UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I - CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM COMPUTAÇÃO

THAIRAM MICHEL SANTOS ATAÍDE

UMA ABORDAGEM DE TESTES DE RESILIÊNCIA PARA SISTEMAS BASEADOS EM MICROSSERVIÇOS

CAMPINA GRANDE - PB
2022

THAIRAM MICHEL SANTOS ATAÍDE

UMA ABORDAGEM DE TESTES DE RESILIÊNCIA PARA SISTEMAS BASEADOS EM MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Computação do Departamento de Computação do Centro de Ciências e Tecnologia da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Computação.

Área de concentração: Testes de Software

Orientador: Dra. Sabrina de Figueirêdo Souto

CAMPINA GRANDE - PB
2022

A862u Ataide, Thairam Michel Santos.

Uma abordagem de testes de resiliência para sistemas baseados em microsserviços [manuscrito] / Thairam Michel Santos Ataide. - 2022.

44 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2022.

"Orientação : Profa. Dra. Sabrina de Figueirêdo Souto , Coordenação do Curso de Computação - CCT."

1. Microsserviços. 2. Resiliência. 3. Internet das Coisas. 4. Desenvolvimento de sistemas. I. Título

21. ed. CDD 006.3

Elaborada por Deivid A. M. Neves - CRB - 15/899 BC/UEPB

THAIRAM MICHEL SANTOS ATAÍDE

UMA ABORDAGEM DE TESTES DE RESILIÊNCIA PARA SISTEMAS BASEADOS EM MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Computação do Departamento de Computação do Centro de Ciências e Tecnologia da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Computação.

Área de concentração: Testes de Software

Trabalho aprovado em 12 de Maio de 2022.

BANCA EXAMINADORA

Sabrina de F. Souto.

Profa. Dra. Sabrina Figueiredo Souto (DC - UEPB)
Orientador(a)

Lucas Barbosa Oliveira

MSc. Lucas Barbora (NUTES - UEPB)
Examinador(a)

Kézia de Vasconcelos Oliveira Santos

Profa. Dra. Kézia Vasconcelos (DC - UEPB)
Examinador(a)

A minha família e amigos, pela dedicação, companheirismo e amizade, DEDICO.

RESUMO

A Internet das Coisas (IoT) vem se tornando uma tendência tecnológica, pesquisas mostram o enorme potencial e crescimento de dispositivos conectados, assim como, um grande aumento no mercado mundial de IoT. Nesse contexto, a arquitetura de microsserviços surge como um diferencial importante para o sucesso do desenvolvimento de sistemas distribuídos. A arquitetura de microsserviços traz diversos benefícios como: flexibilidade, menor granularidade das funcionalidades, independência dos serviços e escalabilidade. A baixa granularidade e a independência dos serviços permitem testes melhores direcionados o que facilita a validação funcional. No entanto, a validação não funcional é um grande desafio em sistemas distribuídos, lidar com problemas inesperados é extremamente difícil e uma arquitetura de microsserviços precisar ser resiliente a falhas. O objetivo principal deste trabalho é propor uma abordagem de testes que valide os desafios não funcionais relacionados à resiliência. O trabalho foi aplicado a um sistema IoT para saúde de código aberto, arquitetado em microsserviços denominado OCARIoT. Através de experimentos mostramos como a arquitetura de microsserviços é afetada pela resiliência.

Palavras-chaves: Microsserviços. Resiliência. Internet das Coisas. Desenvolvimento de sistemas.

ABSTRACT

The Internet of Things (IoT) has become a technological trend, research shows the enormous potential and growth of connected devices, as well as a large increase in the global IoT market. In this context, the microservices architecture emerges as an important differential for the success of the development of distributed systems. The microservices architecture brings several benefits such as: flexibility, less granularity of functionality, service independence and scalability. The finer granularity and independence of the services allow for better targeted testing which facilitates functional validation. However, non-functional validation is a major challenge in distributed systems, dealing with unexpected problems is extremely difficult, and a microservices architecture needs to be resilient to failure. The main objective of this work is to propose a testing approach that validates non-functional challenges related to resilience. The work was applied to an open source IoT system for healthcare, architected in microservices called OCARIoT. Through experiments we show how the microservices architecture is affected by resiliency.

Keywords: Microservices. Resilience. Internet of Things. Systems development.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IoT	Internet of things
MSA	Microservice-based architectures
SUT	System Under Test
GDPR	General Data Protection Regulation
API	Application Programming Interface

SUMÁRIO

1 INTRODUÇÃO

Microsserviços (A..., 2021) (JAMSHIDI et al., 2018) (NEWMAN, 2015) são uma tendência popular na arquitetura de software nos últimos anos para a construção de sistemas distribuídos em larga escala. Os microsserviços empregam princípios de design arquitetônico que levam a contextos limitados baseados em domínio explícito e acoplamento fraco (RADEMACHER; SORGALLA; SACHWEH, 2018), exploram tecnologias modernas baseadas em nuvem, incluindo conteinerização e autocura (ESPOSITO; CASTIGLIONE; CHOO, 2016), e são adequados para paradigmas modernos de engenharia de software, como DevOps (BASS; WEBER; ZHU, 2015), incluindo métodos de desenvolvimento ágil e entrega contínua.

Arquitetura baseadas em microsserviços (MSAs) consistem em pequenos serviços que se concentram em uma funcionalidade específica. Algumas vantagens dos microsserviços são: maior escalabilidade, flexibilidade e menor granularidade da funcionalidade realizada por um serviço. No contexto de teste software, os microsserviços trazem benefícios, pois o foco em uma funcionalidade e a menor granularidade possibilitam testes mais direcionados. Devido aos benefícios mencionados e à alta flexibilidade e evolutividade que caracterizam as arquiteturas de microsserviços, muitas organizações, como Netflix, Amazon, eBay, Twitter, já evoluíram suas aplicações de negócios para MSA (FRANCESCO; MALAVOLTA; LAGO, 2017).

Levando em consideração a criticidade dos sistemas de microsserviços, construí-los se torna uma tarefa desafiadora. Arquiteturas de referência são muito úteis no contexto dos sistemas de microsserviços pois auxiliam na abordagem dos atributos de qualidade (requisitos não funcionais), que são muito relevantes para esse tipo de sistema. Porém, verificar como os MSAs lidam com atributos de qualidade é também um grande desafio.

Um dos grandes desafios apresentados pelos microsserviços é o grande número de serviços que podem compor um aplicativo, além disso, os microsserviços são interdependentes e devem permanecer funcionando mesmo quando outros serviços dos quais eles dependem não estão disponíveis ou não estão respondendo adequadamente. Muitos serviços populares altamente disponíveis sofreram diversas falhas e interrupções (por exemplo, falhas em cascata devido à sobrecarga de dados) (HEORHIADI et al., 2016) e, a maioria dessas interrupções foi causada por falta ou lógica defeituosa de recuperação. Os testes unitários e integrados não são suficientes para detectar bugs na lógica de tratamento de falhas e interrupções.

Empresas como Amazon, Google e Facebook aplicam técnicas de injeção de faltas para verificar a resiliência de seus sistemas (BASIRI et al., 2017). Nesse contexto, o Chaos Monkey (CHAOS..., 2021) é uma ferramenta de código aberto mantida pela Netflix para injeção de faltas aleatórias, sendo utilizada em larga escala. Ela é capaz de simular cenários de falha de instâncias de serviços em execução eliminando uma zona de disponibilidade ou uma região inteira do sistema. Porém, a ferramenta não analisa automaticamente o comportamento do aplicativo, que é algo necessário para encontrar bugs de implementação de maneira ágil. Além disso, a técnica de injeção de faltas não é muito eficiente e muito tempo e recursos são gastos pela exploração de

cenários redundantes.

Existe uma técnica para teste de sistemas altamente configuráveis (KIM et al., 2013) (SOUTO; D'AMORIM; GHEYI, 2017) que pode ser usada para causar interrupções nos serviços em execução e gerar dados que permitem analisar as falhas ocorridas nesse contexto caótico. Esta abordagem exercita o sistema, através da execução repetida dos testes existentes com todas as combinações dos serviços do sistema, simulando assim, cenários onde serviços estão inativos (fora do ar).

No entanto, esta abordagem não é capacitada para observar os recursos do servidor da aplicação durante a execução do sistema, tais consumo de CPU e memória. Além disso, ela também não consegue observar dados como o tempo de resposta dos serviços e a quantidade de requisições atendidas durante a execução dos casos de teste. Esses dados são de suma importância para uma abordagem de teste de resiliência, pois além de identificar as falhas e os tipos das falhas ocorridas, eles contribuem para detectar o nível de degradação dos serviços, de forma que possibilite a tomada de decisão por parte do gestor do sistema.

Dada a necessidade de observar os dados de consumo de CPU e memória do host onde o sistema de microsserviço é implantado, os tempos de execuções dos testes e a quantidade de requisições feitas durantes os testes, a proposta desse trabalho é expandir a técnica existente, de forma que ela seja capaz de provocar intencionalmente interrupções nos serviços, e gerar dados que viabilizem a análise das possíveis falhas que irão ocorrer e os dados mencionados anteriormente. Viabilizando, desta forma, a identificação de possíveis relações entre as interrupções e as falhas, o consumo de recursos do host, para detectar o nível de degradação dos serviços, o tempo de execução dos testes e a quantidade de requisições.

1.1 Objetivos

O objetivo geral deste trabalho é analisar a qualidade, no que diz respeito a resiliência, de sistemas baseados em microsserviços através do teste de microsserviços com ênfase na resiliência.

Com base no objetivo geral, os objetivos específicos são:

- Avaliar a capacidade que um sistema possui de se manter operacional durante eventos extremos;
- Propor uma abordagem para testar microsserviços com ênfase na resiliência;
- Aplicar a abordagem proposta a um estudo de caso com o OCARIoT.

1.2 Estrutura do documento

O trabalho segue a estrutura descrita pelos capítulos citados a seguir:

- Capítulo 2: fornece embasamento teórico para compreensão do problema contextualizado neste trabalho;
- Capítulo 3: apresenta a abordagem para validar a resiliência de sistemas baseados em microsserviços e como ela se aplica;
- Capítulo 4: descreve as questões de pesquisa e a aplicação da abordagem ao objeto de estudo OCARIoT;
- Capítulo 5: apresenta as considerações finais assumidas após a problematização e a aplicação da abordagem proposta como solução, no objeto de estudo OCARIoT, e suas contribuições para o meio de teste de software.

2 REFERENCIAL TEÓRICO

Nesta seção, apresentamos uma base teórica para a compreensão de conceitos que são fundamentais para o entendimento do contexto a ser abordado na pesquisa realizada. Inicialmente serão apresentados os conceitos relacionados a Teste de Software; Microsserviços e por fim Resiliência.

