UNIVERSIDAD LATINA DE COSTA RICA

Proyecto Final

Estudiante:

Javier Chinchilla Lugo

Curso:

Sistemas Oprativos II

Profesor:

CARLOS ANDRES MENDEZ RODRIGUEZ

Diciembre, 2024

Sede San Pedro

Definición del Tema y Objetivos

Tema del Proyecto

"Implementación Básica de Contenedores Docker para Balanceo de Carga en Aplicaciones Web"

Objetivo General

Explorar el uso de Docker para implementar una solución básica de balanceo de carga en aplicaciones web, enfocándose en su configuración y pruebas de rendimiento iniciales.

Objetivos Específicos Simplificados

1. Configurar un entorno con Docker que aloje una aplicación web simple y permita el uso de múltiples contenedores.
2. Implementar balanceo de carga utilizando Docker Compose con un servicio como HAProxy.
3. Realizar pruebas básicas de rendimiento para analizar el impacto del balanceo de carga.
4. Documentar los pasos y resultados obtenidos, proponiendo mejoras simples para el entorno.

Planificación y Metodología

Plan de Trabajo

Cronograma Detallado por Semanas

Semana	Actividades Principales	Entregables
13	- Revisión bibliográfica de literatura relevante. - Redacción de una primera versión del marco teórico.	Borrador del marco teórico y referencias seleccionadas.
14	- Configuración de un entorno Docker con balanceo de carga. - Ejecución de pruebas de carga.	Resultados preliminares de pruebas de rendimiento.
15	- Presentación y redacción del informe final. Informe final y presentación del proyecto.	- Exposición y entrega del proyecto.

Definición de la Metodología de Investigación

Tipo de Investigación

- **Exploratoria:**

Busca comprender las capacidades de Docker para balancear la carga en aplicaciones web mediante la implementación y prueba de un entorno práctico.

Enfoque Metodológico

Diseño Experimental:

- Configuración de un entorno de prueba con instancias de una aplicación web sencilla en contenedores Docker.
- Implementación de un servicio de balanceo de carga con HAProxy.
- Realización de pruebas de rendimiento para evaluar el balanceo.

Fases de la Investigación:

- **Revisión inicial:** Identificar y estudiar conceptos clave sobre contenedores y balanceo de carga.
- **Implementación:** Configurar el entorno y realizar pruebas controladas.
- **Evaluación:** Analizar los resultados para identificar mejoras o ajustes necesarios.

Justificación de la Metodología

La elección de esta metodología permite abordar de manera práctica el uso de Docker y su integración con herramientas de balanceo de carga. Las pruebas controladas ofrecerán datos concretos para validar el impacto de las configuraciones realizadas.

Identificación de Herramientas y Recursos Necesarios

Herramientas Técnicas

1. **Docker:** Creación y administración de contenedores.
2. **Docker Compose:** Configuración de múltiples contenedores.
3. **HAProxy:** Implementar balanceo de carga entre los contenedores.

4. **Apache Benchmark (ab):** Herramienta para realizar pruebas de carga sobre la aplicación web.

Recursos de Hardware y Software

1. Computadora con Sistema operativo Linux o Windows.

Material de Referencia

1. Documentación oficial de Docker y Docker Compose.
2. Manual de configuración de HAProxy.
3. Artículos académicos, IEEE y tutoriales sobre balanceo de carga en entornos Docker.

Marco Teorico

Contenedores y Docker:

Mediante el uso de la herramienta Docker se ha logrado ir transformando el desarrollo, despliegue y gestión de aplicaciones mediante contenedores. Su adaptabilidad le permite sobresalir en varios roles: como un proyecto de código abierto y como un conjunto de herramientas simplificando la creación y administración de contenedores. Este enfoque completo ha sido clave para impulsar los contenedores como una opción rápida y eficaz en el desarrollo de aplicaciones.

El operatividad de Docker se fundamenta en el núcleo de Linux, utilizando propiedades como los nombres de espacio y los grupos de control para ejecutar aplicaciones y procesos de forma aislada. Esto permite a las organizaciones maximizar la utilización de su infraestructura sin

afectar la seguridad. Gracias a su sistema de imágenes, Docker facilita el procedimiento de compartir aplicaciones y sus dependencias, automatizando su implementación en diversos entornos.

