Taller Práctico 4:

Por: Jose Santiago Gonzalez y Oscar Mateo Arrubla

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.

MARCO TEÓRICO

El Problema del Viajante (TSP) consiste en encontrar el recorrido más corto que permite visitar un conjunto de ciudades una sola vez antes de retornar al punto inicial. Es un problema NP-Hard, por lo cual no existen algoritmos eficientes para resolverlo de manera exacta en casos grandes.

El enfoque implementado en este laboratorio combina:

- Fuerza bruta:
  Evaluación de todas las permutaciones posibles de rutas entre N ciudades (N!). Para N=5, existen 120 rutas posibles
- Microservicios:
  La API Flask realiza únicamente el cálculo del costo de una ruta, siguiendo el principio de responsabilidad única. Esto permite escalar el servicio fácilmente
- Docker y contenedores:
  Contenerizar la API permite que sea replicada para aumentar el rendimiento. Aunque para este laboratorio se usó un solo servicio, la arquitectura es compatible con réplica y balanceo (Docker Swarm/Kubernetes)
- Paralelismo:
  El cliente utiliza ThreadPoolExecutor para evaluar múltiples rutas en paralelo, reduciendo significativamente el tiempo de procesamiento.

METODOLOGÍA

1. Implementación de la API Flask:
   - Se creó un endpoint /calculate_distance que recibe un conjunto de ciudades y calcula la distancia total entre ellas usando la métrica euclidiana.
   - Se añadió un endpoint /health para verificar disponibilidad del servicio.

2. Contenerización con Docker:
    - Se preparó un Dockerfile basado en Python 3.9-slim.
    - La imagen se construyó y expuso en el puerto 5000.
    - La API fue ejecutada como un contenedor accesible desde el host.
3. Cliente de fuerza bruta en Python:
    - Se generaron todas las permutaciones de rutas.
    - Cada permutación se envió al API para evaluar su distancia total
    - Se compararon los enfoques secuencial y paralelo con 10 hilos
4. Medición y análisis de rendimiento:
    - Se midieron tiempos totales, promedio por ruta, número de hilos y speedup.

Hardware:

- CPU: 2 vCPU
- RAM 12 GB
- Entorno: Google Colab

Software:

- Lenguaje usado: Python 3.x
- Flask
- Docker
- Librerías usadas: itertools, math, request, concurrent.futures, time.

Algoritmos desarrollados:

API Flask – Cálculo de Distancias:

1. El cliente envía una lista de ciudades en formato JSON mediante un POST al endpoint /calculate_distance.
2. La API valida que la lista sea correcta y que cada ciudad tenga coordenadas x y y.
3. Recorre la lista de ciudades en el orden recibido.
4. Calcula la distancia euclidiana entre cada par consecutivo.
5. Suma todas las distancias para obtener la distancia total de la ruta.
6. Retorna un JSON con distancia total, ruta recorrida y número de ciudades.

```python
def calculate_distance():
    data = request.get_json()
    cities = data['cities']
    total_distance = 0.0
    route_names = []

    for i in range(len(cities)):
        route_names.append(cities[i].get('name', f'City_{i}'))

        if i < len(cities) - 1:
            distance = calculate_euclidean_distance(cities[i], cities[i + 1])
            total_distance += distance

    return jsonify({
        "total_distance": round(total_distance, 2),
        "route": route_names,
        "num_cities": len(cities)
    }), 200
```

Algoritmo Secuencial (Cliente – Fuerza Bruta)

1. Se generan todas las permutaciones posibles de las ciudades (n! rutas).
2. Cada ruta se envía al API mediante una petición POST.
3. El cliente espera la respuesta antes de enviar la siguiente ruta.
4. Se registra la mejor distancia obtenida hasta el momento.
5. Se mide el tiempo total del proceso secuencial.

```python
def brute_force_sequential():
    start_time = time.time()
    best_distance = float('inf')
    best_route = None

    for route in itertools.permutations(CITIES):
        distance, route_names = calculate_route_distance(route)
        if distance is not None and distance < best_distance:
            best_distance = distance
            best_route = route_names

    elapsed = time.time() - start_time
    print(f"Tiempo total secuencial: {elapsed:.2f}s")
    return best_route, best_distance, elapsed
```

Algoritmo Paralelo Multihilos (Cliente – Fuerza Bruta concurrente)

