UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Eric Segala Jovelli

Estudo de padrões de projeto para tolerância a falhas voltados a serviços

Florianópolis/SC
2024

Eric Segala Jovelli

Estudo de padrões de projeto para tolerância a falhas voltados a serviços

Trabalho de Conclusão de Curso de Graduação em Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de graduado em sistemas de informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis/SC
2024

Eric Segala Jovelli

Estudo de padrões de projeto para tolerância a falhas voltados a serviços

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "graduado em sistemas de informação" e aprovado em sua forma final pelo Curso de Graduação em Sistemas de Informação.

Florianópolis/SC, 20 de junho de 2024.

Banca Examinadora:

Prof. Alex Sandro Roschildt Pinto, Dr.
Universidade Federal de Santa Catarina

Prof. Cristian Koliver, Dr.
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal,
Dr.(orientador)
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço à minha família pelo apoio durante todos esses anos e também a todos os professores que se esforçam a cada dia para fazer a diferença no nosso sistema de ensino.

RESUMO

A arquitetura orientada a serviços (SOA) é um conceito amplamente utilizado em projeto de software. Alguns de seus benefícios são: alta qualidade de serviço, reutilização, confiabilidade, escalabilidade, disponibilidade e heterogeneidade de tecnologia e plataforma utilizada. SOA está atualmente evoluindo para incluir microsserviços na resolução de problemas relacionados com os limites da arquitetura monolítica usual. As necessidades das aplicações deram nascimento a opções leves e altamente escaláveis oferecidas pela arquitetura de microsserviços. É essencial a constante busca por alternativas e possibilidades para aprimorar a confiabilidade e disponibilidade dos serviços. O objetivo deste trabalho é comparar padrões arquiteturais de microsserviços para aumentar a confiabilidade de serviços por meio de implementação de padrões voltados à tolerância à falhas. Para as avaliações, foram implementadas provas de conceito e estas foram submetidas a testes de carga, com a injeção de falhas para a observação do comportamento do sistema. A partir desta carga, foram definidos cenários de teste para avaliar o comportamento e desempenho do dos protótipos que implementam cada padrão. Foram realizados estudos de caso para os padrões *retry*, disjuntor, *sidecar* e *throttling*. Para a análise, foram consideradas métricas de latência, saturação e erros, comparando implementações do serviço com e sem mecanismos de tolerância a falhas.

Palavras-chave: Sistemas Distribuídos; Microsserviços; Tolerância a Falhas; Arquitetura de Software

ABSTRACT

Service-oriented architecture (SOA) is a widely used concept in software design. Some of its benefits are: high quality of service, reusability, reliability, scalability, availability and heterogeneity of technology and platform used. SOA is currently evolving to include microservices in solving problems related to the limits of the usual monolithic architecture. Application needs gave birth to lightweight and highly scalable options offered by microservices architecture. It is essential to constantly search for alternatives and possibilities to improve the reliability and availability of services. The objective of this work is to compare microservices architectural patterns to increase the reliability of services through the implementation of fault tolerance-oriented design patterns. For the evaluations, proofs of concept were implemented and these were subjected to load tests, with fault injection to observe the system's behavior. Based on this load, test scenarios were defined to evaluate the behavior and performance of the prototypes that implement each standard. Case studies were carried out for the retry, circuit breaker, sidecar and throttling patterns. For the analysis, latency, saturation and error metrics were considered, comparing service implementations with and without fault tolerance mechanisms.

Keywords: Distributed Systems; Microservices; Fault Tolerance; Software Architecture

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CPU	*Central Processing Unit*
CSV	*Comma Separated Values*
HTTP	*Hypertext Transfer Protocol*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
JWT	*Json Web Token*
MSA	*Microservices Architecture*
MVC	*Model-View-Controller*
REST	*Representational State Transfer*
SOA	*Service Oriented Architecture*
SSL	*Secure Sockets Layer*
TLS	*Transport Layer Security*

SUMÁRIO

1 INTRODUÇÃO

Na década de 90 houve a popularização da Internet e, desde então, a maneira pela qual as aplicações funcionam vem sendo alterada. No início, uma era denominada *Web 1.0*, o conteúdo dos sites era estático e o uso era primordialmente técnico, recheado de sites corporativos, acadêmicos ou em construção (GIL, 2014). Além disso, usava-se bastante a rede para envio de *emails* e buscadores, como o Google. A nova fase foi chamada de *Web 2.0*, uma era conhecida como era social ou participativa. Nessa época os blogs e ferramentas de *chats* e mídias sociais ganharam notoriedade, como o Facebook e o Youtube. No momento atual uma área de estudo que tem tido destaque é a "Web 3.0" (GROOMBRIDGE, 2023), que é a definição dada à uma Internet impactada pelas novas tecnologias, entre elas a inteligência artificial, a Internet das coisas e a *blockchain*.

Desde a época da *Web 1.0*, o grande volume de usuários das aplicações que foi surgindo exigiu que se pensasse em arquiteturas para sistemas lidarem com um alto volume de requisições. O jeito tradicional de desenvolver aplicações é chamado monolítico , no qual se tem um sistema atômico, funcionando em um único processo, em que se tem um *software* com diferente partes interligadas em um único programa (VILLAÇA; PIMENTA JR; AZEVEDO, 2018), funcionando em uma única máquina. Porém, à medida que a complexidade dos sistemas foi crescendo, novas formas de criar aplicações surgiram.

Uma alternativa de grande destaque nas últimas décadas foi SOA[1]. Ela consiste em criar componentes de software implantados separadamente, permitindo que eles se comuniquem e trabalhem juntos de forma a gerarem uma aplicação integrada. Essa arquitetura permite que serviços se comuniquem entre si e sejam reaproveitados por sistemas diferentes. Algumas das vantagens da SOA são reuso do código, redução de redundância de funcionalidades e redução do custo de manutenção (SILVA, 2006). A SOA apareceu no início dos anos 2000 e cresceu até o fim dessa década, tendo um crescimento estagnado a partir da década seguinte.

Uma alternativa à SOA é a arquitetura de microsserviços, formada por um conjunto de serviços. Nela, um serviço pode ser descrito como uma aplicação em seu próprio domínio, capaz de evoluir independentemente e ter sua própria arquitetura, tecnologia e plataforma (XIAO; WIJEGUNARATNE; QIANG, 2016), além de poder ser gerenciado, implantado e escalado de forma independente, seguindo seu próprio ciclo de lançamento. Dessa forma, uma arquitetura de microsserviços consiste em uma aplicação que é composta por componentes independentes que, cada um deles é uma aplicação própria.

Um serviço é limitado a um contexto limitado do negócio (XIAO; WIJEGUNARATNE; QIANG, 2016), então em uma aplicação como um *ecommerce*, um exemplo de uso de microsserviços seria a existência de um serviço para calcular frete, outro para gerenciar estoque e por fim um para gerenciamento de recursos humanos da empresa.

[1] Adotaremos neste trabalho a sigla SOA (do inglês, *Service-Oriented Architecture*), devido à popularidade.

1.1 PROBLEMA

À medida que os negócios crescem e dependem cada vez mais da presença digital, passa a ser de suma importância que os sistemas sejam confiáveis. Casos em que sistemas ficam indisponíveis gerando graves problemas aos usuários não são raros. Um exemplo disso ocorreu dia 29/05/2024, quando em um mesmo dia os bancos Bradesco, Banco do Brasil e Inter ficaram indisponíveis (DIGITAL, 2024), impedindo a realização de transações PIX.

Dessa forma, surgiram diversas técnicas para garantir que aplicações com arquitetura de microsserviços continuem funcionando mesmo que um de seus serviços apresente problemas (TÉCNICAS DE INTEGRAÇÃO ENTRE MICROSSERVIÇOS, 2016). Como o uso de microsserviços tem crescido muito na última década, esse trabalho terá como foco padrões de projeto para tolerância a falhas destinados a serviços.

1.2 OBJETIVOS

Nas seções a seguir estão descritos o objetivo geral e os objetivos específicos deste trabalho.

1.2.1 Objetivo Geral

Comparar os padrões arquiteturais de microsserviços para confiabilidade por meio de implementação dos padrões de arquitetura voltados à tolerância a falhas, possibilitando comparar as vantagens e desvantagens práticas de cada padrão.

1.2.2 Objetivos Específicos

Com o intuito de alcançar o objetivo geral, foram traçados objetivos específicos que direcionaram a pesquisa:

a) Escolher casos de uso envolvendo padrões de projetos voltados à tolerância a falhas para sistemas baseados em microsserviços;

b) Codificar os casos de uso e implantá-los usando a MSA;

c) Realizar testes usando essas implantações;

d) Avaliar quantitativamente e qualitativamente o experimento explicando a eficácia e os custos adicionados ao sistema devido ao uso dos padrões, como por exemplo o aumento na latência gerado.

1.3 ORGANIZAÇÃO DO TRABALHO

O Capítulo 2 faz uma exposição teórica, contextualizando padrões de projeto, padrões arquiteturais e os desafios que acompanham a arquitetura de microsserviços. Já o Capítulo 3 aborda diversos padrões de projeto voltados à tolerância a falhas na MSA e

demonstrações de uso deles, com implementações de situações hipotéticas de uso. Além disso, apresenta resultados de testes para levantar vantagens e desvantagens advindas do uso desses padrões. O Capítulo 4 apresenta trabalhos correlatos. Por fim, o Capítulo 5 apresenta considerações finais sobre a obra e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O objetivo deste capítulo é apresentar um panorama geral e histórico sobre padrões de projeto, além de apresentar categorias de uso, como mensageria, serviços, e orientação a objetos. Além disso, o capítulo contextualiza os principais padrões arquiteturais que foram usados no desenvolvimento de aplicações para Internet desde as primeiras décadas da sua difusão.

2.1 PADRÕES DE PROJETO

Os padrões de projeto são soluções documentadas para problemas frequentemente encontrados no desenvolvimento de aplicações (BURNS; OPPENHEIMER, 2016). São usados para auxiliar na arquitetura de software e são documentados de forma a serem aplicáveis para qualquer linguagem de programação. O uso de padrões de projeto permite aumentar a qualidade, desempenho e viabilidade de uso da programação distribuída.

Eles podem ser implementados para resolver problemas em um nível mais baixo da arquitetura, como na construção de objetos. Eles também podem resolver problemas de um nível mais abstrato, como por exemplo serviços compostos por inúmeras classes interagem, existindo padrões de projetos para resolver problemas de determinados padrões arquiteturais.

Os padrões de projeto ganharam grande destaque com a publicação, em 1994, da obra (GAMMA *et al.*, 1994), que descrevia 23 padrões e ficou conhecida como "Gang Of Four" devido à obra ter sido escrita por quatro autores. Essa obra classifica os padrões em três categorias (ALMADI; HOOSHYAR; AHMAD, 2021): os criacionais, destinados à criação de objetos; os estruturais, voltados à organização da estrutura dos componentes no sistema; e por fim os comportamentais, destinados à organização da interação entre objetos.

Somente para área de mensageria, por exemplo, existem padrões de projetos destinados à construção, roteamento, transformação, canal e pontos de comunicação das mensagens (GREGOR HOHPE, 2023). Existem padrões de projetos até para *smart contracts* (LIU *et al.*, 2018), havendo padrões para criação, relacionamento e comportamento dos contratos, além de existir padrões para adaptabilidade e interoperabilidade do sistema como um todo.

2.2 PADRÕES ARQUITETURAIS

Desde a década de 70, com o aumento da complexidade das aplicações, passou-se a estudar não apenas a organização de entidades de baixo nível, como classes, mas também a organização em alto nível, como serviços formados por inúmeras classes(VALENTE, 2020). É possível extrair desse estudo em alto nível certos padrões, denominados padrões

arquiteturais.

Um dos padrões arquiteturais mais antigos é a arquitetura em camadas, que organiza a aplicação em camadas e cada uma delas pode depender somente de métodos da camada inferior. Um exemplo clássico dessa arquitetura é o conjunto dos protocolos de rede da arquitetura TCP-IP (FOROUZAN; FEGAN, 2009).

Outro padrão arquitetural de grande importância é o padrão MVC. Ele consiste na formação da aplicação em três grupos (DEACON, 2009). O primeiro, o *view*, é responsável pelos elementos visuais, como botões e tabelas. Já o *controller* é responsável por alterar o *view* de acordo com as interações do usuário, e também invocar funções do último componente, o *model*. Este é responsável por realizar operações de busca ou modificação das entidades de domínio e demais operações que envolvem as regras de negócio.

Um outro padrão que tem recebido grande destaque ultimamente é a arquitetura de microsserviços. Ela consiste na fragmentação da aplicação em um conjunto de serviços. Um serviço pode ser descrito como uma pequena aplicação limitada no seu contexto que pode evoluir independentemente e ter sua própria arquitetura, tecnologia e plataforma (XIAO; WIJEGUNARATNE; QIANG, 2016). Além disso, cada um pode ser gerenciada e escalado de forma independente, tendo seu próprio ciclo de implantação e metodologia de desenvolvimento. Cada serviço tem sua base de código própria. Isso permite que seja gerenciado por uma equipe pequena. Outro fato que contribui para isso é que esses componentes oferecem APIs externas, o que permite que as equipes responsáveis por outras áreas não precisem entender os detalhes da implementação, mas apenas integrar com a API.

O padrão de microsserviços (MSA) é composto por serviços que são implantados de forma fragmentada, no qual cada componente pode ser implantado de forma independente e tem suas funcionalidades limitadas a uma área específica e essas partes se comunicam entre si por APIs e em, alguns casos, por serviços de mensagem. Apesar desse padrão ser inspirado na SOA, a diferença entre os dois estilos é a granularidade (MONTESI; WEBER, 2016), ou seja, enquanto na arquitetura SOA o sistema é concentrado em um único executável, no sistema de microsserviços há vários executáveis que serão implantados de forma independente. Diferentemente do modelo tradicional monolítico, que frequentemente tinha uma camada de acesso aos dados, cada serviço é responsável por lidar com a persistência de seus dados.

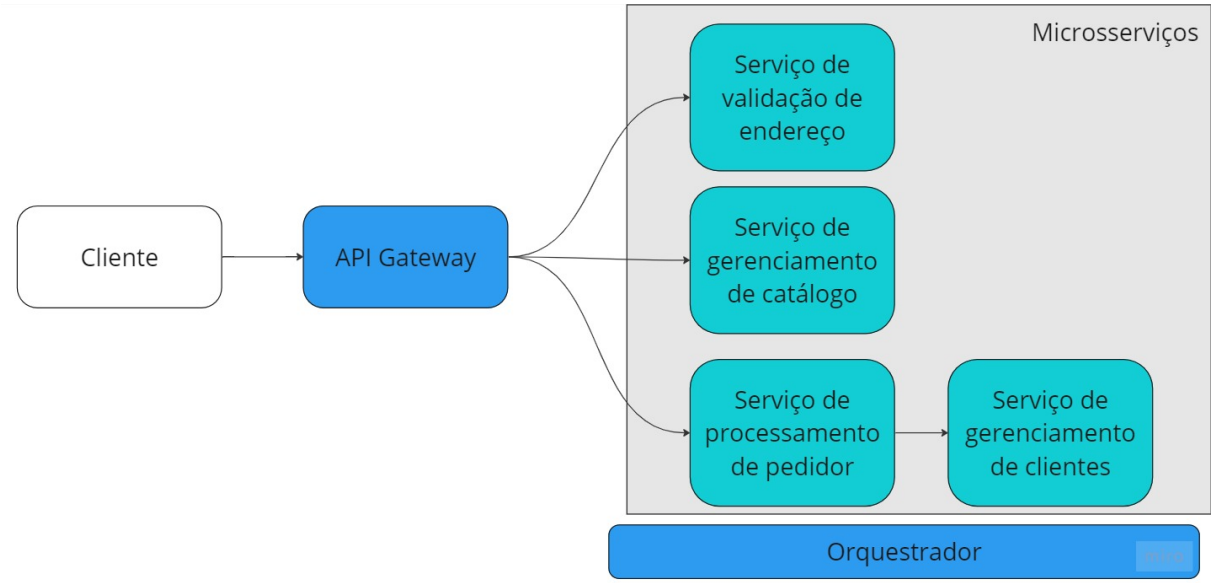
Além dos serviços, outros componentes presentes na MSA são:

Orquestrador Esse componente é responsável por implantar os serviços nos servidores e executar tarefas como detecção de falhas. Exemplos de orquestradores são AWS Fargate e Kubernetes.

API Gateway Responsável por encaminhar os pedidos do cliente ao serviço apropriado, funciona como uma porta de entrada.

Dessa forma, uma arquitetura baseada em microsserviços é definida como uma padrão arquitetural para aplicações distribuídas no qual a aplicação é composta por partes menores independentes e cada uma dessas aplicações é uma pequena aplicação própria (XIAO; WIJEGUNARATNE; QIANG, 2016). Nesse contexto, uma aplicação de *e-commerce*, por exemplo, poderia ser dividida da seguinte forma: um serviço lida com validação de endereço, outro gerencia o catálogo de produtos, outro processa pedidos e um último processa pagamentos.

Figura 1 – Exemplo do uso da MSA em um *e-commerce*.



Fonte: o autor

Dado a natureza distribuída da MSA, desafios que não existiam em sistemas monolíticos passam a existir (ADDEEN, 2019). Entre eles podemos citar a integridade, ou seja, o sistema distribuído se apresentar como um só ao usuário; a descoberta do serviço, ou seja, encaminhamento da requisição ao serviço responsável; o versionamento do código (manter o sistema funcionando de forma coesa com serviços funcionando com código de versões diferentes); segurança e tolerância a falhas.

Devido a tais problemas que surgem com o uso da arquitetura orientada a serviços, existem padrões de projetos específicos para esse tipo de arquitetura e para categorias específicas de problemas. Esse trabalho terá como foco o estudo de padrões de projeto voltados a tratar problemas de tolerância a falhas na MSA.