2.1 Teste de Software

O software deve ser previsível e consistente, não apresentando surpresas aos usuários (MYERS; SANDLER; BADGETT, 2011). Para verificar que o software funciona de acordo com o esperado pelos usuários, é necessário testá-lo. O teste de software é um processo, ou uma série de processos, projetado para garantir que o código de computador faça o que foi projetado para fazer e, inversamente, que não faça nada não intencional.

Para melhor identificar e organizar os erros, os testes de software são separados em tipos diferentes, os mais populares são os testes de caixa branca e caixa preta. No teste de caixa branca o profissional responsável pelos testes tem acesso ao código fonte do sistema e pode observar mais atentamente determinadas etapas do código, como fluxo de dados e entre outros. No teste de caixa preta o profissional não tem acesso ao código fonte, nem a qualquer outro detalhe específico de implementação, ele irá se basear nos requisitos e irá validar fluxos comuns aos usuários finais.

2.1.1 Teste de API

Uma API (Application Programming Interface) é uma interface de programação que possibilita a comunicação e a troca de dados entre dois sistemas de software distintos. Uma API contém a lógica de negócios de um aplicativo, e as regras de como os usuários podem interagir com serviços, dados ou funções do aplicativo.

(ISHA; SHARMA; REVATHI, 2018) observam que o número de APIs está se expandindo exponencialmente a cada ano. Portanto, realizar testes de APIs tornou-se uma etapa fundamental no processo de desenvolvimento de software, pois APIs com defeito ou ineficazes podem resultar em menor aquisição de produtos e, finalmente, perda de receita.

O teste de API é crucial e seu objetivo é verificar a funcionalidade, confiabilidade, desempenho e segurança das interfaces de programação. No teste de API, utiliza-se software para enviar chamadas para a API, obter a saída e validar a resposta do sistema. Os testes de API concentram-se principalmente na camada lógica de negócios da arquitetura de software.

2.1.2 Teste de Sistema

O teste de sistema é uma fase do processo de teste de software, cujo objetivo é validar o comportamento de todo o sistema. Envolve requisitos funcionais e não funcionais e se enquadra

no tipo caixa preta 2.1. Nesse contexto, o ambiente de teste deve ser o mais semelhante possível ao ambiente de produção, a fim de maximizar a identificação de falhas específicas de ambiente e para que os testes sejam executados em condições similares aquelas que o usuário final irá utilizar.

2.2 Microsserviços

Microsserviços (A..., 2021) (JAMSHIDI et al., 2018) (NEWMAN, 2015) são uma tendência popular na arquitetura de software nos últimos anos para a construção de sistemas distribuídos em larga escala. Os microsserviços empregam princípios de design arquitetônico que levam a contextos limitados baseados em domínio explícito e acoplamento fraco (RADEMACHER; SORGALLA; SACHWEH, 2018), exploram tecnologias modernas baseadas em nuvem, incluindo conteinerização e autocura (ESPOSITO; CASTIGLIONE; CHOO, 2016), e são adequados para paradigmas modernos de engenharia de software, como DevOps (BASS; WEBER; ZHU, 2015), incluindo métodos de desenvolvimento ágil e entrega contínua.

2.2.1 Arquitetura de microsserviços

Arquitetura baseadas em microsserviços (MSAs) consistem em pequenos serviços que se concentram em uma funcionalidade específica. Algumas vantagens dos microsserviços são: maior escalabilidade, flexibilidade e menor granularidade da funcionalidade realizada por um serviço. Nesse contexto, os times podem usar stacks e linguagens de programação diferentes para diferentes serviços. Além disso, é possível ajustar a escala dos serviços de forma independente, reduzindo o desperdício e o custo associado ao ajuste de escala de aplicativos inteiros.

Devido aos benefícios mencionados e à alta flexibilidade e evolutividade que caracterizam as arquiteturas de microsserviços, muitas organizações, como Netflix, Amazon, eBay, Twitter, já evoluíram suas aplicações de negócios para MSA (FRANCESCO; MALAVOLTA; LAGO, 2017). Algumas pesquisas recentes (GARCIA-MORENO et al., 2020) (IANCULESCU et al., 2019) sugerem implementar aplicativos de saúde utilizando arquiteturas baseadas em microsserviços.

2.2.2 Teste de microsserviços

O crescente uso de microsserviços em muitos aplicativos de software resultou na necessidade de ferramentas e técnicas de teste mais adequadas para testar esses aplicativos. Para resolver esse problema, novas técnicas e ferramentas de teste estão sendo desenvolvidas com foco na complexidade adicional do aumento da comunicação de dados necessária entre vários serviços (HERNáNDEZ et al., 2021).

Alguns dos grandes desafios apresentados pelos microsserviços é a dificuldade em orquestrar os microsserviços e a maior latência na comunicação, além disso, eles são interdependentes e devem permanecer funcionando mesmo quando outros serviços dos quais eles dependem não estão disponíveis ou não estão respondendo adequadamente. Portanto, testar um aplicativo de

microsserviços necessita de uma estratégia que não só os considere isoladamente, mas também as dependências do serviço e é por isso que uma boa estratégia de testes de microsserviços geralmente verificam o funcionamento de cada um isoladamente e, em seguida, verifica a comunicação entre eles. Os testes de software de microsserviços garantem que os mesmos funcionem de acordo com esperado de maneira eficiente e oportuna. Nesse contexto, os principais tipos de testes são: teste funcional, teste de carga e teste de resiliência.

Diversas ferramentas podem ser usadas para testar microsserviços em diferentes níveis, incluindo testes de ponta a ponta e testes de regressão, entre outros. (HERNáNDEZ et al., 2021) compararam várias dessas ferramentas, observando detalhes e provocando discussões sobre a aplicabilidade delas. Eles observaram que algumas dessa ferramentas fornecem suporte de infraestrutura para a execução de casos de teste.

Uma das empresas de tecnologia mais representativas em todo o mundo, a Netflix (NETFLIX, 2022), desenvolveu sua própria ferramenta de código aberto para testar a resiliência de sistema de microsserviços, o Chaos Monkey, (CHAOS. . . , 2021) esta ferramenta intencionalmente derruba servidores como forma de testar a tolerância a falhas de um ambiente em nuvem. Outra ferramenta interessante é o Hoverfly (HOVERFLY, 2022), que fornece suporte para simular a injeção de falhas nas comunicações de/para uma API, permitindo escrever testes automatizados para simular a comunicação com outros microsserviços. O projeto Gremlin (GREMLIN, 2022) fornece um proxy leve que pode ser anexado a cada microsserviço para processar requisições HTTP, o usuário pode injetar cuidadosamente a falha em hosts ou contêineres com esta ferramenta, independentemente de onde eles estejam, seja na nuvem pública ou em seu próprio data center.

2.3 Resiliência

A resiliência torna-se apropriada quando existe uma expectativa de que um determinado software seja capaz de sobreviver e se recuperar de interrupções (UDAY; MARAIS, 2015). Os sistemas de serviços online têm sido utilizados por milhões de usuários e apesar dos esforços dos profissionais de tecnologia para garantir sua qualidade, ainda existem diversos desafios e ameaças, que eventualmente provocam incidentes que podem diminuir a satisfação do cliente pelo serviço ou ocasionar perdas econômicas. A importância da resiliência foi apontada em muitos livros de microsserviços (NEWMAN, 2015) (WOLFF, 2017) (NADAREISHVILI et al., 2016) e estudos que discutem os principais recursos dos Sistemas MSA (DRAGONI et al., 2016) (DRAGONI et al., 2017). Nestes livros e estudos, alguns mecanismos de sistema típicos consistem em operações que reagem à degradação do serviço, como balanceamento de carga, disjuntores, gateway de API, anteparos, etc. Esses mecanismos são chamados de mecanismos de resiliência. Alguns pesquisadores aprimoraram esses mecanismos de resiliência para sistemas MSA nos últimos anos. (BRILHANTE; COSTA; MARITAN, 2017) (JENKINS et al., 2017) (NIU; LIU; LI, 2018) (CORTE et al., 2018) (MONTESI; WEBER, 2018). Os sistemas MSA resilientes devem manter os serviços degradados de desempenho em um nível muito baixo, o

que é inaceitável pelos usuários, e fazer com que o desempenho dos serviços degradados volte ao normal o mais rápido possível (YIN et al., 2019).

2.3.1 Teste de Resiliência

O teste de software, no geral, envole técnicas para testar os aspectos do software com relação à funcionalidades. Em particular, o teste de resiliência valida requisitos não funcionais do software, e é de fundamental importância para avaliar como um aplicativo funcionará sob estresse ou em circunstâncias "caóticas" que são inevitáveis no ambiente produtivo.

O teste de resiliência mostra quão bem um programa opera sob condições caóticas. O seu objetivo é observar quão bem o aplicativo pode continuar desempenhando suas funções principais e preservar a integridade dos dados durantes tais condições caóticas, além disso, demonstra a capacidade do software de se recuperar após interrupções momentâneas e outros fatores aleatórios (YIN et al., 2019).

Nos dias de hoje, qualquer tempo de inatividade do aplicativo pode ser muito prejudicial para a organização, podendo acarretar na perdas de clientes, portanto, realizar testes de resiliência é crucial para se preparar para essas falhas inevitáveis de software.

2.3.2 Engenharia do Caos

Quando um aplicativo é implantado em produção, ele enfrenta incertezas como rede instável, serviços de terceiros indisponíveis e tráfego desequilibrado. Um aplicativo bem projetado deve ser capaz de suportar esses cenários imprevistos sem nenhum impacto sério para os usuários. Ferramentas para engenharia do caos injetam ativamente diferentes tipos de falhas em um sistema de produção e tentam aprender como o sistema se comporta sob perturbação. Se o sistema se comportar conforme o esperado, esses experimentos aumentam a confiança que os desenvolvedores têm sobre a resiliência do sistema (SIMONSSON et al., 2019).

(BASIRI et al., 2017) observaram que empresas como Amazon, Google e Facebook aplicam técnicas de injeção de faltas para verificar a resiliência de seus sistemas. Chaos Monkey (CHAOS. . . , 2021) é uma ferramenta de código aberto mantida pela Netflix para injeção de faltas aleatórias, sendo utilizada em larga escala. Ela é capaz de simular falhas de instâncias de serviços em execução eliminando uma zona de disponibilidade ou uma região inteira do sistema. Porém, a ferramenta não analisa automaticamente o comportamento do aplicativo, que é algo necessário para encontrar bugs de implementação de maneira ágil. Além disso, a técnica de injeção de falhas não é muito eficiente e muito tempo e recursos são gastos pela exploração de cenários redundantes.

