



PROYECTO FINAL

DESARROLLO DE PATRONES DE TOLERANCIA A FALLOS EN APLICACIONES DISTRIBUIDAS CON KUBERNETES

INTEGRANTES:

DARCY VANESSA MINA - 22II4I6

JUAN CAMILO MORENO - 22I0980

JOSHEP PERALTA CARVAJAL - 2200046

ANDRES FELIPE GARZON - 2I56952

CONTENIDO

1

Introducción

2

Planteamiento del
problema

3

Descripción de las
soluciones

4

Implementación

5

Evaluación y
conclusiones

6

Referencias
Bibliográficas



INTRODUCCIÓN

En la actualidad, la gestión de microservicios es fundamental para las tecnologías utilizadas en diversas áreas de las organizaciones a nivel mundial. Dicha arquitectura ha evolucionado para que estos sistemas sean cada vez más eficientes y estén disponibles para resolver las tareas diarias de distintos entornos laborales.

Con el avance de estas tecnologías, surge la necesidad de diseñar mecanismos que garanticen la continuidad y resiliencia de los sistemas críticos en la nube, pues las nuevas aplicaciones introducen desafíos que requieren entornos más eficaces y con mayor capacidad de respuesta.

PLANTEAMIENTO DEL PROBLEMA

Los sistemas distribuidos enfrentan desafíos constantes, como fallos de red, sobrecargas y caídas de servicios, que pueden afectar su rendimiento y disponibilidad, exponiendo a las organizaciones a riesgos significativos en términos de continuidad operativa. Estos riesgos pueden deberse a:

- Configuraciones erróneas
- Falta de monitoreo
- Sobrecargas del sistema

Convertiendo la tolerancia a fallos en un desafío crítico para garantizar la resiliencia y la disponibilidad de los servicios en la nube

OBJETIVO

Implementar patrones de tolerancia a fallos en una arquitectura de microservicios desplegada en Kubernetes, integrando tecnologías como Resilience4j para la gestión de patrones de diseño, Kubernetes para el despliegue escalable y Prometheus para monitoreo y evaluación del rendimiento, asegurando así una solución robusta y confiable en condiciones de alta demanda.

