Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Análise de Desempenho entre os Frameworks Spring Boot e Jersey na Arquitetura de Microserviços

Igor Fabri Ferreira

JUIZ DE FORA
NOVEMBRO, 2018

Análise de Desempenho entre os Frameworks Spring Boot e Jersey na Arquitetura de Microserviços

Igor Fabri Ferreira

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Eduardo Pagani Julio

Coorientador: Daves Marcio Silva Martins

JUIZ DE FORA

NOVEMBRO, 2018

Análise de Desempenho entre os Frameworks Spring Boot e Jersey na Arquitetura de Microserviços

Igor Fabri Ferreira

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Eduardo Pagani Julio
Doutor em Ciência da Computação (UFF)

Daves Marcio Silva Martins
Mestre em em Sistemas Computacionais (UFRJ)

Alessandreia Marta de Oliveira Julio
Doutora em Ciência da Computação (UFF)

Vânia de Oliveira Neves
Doutora em Ciência da Computação e Matemática Computacional (USP)

JUIZ DE FORA

30 DE NOVEMBRO, 2018

Dedico este trabalho aos meus pais e a todos que me apoiaram e que dedicaram parte de suas vidas para a realização desse sonho.

Resumo

Com o crescente e dominante uso dos sistemas computacionais para realizar diversas tarefas, surge uma grande demanda de recursos computacionais e de sistemas bem estruturados para fornecer os diversos serviços solicitados. Nos dias atuais, há uma ampla gama de problemas novos que emergem do mercado devido às novas demandas de serviços, tornando-se um desafio para as empresas. Devido a essa ampla gama de demanda de serviços, muitos sistemas computacionais são desenvolvidos/migrados para uma arquitetura descentralizada e de rápido desenvolvimento e manutenção. Neste trabalho, são apresentadas as tecnologias *Jersey* e *Spring Boot* e feitas análises de desempenho destas tecnologias desenvolvidas em uma arquitetura de microsserviços, utilizando a tecnologia de virtualização *Docker*. O desempenho dos microsserviços são comparados utilizando os parâmetros de latência, vazão, tempo médio de resposta, taxa de perda de solicitações e valores de 90% e 99% dos tempos de resposta. Os resultados mostram que os microsserviços desenvolvidos com o *framework Jersey* possuem as menores latências, tempo médio de resposta, valores de 90% e 99% dos tempos de resposta e a maior vazão em relação ao *framework Spring Boot*, possuindo uma taxa de perda de solicitações acima de uma das implementações apresentadas do *framework Spring Boot*.

Palavras-chave: Framework, microsserviços, Spring Boot, Jax-Rs, Desempenho, Docker, JMeter, Jersey.

Abstract

With the growing and dominant use of computer systems to perform various tasks, there is a great demand for computational resources and well structured systems to provide the various services requested. Nowadays, there is a wide range of new problems emerging from the market due to the new demands of services, making it a challenge for companies. Because of this wide range of service demand, many computing systems are developed/-migrated to a decentralized and rapidly developing and maintenance architecture. In this work, we present the Jersey and Spring Boot technologies. We also conducted performance analyzes of these technologies developed in a microservice architecture using Docker. The performance of the microservices are compared using the parameters of latency, throughput, average response time, rate of lost requests and 90% and 99% response times. The results show that the microservices developed with the Jersey framework have the lowest latencies, average response time, 90% and 99% response times and the highest throughput in comparison to the Spring Boot framework, with a rate of lost requests above of a presented implementation of the Spring Boot framework.

Keywords: Framework, microsserviços, Spring Boot, Jax-Rs, Desempenho, Docker, JMeter, Jersey.

Agradecimentos

Agradeço à Deus por ter me dado toda força, sabedoria, suporte e resiliência para vencer as dificuldades durante todo o percurso.

Agradeço à minha família pelo apoio, em especial à Maria Regina Fabri Ferreira e José Cândido Ferreira Filho, pelo sustento e confiança durante a minha trajetória e por acreditarem em mim.

Agradeço à minha namorada Mariana Alvim Duque pelo apoio, persistência e esperança confiada a mim.

Agradeço à todos os meus amigos que acreditaram e me apoiaram direta ou indiretamente durante minha trajetória, aos bolsistas do NRC e amigos de curso.

Agradeço à todos os técnicos do NRC pela amizade, confiança e suporte para o desenvolvimento deste trabalho.

Agradeço especialmente ao Alcindo Gandhi Barreto Almeida pela amizade e orientação durante o desenvolvimento do projeto que culminou neste trabalho.

Agradeço à meus orientadores Daves Marcio Silva Martins e Eduardo Pagani Julio pelo apoio, paciência e orientação sem a qual este trabalho não se realizaria.

Agradeço às professoras Alessandreia Marta de Oliveira Julio e Vânia de Oliveira Neves pelas aulas e orientações para melhorar este trabalho.

Aos professores do Departamento de Ciência da Computação pelos seus ensinamentos e aos funcionários e alunos do curso, que durante esses anos, contribuíram para meu enriquecimento profissional.

"A descoberta consiste em ver o que todo mundo viu e pensar o que ninguém pensou".

Jonathan Swift

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

API	*Application Programming Interface*
AWS	*Amazon Web Services*
CPU	*Central Processing Unit*
GB	*Gigabyte*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
ICE	*Instituto de Ciências Exatas*
IoT	*Internet of Things*
Java EE	*Java Enterprise Edition*
JAX-RS	*Java API for RESTful Web Services*
JSON	*JavaScript Object Notation*
MAS	*Microservice Application Servers*
MVC	*Model View Controller*
NRC	*Núcleo de Recursos Computacionais*
Paas	*Platform as a Service*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
ROA	*Resource-Oriented Architecture*
SOAP	*Simple Object Access Protocol*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Identifier*
VM	*Virtual Machine*
WSDL	*Web Service Description Language*
RPC	*Remote Procedure Call*
XML	*eXtensible Markup Language*

1 Introdução

1.1 Apresentação e Contextualização do Problema

Nos dias atuais, diversas empresas ao redor do mundo enfrentam os mesmos desafios inerentes ao desenvolvimento de software e sua exposição no ambiente virtual para acesso público (Web). É cada vez mais crescente a demanda por desenvolvimento de funcionalidades e disponibilização de recursos para os usuários que se encontram imersos no ambiente digital e conectados pela Internet.

Este se tornou um ponto de inflexão na história do desenvolvimento de software. Quase todos os elementos da sociedade atual estão conectados por meio da Internet. Empresas locais podem alcançar consumidores de quase qualquer parte do mundo. Em consequência disso, com uma grande quantidade de consumidores alcançados pela Internet, surge também a competição global. Devido à esta ampla competição, desenvolvedores começaram a repensar a construção de aplicações, para se adequarem às novas características (CARNELL, 2017):

- Aumento da complexidade: Aplicações devem se comunicar com outros serviços e as bases de dados situam não somente nos servidores empresariais, mas também em serviços externos fornecidos em nuvem;
- Rápida entrega: Clientes não esperam por novos lançamentos ou incrementos de software. Ao invés disso, eles esperam as funcionalidades prontas do produto desagregadas, e as novas funcionalidades lançadas rapidamente;
- Escalabilidade e desempenho: Aplicações utilizadas em grande escala, são difíceis de prever o volume de transações sendo gerenciadas. Torna-se necessário escala-las entre múltiplos servidores rapidamente e reduzi-las de acordo com o volume necessário;
- Disponibilidade da aplicação: As aplicações empresariais devem ser resilientes. A

ocorrência de falhas ou problemas em uma parte da aplicação não poderá derrubar toda a aplicação.

Durante muito tempo, o desenvolvimento de sistemas foi baseado na arquitetura monolítica. E conforme o sistema cresce, há um aumento na dificuldade de realizar sua manutenção e atualização, além de levar um tempo maior. Com a chegada da arquitetura de microsserviços, o mercado de software se encontra em um novo cenário tecnológico, onde a escalabilidade, flexibilidade, o rápido desenvolvimento e atualização constante são os requisitos principais para o desenvolvimento de sistemas (JARAMILLO; NGUYEN; SMART, 2016).

1.2 Motivação

Este trabalho foi planejado durante um projeto de treinamento profissional no Núcleo de Recursos Computacionais (NRC), o qual é responsável por toda a parte tecnológica do Instituto de Ciências Exatas (*ICE*). O projeto de treinamento profissional inicialmente foi proposto para o desenvolvimento de serviços na arquitetura de microsserviços para fornecimento de recursos ao sistema Integra UFJF [1]. No sistemas anteriores ao Integra foram utilizadas algumas tecnologias integradas de forma a fornecer diferentes recursos para os estudantes e professores do ICE. Porém, com as novas demandas e necessidades, tornou-se necessário o estudo dessas tecnologias para refatora-lo e assumi-lo de forma integral. A partir dessa motivação, este trabalho foi proposto a ser desenvolvido.

1.3 Justificativa

Nos últimos anos, nota-se um aumento dos *frameworks* e tecnologias relacionadas ao desenvolvimento de sistemas de software, que se expandem em diversas áreas. Isso traz algumas das dificuldades que acabam se tornando problemas frequentes enfrentados por grande parte desenvolvedores. Alguns destes problemas são a manutenção e atualização de sistemas monolíticos, sua consequente evolução para tecnologias mais recentes, e a falta

[1] https://oauth.integra.nrc.ice.ufjf.br/login

de recursos e funcionalidades fornecidas pelo *framework* selecionado durante o projeto do sistema de software.

Diante disso, é necessário fazer uma análise das tecnologias disponíveis na atualidade e que se relacionam com os requisitos levantados durante a fase de projeto, afim de que, para o desenvolvimento de sistemas de software de grande porte, sejam satisfeitas as condições iniciais do projeto e permita a escalabilidade das aplicações. Com isso, obtém-se uma diminuição na quantidade de defeitos durante o desenvolvimento e da perda de desempenho quando sujeito a grandes cargas de dados e garantindo a segurança dos dados transmitidos na aplicação. Propõe-se então uma análise e comparação entre os *frameworks Spring Boot* e *Jersey* para avaliar suas características observando sua aplicação em sistemas escaláveis e de grande porte.

1.4 Objetivos Gerais e Específicos

Este trabalho tem por objetivo analisar o desempenho de serviços utilizando a arquitetura de microsserviços no desenvolvimento de um sistema e os *frameworks Spring Boot*[2] e *Jersey* [3]. Desta forma pretende-se obter os resultados para análise que auxiliem em uma tomada de decisão sobre qual tecnologia melhor atende as demandas crescentes para o sistema Integra UFJF.

1.5 Metodologia

Para realizar a análise dos *frameworks*, foi realizado um estudo visando o aprofundamento nos conceitos e funcionalidades de cada um, de forma a avaliar conforme a sua utilização e definir as métricas que melhor se ajustam e relacionem com eles. Nesta fase, foram adquiridas as características de cada um avaliando segundo os aspectos de desempenho, escalabilidade e implantação, para sua posterior utilização. Dessa forma, foram planejados seu ambiente de teste e sua abordagem para analisar sua utilização em um sistema real, de forma que cubram em grande parte suas demandas em um ambiente de produção.

[2]https://spring.io/projects/spring-boot
[3]https://jersey.github.io/

Devido ao ambiente operacional e às devidas formas de acesso em que se encontravam os sistemas em suas versões anteriores, para melhor avaliar estas tecnologias, foi utilizado uma plataforma que represente suas características de produção de forma a abstrair qualquer interferência de outros sistemas, para obtenção dos dados reais e que forneça os resultados de forma precisa.

Desta forma, foram definidas as métricas (medidas) que reproduzam os aspectos do sistema no ambiente de produção. Foram então organizadas de forma a analisar sob as que possuem maior impacto na arquitetura proposta, sendo sujeitas as condições reais de processamento. Para avaliar as tecnologias propostas, foram desenvolvidos três microsserviços com suas respectivas funcionalidades, fornecendo os recursos relacionados aos serviços reais que já estão em funcionamento nas versões anteriores do sistema do Integra.

Então foram efetuados testes com ferramentas amplamente utilizadas no mercado, de forma a obter os valores precisos destas métricas, para que as análises revelem as principais características de desempenho nas tecnologias escolhidas. As análises foram efetuadas através de gráficos e avaliações das correlações entre as métricas e sua importância no ambiente de produção.

1.6 Organização do Trabalho

A organização deste trabalho está relacionada da seguinte forma: o Capítulo 1 estão descritos os objetivos e as principais motivações desse estudo e a metodologia. O Capítulo 2 estão relacionados os conceitos fundamentais para o entendimento do trabalho. O Capítulo 3 estão descritos trabalhos de outros autores que serviram de base e de motivação, sendo analisadas suas principais características que se relacionam com os propósitos do planejamento e desenvolvimento dos testes e das tecnologias descritas posteriormente.O Capítulo 4 estão descritos todos os procedimentos e testes que foram efetuados para a obtenção dos resultados, tal como sua análise e definição das tecnologias mais relevantes do estudo. O Capítulo 5 estão descritos os trabalhos que serão realizados futuramente para a cobertura de todas as etapas do ciclo de vida do sistema.

2 Fundamentação Teórica

2.1 Introdução

Neste capítulo são apresentados os conceitos e fundamentos relevantes para o entendimento do trabalho. Na seção 2.2 são relacionados os conceitos sobre a arquitetura de software. Na seção 2.3 são mostrados os conceitos da arquitetura monolítica. Na seção 2.4 são descritos os conceitos da arquitetura microsserviços. Na seção 2.5 são mostrados os conceitos de *Web Services*. Na seção 2.6 são descritos conceitos sobre a virtualização com Docker. Na seção 2.7 são relacionados conceitos sobre *REST* (*Representational State Transfer*). Na seção 2.8 são descritos conceitos sobre as tecnologias de microsserviços. Na seção 2.9 são mostrados os conceitos de teste de desempenho. Na seção 2.10 é apresentada a ferramenta de teste deste trabalho. Na seção 2.11 são mostradas as considerações parciais das tecnologias. Na seção 2.12 é apresentado uma conclusão sobre as tecnologias descritas.

2.2 Arquitetura de Software

Durante os anos de 1960, o processamento de dados através de software começou a ser adotado por empresas. Escrever software tornou-se uma atividade lucrativa, ainda que não possuíssem os métodos e recursos padrões dos dias de hoje. E mesmo nos dias atuais, pode-se notar que existem *softwares* desenvolvidos que não fazem uso das metodologias, arquiteturas ou ferramentas de Engenharia de Software atuais (SIM, 2005).

No centro de todo sistema bem desenvolvido está a arquitetura de software, com um conjunto de decisões e padrões que formarão sua base estrutural. Em um sistema de software, pensar na sua arquitetura, é pensar sobre o conjunto das principais decisões de projeto realizadas durante o desenvolvimento e sua evolução (TAYLOR; MEDVIDOVIC; DASHOFY, 2009).

A arquitetura de software se manifesta em todos os aspectos de um sistema, in-

cluindo seus elementos estruturais (componentes, conectores e configurações). As decisões de projeto envolvem também a implantação do sistema, suas propriedades não funcionais (escalabilidade, segurança, eficiência) e seus padrões de evolução e adaptação. Estas arquiteturas são frequentemente utilizadas nos sistemas nas formas de padrões e estilos, abrangendo o conjunto de decisões de projeto, correlacionando as principais decisões que guiam o arquiteto de software (MEDVIDOVIC; TAYLOR, 2010).

Para ter uma visão ampla da estrutura dos sistemas de software e seus conceitos, torna-se importante descrever os conceitos de *desing* arquitetural, padrões e estilos arquiteturais, para compreender e auxiliar nas decisões de projeto.

2.2.1 Design Arquitetural

Um *design* arquitetural de um software está voltado para a compreensão da organização e estrutura do sistema. Este é um processo onde se define a organização do software que irá satisfazer aos requisitos funcionais e não-funcionais. É o primeiro estágio no desenvolvimento do projeto de um software e principal ligação do estágio da engenharia de requisitos e o projeto, identificando os principais componentes e suas relações no software. Como resultado, obtém-se um modelo arquitetural que descreve seus componentes e a organização interna do software. A arquitetura de software pode ser embasada em um padrão ou estilo arquitetural (SOMMERVILLE, 2010).

2.2.2 Padrões e Estilos Arquiteturais

Segundo Shaw e Clements (1996), o estilo arquitetural é determinado como um conjunto de regras de projeto que identificam os tipos de componentes e conectores que podem compor um sistema, unidos com restrições locais ou globais. Componentes podem ser diferenciados pela natureza de sua computação e pela sua interação com outro componentes. Há uma grande interação entre componentes mediadas por conectores, ocasionando em estilos diferentes com características distintas.

Como em Villamizar et al. (2015), é comum encontrar na literatura o termo de arquitetura monolítica e de microsserviços. Entende-se por monolítica, o estilo arquitetural de uma aplicação em que são construídos serviços em uma base de código singular,

sendo gerado um único executável com todas as funcionalidades. Já o estilo arquitetural de microsserviços, estas funcionalidades são separadas em bases de códigos diferentes, e gerado um executável para cada base de código. Assim, os termos de arquitetura monolítica ou arquitetura de microsserviços presentes neste trabalho referem-se aos estilos arquiteturais monolíticos e microsserviços.

Assim, os estilos arquiteturais em que são baseados os principais sistemas Web desenvolvidos por empresas atualmente são as monolíticas e microsserviços, e os sistemas que caracterizam pelo fornecimento de serviços em rede (*Web Services*). Estes estilos arquiteturais possuem diversas características, sendo comum a necessidade de migração entre elas. Nas próximas seções serão descritos conceitos e arquiteturas que foram utilizadas durante a fase de projeto e desenvolvimento deste trabalho.

2.3 Estilo Arquitetural Monolítico

Por muito tempo, a arquitetura monolítica foi utilizada como um padrão de desenvolvimento de aplicações para Internet, sendo construídas por equipes de desenvolvimento encarregados de trabalhar neste padrão centralizado, sujeito a mudanças e de crescimento rápido. Com o passar do tempo, as aplicações monolíticas podem ser sujeitas a diferentes técnicas de desenvolvimento, deixando a aplicação mais complexa e difícil de manter, aumentando os custos de escalabilidade da aplicação (ACEVEDO; JORGE; PATIÑO, 2017).

Os sistemas desenvolvidos utilizando a arquitetura monolítica seguem o padrão de camada única (*single-tier*), ou seja, utilizando uma única base de código para construir suas aplicações e dividindo-a entre todos os desenvolvedores. A Figura 2.1 apresenta uma ilustração desta arquitetura. Com a necessidade de realizar alterações ou adição de funcionalidades ao sistema, os desenvolvedores precisam garantir o correto funcionamento dos outros serviços associados a esta aplicação. Ao final de qualquer alteração, correção, adição de novas funcionalidades, deve-se compilar e implantar a aplicação completa, sendo necessário reiniciar todos os serviços fornecidos, deixando uma experiência ruim para os usuários que utilizavam o serviço. A complexidade da aplicação cresce à medida em que novos serviços são adicionados à aplicação monolítica, limitando a velocidade e desem-

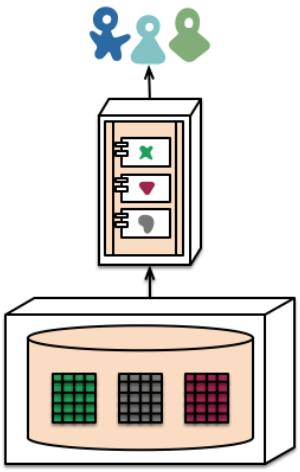


Figura 2.1: Arquitetura monolítica. Adaptada de (FOWLER; LEWIS, 2018)

penho das equipes que realizam manutenção e desenvolvem novas versões da aplicação. Esta arquitetura possui um problema crucial: se ocorrer uma falha na aplicação, todos os serviços da aplicação são afetados por ela (VILLAMIZAR et al., 2015).

A arquitetura monolítica possui diversas características que auxiliam no desempenho e praticidade da aplicação. Porém diante de suas limitações em um sistema de software, torna-se relevante o estudo de outra arquitetura, que surgiu com a proposta de resolver estes problemas. Na próxima seção, está descrita a arquitetura de microsserviços.

2.4 Estilo Arquitetural de Microsserviços

Nos dias atuais, com a grande disputa no mercado de tecnologia, todos os negócios giram em torno de *softwares* e a velocidade de mudança no mercado torna-se um fato determinante. Muitas empresas procuram por formas de desenvolvimento menores e que rapidamente possam sair da ideia para o produto final em produção. Estes "pequenos desenvolvimentos"demandam poucas pessoas para finalizá-lo e são fáceis de lançarem versões, sendo inseridos em produção de forma independente (UEDA; NAKAIKE; OHARA, 2016). Com esta proposta surgem os microsserviços.

Esta arquitetura não foi criada como uma formulação antes de entrar para o mercado de software. Ela foi originada como um padrão de desenvolvimento de sistemas resultante das tendências anteriores no mundo da arquitetura e desenvolvimento de software. Esta arquitetura foi originada do modelo de desenvolvimento dirigido a domínio, da

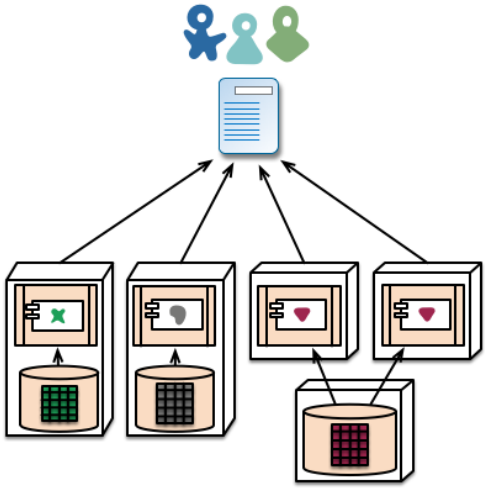


Figura 2.2: Arquitetura de microsserviços. Adaptada de (FOWLER; LEWIS, 2018)

entrega contínua e em demanda. Algumas tecnologias de virtualização já permitem escalar e automatizar os sistemas implementados na arquitetura microsserviços (NEWMAN, 2015).

O estilo arquitetural de microsserviços é formada por um conjunto de serviços menores, cada qual em seu processo. A Figura 2.2 ilustra esta arquitetura, mostrando os serviços como uma camada que liga os usuários às bases de dados. Comumente, nesta arquitetura são usadas formas leves de comunicação (*SOAP* ou *REST*). Desenvolvendo sistemas nesta arquitetura como um conjunto de serviços independentes obtém-se aumento de produtividade, escalabilidade e de facilidade de manutenção. Mas por outro lado, dificulta os desenvolvedores terem uma perspectiva geral do sistema (GRANCHELLI et al., 2017).