1. Se generan todas las permutaciones de ciudades.
2. Cada permutación se envía como tarea independiente a un hilo.
3. Se usa ThreadPoolExecutor para manejar múltiples solicitudes simultáneas al API.
4. Las respuestas se van procesando a medida que se completan.
5. Se selecciona la ruta de menor distancia.
6. Se mide el tiempo de ejecución paralelo.

```python
def brute_force_parallel():
    start = time.time()
    best_distance = float('inf')
    best_route = None

    all_routes = list(itertools.permutations(CITIES))

    with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
        future_to_route = {
            executor.submit(calculate_route_distance, route): route
            for route in all_routes
        }

        for future in as_completed(future_to_route):
            distance, route_names = future.result()
            if distance is not None and distance < best_distance:
                best_distance = distance
                best_route = route_names

    elapsed = time.time() - start
    print(f"Tiempo total paralelo ({MAX_WORKERS} hilos): {elapsed:.2f}s")
    return best_route, best_distance, elapsed
```

Flujo del Sistema Distribuido (Docker Swarm)

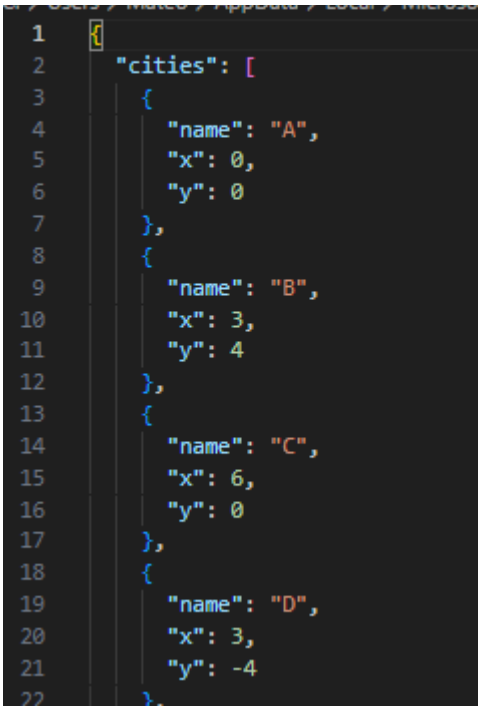
1. Se construye la imagen mediante el Dockerfile.
2. Se publica un servicio escalable en Docker Swarm con:

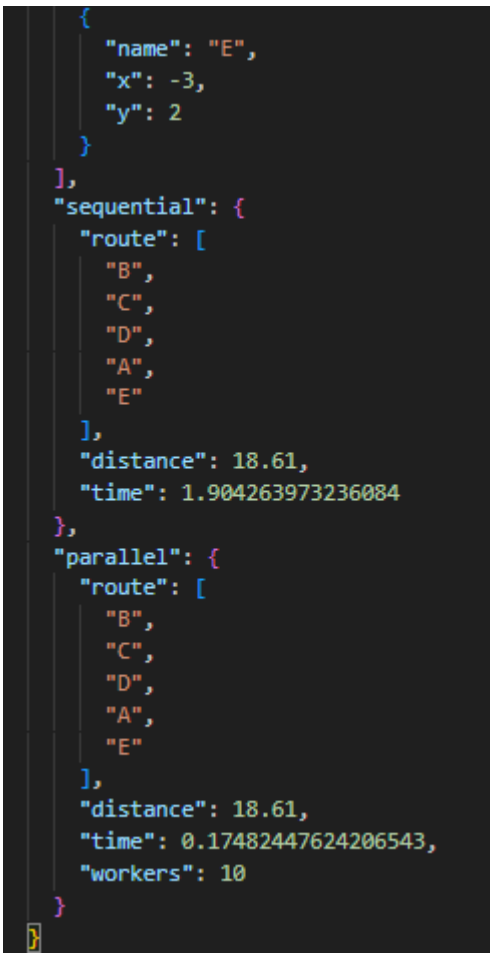
- Balanceo de carga.
- Múltiples réplicas N.
- Enrutamiento automático.