DESCRIPCIÓN DE LAS SOLUCIONES

Tecnologías utilizadas

LIBRERIAS

- Resilience4j



FRAMEWORKS Y LENGUAJES

- Spring Boot



MONITOREO

- Prometheus y grafana
- Spring Boot actuator



CONTENEDORES

- Docker



ORQUESTACIÓN

- Kubernetes



SIMULACIÓN DE FALLOS

- Chaos Monkey



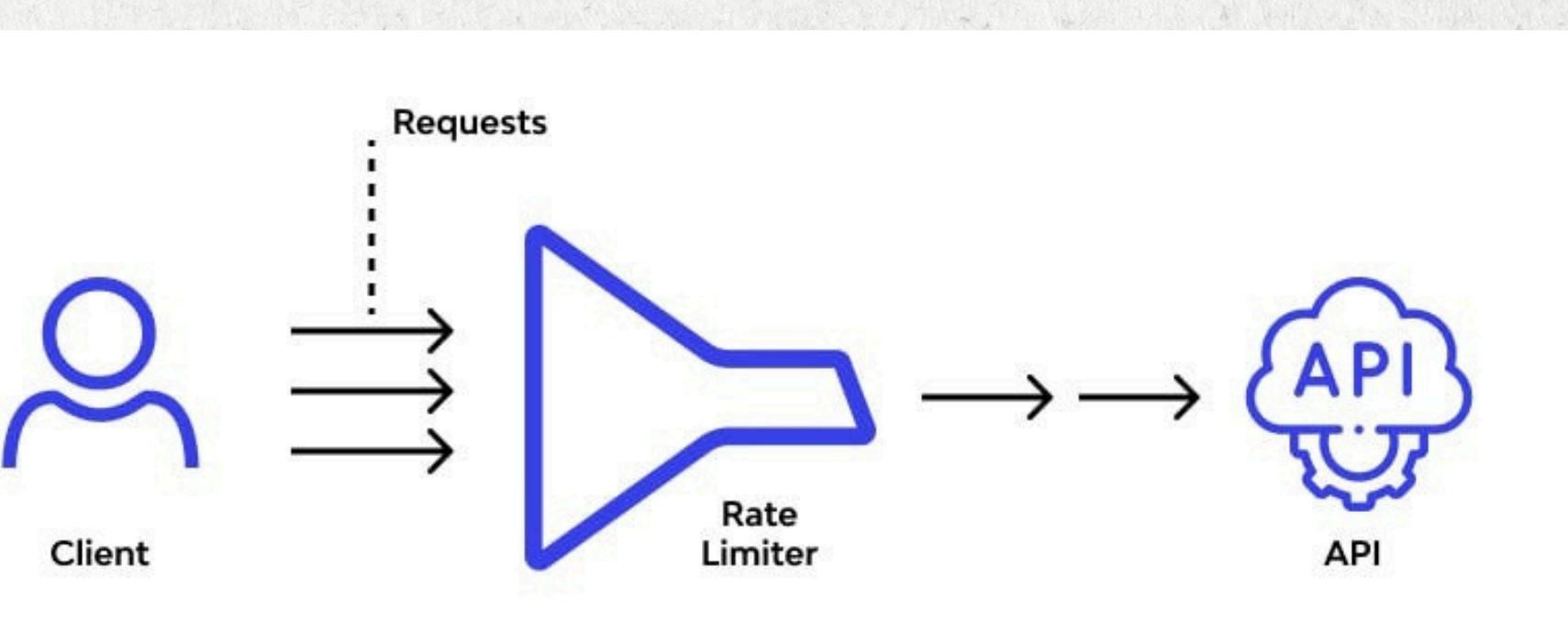
DESCRIPCIÓN DE LAS SOLUCIONES

Diagrama de secuencia



RATE LIMITER

Un Rate Limiter (Limitador de Tasa) es un mecanismo de control utilizado en sistemas informáticos y redes para restringir el número de solicitudes o eventos que se pueden procesar en un servicio, API o recurso dentro de un período de tiempo específico. Su objetivo principal es proteger los recursos de una sobrecarga excesiva, garantizar la disponibilidad del servicio, evitar el abuso y distribuir de manera uniforme la carga entre los usuarios o solicitudes.



CIRCUIT BREAKER (INTERRUPTOR AUTOMÁTICO)

Función: Evita que una aplicación intente realizar operaciones que probablemente fallen, permitiendo así que fallen rápidamente sin que se gestione en la aplicación.

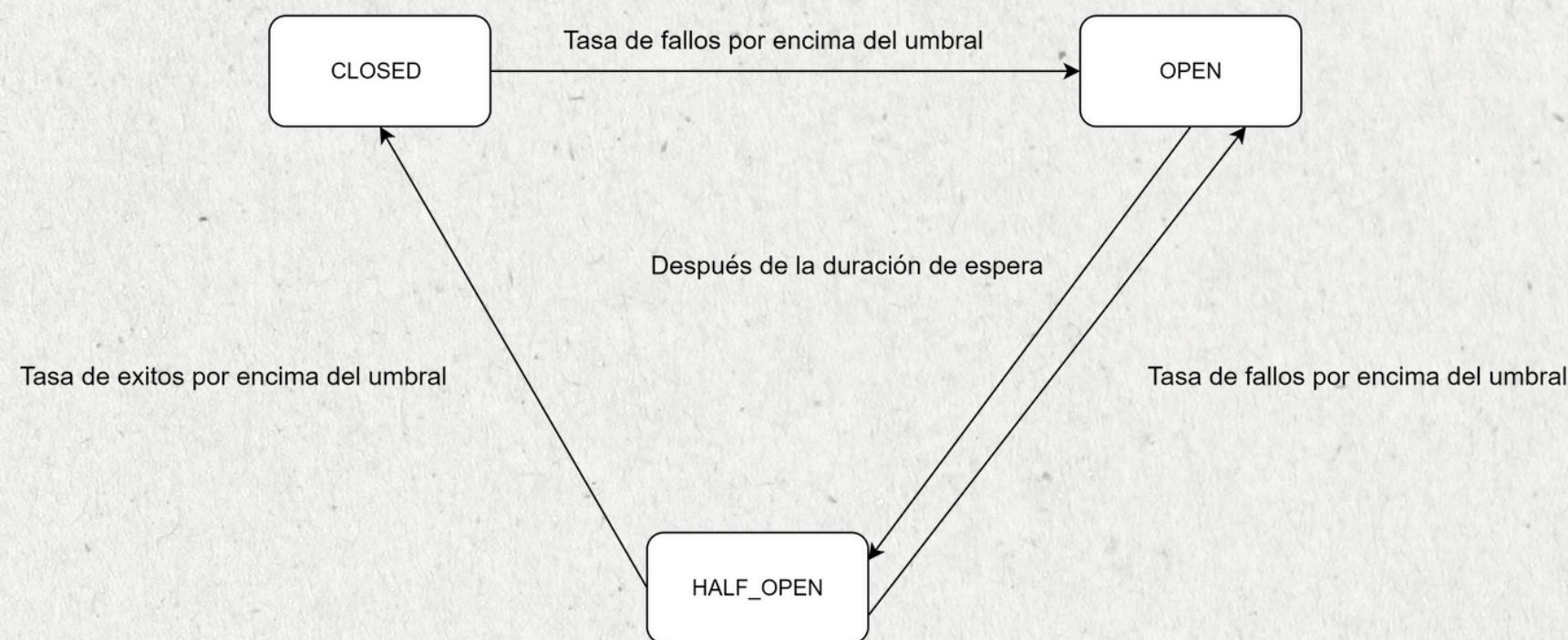
Motivación: En sistemas distribuidos, una falla en un servicio puede provocar un efecto dominó en cascada. (circuit breaker los evita).

