Universidad Sergio Arboleda

Escuela de Ciencias Exactas e Ingeniería

Taller Práctico 4: Optimización de rutas del viajero con clusters de servicio

Juan David López Yara
Julio César Flórez

Docente: Guillermo Andrés de Mendoza Corrales

Bogotá D.C.
2025

1. Objetivo del laboratorio:

   Diseñar y construir una solución distribuida y escalable para el Problema del Viajante (TSP) aplicando arquitecturas modernas de microservicios. Los estudiantes desarrollarán una API RESTful con Flask que realizará los cálculos de distancia, la cual será desplegada como un servicio elástico con N réplicas en Docker Swarm. Finalmente, implementará un programa cliente de fuerza bruta que explotará el balanceo de carga del cluster para encontrar el path óptimo. Al finalizar este taller, los estudiantes serán capaces de:

   1. Diseñar y crear una API REST simple usando Python y Flask para realizar un cálculo.
   2. Contenerizar la aplicación Flask usando Docker.
   3. Desplegar un servicio escalable (swarm) de la API en Docker Swarm.
   4. Implementar un cliente Python de fuerza bruta para interactuar con la API y encontrar la ruta óptima al problema del viajero.

2. Marco teórico:

   Se inicia con la creación de una carpeta la cual aloja todo el proyecto y a la cual se le creará un entorno virtual, dentro de estas se alojan dos carpetas que son venv (herramientas del entorno de desarrollo), api (alojar el server, los requisitos de software y el archivo Dockerfile) y cliente(que aloja las solicitudes que se le harán al server), ademas, se tendra un archivo .yml en la carpeta raíz con el cual le podremos indicar a docker que vamos a generar réplicas del mismo contenedor (microservicios) para atender al cliente. El archivo server o main aloja la función que va a servir como en servicio y que devolverá un resultado al cliente, además guarda las rutas con las cuales se podrá combinar con este último y enviar y recibir datos a través de los request de JSON. Ahora con el docker, inicialmente se tendrá que instalar el docker desktop con el cual mediante el dockerfile, podremos generar una imagen de lo que se le indique en ese archivo y en a qué puerto va a responder esas solicitudes, con ello, al momento de ejecutar en el entorno virtual al cliente, sin haber activado el main, recibirá su respectiva respuesta indicando que el contenedor está trabajando y está respondiendo al puerto al que se le fue asignado el JSON al cliente para enviar sus requests. Con ello, luego se realiza el swarm con la imagen que poseemos, con ella generamos réplicas del mismo contenedor con la aplicación, ahora teniendo más capacidad para responder a más clientes si dado el caso se hacen múltiples solicitudes.

3. Metodología:

Configuración de hardware:

- Partición de la cpu en el docker con contenedores réplicas (automático)

Configuración de software:

- Crear una carpeta y crear su entorno virtual
- Instalar flask al entorno virtual (desde CMD)
- Instalar Gunicorn al entorno virtual (desde CMD)
- Instalar Networkx al entorno virtual (desde CMD)
- Instalar requests al entorno virtual (desde CMD)
- Instalar docker desktop: Instalar WSL 2 (desde cmd) para poder usar docker en windows, es un entorno que permite usar aplicaciones de Linux (instala Ubuntu).

Explicación del algoritmo:

- main.py: Aloja la raíz de flask, donde se alojan todas las rutas que llaman a sus respectivas funciones asignadas y las ejecutan, retornando resultados al cliente mediante JSON. La función del TSP básicamente recibe un path con un orden específico que va a evaluar la función del main dentro de su grafo construido y retorna la distancia total recorrida o los valores de las aristas entre nodos.

```python
@app.route('/')
def inicio():
    return "Servidor TSP en funcionamiento"


@app.route('/calcular_distancia', methods=['POST'])
def calcular_distancia():
    data= request.get_json()

    if 'path' not in data:
        return jsonify({"error": "No se encuentra 'path'"}), 400

    path = data["path"]
    total = 0

    for i in range(len(path) - 1):
        origen = path[i]
        destino = path[i + 1]
        total += grafo[origen][destino]["peso"]

    print("Atendiendo desde:", socket.gethostname())
    return jsonify({"distancia_total": total})


if __name__ == '__main__':
    app.run(debug=True)
```