3 AVALIAÇÃO DE PADRÕES DE PROJETO PARA TOLERÂNCIA A FALHAS VOLTADOS A SERVIÇOS

Por maior que seja a capacidade financeira e o tempo disponível para execução de um projeto, é impossível garantir que não haja falhas durante o tempo de execução do programa (LEE *et al.*, 1990). Se em um cenário ideal um sistema não apresenta falhas, na prática geralmente elas ocorrerão e ocasionarão erros. Assim, são necessárias técnicas para impedir que esses erros causem o colapso do sistema. Essa área de estudos é denominada tolerância a falhas.

Microsserviços se comunicam uns com os outros através da rede, o que traz diversos desafios. A comunicação por meio da rede pode ser falha e isso pode gerar consequências catastróficas. Caso haja um serviço que execute uma operação síncrona, ou seja, depende da resposta de outro para continuar a operação, a espera pela resposta pode ser infinita caso não haja mecanismos que levam em consideração essa possibilidade. Em um sistema em que há uma cadeia de microsserviços que executam operações síncronas interdependentes, um serviço inoperante pode causar uma reação em cadeia que torna todo o sistema inoperante. A solução para esse problema são os mecanismos de tolerância a falhas, que impediriam esse tipo de situação através da detecção das falhas para tratamento imediato.

Nesse capítulo haverá o detalhamento de padrões de projeto explicando o funcionamento de cada um escolhido. Em seguida, haverá um estudo de caso, em que é feita uma demonstração de uso em ambiente de experimentação. Em cada teste, além de demonstrar o funcionamento do padrão, ocorre uma comparação com um serviço que não tem o padrão implementado, de forma a demonstrar as vantagens e os custos adicionais advindos do uso do padrão.

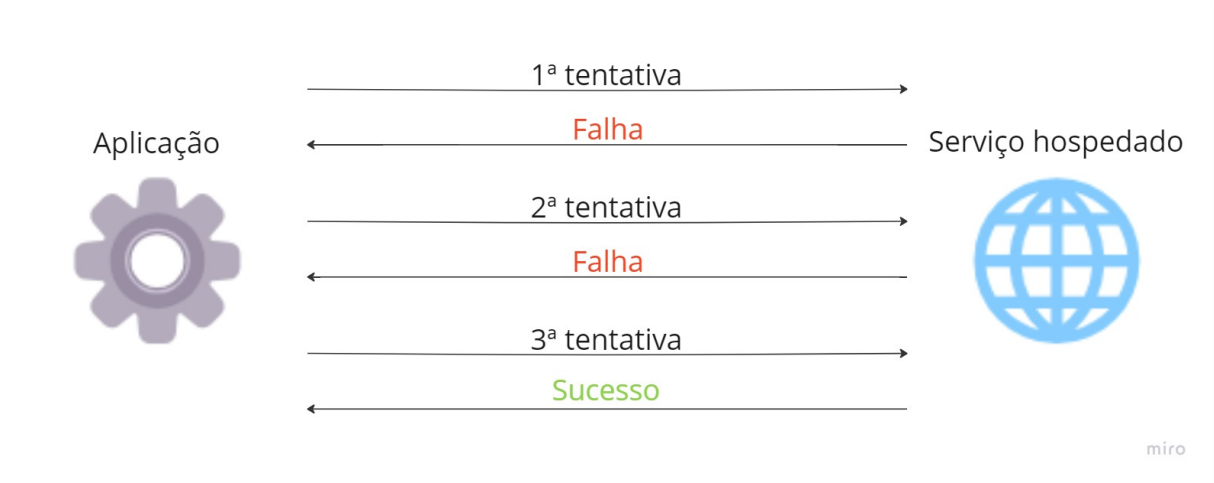
3.1 *RETRY*

O padrão *Retry* permite com que a tentativa de obter uma resposta de outro serviço ocorra mais de uma vez caso a requisição falhe. Ao obter falha na primeira tentativa, o serviço espera por um tempo e tenta novamente. Caso não funcione, é aguardado um tempo maior e há uma nova tentativa. O sistema deve realizar isso por um determinado número de vezes até conseguir a resposta (ADDEEN, 2019). O foco desse padrão de projeto é lidar com falhas que se corrigem em pouco tempo, como, por exemplo, uma falha na rede, um serviço que está sendo reiniciado, ou até mesmo um serviço que está sobrecarregado.

É importante notar que deve existir uma quantidade limite de tentativas e, assim, caso não haja uma resposta na quantidade prevista, as tentativas são encerradas (MICROSOFT, 2023e). Outro ponto importante é que esse sistema evita um ataque de *self-denial*, no qual o próprio sistema faz uma quantidade exaustiva de requisições para si mesmo. A Figura 2 exemplifica uma aplicação que consome os serviços de um serviço hospedado

e usa o *Retry*. Após duas tentativas fracassadas, na terceira esse serviço retorna uma resposta válida.

Figura 2 – *Funcionamento do padrão Retry*



Fonte: (MICROSOFT, 2023e).

3.1.1 Estudo de caso

Para demonstrar um caso de uso do *retry*, foi criado um servidor integrado a um serviço externo. A falha simulada é uma falha nesse serviço externo ou então na conexão entre o servidor principal e o externo.

Em um servidor sem esse mecanismo, o cliente receberia uma resposta de sistema indisponível praticamente imediata. Porém, isso é um problema. No contexto de uma aplicação de comércio eletrônico, por exemplo, ao visualizar o erro o usuário deixaria o site e a empresa perderia a compra. Então, a solução encontrada foi implementar o padrão *retry*. Assim, o sistema tentará obter uma resposta desse serviço externo várias vezes e em momentos diferentes antes de retornar o erro.

Foi usado o Emulab (HIBLER *et al.*, 2008) como infraestrutura para todos os estudos de caso desse trabalho. O Emulab (HIBLER *et al.*, 2008) é uma plataforma de testes voltado a experimentos envolvendo sistemas distribuídos. Ele foi desenvolvido pela fundação Open Source e permite o uso de uma mesa-de-testes para trazer maior realismo aos experimentos. Foi escolhida a máquina t430 para todos os nós usados no experimento.

No Emulab (HIBLER *et al.*, 2008) o experimento foi criado como três nós. Essa estrutura pode ser explicada como uma aplicação na arquitetura cliente-servidor na qual o servidor possui uma integração com uma aplicação externa. Na estrutura cliente-servidor o cliente é um testador de carga que gera requisições ao servidor e registra mensurações. O servidor é uma API que responde ao cliente, consultando um nó banco de dados. Enquanto no nó servidor são implementados mecanismos de tolerância a falhas, é na máquina do serviço externo que são geradas as falhas.

Servidor

Esse nó recebe as requisições do gerador de carga. É nele que está implementado o mecanismo de tolerância a falhas. Foi implementada uma *API Node.js* usando a biblioteca de roteamento *express*. Foi usada a biblioteca Retry para implementar o mecanismo de tolerância a falhas. Não houve grandes dificuldades na parte da implementação, já que a biblioteca abstrai a complexidade do funcionamento interno do mecanismo de *retry*. O tempo entre as novas tentativas é calculado pele algoritmo *exponential backoff* (NPM, INC., 2024b), uma técnica na qual uma operação é feita novamente até que um número limite de tentativas seja atingido (CLOUD, 2024), e o tempo entre essas tentativas é incrementado exponencialmente à medida que elas ocorrem.

```
Math.min(random * minTimeout * Math.pow(factor, attempt), maxTimeout)
```

Cada item da fórmula é descrito a seguir. Para esse trabalho foram usados os valores padrões. Dessa forma, a última tentativa ocorre após 17 minutos e 3 segundos.

retries A quantidade máxima de vezes para tentar novamente a operação. O valor padrão é 10;

factor O fator exponencial a ser usado. O valor padrão é 2;

minTimeout O número de milissegundos antes de iniciar a primeira tentativa. O valor padrão é 1000;

maxTimeout O número máximo de milissegundos entre duas tentativas. O valor padrão é infinito;

randomize Randomiza os tempos limite multiplicando um fator entre 1-2. O valor padrão é *false*.

Nesse trecho temos o mecanismo implementado em um *endpoint*. Na terceira linha, cria-se um objeto chamado *operation* usando as configurações padrões da biblioteca. Em seguida, o método *.attempt* recebe a função que será tentada novamente caso seja possível. Caso haja um erro, já um mecanismo *try/catch* que o captura. Em seguida, *operation.retry(error)* retornará e a função será executada novamente. Caso o limite de tentativas tenha se esgotado, é o bloco *if* contornado e o usuário recebe um código 500, indicando um erro.

```
app.get("/code/:id", async (req, res) => {
  console.log("GET", req.params.id);
  var operation = retry.operation();
  operation.attempt(async function() {
    try {
      const response = await instance.get("code/"+req.params.id);
      res.json(response.data)
    } catch (error) {
```

```
      console.log(error)
      if (operation.retry(error)) {
        return;
      }
      res.status(500).send(error);
    }
  });
});
```

Essa mesma lógica ocorre no *endpoint* */put*:

```
app.put("/code/:id", async (req, res) => {
  var operation = retry.operation();
  operation.attempt(async function() {
    try {
      console.log("PUT", req.params.id, req.body);
      const code = await instance.put("code/"+req.params.id,req.body)
      res.json(code.data)
    } catch (error) {
      console.log(error)
      if (operation.retry(error)) {
        return;
      }
      res.status(500).send(error);
    }
  });
});
```

Serviço Externo

Esse nó consiste em uma API com três *endpoints*, um */put* para criar ou alterar dados, um */get* para obter dados buscando por id, e um */delete* para remoção. Além disso, esse nó possui uma instância do MongoDB 4.4, um sistema gerenciador de banco de dados *NoSQL*. A API é iniciada com o gerenciador de processos(KEYMETRICS, INC., 2024). Nessa parte do código é definido no servido como será o formato da entidade, ou seja, apenas um objeto com um único atributo chamado *code*:

```
const CodeSchema = new Schema({
  code: String,
});
```

Cada um dos *endpoints* possui um bloco apenas um bloco *try/catch* que retorna um objeto em caso de sucesso ou um erro. Demonstração da lógica do *endpoint* */get*:

```
app.get("/code/:id", async (req, res) => {
  try {
    console.log("GET", req.params.id);
    const code = await Code.findById(req.params.id);
    res.json(code);
```

```
  } catch (error) {
    console.log(error);
    res.status(500).send(error);
  }
});
```

Gerador de carga

Esse nó exerce duas funções. A primeira é a preparação do banco de dados, que é feita por um código Java que envia requisições ao serviço externo para limpar o banco de dados, e após isso insere dados nesse banco de dados. Ao inserir os dados no banco, os *ids* dos objetos inseridos são armazenados em um arquivo CSV, que posteriormente é usado na criação das requisições ao realizar os testes de carga.

Além desse *script* de preparação, há nesse nó uma instalada que é usada para os testes, o Jmeter (APACHE SOFTWARE FOUNDATION, 2024), que é um software tradicional de teste de carga feito em Java. Essa ferramenta permite realizar inúmeras configurações para realização de testes, como número de *threads*, duração e duração do período de ativação das *threads*. Essas configurações permitem simular uma aplicação em um contexto real. Além disso, o Jmeter gera relatórios a partir dos resultados dos testes, facilitando o estudo.

Cenários de avaliação

Inicialmente ocorre a etapa de preparação do ambiente. Nessa etapa todos os objetos do banco de dados foram removidos. Em seguida, foram inseridas 5.000 entradas, com códigos com tamanho de 20 caracteres.

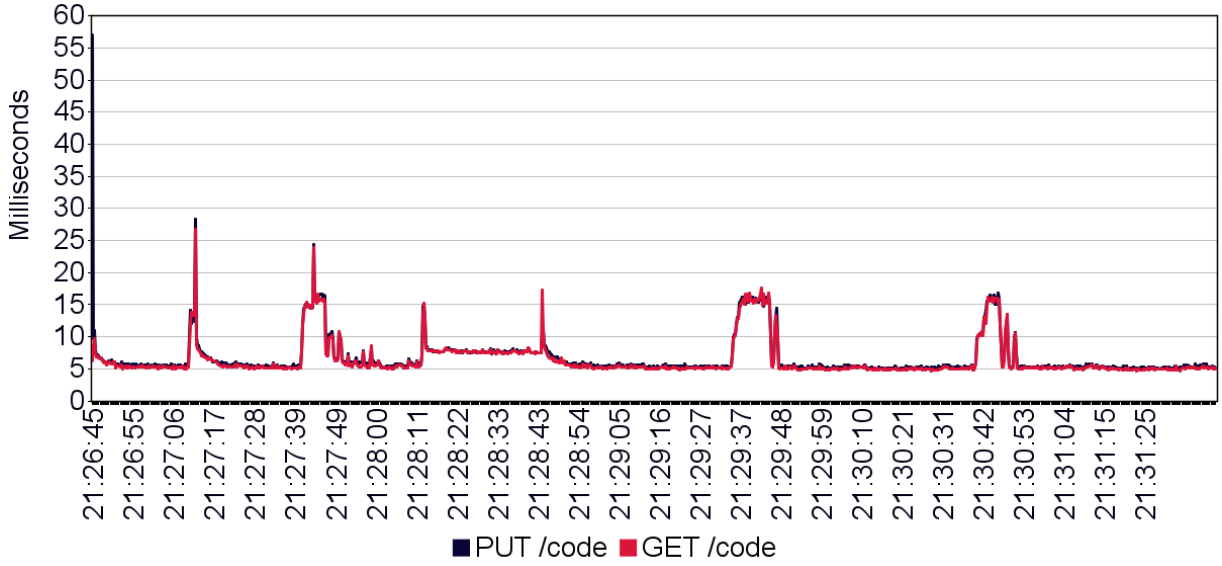
O teste foi executado pelo período de cinco minutos com um servidor sem qualquer mecanismo de tolerância a falhas. Os testes executados seguiram os valores mostrados na Tabela 1. O objetivo é que se tenha três falhas. A primeira com recuperação rápida (serviço externo é reiniciado), a segunda com tempo de recuperação médio (1 segundo) e a terceira longa (30 segundos). Vale notar que essa tabela de testes é aplicada não só no teste do padrão *retry*, mas também é usada para os padrões restantes.

Tempo	Descrição
30 segundos após o início	Serviço externo é reiniciado
30 segundos após a primeira falha	Serviço externo é desligado
1 segundo após desligar o serviço externo	Serviço externo é ligado novamente
30 segundos após iniciar o serviço externo	Serviço externo é desligado novamente
30 segundos após desligar o serviço externo	Serviço externo é ligado novamente