Segundo Hasselbring e Steinacker (2017), a arquitetura de microsserviços surgiu para solucionar as insuficiências da arquitetura monolítica referentes ao desenvolvimento de sistemas como uma única unidade, não havendo a separação das responsabilidades no sistema. Assim, dentre diversas características, as mais relevantes são:

- Decomposição vertical dos serviços: Definir a granularidade de cada microsserviço é fundamental para facilitar a execução do serviço, torna-lo adaptativo e facilitar a evolução e correção do código desenvolvido;

- Perda de acoplamento: Ao descentralizar o gerenciamento de dados por meio de microsserviços, há consequências relacionadas à gerência de atualizações. Pode-se

utilizar transações para ajudar com a consistência dos dados, mas ainda pressupõe alguma acoplagem, transformando-se em um problema através dos serviços. Esta abordagem é usada na arquitetura monolítica;

- Escalabilidade e tolerância a falha: Os atributos de escalabilidade e tolerância à falha são dirigidos para a arquitetura de microsserviços. As aplicações em microsserviços devem ser projetadas para tolerar a falha de serviços individuais. Como os serviços podem falhar a qualquer momento, é essencial detectar as falhas rapidamente e restabelecer os serviços rapidamente. Estas aplicações possuem grande importância no processamento e monitoramento em tempo real da aplicação, verificando suas questões técnicas e métricas de negócios. Assim, as aplicações na arquitetura de microsserviços podem replicar dinamicamente os microsserviços quando a infraestrutura em nuvem está submetida a grande carga, não sendo necessário escalar o sistema completo, como seria em um sistema monolítico;

- Rápida implantação: Utilizar a tecnologia de conteinerização, como o *Docker*, acelera o processo de desenvolvimento e implantação pelos desenvolvedores nas instâncias dos serviços e com menor sobrecarga do que a virtualização feita sobre o sistema operacional;

- Desenvolvimento Escalável: É possível dispor de capacidades de desenvolvimento de acordo com a mudança de requisitos, quando a estrutura é decomposta verticalmente em microsserviços. A arquitetura de microsserviços enfatiza a estrutura modular, que é importante para grandes equipes de desenvolvimento. No entanto, assim como nos microsserviços, desacoplar a equipe é essencial para um bom resultado e ganho de desempenho no desenvolvimento de cada serviço.

A arquitetura de microsserviços em conjunto com ambientes em nuvem para desenvolver e implantar aplicações, e tecnologias da Web que disponibilizam serviços na Internet, compõem o cenário das novas tendências do desenvolvimento de software. A aplicação agora se torna um conjunto de serviços, que podem ser acessados pelos usuários através da Internet (GUO et al., 2016).

2.5 Web Services

Nos últimos anos, desenvolver aplicações no lado servidor tornou-se um grande desafio, tendo em vista a compatibilidade entre dispositivos móveis e computadores. Para isso, foram desenvolvidas arquiteturas com o objetivo de fornecer uma interface comum entre todos os tipos de aplicações, de forma a utilizar os serviços já criados e facilitar a comunicação entre eles.

Um *Web Service* é uma aplicação modular autônoma, que pode ser descrita, publicada, localizada e invocada através da rede *World Wide Web* e executa funções encapsuladas que variam desde requisições e respostas, até o processo de negócio. Ele fornece de forma padronizada, independente de plataforma e amplamente aceita, formas de descrever componentes e composição, mesmo para processos de negócios distribuídos. Assim, processos de negócios distribuídos se tornam uma composição de *Web Services* (MARTENS, 2003).

A arquitetura de *Web Service* é utilizada na integração e comunicação de aplicações. Nesta arquitetura as aplicações podem se comunicar com as que já foram desenvolvidas e aplicações de plataformas diferentes. Os *Web Services* são como componentes, que permitem as aplicações de se comunicarem através de requisições e, geralmente, elas utilizam o formato *XML*[4] (*eXtensible Markup Language*) e *JSON* [5] (*Java Script Object Notation*) para se comunicar com os *Web Services*. Um *Web Service* pode assumir o papel de cliente, que faz solicitação a um outro *Web Service* para realizar uma determinada tarefa, ou de um servidor, que executa a tarefa e retorna o que foi solicitado. Por isso, não se encaixam bem na arquitetura de cliente-servidor.

2.6 Virtualização com Docker

Um dos desafios que os profissionais das equipes de desenvolvimento de aplicações enfrentam todos os dias em suas empresas, é o teste de aplicações em ambientes mais semelhantes possível ao ambiente de produção. Quando se testa aplicações em ambientes diferentes aos ambientes de produção, ao se implantar as aplicações nos ambientes de produção

[4]https://www.w3.org/XML/
[5]http://www.json.org

efetivamente, podem ocorrer erros inesperados. Isso pode atrasar o cronograma e impedir o correto funcionamento da aplicação no ambiente desejado. Com isso, torna-se necessária a utilização de ambientes com características o mais semelhantes ao ambiente em que a aplicação será implantada. Assim dispõe-se de uma tecnologia de virtualização e conteinerização amplamente utilizada nos dias atuais: o *Docker*.

2.6.1 Docker

Docker é uma plataforma de virtualização de contêineres. Ele utiliza a tecnologia de virtualização que fornece controle de recursos e isolamento de processos, sistema e interface de rede. O isolamento permite que os desenvolvedores possam fazer suposições sobre o estado do sistema e suas configurações, e obter as dependências necessárias para o correto funcionamento dele. O controle de recursos permite a configuração da quantidade de *CPU*, memória e acesso a rede da qual uma aplicação precisa. A virtualização também utiliza imagens. Essas imagens são arquivos com as configurações do sistema necessárias para construir o contêiner e executar a aplicação. Um contêiner é um compartimento (pacote) executável e leve, permitindo que as aplicações sejam executadas com isolamento e controle de recursos (*CPU*, memória e disco rígido), reduzindo a sobrecarga em comparação com a máquina virtual (SANTOS et al., 2018). A Figura 2.3 ilustra os contêineres e algumas das tecnologias que são utilizadas nos contêineres do *Docker*.

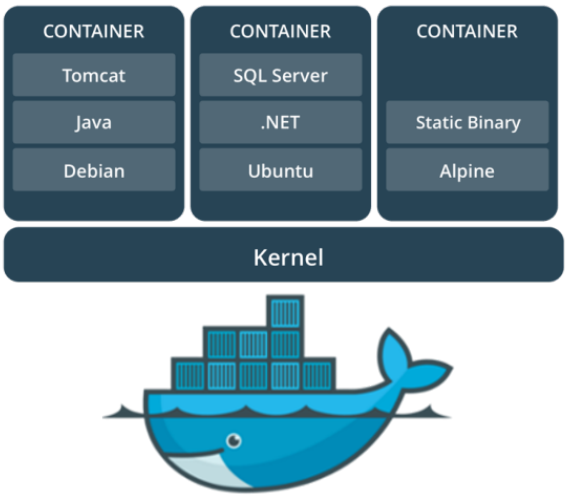


Figura 2.3: Contêineres criados no *Docker*. Adaptada de (DOCKER. . . , 2017)

2.6.2 Docker e Máquina Virtual

No mercado atual, existem duas técnicas de virtualização: a baseada em *hardware* e a baseada em software. Na técnica baseada em hardware, tem-se a Máquina Virtual (VM), que pode ser instalada no mesmo computador, fornecendo um ambiente para hospedagem de sistemas operacionais e isolamento entre outras Máquinas Virtuais instaladas no mesmo computador. As VMs compõem a parte principal da camada de infraestrutura de serviços em nuvem, como Paas (*Platform-as-a-Service*) (MAVRIDIS; KARATZA, 2017). As tecnologias de conteinerização possuem a mesma proposta, porém possuem arquiteturas e implementações. Os contêineres são de grande importância na computação em nuvem e, dentro de suas principais características, estão a arquitetura leve e a redução da sobrecarga (CHUNG et al., 2016).

A Figura 2.4 mostra um esquema de camadas da máquina virtual e o *Docker*. O *Docker* faz uso dos recursos do sistema operacional em que foi instalado. Já a máquina virtual faz uso de um *Hypervisor*. Um *Hypervisor* é uma camada localizada entre o software e o *hardware*, fornecendo ao sistema operacional hóspede uma abstração da máquina virtual, intermediando o controle do sistema operacional ao *hardware* do computador (MERKEL, 2014).

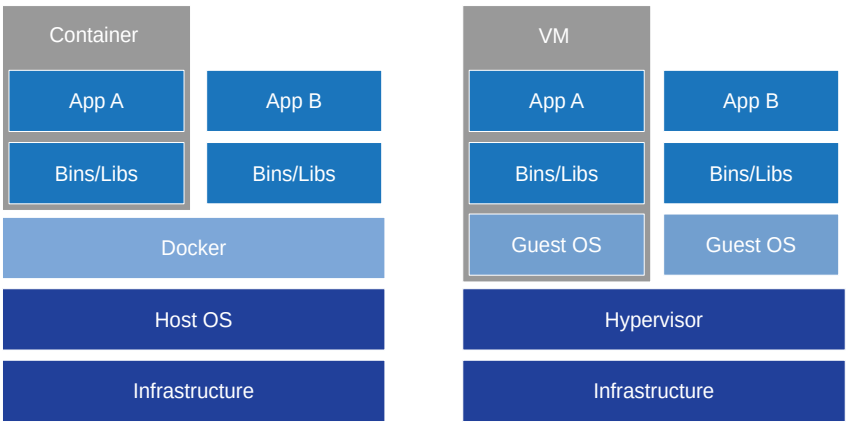


Figura 2.4: Esquema de camadas da *Virtual Machine* e *Docker*. Adaptada de (JOY, 2015)

2.7 Rest

No que se trata de *Web Services*, uma das arquiteturas mais usadas nas empresas é a arquitetura SOA *(Service-Oriented Architecture)*, que traz como proposta que as aplicações sejam compostas por um conjunto de serviços, deixando assim a aplicação mais flexível e reutilizável. Sendo assim, as aplicações baseadas nesta arquitetura, trazem um conjunto de padrões que permitem a criação de protocolos, como SOAP e WSDL (*Web Services Description Language*), conduzindo à interoperabilidade da aplicação (SOUZA et al., 2013).

Outra proposta de arquitetura, que também é utilizada nas empresas, é o REST (*Representational State Transfer*). Foi Roy Fielding quem idealizou REST como arquitetura para sistemas distribuídos, mostrando princípios da Engenharia de Software e restrições de interação na sua dissertação (HAN et al., 2009). A arquitetura REST é orientada a recursos, e é chamada ROA (*Resource-Oriented Architecture*). Traz um conjunto de características que permitem o acesso simples e sem estado aos recursos fornecidos na rede. Um serviço que traz os conceitos mostrados na arquitetura REST são chamados RESTful e fazem uso de quatro métodos-chave do protocolo HTTP (*GET, PUT, POST e DELETE*). Assim, a arquitetura ROA é a concretização da arquitetura RESTful, que transforma um problema em um *Web Service* RESTful (SOUZA et al., 2013).

Este estilo arquitetural traz por definição uma interface para componentes de sistemas baseados em quatro restrições: identificação de recursos, manipulação de recursos através de representações, mensagens auto-descritas e *Hypermedia as the Engine of Application State* (HATEOAS). Este último (HATEOAS), define que a interação com o cliente precisa ser orientada através de controles de hipermídia, os quais fornecem mecanismos de consulta, armazenamento e manipulação de informações. Este princípio implica que a representação do recurso fornecerá também controles de hipermídia para orientar as interações com cliente, informando as operações que podem ser executadas sob os recursos e fornecendo links de outros recursos disponíveis no servidor (SALVADORI; SIQUEIRA, 2015).

Através do Modelo de Maturidade de Richardson Webber, Parastatidis e Robinson (2010) são descritas as APIs Web em quatro níveis seguindo seu projeto. No primeiro

nível estão os serviços que possuem uma URI única e um único método HTTP (como POST), utilizando o HTTP como protocolo de transferência de dados, sem princípios REST nem semântica dos métodos HTTP. No segundo nível são fornecidos dados mapeados através de uma API e modelados em múltiplos recursos, com cada recurso sendo identificado e abordado de forma singular, deixando o cliente manipular individualmente cada recurso. No terceiro nível é necessário o uso correto de mensagens HTTP para lidar com recursos adequadamente, ou seja, utilizar POST para criar, PUT para modificar, GET para obter e DELETE para remover um determinado recurso. Os códigos de status HTTP devem ser usados nas mensagens de resposta para descrever com consistência o resultado das operações realizadas nos recursos. No quarto nível é necessária a adesão ao princípio HATEOAS através da provisão de controles hipermídia pela API da Web. Ela permite aos clientes da API explorar e navegar através de recursos não sendo fortemente acoplados às características internas da implementação (SALVADORI; SIQUEIRA, 2015). Uma das limitações da arquitetura REST é que a comunicação segue o padrão de pedido-resposta, que se inicia pela requisição do cliente (STIRBU, 2010).

Geralmente os *Web Services* desenvolvidos no protocolo SOAP possuem desempenho inferior em comparação aos desenvolvidos com RESTful, e ainda são muito utilizados em processos distribuídos, principalmente quando eles requerem confiabilidade e segurança. A segurança de um *Web Service* RESTful é feita em nível do protocolo de comunicação utilizado, como exemplo do HTTPS para criptografia dos dados com autenticação (SOUZA et al., 2013).

2.8 Tecnologias de microsserviços

Com a rápida evolução das tecnologias no desenvolvimento de software, e sua demanda para construir e operar serviços em escala, atingir os requisitos de negócio, suportar as solicitações feitas pelo cliente e utilização dos recursos para redução da sobrecarga e custos, a lógica das aplicações está sendo reformulada para uma arquitetura descentralizada, independente e ágil. Ao contrário da implementação na arquitetura monolítica, a de microsserviços utiliza melhor os recursos de infraestrutura, realocação de recursos e serviços para as determinadas tarefas de negócio.

2.8.1 Padrão Jax-RS

No mercado atualmente há diversas ferramentes e *frameworks* sendo utilizados na construção de *Web Services* na arquitetura *RESTful*. Estão sendo produzidos desenvolvimentos sobre a padronização de diversos *frameworks*, fornecendo interfaces unificadas com uma grande diversidade de implementações.

O padrão industrial para desenvolvimento portátil foi, durante muito tempo, o Java EE. Uma tecnologia escalável, robusta, portátil e segura no lado servidor. O padrão JAX-RS é uma parte da plataforma Java EE, o qual garante portabilidade ao código implementado utilizando *REST* através dos servidores de aplicação Java EE. Há diversas implementações deste padrão no mercado de hoje (PURUSHOTHAMAN, 2015). Algumas das implementações mais populares são (PURUSHOTHAMAN, 2015):

- Jersey RESTful WS Framework [6]: *Framework* de código fonte aberto para desenvolvimento de *Web Services* em Java, com a referencia de JAX-RS;
- Apache CXF [7]: *Framework* de código fonte aberto, baseado no projeto *JBoss*, para desenvolvimento de *Web Services* em Java, com a referencia de JAX-RS e JAX-WS;
- RESTEasy [8]: *Framework* de código fonte aberto, baseado também no projeto *JBoss*, para desenvolvimento de *Web Services RESTful*;
- Restlet [9]: *Framework* leve, de código fonte aberto, possui bom suporte para desenvolvimento de *Web Services RESTful* e clientes *RESTful*. Também suporta plataformas móveis.

O objetivo principal da especificação JAX-RS é facilitar o desenvolvimento de *Web Services RESTful*, e sua utilização na implementação de novas tecnologias e *frameworks*. Ele oferece anotações para retirar informações das solicitações recebidas do cliente, fornecendo suporte para os métodos e parâmetros de solicitações: *Query*, *URI path*, *Form*, *Cookie*, *Header* e *Matrix*. Estes parâmetros são usados em conjunto com os recursos fundamentais do *REST*, os métodos *GET*, *POST*, *PUT* e *DELETE*. Estes métodos

[6] https://jersey.github.io/
[7] http://cxf.apache.org/
[8] https://resteasy.github.io/
[9] https://restlet.com/

são os elementos fundamentais de qualquer *Web Service RESTful* (PURUSHOTHAMAN, 2015).

2.8.2 Jersey

Uma das várias implementações da especificação *JAX-RS* utilizadas no mercado atual é o *framework Jersey*. Este *framework* disponibiliza ainda propriedades e utilidades adicionais, que não estão presentes na especificação *JAX-RS*, para simplificar ainda mais o desenvolvimento de serviços e clientes utilizando *REST*. Ele possibilita ao desenvolvedor definir e configurar componentes *JAX-RS* durante a implantação do sistema (PURUSHOTHAMAN, 2015).

Segundo Purushothaman (2015), este *framework*, em sua implementação na versão *JAX-RS* 2.0, apresenta suporte para o controle de ligações de hipermídia com recursos, fundamentados na restrição *REST HATEOAS*, permitindo o cliente navegar dinamicamente aos recursos desejados utilizando ligações de hipermídia. Um dos principais benefícios destes recursos de hipermídia no desenvolvimento de *Web Services* é o fato de que o estado, no contexto da aplicação, é conduzido pelo servidor, ou seja, as ligações de hipermídia retornadas pelo servidor definem o estado da aplicação. Na construção de *Web Services* com as restrições *HATEOAS* devem ser incluídas, em suas respostas, todas as possíveis ligações de hipermídia.

Este *framework* oferece ainda, anotações para as ligações declarativas, geradas em tempo de execução a partir da representação dos recursos. Como cada aplicação desenvolvida é otimizada de forma diferente umas das outras, *Jersey* oferece diversas propriedades para otimizar em tempo de execução o desempenho dos *Web Services* criados com recursos *REST* (PURUSHOTHAMAN, 2015).

Ao utilizar dos recursos do *Jersey* diretamente no desenvolvimento, a aplicação poderá ficar presa na implementação *JAX-RS*. Na escolha de uma implementação para utilizar na aplicação, deve-se considerar todos os aspectos e características de forma a definir a tecnologia que fornece os recursos necessários e de melhor desempenho para a aplicação.

2.8.3 Spring Boot

O *framework Spring* ao longo do tempo se tornou a tecnologia de desenvolvimento de aplicações em *Java*. Este traz em seu núcleo o conceito de injeção de dependência. Os *frameworks* que fazem injeção de dependência, assim como o *Spring*, possibilitam o fácil gerenciamento de projetos grandes expondo o relacionamento entre objetos dentro da aplicação através de convenções e anotações, ao invés dos objetos possuírem alto acoplamento entre eles. O *Spring* se situa como mediador entre as diferentes classes da aplicação e gerencia as dependências do projeto. Devido à ampla migração de aplicações da arquitetura monolítica para microsserviços pelas empresas, a equipe de desenvolvimento do *Spring* lançou dois projetos : *Spring Boot* e *Spring Cloud* (CARNELL, 2017).

Spring Boot apresenta um novo paradigma de desenvolvimento de aplicações *Spring*. Utilizando o *framework Spring Boot*, os desenvolvedores podem criar aplicações com maior agilidade e desenvolver as funcionalidades necessárias com a menor preocupação com as configurações do *Spring*. *Spring* iniciou como uma alternativa leve para o desenvolvimento de aplicações para *Java*. Em sua nova versão, foram introduzidas escaneamento de componentes baseado em notações, os quais eliminaram grande parte das configurações *XML* explícitas. Algumas das principais características do *framework Spring Boot* são (GUTIERREZ, 2016):

- Facilidade de rapidez para criação de aplicações;
- Auto-Configuração;
- Integração de dependências automaticamente;
- Gerenciamento de configurações eficaz;
- Suporte em contêiner para *Servlet* embutido;
- Inicialização da aplicação através de uma classe *SpringApplication*;

Segundo Carnell (2017), o *Spring Boot* é previsão do *framework Spring*. Mantendo as características principais do *Spring*, o *Spring Boot* se distancia de características "empresariais"já encontradas no *Spring*, expondo um *framework* voltado para o padrão

Java, com microsserviços orientados ao *REST*. Este *framework* é a tecnologia principal utilizada nas implementações de microsserviços. Ele simplifica o desenvolvimento de microsserviços, reduzindo as tarefas de construção dos microsserviços baseados em *REST*. O *Spring Boot* também simplifica o mapeamento dos verbos estilo *HTTP* (*GET, PUT, POST e DELETE*) para *URL*s e serialização com o protocolo *JSON* no envio e recebimento de objetos *Java*.

O *framework Spring Boot* possui um servidor de aplicação Web embutido (*Web Server*), podendo ser configurado e, ao executar a aplicação, ele é iniciado automaticamente, implantando e realiza a inicialização da aplicação com suas respectivas configurações. O servidor de aplicação nativo ao *Spring Boot* é o *Apache Tomcat*. Mas pode ser substituido por outros servidores de aplicação diferentes, tais como *Jetty* e *Undertow* (WEBB, 2017).

2.9 Teste de Desempenho

Um dos principais desafios enfrentados por empresas nos tempos atuais é otimizar e resolver problemas de desempenho de seus produtos. Outros problemas recorrentes são lacunas ou omissões de requisitos de natureza não funcional, como desempenho, segurança, usabilidade e escalabilidade. Assim é necessário efetuar testes de desempenho e revisões de código para encontrar possíveis perdas de desempenho, defeitos e simular a execução da aplicação (REPASI, 2009).

Segundo Pressman e Maxim (2016), os testes de desempenho são projetados para simular condições reais de carga e descobrir problemas de desempenho na aplicação, como falta de recursos, problemas na rede, banco de dados, sistema operacional, funcionalidades da aplicação com problemas, entre outros; podem levar a uma queda de desempenho na aplicação. Dessa forma, pode-se analisar a aplicação sob dois aspectos: observar a resposta do sistema ao ocorrer aumento de carga e escolher métricas que apontem para os problemas de desempenho. Com isso, são dois os testes de desempenho direcionados para avaliação da aplicação. São eles: teste de carga e de estresse. Cada teste está descrito nas seções posteriores.

2.9.1 Teste de Carga

Os testes de carga e estabilidade tem o objetivo de garantir que o sistema se mantêm estável por longo tempo, submetido a carga máxima. Um sistema poderá permanecer sem falhas quando testado aprimoradamente por analistas experientes, que executam testes precisos em suas funcionalidades. No entanto, com o grande aumento de usuários utilizando sistemas que executam por meses ou anos, sem parar ou reiniciar, alguns problemas comuns podem ocorrer: o sistema apresenta lentidão, problemas de funcionalidade, falha ou/e trava completamente. Estes testes geralmente avaliam o sistema com usuários simulados e medir o seu desempenho, de forma a verificar se o sistema pode ou não suportar a grandes cargas de trabalho. Com isso, os desenvolvedores podem ter uma visão prévia do comportamento do sistema ao ser submetido as situações reais de uso. Diante de um teste de carga e estabilidade, pode-se garantir que o sistema poderá operar em larga escala durante meses (NAIK; TRIPATHY, 2011).

2.9.2 Teste de Estresse

O Teste de Estresse possui a finalidade de avaliar o comportamento de uma funcionalidade do sistema quando submetida a uma carga excessiva, além de sua capacidade atual. Isso inclui testar em ambientes com recursos insuficientes e com incompatibilidades do sistema. Assim, pode-se garantir que o sistema se comportará de forma satisfatória nas piores condições, até mesmo sob um pico de carga. Ao exceder seu limite, o sistema pode falhar, e então, um sistema de recuperação poderá ser acionado para restaurar o funcionamento do sistema (NAIK; TRIPATHY, 2011).

