Informe de Laboratorio 4: Optimización de Rutas del Viajero con Clusters de Servicios

Juan Hurtado, Miguel Flechas, Andres Castro

Universidad Sergio
Arboleda

Noviembre

2025

Computación de Alto Rendimiento (HPC)

1. Introducción y Objetivos

1.1 Introducción

La computación de alto rendimiento (HPC) y los sistemas distribuidos han evolucionado desde grandes mainframes monolíticos hacia arquitecturas flexibles basadas en microservicios y contenedores. En este laboratorio, exploramos la intersección entre problemas clásicos de optimización combinatoria, como el Problema del Viajante (TSP), y las tecnologías modernas de orquestación como Docker Swarm. El propósito es demostrar cómo la paralelización y el balanceo de carga pueden mitigar la complejidad computacional inherente a los algoritmos de fuerza bruta.

1.2 Objetivo General

Diseñar, implementar y desplegar una solución distribuida y escalable para resolver el Problema del Viajante (TSP), utilizando una arquitectura de microservicios orquestada por Docker Swarm para distribuir la carga computacional de manera eficiente.

1.3 Objetivos Específicos

1. Desarrollo de Microservicios: Implementar una API RESTful robusta utilizando Python y Flask que encapsule la lógica de cálculo de distancias euclidianas.
2. Contenerización: Empaquetar la aplicación y sus dependencias en contenedores Docker ligeros y portables, asegurando la consistencia entre entornos de desarrollo y producción.
3. Orquestación y Escalabilidad: Configurar un cluster de Docker Swarm y desplegar el servicio de cálculo con múltiples réplicas (N=4) para habilitar el procesamiento paralelo y la alta disponibilidad.

4. Cliente de Alto Rendimiento: Desarrollar un cliente en Python que implemente un algoritmo de fuerza bruta concurrente, capaz de saturar el cluster de servicios para encontrar la ruta óptima en el menor tiempo posible.

2. Marco Teórico

2.1 El Problema del Viajante de Comercio (TSP)

El TSP (Traveling Salesperson Problem) plantea la siguiente pregunta: "Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen?". Matemáticamente, es un problema de optimización combinatoria clasificado como NP-Hard. Para n ciudades, existen $(n-1)!/2$ rutas únicas posibles (asumiendo simetría). Esto implica que la complejidad temporal es factorial $O(n!)$, lo que hace que los métodos exactos sean inviables para grandes valores de n, pero ideales para demostrar la potencia del cómputo paralelo en instancias pequeñas y medianas.

2.2 Docker y Docker Swarm

- Docker: Plataforma de contenerización que permite empaquetar una aplicación con todas sus dependencias (librerías, runtime, herramientas del sistema) en una unidad estandarizada llamada contenedor.
- Docker Swarm: Es la herramienta de orquestación nativa de Docker. Permite gestionar un cluster de motores Docker como un único sistema virtual. Sus características clave incluyen:
    - Service Discovery: Asignación automática de DNS a los servicios.
    - Load Balancing: Distribución automática de las peticiones entrantes entre los contenedores (réplicas) disponibles.

- Scaling: Capacidad de aumentar o disminuir el número de réplicas con un solo comando.

2.3 Arquitectura REST y Microservicios

La arquitectura REST (Representational State Transfer) define un conjunto de restricciones para crear servicios web escalables. En este laboratorio, el "cálculo de distancia" se desacopla en un microservicio independiente. Esto permite que el componente de cómputo escale independientemente del cliente que genera las permutaciones, siguiendo el principio de responsabilidad única.

3. Metodología

3.1 Entorno de Desarrollo

- Hardware: Arquitectura x86_64, Procesador Multi-core (Intel/AMD), 16GB RAM.
- Sistema Operativo: Windows 10/11 con subsistema Linux (WSL2).
- Software:
    - Python 3.9
    - Docker Desktop (Engine v24+)
    - Librerías: Flask (API), Gunicorn (WSGI Server), Requests (Cliente HTTP).

3.2 Diseño de la Solución

A. Servicio de Cálculo (Backend)

Se desarrolló una API en Flask (

app.py) con un único endpoint POST /calculate_distance.

- Entrada: JSON con una lista ordenada de coordenadas {'route': [{'x':0, 'y':0}, ...]}.

- Lógica: Itera sobre la lista calculando la distancia euclidiana entre puntos adyacentes P_i y P_{i+1}: $$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$
- Salida: JSON con la distancia total acumulada.