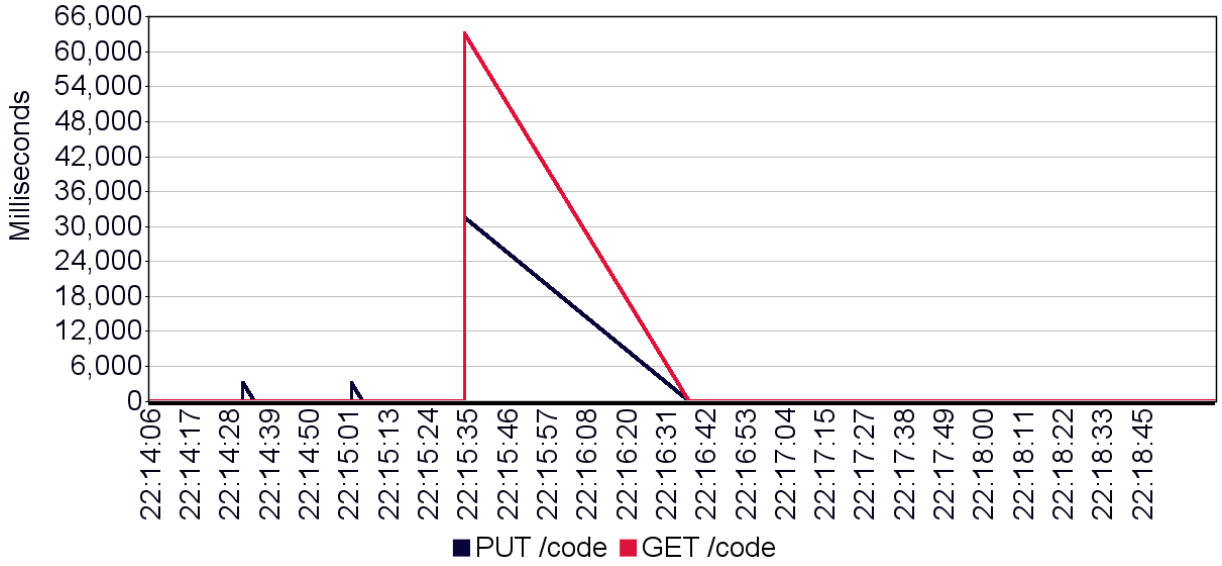
Tabela 1 – Tempo até a falha

3.1.2 *Resultado*

Figura 3 – Latência com e sem o uso do *Retry*



(a) Sem uso do *Retry*

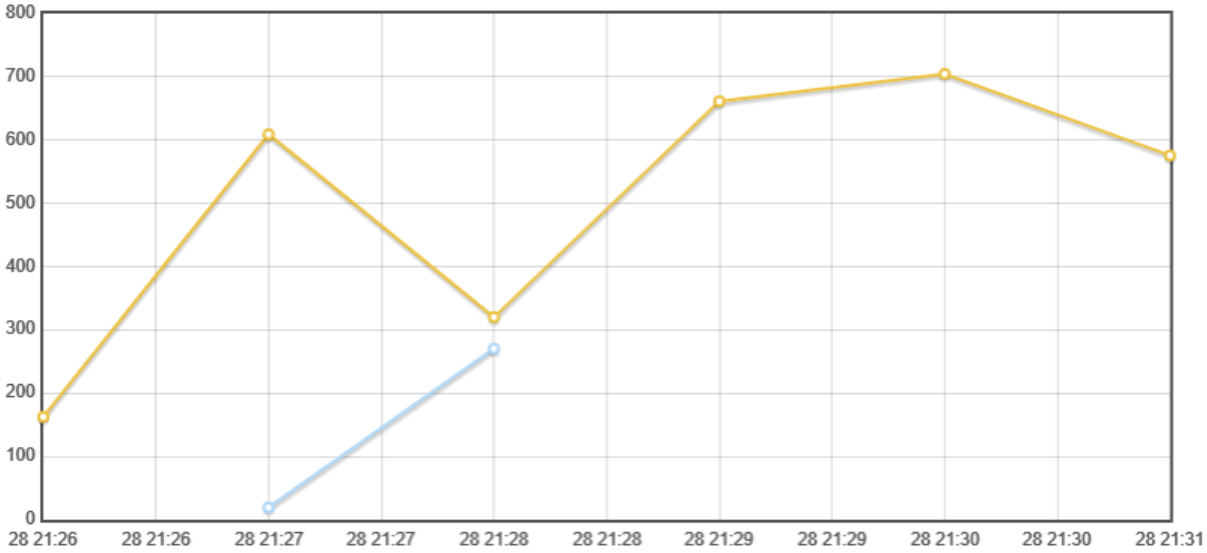


(b) com o uso do *Retry*

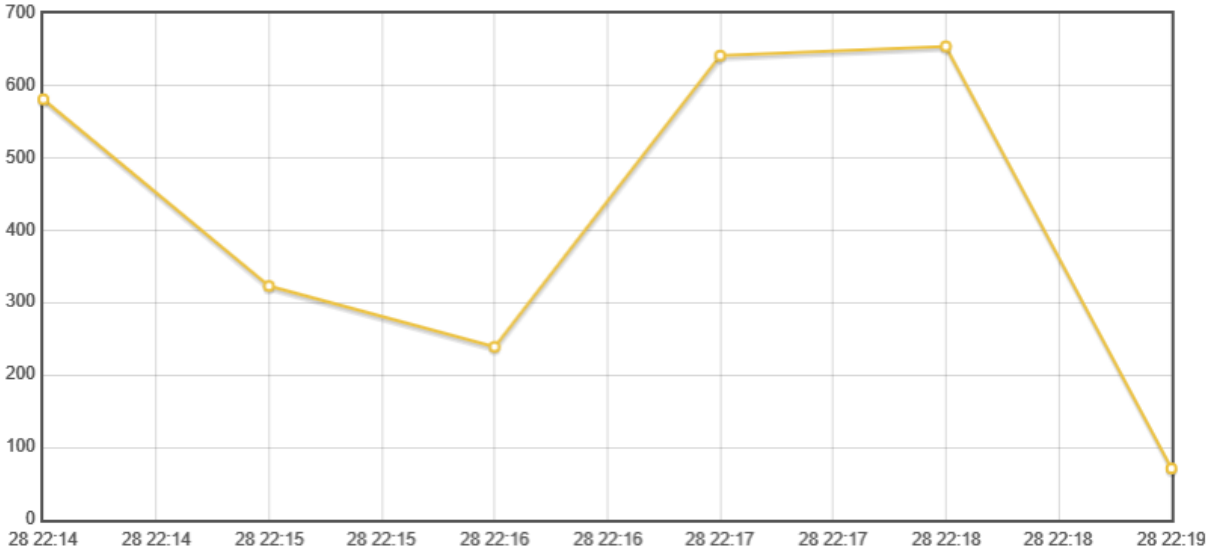
Fonte: o autor

As Figuras 3, 4 e 5 demonstram a comparação dos resultados em testes com e sem uso do padrão *Retry*. No experimento sem mecanismos de tolerância, é possível notar que em alguns momentos do teste há leves aumentos na latência de poucas dezenas de milissegundos, que são causados por variações no tráfego de rede, processamento e outras causas que não envolvem uma espera artificial da requisição. É importante notar que nos momentos das duas falhas iniciais (30 e 60 segundos) o servidor principal retornou erros, que não aparecem no gráfico de vazão por serem quantidades de requisições muito pequenas, não influenciando o gráfico, onde os valores são originados por médias de períodos de 60

Figura 4 – Códigos de resposta com e sem o uso do *Retry*



(a) Sem uso do *Retry*. Linha azul: código 500. Linha amarela: código 200



(b) com o uso do *Retry*. Linha azul: código 500. Linha amarela: código 200

Fonte: o autor

Figura 5 – Estatísticas do teste com e sem o uso do *Retry*

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do *Retry*.

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**150692**	**0**	**0.00%**	**7.89**	**3**	**63231**	**5.00**	**13.00**	**14.00**	**19.00**	**502.70**
GET /code	75345	0	0.00%	7.61	3	63231	5.00	8.00	13.00	17.00	251.46
PUT /code	75347	0	0.00%	8.18	4	63223	5.00	8.00	13.00	17.00	251.35

(b) com o uso do *Retry*.
Fonte: o autor

segundos. Por outro lado, no momento da falha longa (iniciada nos 90 segundos) começa a haver um grande número de repostas código 500, causadas pela falha de duração longa. Esse comportamento ocorre pois a biblioteca HTTP usada no serviço principal não possui mecanismo de *retry*, lançando uma exceção quando o serviço externo retorna um erro, que por sua vez é tratada e resulta no retorno de um código 500.

A Figura 3 compara a latência do cliente original à latência do serviço principal modificado com a inserção de um mecanismo de *retry*. É possível notar que com o uso desse padrão nos momentos das falhas há picos de latência maiores, que são causados pelo tempo que leva para ocorrer as novas tentativas. Por outro lado, a Figura 4, que compara a vazão, não apresenta código 500 quando o mecanismo é usado, o que demonstra que ele foi capaz de tratar as falhas no sistema externo. Na Figura 5, que compara estatísticas de teste, isso é confirmado quando se compara o percentual de erros, que neste último é de 0% enquanto sem o uso do padrão o valor é de 10.34%.

3.2 DISJUNTOR

O disjuntor, traduzido do inglês *Circuit Breaker* é focado em tratar falhas geradas por um tempo de espera infinita ou muito longa ao fazer uma requisição. A falta de resposta da requisição pode ser causada por falhas na comunicação ou problemas no

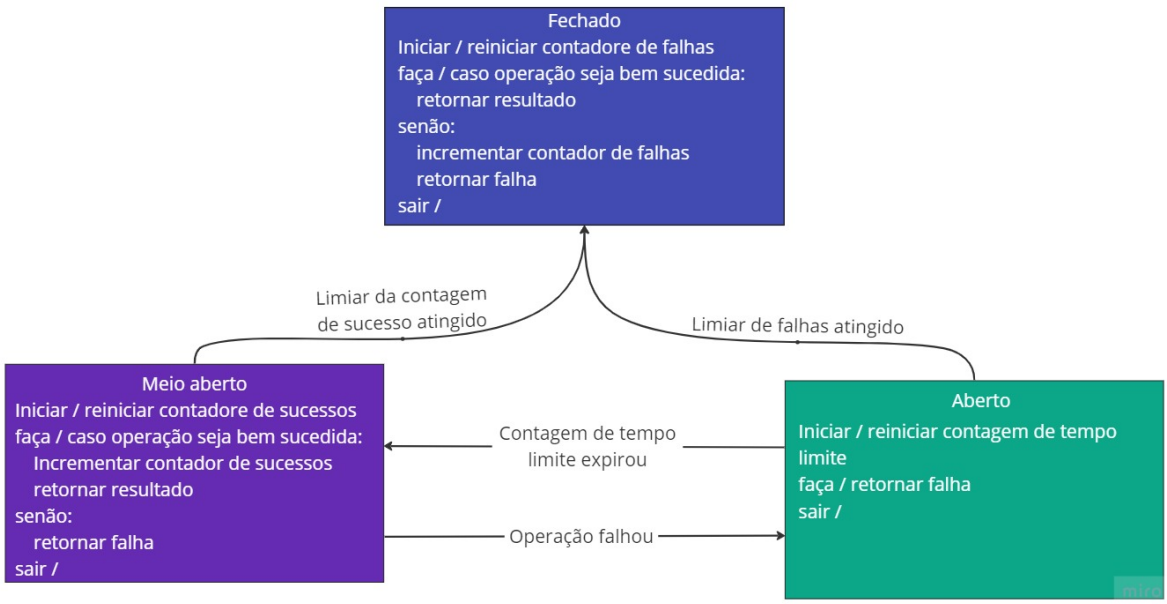
serviço que recebeu a requisição. O uso do disjuntor visa melhorar a estabilidade da aplicação (AZURE, 2023).

Em um sistema distribuído, chamadas a recursos e serviços podem falhar devido a falhas transientes, como rede de Internet lenta, sobrecarga, tempo limite de espera atingido (erro de *timeout*). Essas falhas normalmente se corrigem normalmente após um período curto de tempo e aplicações robustas devem ser capazes de lidar com esse tipo de falha.

Porém, há casos em que as falhas ocorrem devido a problemas inesperados, gerando um tempo de espera maior que o previsto, gerando um tempo de correção muito maior. Nesses casos, não faz sentido uma aplicação continuar tentando uma operação que é improvável que seja bem sucedida. O mais apropriado é que a aplicação aceite que a operação falhar e lide com a situação de outra maneira (AZURE, 2023). O padrão disjuntor propõe evitar que operações que provavelmente irão falhar continuem sendo executadas repetidamente.

O funcionamento de padrão disjuntor é análogo a uma máquina de estados. Existem três estados que mudam de acordo com a frequência que ocorrem falhas no serviço requisitado (MONTESI; WEBER, 2018). No estado fechado, as requisições do cliente ao serviço acontecem normalmente, e são monitoradas se o serviço requisitado está retornando erros, um exemplo desse monitoramento pode ser o uso de *pinging*. Caso haja falhas, o estado é transferido para estado meio aberto. No estado aberto, toda vez que o cliente faz uma requisição é retornado um erro. Após um determinado tempo o estado é transicionado para o meio aberto. No estado meio aberto, um estado de transição, é permitido que seja enviado um número limitado de requisições. Caso não haja falhas por um período de tempo, ocorre a transição para o estado fechado, caso contrário para o aberto.

Figura 6 – Algoritmo e máquina de estados do disjuntor



Fonte: (AZURE, 2023).

3.2.1 Estudo de caso

Foi realizado um experimento usando uma arquitetura e ambiente de testes idênticos ao do experimento anterior. A única diferença é que nesse caso foi adicionado um mecanismo de disjuntor em vez do *retry*. Para isso foi usada a biblioteca Opossum(MANAGER, 2024).

Nesse trecho do código são definidas as configurações do disjuntor:

```
const options = {
  timeout: 3000,
  errorThresholdPercentage: 50,
  resetTimeout: 3000
};
```

Essas configurações definem como é ajustado o disjuntor:

timeout Define o tempo limite de espera para que uma falha seja identificada, medido em milisegundos;

errorThresholdPercentage Define o limite mínimo de requisições falhas para mudar para o estado aberto medido em porcentagem;

resetTimeout Define que a contagem de tempo para transicionar do estado aberto para o meio aberto.

Nessa parte, os objetos da classe *CircuitBreaker* são criados usando as configurações descritas:

```
const getByIdBreaker = new CircuitBreaker(getById, options);
const upsertBreaker = new CircuitBreaker(upsert, options);
```

No trecho a seguir, ha o uso do objeto que acabou de ser criado no *endpoint /get*. Ele é associado ao uso de uma *promise*, um mecanismo de programação assíncrona do Javascript. Caso a operação seja bem sucedida, a *promise* invoca a função *then()* e ocorre o retorno do objeto em formato JSON com código de sucesso. Caso o disjuntor esteja no estado fechado, ocorre o retorno do código 503, que indica que serviço está indisponível no momento.

```
app.get("/code/:id", (req,res) => {
  console.log('api got GET request')
  getByIdBreaker.fire(req,res)
  .then(r => {
    console.log('GET returning response')
    res.json(r)
  })
  .catch(() => {
    console.log('GET returning 503')
    res.status(503).send()
  })
})
```

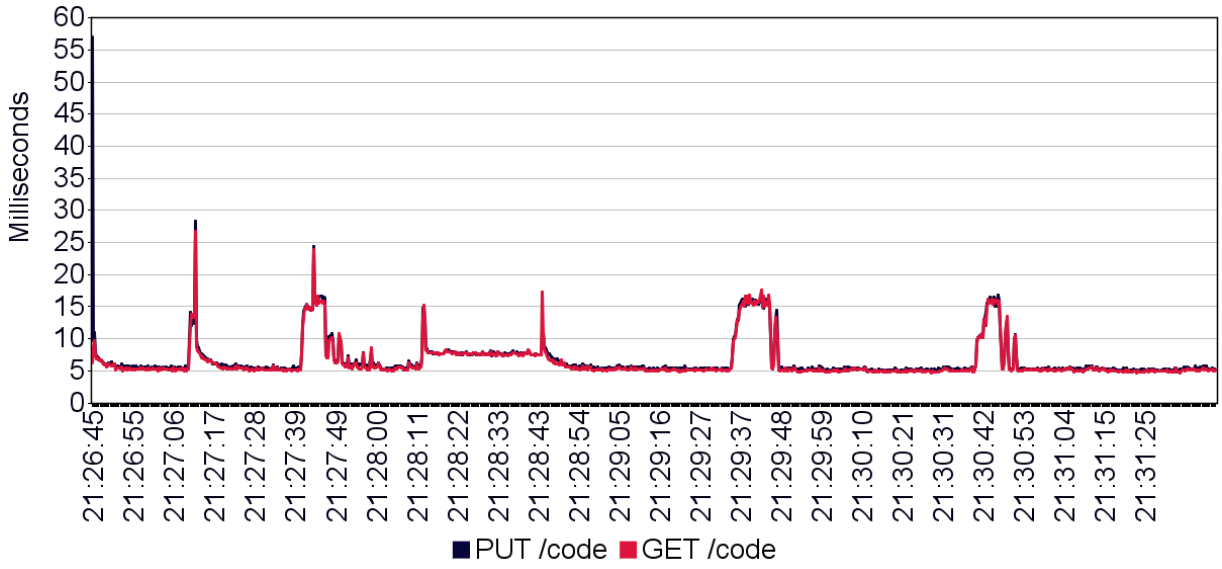
Ao analisar a Figura 7, a comparação das métricas de latência, é possível notar que há picos de latência quando ocorrem as falhas nos momentos 30, 60 e 90 segundos após o início. Porém, esses picos não são significativos, sendo o mais lento menor que um quarto de segundo. Ou seja, independente do estado do disjuntor o cliente irá ter uma resposta rápida. A grande diferença ocorre na vazão.

Na Figura 8, a comparação das métricas de vazão, a linha de respostas falhas (código 503) começa com um valor baixo. Isso ocorre pois houve duas falhas curtas no primeiro minuto. Assim, apenas durante esses dois momentos o disjuntor passou para o estado fechado. Porém, como o período de *resetTimeout* foi definido como um valor de 3 segundos, rapidamente o disjuntor volta ao estado meio aberto e, em seguida fechado.

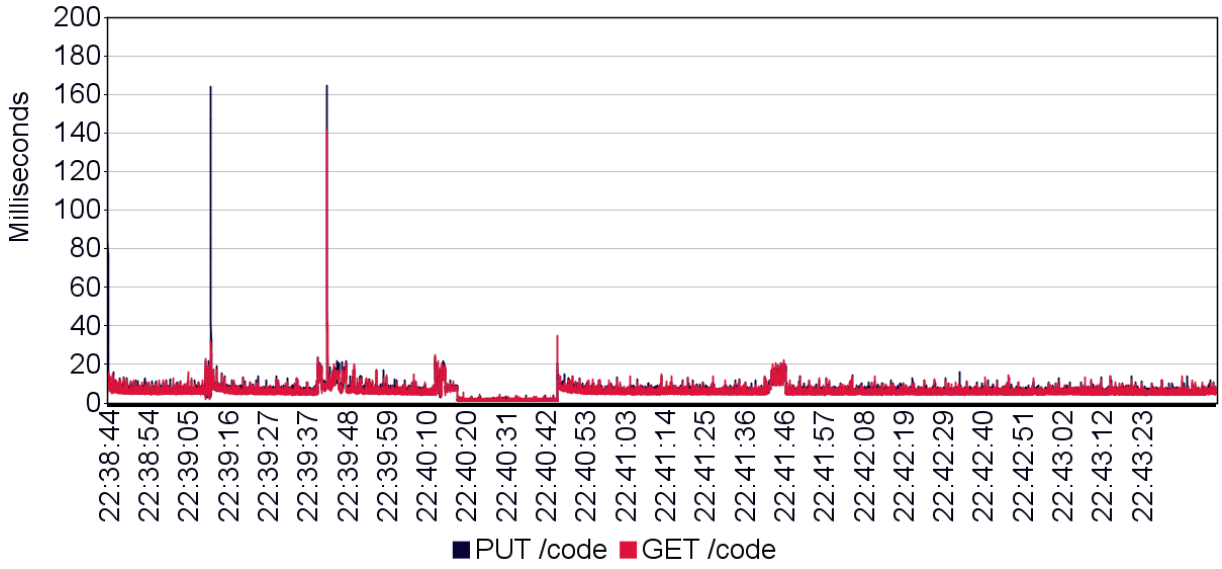
Já na falha longa, que ocorre 90 segundos após o início, há um aumento substancial na quantidade de códigos de falha retornados pelo servidor, o que é representado pela parte íngreme da linha azul na figura 8. Isso se traduz pelo disjuntor em estado aberto ou meio aberto rejeitando requisições. Essa situação só muda quando o serviço externo volta a funcionar e o disjuntor detecta no estado meio aberto que uma quantidade limitada de requisições foi enviada e obteve sucesso. Isso permite transicionar para o estado fechado, o que significa a volta ao estado normal.