Segundo Naik e Tripathy (2011), o teste de estresse revela problemas no sistema como vazamento e alocação de memória. Desta forma, este teste propõe alcançar os limites de todas as características do sistema, e avaliar os limites de seu tempo de resposta e vazão. O melhor método de identificar gargalos no sistema, é executar o teste de estresse, com funcionalidades internas e externas do sistema.

Testar cada funcionalidade implica em submetê-la à sua capacidade de carga absoluta, e depois cada funcionalidade do sistema é testada além de sua capacidade máxima, até testar o sistema completo. Durante o teste, a carga poderá ser acrescentada até o

sistema falhar, e quando o sistema falhar, observar seu comportamento e a causa de sua falha. Uma finalidade deste teste é aperfeiçoar a robustez do sistema e desenvolver métodos para a recuperação do sistema ao falhar (NAIK; TRIPATHY, 2011).

Com as devidas definições realizadas, faz-se necessária a escolha de ferramentas que possam realizar os testes de forma a medir o software para avaliar e simular o impacto em um ambiente de produção real.

2.10 Ferramenta de Teste

Para realizar os teste de desempenho, é preciso definir ferramentas capazes de atribuir e executar cada teste ao sistema, de forma a gerar resultados que meçam os atributos do sistema. Dessa forma, dentre outros *frameworks* do mercado atual, o *JMeter* foi selecionado para realizar os testes devido à sua popularidade e praticidade em efetuar testes de desempenho.

O *Apache JMeter*[10] é um software de código fonte aberto, desenvolvido para efetuar testes de carga e de desempenho nas aplicações de diferentes tecnologias. Foi desenvolvido usando a linguagem de programação *Java* e seu primeiro objetivo era testar aplicações Web. Este software é amplamente utilizado para medir o desempenho das aplicações Web através de recursos estáticos e dinâmicos, sendo que os dinâmicos podem ser gerados a partir das respostas de algumas requisições enviadas. Ele pode ser usado ainda para teste de carga no servidor, carga de usuários em acesso no sistema, rede e analisar o desempenho sob diferentes tipos de carga. Seus testes executados pelo *Apache JMeter* são funcionais (ou caixa-preta). Este software não executa qualquer tecnologia de processamento de forma cliente, como em navegadores, trabalhando a nível de protocolo para execução dos testes (JMETER, 2018).

2.11 Considerações Parciais

Durante a etapa de levantamento de material e estudo para o determinado trabalho, foram verificadas algumas das características relacionadas aos *frameworks* que serão avaliadas

[10]https://jmeter.apache.org/

neste trabalho. Foram selecionados atributos para comparação entre os *frameworks*, como o tempo que levou para desenvolver cada microsserviço nos respectivos *frameworks*, a facilidade com que se configuram os recursos de cada *framework*, a facilidade de desenvolvimento da aplicação (desde criar o projeto até a aplicação estar executando), facilidade de implantação das aplicações nos contêineres em conjunto com as tecnologias de base, a disponibilização das métricas por cada *framework* (métricas disponibilizadas nativamente ao *framework*), os *frameworks* possuírem servidor nativo presente na configuração de suas dependências, os *frameworks* possuírem flexibilização de implantação em outros servidores, a possibilidade de configurar suas dependências e a segurança de cada *framework* em relação as configurações de base.

A Tabela 2.1 mostra algumas das características que foram observadas durante o levantamento teórico e durante o desenvolvimento dos serviços com os respectivos *frameworks*. As representações utilizadas são de (+) para indicar quantidade positiva (como no tempo de desenvolvimento, indica maior tempo de desenvolvimento), (-) para indicar quantidade negativa, (S) para indicar que está presente no *framework* e (N) para indicar que não está presente no *framework*.

Atributos	Spring Boot	Jesey
Tempo de desenvolvimento	-	+
Facilidade de configuração	+	-
Facilidade de desenvolvimento	+	-
Facilidade de implantação	+	+
Disponibilidade de métricas	+	-
Servidor embutido	S	N
Flexibilidade de servidores	S	S
Configuração de dependências	S	S
Segurança	S	S

Tabela 2.1: Comparação de características entre as tecnologias avaliadas.

Estas características podem ser utilizadas como auxilio na tomada de decisão e suporte para a análise sobre os *frameworks* abordados. Com a mudança na arquitetura dos serviços, muda a forma de analisar e testar as aplicações. Pode-se ter equipes especializadas em funcionalidades de negócio, que passarão a tratar o serviço não como somente um componente, mas como um produto com um ciclo de vida independente, escalável e mais próximo do negócio.

3 Trabalhos Relacionados

3.1 Introdução

Durante o desenvolvimento deste trabalho, foram feitas pesquisas de forma a acrescentar e definir os conceitos propostos. Algumas delas possuem aspectos importantes que serviram de base para o experimento e análise das tecnologias. A seguir são descritos trabalhos que serviram de apoio para os experimentos e desenvolvimento deste trabalho.

3.2 Análise e Descrição

No trabalho de Vandikas e Tsiatsis (2016) foram utilizadas as métricas Taxa de Transferência (*Throughput*), Quantidade de Memória Utilizada (Memory Footprint) e Tamanho do Executável (Binary Footprint) para avaliação de desempenho de *Microservice Application Servers (MASs)* em um ambiente de *Internet of Things* (*IoT*) em nuvem. Estes servidores *MAS*s são descritos como *frameworks* de arquitetura cliente/servidor que gerenciam as interações entre a lógica dos microsserviços e o toda a comunicação feita em rede, através do protocolo *TCP/IP*, sendo usada para conectar os microsserviços. Para efetuar o teste de carga nos servidores *MAS* foram criados clientes (C_1...C_n) para simularem as cargas das requisições feitas por aparelhos reais. Foi utilizado um microsserviço pequeno com o padrão de comunicação requisição-resposta, de forma a receber as requisições dos clientes processar e retornar uma resposta em *HTTP 200 OK*. Dentre todas as métricas, a de principal atenção foi a Taxa de Transferência (*Throughput*), sendo medida em diferentes níveis de concorrência (1, 25, 50, 100, 200, 400, 800, 1600, 3200, 6400 e 10000). Para cada nível de concorrência foram enviadas 1.000.000 requisições 3 vezes. Apesar de serem importantes as métricas de Quantidade de Memória Utilizada e Tamanho do Executável, as análises das tecnologias do trabalho abordado possuem o foco maior no desempenho e o comportamento do sistema ao receberem altas quantidades de requisições e seu tempo de resposta. Portanto, com os diferentes níveis de concorrência, no trabalho

proposto foram configuradas níveis de concorrência para as requisições através de classes, utilizando múltiplos valores de clientes e de requisições diferentes.

Levando em consideração as características da arquitetura de microsserviços e a importância de testar o desempenho das aplicações nesta arquitetura, no trabalho de Villamizar et al. (2015) é proposto avaliar uma aplicação implementada em duas arquiteturas distintas: uma utilizando a arquitetura monolítica, e outra a arquitetura de microsserviços. As aplicações foram desenvolvidas no contexto de fornecimento de planos de parcelamento para instituições, com o objetivo de identificar áreas no processo de desenvolvimento de software que afetam em sua arquitetura. Para isso, foram desenvolvidos dois serviços: S_1, um serviço que implementa algoritmos para geração de planos de pagamento, e S_2, que retorna um plano de pagamento com seus respectivos atributos do banco de dados. Em ambas as aplicações foram utilizadas a arquitetura *REST*, com a troca de mensagens no padrão *JSON*, fornecendo recursos a serem consumidos por uma aplicação na arquitetura *MVC (Model View Controller)*, como aplicação cliente. Para isso, ambas as aplicações foram implementadas na linguagem *Java*, com o *framework web Play*, e seu servidor de aplicação embutido *Netty*. Na aplicação cliente foi utilizada a tecnologia mais recente no mercado, o *AngularJs*, e, para hospedar suas aplicações, foi utilizada a infraestrutura do *Amazon Web Services* ou (*AWS*). Para efetuar os testes, foram criadas pequenas instâncias que suportam os requisitos de negócios de ambas as aplicações. Na analise do desempenho das arquiteturas, foi utilizado o *JMeter* nas instâncias da *AWS*, comparando o tempo de resposta de ambas as aplicações. No teste, o *JMeter* foi configurado para executar 30 solicitações por minuto ao serviço S_1, e 1.100 solicitações por minuto para o serviço S_2, durante 10 minutos. Neste trabalho foram analisados o tempo de resposta médio e a linha de 90% do tempo de resposta.

No trabalho de Villamizar et al. (2016) foi efetuado um estudo sobre como os microsserviços podem ajudar a diminuir o custo de infraestrutura em comparação com a arquitetura monolítica. Em seu trabalho, foram desenvolvidos dois serviços para gerar e consultar planos de pagamento para empréstimos de dinheiro entregues por uma instituição aos seus clientes. São eles: S_1, um serviço que implementa algoritmos para geração de planos de pagamento, e S_2, que retorna um plano de pagamento com seus respectivos

atributos do banco de dados. Os tempos de resposta de S_1 e S_2 são respectivamente 3000 e 300 milissegundos. Seus serviços foram desenvolvidos em três aplicações nas arquiteturas monolítica e microsserviços. A aplicação desenvolvida na arquitetura monolítica, é implementada em duas versões utilizando os *frameworks Jax-Rs* e *Play*, publicando ambos os serviços em rede e sendo consumidos por uma aplicação *front-end MVC*. Nas aplicações desenvolvidas na arquitetura de microsserviços, foram divididas cada uma em dois microsserviços, um serviço em cada microsserviço. Uma destas aplicações foi operada por um cliente em nuvem e desenvolvida com o *framework Play*. A outra foi operada pelo serviço *AWS Lambda* e desenvolvida com a tecnologia *Node.js*. Para avaliar os custos dos serviços, todas as aplicações foram implantadas em uma plataforma de serviços em nuvem disponibilizada pela *Amazon* (*AWS*). Para compara o desempenho e os custos de infraestrutura de cada das três arquiteturas foram definidos três cenários. O primeiro cenário (20/80) foram efetuadas 20% das solicitações para o serviço S_1 e 80% para o serviço S_2. No segundo cenário (50/50) foram efetuadas 50% de solicitações para cada serviço. E no terceiro cenário (80/20) foram efetuadas 80% das solicitações para o serviço S_1 e 20% para o serviço S_2. Para este teste foram definidos os limites máximos dos tempos de resposta de S_1 e S_2, 20.000 e 3.000 milissegundos respectivamente. Os testes foram realizados com o software *JMeter*. Os testes de estresse foram executados com o objetivo de identificar o número máximo de solicitações suportadas pela arquitetura monolítica com os *frameworks Play* e *Jax-RS*. Este valor é calculado aumentando o número de solicitações em cada cenário até a aplicação começar a gerar erros ou tempo de resposta não alcançado. O resultado é mostrado na Figura 3.1.

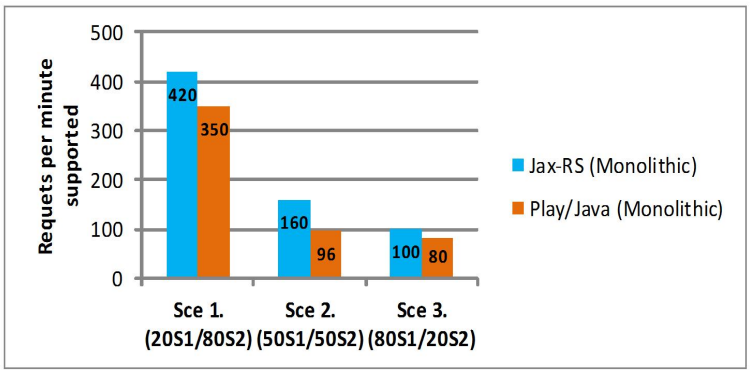


Figura 3.1: Resultado do teste de estresse. Retirado de (VILLAMIZAR et al., 2016).

No trabalho de Viggiato et al. (2018) é proposta uma pesquisa para revelar a utilização dos microsserviços em uma perspectiva prática, com o objetivo de compreender e revelar como os profissionais estão utilizando os microsserviços. Inicialmente, foi realizado um estudo de mapeamento para coletar informações sobre microsserviços em diversas fontes, com o objetivo de obter documentos sobre todos os aspectos dos microsserviços, como documentos com termos e definições gerais, e as melhores características e desafios enfrentados pelos desenvolvedores. Após analisar os documentos obtidos, foram identificados cinco artigos, um livro da literatura cientifica e quinze documentos relevantes de outras fontes, como sites e blogs. Foram lidos todos os 21 documentos relevantes de forma a extrair seus temas e tópicos relevantes, e classificá-los em vantagens e desafios encontrados por desenvolvedores ao utilizar arquiteturas de microsserviços. As vantagens encontradas foram: diversidade tecnológica, implantação independente, escalabilidade e manutenibilidade. Os desafios encontrados foram: transações distribuídas, testes de integração, falhas de serviços e Chamadas de Procedimento Remoto (RPC). Após este estudo de mapeamento, foram pesquisados os desenvolvedores sobre os resultados do estudo de mapeamento, com o objetivo de verificar se a utilização dos microsserviços na indústria está de acordo com as recomendações mencionadas nas literaturas e se as vantagens e desafios encontrados na pesquisa são os que os profissionais enfrentam.

A pesquisa possui 14 questões e foi dividida em três seções. A primeira seção é sobre o histórico dos desenvolvedores, com perguntas sobre experiência com o desenvolvimento de software e microsserviços, tamanho das aplicações e quantidade de serviços que já trabalhou. A segunda seção inclui perguntas sobre o tamanho ideal de um microsserviço, as vantagens e problemas enfrentados pelos desenvolvedores desta tecnologia. E a terceira seção é composta por perguntas abertas sobre as tecnologias utilizadas pelos desenvolvedores para quando implementam microsserviços. Para encontrar os participantes da pesquisa, foi desenvolvido um algoritmo para buscar desenvolvedores de microsserviços na comunidade do *Stack Overflow* [11]. Foram armazenados os nomes dos usuários que possuíam perguntas ou respostas com a etiqueta de microsserviços. Os resultados da pesquisa mostraram que quase 72% dos participantes tinham pelo menos cinco anos

[11] https://stackoverflow.com/

de experiência com desenvolvimento de software, cerca de 74% deles possuem mais de um ano de experiência com microsserviços. Aproximadamente 64% dos participantes são desenvolvedores *back-end* enquanto 11.5% são *DevOps*. Foram coletados dados sobre o tamanho das aplicações e 70.5% dos participantes trabalharam com sistemas monolíticos maiores do que 50 *KLOC* e 54% trabalharam com aplicações baseadas em microsserviços maiores do que 10 *KLOC*. Os resultados mostraram ainda que as linguagens de programação mais utilizadas pelos participantes são: *Java* (31%), *JavaScript* através do *Node.js* (18%), *C#* (12%) e *PHP* (8%); e os sistemas gerenciadores de banco de dados são: *Postgres* (30%), *MySQL* (25%), *MongoDB* (20%), *SQL Server* (12%), and *Oracle* (9%). Em relação às vantagens associadas aos microsserviços, em uma escala de 1 (muito importante) a 4 (não é importante), para a vantagem da implantação independente, mais de 50% dos participantes escolheram 1 ou 2, mostrando que são características relevantes quando trabalha-se com microsserviços. Os principais desafios foram: transações complexas distribuídas (32.8%) com medida 1, testar o sistema por completo (57%), falhas de serviços (49%), com medidas 1 e 2 na escala. Já as RPCs foram importantes para 13.8% dos participantes.

O trabalho de Camargo et al. (2016) apresentam uma nova abordagem para execução de testes de desempenho de forma automatizada, fornecendo um *framework* que implementa os conceitos chave da arquitetura de microsserviços. As tecnologias utilizadas no desenvolvimento foram *Java* usando o *framework Spring Boot*, servidor *Tomcat ou Jetty* e *framework Persistence*. O teste de desempenho tem por objetivo descobrir se o software está em conformidade com os requisitos não funcionais do projeto. Avaliou-se o desempenho com que cada serviço pode entregar individualmente suas requisições, e esses dados conduzem a quantidade de carga que um serviço consegue manipular. Para automatizar os testes, foi anexado uma especificação com os parâmetros de teste para cada serviço, e a aplicação foi configurada com os parâmetros quantidade de *threads*, solicitações, tempo de carregamento, duração do teste entre outros. A solução proposta possui dois conceitos chave: uma especificação, que permite que as outras aplicações possam acessar os parâmetros de teste que servirão para testar o serviço; e um mecanismo para anexar e expor a especificação. Como resultado, foram desenvolvidos: *framework*

API, que permite o desenvolvedor remover a estrutura da aplicação durante o processo de compilação; capacidades de anotações, onde o desenvolvedor define as especificações; dados dinâmicos para testes, que possibilita o uso do banco de dados para obter dados de especificação e validação; processador de anotações, certificando os tipos de retorno na aplicação; e construir recursos para validação de dados, permitindo validações em recursos específicos. Após efetuar testes para definir quais seriam os valores de *threads* e solicitações, foi obtido um valor aproximado de 10.000 solicitações. Assim, foram divididas estas solicitações em 25 *threads* enviando 400 solicitações por segundo. Os testes foram divididos entre WF (com *framework*) e NF (sem *framework*) e para a execução dos testes foi utilizado o software *JMeter*. Foram avaliados o tempo de reposta médio e a taxa de transferência entre WF e NF. O tempo de resposta médio foi de 118 segundos para WF e de 120.5 segundos para NF. A taxa de transferência média por segundo foi de 50.35 para WF e 48.91 para NF. Mesmo com os resultados alcançados não sejam conclusivos correlação ao impacto do *framework* no desempenho da aplicação, nota-se que o tempo médio de resposta reduziu na aplicação com a estrutura em relação a aplicação sem a estrutura, o que permite que seja implantado em qualquer ambiente sem a estrutura anexada.

3.3 Considerações Parciais

Diante das características dos trabalhos acima, pode-se observar métricas que são de grande importância para efetuar o teste de desempenho em aplicações. Desta forma, pode-se utilizar de instâncias crescentes para efetuar o teste com maior precisão e veracidade dos dados dos testes. Assim, pode-se observar diante destes trabalhos os aspectos comuns entre eles, que auxiliam na análise geral das tecnologias abordadas. Na Tabela 3.1 abaixo, estão relacionados os trabalho e seus aspectos e métricas.

Os testes efetuados no desenvolvimento deste trabalho possuem características inspiradas nos trabalhos, mas correlacionadas ao objetivo e as particularidades inerentes ao planejamento e ao sistema em que foi baseado desenvolvimento das funcionalidades descritas neste trabalho. No próximo capítulo são descritas algumas características de testes e seus conceitos, que foram utilizados durante as configurações e testes dos serviços.

Métricas	Vandikas e Tsiatsis (2016)	Villamizar et al. (2015)	Camargo et al. (2016)
Taxa de Transferência	X		X
Quantidade de Memória Utilizada	X		
Tempo de Resposta Médio		X	X
Linha de 90% do Tempo de Resposta		X	
Diferentes níveis de concorrência	X		
Tamanho do Executável	X		

Tabela 3.1: Métricas de Vandikas e Tsiatsis (2016), Villamizar et al. (2015) e Camargo et al. (2016).

4 Análise dos *Frameworks*

4.1 Introdução

O antigo sistema do Integra, desenvolvido pela equipe de analistas do NRC (Núcleo de Recursos Computacionais), foi projetado utilizando a arquitetura monolítica. Devido ao uso desta arquitetura, o sistema foi submetido aos mesmo problemas de escalabilidade, crescimento de código e desempenho inerentes a arquitetura monolítica. Com os avanços da tecnologia e o surgimento da arquitetura de microsserviços, foi necessário efetuar a migração do Integra para a arquitetura de microsserviços, devido a facilidade de manutenção, desenvolvimento de atualizações, a arquitetura descentralizada (em caso de falha, não afeta o sistema completo), rápida escalabilidade e desempenho do sistema. Assim, após um levantamento das principais tecnologias e *frameworks* que são utilizados na arquitetura de microsserviços, os analistas do NRC chegaram a conclusão sobre o uso dos *frameworks Spring Boot* e *Jersey*.

Com isso, é necessário realizar uma análise com foco no desempenho dos *frameworks* aplicada na arquitetura de microsserviços, selecionando um deles com base em sua viabilidade e desempenho para efetuar a migração do sistema Integra. A tecnologia de *Spring Boot* é uma tecnologia que está sendo amplamente utilizada na arquitetura de microsserviços nos dias atuais. Já a tecnologia *Jersey*, é um *framework* que foi desenvolvido seguindo os padrões *JAX-RS*, se tornando sua implementação de referência.

4.2 Configuração Teste de Carga

Para realizar os testes de forma a coletar e analisar os dados, foram geradas instâncias de teste. Esta metodologia foi inspirada no trabalho de Vandikas e Tsiatsis (2016), sendo geradas instâncias com valores de usuário e solicitações variáveis. Estas instâncias foram organizadas em três classes: Classe A, Classe B e Classe C. Na Classe A foram gerados testes com quantidades de usuários de 10 a 100, variando em intervalos de 10 usuários.

Na Classe B foram gerados testes com quantidades de usuários de 110 a 200, variando em intervalos de 10 usuários. E na Classe C foram gerados testes com quantidades de usuários de 210 a 300, variando no mesmo intervalo. Em todas as classes anteriores foi utilizada uma quantidade fixa de 50 solicitações por usuário criado no teste.

4.3 Configuração Teste de Estresse

Com base na metodologia utilizada para testar os serviços no trabalho de Villamizar et al. (2015), em que foram efetuadas solicitações em um intervalo de tempo, foram utilizadas neste trabalho, um número variável de clientes, fazendo solicitações sob um intervalo de tempo constante, definido em 120 segundos. Esta medida base visa avaliar a quantidade de solicitações foram aceitas neste tempo, e observar seu desempenho neste período. O número de clientes a fazerem requisições é incrementado em valores constantes de 3000, sendo o valor inicial com 1000. Seu limite é de 19000 usuários, porém deve-se obter um limite menor devido a limitações nas quantidades de usuários do *JMeter*. A quantidade de solicitações a serem efetuadas não possui limite explícito. Elas são efetuadas enquanto o contador de tempo do *JMeter* não alcançar seu limite estabelecido (120 segundos).

4.4 Métricas

A medição das aplicações compõe uma parte fundamental da engenharia de software. Ela mostra se uma aplicação está pronta para ser implantada, se possui qualidade suficiente, se concorda com todos os requisitos definidos para a aplicação, e ainda, se possui problemas na aplicação (FENTON; BIEMAN, 2014).