Entre los beneficios más sobresalientes de Docker se incluyen su modularidad, la gestión de versiones, la rápida puesta en marcha y la simplicidad para restaurar sistemas. Estas cualidades lo hacen una herramienta perfecta para contextos actuales que implementan metodologías como la integración y entrega continuas (CI/CD) o para aplicaciones en la nube que demandan escalabilidad y adaptabilidad. Sin embargo, también muestra algunas limitaciones, como el manejo de contenedores voluminosos o posibles falencias asociadas al demonio Docker y su relación con el núcleo del sistema operativo anfitrión.

En este proyecto, se utiliza Docker para llevar a cabo una solución de balanceo de carga para aplicaciones web. Este método no solo muestra su efectividad en la regulación del tráfico, sino que también constituye un fundamento para evaluar su influencia en el rendimiento y examinar posibles optimizaciones en situaciones más complejas.

Balanceo de carga en aplicaciones web:

Lo que se encarga el balanceo de carga es distribuir solicitudes de clientes entre varios servidores para mejorar la disponibilidad, el rendimiento y evitar los fallos. Hay dos categorías principales de balanceo de carga las cuales serian:

Hardware-based: Utiliza dispositivos dedicados, costosos pero muy eficientes.

Software-based: Implementaciones flexibles y económicas que pueden instalarse en servidores estándar, pero que consumen recursos de la máquina.

Los algoritmos comunes incluyen:

- Round Robin: Asigna solicitudes a los servidores en orden secuencial.
- Least Connections: Asigna al servidor con menos conexiones activas.
- IP Hash: Utiliza una función cifrado hash del IP del cliente para así garantizar que las solicitudes vayan al mismo servidor.

HAProxy

Es una solución de balanceo de carga de código abierto (gratis) que soporta balanceo tanto a nivel de red como a nivel de aplicación:

- NLB (Network Load Balancer): Basado en información de transporte (como TCP).
- ALB (Application Load Balancer): Analiza datos a nivel de aplicación, como encabezados HTTP.

Ventajas de HAProxy:

- Alta capacidad de procesamiento.
- Algoritmos avanzados como Weighted Round Robin, Least Connection y Source Hashing.
- Configuración ajustable para maximizar el rendimiento.

Aspectos de configuración y optimización

El artículo de HAProxy (2022) resalta la importancia de optimizar configuraciones como:

- Maxconn: Define el número máximo de conexiones simultáneas.
- Nbthread: Permite paralelizar tareas en múltiples hilos.
- Compresión y polling: Mejoran el uso de CPU y la eficiencia de red.

Impacto del balanceo de carga

Según los estudios:

1. El algoritmo Round Robin funciona bien en entornos homogéneos, pero en heterogéneos, Weighted Least Connections e IP Hash logran un mejor uso de recursos y tiempos de respuesta.
2. HAProxy puede ajustarse para escenarios específicos, como aplicaciones web de alto tráfico o sistemas con recursos limitados.
3.

Uso de Docker para balanceo de carga:

Docker es sumamente flexible y portable lo cual causa que sea sumamente utilizado para implementar microservicios y aplicaciones en contenedores. Este genera archivos YAML, lo cual simplifica la orquestación de múltiples contenedores.

Implementación del balanceo de carga

1. Definición de servicios: Docker Compose da la oportunidad de crear múltiples contenedores en un archivo de docker-compose.yml básicamente los que crea el, cada uno con su imagen, puertos y volúmenes, así diferenciando uno del otro.

2. Configuración de balanceadores: A Compose se puede integrar un balanceador de carga como HAProxy que hablamos anteriormente, utilizándolo como servicio adicional.

3. Redes internas: Docker Compose permite crear redes internas que conectan los contenedores.

Beneficios

Facilidad de configuración: Gracias a Docker Compose al crear los entornos multi-contenedor se simplifica la configuración.

Escalabilidad: Permite escalar servicios con un solo comando (docker-compose up --scale servicio=n).

Portabilidad: Las configuraciones son reproducibles en cualquier sistema con Docker.

Integración con herramientas de CI/CD: Compatible con pipelines para automatizar despliegues.

Beneficios en entornos simples

- Simplicidad: Ideal para proyectos pequeños con pocos servicios.
- Eficiencia: Reduce la sobrecarga manual de configuración.
- Bajo costo: No requiere hardware especializado.
- Familiaridad: Herramientas como Docker son conocidas por la mayoría de los desarrolladores.