- cliente.py: Aloja todas las combinaicone sposibles que se puedan generar con la lista de las ciudades establecidas en el grafo, y va enviando una por una mediante un request como un mensaje con id path, el cual recibirá el main o server y con el cual hará todos los cálculos y devolverá una respuesta. Así con todas las permutaciones y las va evaluando y se quedará con la mejor ruta.

```python
import itertools
import requests

ciudades = ["A","B","C","D","E","F"]
mejor_distancia = float("inf")#maximo valor posible para comparar
mejor_ruta = None

for perm in itertools.permutations(ciudades):
    ruta = list(perm) + [perm[0]]  # cerramos ciclo A...A

    response = requests.post(
        url = "http://127.0.0.1:5000/calcular_distancia",
        json={"path": ruta}
    )

    distancia = response.json()["distancia_total"]

    if distancia < mejor_distancia:
        mejor_distancia = distancia
        mejor_ruta = ruta

print("Mejor ruta:", mejor_ruta)
print("Distancia:", mejor_distancia)
```

4. Resultados:

   Se pudo evidenciar que el docker con los contenedores réplicas lograron dar respuesta al request del cliente gracias a la comunicación por el puerto 5000 que es donde estos contenedores responden. Al hacerlos recibir solicitudes, se pudo observar que devolvieron al cliente el resultado esperado (la ruta más óptima posible en el grafo).

```
Mejor ruta: ['A', 'B', 'E', 'D', 'F', 'C', 'A']
Distancia: 19
(venv) PS C:\Users\juand\OneDrive\Escritorio\TSP-flask-docker\cliente>
```

```
[2025-11-26 18:30:06 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2025-11-26 18:30:06 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2025-11-26 18:30:06 +0000] [1] [INFO] Using worker: sync
[2025-11-26 18:30:06 +0000] [7] [INFO] Booting worker with pid: 7
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
Atendiendo desde: a38f40fdac32
```

```
[2025-11-26 18:30:06 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2025-11-26 18:30:06 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2025-11-26 18:30:06 +0000] [1] [INFO] Using worker: sync
[2025-11-26 18:30:06 +0000] [7] [INFO] Booting worker with pid: 7
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
Atendiendo desde: f93e837f812d
```

```
[2025-11-26 18:30:06 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2025-11-26 18:30:06 +0000] [1] [INFO] Listening at: http://0.0.0.0:5000 (1)
[2025-11-26 18:30:06 +0000] [1] [INFO] Using worker: sync
[2025-11-26 18:30:06 +0000] [7] [INFO] Booting worker with pid: 7
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
Atendiendo desde: 781127d9bbbc
```

5. Análisis de rendimiento:

   Los clusters que conforman el docker de servicios para la solicitud del cliente no tomaron mucha capacidad de la cpu, cuando se le asignaba una solicitud a un cluster en específico, su consumo no supera el 0.05% de uso de cpu, mientras que los otros mantenían un nivel de 0.01% de uso.

Name	Container ID	Image	Port(s)	CPU (%)	Last started
contenedor-tsp-api	aa4715b8fdb2	tsp-api		0.03%	4 hours ago
cluster-tsp_tsp-api.2.m9	a38f40fdac32	tsp-api:latest		0.05%	4 hours ago
cluster-tsp_tsp-api.1.771	f93e837f812d	tsp-api:latest		0.01%	4 hours ago
cluster-tsp_tsp-api.3.w8	781127d9bbbc	tsp-api:latest		0.01%	4 hours ago

6. Conclusiones:

   Se pudo trabajar y observar cómo funciona de manera muy pequeña una red de microservicios que tenga el objetivo de atender múltiples solicitudes al mismo servicio, para lo cual este último se debe replicar gracias a swarm en docker, particionando la memoria y ahora sí pudiendo atender varias solicitudes al mismo servicio.