Segundo Fenton e Bieman (2014), medição é um método pelo qual são atribuídos valores para entidades do mundo real de forma a descrevê-las através de seus atributos, sob regras bem definidas. Uma entidade pode ser definida como um objeto ou um evento (tal como um plano de teste), e são distinguíveis entre si através de suas características. Logo, um atributo é uma propriedade da entidade. Assim, as métricas, no domínio da engenharia de software, são observadas como medições das características do software que englobam várias atividades, como custo e esforço para estimar modelos, conjunto de dados,

medidas e qualidade de modelos, modelos confiança, medidas de segurança, estruturas e complexidade, avaliação de maturidade, capacidade, métodos e ferramentas. Definir métricas no âmbito do teste de software é essencial para avaliar o software na direção de encontrar quais aspectos o produto de software possui maior eficiência e qualidade, e em quais necessita de ser corrigido. As métricas utilizadas neste trabalho são descritas a seguir.

4.4.1 Latência

O *framework JMeter* calcula a latência (*Latency*) da rede guardando o momento logo antes de enviar a requisição para a aplicação e logo após que recebe a resposta. A diferença entre o tempo de envio e o tempo de recebimento da resposta é o tempo que a aplicação levou para processar a requisição e responder. A medida de latência é a mais próxima possível de um navegador (JMETER, 2018).

4.4.2 Mediana

A mediana (*Median*) é um valor que divide as amostras ao meio, ou seja, um valor que divide as amostras em duas metades de igual tamanho, sendo metade delas menores que a média e metade maior do que a média (ou igual)(DEVORE, 2011). Pode ser chamada também de *50th Percentile* (JMETER, 2018).

4.4.3 Média

A média (*Arithmetic Average* ou *Mean*) é a soma dos valores de um conjunto de números, dividido pelo número de elementos do conjunto (DEVORE, 2011). No *JMeter*, a média está relacionada ao tempo de resposta médio de resposta de uma solicitação. O tempo de reposta é medido em mili-segundos (JMETER, 2018).

4.4.4 Taxa de Transferência

O *JMeter* calcula a Taxa de Transferência ou Vazão (*Throughput*) desde o inicio da primeira amostra até o final da última amostra, incluindo qualquer intervalo entre as amos-

tras. Desta forma, esta medida representa o valor de carga real do servidor (JMETER, 2018).

4.4.5 Desvio Padrão

O Desvio Padrão (*Standard Deviation*) é a medida de como os valores do conjunto de dados se espalham, ou seja, a variação ou dispersão dos dados (ROSS, 2004).

4.4.6 *90% Line (90th Percentile)*

90th Percentile um valor que possui a propriedade de ter 90% dos dados abaixo desse valor, e os outros 10%, acima (DEVORE, 2011).

4.5 Preparação do Ambiente

Para a medição adequada dos serviços, deve ser escolhido o ambiente que melhor reflete as condições de execução do serviço ao ser implantado no servidor de produção. Logo, o ambiente é preparado para considerar suas características de funcionamento, isolando de fatores externos e outros *softwares* que porventura tenham possibilidade de interferir nos resultados da execução dos testes.

Assim, para obter resultados mais próximos do ambiente de produção real do sistema, foi criada uma máquina no servidor com as mesmas configurações do servidor de produção atual que hospeda o sistema do Integra. Na preparação do ambiente, foi necessária a instalação das seguintes tecnologias:

- *Java* [12], versão 1.8.0.
- *Docker* [13], versão 18.06.1-ce.
- *MySql* [14], versão 8.0.12.
- *JMeter* [15], versão 4.0.

[12] https://www.oracle.com/technetwork/java/javase/downloads/index.html
[13] https://www.docker.com/
[14] https://www.mysql.com/
[15] https://jmeter.apache.org/

- Servidor *Apache Tomcat* [16], versão 8.0.53.

- Sistema Operacional *Linux Ubuntu*, distribuição 16.04 [17]

As tecnologias *Docker* e *JMeter* foram instaladas no sistema operacional da máquina criada no servidor para realizar os testes. O software de banco de dados *MySql* foi instalado dentro de um contêiner no *Docker*, sendo este contêiner de uso somente do *MySql*. Os softwares *Java* e *Apache Tomcat* foram instalados nos contêineres onde foram implantados os microsserviços. A Figura 4.1 ilustra como foram dispostas as tecnologias na máquina do servidor.

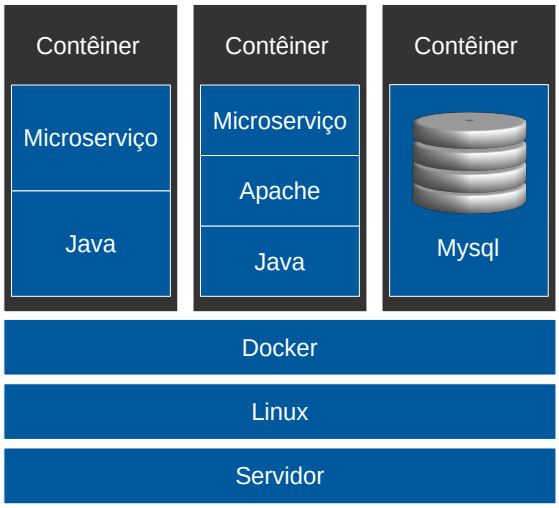


Figura 4.1: Disposição das tecnologias no servidor.

4.6 Desenvolvimento das Tecnologias

Para analisar o desempenho das tecnologias, foram desenvolvidos três microsserviços que possuem algumas das funcionalidades do sistema do Integra para conseguir resultados aplicáveis no sistema. Estes microsserviços foram implementados em ambas as tecnologias (*Spring Boot* e *Jersey*), sendo desenvolvidos em três versões: a primeira versão foi desenvolvido utilizando o *framework Jersey* (a qual é implantada em um servidor de

[16] http://www-us.apache.org/
[17] http://releases.ubuntu.com/16.04/

aplicação externo), a segunda versão utilizando o *framework Spring Boot* (a qual possui um servidor de aplicação nativo embutido dentro do serviço), e a terceira versão no *framework Spring Boot* mas com seu servidor de aplicação interno sendo retirado e implantado em um servidor de aplicação externo. O servidor de aplicação deste último foi retirado para a observação das características da aplicação quando é executada em um servidor externo, avaliando o impacto do servidor nativo no desempenho do recebimento e envio das solicitações dos usuários (clientes) do sistema.

A Figura 4.2 mostra como ficam dispostas as tecnologias em cada contêiner. O primeiro contêiner mostra como foi implantado o microsserviço que utiliza o *framework Spring Boot*, desenvolvido sem o servidor nativo embutido e implantado em conjunto com o software *Java* e o servidor de aplicação *Apache Tomcat*. Em seguida, são mostrados os contêineres implantados com o microsserviço que utiliza o *framework Spring Boot*, desenvolvido com o servidor nativo embutido e implantado em conjunto com o software *Java* (contêiner acima), e o microsserviço que utiliza o *framework Jersey*, implantado em conjunto com o software *Java* e o servidor de aplicação *Apache Tomcat* (contêiner abaixo). E ao lado, o contêiner com o software de banco de dados instalado.

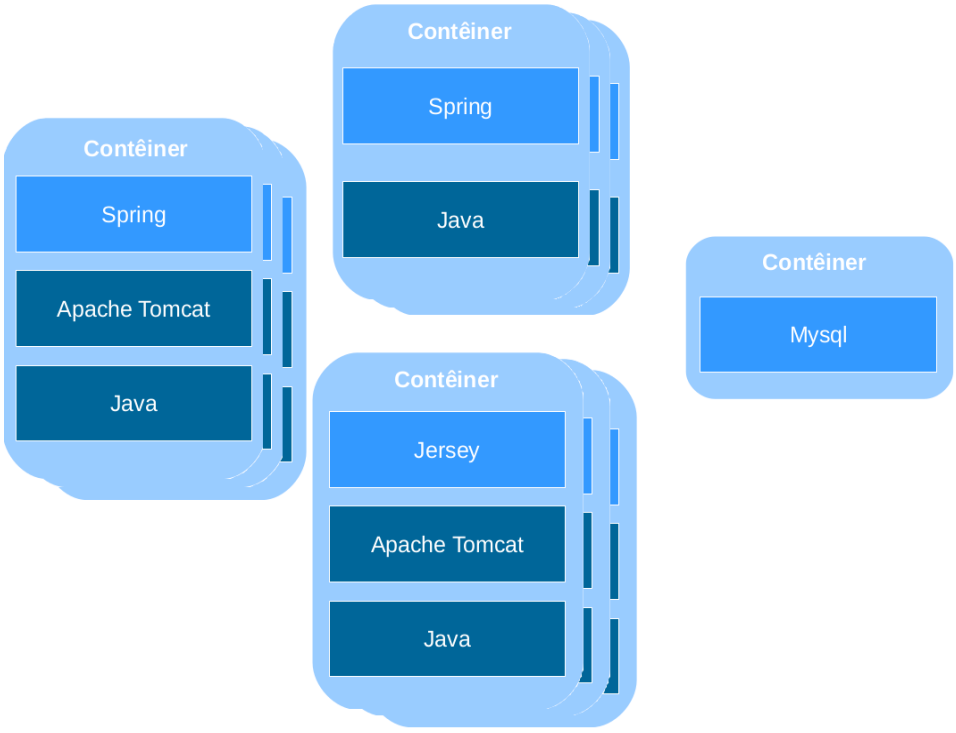


Figura 4.2: Microsserviços de usuário, página e permissão implantados nos contêineres com suas tecnologias de suporte.

Cada microsserviço foi hospedado em um contêiner *Docker* diferente, para serem analisados separadamente e sem influência entre os serviços e de memória usada por eles. Para comunicar os microsserviços com o banco de dados, foi configurada a forma de acesso ao contêiner do banco de dados reproduzindo de forma semelhante ao próprio servidor. Desta forma, todos os serviços dos microsserviços acessam os dados do banco de dados para efetuar suas operações definidas. Seu acesso se dá através de uma rede de dados local, denominada rede ponte (*bridge network*). A rede ponte é um dispositivo da camada de link que encaminha o tráfego entre segmentos da rede. Esta rede permite que contêineres conectados à mesma rede possam se comunicar e, ao mesmo tempo, fornecer isolamento entre os contêineres que não estão conectados a essa rede. Por meio desta rede são transmitidas as solicitações feitas pelos microsserviços ao banco de dados. A Figura 4.3 ilustra como os contêineres se comunicam utilizando uma rede ponte, simulando a transmissão e recebimento de dados.

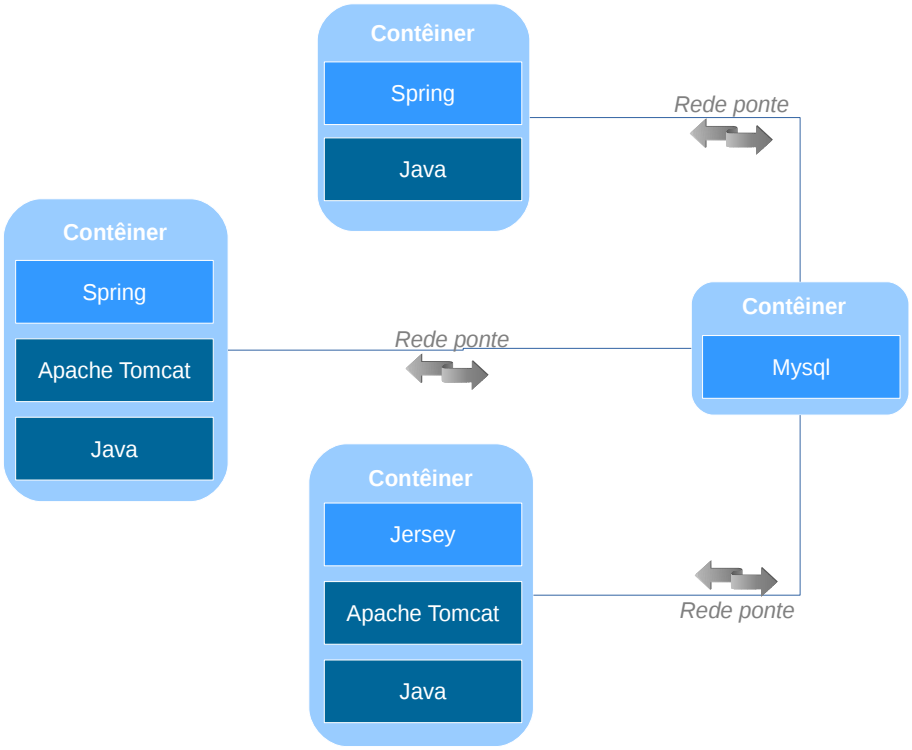


Figura 4.3: Comunicação entre as tecnologias e o banco de dados.

Para a construção de cada contêiner, são criadas e configuradas todas as imagens antes da execução e criação dos contêineres. Após as imagens serem configuradas, é criado o contêiner do banco de dados (*Mysql*) e em seguida selecionada uma imagem dentre as três (de cada tecnologia) para criar o contêiner respectivo. Assim, são mantidos somente

dois contêineres em memória: o contêiner do banco de dados e um contêiner da tecnologia escolhida neste momento. Após, são efetuados os testes neste contêiner e em seguida é apagado, sendo mantida sua imagem de referência em memória. Então é escolhida outra tecnologia para ser criado o contêiner a partir de sua imagem e testado, até terminarem as tecnologias. Por fim, são recolhidos os resultados dos testes nos arquivos gerados pelo *JMeter*, com as respectivas métricas definidas.

Os microsserviços foram desenvolvidos no contexto de gerência de usuários e seus acessos a páginas do sistema do Integra, fornecendo recursos de acesso, navegação e autenticação para os usuário que acessarem o sistema. Assim, foram desenvolvidos três microsserviços, de acordo com os diferentes contextos que envolvem a gerência de acessos aos usuários: um microsserviço para gerência de usuários, onde são fornecidos os recursos para buscar todos os usuário cadastrados no sistema, cadastrar, atualizar, remover e autenticar o usuário que acessará o sistema; um microsserviço para gerência de página, onde são fornecidos os recursos para buscar todas as páginas do sistema, cadastrar, atualizar e remover página; e um microsserviço para gerência de permissão, onde são fornecidos os recursos para buscar todos os tipos de permissão que um usuário possui no sistema, cadastrar, atualizar e remover uma determinada permissão de acesso ao sistema.

Cada microsserviço foi desenvolvido com as mesmas implementações internas dos serviços e disponibilização de recursos. As formas de acesso às funcionalidades dos microsserviços foram através de *URL*s, possuindo o mesmo padrão de chamada das funcionalidades de *GET*, *POST*, *PUT* e *DELETE*, sendo cada uma em seu respectivo microsserviço.

4.7 Infraestrutura

Os testes foram efetuados no servidor que hospeda o sistema do Integra. Neste servidor foi criada uma máquina com as mesmas configurações que são utilizadas na máquina que hospeda o Integra, para a execução dos testes. Nesta máquina, foram configurados os *softwares* necessários para a respectiva execução dos microsserviços com as tecnologias selecionadas. As configurações da máquina onde foi instalada são:

- Processador: 4 *CPU*s de 2 núcleos
- Memória em disco: 50 *GB*
- Memória *RAM*: 8 *GB*

4.8 Análise das Distinções entre as Tecnologias

Durante o estudo e desenvolvimento dos microsserviços, foram observadas as diferenças entre os *frameworks Jersey* e *Spring Boot* sob o aspecto de facilidade e rapidez de desenvolvimento e implantação. Mesmo sendo ambos *frameworks* voltados para desenvolvimento de aplicações para Web, ambos possuem diferenças, não somente em suas características de codificação, mas também em suas formas de configuração, implantação e execução.

4.8.1 Codificação

Em todos os microsserviços, as solicitações foram tratadas através de um filtro de *Servlet*[18], efetuando a autenticação de cada solicitação que é recebida, segundo o atributo chave de seu cabeçalho. Após sua autenticação, a solicitação segue para a classe onde será executado o processamento referente à solicitação (*GET*, *POST*, *PUT* e *DELETE*).

No *framework Jersey* as funções onde serão processadas as solicitações após sua autenticação, possuem explicitamente seu mapeamento, onde serão enviadas as solicitações do filtro para a classe de processamento, e a conversão de sua resposta para o formato *JSON*. Cada método deve possuir uma anotação relativa ao seu processamento, com as anotações *@GET*,*@POST*,*@PUT* e *@DELETE*. A Listagem 4.1 mostra a assinatura do método de Listar Todos (*GET*) do microsserviço de permissão. Quando a aplicação necessita de receber por parâmetro um valor ou objeto em *JSON*, é necessária a anotação *@Consumes(MediaType.APPLICATION_JSON)* para converter o tipo de objeto do formato *JSON* para a forma do objeto corretamente. Durante o desenvolvimento dos microsserviços, alguns dos valores passados por parâmetro não foram convertidos corretamente, ou apresentaram erros de conversão referentes aos seus valores. Assim, foi necessária a implementação de um método para converter estes valores de forma correta.

[18] *Servlet Filter* podem tratar requisições recebidas de clientes Web.

```
@Path("/permissao")
@Produces(MediaType.APPLICATION_JSON)
public class PermissaoService {
    @GET
    public List<Permissao> getListaPermissoes(){}
}
```

Listing 4.1: Assinatura do método GET do microsserviço com o framework Jersey.

Já no *framework Spring Boot*, a anotação *@RestController* converte automaticamente todas as solicitações recebidas e todas as respostas geradas para o formato *JSON*, não necessitando de explicitar em cada chamada sua conversão. É necessário também, definir o mapeamento da classe, o caminho onde em que o filtro enviará para a classe de processamento as solicitações, e o mapeamento de cada método, de forma a assinalar a cada método seu tipo conforme seu processamento (*@GetMapping('Path')*). A Listagem 4.2 mostra o método de Listar Todos (*GET*) do microsserviço de permissão.

```
@RestController
@RequestMapping("/PermissaoMs/webresources/permissao")
public class PermissaoService {
     @GetMapping()
     public List<Permissao> getListaPermissoes(){}
}
```

Listing 4.2: Assinatura do método GET do microsserviço com o framework Spring Boot.

4.8.2 Configuração

Para o gerenciamento das dependências de cada *framework* em cada microsserviço, foi utilizada a ferramenta de automação de projetos e dependências *Maven*[19]. Assim, ao serem declaradas as dependências de cada microsserviço, a própria aplicação efetua o *download* e a configuração de cada dependência.

[19]https://mvnrepository.com/

No *framework Jersey*, o filtro do microsserviço deve ser configurado em um arquivo *XML* (*web.xml*) para que ele seja encontrado pela aplicação do microsserviço e sejam enviadas as solicitações. O arquivo de configuração (*pom.xml*) é configurado adicionando as dependências do *framework Jersey*, para serem aplicados ao microsserviço. Para adicionar as dependências, dentre as várias opções, foi necessário buscar e avaliar as que possuem funcionalidades que necessitam ser utilizadas no microsserviço, e efetuando os testes necessários para a confirmação de seu uso correto.

No *framework Spring Boot*, o filtro do microsserviço é automaticamente registrado na aplicação do microsserviço assim que o microsserviço é executado. Já no arquivo de configuração de dependências (*pom.xml*), foi necessária a declaração de somente duas dependências-chave para o *framework Spring Boot* ser configurado no microsserviço: *Spring Boot Starter Parent* e *Spring Boot Starter Web*. Com estas duas dependências, o *Spring Boot* é configurado e pode ser inicializado com o servidor nativo *Apache Tomcat*. Este *framework* também, disponibiliza um arquivo de configuração interno (*application.properties*) onde pode ser gerenciadas todas as configurações diponibilizadas pelo *framework*, como: arquivos e *logs* de *debug*, propriedades de *logging*, habilitar o *proxy* automático, nome da aplicação, configurações de administrador, informações de classes de base (*beans*[20]), *cache*, informações internas da aplicação, *email*, internacionalização e codificação, entre outros.

No microsserviço com o *framework Spring Boot* com o servidor externo, o arquivo de dependências (*pom.xml*) é configurado para que o microsserviço ignore o servidor nativo, que traz como padrão na configuração de sua dependência (*Spring Boot Starter Web*). Assim, o microsserviço pode ser implantado em um servidor externo como uma aplicação Web comum.

4.8.3 Conclusão

Mesmo os *frameworks* sendo voltados para o desenvolvimento de microsserviços no ambiente Web, as diferenças entre ambos é notável. O *framework Jersey* apresenta configuração de classes através de anotações e filtros sob a forma de arquivo *web.xml*. Ao

[20]Um *bean* é uma classe Java que expõe propriedades com através de *getter* e *setter*.

desenvolver a aplicação neste *framework* é necessário efetuar uma pesquisa sobre as funcionalidades desejadas em suas dependências para adicioná-las ao arquivo de configuração (*pom.xml*). Por outro lado, o *framework Spring Boot* possui sua configuração de classes através de anotações e a configuração de filtros é feita de forma automática. Para desenvolver e executar a aplicação, são necessárias somente duas dependências base, para configurar a aplicação completa e deixá-la pronta para uso.

4.9 Análise dos Resultados

As métricas a serem avaliadas através dos testes de desempenho nos microsserviços são a média, vazão, taxa de perda de solicitações, linhas de 90% e 99% e a latência média dos tempos de resposta das solicitações enviadas e recebidas aos microsserviços. Estas métricas são importantes para a avaliação do tempo de resposta das solicitações efetuadas e, principalmente, em períodos de maior utilização dos serviços pelos usuários. Neste tempo são incluídas desde o envio, o tempo em que o microsserviço executa suas operações lógicas e acessa os dados no banco de dados, até a resposta ser recebida. A análise será feita comparando os microsserviços de mesmo contexto, ou seja, os microsserviços implementados com as tecnologias *Spring Boot* e *Jersey* comparando os resultados dos testes efetuados sob as funcionalidades *GET*, *POST*, *PUT* e *DELETE*, que permitem listar, cadastrar, atualizar e remover permissões, usuários e páginas, cada um em seu respectivo microsserviço. As instâncias de teste utilizadas neste trabalho, possuem a notação no formato (N)u(M)r, sendo N o número de usuários que efetuarão as solicitações, e M o número de solicitações.

