[1]

Proyecto final Sistema Operativos

Sergio Grajales Clavijo - Juan Diego Higinio Aranzazu - Josue Daniel CastaÑo Montoya - Juan Camilo Bueno

***Resumen*—Este proyecto presenta el diseño e implementación de un sistema distribuido híbrido utilizando contenedores Docker. La arquitectura integra una base de datos en la nube (simulada en una máquina virtual Linux) y microservicios locales para visualización y respaldo. Se emplearon técnicas de orquestación con Docker Compose, proxys inversos con Nginx para encapsulamiento TCP y algoritmos de concurrencia en Python para el consumo eficiente de APIs externas. Los resultados demuestran la eficacia de los volúmenes para la persistencia de datos ante fallos críticos y la capacidad de los microservicios para operar con recursos limitados.**

I. Introducción

En la industria actual, la capacidad de desplegar aplicaciones escalables y resilientes es fundamental. El objetivo de este proyecto es diseñar, desplegar y orquestar un ecosistema de microservicios distribuidos, simulando una arquitectura híbrida (Nube + Local). El sistema aborda desafíos clave como la gestión de redes definidas por software, la persistencia de datos en entornos efímeros y la implementación de concurrencia para optimizar el rendimiento del lado del cliente.

II. Marco Conceptual

Para fundamentar la solución técnica, se investigaron los siguientes conceptos clave de la tecnología de contenedores:

A. Docker Networks y Aislamiento

Docker permite crear redes definidas por el usuario (como bridge) para aislar contenedores. Esto es crucial por seguridad: impide que contenedores externos accedan a servicios sensibles (como la base de datos) si no están en la misma red. Además, Docker incluye un servidor DNS interno embebido en estas redes. Esto permite la Resolución de Nombres, donde los servicios se comunican usando sus nombres de contenedor (ej. db-master) en lugar de direcciones IP volátiles, garantizando la conectividad tras reinicios.

B. Docker Compose

Docker Compose es una herramienta para definir y ejecutar aplicaciones multi-contenedor. Funciona como un orquestador local que permite declarar la infraestructura como código (IaC) mediante un archivo YAML. Simplifica la gestión al levantar servicios, redes y volúmenes con un solo comando, asegurando que el entorno sea reproducible idénticamente en desarrollo y producción.

C. Persistencia: Volúmenes vs. Contenedores Efímeros

Los contenedores son, por naturaleza, efímeros; si se eliminan, su sistema de archivos se pierde. Para bases de datos, esto es crítico.

Volúmenes (Docker Volumes): Son gestionados por Docker y se almacenan en una parte del sistema operativo anfitrión (/var/lib/docker/volumes). Son la opción preferida para persistencia de datos críticos.

Bind Mounts: Vinculan una carpeta específica del anfitrión al contenedor.

D. Cultura DevOps y CI/CD

Los contenedores son pilares de la cultura DevOps ya que eliminan el problema de "funciona en mi máquina". Facilitan la Integración y Despliegue Continuo (CI/CD) al empaquetar la aplicación con todas sus dependencias, permitiendo pruebas y despliegues rápidos, consistentes y automatizados en cualquier entorno.

E. Orquestación: Kubernetes (K8s)

Aunque este proyecto usa Compose, Kubernetes es el estándar para orquestación a gran escala. A diferencia de Docker (que es el runtime), K8s gestiona clústeres de contenedores. Resuelve problemas complejos como el Auto-scaling (aumentar réplicas según demanda) y el Auto-healing (reiniciar contenedores fallidos automáticamente), capacidades que Docker por sí solo no ofrece nativamente a ese nivel.

III. METODOLOGÍA

Se diseñó una arquitectura híbrida compuesta por dos nodos lógicos:

Entorno Nube (Servidor): Implementado sobre una Máquina Virtual Linux Mint. Aloja la capa de persistencia (MySQL) y seguridad (Nginx).