3. Cada petición del cliente es enviada a una réplica diferente.
4. Al aumentar los hilos (MAX_WORKERS), el Swarm distribuye las peticiones entre varias réplicas, esto reduce el tiempo total del procesamiento de fuerza brut

RESULTADOS

```
C:\Users\santi\OneDrive\Escritorio\HOMEWORK\HPC\client> python brute_force_client.py
============================================================
PROBLEMA DEL VIAJANTE - BÚSQUEDA POR FUERZA BRUTA
============================================================
Ciudades a visitar: 5
  - A: (0, 0)
  - B: (3, 4)
  - C: (6, 0)
  - D: (3, -4)
  - E: (-3, 2)
Total de permutaciones posibles: 120
✓ Conexión con API establecida
=== BÚSQUEDA SECUENCIAL ===
Nueva mejor ruta encontrada: A -> B -> C -> D -> E = 23.49
Nueva mejor ruta encontrada: A -> D -> C -> B -> E = 21.32
Nueva mejor ruta encontrada: A -> E -> B -> C -> D = 19.93
Nueva mejor ruta encontrada: B -> C -> D -> A -> E = 18.61
--- RESULTADOS SECUENCIALES ---
Rutas evaluadas: 120
Mejor ruta: B -> C -> D -> A -> E
Distancia óptima: 18.61
Tiempo total: 1.90 segundos
Promedio por ruta: 15.87 ms
=== BÚSQUEDA PARALELA ===
Nueva mejor ruta encontrada: A -> B -> D -> C -> E = 27.22
Nueva mejor ruta encontrada: A -> B -> C -> D -> E = 23.49
Nueva mejor ruta encontrada: A -> E -> B -> C -> D = 19.93
Nueva mejor ruta encontrada: B -> C -> D -> A -> E = 18.61
--- RESULTADOS PARALELOS ---
Rutas evaluadas: 120
Mejor ruta: B -> C -> D -> A -> E
Distancia óptima: 18.61
Tiempo total: 0.17 segundos
Promedio por ruta: 1.46 ms
Hilos usados: 10
============================================================
COMPARACIÓN DE RENDIMIENTO
============================================================
Tiempo secuencial:  1.90 segundos
Tiempo paralelo:    0.17 segundos
Speedup:            10.89x
Mejora:             90.8%
Resultados guardados en 'results.json'
```





Mejor ruta encontrada:
B → C → D → A → E
Distancia óptima:
18.61 unidades

Método	Tiempo total	Promedio de ruta	Rutas evaluadas
Secuencial	**1.9s**	**15.87ms**	**120**
Paralelo (10 hilos)	**0.17s**	**1.46ms**	**120**

ANÁLISIS DE RENDIMIENTO

1. El paralelismo no siempre garantiza la máxima aceleración.Aunque la versión paralela logra un speedup notable (de 1.9 s a 0.17 s), existen ciertos límites prácticos:

- Cada solicitud HTTP al microservicio introduce latencia de red, la cual se mantiene incluso si los hilos aumentan.
- El servidor Flask no está optimizado para alto rendimiento; atiende pocas peticiones simultáneas por proceso.

- El cliente debe serializar y deserializar JSON para cada ruta, lo que genera sobrecosto adicional.

2. La cantidad de rutas influye directamente en el speedup
Para 120 rutas (5 ciudades), el paralelismo muestra mejoras claras, pero al ser un número pequeño, el tiempo de comunicación y creación de hilos se vuelve comparable al tiempo de cómputo.
Si se aumentara el número de ciudades:
- 6 → 720 rutas
- 7 → 5040 rutas
- 8 → 40320 rutas

El beneficio del paralelismo sería mayor, porque el costo del cálculo dominaría sobre la latencia.

3. El paralelismo es más útil en tareas distribuidas que involucran espera.
Cada hilo del cliente:
- Envía la ruta por HTTP.
- Espera a que el microservicio responda

Esto es principalmente una operación de I/O (red) y no de CPU, por lo que:

- El paralelismo por multithreading funciona muy bien (a diferencia de tareas CPU-bound que chocan con el GIL).
- El cluster Docker Swarm puede balancear solicitudes entre múltiples réplicas, eliminando cuellos de botella.

A medida que se incrementan las réplicas del servicio Flask, el sistema puede atender más solicitudes simultáneas, mostrando un comportamiento cercano a un sistema distribuido real.

CONCLUSIÓN

El experimento demostró que la combinación de microservicios, contenedores y paralelismo permite acelerar significativamente la solución exhaustiva del TSP. Sin embargo, también permitió identificar que:

- El paralelismo no siempre genera mejoras proporcionales debido a la latencia de red y la sobrecarga de comunicación HTTP.
- Para problemas pequeños (como 5 ciudades), el impacto del paralelismo es limitado, pero en escenarios reales con muchos más nodos el speedup sería considerable.
- El uso de Docker Swarm facilita la escalabilidad horizontal, permitiendo distribuir la carga entre varias réplicas sin modificar el código del cliente.
- El enfoque de microservicios, aunque modular y escalable, introduce sobrecostos que no están presentes en implementaciones locales o monolíticas.