2.3.3 Degradação do serviço

Confiabilidade, disponibilidade, tolerância a falhas, etc. são propriedades tradicionais do sistema para avaliar a capacidade de um sistema de software para lidar com as falhas (ISO

Central Secretary, 2011). Tais propriedades possuem métricas que são utilizadas para validar o estado de um sistema de software, basicamente, assumindo que dois estados são possíveis, como “disponível / indisponível” ou “confiável / não confiável”. Porém, alguns estudos (GUNAWI et al., 2016) (GUNAWI et al., 2014) (GUNAWI et al., 2018) observaram que os sistemas em nuvem são mais propensos a estar em um estado “manco” ao invés de totalmente indisponíveis. O estado “manco” (GUNAWI et al., 2018) significa que embora um sistema em nuvem possa fornecer serviços com funcionalidades normais, os serviços funcionam em desempenho de acordo com a satisfação dos usuários, o que é conhecido como degradação do serviço. A degradação do serviço é um fenômeno que ocorre nos sistemas MSA em que um serviço é mantido degradado devido a uma interrupção. O estado degradado de um serviço é confirmado avaliando se o desempenho do serviço é inferior ao benchmark de desempenho do serviço (YIN et al., 2019).

2.4 Considerações finais

Este trabalho, se concentra em avaliar a resiliência de arquiteturas baseadas em microsserviços. Para tal, foi elaborada uma abordagem de teste de resiliência que causa a interrupção de um ou mais microsserviços durante a execução de testes de API do sistema em teste (SUT) cujo objetivo é observar possíveis falhas e a degradação dos serviços. A abordagem desenvolvida foi aplicada ao OCARIoT e foi verificado falhas relacionadas a rede, tempo limite, indisponibilidade, comportamentos inesperados e degradação dos serviços.

3 ABORDAGEM

Nesta seção, apresentamos o modelo da abordagem de teste de resiliência proposta e suas características, como: a estrutura da abordagem, os dados coletados, e como foi feita organização e análise desses dados.

3.1 SPLat

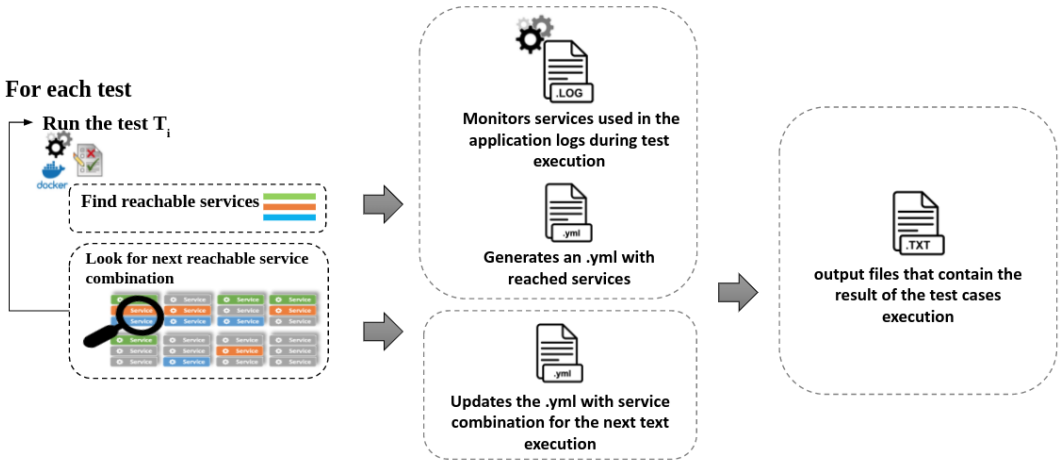
Supondo que estamos validando um sistema de microsserviços, o que podemos fazer para observar o comportamento do sistema se algum desses serviços falhar? De forma simples, o primeiro passo seria causar a falha (interrupção) em um dos serviços, utilizar uma ou mais funcionalidades do sistema (realizar algum fluxo conhecido) e observar como o sistema se comportou durante esse fluxo. Digamos que nosso sistema possui 3 serviços (S_1, S_2 e S_3), e 2 funcionalidades conhecidas (T_1 e T_2), é muito importante entender como nosso sistema irá se comportar, em um contexto de falha nos serviços, durante a execução de cada uma dessas funcionalidades. Para uma análise mais acurada, é imprescindível que analisemos a conduta do sistema para cada fluxo conhecido, em combinações distintas de interrupções dos serviços. Portanto, precisamos executar cada uma das 2 funcionalidades conhecidas nas seguintes combinações:

1. nenhum serviço foi interrompido
2. o serviço S_1 foi interrompido
3. o serviço S_2 foi interrompido
4. o serviço S_3 foi interrompido
5. os serviços S_1 e S_2 foram interrompidos
6. os serviços S_1 e S_3 foram interrompidos
7. os serviços S_2 e S_3 foram interrompidos
8. os serviços S_1, S_2 e S_3 foram interrompidos

O SPLat é uma ferramenta para teste de sistemas altamente configuráveis (KIM et al., 2013) (SOUTO; D'AMORIM; GHEYI, 2017) que nos permite realizar as tarefas mencionadas anteriormente. Empregando o mesmo exemplo, onde temos um sistema que possui 3 serviços e 2 funcionalidades conhecidas, o SPLat é utilizado para gerar todas as possíveis combinações de serviços para cada funcionalidade (nesta abordagem, utilizamos casos de testes que serão melhor detalhados posteriormente), configurar o sistema com os serviços eleitos na combinação, executar a funcionalidade e colher informações que nos permite analisar os comportamentos do sistema. Esta abordagem recebe como entrada o sistema de microsserviço, uma lista de serviços

e um conjunto de testes de sistema 2.1.2 já existentes. Para cada caso de teste, é realizado um procedimento para descobrir quais serviços são alcançados durante a execução do teste 3.1.1, identificados os serviços alcançados são geradas combinações 3.1.2.1 dos serviços que estarão ativos nas próximas execuções 3.1.2.2. Ao término da execução dos testes, são gerados arquivos que irão conter informações pertinentes a execução completa de cada caso de teste 3.1.3. Mais detalhes sobre a abordagem e as etapas mencionadas, serão apresentados nas próximas seções. A Figura 1 ilustra a abordagem baseada em SPLat.

Figura 1 – Arquitetura da abordagem original do SPLAT



Fonte: Elaborada pelo autor (2022).

3.1.1 Descoberta dos serviços alcançados pelo caso de teste

A lista de serviços dada como entrada para a abordagem contém todos os serviços do sistema e é utilizada para que o mesmo seja inicialmente executado com todos os seus serviços. Inicializado o sistema de microsserviço, é realizada uma verificação para garantir que todos os serviços estão ativos e em pleno funcionamento. Posteriormente, o primeiro caso de teste do conjunto de testes é identificado e executado, ao longo da execução os logs da API Gateway são monitorados para verificar quais serviços do sistema foram utilizados, após o término da execução do teste, é gravado em um arquivo o nome de todos os serviços que foram alcançados durante a execução.

3.1.2 Execuções do caso de teste

A lista gerada com os serviços alcançados pelo caso de teste, viabiliza a identificação dos serviços utilizados pelo teste e essa informação é utilizada para causar interrupções nesses serviços, executar o teste e observar o comportamento do sistema nesse contexto de intermitência.

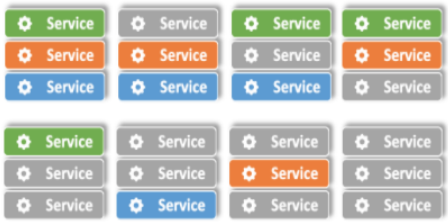
Todos os serviços alcançados são considerados para a interrupção, com exceção de alguns serviços, como: rabbitmq, redis, vault, consul e outros que são imprescindíveis para o funcionamento do sistema e são indicados para o SPLat na lista de serviços fornecida como

dado de entrada. Considerando a lista de serviços alcançados pelo teste, a abordagem irá gerar combinações distintas dos serviços que deverão estar ativos nas próximas execuções do caso de teste. A quantidade de combinações que serão geradas está diretamente relacionada a quantidade de serviços alcançados.

3.1.2.1 Geração da próxima combinação de serviços

Nessa etapa é gerada a combinação dos serviços que devem estar ativos na execução do teste. Para todo caso de teste do conjunto de entrada, a primeira combinação de serviços gerada sempre terá todos os serviços alcançados ativos, e a partir desta são geradas combinações que excluem um ou mais serviços. Cada combinação gerada é gravada em um arquivo, que posteriormente será utilizado para executar o sistema ativando apenas os serviços descritos. A Figura 2 ilustra um exemplo das combinações geradas dado que 3 serviços foram alcançados pelo teste, o preenchimento em cinza indica que naquela combinação o serviço deve estar inativo.

Figura 2 – Exemplo das combinações possíveis para 3 serviços



Fonte: Elaborada pelo autor (2022).

3.1.2.2 Execução do caso de teste com a combinação de serviços gerada

Nessa etapa, o arquivo (mencionado na subseção anterior 3.1.2.1) que possui a informação dos serviços que devem estar ativos para a execução é lido e o sistema inicializado de acordo, em seguida é realizada uma verificação para garantir que os serviços eleitos estão ativos e em pleno funcionamento, verificando para cada serviço se a sua porta estava ocupada. Posteriormente o caso de teste é executado.

3.1.3 Saída da abordagem

Para cada caso de teste do conjunto, serão geradas 2^n combinações, onde 2 é referente aos estados possíveis (ativo ou inativo) e n é a quantidade serviços alcançados. Para cada combinação a abordagem registra em arquivos os resultados pertinentes a execução do teste, o status da execução do teste (passou ou falhou), os serviços ativos durante a execução e nos casos onde o teste falhou, o tipo de falha.

A Tabela 1 ilustra o funcionamento da abordagem para a execução de um caso de teste que alcança 3 serviços (S_1, S_2 e S_3). Na primeira execução (#Rodada = 0) todos os serviços alcançados estão ativos. Nesse contexto, o teste teve o resultado esperado (#Status = passou, sem

ocorrência de falha). Como 3 serviços são alcançados pelo teste, 8 combinações distintas de serviços ativos são possíveis, como exemplificado na Tabela 1 (coluna #Microsserviços ativos, linhas 0-7). Na rodada de número 1, o serviço S_1 é interrompido propositalmente, ficando ativos apenas os serviços S_2 e S_3, por conta da interrupção do serviço S_1 a execução do teste falha resultando na falha F_1 (coluna #Falha). Nas próximas rodadas os serviços continuam sendo combinados de forma distinta, até a última rodada (número 7) onde todos os serviços estão interrompidos e o teste falha resultando no falha F_3.