Limitaciones en entornos simples

Complejidad innecesaria: En aplicaciones monolíticas, puede agregar complejidad sin beneficios tangibles.

Dependencias adicionales: Necesita herramientas como Docker Compose y un balanceador.

Rendimiento limitado: En configuraciones básicas, el balanceador puede convertirse en un cuello de botella.

Problemas y Limitaciones Identificadas en Investigaciones

Inexactitudes y Complejidad en las Especificaciones de Docker Compose

Aumenta bastante la complejidad cuando se amplía el número de servicios o la variedad en las configuraciones, complicando todo el proceso, la depuración y en como se entienden las dependencias entre los servicios.

Además opciones avanzadas, como la administración de volúmenes y la asignación de puertos, pueden ser confusas para usuarios nuevos o usuarios que no tienen experiencia.

Más de un cuarto de los proyectos analizados que emplean Docker Compose podrían ser implementados sin esta herramienta, lo que indica un uso superfluo que incrementa la complejidad.

Dificultades en la Elaboración de Especificaciones IaC

El desarrollo de especificaciones como Dockerfiles y archivos YAML de Docker Compose es susceptible a errores debido a métodos de prueba y error y a la escasez de herramientas de apoyo.

Una proporción significativa de los Dockerfiles localizados en repositorios públicos no se construyen adecuadamente por problemas de configuración y calidad, como la ausencia de etiquetas de mantenimiento o versiones específicas para imágenes base.

Ajuste y Mejora de HAProxy

Aunque HAProxy es muy configurable, ajustar parámetros ciertos parámetros erróneamente puede causar grandes errores, problemas en el rendimiento o comportamientos extraños. Parámetros como la cantidad de conexiones máximas, el número de hilos y procesos puede ser complejo.

Los algoritmos de distribución de carga, aunque son eficaces, enfrentan dificultades en entornos heterogéneos, donde los pesos otorgados a los servidores necesitan ajustarse con precisión para prevenir desigualdades importantes.

Efecto del Contexto de Uso

La gran parte de los proyectos examinados empleaban únicamente funciones básicas de Docker Compose, omitiendo configuraciones avanzadas como la seguridad o el monitoreo, lo que sugiere una posible subutilización de las capacidades de la herramienta.Inexactitudes

En contextos heterogéneos, los algoritmos sencillos como Round Robin no son adecuados, requiriendo ajustes más sofisticados como pesos dinámicos para un uso óptimo

Bibliografía

DOCKER

Anderson, C. (2015). Docker [Software engineering]. *IEEE Software, 32*(3), 102–c3. https://doi.org/10.1109/MS.2015.62

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal, 2014*(239), 2. Recuperado de https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf

LOAD BALANCING

Rawls, C., & Salehi, M. A. (2022). Load Balancer Tuning: Comparative Analysis of HAProxy Load Balancing Methods. *High Performance Cloud Computing Lab, University of Louisiana*. Recuperado de https://arxiv.org/pdf/2212.14198

Ibrahim, I. M., Ameen, S. Y., Yasin, H. M., Omar, N., Kak, S. F., Rashid, Z. N., Salih, A. A., Salim, N. O. M., & Ahmed, D. M. (2021). Web Server Performance Improvement Using Dynamic Load Balancing Techniques: A Review. *Asian Journal of Research in Computer Science, 10*(1), 47–62. https://doi.org/10.9734/ajrcos/2021/v10i130234

DOCKER LOAD BALANCING

Reis, D., Piedade, B., Correia, F. F., Dias, J. P., & Aguiar, A. (2022). Developing Docker and Docker-Compose Specifications: A Developers' Survey. *IEEE Access, 10*, 2318–2329. https://doi.org/10.1109/ACCESS.2021.3137671

Piedade, B., Dias, J. P., & Correia, F. F. (2020). An Empirical Study on Visual Programming Docker Compose Configurations. *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion)*, October 18–23, 2020. https://doi.org/10.1145/3417990.3420194

CONTAINERS

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal, 2014*(239), 2. Recuperado de https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf

Desarrollo y Análisis de Resultados:

Este proyecto consistió en configurar un entorno que aloje una aplicación web simple utilizando Docker y permita el balanceo de carga con múltiples contenedores. Se implementaron los pasos necesarios para levantar los servicios, ejecutar pruebas de rendimiento y proponer mejoras para el entorno configurado.