Por fim, as estatísticas demonstradas da gráfico da Figura 9 revelam que esse padrão apresentou uma taxa de erro de 27.76%. Porém, apesar desse valor inicialmente parecer algo negativo, não é, pois o objetivo do padrão não é minimizar a quantidade de respostas fracassadas, mas sim aliviar a quantidade de requisições ao serviço externo

Figura 7 – Latência dos testes com e sem o uso do disjuntor



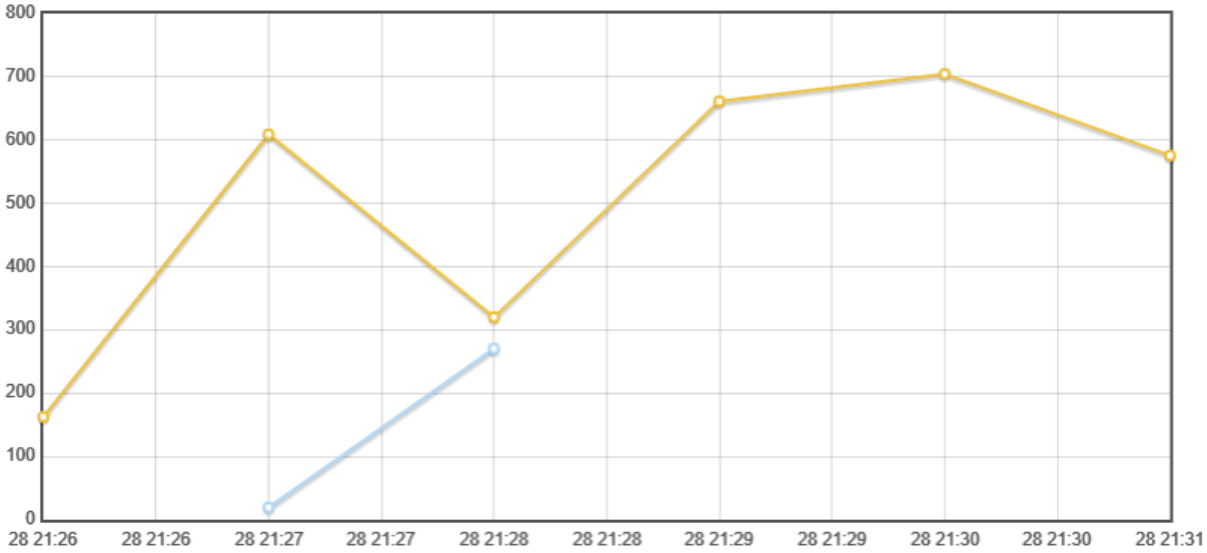
(a) Sem uso do disjuntor.



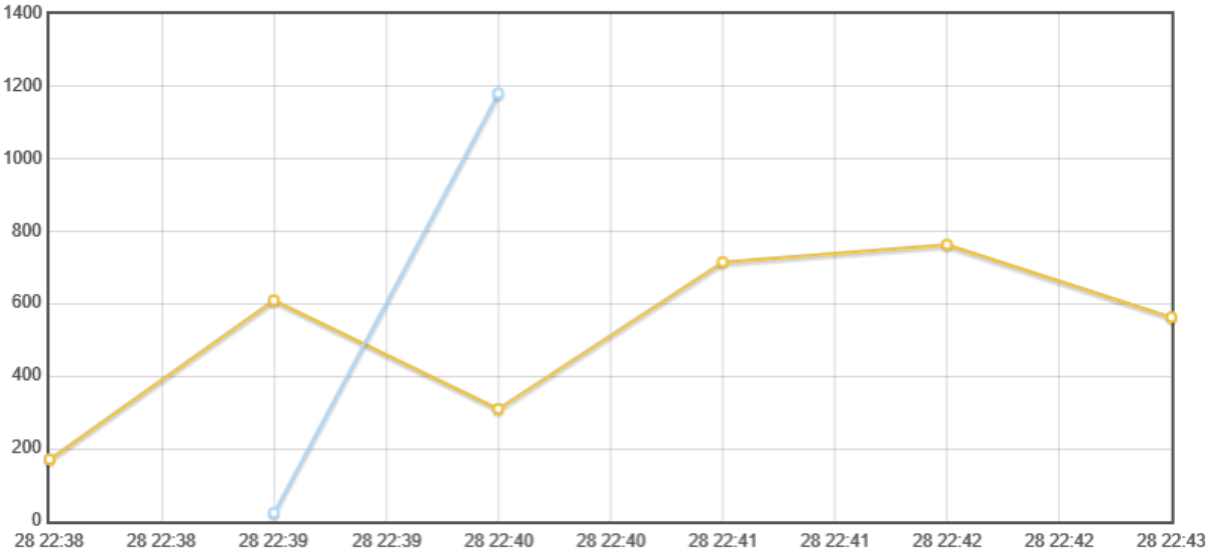
(b) com o uso do disjuntor.

Fonte: o autor

Figura 8 – Vazão dos testes com e sem o uso do disjuntor



(a) Sem uso do disjuntor. Linha azul: código 500. Linha amarela: código 200



(b) com o uso do disjuntor. Linha azul: código 503. Linha amarela: código 200

Fonte: o autor

Figura 9 – Estatísticas dos testes com e sem o uso do disjuntor

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do disjuntor.

Requests	Executions			Response Times (ms)						Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**260316**	**72265**	**27.76%**	**4.54**	**0**	**165**	**5.00**	**6.00**	**7.00**	**9.00**	**868.40**
GET /code	130157	36132	27.76%	4.46	0	159	5.00	6.00	7.00	9.00	434.40
PUT /code	130159	36133	27.76%	4.63	0	165	5.00	6.00	8.00	9.00	434.20

(b) com o uso do disjuntor.
Fonte: o autor

quando este apresenta estado de falha. Para fazer isso, o disjuntor passa alguns segundos sem enviar requisições, o que justifica essa taxa de erro maior.

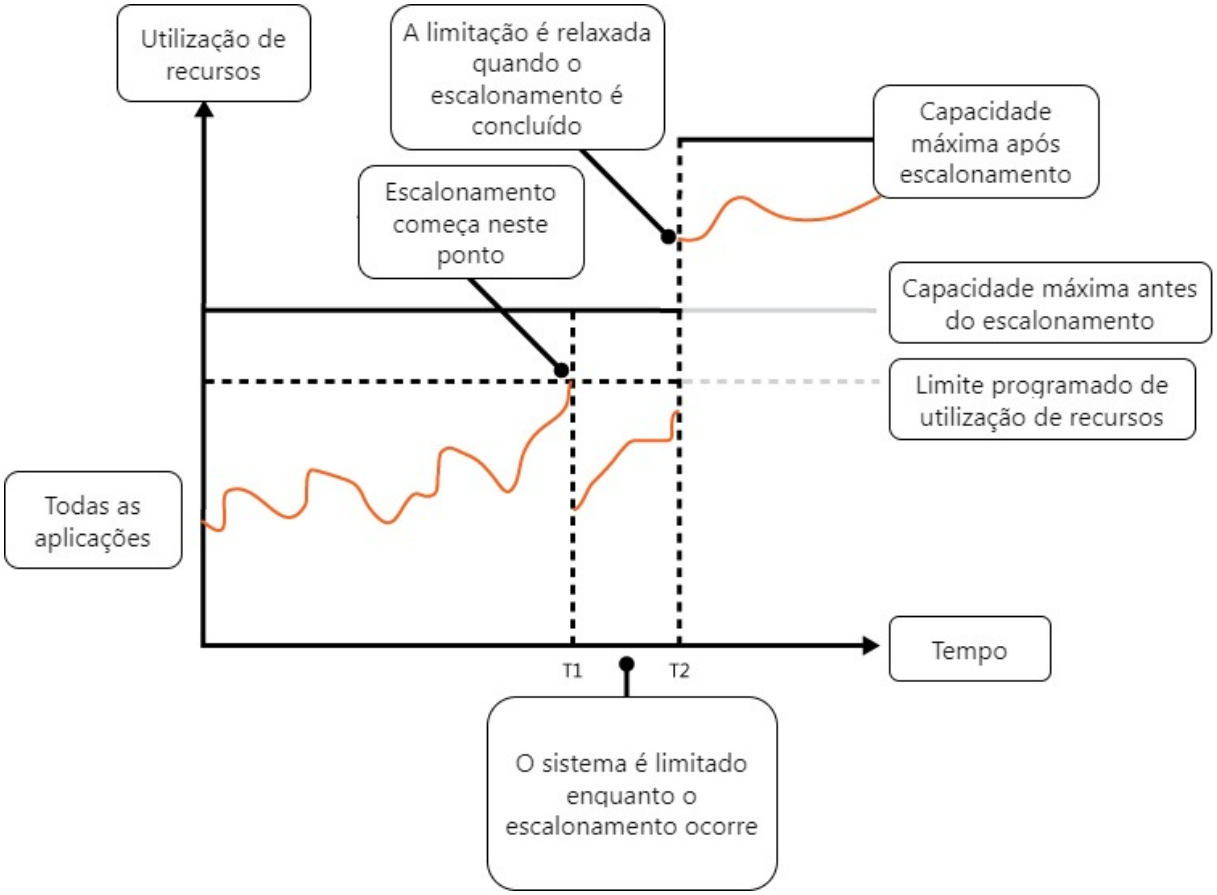
3.3 *THROTTLING*

Há sistemas em que a atividade dos usuários pode variar com o tempo. Isso pode ocorrer por exemplo pelo uso do sistema em horário comercial ou então em atividades específicas, como por exemplo um sistema de pagamento de funcionários no início do mês. Nesses casos, é preciso lidar com esse pico de atividade de forma que o excesso de atividade não sobrecarregue o sistema de forma a causar panes ou lentidão (VEGESNA, 2022). Caso isso aconteça as consequências podem ser graves, como o descumprimento de um acordo de nível de serviço, ou a interrupção de outros serviços por quais o servidor sobrecarregado também é responsável. Para evitar tais problemas existe o *Throttling*.

O *Throttling* é um padrão que consiste na limitação do número de chamadas para uma API. Essa limitação pode englobar um conjunto de *endpoints*, um *endpoint* específico ou até mesmo uma combinação de *endpoint* com um verbo HTTP específico. Essa restrição pode ser voltada a um endereço IP específico (LAMBA, 2019), realizando uma limitação voltada a um determinado usuário, ou então pode ser geral a qualquer usuário.

Este padrão funciona muito bem quando combinado com o escalamento automático

Figura 10 – *Throttling* usado com escalamento automático



Fonte: o autor

da aplicação. À medida que a atividade dos usuários aumenta, chega-se a um ponto que o sistema ficaria sobrecarregado, mas o padrão impede isso retornando um erro (MICROSOFT, 2024). Enquanto isso, novas máquinas virtuais são iniciadas. Durante o período de inicialização o servidor retorna erro propositalmente às requisições após ultrapassar o limite estabelecido. Quando a inicialização das novas instâncias é concluída, a carga de requisições é distribuída, e o sistema pode responder ao pico de demanda de forma controlada. A Figura 10 exemplifica esse uso.

3.3.1 Estudo de caso

Para estudar esse padrão foi criada uma cópia da aplicação principal com uma adaptação: foi inserido um mecanismo de *throttling* no *endpoint* */GET*. Isso foi feito com uso da biblioteca express-rate-limit (EXPRESS-RATE-LIMIT. . . , s.d.).

Nesse trecho é criado um *middleware* que exercerá a limitação e, em seguida, ele é aplicado no roteador com o *.use()*.

```
const limiter = rateLimit({
  windowMs: 60 * 1000,
  max: 10,
  message: 'Too many requests from this IP, please try again after a
     minute',
  skip: (req) => req.method !== 'PUT'
});

app.use(limiter)
```

As configurações descrevem como a limitação deve ocorrer:

windowMs : janela de tempo usada na contagem das requisições;

max : número máximo de requisições permitido na janela de tempo definida;

message : mensagem de erro retornada ao usuário após exceder o limite de requisições;

skip : função que define condições em que a limitação não é aplicada. Nesse caso foi escolhido casos em que o verbo http seja diferente de *PUT*.

Para testar esse padrão foram feitos dois experimentos, ambos sem a simulação de falhas. Um experimento usou um servidor sem limitador e o outro usou o *throttling*. Para simular o sobrecarregamento, no *endpoint* */PUT* a lógica de alterar dados foi substituída por um *loop* que conta até 100 milhões, conforme o código a seguir:

```
app.put("/code/:id", async (req, res) => {
  try {
    console.log("PUT", req.params.id, req.body);
    let result = 0;
    for (let i = 0; i < 1e8; i++) {
      result += i;
    }
    res.json({result});
    ...
});
```

Figura 11 – Estatísticas do servidor com processos lentos com e sem o *throttling*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**875**	**0**	**0.00%**	**24195.11**	**339**	**120809**	**24158.00**	**24177.00**	**26752.60**	**75310.84**	**2.70**
GET /code	15	0	0.00%	81969.73	45696	120809	84215.00	106319.60	120809.00	120809.00	0.05
PUT /code	860	0	0.00%	23187.41	339	47562	24157.00	24176.00	24179.00	34074.08	2.66

(a) Sem uso do *throttling*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**680580**	**671762**	**98.70%**	**29.85**	**2**	**3923**	**26.00**	**30.00**	**31.00**	**129.00**	**2270.80**
GET /code	8768	0	0.00%	136.55	12	3923	132.00	143.00	157.00	164.00	29.25
PUT /code	671812	671762	99.99%	28.46	2	3833	26.00	29.00	30.00	33.00	2241.59

(b) com o uso do *throttling*
Fonte: o autor

A Figura 11, que compara as métricas dos testes com e sem o uso desse padrão, demonstra que após o uso houve um aumento significativo das requisições por segundo do *endpoint* não sobrecarregado, o */GET*, já que o valor dessa métrica passou de 0.05 para 29.25. Isso permite concluir que o padrão fez com que o uso excessivo de uma parte do sistema não sobrecarregasse as outras. Por outro lado, é possível notar uma taxa de erro maior, pois na coluna *Error* há passa a haver uma taxa de 98.70% com o uso do padrão, enquanto sem o padrão essa taxa é 0%. Esses erros são esperados, e são as respostas de código 429 retornadas que indicam que a requisição foi rejeitada por ultrapassar o limite de requisições por período.

3.4 SIDECAR

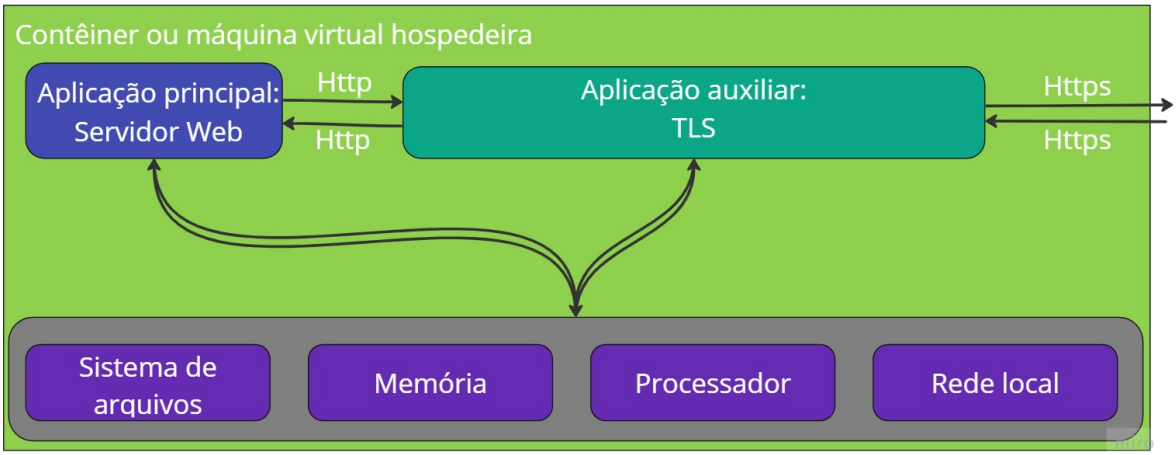
Frequentemente, um serviço principal precisa de serviços auxiliares, e é comum que esses serviços auxiliares sejam agregados na mesma base de código do serviço principal. Mas em alguns casos, é conveniente que os serviços auxiliares sejam independentes da linguagem de programação ou do ambiente de execução do serviço principal, já que isso traz diversas vantagens, como a possibilidade de reuso desse serviço auxiliar (BURNS; OPPENHEIMER, 2016). Uma possibilidade para implantação desses serviços auxiliares seria a criação de um novo serviço separado, a outra alternativa é implantar no mesmo *host*, em um contêiner separado, o que constitui o *sidecar*.

O *sidecar* se baseia na implantação de uma aplicação auxiliar segregada da principal usando um processo ou contêiner separado para que haja isolamento e encapsulamento. Esse padrão também é responsável por permitir que a aplicação principal e a auxiliar sejam compostas por tecnologias heterogêneas (MICROSOFT, 2023f). A nomenclatura se dá devido ao fato do componente lembrar um *sidecar* (que é uma cabine que é conectada a uma moto para que um ocupante extra seja transportado)

Esse padrão arquitetural é formado por dois contêiners que são alocados em um mesmo servidor. O primeiro contêiner é a aplicação principal, que contém as funcionalidades principais. O segundo contêiner é uma aplicação auxiliar, que contém funcionalidades complementares ao contêiner principal (BURNS, 2018). É comum que o principal contêiner sequer tenha ciência da existência do auxiliar.

Essa arquitetura na prática pode ser exemplificada por um servidor web legado que ainda possui o protocolo HTTP como padrão e é necessário que esse protocolo seja substituído pelo HTTPS, mas o servidor que o hospeda funciona com uma versão antiga do Linux que não possui compatibilidade das aplicações que provêm o HTTPS. Uma solução é implementar o *sidecar*, onde o contêiner principal é o servidor web que fica configurado para funcionar na rede local. Já o auxiliar é um redirecionador, que converte pacotes vindos da rede externa de HTTPS e os envia no formato HTTP para o servidor web hospedado no contêiner principal e *vice versa*.

Figura 12 – Funcionamento do *Sidecar* destinado a TLS



Fonte: o autor

A segregação das funcionalidades auxiliares em um contêiner separado apresenta diversas vantagens frente a implementá-las no contêiner principal. A primeira delas é o ajuste individual de configurações. Por exemplo, o contêiner principal pode ser ajustado para ocupar grande parte do processamento de CPU e o auxiliar funcionar apenas nos ciclos ociosos. Outra vantagem é a reusabilidade, pois os serviços do contêiner auxiliar podem ser usados por diversos outros. Além disso, a implementação do contêiner auxiliar passa a

ser independente, e ficam melhor divididas as tarefas entre as equipes de desenvolvimento. Por fim, é possível que a testabilidade seja simplificada, já que o sistema fica dividido em componentes menores.