Tabela 1 – Exemplo de execução do SPLat

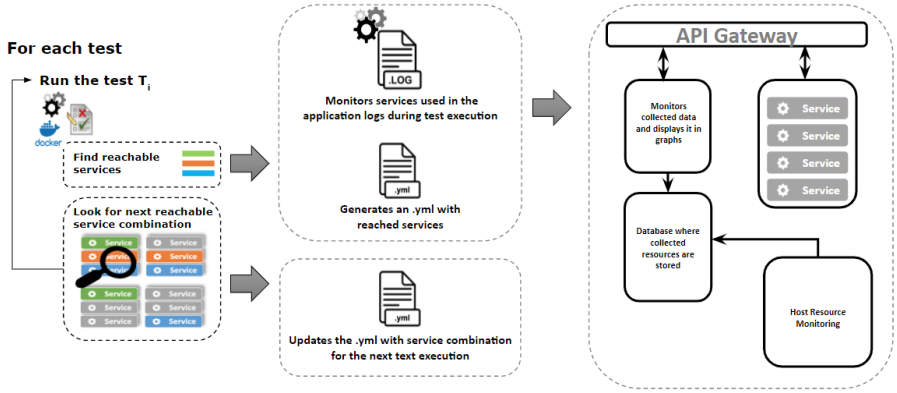
#Rodada	#Status	#Microsserviços ativos	#Falha
0	P	S_1, S_2, S_3	
1	F	~~S_1~~, S_2, S_3	F_1
2	P	S_1, ~~S_2~~, S_3	
3	F	S_1, S_2, ~~S_3~~	F_2
4	F	~~S_1~~, ~~S_2~~, S_3	F_2
5	F	~~S_1~~, S_2, ~~S_3~~	F_3
6	F	S_1, ~~S_2~~, ~~S_3~~	F_4
7	F	~~S_1~~, ~~S_2~~, ~~S_3~~	F_3

Fonte: Elaborada pelo autor (2022).

3.2 SPLat++

Adicionamos na arquitetura do SPLat, mencionada anteriormente, configurações para observar o consumo de recursos do host 3.2.1 onde o sistema de microsserviços é implantado e executado. Nosso objetivo é coletar métricas pertinentes aos serviços envolvidos, como uso de CPU e memória e o tempo de execução de cada teste de sistema e utilizar esses dados para analisar o comportamento do sistema. A abordagem gera como saída uma quantidade considerável de dados, esse conjunto de dados é posteriormente organizado 3.2.3 com a finalidade de facilitar a análise, nos possibilitando encontrar relações com o comportamento do sistema. Mais detalhes sobre a abordagem proposta e as etapas mencionadas, serão apresentados nas próximas seções. A Figura 3 ilustra a abordagem de teste de resiliência proposta.

Figura 3 – Arquitetura da abordagem proposta incluindo projeto de monitoramento dos serviços.

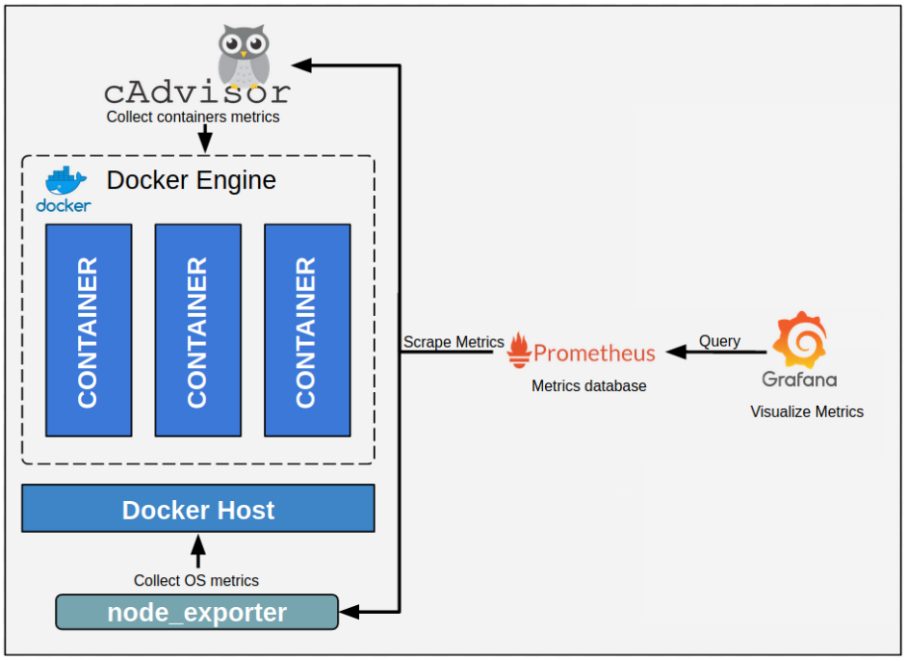


Fonte: Elaborada pelo autor (2022).

3.2.1 Monitoramento dos recursos do host

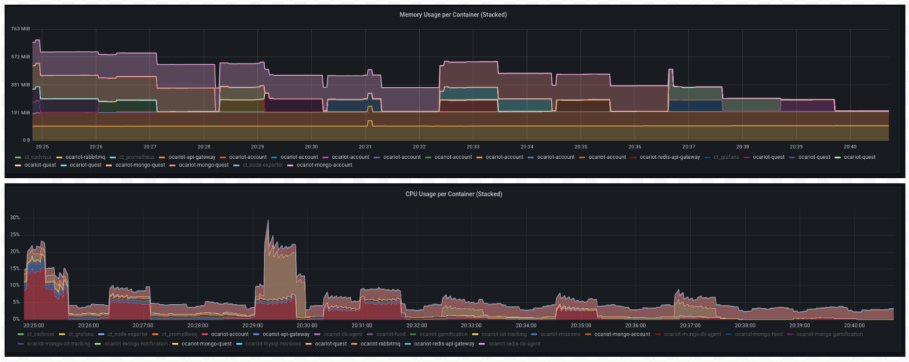
A Figura 4 ilustra o monitoramento dos serviços da plataforma OCARIoT que é realizado através dos serviços Grafana (OPERATIONAL..., 2022), Prometheus (FROM..., 2022), cAdvisor (CADVISOR, 2022) e node_expoter (PROMETHEUS..., 2022). O cAdvisor e o node_exporter são responsáveis, respectivamente, por coletar as métricas dos containers e do host onde a implantação é realizada (O cAdvisor também consegue coletar dados de um cluster com múltiplos nós). Os dados coletados são capturados pelo Prometheus, que tem o cAdvisor e o node_exporter como seus provedores de dados. Por fim, o Grafana se conecta ao Prometheus e monitora os dados coletados, exibindo-os em gráficos, como ilustrado na Figura 5. No Grafana podemos exportar os dados coletados por meio de arquivos (.xlsx, .csv, .json), além disso, também é possível filtrar os dados que serão exportados, selecionando um período específico de tempo, por meio da data e hora de início e término.

Figura 4 – Monitoramento dos serviços da plataforma OCARIoT.



Fonte: Elaborada pelo autor (2022).

Figura 5 – Gráficos gerados pelo Grafana.



Fonte: Elaborada pelo autor (2022).

3.2.2 Tempo de execução dos testes

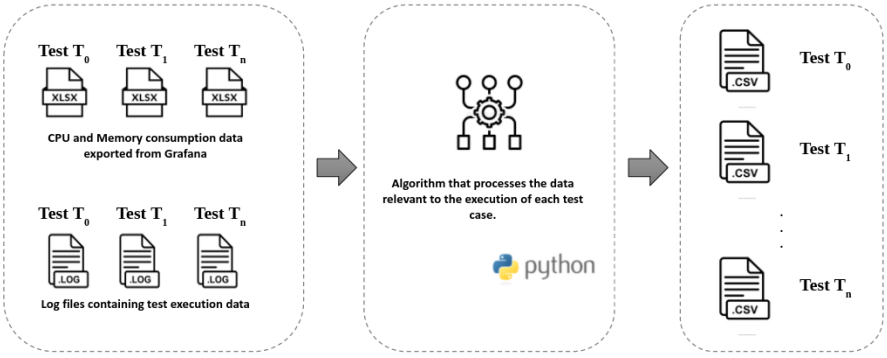
Executando o plano de testes (para mais informações sobre os testes, visualizar a seção 4.3.3), diversas requisições são feitas aos serviços do OCARIoT. Monitoramos o tempo de execução de cada combinação de um caso de teste e analisamos cada tempo de dois modos diferentes: (1) analisando apenas o tempo gasto para que as requisições pertinentes ao teste fossem atendidas pela API Gateway, que denominamos de PR (tempo por requisições) e (2) observando totalmente a execução dos testes pelo SPLat, contemplando a instrumentação, execução e finalização do teste (Total).

3.2.3 Organização dos dados coletados

Uma grande quantidade de dados é coletada após a execução do conjunto de testes, uma parcela desses dados são os arquivos de logs gerados pelo SPLat e neles estão presentes, para cada combinação de todos os testes do conjunto, o status da execução (passou ou falhou), os microsserviços ativos, as falhas ocorridas e a data e hora de cada execução. Grande parte dos dados é gerada pela abordagem proposta, importamos esses dados do Grafana em arquivos *.xlsx*. Utilizando os dados de data e hora da execução dos casos de testes do conjunto de entrada, conseguimos exportar do Grafana os dados relativos a cada teste separadamente, logo, cada arquivo irá conter informações sobre o consumo dos recursos do host por cada serviço que estava sendo executado durante cada combinação do teste. Para que a análise dos dados seja possível e mais produtiva, devido a enorme quantidade, é preciso extrair de cada conjunto de dados (SPLat e SPLat++), as informações pertinentes a cada caso de teste, e uma vez extraídas, agrupá-las em um único ponto, esta organização dos dados melhora a eficiência da análise, possibilitando encontrar relações entre eles.

Devido a quantidade massiva de dados gerados, realizar o processamento mencionado anteriormente exige muito esforço de tempo e o risco de extrair manualmente um dado errado é imenso. A Figura 6 ilustra o algoritmo que implementamos, que recebe como entrada os arquivos *.xlsx* exportados do Grafana e os dados extraídos do SPLat, realiza o processamento e gera novos arquivos *.csv* para cada caso de teste, de modo que podemos melhor observar as métricas pertinentes a cada cenário de teste.

Figura 6 – Processamento dos dados coletados



Fonte: Elaborada pelo autor (2022).

De posse desses resultados temos condições de analisar o comportamento do OCARIoT e identificar possíveis relações que podem diagnosticar estes comportamentos.

Um exemplo do processamento é ilustrado na Tabela 2 onde temos o resultado gerado após execução de um caso de teste que alcança os serviços S_1, S_2 e S_3. Agora, além dos erros, para cada execução temos o consumo dos recursos de cpu e memória, tempo de execução do teste e o total de requisições atendidas pela API Gateway durante cada combinação do caso de

teste.

