Facultad de Ingeniería
Escuela de computación

INVESTIGACIÓN APLICADA 1

- **Integrantes:**

Cindy Valeria González León	**GL221827**
Mariana Michelle Ponce Rivera	**PR230356**
Marlon Josue Ramos Galo	**RG231707**
Erick Salvador Chinchilla Chiquillo	**CC230876**

- **Asignatura:** **Lenguajes Interpretados en el Servidor**
- **Docente**: **Ing. Guillermo Carderón**
- **Tema:** **Estrategias para escalar aplicaciones web**

Parte I

1) Estrategias de escalabilidad para aplicaciones web

Las aplicaciones web desarrolladas con lenguajes interpretados en el servidor como lo es Python php o node.js pueden enfrentar algunos problemas de rendimiento. Para poder tener una mejor experiencia y poder garantizar un mejor funcionamiento. Es importante implementar estrategias de escalabilidad como el balanceo de carga ,caching bases de datos , arquitecturas modernas etc esto puede mejorar la capacidad de respuesta de las aplicaciones ante lo que es el crecimiento del tráfico.

Tipos de Estabilidad

- Escabilidad Vertical
- Escabilidad Horizontal

Estrategias para Mejorar la Escalabilidad

Algunas de ellas son:

- Balanceo de carga
- Carching de datos
- La Optimización de consultas a la Base de datos
- Uso de teconologias asincrónicas y eventos
- Microservicios y arquitecturas basadas en contenedores
- Uso de CDNS (content Delivery Networks)

2) Horizontalización y balanceo de carga

Horizontalización:

Es un concepto que se utiliza en tecnología y en lo que es la arquitectura de Sistemas habitualmente para describir la estrategia de Escalar un Sistema añadiendo más unidades de un mismo tipo.

También es la estrategia preferida en sistemas distribuidos y en la nube ya que permiten:

- La mayor disponibilidad.
- Crecimiento dinámico sin ninguna interrupción

Algunas Ventajas son:

- La Alta disponibilidad y la tolerancia a fallos

- Mantenimiento sin Ninguna interrupción
- La Distribución de carga eficiente
- Costos más Flexibles (o más considerandos)

Balanceo de carga

El balanceo de carga es una técnica que es mayor mente utiliza en informática y en redes que distribuyen la carga de trabajo entre varios servidores, dispositivo, discos u otros recursos. Esto se hace para evitar que un solo servidor o ubicación soporte o eviten demasiada sobrecarga.

El balanceo de carga mejora:

- El rendimiento
- La capacidad de respuesta
- La disponibilidad de las aplicaciones.

Algunas Ventajas del Balanceo de Carga son:

- Reduce los tiempos de respuesta
- Aumenta la seguridad contra fallos y caídas
- Garantiza que una página siga funcionando, aunque el servidor deje de estar disponible
- Facilita el mantenimiento del sistema de servidores

3) Cómo la replicación de instancias y el uso de balanceadores de carga mejoran la disponibilidad y el rendimiento de las aplicaciones.

Fundamentalmente la replicación de instancias y el uso de balanceadores de carga Son herramientas muy beneficiosas que ayudan a mejorar la disponibilidad y el rendimiento de las aplicaciones garantizando un servicio eficiente y así mejorado la experiencia del usuario y distribuyendo la carga de trabajo.

La replicación de instancias: es la capacidad de crear copias de una instancia de base de datos o de una máquina virtual (esto mejora la disponibilidad y el rendimiento).

- Crea copias de datos en diferentes ubicaciones de la red.
- Si un nodo falla, otros nodos con datos replicados pueden seguir con su funcionamiento
- Minimiza el tiempo de inactividad y reduce el riesgo de pérdida de datos
- Permite que el tráfico de usuarios sea repartido entre diferentes instancias esto beneficia o nos evita que un servidor no vaya a sufrir un sobrecargo

Balanceadores de carga: Es un dispositivo que reparte el tráfico entre varios servidores para evitar que se sobrecarguen Esto mejora el rendimiento, la capacidad de respuesta y la disponibilidad de las aplicaciones.

- Se asegura de que las solicitudes se han enviadas a instancias con una menor carga
- Al momento del fallo de alguna instancia el balanceador es el encargado de redirigir las respectivas solicitudes hacia la instancia que sigue con su funcionamiento
- Distribuyen la carga de trabajo
- Permiten agregar y eliminar recursos sin interrumpir el flujo general de solicitudes a las aplicaciones

4) Analizar los contenedores (Docker)

Los contenedores Docker son entornos que permiten ejecutar aplicaciones y servicios de software de forma independiente, se caracterizan por ser ligeras y portátiles ya que permiten ejecutar aplicaciones de manera rápida y fiable.