Circuit Braker monitoriza las llamadas a servicios remotos y abre el circuito (corta la comunicación) si detecta un número determinado de fallos, impidiendo nuevas llamadas hasta que el sistema se recupere.

Closed: Todas las solicitudes pasan sin restricción al servicio externo

Open: Todas las solicitudes fallan sin intentar la conexión con el servicio externo

Half-Open: Permite un número limitado de solicitudes. Y así verificar si el servicio externo se recuperó, si tiene éxito regresa al estado closed si no al estado open.



RETRY

Función:

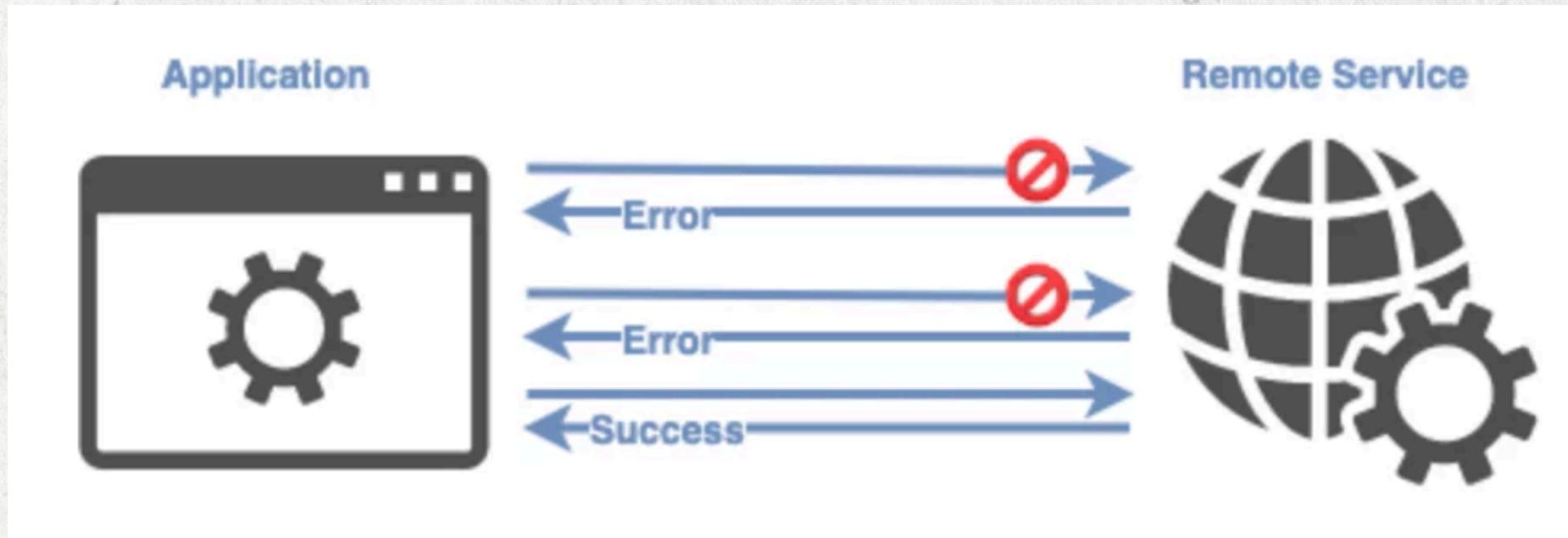
- Se usa para el manejo de fallos transitorios en sistemas distribuidos
- Permite que una aplicación reintente una operación fallida tras un tiempo estipulado
- Es útil donde el fallo es temporal, como la perdida momentánea de la conectividad o servicios temporalmente no disponibles

Motivación:

- Se aplica donde los fallos transitorios son comunes y que tenga alta probabilidad de que al 2 intento continúe su operatividad.
- No genera interrupciones complejas o graves dentro de los sistemas.