4.9.1 Teste de Carga

Submetendo os microsserviços ao teste de carga, pode-se observar os diferentes efeitos das tecnologias em seu uso cotidiano em sistemas computacionais. A quantidade de usuários deve ser produzida de forma que possa ser semelhante a carga real que o sistema em que está em uso no ambiente de produção. Assim, é possível analisar suas métricas segundo as necessidades atuais de uso do sistema. Desta forma, são efetuados os testes nas

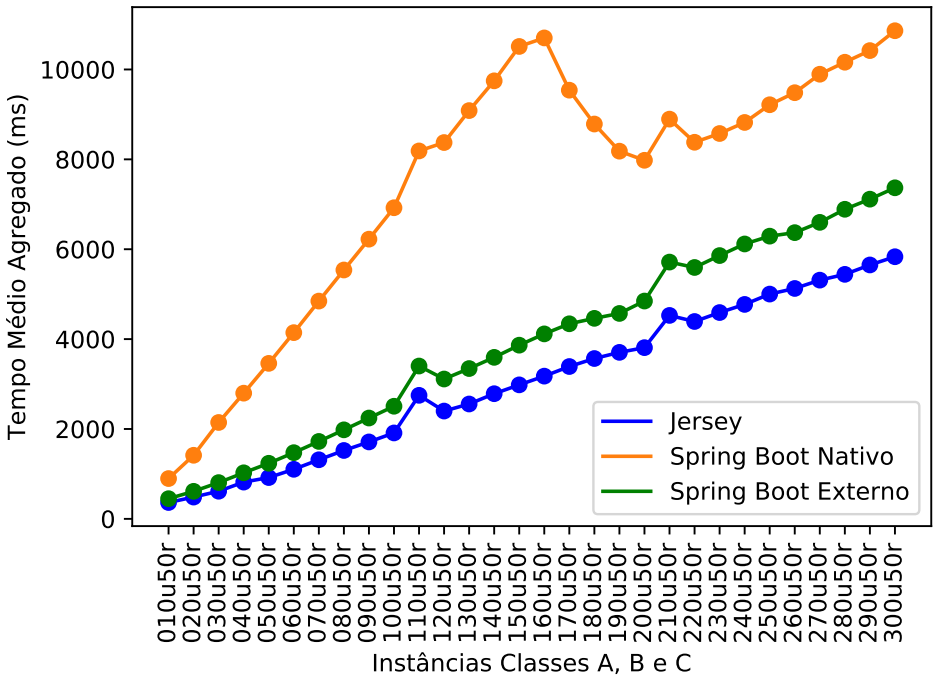
funcionalidades de *GET*, *POST*, *PUT* e *DELETE* dos microsserviços, e analisados comparando cada funcionalidade entre os microsserviços de diferentes tecnologias. Durante as execuções dos testes em cada classe, os microsserviços são iniciados cada qual em sua vez para serem testados, isoladamente. Após o teste ele é desligado e removido sua instância para que outro microsserviço seja iniciado, não sendo executados dois ao mesmo tempo. Com isso, não ocorre interferência dos resultados entre eles.

Análise da Latência

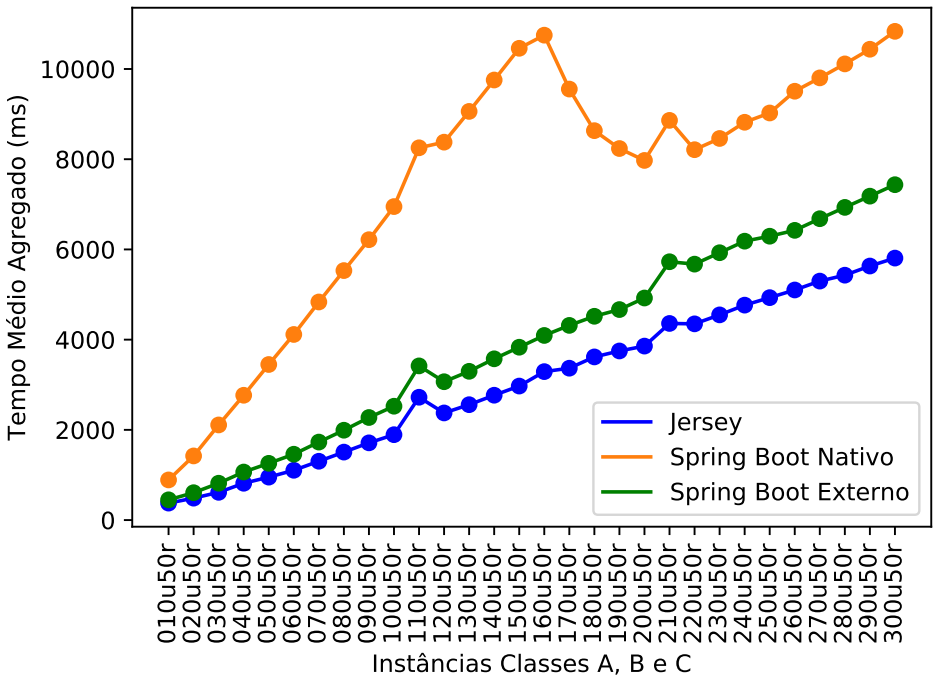
Ao efetuar o teste de carga nos microsserviços e avaliando sua latência, nota-se uma semelhança entre as latências de cada microsserviço e entre os microsserviços de contextos diferentes (página e usuários). Com isso, pode-se verificar que o microsserviço desenvolvido com o *framework Spring Boot* com o servidor interno nativo, possui uma latência maior em comparação aos outros microsserviços avaliados. Este fato ocorre em todas as instâncias de teste efetuadas, sendo que nas instâncias da Classe B (110 a 200 usuários), agrava-se ainda mais essa diferença. Esta diferença ocorre em ambos os recursos de Listar, Cadastrar, Atualizar e Remover páginas, permissões e usuários. Os microsserviços desenvolvidos com *framework Spring Boot* implantado em um servidor externo e o *framework Jersey*, em geral, possuem uma diferença pequena entre seus valores de latência, sendo que nas primeiras instâncias, possuem latências semelhantes. Este fato é mostrado nos gráficos das Figuras 4.4(a), 4.4(b), 4.5(a) e 4.5(b).

Porém, há uma pequena variância na instância de 210u50r no microsserviço de permissão, com recurso de Listar Todas as Permissões e Castrar Permissão, com o *framework Spring Boot* com servidor nativo embutido. Esta é uma instância do teste na Classe C, sendo executada logo após o servidor ser iniciado para este teste. Este fato é mostrado nas Figuras 4.6(a) e 4.6(b). Como todos os testes foram realizados nas mesmas circunstâncias e sob os mesmos critérios, esta latência pode significar um atraso do *framework Spring Boot* durante seu funcionamento inicial, ao receber as solicitações, levando um tempo maior para serem processadas as solicitações. Os testes específicos para observar estas causas deverão ser feitos como trabalhos futuros.

Há também uma variação nos valores do *framework Jersey* no recurso de Ca-



(a) Recurso de listar todas as páginas / todos os usuários.

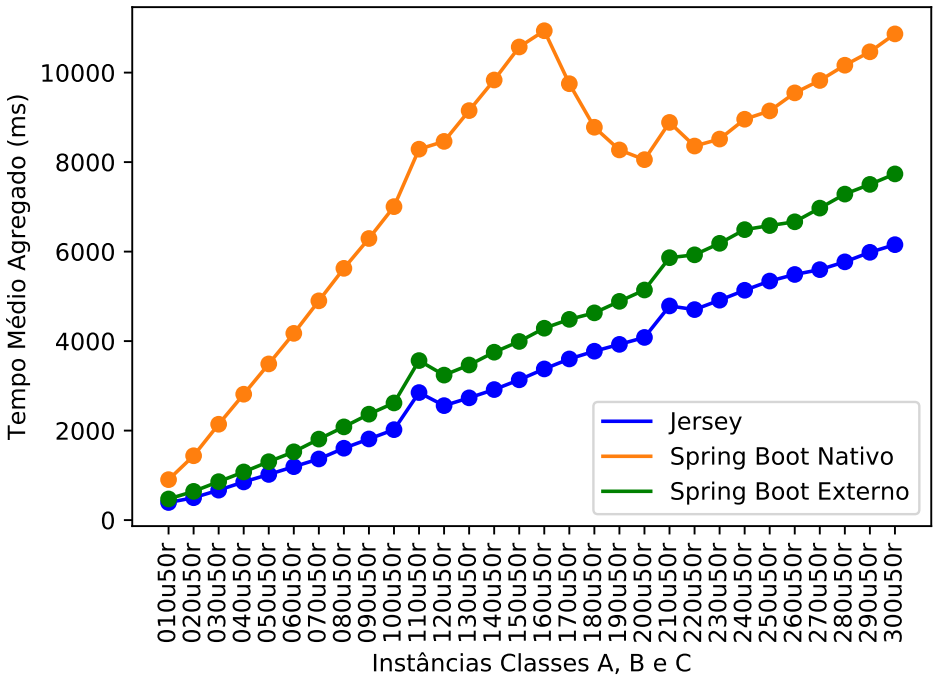


(b) Recurso de atualizar página / usuário.

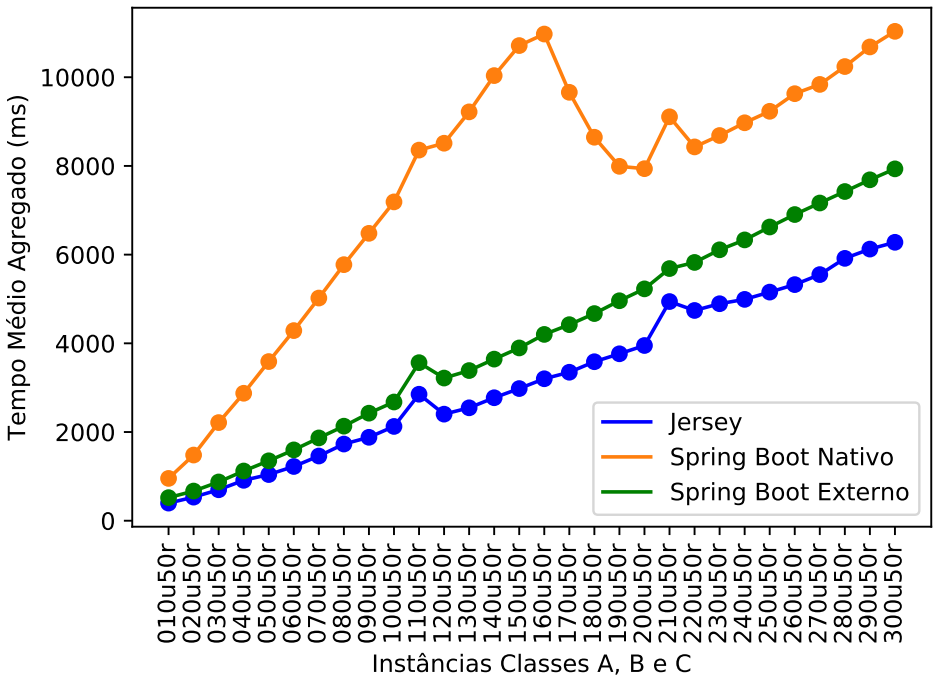
Figura 4.4: Resultados das latências dos microsserviços.

dastro, nos microsserviços de usuários e permissões. Esta variação ocorreu na instância 280u50r na Classe C das instâncias de teste. Pode ser visualizado nas Figuras 4.7(a) e 4.7(b).

Em todos os resultados, o tempo entre o envio da solicitação para o microsserviço



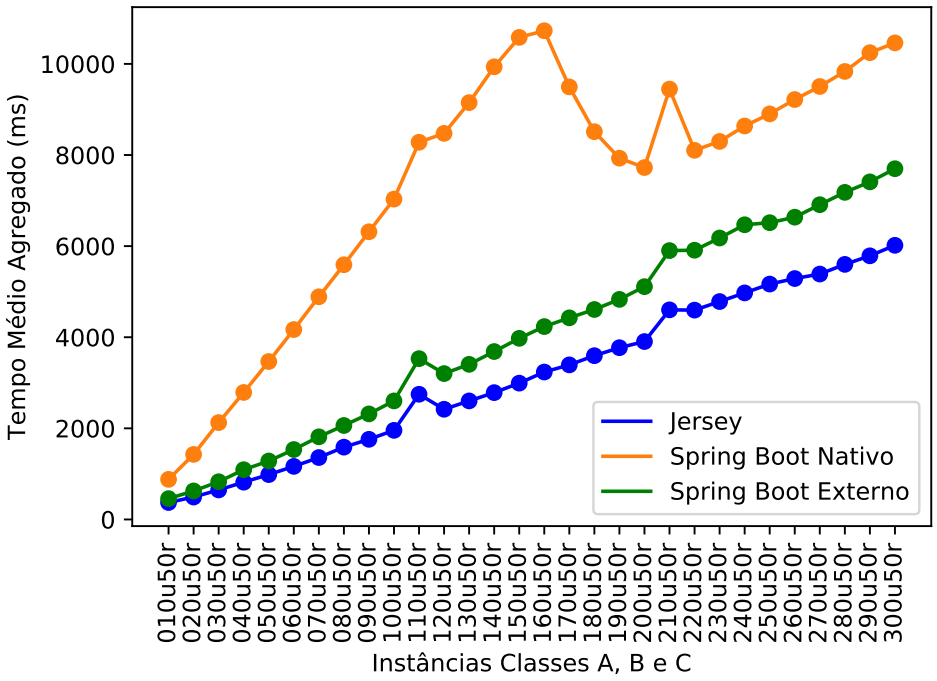
(a) Recurso de cadastrar página.



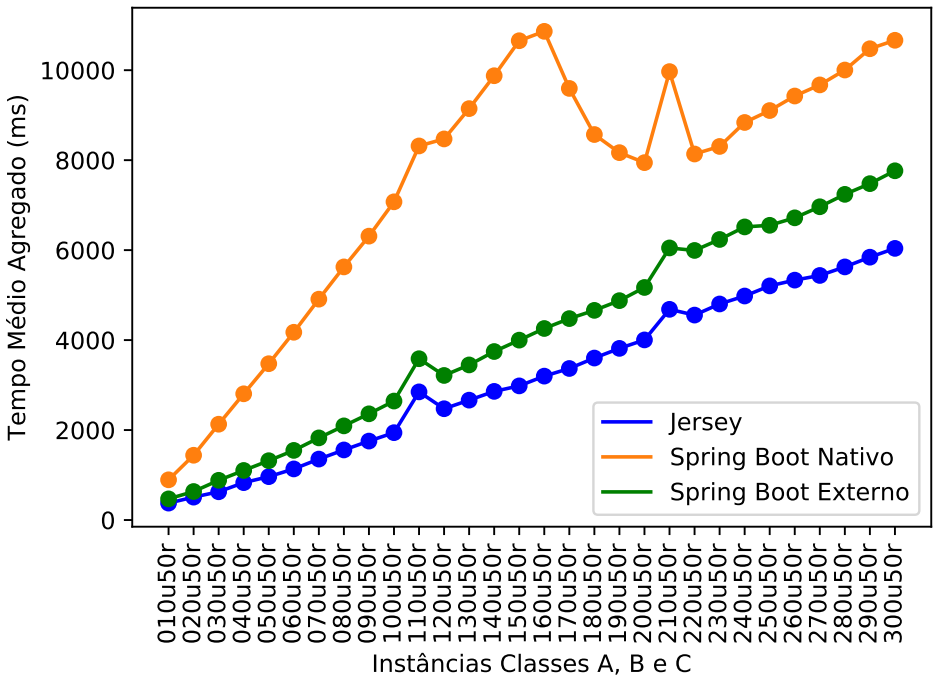
(b) Recurso de remover usuário / página.

Figura 4.5: Resultados das latências dos microsserviços.

e o começo de sua resposta, é amplamente maior no *framework Spring Boot* (que possui o servidor embutido), do que nos microsserviços *Spring Boot* (com servidor externo) e *Jersey*. Com isso, observando-se somente a tecnologia do *Spring Boot*, pode-se notar uma diferença entre possuir o servidor internamente (servidor nativo), e implantá-lo em um



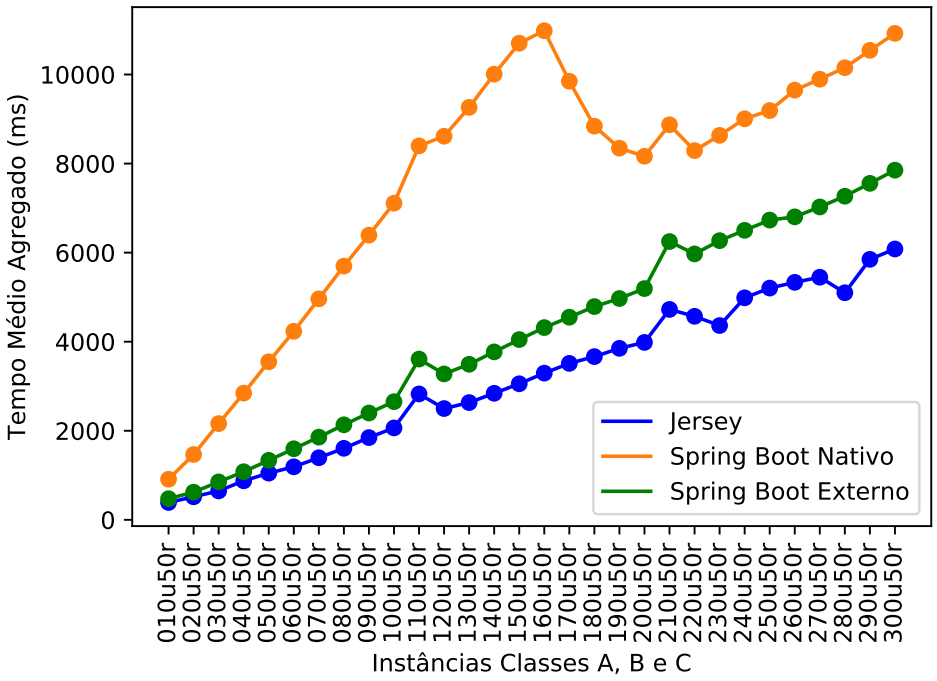
(a) Recurso de listar todas as permissões.



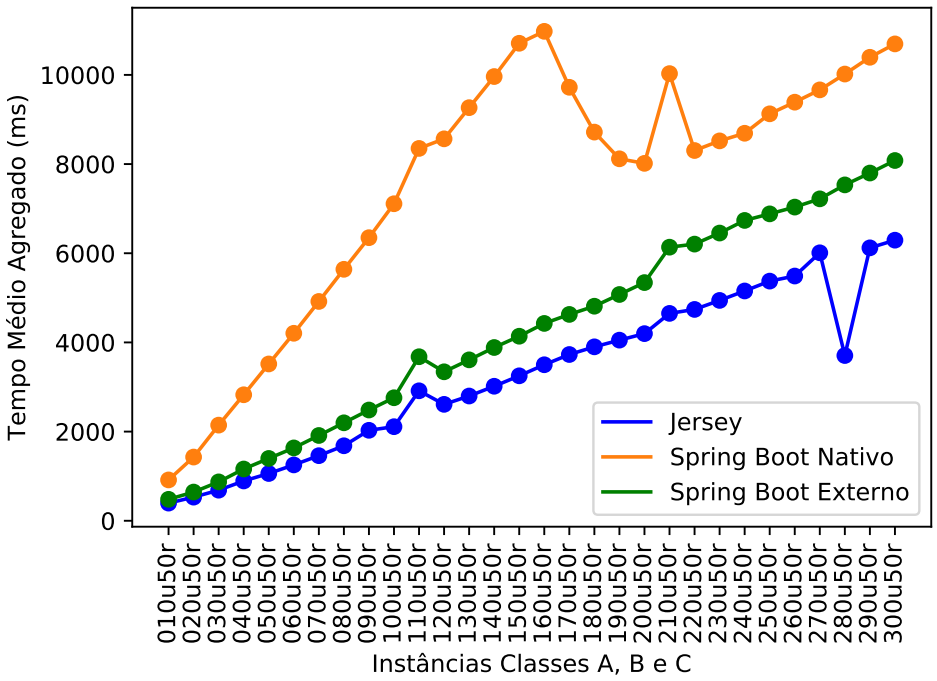
(b) Recurso de listar todas as permissões / remover permissão.

Figura 4.6: Resultados das latências dos microsserviços.

servidor externo. A avaliação destas características e o aprofundamento nos efeitos do servidor, serão efetuados em trabalhos futuros.



(a) Recurso de cadastrar de usuário.



(b) Recurso de cadastrar permissão.

Figura 4.7: Resultados das latências dos microsserviços.

Análise da Taxa de Transferência (Vazão)

A taxa de transferência relacionada com os microsserviços mostra que o *framework Jersey* é capaz de processar uma quantidade de solicitações maior do que os outros microsserviços. Sua taxa aproxima-se do *framework Spring Boot*, implantado no servidor externo, na

primeira instância (10u50r) somente. O *framework Spring Boot*, implantado no servidor externo, tende a seguir o mesmo padrão de processamento de solicitações, porém com uma taxa de transferência inferior ao *Jersey*. O *framework Spring Boot* com o servidor nativo, possui a menor taxa de transferência, não se aproximando de nenhum dos demais microsserviços. Estes dados podem ser observados nas Figuras 4.8(a), 4.8(b), 4.9(a) e 4.9(b).

Isso implica que o *framework Spring Boot* com servidor nativo, não consegue lidar com grandes cargas de dados, ou seja, não é capaz de processar grandes quantidades de usuários concorrentes efetuando solicitações no mesmo instante de tempo para o microsserviço. Neste ponto ainda possui uma diferença grande entre os microsserviços desenvolvidos com o *framework Spring Boot*. O microsserviço desenvolvido com *Spring Boot* e implantado em um servidor externo, possui uma taxa de transferência superior ao microsserviço *Spring Boot* com o servidor nativo.

Análise da Média do Tempo de Resposta

Avaliando os microsserviços e o tempo médio de resposta (em milissegundos), pode-se observar o mesmo padrão de valores vistos na análise da latência. Porém tomando por medida um valor de tempo base para a avaliação, seja de 1 segundo para as solicitações efetuadas, pode-se notar que o *framework* Jersey possui 83.3% de solicitações (englobando todas as instâncias) acima do valor limite de 1 segundo no microsserviço de página, e 86.6% nos microsserviços de permissão e usuário. Já o *framework Spring Boot* com servidor externo, possui 90% de solicitações acima do valor limite. O *framework Spring Boot* com servidor embutido, possui 96.6% das solicitações acima do limite de 1 segundo, em ambos os microsserviços. Os dados das médias de tempo, agrupados por recursos, são mostrados nas Tabelas 4.1, 4.2 e 4.3, mostrando os valores médios calculados a partir dos recursos de cada microsserviço (página, permissão e usuário), para os *frameworks Jersey*, *Spring Boot* com servidor embutido e externo.

