetry_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.
 - *routes_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*routes_manager_api.py* y *routes_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.

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 11.

Se recomienda que, en este momento, se copien todos los ficheros de código y configuración desarrollados en la sesión 10 a la carpeta en la que se está desarrollando el código de la sesión 11, contemplando la estructura de carpetas presentada en el apartado anterior.

Desarrollo del microservicio para la gestión de rutas

En la carpeta *IoTCloudServices/microservices/routes_microservice/code* se tienen que crear dos ficheros con el código fuente que será necesario desarrollar: *routes_manager_api.py* y *routes_db_manager.py*.

Desarrollo de la interfaz API REST con Flask

La interfaz API REST que se implementará con Flask para el microservicio *routes_microservice*, se incluirá en el fichero *routes_manager_api.py*.

Los requisitos que se tienen que satisfacer para la implementación de esta interfaz son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con los microservicios desarrollados en el ejercicio 10.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4 del ejercicio 10. Sin embargo, en este caso, es necesario configurar la URL y el puerto por el cual se publicará la API REST. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5003.
3. La API REST tendrá los siguientes métodos:
 - `@app.route('/routes/assign/', methods=['POST'])`

Este método recibe como parámetros de entrada los datos de la ruta a asignar:

```
{
  "plate": "1234BBC",
  "origin": "Navalcarnero",
  "destination": "Carranque"
}
```


La instrucción para obtener los parámetros de entrada es la siguiente:

```
params = request.get_json()
```

El procesamiento de los datos recibidos se hará en dos pasos:

Primero, por un método denominado *assign_new_route*, que se implementará en el fichero *routes_db_manager.py*. Su lógica se especifica en el siguiente apartado.

En caso de que los datos proporcionados sean correctos y se hayan podido realizar la inserción en la base de datos, se procederá a solicitar al Message Router para que envíe la ruta al vehículo.

En caso de que todo haya ido bien, este método devolverá un mensaje informativo, junto con el código de éxito 201:

```
return {"result": "Route assigned"}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"result": "Error assigning a new route"}, 500
```

- `@app.route('/routes/retrieve/', methods=['GET'])`

Este método recibe como parámetros de entrada los datos del vehículo en formato JSON.

```
{"Plate": "<id>"}
```

Este método devuelve como resultado una lista en formato JSON con las rutas asignadas al vehículo que se indica como parámetro de entrada.

El procesamiento de los datos recibidos se hará por un método denominado *get_routes_assigned_to_vehicle*, que se implementará en el fichero *routes_db_manager.py*. Su lógica se especifica en el siguiente apartado.

Desarrollo de la lógica para el almacenamiento y consulta de datos

La lógica para el almacenamiento y consulta de los datos gestionados por el microservicio de los vehículos son los siguientes:

- `connect_database()`. Se emplea el mismo código que el especificado para los microservicios del ejercicio 10.

- `get_routes_assigned_to_vehicle(params)`

Este método ejecuta una consulta a la tabla de rutas correspondiente con el vehículo cuya matrícula se pasa como parámetro de entrada y devuelve todas las rutas que tiene asignadas.

- `assign_new_route (params)`

Este método inserta en la tabla de rutas la nueva que es asignada a un vehículo. En caso de que el resultado sea correcto, se devuelve True y, en caso de que haya algún error en la inserción de la telemetría, este método devolverá el resultado False.

```
with mydb.cursor() as mycursor:
    sql = "INSERT INTO vehicles_telemetry (origin, destination,
plate, time_stamp) VALUES (%s, %s, %s, %s)"
    vehicle_plate = params["plate"]
    origin = params["origin"]
    destination = params["destination"]
    time_stamp = datetime.datetime.now()

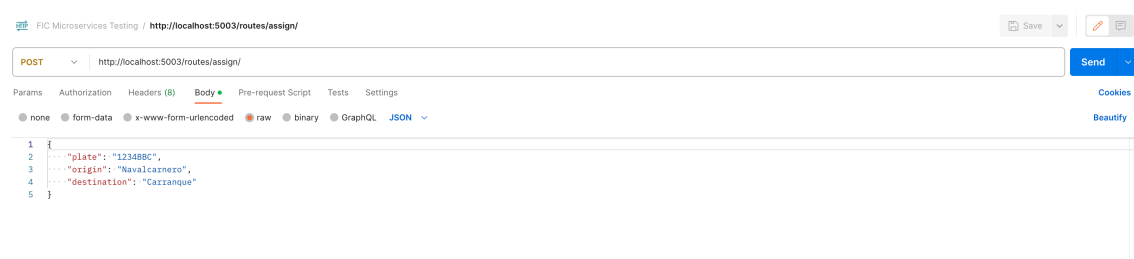
    tuples = (vehicle_plate, origin, destination, time_stamp)
    try:
        mycursor.execute(sql, tuples)
        mydb.commit()
        print(mycursor.rowcount, "Route inserted.")
        return True
    except:
        print("Error inserting a route")
        return False
```

Prueba de Routes Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

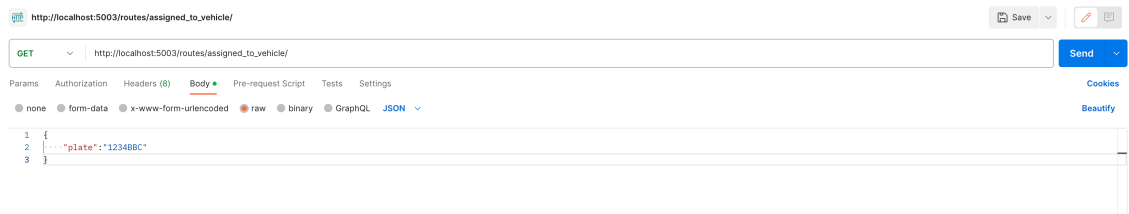
Los dos tipos de prueba a realizar con Postman son los siguientes:

- `@app.route('/routes/assign/', methods=['POST'])`



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- `@app.route('/reoutes/assigned_to_vehicle/', methods=['GET'])`



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

Cuando finalice la verificación de que el código funciona como se espera, el código desarrollado se debe publicar en el repositorio de control de versiones.

