Taller Práctico 4: Optimización de Rutas del Viajero con Clusters de Servicios

Kevin García

Juan Marquez
Universidad Sergio Arboleda

Noviembre 2025

Resumen

Este informe documenta la implementación de una solución distribuida y escalable para el Problema del Viajante (TSP) utilizando arquitectura de microservicios. Se desarrolló una API RESTful con Flask, se containerizó con Docker, se desplegó en Docker Swarm con múltiples réplicas, y se implementó un cliente de fuerza bruta con paralelización usando concurrent.futures. El análisis compara el rendimiento entre enfoques secuenciales y paralelos con diferentes configuraciones de workers.

1. Objetivos del Laboratorio

- Diseñar y crear una API REST simple usando Python y Flask para cálculos de distancia
- Containerizar la aplicación Flask usando Docker
- Desplegar un servicio escalable de la API en Docker Swarm
- Implementar un cliente Python de fuerza bruta para resolver TSP
- Paralelizar el cliente usando concurrent.futures para explotar el balanceo de carga

2. Marco Teórico

El Problema del Viajante (TSP) es un problema de optimización combinatoria NP-duro que busca encontrar la ruta más corta que visite un conjunto de ciudades exactamente una vez y regrese al origen.

En el contexto de procesamiento distribuido, Docker Swarm permite desplegar múltiples réplicas de un servicio, proporcionando escalabilidad y balanceo de carga automático. El uso de concurrent.futures en el cliente permite enviar múltiples solicitudes simultáneamente, aprovechando la arquitectura de microservicios.

3. Metodología

3.1. Arquitectura de la Solución

- **Servidor Flask**: API REST con endpoint `/calculate_distance`
- **Docker Container**: Aplicación Flask containerizada
- **Docker Swarm**: Cluster con 4 réplicas del servicio
- **Cliente Python**: Implementación secuencial y paralela con concurrent.futures

3.2. Configuración del Hardware

- Sistema: Linux
- Cluster Docker Swarm con múltiples nodos
- Configuración de red: Balanceador de carga integrado de Swarm

3.3. Configuración del Software

- Python 3.x
- Flask para el servidor de cálculo de distancias
- Docker y Docker Swarm para orquestación
- Requests con retry strategy y connection pooling
- ThreadPoolExecutor de concurrent.futures para paralelización

3.4. Algoritmo Desarrollado

3.4.1. Servidor Flask

```
@app.route('/calculate_distance', methods=['POST'])
def calculate_distance():
    cities = request.get_json()['cities']
    total = 0.0
    for i in range(len(cities) - 1):
        distance = euclidean(cities[i], cities[i+1])
        total += distance
    return jsonify({'total_distance': total})
```

3.4.2. Cliente Paralelo con Swarm

```
# Configuraci n para Docker Swarm
session = requests.Session()
retry_strategy = Retry(total=5, backoff_factor=0.2)
adapter = HTTPAdapter(pool_connections=100, pool_maxsize=100)
session.mount("http://", adapter)

# Paralelizaci n con ThreadPoolExecutor
with concurrent.futures.ThreadPoolExecutor(max_workers=workers) as executor
    for perm in generate_reduced_permutations(names):
        futures.append(executor.submit(worker_task, perm, cities_dict))
```

3.4.3. Despliegue en Swarm

```
docker swarm init
docker service create --name calculator --replicas 4 -p 5000:5000 travel-ca
```

4. Resultados

4.1. Pruebas con 6 Ciudades en Swarm

Cuadro 1: Rendimiento TSP - 6 Ciudades (120 permutaciones) con Swarm

Enfoque	Workers	Tiempo Total (s)	Tiempo/Perm (s)	Llamadas API
Secuencial	-	0.0004	0.00000307	-
Paralelo + Swarm	1	0.7612	0.01268644	360
Paralelo + Swarm	2	0.5431	0.00905172	360
Paralelo + Swarm	4	0.5182	0.00863639	360
Paralelo + Swarm	10	0.5043	0.00840469	360
Paralelo + Swarm	20	0.5411	0.00901841	360