Tabela 2 – Exemplo de execução da abordagem proposta

#Rodada	#Status	#Microsserviços ativos
0	P	S_1, S_2, S_3

#Tempo de Execução PR (s)	#Tempo de Execução Total (s)	#Consumo de Memória (GB)
T_0	T_0	M_0

#Consumo de CPU (%)	#Total de requisições	#Falha
C_0	R_0	

Fonte: Elaborada pelo autor (2022).

4 ESTUDO DE CASO

Nesta seção, apresentamos as questões de pesquisas investigadas neste trabalho, em seguida mostramos detalhes da arquitetura de microsserviços da plataforma OCARIoT, as configurações do ambiente e da máquina onde o ambiente foi implantado, os testes utilizados e suas configurações, por fim, respondemos as questões de pesquisa relacionadas a abordagem e a resiliência da plataforma.

4.1 Questões de Pesquisa

4.1.1 É possível identificar alguma relação entre a quantidade de serviços indisponíveis e o tipo de erro?

Avaliamos a distribuição da quantidade de serviços indisponíveis quando cada erro ocorreu. Isso nos fornece uma perspectiva de quais erros geralmente ocorrem dado que uma determinada quantidade de serviços tiveram seu funcionamento interrompido. Ao analisar esses dados podemos observar se o sistema exibe um padrão de comportamento de acordo com a quantidade de serviços inoperantes.

4.1.2 É possível identificar quais tipos de erro ocorreram com maior frequência?

Avaliamos a quantidade de cada tipo de erro que ocorreu durante as combinações que falharam. Ao analisar esses dados podemos identificar a frequência de cada tipo de erro e quais ocorreram com maior frequência.

4.1.3 É possível identificar quais serviços são mais críticos para a plataforma?

Avaliamos quais os serviços que se encontravam inoperantes a cada falha ocorrida, com o objetivo de identificar o grau de importância da operacionalidade de determinados serviços. Ao analisar esses dados podemos observar quais são os serviços mais críticos para a plataforma.

4.1.4 É possível identificar se Houve degradação dos serviços?

A degradação do serviço é um fenômeno que ocorre nos sistemas MSA em que um serviço é mantido degradado devido a uma interrupção. O estado degradado de um serviço é confirmado avaliando se o desempenho do serviço é inferior ao benchmark de desempenho do serviço. (YIN et al., 2019). Como mencionado na seção de abordagem (3), colhemos informações pertinentes ao tempo de execução e a quantidade de requisições feitas a API Gateway durante cada combinação de testes. Analisar esses dados nos permite observar se ocorreu degradação dos serviços oferecidos pela plataforma.

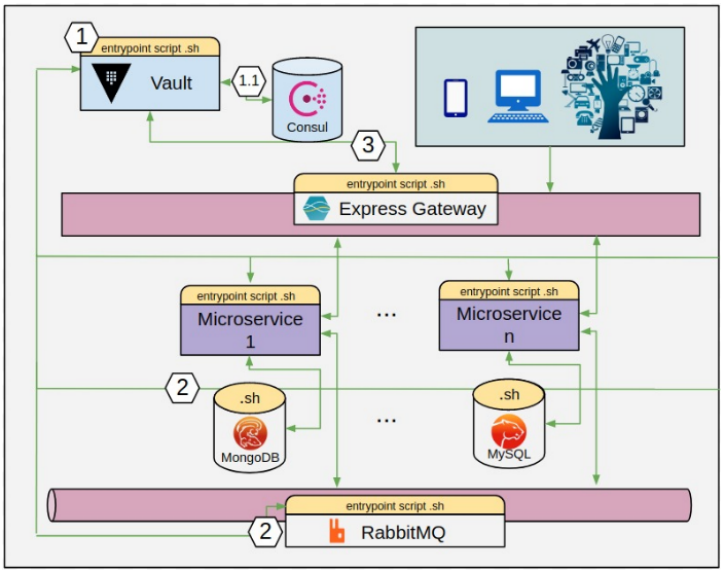
4.1.5 É possível identificar alguma relação entre o consumo de recursos da máquina e o tipo de erro?

Avaliamos a distribuição do consumo de CPU e Memória quando cada erro aconteceu. Esses dados nos possibilita analisar qual a relação existente entre o consumo de recursos da máquina com o tipo de erro ocorrido.

4.2 OCARIoT

A Figura 7 representa a ideia geral da arquitetura de software definida para IoT de saúde (JAMSHIDI et al., 2018) (A. . . , 2021). Os microsserviços podem ser desenvolvidos utilizando diferentes tecnologias, linguagens e banco de dados. No caso do OCARIoT existem componentes desenvolvidos em Nodejs, Python e Java que é integrado a um barramento de canal de mensagens. Para persistência dos dados, são utilizadas bibliotecas de criptografia para microsserviços que utiliza bancos de dados PSMDB (Percona Server for MongoDB) (PERCONA. . . , 2022a) e PSMYSQL (Percona Server for MySQL) (PERCONA. . . , 2022b) que estão em conformidade com requisitos da GDPR (GENERAL. . . , 2021) na Europa. Foi definido um único ponto de entrada para microsserviços, um API Gateway de código aberto (MICROSERVICES. . . , 2021) construído em Express.js. Foi utilizado o RabbitMQ (MESSAGING. . . , 2021), um muito popular intermediário de mensagem de código aberto, para os esforços em alta disponibilidade e escalabilidade do sistema. O gerenciamento de privacidade e segurança da plataforma foi desenvolvido baseado nas ferramentas de código aberto Vault (MANAGE. . . , 2021) e Consul (SERVICE. . . , 2021).

Figura 7 – Arquitetura de microsserviços do OCARIoT incluindo os serviços de infraestrutura.



Fonte: Elaborada pelo autor (2022).

A Tabela 3 apresenta os principais componentes da arquitetura do OCARIoT. A inicialização segue a ordem indicada na Figura 7. Em (1) o Vault gera as chaves de criptografia e o token raiz. Em (1.1) o Consul é inicializado e servirá como back-end do Vault, fornecendo o armazenamento de dados criptografados e gerando outras chaves secretas, como as de banco de dados, canal de mensagens, etc. Em (2), ocorre a inicialização do barramento de mensagens e de todos os bancos de dados dos microsserviços. Em (3) temos a inicialização dos microsserviços. A próxima seção fornece mais detalhes por meio da instanciação do projeto OCARIoT.

Tabela 3 – Componentes arquitetônicos.

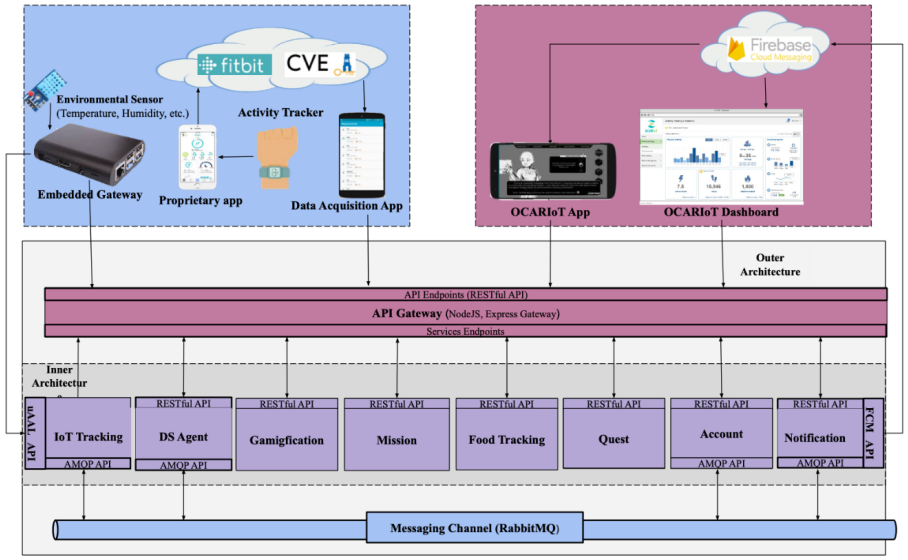
Componente	Descrição
Apps, Dashboards, IoT Gateways	Ferramentas de IoT de saúde para agregar dados e serviços de sincronização. Os dados podem ser sincronizados diretamente ou podem ser solicitados por um serviço de terceiros
Vault	Oferece políticas de gerenciamento de alto nível para proteger, armazenar e controlar o acesso garantindo a segurança
Consul	Estende as funções do Vault criando alta disponibilidade para gerenciar chaves, certificados, entre outros
Express Gateway	Ponto de entrada único para interação de clientes e microsserviços. Realização dos padrões de gateway e agregador de API. É usado para gerenciar a chamada de serviço entre os componentes externos e os microsserviços
RabbitMQ Bus	Implementação de um canal de mensagens, um mecanismo usado para permitir a comunicação assíncrona entre microsserviços

Fonte: Elaborada pelo autor (2022).

4.2.1 Instanciação

A arquitetura de software da plataforma OCARIoT compreende a arquitetura de microsserviços proposta da Figura 8.

Figura 8 – Arquitetura de microsserviços do projeto OCARIoT.



Fonte: Elaborada pelo autor (2022).

Os principais componentes são:

1. Dispositivos Vestíveis: dispositivos comerciais (por exemplo, Fitbit) que são usados durante os pilotos, o acesso ao servidor proprietário é permitido pelos fornecedores.

2. Gateway Embarcado: O gateway embarcado proporciona a integração dos sensores com a plataforma.

3. Aplicativo de Aquisição de Dados: aplicativo para celular que gerencia a coleta de dados de diferentes dispositivos durante os pilotos e mostra informações importantes para os profissionais de saúde. Disponível em github.com/ocariot/da-app.

4. API Gateway: implementado utilizando o Express Gateway conforme sugerido na arquitetura do software. Disponível em github.com/ocariot/api-gateway.

5. Canal de Mensagem: implementado utilizando RabbitMQ conforme sugerido na arquitetura do software. Disponível em github.com/ocariot/rabbitmq-client-node.

Oito microsserviços foram definidos. Alguns não estão disponíveis devido à cláusulas proprietárias. Alguns disponíveis são:

1. Account: gerenciamento e autenticação de usuários. Disponível em github.com/ocariot/account.

2. IoT Tracking: rastreamento de atividades, sono, dados ambientais e medições. Disponível em github.com/ocariot/iot-tracking.

3. Data Sync: sincronização de dados de plataformas vestíveis, permitindo gerenciamento de token, revogação e publicação de dados sincronizados no canal de mensagens. Disponível em github.com/ocariot/data-sync-agent.

4. Notification: Envio de notificações aos aplicativos clientes do usuário da plataforma OCARIoT. Disponível em https://github.com/ocariot/notification-service.

5. Gamification: Fornece missões no jogo do aplicativo OCARIoT, a fim de desencadear a mudança comportamental de longo prazo das crianças para hábitos saudáveis.

6. Mission: Vincula a inteligência da solução OCARIoT dentro do aplicativo e o dashboard através da estratégia de gamificação.

