Taller 4: Optimización de Rutas del viajero con clusters de servicios

Juan Joya - William Aristizabal - Ana Hernandez

1. Objetivo del Laboratorio:

Diseñar y construir una solución distribuida y escalable para el **Problema del Viajante (TSP)** aplicando arquitecturas modernas de microservicios.

2. Marco Teórico:

El problema del Viajante (TSP) es NP-difícil: el número de rutas posibles crece factorialmente con el número de ciudades (n!), por lo que el enfoque de fuerza bruta solo es viable para tamaños pequeños. El cálculo de la distancia de una ruta se basa en la suma de distancias euclidianas entre puntos consecutivos ($\sqrt{((x_2-x_1)^2+(y_2-y_1)^2)}$). Para aliviar la carga de cómputo y facilitar la experimentación, se emplea una arquitectura de microservicio expuesta vía una API REST (Flask) que encapsula el cálculo de distancias. La contenedorización con Docker y la orquestación con Docker Swarm permiten replicar el servicio, balancear carga y aislar dependencias, alineándose con principios de cómputo de alto rendimiento (HPC) y escalabilidad horizontal.

3. Metodologia:

- ### Configuración del Hardware:

  Host de laboratorio con CPU multinúcleo, 16 GB de RAM, disco SSD y red local estable (1 Gbps). El despliegue en Swarm se realizó sobre el mismo host en modo manager, por lo que la comunicación servicio-cliente es local (localhost:5000), reduciendo la latencia de red externa.

- ### Configuración del Software:

  Sistema operativo Linux (Mint 22.1). Docker Engine con Swarm inicializado (docker swarm init). Imagen construida desde dockerfile (docker build -t calculator:1 ...) y desplegada como servicio replicado

(docker service create --replicas 4 -p 5000:5000 calculator (line 1)). Entorno Python 3.x aislado con venv; dependencias instaladas con pip install -r requirements.txt (Flask para la API y requests para el cliente).

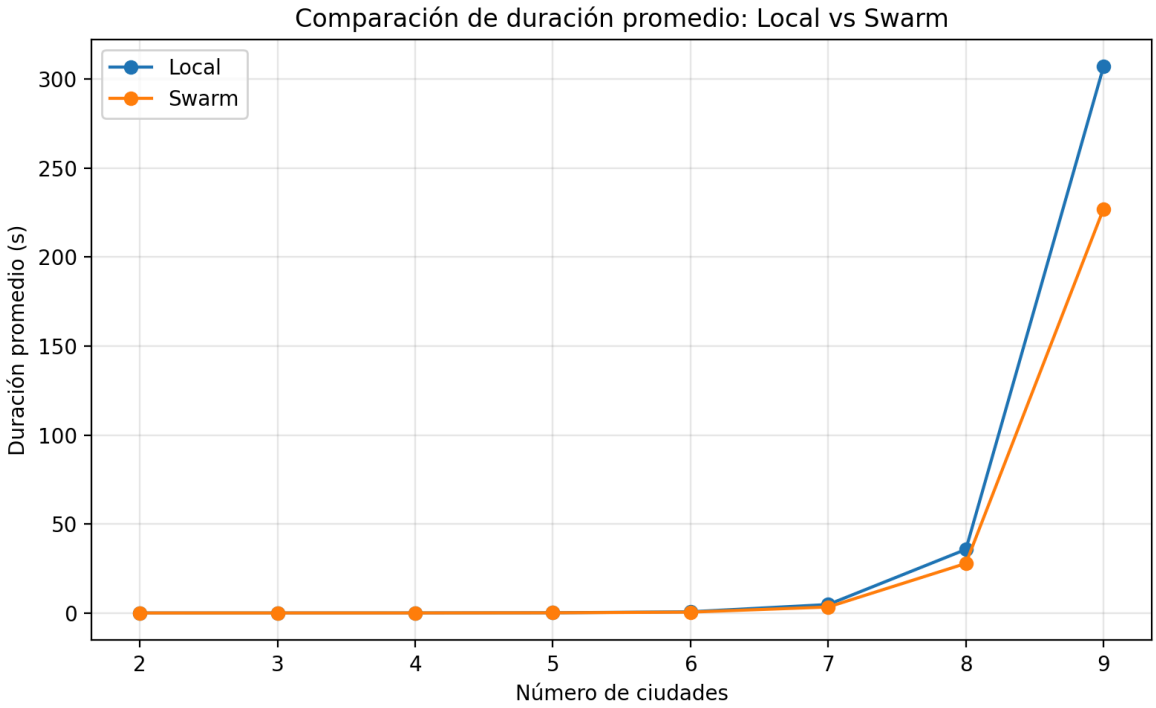
- **Explicación del algoritmo desarrollado**

    - La API Flask expone POST /calculate_distance: valida el JSON de entrada (cities), convierte coordenadas a float, calcula la distancia total con suma de distancias euclidianas entre ciudades consecutivas y devuelve total_distance.
    - El script bruteForce.py genera todas las permutaciones de los identificadores de ciudades (enfoque de fuerza bruta, complejidad O(n!)). Para cada permutación construye el payload, llama al endpoint y recibe la distancia calculada por el servicio.
    - Se mantiene la mejor ruta encontrada (menor distancia) comparando cada resultado; al finalizar se imprime la secuencia óptima y su costo total.
    - Ejecutar el cálculo desde el cliente, mientras el servicio corre replicado en Swarm, permite repartir las solicitudes entre réplicas y observar el comportamiento del microservicio bajo carga sin modificar la lógica de búsqueda.

4. Resultados:

Ciudades	Local (s)	Swarm (s)
2	0.0053	0.0035
3	0.0084	0.0070
4	0.0258	0.0226
5	0.1073	0.1011
6	0.6706	0.4907
7	4.7440	3.3622
8	35.7130	27.8386
9	307.1026	227.1474

Tabla 1: *Comparación de rendimiento entre ejecucion local y ejecucion con el cluster dockerizado*



Gráfica 1: *Gráfico de línea mostrando comparación de rendimiento entre ejecución local y ejecución con el cluster dockerizado*

5. Análisis de Rendimiento:

Las curvas y la tabla muestran que Swarm recorta tiempo en todas las escalas: la mejora de duración va desde 6% en 5 ciudades hasta 33% en casos pequeños (2 ciudades), estabilizándose en 22–29% para problemas grandes (8–9 ciudades). En promedio, **Swarm es 21.5% más rápido** que la ejecución local, manteniendo distancias óptimas similares; es decir, la ganancia principal es de rendimiento sin degradar la calidad de la solución.

6. Conclusiones:

El análisis realizado demuestra que el uso de Docker Swarm para la ejecución del algoritmo del Problema del Viajante (TSP) ofrece una mejora significativa en el rendimiento, especialmente en problemas con mayor cantidad de ciudades. La comparación de tiempos de ejecución entre el entorno local y el cluster dockerizado muestra que Swarm es un 21.5% más rápido en promedio, sin comprometer la calidad de la solución. Esto resalta la eficacia de la contenedorización y orquestación en la mejora de la escalabilidad y el balanceo de carga, lo que permite optimizar el uso de los recursos disponibles. De esta manera, la solución propuesta no solo optimiza el tiempo de cómputo, sino que también demuestra ser una herramienta eficiente para enfrentar problemas de optimización a gran escala.