1. Aislamiento

Los contenedores Docker proporcionan un entorno aislado para la ejecución de aplicaciones, lo que significa que cada contenedor tiene su propio sistema de archivos, procesos y dependencias. Este aislamiento ofrece varias ventajas:

- Evita conflictos de dependencias: Cada contenedor incluye todas las bibliotecas y archivos necesarios para su funcionamiento, eliminando incompatibilidades con otros programas en el mismo sistema.
- Mayor seguridad: Un contenedor comprometido no afecta directamente al sistema anfitrión ni a otros contenedores, reduciendo el riesgo de ataques.
- Consistencia en entornos de desarrollo y producción: Se minimizan los problemas causados por diferencias en configuraciones de servidores y entornos de ejecución.

Aunque los contenedores ofrecen un aislamiento eficaz, comparten el mismo kernel del sistema operativo anfitrión, por lo que no son tan seguros como las máquinas virtuales en ciertos escenarios.

2. Portabilidad

Docker permite empaquetar aplicaciones junto con todas sus dependencias en una imagen estándar, lo que garantiza su ejecución en cualquier sistema que tenga Docker instalado. Esto ofrece beneficios como:

- Compatibilidad entre diferentes entornos: Las aplicaciones pueden ejecutarse de manera idéntica en entornos de desarrollo, prueba y producción sin necesidad de ajustes adicionales.

- Facilidad de migración: Las imágenes Docker pueden ejecutarse en cualquier infraestructura que soporte contenedores, ya sea en servidores locales, nubes públicas o entornos híbridos.
- Menos problemas de configuración: Al contener todas las dependencias necesarias, se evitan errores relacionados con bibliotecas faltantes o versiones incompatibles.

Gracias a esta portabilidad, Docker se ha convertido en una herramienta esencial para el desarrollo ágil y el despliegue de software en la nube.

3. Eficiencia

En comparación con las máquinas virtuales, Docker es una solución más eficiente en términos de uso de recursos y rendimiento:

- Menor consumo de recursos: No requiere un sistema operativo completo por cada instancia, ya que todos los contenedores comparten el mismo kernel del sistema anfitrión.
- Inicio y apagado rápidos: Mientras que una máquina virtual puede tardar minutos en iniciarse, un contenedor Docker arranca en segundos.
- Optimización del hardware: Permite ejecutar más aplicaciones en el mismo servidor sin desperdiciar capacidad de procesamiento o memoria.

Sin embargo, en aplicaciones que requieren modificaciones del kernel o un entorno altamente personalizado, las máquinas virtuales pueden ser una mejor opción.

4. Escalabilidad

Docker facilita la escalabilidad de aplicaciones, especialmente en arquitecturas de microservicios y entornos en la nube, gracias a:

- Orquestación con Kubernetes y Docker Swarm: Permiten administrar múltiples contenedores en entornos distribuidos de manera eficiente.
- Despliegues automatizados: Los contenedores pueden crearse y eliminarse dinámicamente en función de la demanda.
- Balanceo de carga eficiente: Se pueden distribuir instancias de aplicaciones en diferentes nodos para optimizar el rendimiento y la disponibilidad.

Estas características hacen que Docker sea una tecnología clave en infraestructuras escalables y de alta disponibilidad.

5) Los orquestadores (Kubernetes) y su gestión eficiente del escalado

- **Automatización del despliegue**
  - Manifiestos YAML: Definen servicios, pods,deployment y otros recursos
  - Helm Charts: Facilitan la configuracion e instalacion de aplicaciones complejas
  - Operadores y Controladores: Amplían las funciones de Kubernetes para tareas personalizadas

- **Escalado Automática**
  - Escalado Horizontal de Pods: Ajusta dinámicamente el número de pods según métrica como CPU, memorias RAM o métricas personalizadas.
  - Escalado Vertical: Se ajustan recursos (CPU/RAM) para su ejecución, y se pueden configurar de forma automática o manual
  - Escalado de Cluster: Aumenta o reduce la cantidad de nodos en función a la demanda de pods además que es compatible con proveedores como AWS y Azure.