Creación de la imagen con Dockerfile

Una vez que se ha probado que el código del microservicio funciona correctamente en el equipo de desarrollo, se procederá a la preparación para su despliegue con contenedores.

Para ello, en la carpeta *IoTCloudServices/microservices/telemetry_microservice* se creará el fichero Dockerfile. El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.11.1
- 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: *Flask*, *Flask-Cors* y *mysql-connector-python*, para ello ejecutar `pip install <paquetes necesarios>`
- Ejecutar el código incluido en el fichero *routes_manager_api.py*

Configuración de la orquestación del microservicio

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 *routes_microservice* de acuerdo con el siguiente código:

MICROSERVICIOS PARA IOT - ORDENAR RUTAS

```
routes_microservice:
  build: ./microservices/routes_microservice
  ports:
    - '5003:5003'
  links:
    - "dbService:dbService"
    - "message_router:message_router"
  environment:
    - HOST=0.0.0.0
    - PORT=5003
    - DBHOST=dbService
    - DBUSER=fic_db_user
    - DBPASSWORD=RP#64nY7*E*H
    - DBDATABASE=fic_data
    - MESSAGE_ROUTER_ADDRESS=message_router
    - MESSAGE_ROUTER_PORT=5000
  volumes:
    - ".:/microservices/routes_microservice/code:/etc/usr/src/app"
  depends_on:
    - dbService
```

Despliegue del microservicio

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-cloud-services*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-cloud-services*. 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 *IoTCloudServices* 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>
```

Actualización del servicio Message Router

Para actualizar 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 resto de servicios considerados para la arquitectura de nube.

Retirada de código anterior

El código que se ha desarrollado en sesiones anteriores relativos al envío de rutas a los gemelos digitales del simulador del vehículo se tiene que retirar porque, en el ámbito de este ejercicio, esta funcionalidad se tendrá que implementar a partir de la integración con el microservicio de rutas desarrollado en la sección 3 de este ejercicio.

Configuración de Message Router como microservicio con API REST

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. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con los microservicios desarrollados en el ejercicio 10.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4 del ejercicio 10. Sin embargo, en este caso, es necesario configurar la URL y el puerto por el cual se publicará la API REST. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5000.
3. La API REST tendrá el siguiente método:

- `@app.route('/routes/send/', methods=['POST'])`

Este método recibe como parámetros de entrada los datos de la ruta a enviar:

```
{
  "plate": "1234BBC",
  "origin": "Navalcarnero",
  "destination": "Carranque"
}
```

El procesamiento de los datos recibidos se hará publicando la ruta recibida por el correspondiente topic:

```
params = request.get_json()
route = {"Origin": params["Origin"], "Destination":
params["Destination"]}
client.publish("/fic/vehicles/" + params["Plate"] + "/routes",
              payload=json.dumps(route), qos=1, retain=False)
```

En caso de que el envío se haya podido realizar correctamente, se devolverá un mensaje informativo, junto con el código de éxito 201:

```
return {"Result": "Route successfully sent"}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"Result": "Internal Server Error"}, 500
```

Traslado de las comunicaciones MQTT a un hilo de ejecución alternativo

Para que el servidor Flask pueda ejecutarse al mismo tiempo que se realizan las comunicaciones a través del protocolo MQTT es necesario que los elementos de Flask estén en hilos de ejecución diferentes, siendo obligatorio que Flask se encuentre en el hilo principal de ejecución.

Por tanto, será necesario realizar las siguientes modificaciones sobre el código actual de Message Router:

1. Crear un Daemon Thread para la ejecución del protocolo MQTT antes de lanzar la aplicación Flask

```
global client

client = mqtt.Client()

t1 = threading.Thread(target=mqtt_listener, daemon=True)
t1.start()
CORS(app)
app.run(host=API_HOST, port=API_PORT, debug=True)
```

2. En el método que implementa este nuevo hilo de ejecución se tiene que realizar la conexión a MQTT y se tiene que quedar en escucha de los mensajes que pueda recibir a través de Mosquitto.

```
client.username_pw_set(username="fic_server",
password="fic_password")
client.on_connect = on_connect
client.on_message = on_message
client.connect(MQTT_SERVER, MQTT_PORT, 60)
client.loop_forever()
```

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

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
    - VEHICLES_MICROSERVICE_ADDRESS= vehicles_microservice
    - VEHICLES_MICROSERVICE_PORT=5001
    - TELEMETRY_MICROSERVICE_ADDRESS= telemetry_microservice
    - TELEMETRY_MICROSERVICE_PORT=5002
    - TELEMETRY_MICROSERVICE_ADDRESS= routes_microservice
    - TELEMETRY_MICROSERVICE_PORT=5003
    - PYTHONUNBUFFERED=1
    - HOST=0.0.0.0
    - PORT=5000
  ports:
    - '5000:5000'
  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 anteriormente.

MICROSERVICIOS PARA IOT - ORDENAR RUTAS

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>
```


Criterios de evaluación y procedimiento de entrega



Criterios de Evaluación

- Routes Microservice – Registro de Rutas – 3,5 puntos
- Routes Microservice – Consulta de Rutas – 3,5 puntos
- Message Router – Actualización para que funcione como API REST – 3 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 Sesión11 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 Microservicio de Gestión de Rutas, 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.



Sugerencias

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