3.4.1 Estudo de caso

Foi desenvolvido um contêiner auxiliar que exerce a função de cache. O objetivo é que esse sidecar reduza a quantidade de erros caso o serviço externo esteja indisponível. O servidor principal é o mesmo usado como *benchmark* para os outros testes. O cache é feito em memória volátil com a biblioteca Redis (NPM, INC., 2024a), uma implementação Node.js do servidor noSQL Redis. No experimento, o *Sidecar* é usado apenas para operações de consulta. Dessa forma, o cliente faz chamadas */get* para o *sidecar*, enquanto as requisições */put* são feitas diretamente para o serviço externo.

O *caching* consiste em armazenar registos que são frequentemente acessados em memória, o que permite que o acesso seja mais rápido quando comparado ao acesso em disco (BLANKSTEIN; SEN; FREEDMAN, 2017). O Redis é um software amplamente utilizado em memória, e armazena os dados no formato chave-valor. No código a seguir é implementado o *endpoint* /GET com uso do *caching*, sendo realizadas as seguintes operações:

1. É verificado se o item buscado está armazenado em memória;
2. Caso o item esteja, ele é retornado;
3. Caso não esteja, ele é buscado no serviço externo, retornado, e adicionado ao *caching* em seguida.

```
app.get('/code/:id', async (req, res) => {
  try {
    const data = await redisClient.get(req.params.id)
    if (data) {
      res.json(JSON.parse(data));
    } else {
      try {
        const response = await instance.get("code/"+req.params.id);
        if (response.data) {
          redisClient.set(req.params.id, JSON.stringify(response.data));
        }
        res.json(response.data);
      } catch (error) {
        console.log(error)
        res.status(503).json({ error: 'External service unavailable' });
      }
    }
  } catch (error) {
    console.log(error)
    res.status(500).json({ error: 'Internal server error' });
```

```
  }
});
```

A Figura 13 demonstra que o uso desse padrão reduziu a zero o percentual de erros no *endpoint /get*, já que nos momentos de falha o servidor *cache* foi capaz de retornar os valores de seu banco de dados em vez de buscar o resultado no servidor em estado de falha. Além disso, a métrica de erros totais foi reduzida de 8.73% para 4.28%.

Figura 13 – Comparação de resultados estatísticos com e sem o uso do *Sidecar*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do *Sidecar*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**203794**	**8720**	**4.28%**	**5.82**	**2**	**182**	**5.00**	**8.00**	**8.00**	**11.00**	**679.86**
GET /code	101896	0	0.00%	4.56	2	19	4.00	6.00	6.00	9.00	340.00
PUT /code	101898	8720	8.56%	7.08	2	182	7.00	8.00	8.00	11.00	339.94

(b) com o uso do *Sidecar*
Fonte: o autor

3.5 OUTROS PADRÕES

A computação distribuída possui muitas aplicações e, dessa maneira, uma variedade imensa de desafios existe e exige a criação de padrões de projetos para cada tipo de problema. Existem muitos outros padrões. Alguns deles voltados à tolerância a falhas (MICROSOFT, 2023b):

***Rate Limit Pattern*:** em casos em que há integração com sistemas com limitação de serviço (*throttling*), é importante o cliente respeitar as limitações do servidor para evitar erros desse mecanismo (AZURE, M., Accessed: June 10, 2024), pois isso gera um problema adicional de tratar as requisições que retornaram erro.

Para tratar esse problema uma alternativa é usar um serviço de filas de mensagens capaz de receber o grande número de requisições enviado. Esse serviço agrupa essas requisições em lotes menores, denominados *job processors*. Esses agrupamentos então são enviados em um ritmo menor ao serviço limitado, de forma a não ultrapassar seus limites. O uso desse padrão é altamente indicado em casos onde há operações repetitivas, como em casos de processamentos de grandes volumes de dados.

***Bulkhead*:** O termo *Bulkhead* em seu contexto original é um termo náutico que significa uma parede dentro do casco do navio que gera divisórias, e assim evita o alagamento completo caso entre água em alguma das repartições (MICROSOFT, 2023a). De forma análoga, na computação o padrão consiste na separação das operações em grupos para que, caso um grupo falhe, o resto continue funcionando. Em termos práticos, ocorre a criação de grupos de *threads* isolados quando há conexão com diferentes microsserviços (ALASHQAR; BALASOORIYA; KAFROUNI, 2022), o que permite reduzir o impacto da falha em um microsserviço nos outros.

***Queue-Based Load Leveling*:** Esse padrão consiste no uso de uma fila entre um serviço e o cliente para evitar a sobrecarga desse serviço quando há picos de demanda (MICROSOFT, 2023d). Para atingir esse objetivo, há vários componentes, os produtores e consumidores, que respetivamente enviam e processam as tarefas, e uma fila de mensagens responsável por receber as mensagens dos produtores e enviar aos consumidores respeitando seus limites.

Esse padrão permite controlar os sistemas quando há picos de demanda, através do desacoplamento, garantindo maior resiliência e tolerância a falhas. O lado negativo é o aumento na complexidade gerado.

***Eleição de líder*:** Em sistemas distribuídos, podem haver casos em que vários serviços precisam de uma coordenação para que o sistema funcione de forma coesa (MICROSOFT, 2023c). Um exemplo seriam dois serviços que, quando invocada uma função, modificam um mesmo recurso. Assim, é necessário impedir que um serviço sobrescreva as modificações feitas pelo outro. Outra situação seria dividir uma tarefa em partes menores que são executadas de forma paralela pelos serviços. Isso requer que haja uma coordenação para juntar os resultados no final.

Para resolver o problema supracitado, o padrão da eleição de líder propõe que um dos serviços seja responsável por liderar um conjunto (FETZER; CRISTIAN, 1999). Para que isso funcione de forma apropriada, o algoritmo deve levar em conta problemas que podem inativar o nó líder, desde falhas no seu processo até problemas de rede. Logo, caso haja problemas com o líder, o sistema deve realizar uma nova eleição.

Existem variadas formas de eleição de líder. Algumas mais simples envolvem simplesmente escolher o processo com o *ID* mais baixo. Outras soluções envolvem uso de exclusão mútua (ou seja, os serviços disputam para se tornarem líderes e o primeiro a conseguir, por meio de um mecanismo de trava, garante a liderança).

4 TRABALHOS RELACIONADOS

Esse capítulo tem como objetivo apresentar trabalhos relacionados. Foram buscados trabalhos que realizam melhorias em padrões voltados à tolerância a falhas, obras que estudam problemas relacionados à segurança na MSA e propostas para aumento de disponibilidade em sistemas baseados em serviços com reinicialização frequente.

4.1 DESAFIOS E SOLUÇÕES DE SEGURANÇA TRAZIDOS COM O USO DA MSA

No que tange aos problemas relacionados a cibersegurança em arquitetura de microsserviços (RUDRABHATLA, 2020) explica desafios de cibersegurança que não eram presentes na arquitetura monolítica. Dada sua natureza distribuída da MSA, passa a haver novos problemas, como aumento da superfície de ataque, maior complexidade no controle de autorização e segurança dos dados.

O objetivo do artigo é explicar os novos desafios de cibersegurança trazidos pela MSA e também propõe padrões e boas práticas para solucionar tais problemas. Para fazer isso, o trabalho explica seis problemas envolvendo cibersegurança trazidos com a MSA:

4.1.1 Segurança na superfície de ataque

A arquitetura de microsserviços oferece muitas vantagens, como segregação de responsabilidades, isolamento, ciclos de desenvolvimento mais rápidos, escalabilidade horizontal entre outros. Por outro lado, essa arquitetura apresenta desafios de segurança, já que a camada de lógica está dividida, havendo uma maior superfície de ataque. Outro motivo que aumenta a superfície de ataque é a presença de vários *front ends* conversando com o *back end*. Por fim, a crescente onda de dispositivos IoT exige autenticação sem intervenção humana, dificultando a segurança.

No modelo tradicional, onde se tinha o *front end* e *back ends* unificados, usava-se agentes como o SiteMinder que atuavam como camada de autenticação. Com o uso de conteinerização em nuvens públicas e a presença de vários *front ends* essa técnica perdeu a eficiência quando usada na MSA.

A solução para esse problema consiste no uso de novos protocolos de segurança, que funcinam com APIs REST e JSON. Entre os mais notáveis estão o JWT e Oauth 2.0. As requisições provenientes dos front ends são interceptadas por um *API Gateway*, que é responsável por garantir a autenticação e direcionar as requisições ao microsserviço adequado. No protocolo JWT o usuário, ao se autenticar, recebe um *access token* de curta duração e um *refresh token* que é usado para obter um novo *access token* quanto este se expirar. Um vez autenticado, o usuário usa o `access token` nos cabeçalhos das requisições para obter permissão. A autenticação do usuário e a validação de *tokens* é feita por um servidor de autenticação que se comunica com o *API Gateway*.

4.1.2 Segurança em profundidade

Um outro problema na MSA é que alguns microsserviços podem ter diferentes requisitos de segurança que outros por exemplo. Um conjunto A pode ser destinado ao acesso público enquanto um conjunto B ser destinado apenas a administradores. A solução consiste na implementação de mecanismos de *defence-in-depth*, ou seja, não se tem apenas o *API gateway* realizando o controle de acesso, mas além dele tem-se outros filtros destinados ao controle de microsserviços específicos.

4.1.3 Princípio do acesso mínimo

Um padrão de projeto comum na arquitetura de microsserviços é o banco de dados por serviço, ou seja, além do microsserviço ser independente, seu banco de dados também é. Assim, os serviços precisam se comunicar uns com os outros, o que pode ser complexo e o mecanismo de permissões também se torna desafiante. Para contornar a situação, o aconselhado é usar o princípio do privilégio mínimo, ou seja, a API de um microsserviço concede ao usuário as mínimas necessárias, e expande a medida que surgem novas demandas.

4.1.4 Criptografia onipresente

Apesar de ser possível decompor microsserviços eficientemente, usando técnicas como o princípio da responsabilidade única, a intercomunicação entre eles é inevitável. Assim os dados são armazenados por tecnologias heterogêneas, e a segurança da informação se torna um desafio. Para contornar tal problema a solução é a encriptação onipresente, ou seja, encriptar o dado não apenas em transporte, com SSL mas também enquanto ele está em repouso, seja em armazenamento de arquivos ou em sistema de cache usando tecnologias como a criptografia transparente volumétrica e senhas que rotacionam em um prazo curto de dias.

4.1.5 *DevSecOps*

Como na MSA os contêiners são heterogêneos, independentes e sem estado, é desafiante criar um padrão e *framework* uniforme de *logs* e isso dificulta a identificação de ameaças. Para contornar esse desafio é preciso ter uma solução que concentre os *logs*. Além disso, é primordial o uso de técnicas de *DAST* (teste dinâmico de segurança da aplicação) e que as técnicas de *devops* não se façam isoladas, mas integradas às de segurança, formando o chamado *DevSecOps*

Por fim, o trabalho expõe tendências de soluções que estão sendo pesquisadas envolvendo segurança de MSA. Uma tendência é a implementação de inteligência artificial para detectar o uso de padrões de requisições, respostas e interações e ativar disjuntores

para parar o ataque. Outra área de estudo é a implementação de autenticação multi fator em dispositivos de internet das coisas, em que não há interação humana.

Como semelhanças com esse trabalho, ambos os trabalhos estudam desafios relacionados à MSA. A diferença é que o trabalho estuda problemas e soluções envolvendo segurança da informação e este artigo tem como foco problemas e soluções envolvendo tolerância a falhas.

4.2 AUMENTO DA DISPONIBILIDADE EM SISTEMAS BASEADOS EM MSA COM *DEPLOY* FREQUENTE

O trabalho (RASHEEDH; SARADHA, 2022) aborda o problema da reinicialização frequente de microsserviços causada pelas metodologias ágeis e propõe uma solução. Com a difusão de metodologias ágeis para gerenciamento de projetos, as implantações de novas funcionalidades ou correções de software passaram a ocorrer com maior frequência. Essas implantações exigem que os serviços sejam reinicializados, o que é um problema para aplicações que prometem *downtime* zero.

O artigo propõe um *framework* para implantações em MSA que não necessita a reinicialização dos microsserviços. A proposta envolve o uso das práticas de desenvolvimento de software *SOLID* e o uso de um motor de mistura que funde os novos objetos com o código existente. O *SOLID* é uma sigla que engloba os seguintes conceitos:

***Single Responsability*:** cada serviço deve ser destinado a uma tarefa específica.

***Open/Close*:** as entidades de software deve estar abertas a extensão e fechadas para modificação.

***Liskov Substitution*:** um objeto de um tipo T pode ser substituído por um objeto S quando S é um subtipo de T, sem que as características do sistema mudem.

***Interface Segregation*:** Um objeto deve depender das dependências que implementa sem que seja forçado a implementar métodos que não são requeridos.

***Dependency Inversion*:** Módulos de alto nível não dependem dos de baixo nível. Ambos dependem de abstrações. Abstrações não dependem de detalhes, ocorre o oposto.

As práticas SOLID permitem a independência e o desacoplamento entre as partes do sistema e, dessa forma, permitem a fusão dinâmica. Seguindo essas práticas é possível adicionar funcionalidades sem ser necessária a reinicialização do sistema.

A solução consiste na existência de um controlador central, o cliente, que interage com as interfaces, os microsserviços, e um conjunto de serviços associados no controlador. Assim, ocorre a criação de serviços de acordo com o *workflow* e os microsserviços são fundidos com base nas necessidades do usuário. Quando ocorre uma fusão, o sistema deve

passar por uma transformação desatada do estado existente para o novo estado, sem que haja reinicialização. Dessa forma, tem-se todos os serviços completamente isolados.

O uso desse *framework* resulta em um desempenho maior, índice de erro menor e uso de CPU mais baixo quando comparado com arquitetura monolítica ou microsserviços tradicional.

Tanto este trabalho como o artigo descrito buscam soluções para problemas advindos da MSA. O foco da obra em reduzir o *downtime* a encaixa na área de estudo que visa solucionar problemas de disponibilidade, que, apesar de ser uma área de estudo próxima, não é exatamente a mesma deste artigo, cujo estudo é voltado para área de tolerância a falhas na MSA.

4.3 MELHORIA NO PADRÃO DISJUNTOR COM O USO DE CADEIAS DE MARKOV

A tese de mestrado (ADDEEN, 2019) aborda o problema da comunicação entre os microsserviços, advindo da intercomunicação via rede. Uma situação que pode ocorrer é um serviço ficar esperando infinitamente por uma resposta, e uma das soluções é o padrão disjuntor, que usa um tempo limite associado a máquinas de estado para lidar com falhas. Porém, o tempo de espera precisa ser dinâmico e é um desafio a escolha deste tempo limite.

Esse trabalho correlato propõe um padrão que une o disjuntor à cadeias de Markov, o que busca resolver o desafio da definição do tempo limite. A cadeia de Markov é uma teoria matemática de mudança de um estado para outro. O espaço de estado, que já é previamente conhecido, é usado para usado para gerar as cadeias. Para prever o próximo estado o cálculo é feito com base apenas no estado atual e com regras probabilísticas, sem considerar os estados anteriores.

O objetivo do uso das Cadeias de Markov é substituir o tempo limite. As cadeias de Markov usam apenas o estado atual e uma tabela de combinações para calcular o próximo estado, usando um modelo probabilístico com base nos tempos de resposta.

O resultado é que o disjuntor proposto é 43% mais rápido que o disjuntor tradicional. Para calcular essa performance realizou-se 10 testes para mensurar o tempo de recuperação do sistema usando o disjuntor tradicional e o modelo proposto. A métrica consiste na diferença de tempo médio (tempo médio do disjuntor modificado subtraída pelo tempo do disjuntor tradicional) sobre o tempo do disjuntor tradicional.

Essa obra correlata possui uma semelhança significativa já que os dois trabalhos estudam o padrão disjuntor. A diferença é que enquanto a tese é um estudo vertical e aprofunda-se no padrão disjuntor, esse trabalho busca um estudo horizontal, comparando o disjuntor a outros padrões.

5 CONCLUSÃO

Esse trabalho contextualizou os padrões de projeto, padrões arquiteturais e a arquitetura de microsserviços. A obra explicou também o funcionamento da arquitetura de microsserviços e suas vantagens. Além disso, apresentou a área de tolerância a falhas em sistemas orientados a serviços e sua importância.

Foi explicado o funcionamento do padrão de projeto *retry* e o seu algoritmo de cálculo de tempo entre as tentativas, o *exponential backoff*. O trabalho também sugeriu em que tipo de situação ele pode ser empregado. Ademais, demonstrou seu funcionamento na prática com um estudo de caso, que concluiu que o padrão pode tratar falhas transientes, com o ônus de uma latência maior.

Foram apresentados o padrão disjuntor, sua máquina de estado, os algoritmos de cada estado e o contexto em que é pertinente haver uso desse padrão. Um estudo de caso prático demonstrou seu uso, provando que ele é capaz de evitar a sobrecarga de um sistema em estado de falha, ao custo do cliente ter uma taxa de erro maior devido ao tempo necessário para transição de estados.

O padrão *throttling* também foi abordado. Foi explicado o funcionamento desse padrão e como ele pode melhorar a estabilidade de sistemas, com ênfase em sua aplicação em situações de escalamento automático. Um estudo de caso demonstrou que o padrão permite evitar que o uso intensivo de uma parte do sistema interfira no resto, sob o ônus de limitar usuários que queiram fazer uso extremo de um serviço específico.

O funcionamento do *Sidecar* foi explicado, além das situações em que é vantajoso seu uso e das motivações que justificam sua implantação. Foi feito um estudo de caso no qual esse padrão foi usado para encapsular um servidor de *caching*, o que permitiu reduzir a taxa de erro em 49% ao retornar respostas em *cache* em vez de buscá-las em um serviço falho.

Além dos padrões supracitados, houve uma análise teórica dos padrões Eleição de líder, *Queue-Based Load Leveling*, *Bulkhead* e *Rate Limit*. Foi apresentado o funcionamento e o contexto em que o uso desses padrões é pertinente.