Implementación:

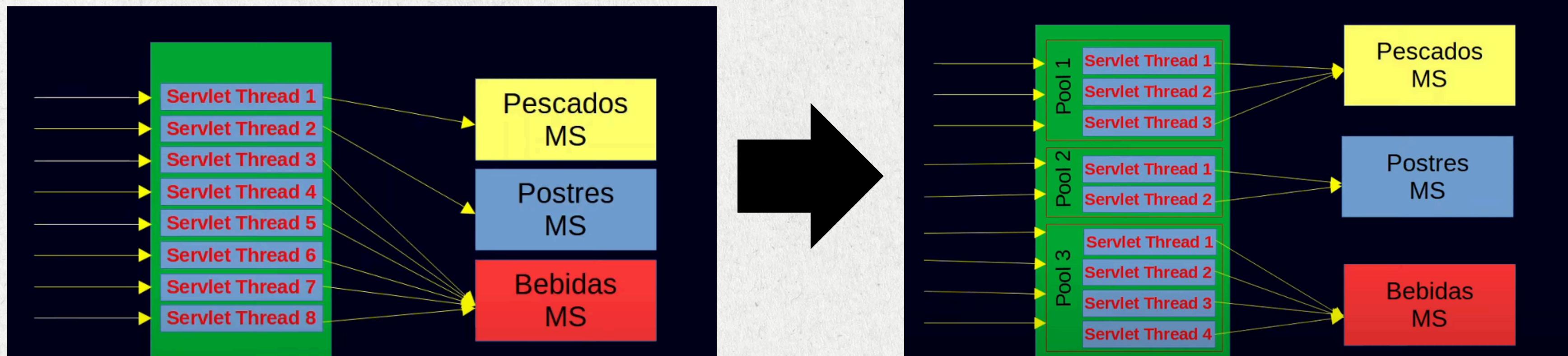
- Se establece un mecanismo que detecta el fallo y vuelve a intentar la operación después de un retraso predefinido
- Se debe definir correctamente el intervalo entre intentos (lineal o exponencial)
- Se debe definir el número máximo de reintentos antes de considerar el fallo definitivo.



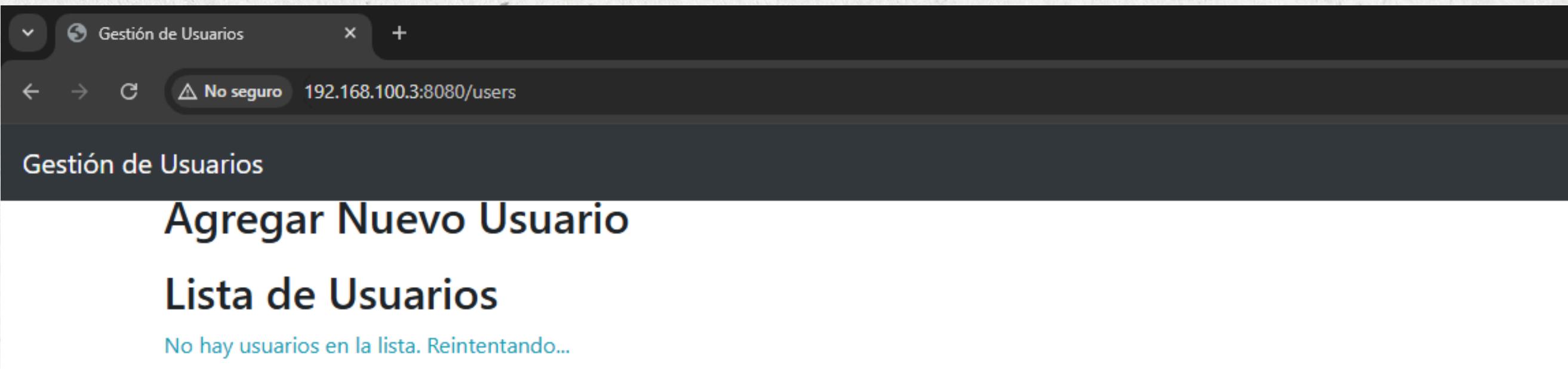
BULKHEAD

El patrón bulkhead se define como un recurso basado en aislar elementos de una aplicación en pools (o grupos), de forma que, si una falla, las otras puedan seguir funcionando. Cabe destacar que las labores de esta opción evitan la pérdida de disponibilidad o inactividad de los sistemas tecnológicos. Además de esto, el patrón bulkhead se reconoce por ser un tipo de diseño de aplicación enfocado en la tolerancia a fallos y eventualidades que puedan ocurrir en el sistema.