B. Despliegue en Swarm

Se creó un

Dockerfile optimizado basado en python:3.9-slim para minimizar el tamaño de la imagen. El despliegue se automatizó mediante scripts de PowerShell que realizan:

1. Construcción de la imagen (docker build).
2. Inicialización del Swarm (docker swarm init).
3. Creación del servicio con 4 réplicas (docker service create --replicas 4).

C. Cliente Concurrente

El cliente (

client.py) no solo genera permutaciones, sino que actúa como un orquestador de carga:

1. Generación de Rutas: Utiliza itertools.permutations para obtener todas las combinaciones posibles de ciudades.
2. Paralelismo: Implementa concurrent.futures.ThreadPoolExecutor. Esto permite lanzar múltiples hilos (threads) donde cada uno realiza una petición HTTP al Swarm.
3. Optimización: Mantiene una variable global (protegida o atómica en el contexto de GIL) para rastrear la distancia mínima encontrada hasta el momento.

4. Resultados Experimentales

4.1 Escenario de Prueba

Se definió un escenario con 5 ciudades distribuidas en un plano cartesiano 10x10:

- A: (0, 0)

- B: (10, 0)
- C: (10, 10)
- D: (0, 10)
- E: (5, 5)

Espacio de búsqueda: $5! = 120$ rutas posibles.

4.2 Ejecución y Métricas

Al ejecutar el cliente contra el cluster Swarm de 4 nodos, se obtuvieron los siguientes datos:

Métrica	Valor
Total Rutas Evaluadas	120
Tiempo Total de Ejecución	0.23 segundos
Ruta Óptima Encontrada	A -> B -> C -> E -> D
Distancia Mínima	34.14 unidades
Throughput Estimado	~521 peticiones/segundo

4.3 Evidencia de Balanceo de Carga

Mediante la inspección de logs (docker service logs calculator), se confirmó que las peticiones fueron atendidas por diferentes contenedores (IDs distintos), validando que el balanceador de carga interno de Docker Swarm (Ingress Network) funcionó correctamente distribuyendo el tráfico Round-Robin.

5. Análisis de Resultados

1. Eficacia del Algoritmo: El sistema encontró correctamente la ruta óptima (perímetro del cuadrado más la entrada al centro), validando la lógica matemática del servidor y la lógica de permutación del cliente.
2. Impacto de la Concurrencia:
    - El uso de ThreadPoolExecutor fue crítico. En una prueba secuencial (un solo hilo), el tiempo de ejecución habría sido la suma lineal de todas las latencias de red + tiempos de cómputo ($T_{total} = \sum t_i$).
    - Con 10 hilos concurrentes y 4 réplicas de servidor, el sistema se aproximó a un comportamiento donde $T_{total} \approx (\sum t_i) / 4$, limitado principalmente por el overhead de la red local y la creación de conexiones HTTP.
3. Escalabilidad Horizontal: El diseño permite que, si el número de ciudades aumentara a 6 (720 rutas) o 7 (5040 rutas), podríamos simplemente añadir más réplicas al servicio (docker service scale calculator=10) sin modificar una sola línea de código, manteniendo tiempos de respuesta aceptables.

6. Conclusiones

El desarrollo de este laboratorio permitió evidenciar en la práctica los beneficios de la computación distribuida aplicada a problemas de optimización:

1. Desacoplamiento Exitoso: La separación entre la lógica de generación de problemas (Cliente) y la lógica de evaluación (Servidor en Swarm) permite arquitecturas flexibles donde el poder de cómputo puede crecer bajo demanda.
2. Transparencia del Middleware: Docker Swarm abstrae la complejidad de la red. Para el cliente, el cluster parece ser un único servidor potente; no necesita conocer la

existencia de las 4 réplicas, simplificando enormemente el desarrollo del software cliente.

3. Viabilidad de Fuerza Bruta Distribuida: Aunque el TSP es NP-Hard, para instancias pequeñas y medianas, la fuerza bruta distribuida se vuelve viable gracias a la paralelización masiva. Este mismo patrón arquitectónico es aplicable a otros problemas complejos como el plegamiento de proteínas o el renderizado de gráficos.
4. Robustez: La arquitectura de microservicios demostró ser robusta; si una réplica fallara, Swarm redirigiría el tráfico a las restantes, garantizando que el cálculo de la ruta óptima no se detenga.