5.1 TRABALHOS FUTUROS

Ampliar os estudos práticos Adicionar estudos de caso com experimentos para os padrões Eleição de líder, *Queue-Based Load Leveling*, *Bulkhead* e *Rate Limit*.

Combinar padrões Criar estudos de casos mais complexos combinando padrões, por exemplo, combinar o disjuntor com o *retry*, o que pode evitar a sobrecarga de um serviço falho e, ao mesmo tempo, lidar com falhas transientes.

REFERÊNCIAS

ADDEEN, Hajar Hameed. **A Dynamic Fault Tolerance Model for Microservices Architecture**. [*S.l.*]: South Dakota State University, 2019.

ALASHQAR, Abdelkareem; BALASOORIYA, Janaka; KAFROUNI, Said. Examining the Design Patterns of Microservices for Achieving Performance Quality Tactics. **ResearchGate**, 2022. Accessed: 2024-06-10.

ALMADI, Sara HS; HOOSHYAR, Danial; AHMAD, Rodina Binti. Bad smells of gang of four design patterns: A decade systematic literature review. **Sustainability**, MDPI, v. 13, n. 18, p. 10256, 2021.

APACHE SOFTWARE FOUNDATION. **Apache JMeter**. [*S.l.*: *s.n.*], 2024. Accessed: 2024-05-22. Disponível em: https://jmeter.apache.org/.

AZURE. **Circuit Breaker pattern – Azure Architecture Center**. [*S.l.*], 2023. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker.

AZURE, Microsoft. **Rate Limiting Pattern**. [*S.l.*: *s.n.*], Accessed: June 10, 2024. https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting-pattern.

BLANKSTEIN, Aaron; SEN, Siddhartha; FREEDMAN, Michael J. Hyperbolic caching: Flexible caching for web applications. *In*: 2017 USENIX Annual Technical Conference (USENIX ATC 17). [*S.l.*: *s.n.*], 2017. P. 499–511.

BURNS, Brendan. **Designing Distributed Systems**. [*S.l.*]: O'Reilly, 2018.

BURNS, Brendan; OPPENHEIMER, David. Design patterns for container-based distributed systems. *In*: 8TH USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). [*S.l.*: *s.n.*], 2016.

CLOUD, Google. **Exponential Backoff**. [*S.l.*: *s.n.*], 2024. https://cloud.google.com/memorystore/docs/redis/exponential-backoff. Accessed: 2024-05-22.

DEACON, John. Model-view-controller (mvc) architecture. **Online][Citado em: 10 de março de 2006.] http://www.jdl.co.uk/briefings/MVC. pdf**, v. 28, 2009.

DIGITAL, Olhar. **Bradesco, Banco do Brasil e Inter apresentam instabilidade nesta quarta (29)**. 2024. Disponível em: https://olhardigital.com.br/2024/05/29/pro/bradesco-banco-do-brasil-e-inter-fora-do-ar-nesta-quarta-29/#google_vignette.

EXPRESS-RATE-LIMIT - npm. [*S.l.*: *s.n.*]. https://www.npmjs.com/package/express-rate-limit. Accessed: May 28, 2024.

FETZER, Christof; CRISTIAN, Flaviu. A highly available local leader election service. **IEEE Transactions on Software Engineering**, IEEE, v. 25, n. 5, p. 603–618, 1999.

FOROUZAN, Behrouz A; FEGAN, Sophia Chung. **Protocolo TCP/IP-3**. [*S.l.*]: AMGH Editora, 2009.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. [*S.l.*]: Addison-Wesley Professional, 1994.

GIL, Henrique. A passagem da Web 1.0 para a Web 2.0 e... Web 3.0: potenciais consequências para uma «humanização» em contexto educativo. **Educatic: boletim informativo**, Educatic, p. 1–2, 2014.

GREGOR HOHPE, Bobby Woolf. **Messaging Patterns Overview**. [*S.l.*], 2023. Disponível em: https://www.enterpriseintegrationpatterns.com/patterns/messaging/.

GROOMBRIDGE, David. **Gartner Top 10 Strategic Technology Trends for 2023**. [*S.l.*], 2023. Disponível em: https://gartner.com/en/articles/gartner-top-10-strategic-technology-trends-for-2023.

HIBLER, Mike; RICCI, Robert; STOLLER, Leigh; DUERIG, Jonathon; GURUPRASAD, Shashi; STACK, Tim; WEBB, Kirk; LEPREAU, Jay. Large-scale virtualization in the emulab network testbed. *In*: 2008 USENIX Annual Technical Conference (USENIX ATC 08). [*S.l.*: *s.n.*], 2008.

KEYMETRICS, INC. **PM2: Advanced, production process manager for Node.js**. [*S.l.*: *s.n.*], 2024. https://pm2.keymetrics.io/. Accessed: 2024-05-22.

LAMBA, Anil. API Design Principles & Security Best Practices–Accelerate your business without compromising security. Cybernomics, 2019.

LEE, Peter Alan; ANDERSON, Thomas; LEE, Peter Alan; ANDERSON, Thomas. **Fault tolerance**. [*S.l.*]: Springer, 1990.

LIU, Yue; LU, Qinghua; XU, Xiwei; ZHU, Liming; YAO, Haonan. Applying design patterns in smart contracts: A case study on a blockchain-based traceability application. *In*: SPRINGER. BLOCKCHAIN–ICBC 2018: First International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings 1. [*S.l.*: *s.n.*], 2018. P. 92–106.

MANAGER, Node Package. **Opossum: A Node.js Circuit Breaker**. [*S.l.*: *s.n.*], 2024. https://www.npmjs.com/package/opossum. Accessed: 2024-05-22.

MICROSOFT. **Bulkhead pattern - Cloud Design Patterns**. [*S.l.*: *s.n.*], 2023. Accessed: 2024-06-10. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead.

MICROSOFT. **Cloud design patterns that support reliability**. [*S.l.*], 2023. Disponível em: https://learn.microsoft.com/en-us/azure/well-architected/resiliency/reliability-patterns.

MICROSOFT. **Leader Election pattern**. [*S.l.*], 2023. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/leader-election.

MICROSOFT. **Queue-based load leveling pattern - Cloud Design Patterns**. [*S.l.*: *s.n.*], 2023. Accessed: 2024-06-10. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling.

MICROSOFT. **Retry pattern**. [*S.l.*], 2023. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/retry.

MICROSOFT. **Sidecar pattern**. [*S.l.*], 2023. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar.

MICROSOFT. **Throttling pattern**. [*S.l.*], 2024. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling.

MONTESI, Fabrizio; WEBER, Janine. Circuit breakers, discovery, and API gateways in microservices. **arXiv preprint arXiv:1609.05830**, 2016.

MONTESI, Fabrizio; WEBER, Janine. From the decorator pattern to circuit breakers in microservices. *In*: PROCEEDINGS of the 33rd Annual ACM Symposium on Applied Computing. [*S.l.*: *s.n.*], 2018. P. 1733–1735.

NPM, INC. **redis**. [*S.l.*: *s.n.*], 2024. Accessed: 2024-05-28. Disponível em: https://www.npmjs.com/package/redis.

NPM, INC. **retry - npm**. [*S.l.*: *s.n.*], 2024. https://www.npmjs.com/package/retry. Accessed: 2024-05-22.

RASHEEDH, JA; SARADHA, S. Design and Development of Resilient Microservices Architecture for Cloud Based Applications using Hybrid Design Patterns. **Indian J. Comput. Sci. Eng**, v. 13, n. 2, p. 365–378, 2022.

RUDRABHATLA, Chaitanya K. **Security Design Patterns in Distributed Microservice Architecture**. [*S.l.*: *s.n.*], 2020. arXiv: 2008.03395 [cs.CR].

SILVA, Filipe Madeira da. **Trabalho de Conclusão de Curso: SOA - Arquitetura Orientada a Serviços**. [*S.l.*], 2006. Disponível em: https://linux.ime.usp.br/~cef/mac499-06/monografias/filipemadeira/monografia.pdf.

TÉCNICAS DE INTEGRAÇÃO ENTRE MICROSSERVIÇOS, Análise Comparativa de. **Análise Comparativa de Técnicas de Integração entre Microsserviços**. 2016. TCC de Graduação – UFSC, Florianópolis. Trabalho de Conclusão de Curso (Graduação).

VALENTE, Marco Tulio. Engenharia de software moderna. **Princípios e Práticas para Desenvolvimento de Software com Produtividade**, v. 1, p. 24, 2020.

VEGESNA, Vinod Varma. Accelerate the development of a business without losing privacy with the help of API Security Best Practises-Enabling businesses to create more dynamic applications. **International Journal of Management, Technology and Engineering**, v. 12, 2022.

VILLAÇA, LH; PIMENTA JR, Antônio Francisco; AZEVEDO, Leonardo Guerreiro. Construindo aplicações distribuídas com microsserviços. **Tópicos em Sistemas de Informação: Minicursos XV Simpósio Brasileiro de Sistemas de Informação. SBC**, 2018.

XIAO, Zhongxiang; WIJEGUNARATNE, Inji; QIANG, Xinjian. Reflections on SOA and Microservices. *In*: 2016 4th International Conference on Enterprise Systems (ES). [*S.l.*: *s.n.*], 2016. P. 60–67.

APÊNDICE A – Código Fonte

O código completo está disponível em repositório público e pode ser encontrado em: https://gitlab.com/ericsegalajovelli/reliability-patterns

Estudo de padrões de projeto para tolerância a falhas voltados a serviços

Eric Segala Jovelli[1]

[1]Universidade Federal de Santa Catarina (UFSC)

Abstract. *The objective of this work is to compare microservices architectural patterns to increase the reliability of services through the implementation of fault tolerance-oriented design patterns. For the evaluations, proofs of concept were implemented and these were subjected to load tests, with fault injection to observe the system's behavior. Based on this load, test scenarios were defined to evaluate the behavior and performance of the prototypes that implement each standard. Case studies were carried out for the retry, circuit breaker, sidecar and throttling patterns. For the analysis, latency, saturation and error metrics were considered, comparing service implementations with and without fault tolerance mechanisms.*

Resumo. *O objetivo deste trabalho é comparar padrões arquiteturais de microsserviços para aumentar a confiabilidade de serviços por meio de implementação de padrões voltados à tolerância à falhas. Para as avaliações, foram implementadas provas de conceito e estas foram submetidas a testes de carga, com a injeção de falhas para a observação do comportamento do sistema. A partir desta carga, foram definidos cenários de teste para avaliar o comportamento e desempenho do dos protótipos que implementam cada padrão. Foram realizados estudos de caso para os padrões retry, disjuntor, sidecar e throttling. Para a análise, foram consideradas métricas de latência, saturação e erros, comparando implementações do serviço com e sem mecanismos de tolerância a falhas.*

1. Introdução

Na década de 90 houve a popularização da Internet e, desde então, a maneira pela qual as aplicações funcionam vem sendo alterada. No início, uma era denominada *Web 1.0*, o conteúdo dos sites era estático e o uso era primordialmente técnico, recheado de sites corporativos, acadêmicos ou em construção [Gil 2014]. Além disso, usava-se bastante a rede para envio de *emails* e buscadores, como o Google. A nova fase foi chamada de *Web 2.0*, uma era conhecida como era social ou participativa. Nessa época os blogs e ferramentas de *chats* e mídias sociais ganharam notoriedade, como o Facebook e o Youtube. No momento atual uma área de estudo que tem tido destaque é a "Web 3.0" [Groombridge 2023], que é a definição dada à uma Internet impactada pelas novas tecnologias, entre elas a inteligência artificial, a Internet das coisas e a *blockchain*.

Desde a época da *Web 1.0*, o grande volume de usuários das aplicações que foi surgindo exigiu que se pensasse em arquiteturas para sistemas lidarem com um alto volume de requisições. O jeito tradicional de desenvolver aplicações é chamado monolítico , no qual se tem um sistema atômico, funcionando em um único processo,

em que se tem um *software* com diferente partes interligadas em um único programa [Villaça et al. 2018], funcionando em uma única máquina. Porém, à medida que a complexidade dos sistemas foi crescendo, novas formas de criar aplicações surgiram.

Uma alternativa à SOA é a arquitetura de microsserviços, formada por um conjunto de serviços. Nela, um serviço pode ser descrito como uma aplicação em seu próprio domínio, capaz de evoluir independentemente e ter sua própria arquitetura, tecnologia e plataforma [Xiao et al. 2016], além de poder ser gerenciado, implantado e escalado de forma independente, seguindo seu próprio ciclo de lançamento. Dessa forma, uma arquitetura de microsserviços consiste em uma aplicação que é composta por componentes independentes que, cada um deles é uma aplicação própria.

2. Problema

À medida que os negócios crescem e dependem cada vez mais da presença digital, passa a ser de suma importância que os sistemas sejam confiáveis. Casos em que sistemas ficam indisponíveis gerando graves problemas aos usuários não são raros. Um exemplo disso ocorreu dia 29/05/2024, quando em um mesmo dia os bancos Bradesco, Banco do Brasil e Inter ficaram indisponíveis [Digital 2024], impedindo a realização de transações PIX.

Dessa forma, surgiram diversas técnicas para garantir que aplicações com arquitetura de microsserviços continuem funcionando mesmo que um de seus serviços apresente problemas [de Técnicas de Integração entre Microsserviços 2016]. Como o uso de microsserviços tem crescido muito na última década, esse trabalho terá como foco padrões de projeto para tolerância a falhas destinados a serviços.

3. Objetivo

Comparar os padrões arquiteturais de microsserviços para confiabilidade por meio de implementação dos padrões de arquitetura voltados à tolerância a falhas, possibilitando comparar as vantagens e desvantagens práticas de cada padrão.

4. Fundamentação teórica

4.1. Padrões de projeto

Os padrões de projeto são soluções documentadas para problemas frequentemente encontrados no desenvolvimento de aplicações [Burns and Oppenheimer 2016]. São usados para auxiliar na arquitetura de software e são documentados de forma a serem aplicáveis para qualquer linguagem de programação. O uso de padrões de projeto permite aumentar a qualidade, desempenho e viabilidade de uso da programação distribuída.

4.2. Padrões arquiteturais

Desde a década de 70, com o aumento da complexidade das aplicações, passou-se a estudar não apenas a organização de entidades de baixo nível, como classes, mas também a organização em alto nível, como serviços formados por inúmeras classes[Valente 2020]. É possível extrair desse estudo em alto nível certos padrões, denominados padrões arquiteturais. Um padrão que tem recebido grande destaque ultimamente é a arquitetura de microsserviços. Ela consiste na fragmentação da aplicação em um conjunto de serviços. Um serviço pode ser descrito como uma pequena aplicação limitada no seu contexto que

pode evoluir independentemente e ter sua própria arquitetura, tecnologia e plataforma [Xiao et al. 2016]. Além disso, cada um pode ser gerenciada e escalado de forma independente, tendo seu próprio ciclo de implantação e metodologia de desenvolvimento. Cada serviço tem sua base de código própria. Isso permite que seja gerenciado por uma equipe pequena. Outro fato que contribui para isso é que esses componentes oferecem APIs externas, o que permite que as equipes responsáveis por outras áreas não precisem entender os detalhes da implementação, mas apenas integrar com a API.

5. valiação de padrões de projeto para tolerância a falhas voltados a serviços

Nesse capítulo haverá o detalhamento de padrões de projeto explicando o funcionamento de cada um escolhido.

5.1. Retry

O padrão *Retry* permite com que a tentativa de obter uma resposta de outro serviço ocorra mais de uma vez caso a requisição falhe. Ao obter falha na primeira tentativa, o serviço espera por um tempo e tenta novamente. Para demonstrar um caso de uso do *retry*, foi criado um servidor integrado a um serviço externo. A falha simulada é uma falha nesse serviço externo ou então na conexão entre o servidor principal e o externo.

Foi usado o Emulab [Hibler et al. 2008] como infraestrutura para todos os estudos de caso desse trabalho. O Emulab [Hibler et al. 2008] é uma plataforma de testes voltado a experimentos envolvendo sistemas distribuídos. Ele foi desenvolvido pela fundação Open Source e permite o uso de uma mesa-de-testes para trazer maior realismo aos experimentos. Foi escolhida a máquina t430 para todos os nós usados no experimento.

No Emulab [Hibler et al. 2008] o experimento foi criado como três nós. Essa estrutura pode ser explicada como uma aplicação na arquitetura cliente-servidor na qual o servidor possui uma integração com uma aplicação externa. Na estrutura cliente-servidor o cliente é um testador de carga que gera requisições ao servidor e registra mensurações. O servidor é uma API que responde ao cliente, consultando um nó banco de dados. Enquanto no nó servidor são implementados mecanismos de tolerância a falhas, é na máquina do serviço externo que são geradas as falhas. O código completo pode ser encontrado em: `https://gitlab.com/ericsegalajovelli/reliability-patterns`

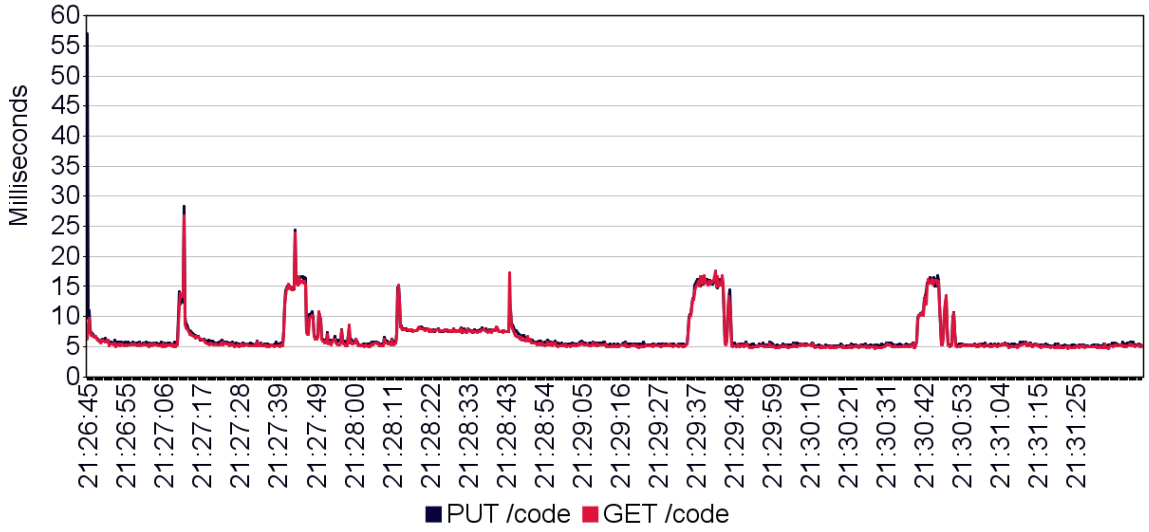
Cenários de avaliação

Inicialmente ocorre a etapa de preparação do ambiente. Nessa etapa todos os objetos do banco de dados foram removidos. Em seguida, foram inseridas 5.000 entradas, com códigos com tamanho de 20 caracteres. O teste foi executado pelo período de cinco minutos com um servidor sem qualquer mecanismo de tolerância a falhas. Os testes executados seguiram os valores mostrados na Tabela 1. O objetivo é que se tenha três falhas. A primeira com recuperação rápida (serviço externo é reiniciado), a segunda com tempo de recuperação médio (1 segundo) e a terceira longa (30 segundos). Vale notar que essa tabela de testes é aplicada não só no teste do padrão *retry*, mas também é usada para os padrões restantes.