- **Gestión de fallos y Alta Disponibilidad**
  - Reprogramación de Pods: Si un nodo falla, los nodos que serán afectados se trasladaran en nodos sanos
  - Auto-reinicio de Pods: Kubernetes reinicia contenedores que no respondes
  - Replicas Sets: Garantizan que una cantidad pequeña de nodos estarán siempre disponibles
  - Toleraciones y Afinidades: Controlan en qué nodos se ejecutara los pods para que no caigan en fallos.

- **Balanceo de carga**
  - Service: En estos tenemos ClusterIP que da acceso interno del cluster, también contamos con NodePort que expone el servicio en un puerto fijo en cada nodo, y por último LoadBalancer que balancea la carga de los proveedores cloud
  - Ingress Controller: Permite definir reglamentos de enrutamiento HTTP/HTTPS, es compatible con Nginx y HAProxy
  - Service Mesh: Balancea el tráfico basado en métricas avanzadas, al igual que ofrece un tráfico inteligente.

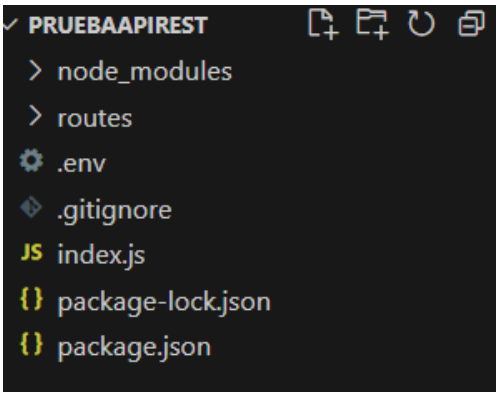
- **Actualizaciones Continuas y Rollbacks**
  - Rolling Updates: Reemplaza los pods obsoletos y antiguos por nuevos de forma gradual
  - Canary Realeses: Despliega nuevas versiones a una parte de los usuarios antes de lanzarlo a nivel global
  - Blue-Green Deployments: Se mantiene con dos versiones activas las cuales son una “Blue” que está en producción y otra “Green” para las pruebas

Parte II

Documentación del Proceso

Creación y Empaquetado de la API en una Imagen Docker

1. Asegurarse que la API esté desarrollada y lista para ser empaquetada en un contenedor Docker. Ejemplo de estructura de archivos en la API:



2. Crear el Dockerfile, ejemplo de lo que debería ir dentro del Dockerfile:

```
1   FROM node:23
2
3   WORKDIR /app
4
5   COPY package*.json ./
6
7   RUN npm install
8
9   COPY . .
10
11  CMD ["npm", "start"]
```

3. Construir y Probar la Imagen Docker, ejecutar los siguientes comandos en la terminal:

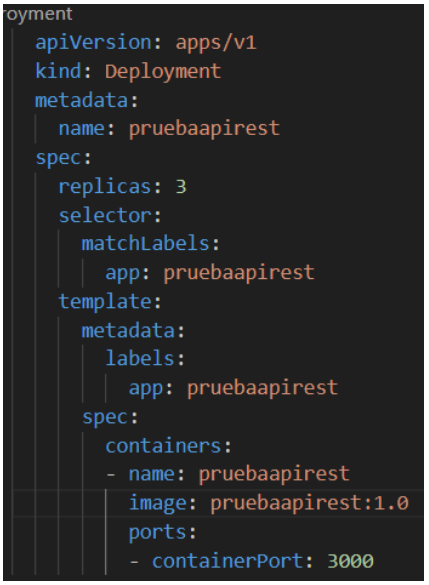
 # Construir la imagen Docker
docker build -t api_image .

 # Ejecutar el contenedor en modo interactivo y mapear el puerto
docker run -p 3000:3000 api_image

Este comando ejecuta un contenedor basado en la imagen creada y mapea el puerto 3000 del contenedor al puerto 3000 de tu máquina local.