Desta forma, o tempo entre o envio da solicitação para o microsserviço, seu processamento e o término do envio de todas os pacotes de sua resposta, é amplamente maior no *framework Spring Boot* (que possui o servidor nativo), do que nos microsserviços *Spring*

Framework Jersey			
Instâncias	Página	Permissão	Usuário
010u50r	377 ± 14.5	381 ± 12.0	380 ± 15.5
020u50r	490 ± 7.5	510 ± 19.0	507 ± 15.0
030u50r	633 ± 27.0	654 ± 29.0	641 ± 26.0
040u50r	829 ± 18.5	847 ± 37.5	842 ± 45.0
050u50r	963 ± 50.5	1003 ± 48.0	1004 ± 64.0
060u50r	1133 ± 45.5	1184 ± 58.5	1163 ± 67.0
070u50r	1328 ± 31.0	1391 ± 52.5	1377 ± 57.5
080u50r	1545 ± 49.5	1609 ± 62.0	1587 ± 71.0
090u50r	1745 ± 50.5	1847 ± 137.0	1806 ± 68.0
100u50r	1942 ± 64.0	2002 ± 84.0	2017 ± 78.0
110u50r	2774 ± 63.5	2837 ± 83.5	2815 ± 91.5
120u50r	2444 ± 91.5	2501 ± 96.0	2499 ± 90.5
130u50r	2615 ± 88.0	2688 ± 98.0	2646 ± 89.5
140u50r	2823 ± 74.5	2887 ± 116.5	2868 ± 102.5
150u50r	3029 ± 82.0	3075 ± 134.5	3068 ± 118.5
160u50r	3281 ± 101.0	3311 ± 150.0	3296 ± 130.5
170u50r	3452 ± 117.0	3497 ± 180.0	3505 ± 158.5
180u50r	3653 ± 103.0	3699 ± 153.0	3690 ± 133.5
190u50r	3793 ± 111.0	3879 ± 140.5	3850 ± 123.5
200u50r	3916 ± 136.0	4035 ± 146.0	3972 ± 136.0
210u50r	4557 ± 213.5	4645 ± 41.5	4599 ± 175.5
220u50r	4482 ± 177.0	4630 ± 92.0	4541 ± 143.0
230u50r	4684 ± 182.5	4842 ± 79.5	4623 ± 277.0
240u50r	4891 ± 185.5	5035 ± 90.5	4963 ± 180.5
250u50r	5091 ± 205.5	5249 ± 104.0	5186 ± 198.0
260u50r	5239 ± 194.0	5370 ± 100.5	5285 ± 173.0
270u50r	5402 ± 149.5	5611 ± 311.5	5452 ± 196.0
280u50r	5547 ± 171.0	4977 ± 960.5	5447 ± 354.0
290u50r	5755 ± 176.0	5917 ± 167.0	5832 ± 211.5
300u50r	5932 ± 173.5	6116 ± 137.5	6041 ± 190.5

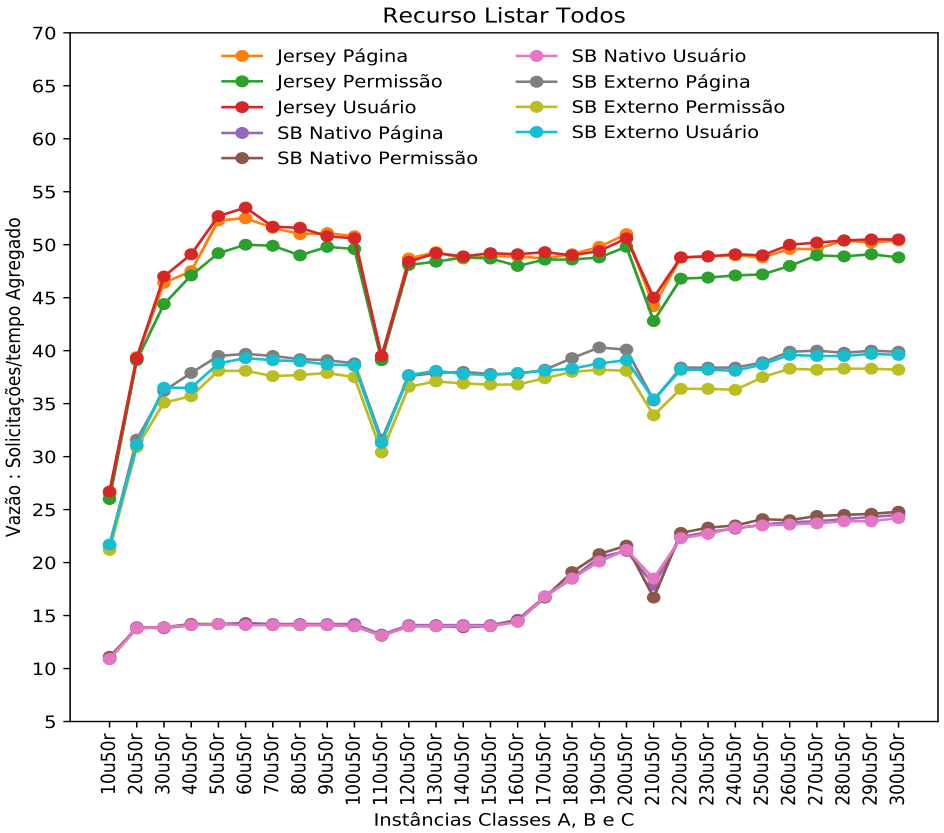
Tabela 4.1: Valores médios de tempo em milissegundos dos recursos por instância do *framework Jersey*.

Framework Spring Boot com servidor externo			
Instâncias	Página	Permissão	Usuário
010u50r	456 ± 11.5	469 ± 12.0	465 ± 16.5
020u50r	620 ± 17.5	634 ± 9.0	637 ± 23.5
030u50r	827 ± 26.0	861 ± 27.5	836 ± 30.5
040u50r	1055 ± 25.5	1118 ± 34.5	1073 ± 8.0
050u50r	1266 ± 34.0	1333 ± 57.5	1317 ± 53.0
060u50r	1486 ± 33.0	1572 ± 49.5	1544 ± 55.0
070u50r	1754 ± 43.5	1851 ± 49.5	1804 ± 60.0
080u50r	2018 ± 51.0	2117 ± 67.0	2074 ± 67.5
090u50r	2295 ± 61.5	2387 ± 83.5	2324 ± 65.5
100u50r	2549 ± 56.0	2668 ± 78.5	2579 ± 66.0
110u50r	3461 ± 82.0	3598 ± 74.5	3550 ± 98.5
120u50r	3139 ± 86.5	3252 ± 69.5	3222 ± 90.0
130u50r	3369 ± 84.0	3487 ± 102.0	3444 ± 91.5
140u50r	3640 ± 88.5	3772 ± 100.0	3727 ± 94.0
150u50r	3896 ± 80.0	4038 ± 81.5	4014 ± 114.0
160u50r	4165 ± 97.5	4306 ± 96.0	4261 ± 120.0
170u50r	4381 ± 85.0	4510 ± 101.0	4488 ± 120.5
180u50r	4537 ± 84.5	4694 ± 101.5	4726 ± 125.5
190u50r	4708 ± 158.5	4927 ± 122.0	4903 ± 116.5
200u50r	4970 ± 147.5	5207 ± 116.0	5107 ± 116.5
210u50r	5768 ± 75.0	6029 ± 117.0	5958 ± 298.0
220u50r	5732 ± 167.0	6036 ± 147.5	5868 ± 202.5
230u50r	5990 ± 163.0	6289 ± 137.5	6146 ± 193.5
240u50r	6263 ± 187.0	6573 ± 133.5	6404 ± 190.5
250u50r	6388 ± 145.5	6649 ± 184.5	6558 ± 204.0
260u50r	6486 ± 148.5	6796 ± 200.0	6672 ± 192.5
270u50r	6750 ± 187.0	7031 ± 155.5	6920 ± 184.0
280u50r	7035 ± 198.0	7317 ± 176.5	7175 ± 195.0
290u50r	7265 ± 194.0	7562 ± 194.5	7433 ± 213.5
300u50r	7512 ± 184.5	7847 ± 190.5	7712 ± 209.5

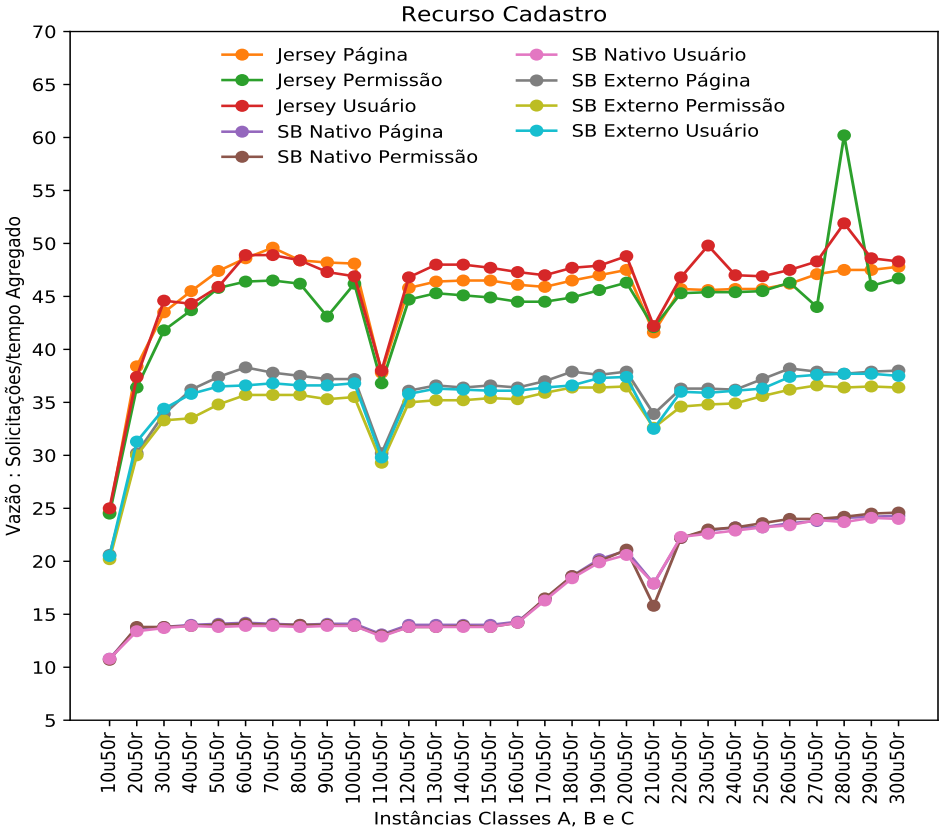
Tabela 4.2: Valores médios de tempo em milissegundos dos recursos por instância do *framework Spring Boot* com servidor externo.

Framework Spring Boot com servidor embutido			
Instancias	Pagina	Permissao	Usuario
010u50r	897 ± 7.0	896 ± 18.0	909 ± 6.5
020u50r	1425 ± 11.5	1432 ± 6.0	1447 ± 20.0
030u50r	2132 ± 18.0	2133 ± 9.5	2145 ± 18.0
040u50r	2792 ± 23.0	2806 ± 19.0	2815 ± 28.0
050u50r	3466 ± 20.5	3486 ± 25.0	3515 ± 37.5
060u50r	4144 ± 29.0	4182 ± 18.0	4212 ± 31.0
070u50r	4860 ± 32.5	4905 ± 15.5	4928 ± 44.5
080u50r	5565 ± 47.5	5619 ± 24.5	5659 ± 55.5
090u50r	6244 ± 38.5	6325 ± 19.0	6360 ± 47.0
100u50r	6959 ± 41.0	7072 ± 39.0	7099 ± 64.0
110u50r	8243 ± 51.0	8315 ± 35.5	8341 ± 74.5
120u50r	8405 ± 44.5	8504 ± 47.0	8545 ± 78.5
130u50r	9098 ± 45.5	9187 ± 59.5	9240 ± 94.0
140u50r	9779 ± 44.0	9925 ± 43.0	9943 ± 91.5
150u50r	10514 ± 57.0	10648 ± 63.5	10641 ± 74.5
160u50r	10796 ± 117.0	10856 ± 124.5	10924 ± 84.0
170u50r	9615 ± 106.5	9604 ± 113.5	9729 ± 169.5
180u50r	8732 ± 76.5	8599 ± 102.5	8773 ± 71.5
190u50r	8230 ± 45.0	8071 ± 118.5	8294 ± 68.5
200u50r	8002 ± 41.0	7895 ± 145.5	8035 ± 103.5
210u50r	8881 ± 17.0	9815 ± 291.0	8852 ± 201.0
220u50r	8316 ± 84.5	8181 ± 100.5	8303 ± 10.0
230u50r	8516 ± 57.5	8374 ± 111.0	8575 ± 72.5
240u50r	8866 ± 69.5	8722 ± 100.5	8897 ± 80.5
250u50r	9128 ± 95.0	9044 ± 112.5	9193 ± 81.0
260u50r	9512 ± 32.0	9344 ± 104.0	9533 ± 102.0
270u50r	9840 ± 45.5	9613 ± 84.5	9911 ± 20.0
280u50r	10146 ± 26.0	9952 ± 92.5	10216 ± 70.5
290u50r	10441 ± 22.5	10372 ± 115.5	10593 ± 49.5
300u50r	10854 ± 15.0	10607 ± 117.0	10926 ± 94.0

Tabela 4.3: Valores médios de tempo em milissegundos dos recursos por instância do *framework Spring Boot* com servidor embutido.

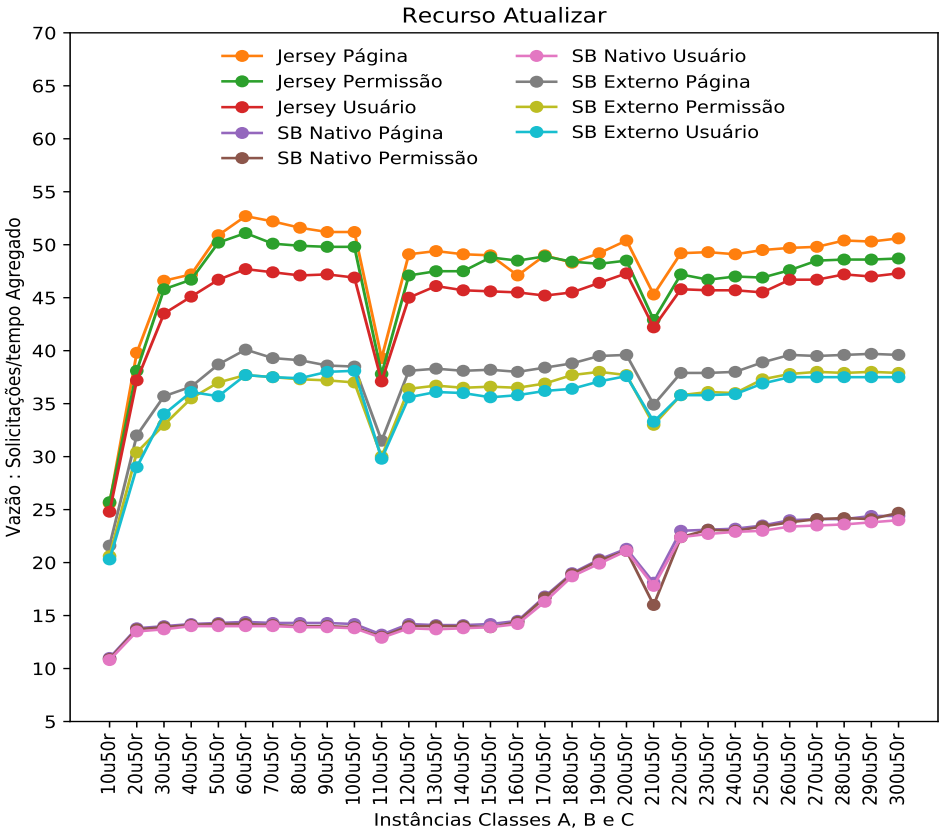


(a) Resultados das tecnologias no recurso de Listar Todos.

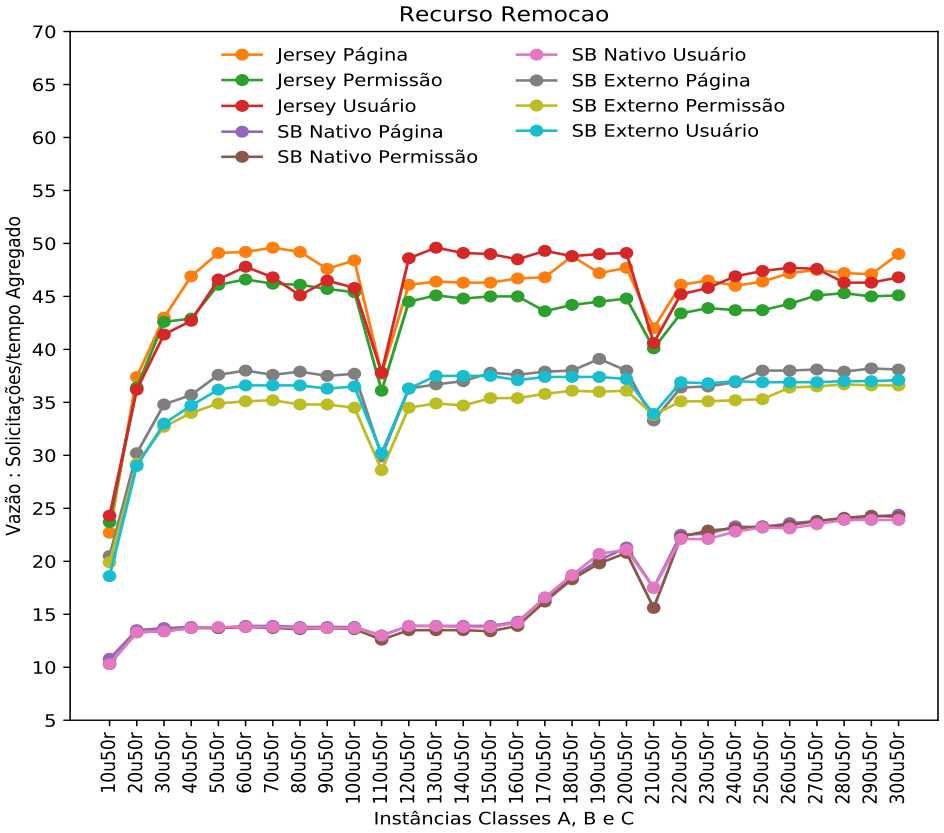


(b) Resultados das tecnologias no recurso de Cadastrar.

Figura 4.8: Resultados de vazão dos microsserviços.



(a) Resultados das tecnologias no recurso de Atualizar.



(b) Resultados das tecnologias no recurso de Remoção.

Figura 4.9: Resultados de vazão dos microsserviços.

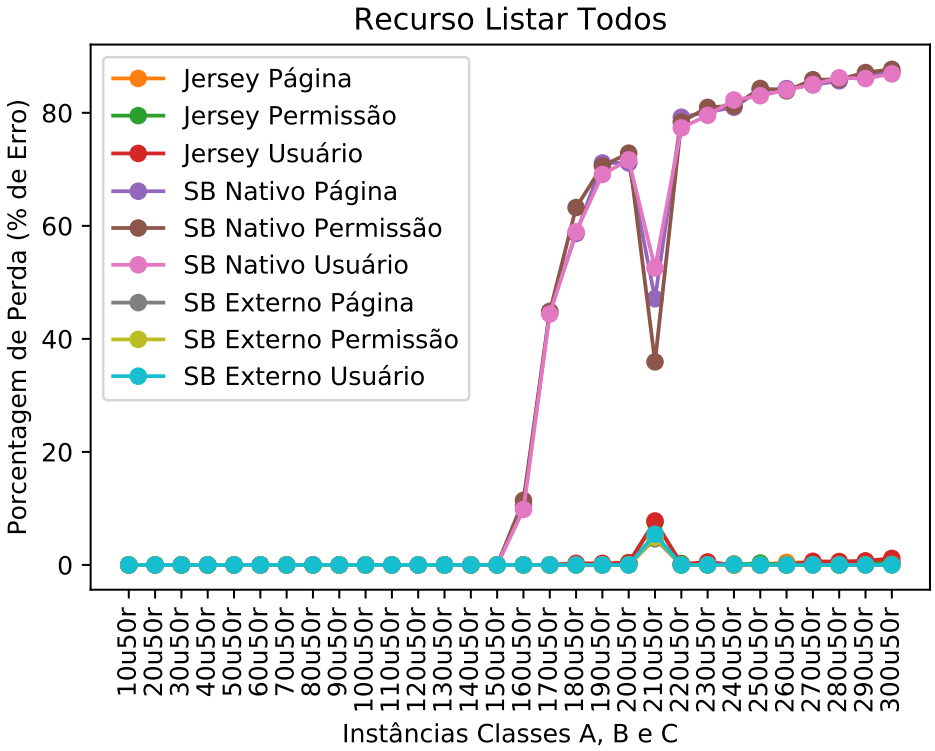
Boot com servidor externo e *Jersey*. Sendo assim, ainda pode-se observar a diferença entre possuir o servidor internamente (servidor nativo), e implantá-lo em um servidor externo nas tecnologias *Spring Boot*.

Análise da Taxa de Perda de Solicitações

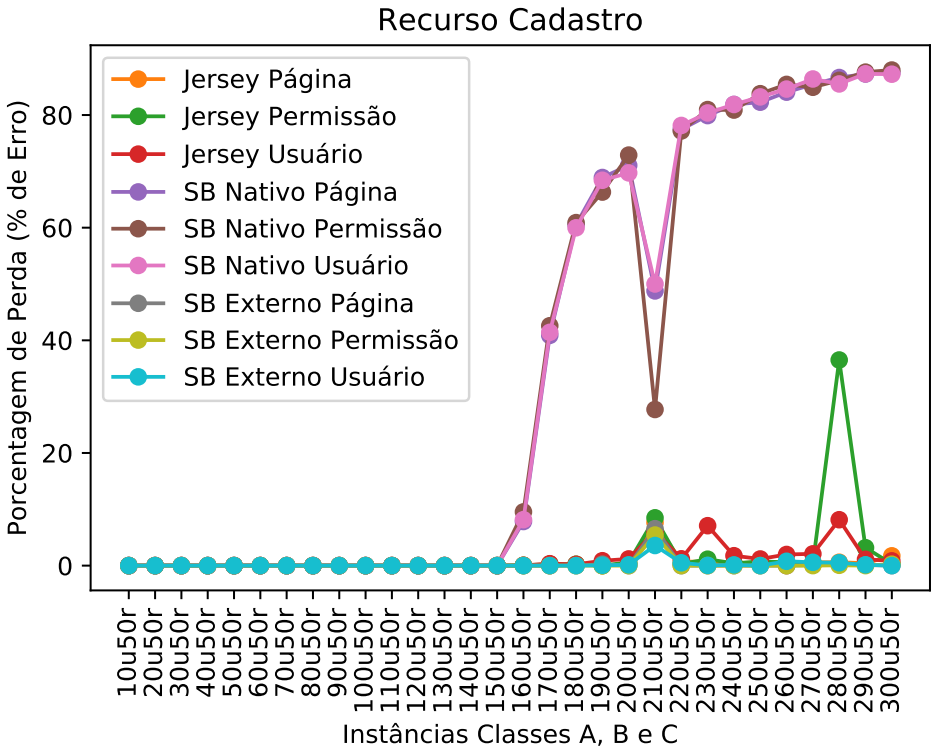
Durante o teste de carga, os microsserviços apresentaram algumas perdas das solicitações (% de erro) ao longo de seu processamento. Como resultado o *framework Spring Boot* com servidor nativo começou a apresentar erro a partir da instância 150u50r, aumentando demasiadamente com o crescimento das instâncias. Ambos os *frameworks Spring Boot*, implantado em servidor externo, e *Jersey*, em geral, tiveram taxas semelhantes de perda, sendo que somente na instância 210u50r apresentaram, ambas, alguma perda. Entre os *frameworks Spring Boot*, implantado em servidor externo, e *Jersey*, o que apresentou menor perda foi o *Spring Boot*, mesmo com pouca diferença. Este fato pode ser mostrado nas Figuras 4.10(a), 4.10(b), 4.11(a) e 4.11(b).