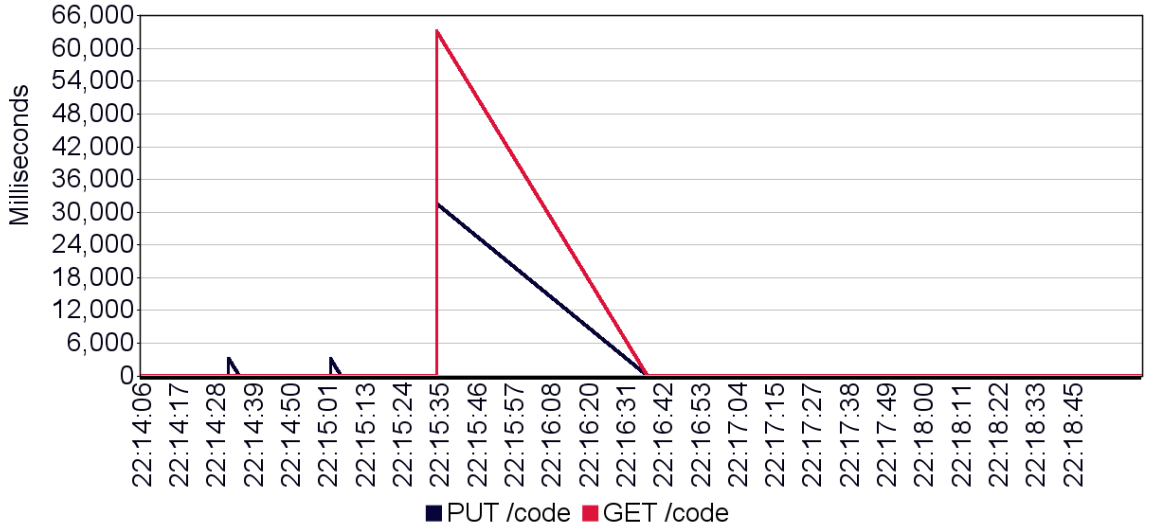
Tempo	Descrição
30 segundos após o início	Serviço externo é reiniciado
30 segundos após a primeira falha	Serviço externo é desligado
1 segundo após desligar o serviço externo	Serviço externo é ligado novamente
30 segundos após iniciar o serviço externo	Serviço externo é desligado novamente
30 segundos após desligar o serviço externo	Serviço externo é ligado novamente

Table 1. Tempo até a falha

Figure 1. Latência com e sem o uso do *Retry*



(a) Sem uso do *Retry*

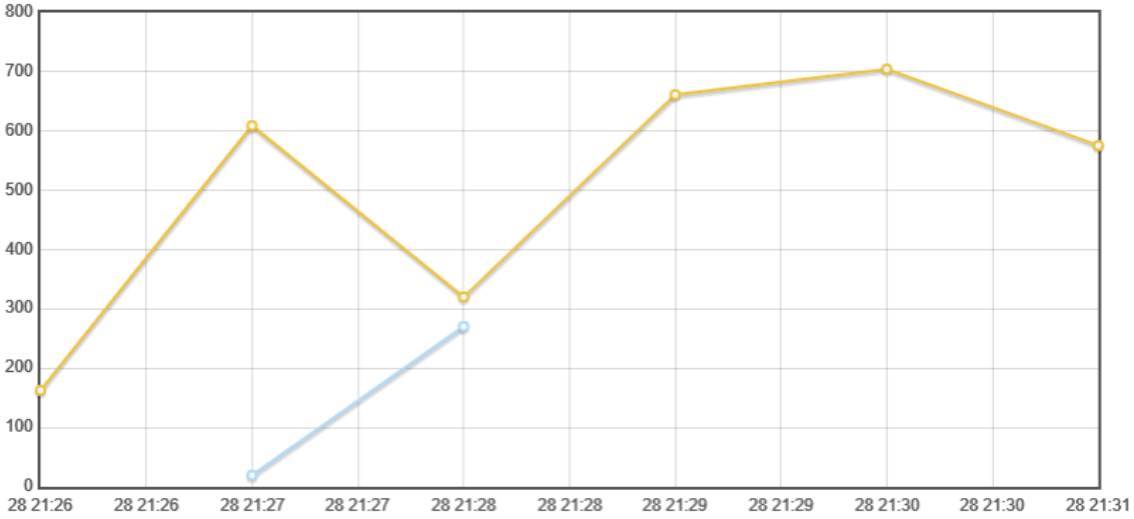


(b) com o uso do *Retry*

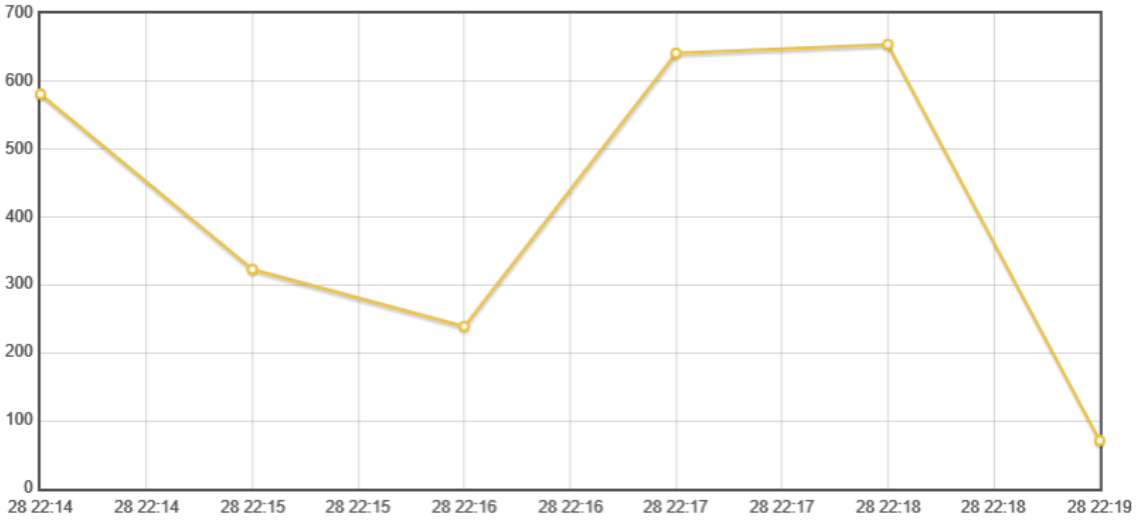
Figure 2. *

Fonte: o autor

Figure 3. Códigos de resposta com e sem o uso do *Retry*



(a) Sem uso do *Retry*. Linha azul: código 500. Linha amarela: código 200



(b) com o uso do *Retry*. Linha azul: código 500. Linha amarela: código 200

Figure 4. *
Fonte: o autor

Figure 5. Estatísticas do teste com e sem o uso do *Retry*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do *Retry*.

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**150692**	**0**	**0.00%**	**7.89**	**3**	**63231**	**5.00**	**13.00**	**14.00**	**19.00**	**502.70**
GET /code	75345	0	0.00%	7.61	3	63231	5.00	8.00	13.00	17.00	251.46
PUT /code	75347	0	0.00%	8.18	4	63223	5.00	8.00	13.00	17.00	251.35

(b) com o uso do *Retry*.

Figure 6. *
Fonte: o autor

Resultado

As Figuras 2, 4 e 6 demonstram a comparação dos resultados em testes com e sem uso do padrão *Retry*. No experimento sem mecanismos de tolerância, é possível notar que em alguns momentos do teste há leves aumentos na latência de poucas dezenas de milissegundos, que são causados por variações no tráfego de rede, processamento e outras causas que não envolvem uma espera artificial da requisição. É importante notar que nos momentos das duas falhas iniciais (30 e 60 segundos) o servidor principal retornou erros, que não aparecem no gráfico de vazão por serem quantidades de requisições muito pequenas, não influenciando o gráfico, onde os valores são originados por médias de períodos de 60 segundos. Por outro lado, no momento da falha longa (iniciada nos 90 segundos) começa a haver um grande número de repostas código 500, causadas pela falha de duração longa. Esse comportamento ocorre pois a biblioteca HTTP usada no serviço principal não possui mecanismo de *retry*, lançando uma exceção quando o serviço externo retorna um erro, que por sua vez é tratada e resulta no retorno de um código 500.

A Figura 2 compara a latência do cliente original à latência do serviço principal modificado com a inserção de um mecanismo de *retry*. É possível notar que com o uso desse padrão nos momentos das falhas há picos de latência maiores, que são causados pelo tempo que leva para ocorrer as novas tentativas. Por outro lado, a Figura 4, que compara a vazão, não apresenta código 500 quando o mecanismo é usado, o que demonstra que ele foi capaz de tratar as falhas no sistema externo. Na Figura 6, que compara estatísticas de teste, isso é confirmado quando se compara o percentual de erros, que neste último é de 0% enquanto sem o uso do padrão o valor é de 10.34%.

5.2. Disjuntor

O disjuntor, traduzido do inglês *Circuit Breaker* é focado em tratar falhas geradas por um tempo de espera infinita ou muito longa ao fazer uma requisição. A falta de resposta da requisição pode ser causada por falhas na comunicação ou problemas no serviço que recebeu a requisição. O uso do disjuntor visa melhorar a estabilidade da aplicação [Azure 2023].

Em um sistema distribuído, chamadas a recursos e serviços podem falhar devido a falhas transientes, como rede de Internet lenta, sobrecarga, tempo limite de espera atingido (erro de *timeout*). Essas falhas normalmente se corrigem normalmente após um período curto de tempo e aplicações robustas devem ser capazes de lidar com esse tipo de falha.

Porém, há casos em que as falhas ocorrem devido a problemas inesperados, gerando um tempo de espera maior que o previsto, gerando um tempo de correção muito maior. Nesses casos, não faz sentido uma aplicação continuar tentando uma operação que é improvável que seja bem sucedida. O mais apropriado é que a aplicação aceite que a operação falhar e lide com a situação de outra maneira [Azure 2023]. O padrão disjuntor propõe evitar que operações que provavelmente irão falhar continuem sendo executadas repetidamente.

O funcionamento de padrão disjuntor é análogo a uma máquina de estados. Existem três estados que mudam de acordo com a frequência que ocorrem falhas no serviço requisitado [Montesi and Weber 2018]. No estado fechado, as requisições do cliente ao

serviço acontecem normalmente, e são monitoradas se o serviço requisitado está retornando erros, um exemplo desse monitoramento pode ser o uso de *pinging*. Caso haja falhas, o estado é transferido para estado meio aberto. No estado aberto, toda vez que o cliente faz uma requisição é retornado um erro. Após um determinado tempo o estado é transicionado para o meio aberto. No estado meio aberto, um estado de transição, é permitido que seja enviado um número limitado de requisições. Caso não haja falhas por um período de tempo, ocorre a transição para o estado fechado, caso contrário para o aberto.

Estudo de caso

Foi realizado um experimento usando uma arquitetura e ambiente de testes idênticos ao do experimento anterior. A única diferença é que nesse caso foi adicionado um mecanismo de disjuntor em vez do *retry*. Para isso foi usada a biblioteca Opossum[Manager 2024].

Ao analisar a Figura 8, a comparação das métricas de latência, é possível notar que há picos de latência quando ocorrem as falhas nos momentos 30, 60 e 90 segundos após o início. Porém, esses picos não são significativos, sendo o mais lento menor que um quarto de segundo. Ou seja, independente do estado do disjuntor o cliente irá ter uma resposta rápida. A grande diferença ocorre na vazão.

Na Figura 10, a comparação das métricas de vazão, a linha de respostas falhas (código 503) começa com um valor baixo. Isso ocorre pois houve duas falhas curtas no primeiro minuto. Assim, apenas durante esses dois momentos o disjuntor passou para o estado fechado. Porém, como o período de *resetTimeout* foi definido como um valor de 3 segundos, rapidamente o disjuntor volta ao estado meio aberto e, em seguida fechado.

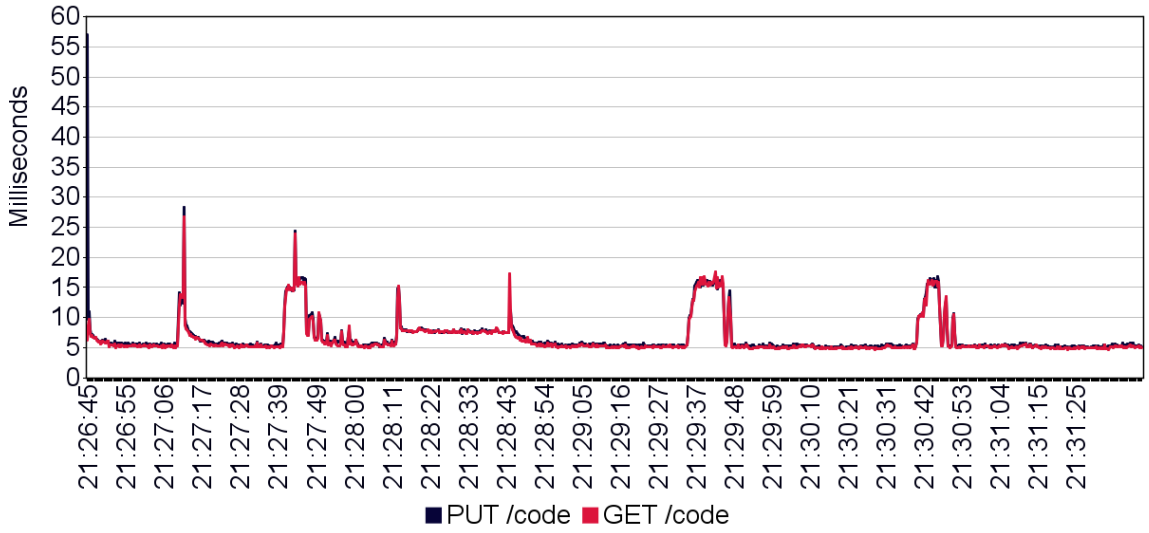
Já na falha longa, que ocorre 90 segundos após o início, há um aumento substancial na quantidade de códigos de falha retornados pelo servidor, o que é representado pela parte íngreme da linha azul na figura 10. Isso se traduz pelo disjuntor em estado aberto ou meio aberto rejeitando requisições. Essa situação só muda quando o serviço externo volta a funcionar e o disjuntor detecta no estado meio aberto que uma quantidade limitada de requisições foi enviada e obteve sucesso. Isso permite transicionar para o estado fechado, o que significa a volta ao estado normal.

Por fim, as estatísticas demonstradas da gráfico da Figura 12 revelam que esse padrão apresentou uma taxa de erro de 27.76%. Porém, apesar desse valor inicialmente parecer algo negativo, não é, pois o objetivo do padrão não é minimizar a quantidade de respostas fracassadas, mas sim aliviar a quantidade de requisições ao serviço externo quando este apresenta estado de falha. Para fazer isso, o disjuntor passa alguns segundos sem enviar requisições, o que justifica essa taxa de erro maior.

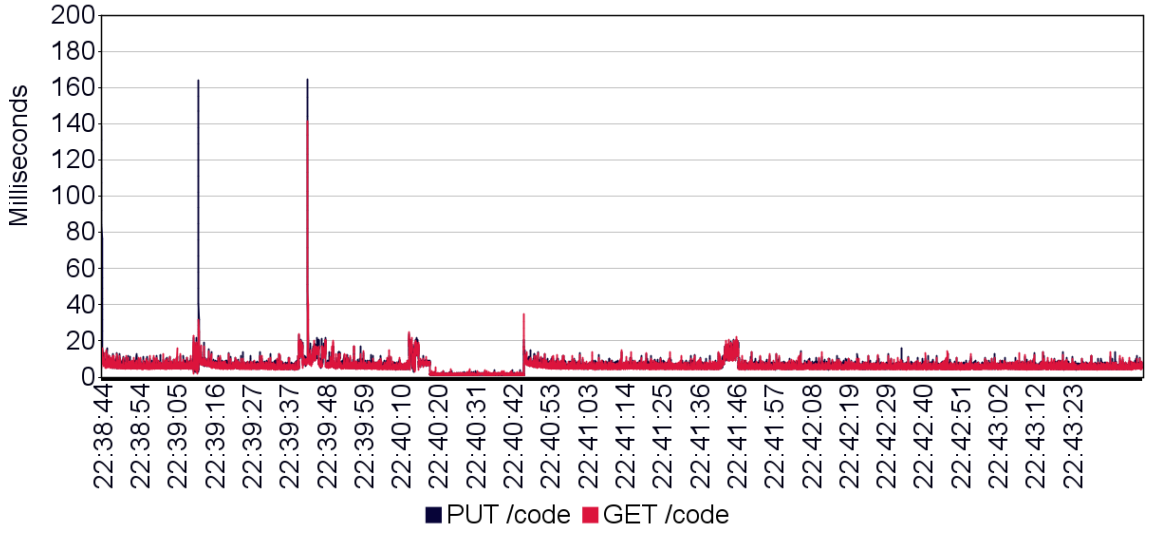
5.3. *Throttling*

Há sistemas em que a atividade dos usuários pode variar com o tempo. Isso pode ocorrer por exemplo pelo uso do sistema em horário comercial ou então em atividades específicas, como por exemplo um sistema de pagamento de funcionários no início do mês. Nesses casos, é preciso lidar com esse pico de atividade de forma que o excesso de atividade não sobrecarregue o sistema de forma a causar panes ou lentidão [Vegesna 2022]. Caso isso aconteça as consequências podem ser graves, como o descumprimento de um acordo de nível de serviço, ou a interrupção de outros serviços por quais o servidor sobrecarregado também é responsável. Para evitar tais problemas existe o *Throttling*.