4.2. Pruebas con 8 Ciudades - Versión Inicial con Swarm

Cuadro 2: Rendimiento TSP - 8 Ciudades (5040 permutaciones) con Swarm

Enfoque	Workers	Tiempo Total (s)	Tiempo/Perm (s)	Llamadas API
Secuencial	-	0.0076	0.00000151	-
Paralelo + Swarm	1	63.6502	0.02525802	20160
Paralelo + Swarm	2	44.9751	0.01784726	20160
Paralelo + Swarm	4	48.4258	0.01921658	20160
Paralelo + Swarm	10	43.5736	0.01729112	20160
Paralelo + Swarm	20	53.1277	0.02108242	20160

4.3. Pruebas con 8 Ciudades - Versión Optimizada

Cuadro 3: Rendimiento TSP Optimizado - 8 Ciudades con Ruta Completa

Workers	Tiempo Total (s)	Permutaciones	Llamadas API	Mejores Rutas
1	6.8460	2520	2521	7
2	5.0038	2520	2521	6
4	4.9715	2520	2521	8
10	5.0939	2520	2521	8

5. Análisis de Rendimiento

5.1. Impacto de Docker Swarm y Concurrent.Futures

- **Balanceo de Carga**: Las 4 réplicas en Swarm distribuyeron automáticamente las solicitudes entre los contenedores
- **Connection Pooling**: La configuración de `HTTPAdapter` con `pool_connections=100` permitió reutilizar conexiones HTTP
- **Retry Strategy**: La estrategia de reintentos manejó fallos temporales de red o sobrecarga del servidor

5.2. Paradoja del Rendimiento

¿Por qué el enfoque paralelo es más lento a pesar de usar Swarm y concurrent.futures?

1. **Overhead de Red**: Cada llamada HTTP introduce latencia (10-100ms) vs cálculo local (microsegundos)
2. **Serialización JSON**: Conversión de objetos Python a JSON y viceversa en cada solicitud
3. **Coordinación de Threads**: El `ThreadPoolExecutor` introduce overhead de sincronización
4. **Límites de Swarm**: Aunque hay 4 réplicas, el balanceador de carga y la red comparten recursos

5.3. Análisis de Escalabilidad

- **6 Ciudades**: El paralelismo no justifica el overhead (0.0004s vs 0.5s)
- **8 Ciudades**: Mejora significativa con optimización (63.65s a 4.97s)
- **Punto Óptimo**: 4 workers coinciden con las 4 réplicas del Swarm
- **Saturación**: Más de 10 workers degradan el rendimiento por contención

5.4. Efectividad de la Optimización

Cuadro 4: Comparación de Estrategias de Comunicación

Estrategia	**Llamadas API**	**Tiempo (s)**	**Reducción**
Segmentada (1 edge/llamada)	20,160	43.57	-
Ruta Completa	2,521	4.97	88.6 %

6. Conclusiones

6.1. Logros del Taller

1. **Arquitectura Exitosa**: Se implementó exitosamente el sistema completo con Flask, Docker, Swarm y cliente paralelo
2. **Balanceo de Carga Funcional**: Las 4 réplicas en Swarm distribuyeron la carga efectivamente
3. **Paralelización Efectiva**: `concurrent.futures` permitió explotar el paralelismo a nivel de cliente
4. **Optimización Significativa**: La reducción de llamadas API mejoró el rendimiento en casi 90 %

6.2. Lecciones Aprendidas

- **Microservicios No Son Silver Bullet**: Para problemas computacionalmente simples, el overhead de comunicación domina
- **Swarm para Carga Real**: El balanceo de carga es efectivo cuando el tiempo de cálculo ¿tiempo de comunicación
- `concurrent.futures` es Adecuado para I/O: Ideal para operaciones de red como llamadas HTTP
- **Optimizar Comunicación**: Agrupar operaciones reduce dramáticamente el overhead

6.3. Conclusión General

El taller demostró exitosamente los principios de arquitectura de microservicios y procesamiento paralelo. Si bien se identificaron limitaciones en el rendimiento para problemas pequeños, las técnicas implementadas (Docker Swarm, concurrent.futures, optimización de comunicaciones) proporcionan una base sólida para sistemas distribuidos escalables. La clave está en matching la arquitectura con la complejidad computacional del problema.