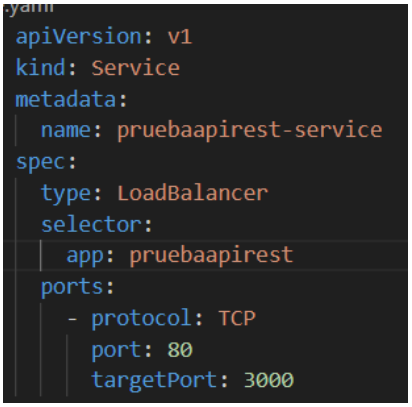
Configuración del Clúster de Kubernetes

1. Para desplegar la API en Kubernetes y configurar el balanceo de carga hay que crear un archivo de despliegue (Deployment): Este archivo define cómo se desplegará la aplicación en el clúster. A continuación, se muestra un ejemplo de un archivo deployment.yaml:



```yaml
oyment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pruebaapirest
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pruebaapirest
  template:
    metadata:
      labels:
        app: pruebaapirest
    spec:
      containers:
      - name: pruebaapirest
        image: pruebaapirest:1.0
        ports:
        - containerPort: 3000
```

2. Crear un archivo de servicio (Service):Este archivo crea un servicio de tipo `LoadBalancer` que distribuye el tráfico entrante en el puerto 80 entre las réplicas de la API que están escuchando en el puerto 3000.



```yaml
.yaml
apiVersion: v1
kind: Service
metadata:
  name: pruebaapirest-service
spec:
  type: LoadBalancer
  selector:
    app: pruebaapirest
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

3. Desplegar la aplicación en el clúster de Kubernetes: Aquí se ejecutan los siguientes comandos para aplicar los archivos de despliegue y servicio:

   kubectl apply -f deployment.yaml

   kubectl apply -f service.yaml

   Estos comandos crean los recursos definidos en los archivos YAML en el clúster de Kubernetes.

Implementar el sistema de escalado horizontal

1. Para permitir que Kubernetes ajuste automáticamente las réplicas de la API según la carga:

```
kubectl autoscale deployment pruebaapirest --cpu-percent=50 --min=3 --max=10
```

Este comando crea un HPA que mantiene entre 3 y 10 réplicas de la API, ajustando el número de pods para mantener la utilización de CPU en un 50%.

2. Para verificar el estado del escalado:

```
> kubectl get hpa
```

Balanceo de carga y alta disponibilidad

Kubernetes gestiona el tráfico entrante mediante el Service de tipo LoadBalancer, este asignará una IP pública automáticamente.

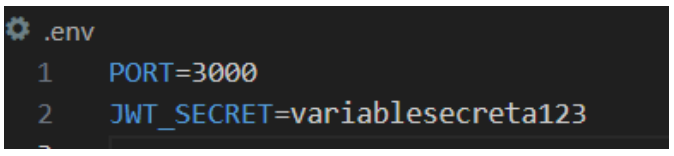
```
> kubectl port-forward service/api-service 8080:80
```

Seguridad y Autenticación

1. Implementación de autenticación con JWT en la API: primero debemos instalar los paquetes necesarios.

```
\pruebaapirest> npm install jsonwebtoken bcryptjs express dotenv
```

2. Para mayor seguridad, definimos una clave secreta en un archivo .env



3. En el archivo auth.js, implementamos el registro y el inicio de sesión

```
//este archivo sirve para asegurar las autenticaciones

import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';
import { usersDB } from './usuarios.js';  // aqui se accede a la base local

// Función para registrar un nuevo usuario
export const registerUser = (req, res) => {
  const { username, password } = req.body;

  // Verificar si el usuario ya existe
  const existingUser = usersDB.find(user => user.username === username);
  if (existingUser) {
    return res.status(400).json({ message: 'Usuario ya existe' });
  }

  // hashear la contraseña
  const hashedPassword = bcrypt.hashSync(password, 8);
```

```
// Función para iniciar sesión (generar token JWT)
export const loginUser = (req, res) => {
  const { username, password } = req.body;

  // Verificar si el usuario existe
  const user = usersDB.find(user => user.username === username);
  if (!user) {
    return res.status(400).json({ message: 'Usuario no encontrado' });
  }

  // Verificar si la contraseña es correcta
  const passwordIsValid = bcrypt.compareSync(password, user.password);
  if (!passwordIsValid) {
    return res.status(401).json({ message: 'Contraseña incorrecta' });
  }

  // Generar un token JWT
  const token = jwt.sign({ username: user.username }, process.env.JWT_SECRET

  res.json({ message: 'Inicio de sesión exitoso', token });
};
```

4. Middleware para Proteger Rutas con Autenticación: Para asegurar que solo los usuarios autenticados accedan a ciertas rutas, usamos un middleware.

```
// Middleware para verificar el token JWT en rutas protegidas
export const authMiddleware = (req, res, next) => {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(403).json({ message: 'No se proporcionó token de autenticación' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;  // Guardamos la información del usuario en la solicitud
    next();
  } catch (error) {
    return res.status(401).json({ message: 'Token inválido o expirado' });
  }
};
```