7. Quest: Fornece questionários a fim de coletar informações sobre hábitos nutricionais, condições de saúde e sociodemográficas, atividade física e de sono de crianças e pais.

8. Food Tracking: Permite coletar dados sobre hábitos nutricionais e de atividade física de crianças e pais. Além disso, reúne informações sobre condições sociodemográficas e de saúde.

4.3 Avaliação

É um grande desafio criar e implantar um sistema baseado em microsserviços. Além disso, é necessário manter o aplicativo em execução em um ambiente em que algum tipo de falha certamente ocorrerá. A infraestrutura do ambiente de um sistema de microsserviços é muito dinâmica e eventualmente partes desses ambientes se deparam com incertezas. Um servidor que executa um serviço pode falhar ou ficar indisponível, a rede pode sofrer instabilidade ou um banco de dados pode ter problemas. Sistemas de microsserviços devem ser projetados para lidar com falhas de infraestrutura. É desejável que o sistema seja capaz de responder à falhas de modo a evitar tempo de inatividade ou perda de dados. A meta de resiliência é retornar o sistema para um estado totalmente funcional após uma falha, logo, uma boa abordagem de testes é verificar se ele pode dar continuidade as suas operações dado que ocorreram falhas nos recursos subjacentes.

4.3.1 Configuração do Ambiente

A abordagem proposta foi aplicada ao sistema OCARIoT e para tal foram necessárias algumas configurações do ambiente e dos testes. Os serviços RabbitMQ, API Gateway e Redis API Gateway, juntamente com suas dependências, são obrigatórios para a execução do sistema e por isso não podem ser desabilitados. Portanto, esses serviços são fixos para toda e qualquer combinação de teste gerada pela abordagem.

4.3.2 Configuração da Máquina

Aplicamos a abordagem proposta para o sistema OCARIoT usando as seguintes configurações de máquina:

- Local Host (Máquina local)
- Sistema Operacional: Ubuntu 20.04.3 LTS
- CPU: 8 cores 1.80GHz (Intel® Core™ i7-8565U CPU)
- RAM: 16GB
- SSD: 512GB PCIe M.2

4.3.3 Testes utilizados

Todos os testes utilizados são testes de sistema já existentes, baseados na API Gateway que valida a plataforma OCARIoT. Cada teste pode depender de outra funcionalidade do sistema. Portanto, durante a configuração do teste é comum realizar algumas outras operações para que sua execução seja bem sucedida, as quais podem incluir operações de outros serviços. Devido a essas dependências, um teste geralmente exige mais de um serviço da plataforma OCARIoT para ser executado com sucesso.

Analisando os testes de API mencionados anteriormente, identificamos que eles atingem 7 ou 8 serviços, onde apenas 2 correspondem aos serviços da aplicação (Account, que está presente em todos os testes e outro) os demais são serviços de infraestrutura. Três desses serviços são fixos (RabbitMQ, API Gateway e Redis API Gateway), e estarão sempre presentes. Desta forma, selecionamos os testes que atingem 4 ou mais serviços, num total de 6 testes. Este conjunto de testes exercita grande parte dos serviços da plataforma OCARIoT.

Para os 6 testes, nossa abordagem gerou 112 combinações de serviços no total, 104 combinações falharam quando executadas com seu teste correspondente e 8 foram aprovadas. A Tabela 4 ilustra essas estatísticas e apresenta o serviço da aplicação que está sendo testado (coluna #Serviço) por cada caso de teste, e o número total de serviços tocados na coluna #Serviços. Vale salientar que o serviço Account não aparece na Tabela 4 porque todos os testes dependem dele, uma vez que este serviço está vinculado a autorização de acesso a determinados recursos da API da plataforma e por isso ele é tocado por todos os testes.

Tabela 4 – Estatísticas gerais para a aplicação da abordagem de testes no OCARIoT

Identificador do Teste	#Serviço	#Total de Combinações	#Passaram	#Falharam	#Serviços
0	quest	16	1	15	7
1	missions	16	1	15	7
2	missions	16	1	15	7
3	gamification	16	1	15	7
4	food	16	2	14	7
5	ds-agent	32	2	30	8

Fonte: Elaborada pelo autor (2022).

Observamos também que os casos de testes de 0 a 3 tinha apenas uma combinação de serviços aprovada, que equivale à combinação com todos os serviços disponíveis. Os casos de teste 4 e 5 apresentam duas combinações de serviços que passam por dependência parcial de um dos serviços, possibilitando a execução dos casos de testes com sucesso.

4.4 Resultados

4.4.1 Relação entre a quantidade de serviços indisponíveis e o tipo de erro

Para cada teste que falhou, identificamos o erro ocorrido e a quantidade de serviços indisponíveis durante a execução do teste.

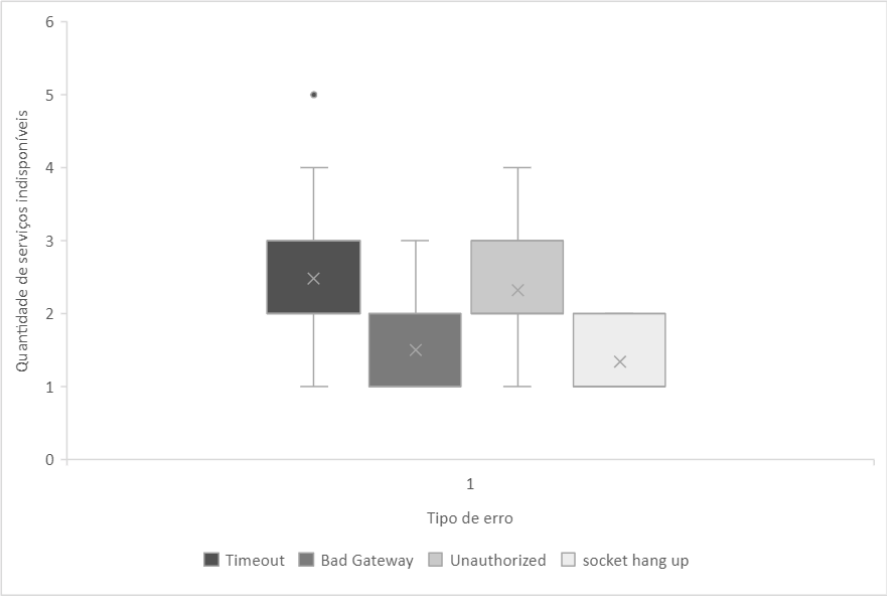
A Tabela 5 apresenta a quantidade de serviços indisponíveis quando cada tipo de erro ocorreu. De acordo com a Figura 9 identificamos que no máximo 5 serviços estão indisponíveis quando ocorre o erro de tempo excedido, mas na maioria dos casos há 2 ou 3 serviços indisponíveis e para o erro de “desligar o soquete” no máximo 2 serviços estão inoperantes.

Tabela 5 – Estatísticas da quantidade de serviços indisponíveis quando cada tipo de erro ocorreu

Qtd de serviços indisponíveis	Timeout	Bad Gateway	Unauthorized	Socket hang up
1	9	7	5	2
2	22	4	13	1
3	18	1	11	0
4	8	0	2	0
5	1	0	0	0

Fonte: Elaborada pelo autor (2022).

Figura 9 – Representa os serviços indisponíveis por tipo de erro.



Fonte: Elaborada pelo autor (2022).

4.4.2 Tipos de erro que ocorreram com maior frequência

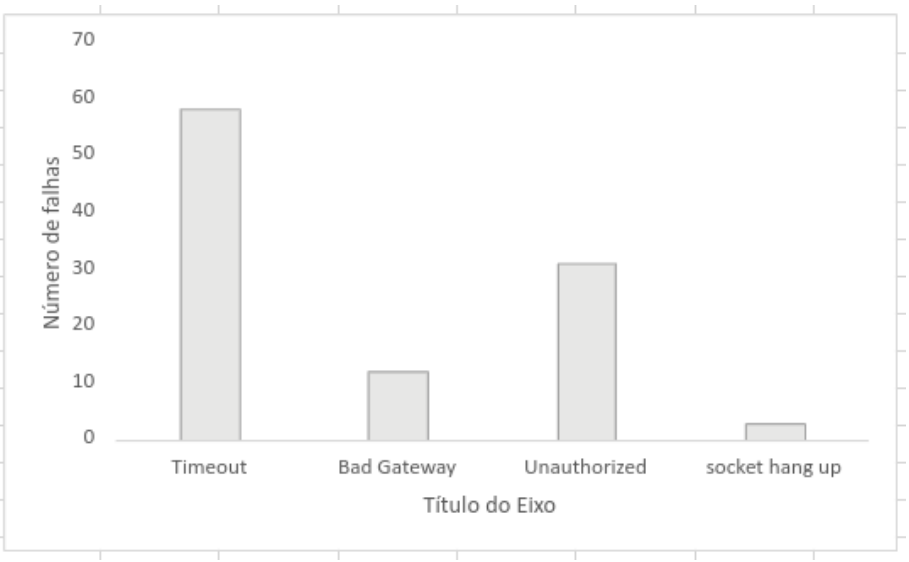
Identificamos a quantidade de combinações que falharam por tipo de erro. A Tabela 6 nos mostra a frequência de cada tipo de erro. De acordo com a Figura 10 podemos identificar que a maioria das falhas está nos erros Tempo excedido e Não autorizado.

Tabela 6 – Estatísticas da quantidade de combinações que falharam por tipo de erro

Tipo de Erro	Número de falhas
Timeout	58
Bad Gateway	12
Unauthorized	31
Socket hang up	3

Fonte: Elaborada pelo autor (2022).

Figura 10 – Representa o número de falhas por tipo de erro.

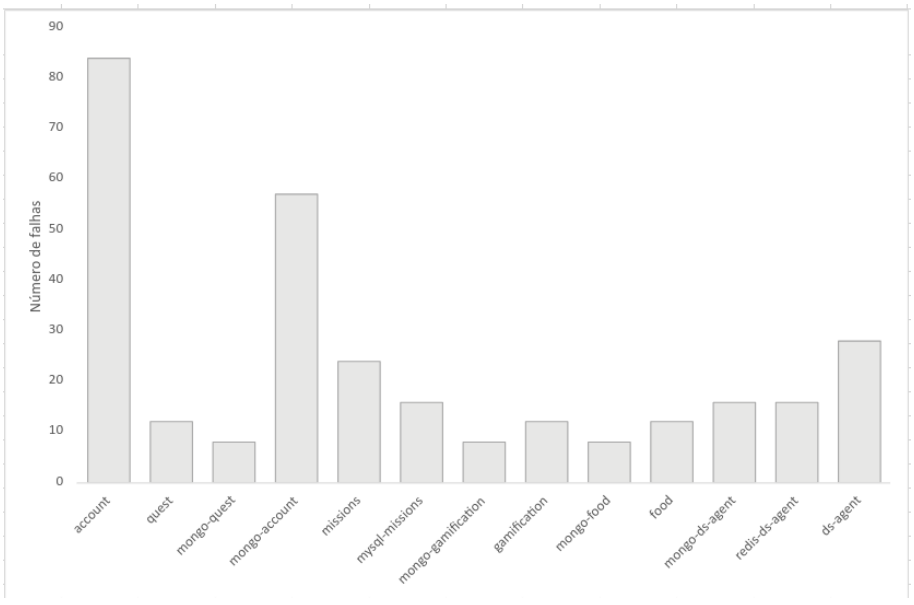


Fonte: Elaborada pelo autor (2022).