De la misma forma, el bulkhead pattern destaca gracias a su implementación, ya que contribuye en los procesos de asignación de límites a los recursos que se pueden usar para determinados servicios.



IMPLEMENTACIÓN RETRY



application.properties

```
resilience4j.retry.instances.userService.maxAttempts=5
resilience4j.retry.instances.userService.waitDuration=5s
```

UserService.java

```
@Retry(name = "userService", fallbackMethod = "fallbackGetAllUsers")
public List<User> getAllUsers() {
    if (users.isEmpty()) {
        System.out.println("No hay usuarios en la lista. Reintentando...");
        throw new RuntimeException("No hay usuarios en la lista.");
    }

    System.out.println("Lista de usuarios recuperada exitosamente.");
    return users;
}
```

IMPLEMENTACIÓN RATE LIMITER

APPLICATION.PROPERTIES

```
resilience4j.ratelimiter.instances.userService.limitForPeriod=1  
resilience4j.ratelimiter.instances.userService.limitRefreshPeriod=10s  
resilience4j.ratelimiter.instances.userService.timeoutDuration=1s
```

POM.XML

```
<dependency>  
    <groupId>io.github.resilience4j</groupId>  
    <artifactId>resilience4j-spring-boot3</artifactId>  
    <version>2.0.2</version>  
</dependency>
```

USERSERVICE

```
import io.github.resilience4j.ratelimiter.annotation.RateLimiter;
```

USERSERVICE

```
@RateLimiter(name = "userService")  
public User createUser(User user) {  
    user.setId(counter.getAndIncrement());  
    users.add(user);  
    return user;  
}
```

```
"timestamp": "2024-11-06T18:57:43.223+00:00",  
"status": 500,  
"error": "Internal Server Error",  
"path": "/api/users"
```

IMPLEMENTACIÓN BULKHEAD

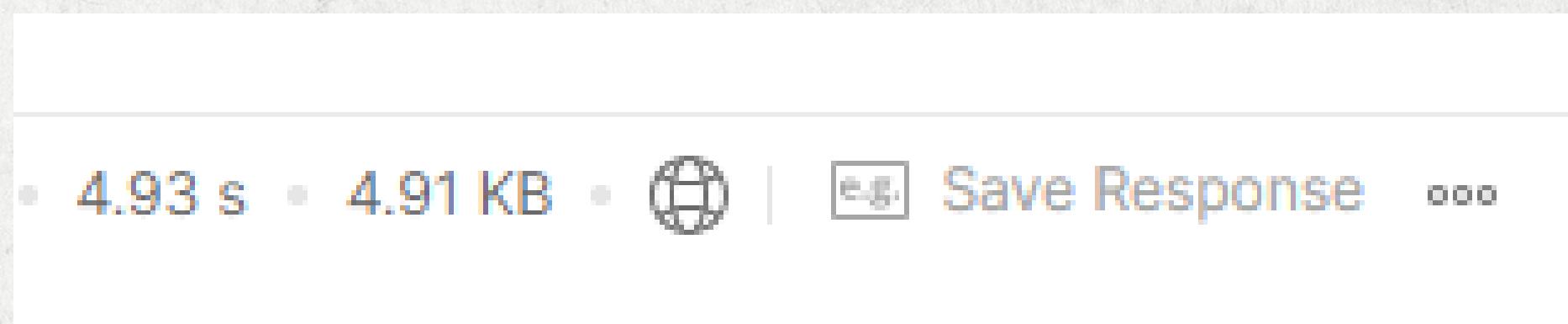
POM.XML

```
<groupId>io.github.resilience4j</groupId>
<artifactId>resilience4j-spring-boot3</artifactId>
<version>2.0.2</version>
</dependency>
```

APPLICATION.PROPERTIES

```
# Configuración de Bulkhead
resilience4j.bulkhead.instances.userService.maxConcurrentCalls=1
resilience4j.bulkhead.instances.userService.maxWaitDuration=0ms
```