Entorno Local (Cliente): Implementado en Windows con Docker Desktop. Aloja la lógica de negocio (Frontend), simulación de sensores y copias de seguridad (Backup Worker).

Las tecnologías seleccionadas fueron Docker para la contenerización, Nginx con el módulo stream para proxy TCP, MySQL 8.0 para la base de datos y Python (Flask) para el desarrollo de los microservicios.

IV. IMPLEMENTACIÓN TÉCNICA

A. Configuración de Redes y Volúmenes

Se utilizó el archivo docker-compose.yml para orquestar la infraestructura.

Redes: Se definieron redes tipo bridge (backend-net en nube y local-net en local) para aislar el tráfico.

Volúmenes: Se declaró el volumen

[1*] Revista Argentina de Trabajos Estudiantiles. Patrocinada por la IEEE.

mysql_data:/var/lib/mysql en el servicio de base de datos. Esto asegura que, aunque el contenedor db-master sea eliminado, la información persista en el disco del anfitrión.

B. Dockerfiles y Optimizaciones

Para los microservicios personalizados (Frontend, Sensor, Backup), se utilizaron imágenes base ligeras (python:3.9-slim) para reducir el tamaño final y la superficie de ataque, optimizando el uso de recursos frente a imágenes completas.

C. Proxy Inverso con Nginx

En el entorno nube, Nginx no se configuró como servidor web tradicional, sino como un Proxy de Flujo (Stream Proxy).

Configuración: Redirige el tráfico TCP del puerto externo 33060 hacia el puerto interno 3306 del contenedor db-master. Esto añade una capa de abstracción y seguridad, ocultando la base de datos directa.

D. Script de Backup y Sincronización

El servicio backup-worker ejecuta un script en Python que realiza un ciclo infinito (bucle while True).

Lógica: Cada 600 segundos (configurable), conecta a la base de datos remota, ejecuta una consulta SELECT formateando las fechas y guarda el resultado en un archivo JSON local con timestamp, garantizando un historial de versiones de los datos.

E. Concurrencia en el Frontend

Para cumplir con el requisito de consultar una API externa sin bloquear la interfaz de usuario, se implementó concurrencia utilizando ThreadPoolExecutor de Python.

Fragmento de código clave (app.py):

```
# Uso de ThreadPoolExecutor para operaciones I/O Bound
def get_external_data():
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=2)
    # La consulta a la API se ejecuta en un hilo separado
    future = executor.submit(fetch_weather_api, selected_city)
    data = future.result() # Espera no bloqueante
    return jsonify(data)
```

Esto permite que el servidor web maneje múltiples solicitudes mientras espera la respuesta de la API de clima (Open-Meteo).

V. RESULTADOS Y PRUEBAS

A. Rendimiento bajo recursos limitados

El contenedor frontend-app se desplegó con un límite estricto de 2 vCPU y 2 GB de RAM. Durante las pruebas de carga (actualización simultánea de paneles cada 15 segundos), el servicio se mantuvo estable sin latencia perceptible, validando la eficiencia de la imagen slim y el manejo de hilos de Flask.

B. Verificación de Persistencia y Tolerancia a Fallos

Se realizó una prueba de "caos" eliminando forzosamente los contenedores de la nube (docker rm -f).

Observación: El Frontend mostró un error de conexión controlado (degradación elegante), pero siguió mostrando datos de la API del clima.

Recuperación: Al regenerar los contenedores (docker-compose up -d), se verificó que el historial de datos (incluyendo registros antiguos) reapareció intacto, confirmando el correcto funcionamiento del volumen mysql_data.

B. Automatización en Tiempo Real

Mediante el microservicio sensor-simulator, se logró la inyección automática de datos climáticos reales cada 15 segundos, visualizándose cambios dinámicos en el Frontend sin intervención manual.

(Aquí insertas tus capturas de pantalla: Panel funcionando, JSON de backup generado, y la terminal mostrando la persistencia tras el reinicio).