Configurar un entorno con Docker que aloje una aplicación web simple y permita el uso de múltiples contenedores en Ubuntu

Actualizar paquetes del sistema:

```
sudo apt update
```

```
sudo apt upgrade
```

Instalar Docker:

```
sudo apt install docker.io -y
```

Habilitar y verificar el servicio Docker:

```
sudo systemctl start docker
```

```
sudo systemctl enable docker
```

```
sudo systemctl status docker
```

Instalar Docker Compose:

```
sudo apt install docker-compose -y
```

Verificar que Docker y Docker Compose están instalados:

```
docker --version
```

```
docker-compose --version
```

Agregar permisos a tu usuario para usar Docker sin sudo:

```
sudo usermod -aG docker $USER
```

Reinicia sesión para que los cambios surtan efecto:

```
Exit
```

Inicio sesión nuevamente.

Preparar los archivos del proyecto

Sigue los pasos para crear los archivos directamente en la terminal de Ubuntu.

Crear el directorio del proyecto:

```
mkdir docker-loadbalancer
```

```
cd docker-loadbalancer
```

Crear los archivos requeridos:

Crear el archivo app.py:

```
nano app.py
```

Codigo:

```
from flask import Flask
import os

app = Flask(__name__)

@app.route("/")
def hello():
    return f"Hello from container {os.getenv('HOSTNAME')}!"

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Este archivo define una aplicación web simple usando Flask. La aplicación responde a las solicitudes en la ruta principal mostrando un mensaje que incluye el nombre del contenedor (variable `HOSTNAME`).

Crear el archivo Dockerfile:

nano Dockerfile

Codigo:

```
FROM python:3.9-slim

WORKDIR /app

COPY app.py /app

RUN pip install flask

CMD ["python", "app.py"]
```

El Dockerfile define la imagen del contenedor. Usa una imagen base de Python, copia el código de la aplicación, instala Flask y especifica el comando para ejecutar la aplicación.

Crear el archivo docker-compose.yml:

nano docker-compose.yml

Codigo:

```
version: '3.8'
```

```
services:

app:

        image: myapp

        build:

                context: .

        deploy:

                replicas: 3

        environment:

                - HOSTNAME=${HOSTNAME}

        haproxy:

                image: haproxy:latest

                ports:

                        - "80:80"

                volumes:

                        - ./haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg
```

Este archivo configura los servicios de Docker Compose. Define un servicio `app` que utiliza tres réplicas de la aplicación web y un servicio `haproxy` que gestiona el balanceo de carga.

Crear el archivo de configuración de HAProxy:

nano haproxy.cfg

Codigo:

```
global
    log stdout format raw local0
defaults
    log global
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms
frontend http_front
    bind *:80
    default_backend http_back
backend http_back
```

```
    balance roundrobin
    server app1 app:5000 check
    server app2 app:5000 check
    server app3 app:5000 check
```

Este archivo configura HAProxy para actuar como balanceador de carga. Define un frontend que escucha en el puerto 80 y un backend que distribuye las solicitudes entre tres servidores usando el algoritmo `roundrobin`.

Ejecutar los contenedores en Ubuntu

Construir y ejecutar con Docker Compose:

Levantar todos los servicios:

```
docker-compose up --build
```

```
Successfully built 86ceba9ba471
Successfully tagged myapp:latest
Stopping and removing docker-loadbalancer_app_4 ... done
Stopping and removing docker-loadbalancer_app_5 ... done
Starting docker-loadbalancer_haproxy_1          ... done
Starting docker-loadbalancer_app_1              ... done
Starting docker-loadbalancer_app_2              ... done
Starting docker-loadbalancer_app_3              ... done
Attaching to docker-loadbalancer_haproxy_1, docker-loadbalancer_app_1, docker-lo
er-loadbalancer_app_3
haproxy_1  | [NOTICE]   (1) : Initializing new worker (7)
haproxy_1  | [NOTICE]   (1) : Loading success.
app_1      |  * Serving Flask app 'app'
app_1      |  * Debug mode: off
app_1      | WARNING: This is a development server. Do not use it in a productio
oduction WSGI server instead.
app_1      |  * Running on all addresses (0.0.0.0)
app_1      |  * Running on http://127.0.0.1:5000
app_1      |  * Running on http://172.20.0.3:5000
app_1      | Press CTRL+C to quit
app_3      |  * Serving Flask app 'app'
app_3      |  * Debug mode: off
app_3      | WARNING: This is a development server. Do not use it in a productio
oduction WSGI server instead.
app_3      |  * Running on all addresses (0.0.0.0)
app_3      |  * Running on http://127.0.0.1:5000
app_3      |  * Running on http://172.20.0.4:5000
app_3      | Press CTRL+C to quit
app_2      |  * Serving Flask app 'app'
app_2      |  * Debug mode: off
app_2      | WARNING: This is a development server. Do not use it in a productio
oduction WSGI server instead.
app_2      |  * Running on all addresses (0.0.0.0)
app_2      |  * Running on http://127.0.0.1:5000
app_2      |  * Running on http://172.20.0.5:5000
app_2      | Press CTRL+C to quit
```

Verifica que todo está funcionando: curl http://localhost

```
jchinchilla@jchinchillaserver:~/docker-loadbalancer$ curl http://localhost
Hello from container !jchinchilla@jchinchillaserver:~/docker-loadbalancer$
```

Escalar réplicas manualmente:

docker-compose up --scale app=5

---

4. Realizar pruebas de rendimiento

Instalar Apache Benchmark o Siege:

Instalar Siege

sudo apt install siege

Ejecutar una prueba:

siege -c10 -r10 http://localhost/

Simula 100 solicitudes con 10 solicitudes concurrentes.

```
New configuration template added to /home/jchinchilla/.siege
Run siege -C to view the current settings in that file

{       "transactions":                        100,
        "availability":                     100.00,
        "elapsed_time":                       0.34,
        "data_transferred":                   0.00,
        "response_time":                      0.03,
        "transaction_rate":                 294.12,
        "throughput":                         0.01,
        "concurrency":                        9.44,
        "successful_transactions":             100,
        "failed_transactions":                   0,
        "longest_transaction":                0.20,
        "shortest_transaction":               0.00
}
jchinchilla@jchinchillaserver:~/docker-loadbalancer$
```

Resultados obtenidos:

- Transacciones: 100
- Disponibilidad: 100%
- Tiempo de respuesta promedio: 0.03 segundos
- Transacciones fall
- idas: 0

50 usuarios, 10 repeticiones

```
jchinchilla@jchinchillaserver:~/docker-loadbalancer$ siege -c50 -r10 http://localhost/

{       "transactions":                        500,
        "availability":                      100.00,
        "elapsed_time":                        0.78,
        "data_transferred":                    0.01,
        "response_time":                       0.07,
        "transaction_rate":                  641.03,
        "throughput":                          0.01,
        "concurrency":                        44.38,
        "successful_transactions":             500,
        "failed_transactions":                   0,
        "longest_transaction":                 0.14,
        "shortest_transaction":                0.00
}
```

Resultados obtenidos:

- Transacciones: 500
- Disponibilidad: 100%
- Tiempo de respuesta promedio: 0.07 segundos

- Transacciones fallidas: 0

MAS PRUEBAS

siege -c50 -r10 http://localhost/

Propuestas de Mejora

1. **Configurar un Servidor WSGI para Producción:** El servidor Flask utilizado es para desarrollo. Implementar un servidor WSGI como Gunicorn o uWSGI.
2. **Automatizar Escalado:** Configurar un sistema de orquestación como Kubernetes o Docker Swarm para gestionar automáticamente el número de réplicas según la carga.
3. **Mejorar la Seguridad de HAProxy:**
    a. Configurar HTTPS utilizando certificados SSL.
    b. Restringir accesos no autorizados mediante reglas ACL en HAProxy.
4. **Monitoreo y Logging:**
    a. Implementar herramientas como Prometheus y Grafana para monitorear el rendimiento.
    b. Configurar logs detallados en HAProxy para analizar tráfico y detectar posibles problemas.
5. **Pruebas de Carga Adicionales:**
    a. Realizar pruebas más extensivas con herramientas como Apache JMeter.
    b. Analizar el rendimiento bajo diferentes niveles de concurrencia.