Figure 7. Latência dos testes com e sem o uso do disjuntor



(a) Sem uso do disjuntor.

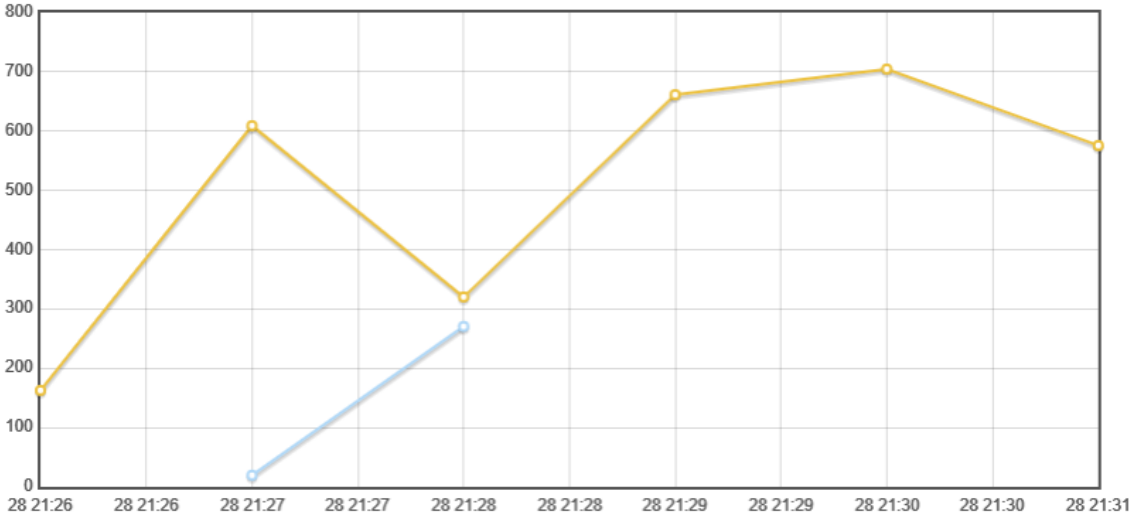


(b) com o uso do disjuntor.

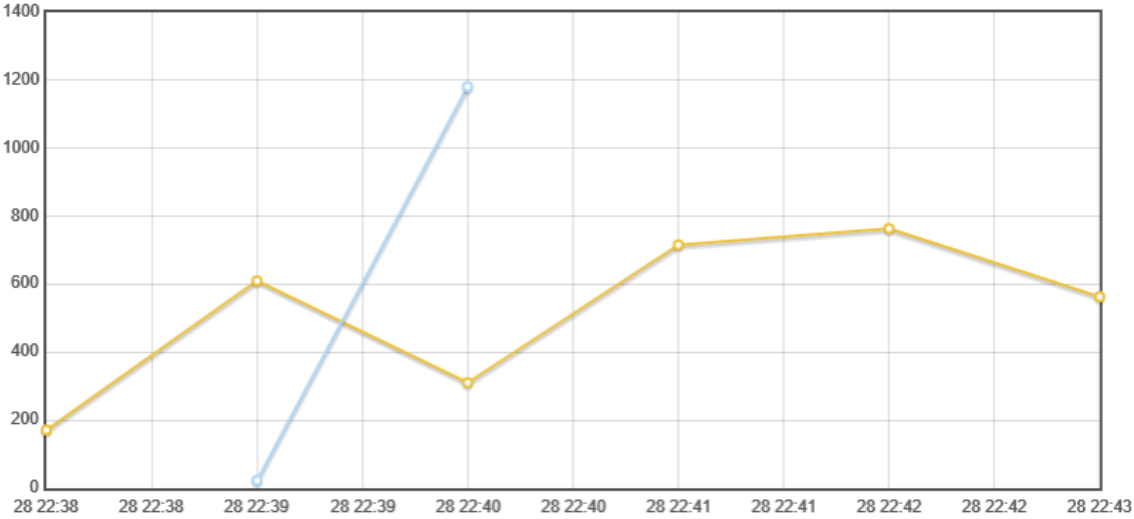
Figure 8. *

Fonte: o autor

Figure 9. Vazão dos testes com e sem o uso do disjuntor



(a) Sem uso do disjuntor. Linha azul: código 500. Linha amarela: código 200



(b) com o uso do disjuntor. Linha azul: código 503. Linha amarela: código 200

Figure 10. *
Fonte: o autor

Figure 11. Estatísticas dos testes com e sem o uso do disjuntor

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do disjuntor.

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**260316**	**72265**	**27.76%**	**4.54**	**0**	**165**	**5.00**	**6.00**	**7.00**	**9.00**	**868.40**
GET /code	130157	36132	27.76%	4.46	0	159	5.00	6.00	7.00	9.00	434.40
PUT /code	130159	36133	27.76%	4.63	0	165	5.00	6.00	8.00	9.00	434.20

(b) com o uso do disjuntor.

Figure 12. *
Fonte: o autor

Figure 13. Estatísticas do servidor com processos lentos com e sem o *throttling*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**875**	**0**	**0.00%**	**24195.11**	**339**	**120809**	**24158.00**	**24177.00**	**26752.60**	**75310.84**	**2.70**
GET /code	15	0	0.00%	81969.73	45696	120809	84215.00	106319.60	120809.00	120809.00	0.05
PUT /code	860	0	0.00%	23187.41	339	47562	24157.00	24176.00	24179.00	34074.08	2.66

(a) Sem uso do *throttling*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**680580**	**671762**	**98.70%**	**29.85**	**2**	**3923**	**26.00**	**30.00**	**31.00**	**129.00**	**2270.80**
GET /code	8768	0	0.00%	136.55	12	3923	132.00	143.00	157.00	164.00	29.25
PUT /code	671812	671762	99.99%	28.46	2	3833	26.00	29.00	30.00	33.00	2241.59

(b) com o uso do *throttling*

Figure 14. *
Fonte: o autor

O *Throttling* é um padrão que consiste na limitação do número de chamadas para uma API. Essa limitação pode englobar um conjunto de *endpoints*, um *endpoint* específico ou até mesmo uma combinação de *endpoint* com um verbo HTTP específico. Essa restrição pode ser voltada a um endereço IP específico [Lamba 2019], realizando uma limitação voltada a um determinado usuário, ou então pode ser geral a qualquer usuário.

Estudo de caso

Para estudar esse padrão foi criada uma cópia da aplicação principal com uma adaptação: foi inserido um mecanismo de *throttling* no *endpoint /GET*. Isso foi feito com uso da biblioteca express-rate-limit [exp].

A Figura 14, que compara as métricas dos testes com e sem o uso desse padrão, demonstra que após o uso houve um aumento significativo das requisições por segundo do *endpoint* não sobrecarregado, o */GET*, já que o valor dessa métrica passou de 0.05 para 29.25. Isso permite concluir que o padrão fez com que o uso excessivo de uma parte do sistema não sobrecarregasse as outras. Por outro lado, é possível notar uma taxa de erro maior, pois na coluna *Error* há passa a haver uma taxa de 98.70% com o uso do padrão, enquanto sem o padrão essa taxa é 0%. Esses erros são esperados, e são as respostas de código 429 retornadas que indicam que a requisição foi rejeitada por ultrapassar o limite de requisições por período.

5.4. Sidecar

Frequentemente, um serviço principal precisa de serviços auxiliares, e é comum que esses serviços auxiliares sejam agregados na mesma base de código do serviço principal. Mas em alguns casos, é conveniente que os serviços auxiliares sejam independentes da linguagem de programação ou do ambiente de execução do serviço principal, já que isso traz diversas vantagens, como a possibilidade de reuso desse serviço auxiliar [Burns and Oppenheimer 2016]. Uma possibilidade para implantação desses serviços auxiliares seria a criação de um novo serviço separado, a outra alternativa é implantar no mesmo *host*, em um contêiner separado, o que constitui o *sidecar*.

Estudo de caso

Foi desenvolvido um contêiner auxiliar que exerce a função de cache. O objetivo é que esse sidecar reduza a quantidade de erros caso o serviço externo esteja indisponível. O servidor principal é o mesmo usado como *benchmark* para os outros testes. O cache é feito em memória volátil com a biblioteca Redis [npm, Inc. 2024], uma implementação Node.js do servidor noSQL Redis. No experimento, o *Sidecar* é usado apenas para operações de consulta. Dessa forma, o cliente faz chamadas */get* para o *sidecar*, enquanto as requisições */put* são feitas diretamente para o serviço externo.

A Figura 16 demonstra que o uso desse padrão reduziu a zero o percentual de erros no *endpoint /get*, já que nos momentos de falha o servidor *cache* foi capaz de retornar os valores de seu banco de dados em vez de buscar o resultado no servidor em estado de falha. Além disso, a métrica de erros totais foi reduzida de 8.73% para 4.28%.

6. Trabalhos Relacionados

6.1. Desafios e soluções de segurança trazidos com o uso da MSA

No que tange aos problemas relacionados a cibersegurança em arquitetura de microsserviços [Rudrabhatla 2020] explica desafios de cibersegurança que não eram presentes na arquitetura monolítica. Dada sua natureza distribuída da MSA, passa a haver novos problemas, como aumento da superfície de ataque, maior complexidade no controle de autorização e segurança dos dados.

O objetivo do artigo é explicar os novos desafios de cibersegurança trazidos pela MSA e também propõe padrões e boas práticas para solucionar tais problemas. Para fazer isso, o trabalho explica seis problemas envolvendo cibersegurança trazidos com a MSA: segurança na superfície de ataque, segurança em profundidade, princípio do acesso mínimo, criptografia onipresente e devSecOps. Por fim, o trabalho expõe tendências de soluções que estão sendo pesquisadas envolvendo segurança de MSA.

Figure 15. Comparação de resultados estatísticos com e sem o uso do *Sidecar*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**199463**	**17409**	**8.73%**	**5.95**	**2**	**166**	**5.00**	**6.00**	**7.00**	**9.00**	**665.40**
GET /code	99730	8703	8.73%	5.84	2	166	5.00	6.00	7.00	9.00	332.77
PUT /code	99733	8706	8.73%	6.06	2	165	5.00	6.00	8.00	9.00	332.71

(a) Sem uso do *Sidecar*

Requests	Executions			Response Times (ms)							Throughput
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s
Total	**203794**	**8720**	**4.28%**	**5.82**	**2**	**182**	**5.00**	**8.00**	**8.00**	**11.00**	**679.86**
GET /code	101896	0	0.00%	4.56	2	19	4.00	6.00	6.00	9.00	340.00
PUT /code	101898	8720	8.56%	7.08	2	182	7.00	8.00	8.00	11.00	339.94

(b) com o uso do *Sidecar*

Figure 16. *
Fonte: o autor

6.2. Aumento da disponibilidade em sistemas baseados em MSA com *deploy* frequente

O trabalho [Rasheedh and Saradha 2022] aborda o problema da reinicialização frequente de microsserviços causada pelas metodologias ágeis e propõe uma solução. Com a difusão de metodologias ágeis para gerenciamento de projetos, as implantações de novas funcionalidades ou correções de software passaram a ocorrer com maior frequência. Essas implantações exigem que os serviços sejam reinicializados, o que é um problema para aplicações que prometem *downtime* zero.

O artigo propõe um *framework* para implantações em MSA que não necessita a reinicialização dos microsserviços. A proposta envolve o uso das práticas de desenvolvimento de software *SOLID* e o uso de um motor de mistura que funde os novos objetos com o código existente.

6.3. Melhoria no padrão disjuntor com o uso de cadeias de Markov

A tese de mestrado [Addeen 2019] aborda o problema da comunicação entre os microsserviços, advindo da intercomunicação via rede. Uma situação que pode ocorrer é um serviço ficar esperando infinitamente por uma resposta, e uma das soluções é o padrão disjuntor, que usa um tempo limite associado a máquinas de estado para lidar com falhas. Porém, o tempo de espera precisa ser dinâmico e é um desafio a escolha deste tempo limite.

Esse trabalho correlato propõe um padrão que une o disjuntor à cadeias de Markov, o que busca resolver o desafio da definição do tempo limite. A cadeia de Markov é uma teoria matemática de mudança de um estado para outro. O espaço de estado, que já é previamente conhecido, é usado para usado para gerar as cadeias. Para prever o próximo estado o cálculo é feito com base apenas no estado atual e com regras probabilísticas, sem considerar os estados anteriores. O resultado é que o disjuntor proposto é 43% mais rápido que o disjuntor tradicional.

7. Conclusão

Esse trabalho contextualizou os padrões de projeto, padrões arquiteturais e a arquitetura de microsserviços. A obra explicou também o funcionamento da arquitetura de microsserviços e suas vantagens. Além disso, apresentou a área de tolerância a falhas em sistemas orientados a serviços e sua importância.

Foi explicado o funcionamento do padrão de projeto *retry* e o seu algoritmo de cálculo de tempo entre as tentativas, o *exponential backoff*. O trabalho também sugeriu em que tipo de situação ele pode ser empregado. Ademais, demonstrou seu funcionamento na prática com um estudo de caso, que concluiu que o padrão pode tratar falhas transientes, com o ônus de uma latência maior.

Foram apresentados o padrão disjuntor, sua máquina de estado, os algoritmos de cada estado e o contexto em que é pertinente haver uso desse padrão. Um estudo de caso prático demonstrou seu uso, provando que ele é capaz de evitar a sobrecarga de um sistema em estado de falha, ao custo do cliente ter uma taxa de erro maior devido ao tempo necessário para transição de estados.

O padrão *throttling* também foi abordado. Foi explicado o funcionamento desse padrão e como ele pode melhorar a estabilidade de sistemas, com ênfase em sua aplicação em situações de escalamento automático. Um estudo de caso demonstrou que o padrão permite evitar que o uso intensivo de uma parte do sistema interfira no resto, sob o ônus de limitar usuários que queiram fazer uso extremo de um serviço específico.

O funcionamento do *Sidecar* foi explicado, além das situações em que é vantajoso seu uso e das motivações que justificam sua implantação. Foi feito um estudo de caso no qual esse padrão foi usado para encapsular um servidor de *caching*, o que permitiu reduzir a taxa de erro em 49% ao retornar respostas em *cache* em vez de buscá-las em um serviço falho.

8. Trabalhos Futuros

Ampliar os estudos práticos Adicionar estudos de caso com experimentos para os padrões Eleição de líder, *Queue-Based Load Leveling*, *Bulkhead* e *Rate Limit*.

Combinar padrões Criar estudos de casos mais complexos combinando padrões, por exemplo, combinar o disjuntor com o *retry*, o que pode evitar a sobrecarga de um serviço falho e, ao mesmo tempo, lidar com falhas transientes.

9. References

References

express-rate-limit - npm. `https://www.npmjs.com/package/express-rate-limit`. Accessed: May 28, 2024.

Addeen, H. H. (2019). *A Dynamic Fault Tolerance Model for Microservices Architecture*. South Dakota State University.

Azure (2023). *Circuit Breaker pattern – Azure Architecture Center*. Microsoft.

Burns, B. and Oppenheimer, D. (2016). Design patterns for container-based distributed systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

de Técnicas de Integração entre Microsserviços, A. C. (2016). Análise comparativa de técnicas de integração entre microsserviços. Master's thesis, UFSC, Florianópolis. Trabalho de Conclusão de Curso (Graduação).

Digital, O. (2024). Bradesco, banco do brasil e inter apresentam instabilidade nesta quarta (29).

Gil, H. (2014). A passagem da web 1.0 para a web 2.0 e... web 3.0: potenciais consequências para uma humanização em contexto educativo. *Educatic: boletim informativo*, pages 1–2.

Groombridge, D. (2023). *Gartner Top 10 Strategic Technology Trends for 2023*. Gartner.

Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., and Lepreau, J. (2008). Large-scale virtualization in the emulab network testbed. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

Lamba, A. (2019). Api design principles & security best practices–accelerate your business without compromising security.

Manager, N. P. (2024). Opossum: A node.js circuit breaker. `https://www.npmjs.com/package/opossum`. Accessed: 2024-05-22.

Montesi, F. and Weber, J. (2018). From the decorator pattern to circuit breakers in microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1733–1735.

npm, Inc. (2024). redis. Accessed: 2024-05-28.

Rasheedh, J. and Saradha, S. (2022). Design and development of resilient microservices architecture for cloud based applications using hybrid design patterns. *Indian J. Comput. Sci. Eng*, 13(2):365–378.

Rudrabhatla, C. K. (2020). Security design patterns in distributed microservice architecture.

Valente, M. T. (2020). Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, 1:24.

Vegesna, V. V. (2022). Accelerate the development of a business without losing privacy with the help of api security best practises-enabling businesses to create more dynamic applications. *International Journal of Management, Technology and Engineering*, 12.

Villaça, L., Pimenta Jr, A. F., and Azevedo, L. G. (2018). Construindo aplicações distribuídas com microsserviços. *Tópicos em Sistemas de Informação: Minicursos XV Simpósio Brasileiro de Sistemas de Informação. SBC*.

Xiao, Z., Wijegunaratne, I., and Qiang, X. (2016). Reflections on soa and microservices. In *2016 4th International Conference on Enterprise Systems (ES)*, pages 60–67.