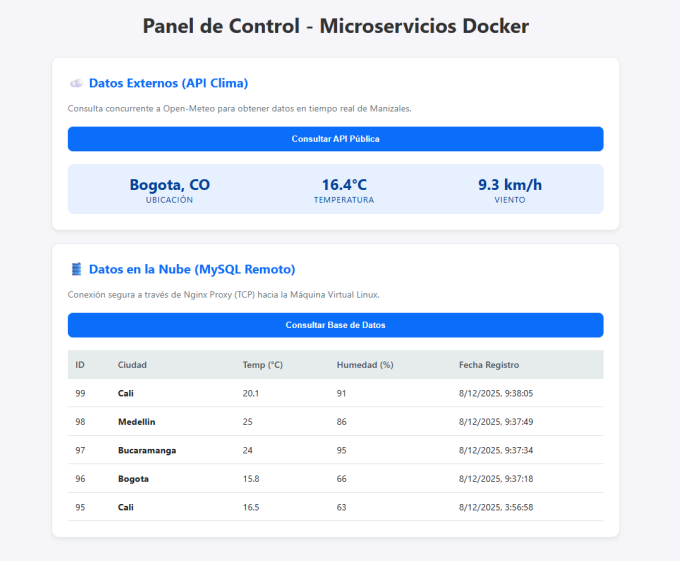

Fig. 1.Previsualización de los datos entregados por front en la dirección local http://localhost:5000/ (Dirección definida para puerto de ejecución local).

```
    {
        "id": 99,
        "ciudad": "Cali",
        "temperatura": 20.1,
        "humedad": 91,
        "fecha_registro": "2025-12-08 09:38:05"
    },
    {
        "id": 100,
        "ciudad": "Medellin",
        "temperatura": 26.0,
        "humedad": 63,
        "fecha_registro": "2025-12-08 10:41:16"
    },
    {
        "id": 101,
        "ciudad": "Manizales",
        "temperatura": 25.8,
        "humedad": 42,
        "fecha_registro": "2025-12-08 10:41:32"
    },
    {
        "id": 102,
        "ciudad": "Manizales",
        "temperatura": 25.8,
        "humedad": 48,
        "fecha_registro": "2025-12-08 10:41:48"
    },
    {
        "id": 103,
        "ciudad": "Manizales",
        "temperatura": 25.8,
        "humedad": 42,
        "fecha_registro": "2025-12-08 10:42:04"
    },
    {
        "id": 104,
        "ciudad": "Medellin",
        "temperatura": 26.0,
        "humedad": 42,
        "fecha_registro": "2025-12-08 10:42:20"
    }
]
```


Fig. 2.Datos almacenado automáticamente por medio de un JSON el cual fue programado para almacenar los registros de tiempo que se van obteniendo cada un (1) minuto.

```
sergio@sergio-VirtualBox:~/proyecto-distribuido/database$ docker-compose down
[+] Running 3/3
 ✔ Container nginx-proxy          Removed                                0.7s
 ✔ Container db-master            Removed                                1.8s
 ✔ Network database_backend-net   Removed                                0.3s
sergio@sergio-VirtualBox:~/proyecto-distribuido/database$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Fig. 3.Comando `docker-compose down` Con el cual damos de baja la nube y con esto probamos la persistencia.

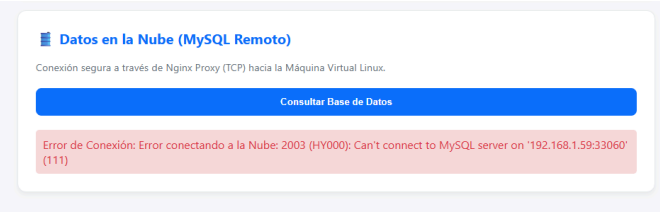

Fig. 4.Evidencia de que el sistema continúa sus intentos automáticos por conectarse a la nube y extraer los datos.