4.4.3 Serviços mais críticos para a plataforma

A Tabela 7 apresenta o número de falhas que ocorreram quando um determinado serviço estava inoperante. Observando a Figura 11 identificamos os serviços que tem alta criticidade na plataforma e vimos que 84 combinações falharam quando o serviço de account estava indisponível. A partir dessa análise podemos avaliar e planejar melhorias como aumentar o número de réplicas para os serviços mais críticos.

Figura 11 – Representa o número de falhas por serviços indisponíveis.



Fonte: Elaborada pelo autor (2022).

Tabela 7 – Estatísticas do número de falhas por serviços indisponíveis

Microsserviço	Número de falhas
account	84
quest	12
psmdb-quest	8
psmdb-account	57
missions	24
psmysql-missions	16
psmdb-gamification	8
gamification	12
psmdb-food	8
food	12
psmdb-ds-agent	16
redis-ds-agent	16
ds-agent	28

Fonte: Elaborada pelo autor (2022).

4.4.4 Degradação dos serviços

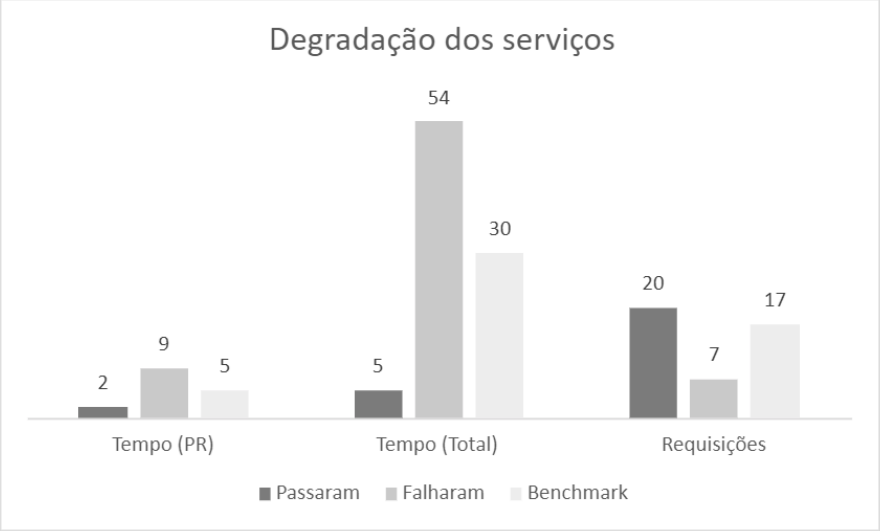
A Tabela 8 representa as médias de tempo de execução de cada combinação de testes e número de requisições. Observamos que quando não havia nenhuma interrupção de serviços e os testes passaram, o tempo médio de execução dos testes foram de 2 segundos considerando apenas o tempo que as requisições foram atendidas pela API Gateway 3.2.2 e 5 segundos considerando o tempo de configuração dos serviços pela ferramenta mais a execução dos testes e em média 20 requisições foram feitas a API Gateway durante os cenários de sucesso. De acordo com a Figura 12 avaliamos que houve degradação do serviço com relação ao tempo de execução, pois estabelecemos como aceitável que em média os cenários de testes deveriam ser executados em até 5 segundos, um pouco mais que o dobro de 2 segundos para quando não havia interrupção de nenhum serviço e observamos que a média obtida foi de 9 segundos, 4 segundos a mais do que o aceitável, é importante salientar que a média obtida poderia ser bem maior nesse caso, porém identificamos que em vários cenários de falha, grande parte das requisições pertinentes aos testes não foram feitas a API Gateway devido a interrupção dos serviços, isso ficou claro quando analisamos as estatísticas referentes as requisições na Figura 12 onde estabelecemos como aceitável que 15% das 20 requisições fossem perdidas no contexto de interrupções, no entanto, vimos que em média apenas 7 requisições foram feitas neste contexto, mostrando que houve degradação do serviço e uma perda média de aproximadamente 65% das requisições.

Tabela 8 – Estatísticas das médias de tempo e número de requisições feitas durante os testes

Status	Tempo (PR)	Tempo (T)	Número de Requisições
P	2s	5s	20
F	9s	54s	7

Fonte: Elaborada pelo autor (2022).

Figura 12 – Representa estatísticas de degradação dos serviços

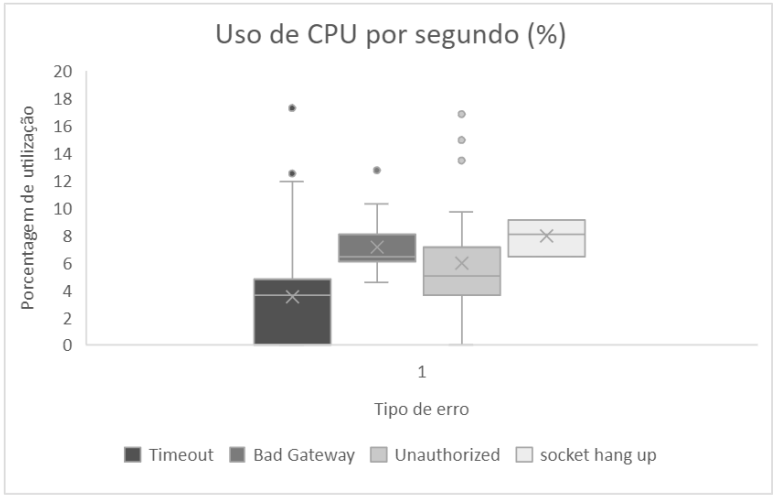


Fonte: Elaborada pelo autor (2022).

4.4.5 Relação entre o consumo de recursos da máquina e o tipo de erro

Para cada teste que falhou, obtivemos o consumo de cpu e memória da máquina hospedeira, obtidos em porcentagem e gigabytes por segundo respectivamente, então dividimos os valores obtidos pelo tempo de execução do teste para que tenhamos o consumo por segundo. Analisando a Figura 13 identificamos que houve maior consumo de cpu entre os erros de Tempo excedido, aproximadamente 18% por segundo e Não autorizado, aproximadamente 17% por segundo. Além disso, percebemos que apenas nestes mesmos erros houveram cenários em que não houve consumo de cpu e memória, logo, entendemos também que a ocorrência dos erros de Tempo excedido e Não autorizado de certo modo está vinculada a interrupção completa do fluxo de requisições de alguns cenários.

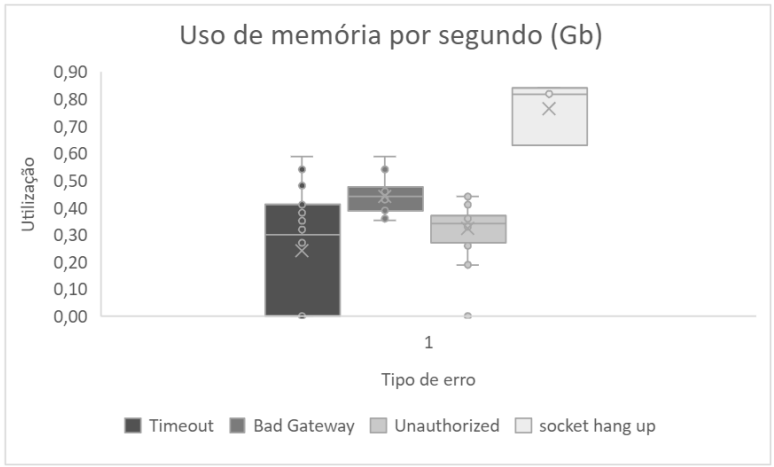
Figura 13 – Representa o consumo de CPU por tipo de erro.



Fonte: Elaborada pelo autor (2022).

Com relação a memória, identificamos que houve um maior consumo, aproximadamente 0,90 gigabytes por segundo, quando ocorreu o erro de “desligar o soquete” como mostra a Figura 14. Além disso, identificamos que em nenhum momento na ocorrência dos outros tipos de erro, o pico de consumo de memória foi equivalente ao do erro de “desligar o soquete” que superou todos os outros, essa estatística pode ser útil para identificar possíveis gargalos e avaliar melhorias que possam ser aplicadas. Novamente, percebemos que nos erros de Tempo excedido e Não autorizado não houve o consumo de memória, reforçando a análise de que estes tipos de erros possuem relação com a interrupção completa de alguns cenários de testes.

Figura 14 – Representa o consumo de Memória por tipo de erro.



Fonte: Elaborada pelo autor (2022).

4.5 Ameaças à validade

Uma ameaça à validade do SPLat++ é que dois dos serviços do OCARIoT não foram considerados (IoT Tracking e Notification), pois os testes de sistema existentes para estes serviços estavam depreciados e quando executados, mesmo com todos os serviços ativos, os testes falhavam. Portanto, dois casos de testes em particular foram desprezados, com isso, os serviços que esses testes tinham como alvo não foram considerados, e suas atuações não fizeram parte dos nossos experimentos.

Outra ameaça é que os tempos de execução para cada combinação de testes foi extraído de forma manual. Fizemos 2 revisões para nos certificar de que os dados foram extraídos precisamente. Os arquivos de logs da API Gateway gerados contém todas as requisições realizadas durante todas as combinações do caso de teste, porém estes dados não estão classificados por combinação. Trabalhos futuros adicionarão a abordagem uma maneira de logar esses tempos de forma sincronizada com cada combinação.

Outra ameaça é a generalização dos resultados experimentais. O SPLat++ foi avaliado apenas no OCARIoT que é um sistema IoT baseado em microsserviços, implementado com diferentes tecnologias que contém 8 serviços em contêiner. A realização de mais experimentos em outros tipos de aplicativos pode alterar algumas respostas às questões de pesquisa. Trabalhos futuros são necessários para explorar outros domínios de aplicação e avaliar a generalização dos resultados experimentais.