No *framework Spring Boot* com servidor nativo, em todos os recursos do microsserviço de permissão, houveram menores taxas de erro na instância 210u50r. Por ser uma instância do inicio da Classe C, pode-se perceber que seu comportamento vai de encontro ao comportamento das outras duas tecnologias. E mesmo apresentando uma taxa amplamente maior do que as outras tecnologias, por conter o mesmo *framework Spring Boot* em ambos os lados, diferenciando-se somente com relação ao servidor interno ou externo, pode-se notar que o *framework* que possui servidor embutido, no inicio do teste, tende a possui uma taxa menor de perda, em relação as outras instâncias do mesmo. Porém, as outras duas tecnologias apresentam uma taxa de perda relativamente pequena, mas ainda superior as outras instâncias de teste das mesmas.

Em algumas instâncias porém, houveram alterações na taxa de perda do *framework Jersey*, e em sua maioria, localiza-se nas instâncias de Classe C. Os efeitos analisados anteriormente ainda são presentes nas taxas destas instâncias. Este fato ocorre no recurso de Cadastro dos microsserviços de usuário e de permissão, sendo com mais frequência no microsserviço de usuário e com taxa maior no microsserviço de permissão. Pode-se observar este fato anteriormente na Figura 4.10(b).



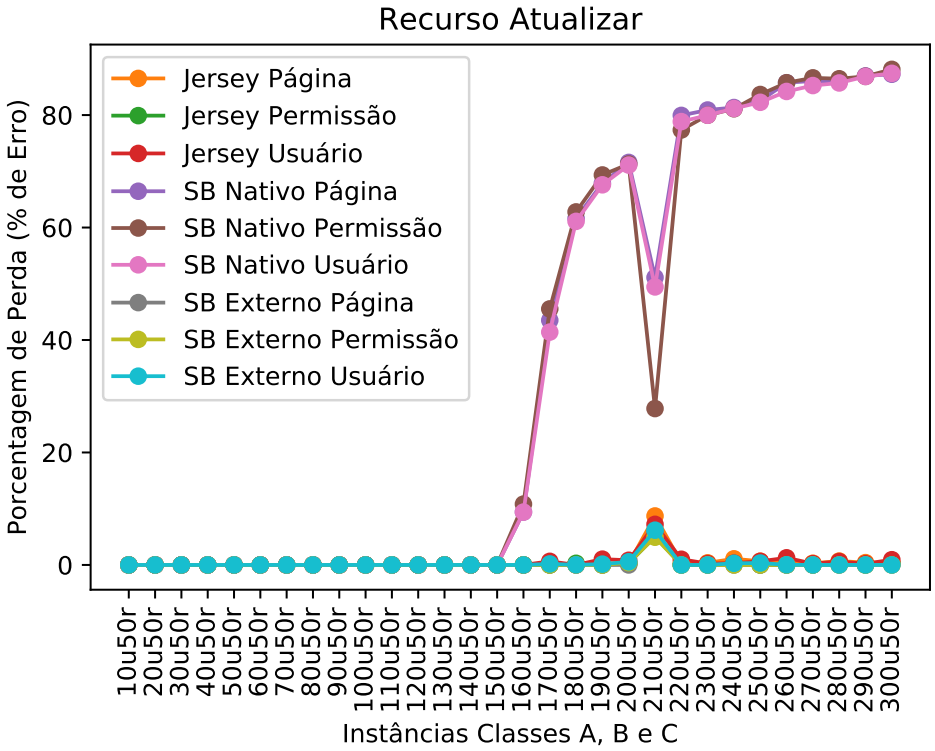
(a) Resultados das tecnologias no recurso de Listar Todos.



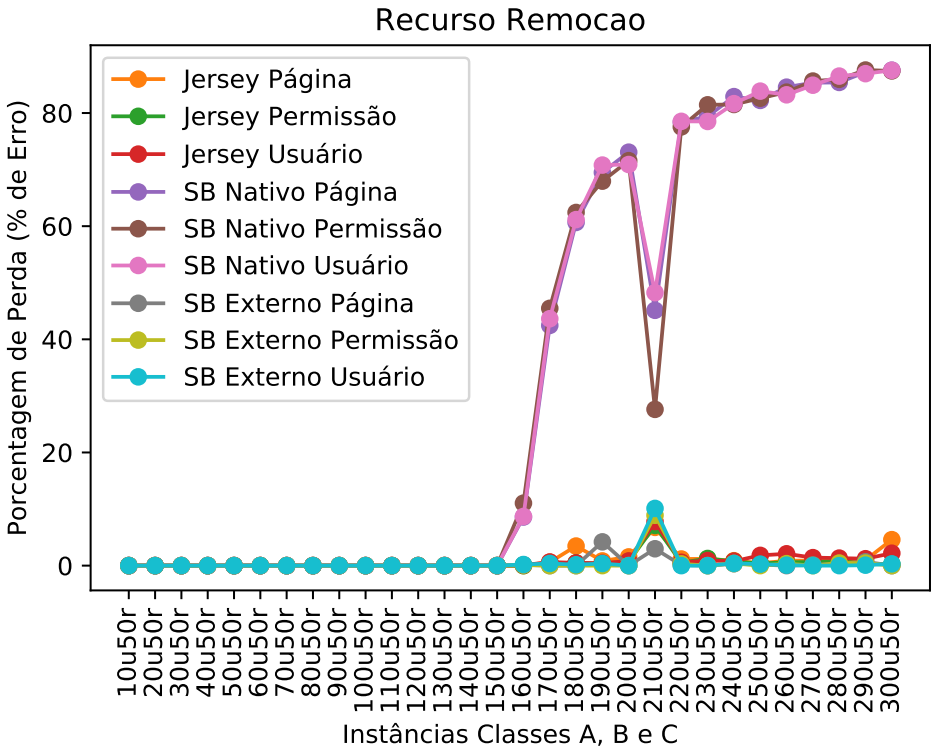
(b) Resultados das tecnologias no recurso de Cadastrar.

Figura 4.10: Resultados da porcentagem de perda de solicitações

Este efeito implica que muitas das solicitações que podem ser enviadas em dias em que muitos usuários se comunicam com o microsserviço ao mesmo tempo (semelhante



(a) Resultados das tecnologias no recurso de Atualizar.



(b) Resultados das tecnologias no recurso de Remoção.

Figura 4.11: Resultados da porcentagem de perda de solicitações

a um período de inscrição), pode haver uma demora ou mesmo uma falha do servidor ao responder para o usuário com as informações solicitadas, ocorrendo principalmente no

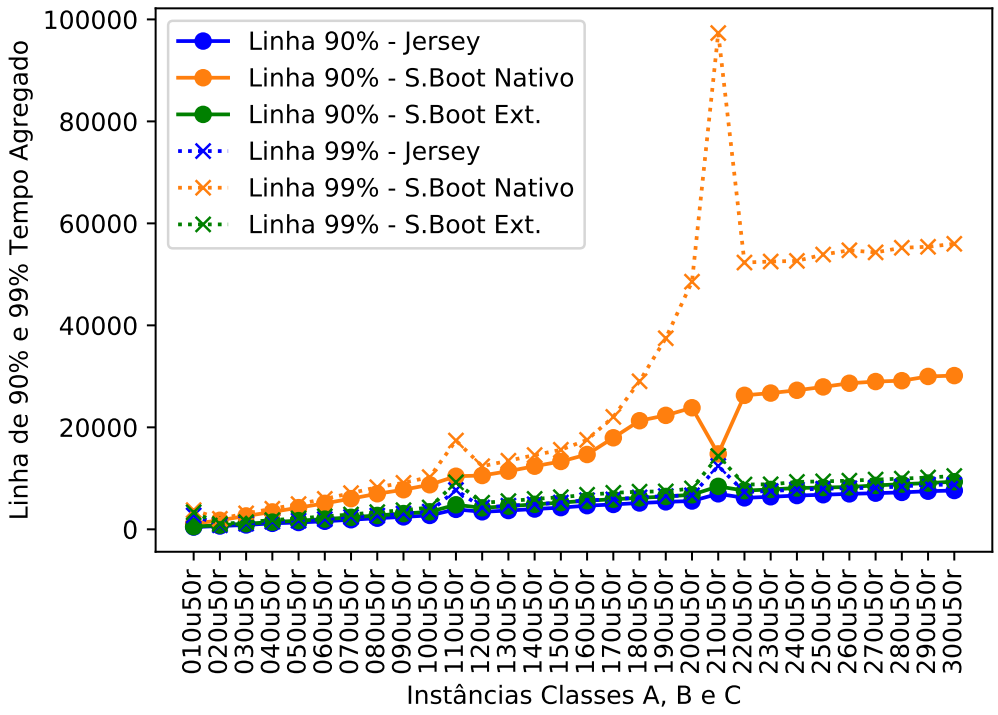
framework Spring Boot com servidor nativo. Nos outros microsserviços, pouca diferença poderá ser notada com relação à falha de solicitações. Em ambos os casos o *framework Spring Boot* com servidor externo, apresentou as menores taxas de perda de solicitações, sendo em seu pior caso, igual ao *framework Jersey*.

Análise da linha de 90% e 99%

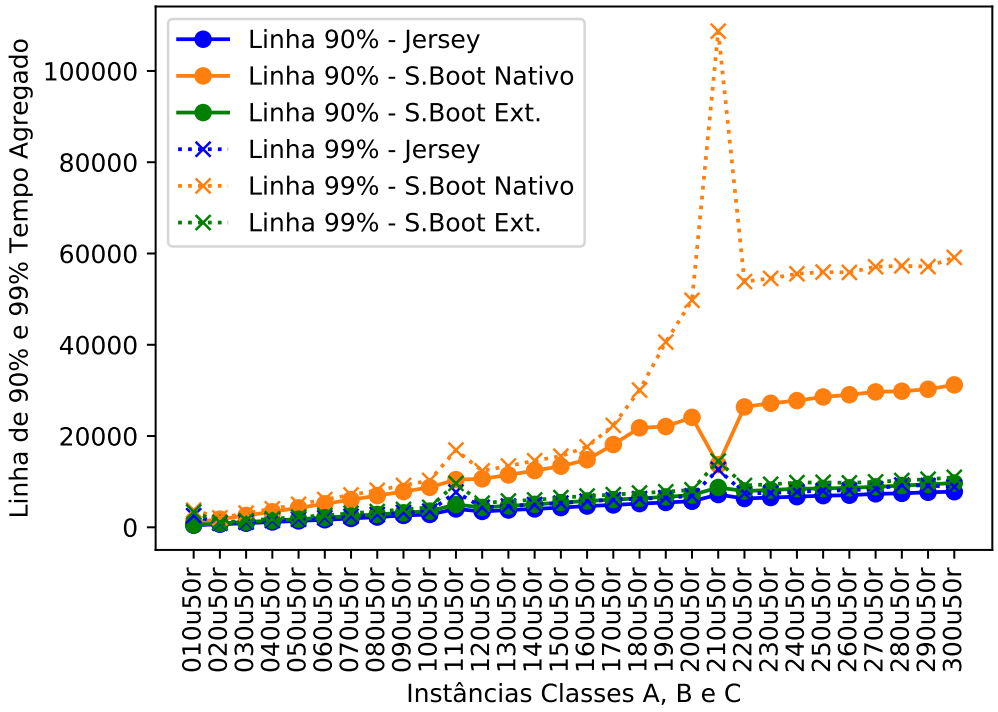
Analisando os microsserviços sob o parâmetro de linha de 90% e 99% do tempo de solicitações, pode-se observar o mesmo comportamento em todos os recursos de cada microsserviço nos *frameworks Jersey* e *Spring Boot* com servidor externo. Nos microsserviços destes *frameworks*, a distância entre os valores da linha de 90% e 99% são menores, sendo quase semelhantes. Já no *framework Spring Boot* com servidor embutido, ocorre o oposto. A distância entre seus valores cresce à medida em que as instâncias aumentam apresentando valores de tempo de solicitações sobre a linha de 99% superiores as demais tecnologias, ocorrendo principalmente na instância 210u50r. Porém, seus valores sob a linha de 90% de solicitações, se aproximam dos valores dos outros microsserviços, principalmente na instância mencionada. Este comportamento diverge do esperado para este *framework*, possuindo uma discrepância grande entre as linhas de 90% e 99%. Esta forma, pode-se observar que pelo menos 9% das solicitações obtiveram um tempo relativamente alto, comparado a linha de 90% e aos outros microsserviços. Este fato é mostrado nas Figuras 4.12(a), 4.12(b), e 4.13, avaliando-se a média dos valores somando os valores de cada recurso em cada *framework*, dos *frameworks* avaliados em página, permissão e usuário (média do *framework* = (Listar Todos + Cadastrar + Atualizar) / 3).

4.9.2 Teste de Estresse

Durante o teste dos microsserviços, ao criar os usuários para realizarem as solicitações, nas instâncias iguais ou acima de 13000 usuários, foram obtidas mensagens do software de teste (*JMeter*) sobre o alcance do limite máximo de memória. Mesmo aumentando a memória de uso, o *JMeter* limitou-se a criar até 12232 usuários para efetuarem solicitações aos microsserviços. Assim, devido a essa limitação, não foi possível desabilitar o servidor e a aplicação com o uso de carga excessiva. Nas análises posteriores, mesmo se referindo



(a) Resultados dos microsserviços de página.



(b) Resultados dos microsserviços de permissão.

Figura 4.12: Linha de 90% e 99% dos tempo de solicitações.

a instâncias maiores, o limite destes usuários foi de 12232 usuários. Como os testes de estresse não foram efetuados com quantidades constantes de solicitações, mas foi utilizado um período de tempo constante para a execução dos testes, logo a notação foi alterada

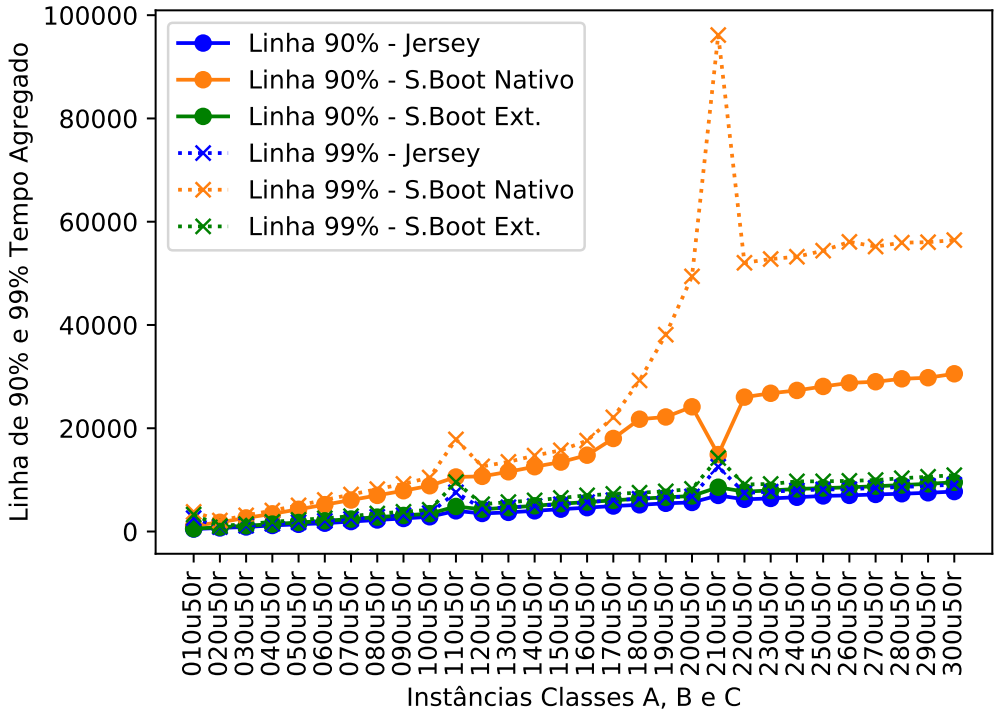
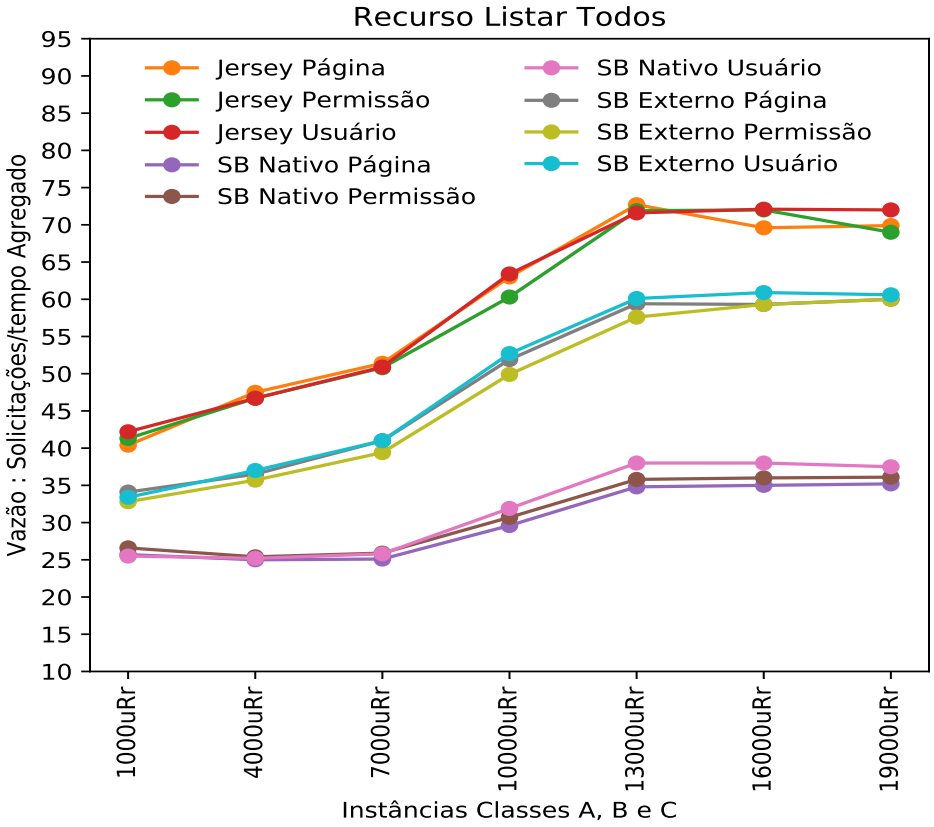


Figura 4.13: Resultados dos microsserviços de usuário da linha de 90% e 99% dos tempo de solicitações.

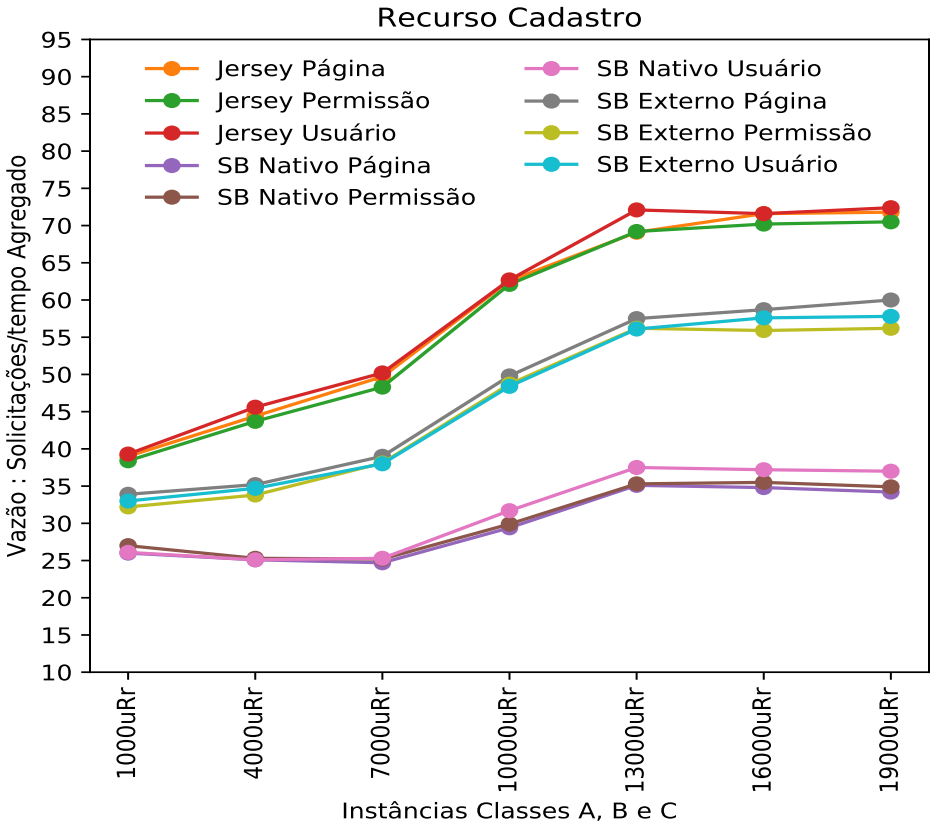
para (N)uRr, sendo N o número de usuários que efetuarão as solicitações, e R indica uma quantidade variável de solicitações.

Análise da Taxa de Transferência (Vazão)

Devido a este teste ser efetuado utilizando grandes quantidades de usuários, espera-se que nas maiores instâncias, os *frameworks* tenham uma taxa menor de transferência. Porém, efetuando o teste de estresse nos microsserviços, observa-se que o *framework Jersey* possui uma taxa de transferência (vazão) acima dos outros microsserviços. Esta diferença já havia sido abordada no teste de carga, mas neste teste, esta diferença fica ainda mais explícita. Esta distinção é notável em todos as instâncias efetuadas no teste, e também observa-se o mesmo efeito nos recursos de Listar Todos, Cadastrar e Atualizar, abordados nos testes. A segunda tecnologia com melhor taxa é o *Spring Boot* (com servidor externo), possuindo uma diferença significativa entre o *Jersey* e o *Spring Boot* (com servidor nativo). Desta forma, ainda conserva-se a diferença existente entre o *Spring Boot* (com servidor externo) e o *Spring Boot* (com servidor nativo). Os gráficos das Figuras 4.14(a), 4.14(b), e 4.15, ilustram os valores da taxa de transferência dos microsserviços.



(a) Resultados das tecnologias no recurso de Listar Todos.



(b) Resultados das tecnologias no recurso de Cadastrar.

Figura 4.14: Resultados da vazão dos microsserviços.