```
sergio@sergio-VirtualBox:~/proyecto-distribuido/database$ docker-compose up -d
[+] Running 3/3
 ✔ Network database_backend-net  Created                                 0.2s
 ✔ Container db-master           Started                                 0.2s
 ✔ Container nginx-proxy         Started                                 0.1s
sergio@sergio-VirtualBox:~/proyecto-distribuido/database$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS
       PORTS                                                  NAMES
e3f0b4ed7cea   nginx:latest   "/docker-entrypoint.…"   30 seconds ago   Up 28 se
conds   80/tcp, 0.0.0.0:33060->3306/tcp, [::]:33060->3306/tcp   nginx-proxy
45aa625acc2f   mysql:8.0      "docker-entrypoint.s…"   30 seconds ago   Up 29 se
conds   3306/tcp, 33060/tcp                                    db-master
```

Fig. 5.Ejecución del comando `docker-compose up -d` con el cual levantamos nuevamente el servicio de la nube.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 53
Server version: 8.0.44 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> SELECT * FROM mediciones ORDER BY id DESC LIMIT 15;
+-----+------------+-------------+---------+---------------------+
| id  | ciudad     | temperatura | humedad | fecha_registro      |
+-----+------------+-------------+---------+---------------------+
| 138 | Cartagena  |        28.4 |      74 | 2025-12-08 10:55:14 |
| 137 | Medellin   |        26.3 |      43 | 2025-12-08 10:54:58 |
| 136 | Cartagena  |        28.4 |      50 | 2025-12-08 10:54:42 |
| 135 | Medellin   |        26.3 |      79 | 2025-12-08 10:54:26 |
| 134 | Manizales  |        26.1 |      45 | 2025-12-08 10:54:10 |
| 133 | Pereira    |        24.9 |      74 | 2025-12-08 10:53:54 |
| 132 | Bogota     |        16.5 |      56 | 2025-12-08 10:53:38 |
| 131 | Bogota     |        16.5 |      49 | 2025-12-08 10:53:22 |
| 130 | Bogota     |        16.5 |      69 | 2025-12-08 10:53:06 |
| 129 | Cali       |        21.8 |      64 | 2025-12-08 10:52:50 |
| 128 | Cali       |        21.8 |      57 | 2025-12-08 10:52:34 |
| 127 | Bogota     |        16.5 |      81 | 2025-12-08 10:52:18 |
| 126 | Bucaramanga |        24.5 |      46 | 2025-12-08 10:52:02 |
| 125 | Bucaramanga |        24.5 |      40 | 2025-12-08 10:51:46 |
| 124 | Cali       |        21.8 |      91 | 2025-12-08 10:51:30 |
+-----+------------+-------------+---------+---------------------+
15 rows in set (0.01 sec)
```

Fig. 6.Al consultar la base de datos dentro de ssh podemos ver la persistencia de los datos, ya que guardo todos los registros anteriores y ahora continuará almacenando los nuevos y registrándose dentro de los backups periódicos.

VI. CONCLUSIONES

1. Contenedores vs. VMs: La implementación demostró que los contenedores son significativamente más ligeros y rápidos de iniciar que las máquinas virtuales tradicionales, permitiendo desplegar una arquitectura compleja de 5 nodos en segundos.

2. Ventajas de la Orquestación: Docker Compose fue vital para gestionar la dependencia entre servicios (ej. que Nginx espere a la BD) y simplificar la configuración de redes, reduciendo errores humanos frente a comandos manuales.

3. Resiliencia: El desacoplamiento de servicios permitió que la aplicación web sobreviviera a la caída de la base de datos, una característica esencial en sistemas distribuidos modernos.

REFERENCIAS

[1] Documentación Oficial de Docker. "Networking in Compose".
[2] Nginx. "TCP and UDP Load Balancing".
[3] Python Software Foundation. "concurrent.futures — Launching parallel tasks".
[4] Open-Meteo. "Weather Forecast API".