REACCION DEL BULKHEAD



USERSERVICE

```
@Bulkhead(name = "userService", type = Bulkhead.Type.SEMAPHORE)
public User authenticate(String username, String password) {
    return users.stream()
        .filter(u -> u.getUsername().equals(username) && u.getPassword().equals(password))
        .findFirst()
        .orElse(null);
}

@Bulkhead(name = "userService", type = Bulkhead.Type.SEMAPHORE)
public List<User> getAllUsers() {
    return users;
}

@Bulkhead(name = "userService", type = Bulkhead.Type.SEMAPHORE)
public User createUser(User user) {
    user.setId(counter.getAndIncrement());
    users.add(user);
    return user;
}

@Bulkhead(name = "userService", type = Bulkhead.Type.SEMAPHORE)
public User updateUser(Long id, User updatedUser) {
    User user = getUserId(id);
    if (user != null) {
        user.setName(updatedUser.getName());
        user.setEmail(updatedUser.getEmail());
        user.setUsername(updatedUser.getUsername());
        user.setPassword(updatedUser.getPassword());
        return user;
    }
    return null;
}

@Bulkhead(name = "userService", type = Bulkhead.Type.SEMAPHORE)
public boolean deleteUser(Long id) {
    return users.removeIf(u -> u.getId().equals(id));
}

private User getUserId(Long id) {
    return users.stream().filter(u -> u.getId().equals(id)).findFirst().orElse(null);
}
```

IMPLEMENTACIÓN CIRCUIT BREAKER

application.properties

```
server.port=8080

# Configuración de Resilience4j para el Circuit Breaker
resilience4j.circuitbreaker.instances.microuserCircuitBreaker.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.microuserCircuitBreaker.slidingWindowSize=20
resilience4j.circuitbreaker.instances.microuserCircuitBreaker.permittedNumberOfCallsInHalfOpenState=3
resilience4j.circuitbreaker.instances.microuserCircuitBreaker.waitDurationInOpenState=10000
resilience4j.circuitbreaker.instances.microuserCircuitBreaker.failureRateThreshold=50
```

ViewController.java

```
// frontend/src/main/java/com/example/frontend/controller/ViewController.java
package com.example.frontend.controller;

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.beans.factory.annotation.Autowired;

// Método para manejar el inicio de sesión
@PostMapping("/login")
@ResponseBody
@CircuitBreaker(name = MICROUSERS_CIRCUIT_BREAKER, fallbackMethod = "loginFallback")
public ResponseEntity<?> login(@RequestBody Map<String, String> credentials) {
    String url = "http://192.168.50.3:8081/api/login";
    try {
        return restTemplate.postForEntity(url, credentials, String.class);
    } catch (Exception e) {
        throw e; // lanza la excepción para que el Circuit Breaker la detecte
    }
}

// Método de fallback para login
public ResponseEntity<?> loginFallback(@RequestBody Map<String, String> credentials, Throwable throwable) {
    return ResponseEntity.status(503).body("Servicio de autenticación no disponible. Por favor, inténtalo más tarde.");
}

// Método para obtener usuarios
@GetMapping("/getUsers")
@ResponseBody
@CircuitBreaker(name = MICROUSERS_CIRCUIT_BREAKER, fallbackMethod = "getUsersFallback")
```

FrotendApplication.java

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

EVALUACIÓN Y CONCLUSIONES

Los distintos patrones de tolerancia a fallos, como Circuit Breaker, Retry, Bulkhead y Rate Limiter, ofrecen enfoques complementarios para mejorar la resiliencia en servicios en la nube. Cada uno aborda fallos desde diferentes perspectivas, permitiendo prevenir colapsos, gestionar fallos temporales y evitar sobrecargas. La implementación de estos patrones en conjunto asegura una mayor estabilidad y continuidad del servicio, fundamentales en arquitecturas de microservicios donde la disponibilidad y la respuesta rápida ante fallos son críticas. Su integración fortalece la infraestructura en la nube, garantizando una experiencia de usuario confiable y adaptada a la alta demanda.

REFERENCIAS BIBLIOGRAFICAS

1. Livora, T. (2017). Fault tolerance in microservices (Doctoral dissertation, Masarykova univerzita, Fakulta informatiky). <https://is.muni.cz/th/ubkja/masters-thesis.pdf>
2. Mendoca, N. C., et al. (2020). Model-based analysis of microservice resiliency patterns. 2020 IEEE International Conference on Software Architecture (ICSA), 114-12 <https://doi.org/10.1109/ICSA47634.2020.00021>

iGRACIAS!