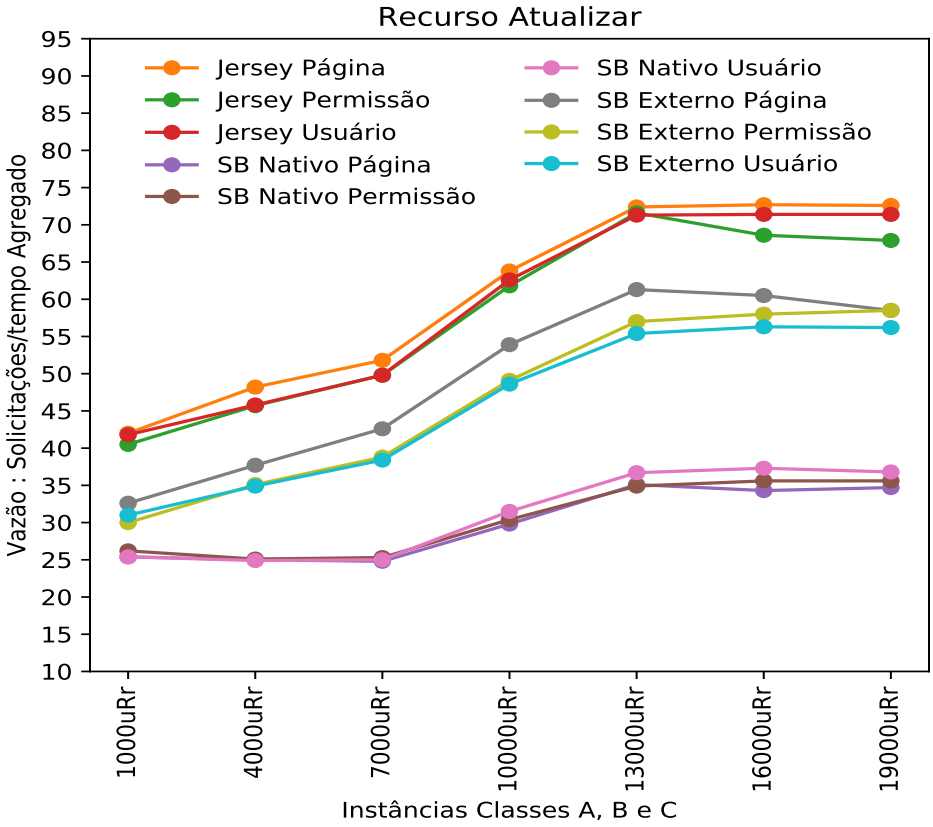
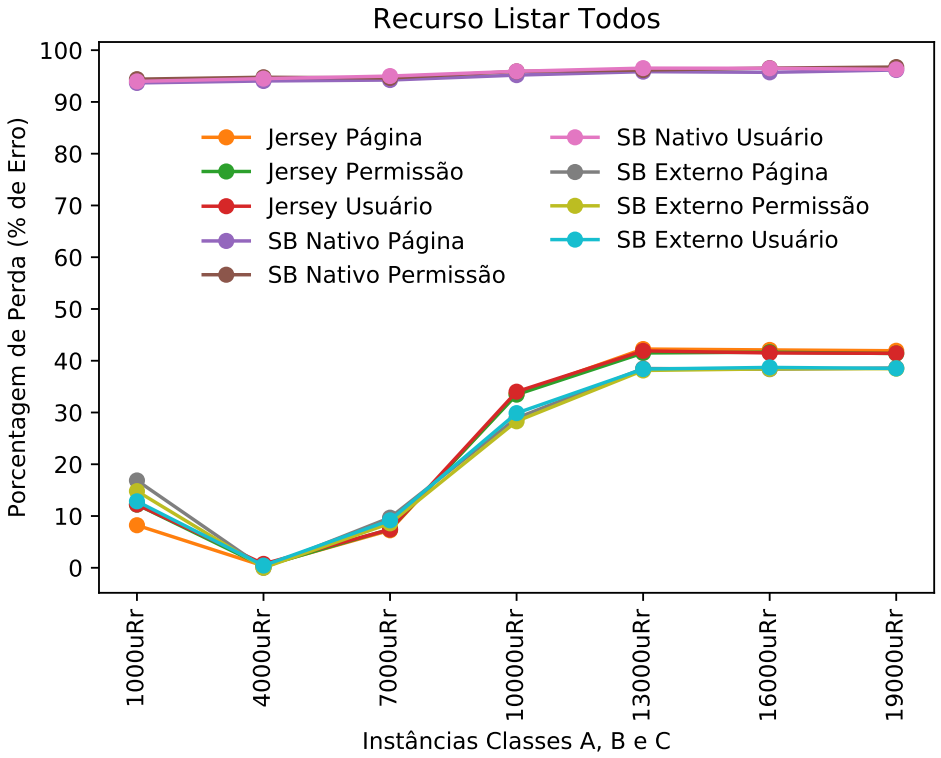


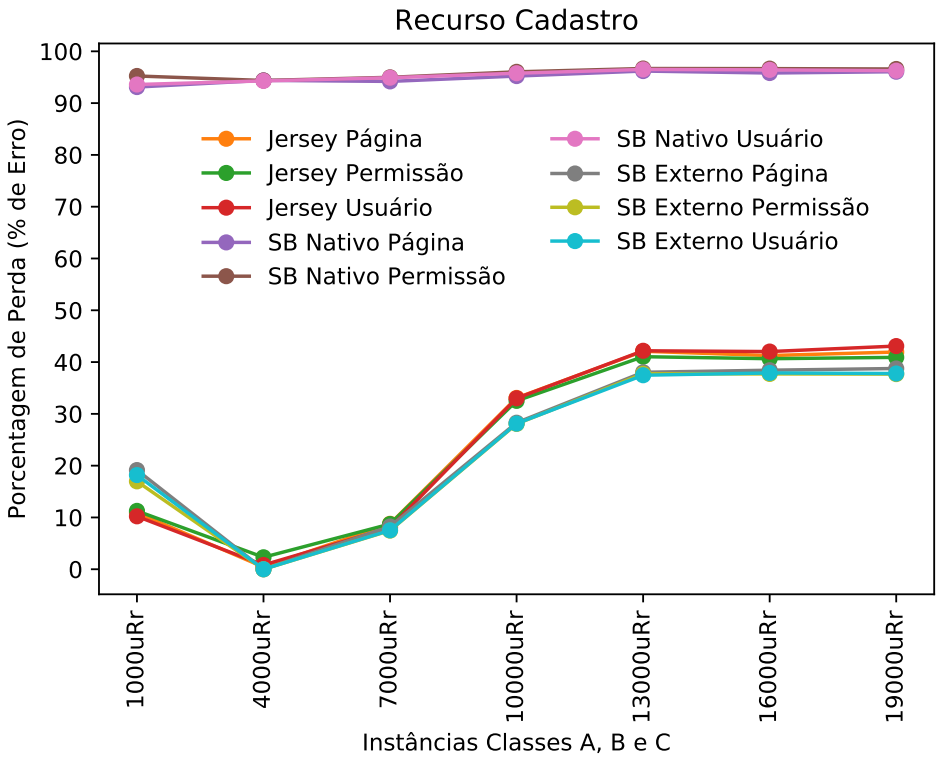
Figura 4.15: Resultados da vazão dos microsserviços no recurso de Atualizar.

Análise da Taxa de Perda de Solicitações

A taxa de perda de repostas neste teste, verifica-se por ser ainda mais alta comparando ao teste de carga, devido ao seu alto número de usuários consumindo os microsserviços. Desta forma, pode-se observar que em ambos os recursos (Listar Todos, Cadastrar e Atualizar), o microsserviço desenvolvido com o *framework Spring Boot* (com servidor nativo), possui a maior taxa de perda de solicitações. Esta taxa permanece superior a 90% durante grande parte das instâncias. Os *frameworks Spring Boot* (com servidor externo) e *Jersey*, possuem uma taxa semelhante na segunda instância, e em alguns microsserviços, na primeira e terceira instâncias. Porém, nas instâncias iguais e acima de 10000 usuários, o *framework Spring Boot* (com servidor externo) possui uma taxa de perda menor do que o *framework Jersey*. Estes dados são mostrados nas Figuras 4.16(a), 4.16(b), e 4.17.



(a) Resultados das tecnologias no recurso de Listar Todos.



(b) Resultados das tecnologias no recurso de Cadastrar.

Figura 4.16: Resultados da porcentagem de perda de solicitações.

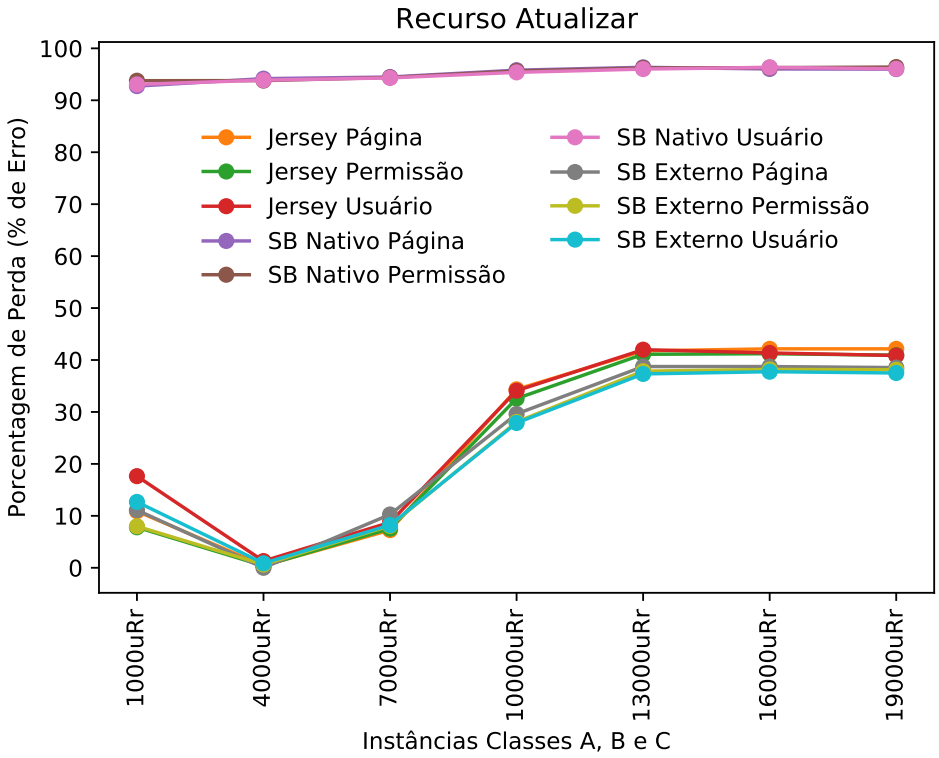


Figura 4.17: Resultados da porcentagem de perda de solicitações dos microsserviços no recurso de Atualizar.

4.10 Considerações Parciais

Este capítulo apresentou uma análise dos *frameworks Jersey* e *Spring Boot* em uma implementação que se assemelha ao sistema do Integra, utilizando recursos de virtualização do software *Docker*, e efetuando testes de carga e estresse para avaliar seu desempenho em um ambiente de produção real. Foram selecionadas algumas métricas como base para esta avaliação, com a finalidade de apoiar a escolha de uma tecnologia para a migração do sistema Integra para a arquitetura de microsserviços.

Observou-se nos testes de carga que o *framework Jersey* obteve melhor desempenho em relação ao *framework Spring Boot*. Pela análise efetuada anteriormente, pode-se observar que o *framework Jersey* obteve a menor latência média e tempo médio de resposta em todos os microsserviços testados. Este ainda obteve a maior taxa de transferência dentre as tecnologias do *framework Spring Boot* testadas, em todas as instâncias de teste, e menores valores nas linhas de 90% e 99%. Este *framework* somente obteve uma taxa de perda superior ao *Spring Boot* (com servidor externo) em poucas instâncias. Após o *framework Jersey*, a tecnologia que melhor apresentou resultados foi o *framework Spring*

Boot (com servidor externo). Este por sua vez, obteve valores de latência média e tempo médio de resposta próximas ao *framework Jersey*. Sua taxa de transferência foi relativamente menor, seus resultados sobre a linha de 90% e 99% quase semelhantes aos valores do *framework Jersey*, e obteve as menores taxas de perda de solicitações durante os testes. O *framework Spring Boot* (com servidor nativo) obteve os piores resultados, mesmo em relação ao mesmo *framework* com servidor externo. Nota-se através desta análise, o impacto do servidor embutido ao *framework* e sua capacidade de processamento em relação ao servidor externo, sendo o desempenho em um servidor externo superior ao servidor embutido.

Um fator que também possui um impacto importante, é a facilidade de desenvolvimento e a velocidade com que a aplicação pode sair do planejamento até sua entrada em produção. Neste aspecto, o *framework Spring Boot* possui o melhor resultado em comparação com o *Jersey*, devido à sua facilidade de desenvolvimento, implantação, e as diversas possibilidades de configurações que podem ser efetuadas, utilizando somente um arquivo de configuração dentro da aplicação.

5 Considerações Finais

5.1 Visão Geral

O presente trabalho apresentou um estudo de caso para avaliar o desempenho das tecnologias *Spring Boot* e *Jersey* aplicadas na arquitetura de microsserviços utilizando a tecnologia de virtualização *Docker*. Este trabalho foi desenvolvido com o objetivo de avaliar as tecnologias de desenvolvimento de microsserviços e ajudar na tomada de decisão sobre as tecnologias que melhor se adéquam ao desenvolvimento do novo sistema do Integra, seguindo seu ambiente de produção em condições normais e sua quantidade de acessos simulando períodos com alta quantidade de acessos. O antigo sistema do Integra, desenvolvido na arquitetura monolítica, possui a necessidade de migração para a arquitetura de microsserviços. Sendo assim, é necessário realizar uma avaliação das tecnologias que utilizem microsserviços para o desenvolvimento do sistema de forma a não somente cumprir todos os requisitos do sistema, mas também, avaliar qual o seu desempenho, já que será submetido ao acesso aos diversos estudantes e professores de toda a universidade.

A importância deste trabalho decorre da necessidade de efetuar testes de desempenho desde o âmbito de serviços, até no sistema por completo. Realizando o teste de desempenho em microsserviços, pode-se não somente verificar sua disponibilidade e validade das respostas, mas também seu comportamento quando submetido a altas cargas de trabalho e os limites das tecnologias em que foram desenvolvidos os microsserviços. Outra contribuição conceitual deste trabalho, decorre da separação de grupos de instâncias para efetuar o teste e a análise de seus resultados. Desta forma, alguns aspectos do teste podem ser observados desde seu início até o término de cada grupo.

5.2 Conclusões Gerais

Com base nos resultados obtidos, conclui-se que o *framework Jersey* apresentou as menores taxas de latência, maiores taxas de transferência, menores taxas de tempo médio

de resposta, uma taxa de perda relativamente pequena e tempos de solicitações menores nas linhas de 90% e 99%, possuindo o melhor desempenho. Observa-se também, que o *framework Spring Boot* com servidor externo possui valores de latência, taxas de transferência, tempo médio de resposta, taxa de perda e tempos de solicitações nas linhas de 90% e 99% próximos do *Jersey*. Assim, o *framework Spring Boot* com servidor nativo, possui o pior desempenho dentre os microsserviços avaliados.

5.3 Trabalhos Futuros

O novo sistema do Integra será desenvolvido na arquitetura de microsserviços e, para assegurar que o correto funcionamento dos microsserviços, ao serem implantados em produção, deve-se realizar os testes de integração e segurança. E para determinar o correto entendimento e utilização dos usuários, efetuar o teste de aceitação. Estes testes serão melhores aproveitados sendo efetuados sob a forma de testes automatizados. Para isto, poderão ser utilizadas as ferramentas de *Jenkins* para automação e ferramentas de teste como o próprio *JMeter*, abordado nas seções anteriores, ou também o *Gatling*, para o testes de desempenho dos microsserviços.

Como recurso para a escalabilidade automática dos microsserviços, pode ser utilizado o *Spring Cloud*, que possui recursos de registro automático e auto-descoberta em rede de microsserviços em execução. Seus metadados são registrados por um serviço no momento em que os microsserviços estiverem aceitando trafegar informações. No momento em que são registrados, os microsserviços já poderão ser consumidos.

No contexto de microsserviços, há também a necessidade de efetuar testes distribuídos. Estes testes, em sua maioria, são realizados no âmbito de serviços, sendo verificada sua disponibilidade e a validade dos dados de resposta. Neste caso, pode-se verificar também seu desempenho realizando o teste de forma descentralizada.

Durante o desenvolvimento deste trabalho foram observadas algumas limitações quanto ao servidor embutido na tecnologia *Spring Boot*. Por ser um servidor nativo e embutido nesta tecnologia, para obter uma resposta que possua acuracidade e seja consistente com seu funcionamento dentro da tecnologia, é necessário um estudo com maior profundidade neste aspecto, que poderia trazer resultados melhores na comparação

de desempenho. Os microsserviços podem ainda serem desenvolvidos e configurados de forma a efetuar múltiplas conexões ao banco de dados. Desta forma o tempo em que o microsserviço leva para efetuar suas operações de busca, inserção, atualização e remoção no banco de dados, é reduzida. Geralmente, ao desenvolver os microsserviços utilizando o *framework Spring*, utiliza-se também o *Spring JPA* para efetuar as múltiplas conexões ao banco de dados.

Bibliografia

ACEVEDO, C. A. J.; JORGE, J. P. G. y; PATIÑO, I. R. Methodology to transform a monolithic software into a microservice architecture. In: IEEE. *Software Process Improvement (CIMPS), 2017 6th International Conference on*. [S.l.], 2017. p. 1–6.

CAMARGO, A. de et al. An architecture to automate performance tests on microservices. In: ACM. *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. [S.l.], 2016. p. 422–429.

CARNELL, J. *Spring Microservices in Action*. [S.l.]: Manning Publications Company, 2017.

CHUNG, M. T. et al. Using docker in high performance computing applications. In: IEEE. *Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on*. [S.l.], 2016. p. 52–57.

DEVORE, J. L. *Probability and Statistics for Engineering and the Sciences*. [S.l.]: Cengage learning, 2011.

DOCKER 101: Introduction to Docker webinar recap. 2017. Disponível em: ⟨https://blog.docker.com/2017/08/docker-101-introduction-docker-webinar-recap/⟩.

FENTON, N.; BIEMAN, J. *Software metrics: a rigorous and practical approach*. [S.l.]: CRC press, 2014.

FOWLER, M.; LEWIS, J. *Microservices a definition of this new architectural term (2014)*. 2018. Disponível em: ⟨https://martinfowler.com/articles/microservices.html⟩.

GRANCHELLI, G. et al. Towards recovering the software architecture of microservice-based systems. In: IEEE. *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. [S.l.], 2017. p. 46–53.

GUO, D. et al. Microservices architecture based cloudware deployment platform for service computing. In: IEEE. *Service-Oriented System Engineering (SOSE), 2016 IEEE Symposium on*. [S.l.], 2016. p. 358–363.

GUTIERREZ, F. *Pro Spring Boot*. [S.l.]: Springer, 2016.

HAN, H. et al. A restful approach to the management of cloud infrastructure. In: IEEE. *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*. [S.l.], 2009. p. 139–142.

HASSELBRING, W.; STEINACKER, G. Microservice architectures for scalability, agility and reliability in e-commerce. In: IEEE. *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. [S.l.], 2017. p. 243–246.

JARAMILLO, D.; NGUYEN, D. V.; SMART, R. Leveraging microservices architecture by using docker technology. In: IEEE. *SoutheastCon, 2016*. [S.l.], 2016. p. 1–5.

JMETER, A. Apache software foundation, disponível em: http://jmeter. apache. org. *Access in: September, 24th*, 2018.

JMETER, A. Web page at https://jmeter.apache.org/usermanual/glossary.html. *Date of last access: September*, v. 18, 2018.

JOY, A. M. Performance comparison between linux containers and virtual machines. In: IEEE. *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. [S.l.], 2015. p. 342–346.

MARTENS, A. On compatibility of web services. *Petri Net Newsletter*, v. 65, n. 12-20, p. 100, 2003.

MAVRIDIS, I.; KARATZA, H. Performance and overhead study of containers running on top of virtual machines. In: IEEE. *Business Informatics (CBI), 2017 IEEE 19th Conference on*. [S.l.], 2017. v. 2, p. 32–38.

MEDVIDOVIC, N.; TAYLOR, R. N. Software architecture: foundations, theory, and practice. In: ACM. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. [S.l.], 2010. p. 471–472.

MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, Belltown Media, Houston, TX, v. 2014, n. 239, mar. 2014. ISSN 1075-3583. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=2600239.2600241⟩.

NAIK, K.; TRIPATHY, P. *Software testing and quality assurance: theory and practice*. [S.l.]: John Wiley & Sons, 2011.

NEWMAN, S. *Building microservices: designing fine-grained systems*. [S.l.]: "O'Reilly Media, Inc.", 2015.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016.

PURUSHOTHAMAN, J. *RESTful Java Web Services*. [S.l.]: Packt Publishing Ltd, 2015.

REPASI, T. Software testing-state of the art and current research challanges. In: IEEE. *Applied Computational Intelligence and Informatics, 2009. SACI'09. 5th International Symposium on*. [S.l.], 2009. p. 47–50.

ROSS, S. M. *Introduction to probability and statistics for engineers and scientists*. [S.l.]: Elsevier, 2004.

SALVADORI, I.; SIQUEIRA, F. A maturity model for semantic restful web apis. In: IEEE. *Web Services (ICWS), 2015 IEEE International Conference on*. [S.l.], 2015. p. 703–710.

SANTOS, E. A. et al. How does docker affect energy consumption? evaluating workloads in and out of docker containers. *Journal of Systems and Software*, Elsevier, 2018.

SHAW, M.; CLEMENTS, P. Toward boxology: Preliminary classification of architectural styles. In: ACM. *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*. [S.l.], 1996. p. 50–54.

SIM, S. E. A small social history of software architecture. In: IEEE. *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. [S.l.], 2005. p. 341–344.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151.

SOUZA, R. R. de et al. Avaliação de desempenho entre web services soap e restful com o uso do apache cxf. 06 2013.

STIRBU, V. A restful architecture for adaptive and multi-device application sharing. In: ACM. *Proceedings of the First International Workshop on RESTful Design*. [S.l.], 2010. p. 62–65.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. *Software Architecture: Foundations, Theory, and Practice. 2009*. [S.l.]: Wiley, 2009.

UEDA, T.; NAKAIKE, T.; OHARA, M. Workload characterization for microservices. In: IEEE. *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. [S.l.], 2016. p. 1–10.

VANDIKAS, K.; TSIATSIS, V. Microservices in iot clouds. In: IEEE. *Cloudification of the Internet of Things (CIoT)*. [S.l.], 2016. p. 1–6.

VIGGIATO, M. et al. Microservices in practice: A survey study. *arXiv preprint arXiv:1808.04836*, 2018.

VILLAMIZAR, M. et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. *Computing Colombian Conference (10CCC), 2015 10th*. [S.l.], 2015. p. 583–590.

VILLAMIZAR, M. et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. [S.l.], 2016. p. 179–182.

WEBB, P. e. a. *How-to guides*. 2017. Disponível em: ⟨https://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-web-servers.html⟩.

WEBBER, J.; PARASTATIDIS, S.; ROBINSON, I. *REST in practice: Hypermedia and systems architecture*. [S.l.]: "O'Reilly Media, Inc.", 2010.