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou de forma geral arquiteturas de referência para saúde baseadas em IoT e microsserviços, e contribuiu com a implementação de uma abordagem de avaliação do requisito não funcional resiliência. Foi realizado um estudo empírico e observamos a resiliência de uma arquitetura baseada em microsserviços para o projeto OCARIoT. Produzimos uma abordagem de testes de resiliência que propositalmente causa a interrupção de um ou mais serviços durante a execução de testes de API existentes, com o objetivo de identificar falhas ou degradação do serviço. Esta abordagem foi aplicada ao projeto OCARIoT e observamos falhas reais relacionadas a tempo limite, erro de servidor, rede e comportamentos inesperados, além disso, contemplamos a degradação do serviços no que diz respeito à quantidade de requisições perdidas durante os cenários de testes e ao aumento considerável no tempo de execução dos mesmos.

5.1 Trabalhos futuros

- Utilização de concentrador de logs com o intuito de descobrir os serviços que são alcançados por um determinado teste;
- Realizar os testes de resiliência fazendo uso com redundância, réplicas e políticas de reinicialização;
- O trabalho permite realizar um plano de ações em cima das falhas coletadas, entretanto é necessário discriminativo no tocando a exibir quais serviços;
- Definir os requisitos mínimos para cada um dos serviços inerentes a arquitetura;
- Definir quais pontos (serviços) seriam passíveis de redundância.

5.2 Ferramentas utilizadas

- SPLat++ 3.2: Abordagem proposta. Disponível em https://github.com/TCC-UEPB/splat-plus.
- Projeto para monitoramento dos recursos do host 3.2.1: Monitora os recursos de todos os containers docker sendo executados na máquina. Disponível em https://github.com/TCC-UEPB/monitoring-project.
- Ferramenta para organizar os dados coletados 3.2.3: Recebe os arquivos de logs do SPLat++ e os arquivos contendo dados de consumo de cpu e memória do host e gera novos arquivos com esses dados organizados por caso de teste. Disponível em https://github.com/TCC-UEPB/grafana-results-processor.

REFERÊNCIAS

A definition of this new architectural term. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado 3 vezes nas páginas 11, 15 e 28.

BASIRI, A. et al. Chaos engineering. CoRR, abs/1702.05843, 2017. Disponível em: <http://arxiv.org/abs/1702.05843>. Citado 2 vezes nas páginas 11 e 17.

BASS, L.; WEBER, I.; ZHU, L. DevOps: A Software Architect's Perspective. 1st. ed. [S.l.]: Addison-Wesley Professional, 2015. ISBN 0134049845. Citado 2 vezes nas páginas 11 e 15.

BRILHANTE, J.; COSTA, R.; MARITAN, T. Asynchronous queue based approach for building reactive microservices. In: Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web. New York, NY, USA: Association for Computing Machinery, 2017. (WebMedia '17), p. 373–380. ISBN 9781450350969. Disponível em: <https://doi.org/10.1145/3126858.3126873>. Citado na página 16.

CADVISOR. 2022. Acesso em: 17 jan. 2022. Disponível em: <https://github.com/google/cadvisor>. Citado na página 23.

CHAOS Monkey. Netflix Open Source Platform. 2021. Acesso em: 17 nov. 2021. Disponível em: <https://netflix.github.io/chaosmonkey/>. Citado 3 vezes nas páginas 11, 16 e 17.

CORTE, R. D. et al. An exploratory study on zeroconf monitoring of microservices systems. In: . [S.l.: s.n.], 2018. Citado na página 16.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. CoRR, abs/1606.04036, 2016. Disponível em: <http://arxiv.org/abs/1606.04036>. Citado na página 16.

DRAGONI, N. et al. Microservices: How to make your application scale. CoRR, abs/1702.07149, 2017. Disponível em: <http://arxiv.org/abs/1702.07149>. Citado na página 16.

ESPOSITO, C.; CASTIGLIONE, A.; CHOO, K.-K. R. Challenges in delivering software in the cloud as microservices. IEEE Cloud Computing, v. 3, n. 5, p. 10–14, 2016. Citado 2 vezes nas páginas 11 e 15.

FRANCESCO, P. D.; MALAVOLTA, I.; LAGO, P. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA). [S.l.: s.n.], 2017. p. 21–30. Citado 2 vezes nas páginas 11 e 15.

FROM metrics to insight. 2022. Acesso em: 17 jan. 2022. Disponível em: <https://prometheus.io/>. Citado na página 23.

GARCIA-MORENO, F. M. et al. A microservices e-health system for ecological frailty assessment using wearables. Sensors, v. 20, n. 12, 2020. ISSN 1424-8220. Disponível em: <https://www.mdpi.com/1424-8220/20/12/3427>. Citado na página 15.

GENERAL Data Protection Regulation. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://gdpr-info.eu/>. Citado na página 28.

GREMLIN. 2022. Acesso em: 03 mai. 2022. Disponível em: <https://www.gremlin.com/>. Citado na página 16.

GUNAWI, H. S. et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In: Proceedings of the ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2014. (SOCC '14), p. 1–14. ISBN 9781450332521. Disponível em: <https://doi.org/10.1145/2670979.2670986>. Citado na página 18.

GUNAWI, H. S. et al. Why does the cloud stop computing? lessons from hundreds of service outages. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2016. (SoCC '16), p. 1–16. ISBN 9781450345255. Disponível em: <https://doi.org/10.1145/2987550.2987583>. Citado na página 18.

GUNAWI, H. S. et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. ACM Trans. Storage, Association for Computing Machinery, New York, NY, USA, v. 14, n. 3, oct 2018. ISSN 1553-3077. Disponível em: <https://doi.org/10.1145/3242086>. Citado na página 18.

HEORHIADI, V. et al. Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). [S.l.: s.n.], 2016. p. 57–66. Citado na página 11.

HERNáNDEZ, C. et al. Comparison of end-to-end testing tools for microservices: A case study. In: ____. [S.l.: s.n.], 2021. p. 407–416. ISBN 978-3-030-68284-2. Citado 2 vezes nas páginas 15 e 16.

HOVERFLY. 2022. Acesso em: 03 mai. 2022. Disponível em: <https://hoverfly.io/>. Citado na página 16.

IANCULESCU, M. et al. Microservice-based approach to enforce an ioht oriented architecture. In: . [S.l.: s.n.], 2019. p. 1–4. Citado na página 15.

ISHA; SHARMA, A.; REVATHI, M. Automated api testing. In: 2018 3rd International Conference on Inventive Computation Technologies (ICICT). [S.l.: s.n.], 2018. p. 788–791. Citado na página 14.

ISO Central Secretary. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Geneva, CH, 2011. Disponível em: <https://www.iso.org/standard/35733.html>. Citado na página 18.

JAMSHIDI, P. et al. Microservices: The journey so far and challenges ahead. IEEE Software, v. 35, n. 3, p. 24–35, 2018. Citado 3 vezes nas páginas 11, 15 e 28.

JENKINS, J. et al. A case study in computational caching microservices for HPC. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017, Orlando / Buena Vista, FL, USA, May 29 - June 2, 2017. IEEE Computer Society, 2017. p. 1309–1316. Disponível em: <https://doi.org/10.1109/IPDPSW.2017.40>. Citado na página 16.

KIM, C. H. P. et al. Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2013. (ESEC/FSE 2013), p. 257–267. ISBN 9781450322379. Disponível em: <https://doi.org/10.1145/2491411.2491459>. Citado 2 vezes nas páginas 12 e 19.

MANAGE Secrets and Protect Sensitive Data. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://www.vaultproject.io/>. Citado na página 28.

MESSAGING that just works. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://www.rabbitmq.com/>. Citado na página 28.

MICROSERVICES and Serverless to Production Insanely Fast. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://www.express-gateway.io/>. Citado na página 28.

MONTESI, F.; WEBER, J. From the decorator pattern to circuit breakers in microservices. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. New York, NY, USA: Association for Computing Machinery, 2018. (SAC '18), p. 1733–1735. ISBN 9781450351911. Disponível em: <https://doi.org/10.1145/3167132.3167427>. Citado na página 16.

MYERS, G. J.; SANDLER, C.; BADGETT, T. The Art of Software Testing. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962. Citado na página 14.

NADAREISHVILI, I. et al. Microservice Architecture: Aligning Principles, Practices, and Culture. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 1491956259. Citado na página 16.

NETFLIX. 2022. Acesso em: 03 mai. 2022. Disponível em: <https://www.netflix.com/>. Citado na página 16.

NEWMAN, S. Building Microservices: Designing Fine-Grained Systems. 1st. ed. [S.l.]: O'Reilly Media, 2015. 280 p. ISBN 978-1491950357. Citado 3 vezes nas páginas 11, 15 e 16.

NIU, Y.; LIU, F.; LI, Z. Load balancing across microservices. IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, p. 198–206, 2018. Citado na página 16.

OPERATIONAL dashboards for your data here, there, or anywhere. 2022. Acesso em: 17 jan. 2022. Disponível em: <https://grafana.com/>. Citado na página 23.

PERCONA Server for MongoDB. 2022. Acesso em: 17 mai. 2022. Disponível em: <https://www.percona.com/software/mongodb/percona-server-for-mongodb>. Citado na página 28.

PERCONA Server for MySQL. 2022. Acesso em: 17 mai. 2022. Disponível em: <https://www.percona.com/software/mysql-database/percona-server>. Citado na página 28.

PROMETHEUS exporter for hardware and OS metrics exposed by *NIX kernels, written in Go with pluggable metric collectors. 2022. Acesso em: 17 jan. 2022. Disponível em: <https://github.com/prometheus/node_exporter>. Citado na página 23.

RADEMACHER, F.; SORGALLA, J.; SACHWEH, S. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, v. 35, n. 3, p. 36–43, 2018. Citado 2 vezes nas páginas 11 e 15.

SERVICE Mesh for any runtime or cloud. 2021. Acesso em: 02 dez. 2021. Disponível em: <https://www.consul.io/>. Citado na página 28.

SIMONSSON, J. et al. Observability and chaos engineering on system calls for containerized applications in docker. CoRR, abs/1907.13039, 2019. Disponível em: <http://arxiv.org/abs/1907.13039>. Citado na página 17.

SOUTO, S.; D'AMORIM, M.; GHEYI, R. Balancing soundness and efficiency for practical testing of configurable systems. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). [S.l.: s.n.], 2017. p. 632–642. Citado 2 vezes nas páginas 12 e 19.

UDAY, P.; MARAIS, K. Designing resilient systems-of-systems: A survey of metrics, methods, and challenges. Systems Engineering, v. 18, n. 5, p. 491–510, 2015. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.21325>. Citado na página 16.

WOLFF, E. Microservices: Flexible Software Architecture. Addison-Wesley, 2017. ISBN 9780134650449. Disponível em: <https://books.google.com.br/books?id=YDRSAQAACAAJ>. Citado na página 16.

YIN, K. et al. On representing and eliciting resilience requirements of microservice architecture systems. CoRR, abs/1909.13096, 2019. Disponível em: <http://arxiv.org/abs/1909.13096>. Citado 3 vezes nas páginas 17, 18 e 27.