UNIVERSIDADE FEDERAL DE SÃO CARLOS
ENGENHARIA DE COMPUTAÇÃO

MATHEUS SANTOS DE ALMEIDA

UMA ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE DIFERENTES TECNOLOGIAS DE EXECUÇÃO DE APLICAÇÕES WEB DO LADO DO SERVIDOR

SÃO CARLOS
2020

MATHEUS SANTOS DE ALMEIDA

UMA ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE DIFERENTES TECNOLOGIAS DE EXECUÇÃO DE APLICAÇÕES WEB DO LADO DO SERVIDOR

Monografia apresentada ao Departamento de Computação da Universidade Federal de São Carlos como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Daniel Lucrédio

SÃO CARLOS
2020

UNIVERSIDADE FEDERAL DE SÃO CARLOS
ENGENHARIA DE COMPUTAÇÃO

MATHEUS SANTOS DE ALMEIDA

UMA ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE DIFERENTES TECNOLOGIAS DE EXECUÇÃO DE APLICAÇÕES WEB DO LADO DO SERVIDOR

Monografia apresentada ao Departamento de Computação da Universidade Federal de São Carlos como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Banca Examinadora

_____
Prof Dr. Daniel Lucrédio
(Orientador – DC UFSCar)

_____
Prof Dr. Auri Marcelo Rizzo Vincenzi

_____
Prof Dr. Delano Medeiros Beder

RESUMO

Este trabalho compara o desempenho de aplicações nativas com aplicações executadas em sistemas de execução. As aplicações foram feitas com dois frameworks para a construção de aplicações web do lado do servidor com Java: Spring Boot e Quarkus. Foi construída uma aplicação para cada framework. As aplicações foram compiladas em código nativo com o GraalVM e em pacotes *jar* para serem executadas pela JVM. O critério adotado para avaliar o desempenho foi o tempo de resposta médio do servidor para o cliente em cada cenário criado, também foi avaliado o tempo de inicialização de cada aplicação. Ambas as aplicações funcionam como APIs que acessam um banco de dados, também foi criado um método para fazer a ordenação de um vetor de números passado no corpo da requisição HTTP. Foram usadas duas máquinas como cliente e servidor alternadamente, o servidor executava a aplicação e o cliente acessava o servidor e registrava as estatísticas dos resultados. As tabelas dos resultados registrados estão expostas neste trabalho. Foi constatado que na maioria dos casos a aplicação em Quarkus teve uma diminuição no tempo médio de resposta. Para o Spring Boot não houve confirmação de ganho de desempenho para a aplicação nativa. As aplicações também foram encapsuladas em Docker para testar o desempenho nessa circunstância, não foi observada uma piora no desempenho da versão encapsulada, embora este resultado seja inconclusivo.

Palavras-chave: GraalVM. Java. Docker. Tempo de resposta. Spring Boot. Quarkus.

ABSTRACT

This work compares the performance of native applications against applications to be executed by execution systems. The applications were built with two frameworks meant for server-side application development with Java: Spring Boot and Quarkus. It was built an application for each framework. Both applications were compiled to native code with GraalVM and to jar packages to be executed by JVM. The adopted criterion to check the performance was the average response time from server to client for each created scenario, it was also evaluated the initialization time of each application. Both applications work like APIs that access a database. It was also created a method to sort a vector of numbers passed by the HTTP request body. It was used two machines to work as server and client alternately, the server executed the application while the client accessed the server and registered the results statistics. The tables of registered results are shown in this work. It was noted that in most of the tested cases the Quarkus native application had a reduction in average response time. To Spring Boot there wasn't any confirmation of performance improvement to the native application. The applications were also encapsulated in Docker to observe how they perform in this circumstance. There wasn't a worsening observed in the encapsulated version performance, although this result isn't conclusive.

Key-words: GraalVM. Java. Docker. Response time. Spring Boot. Quarkus.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 VISÃO GERAL

Os serviços de computação em nuvem sob demanda tornam-se cada vez mais populares e substituem servidores tradicionais mantidos pelos próprios responsáveis pelo desenvolvimento de uma aplicação. Junto com essa tendência, veio a demanda pela virtualização de sistemas operacionais. Para Verma e Dhawan (2017, p. 1, tradução nossa) “a tecnologia de virtualização a nível de sistema operacional, ou *containers*, como ela é conhecida, representa a próxima geração de virtualização leve e é representada principalmente pelo Docker”. Um dos motivos para que isso aconteça é o poder de isolamento que a virtualização oferece para as aplicações poderem utilizar o mesmo sistema operacional base (VERMA; DHAWAN, 2017, p. 1). Embora haja diversas implementações comerciais de *containers,* o Docker tem uma popularidade muito maior que as demais por oferecer uma maneira fácil de encapsular e distribuir aplicações, este fato motivou várias ofertas de serviços de computação em nuvem baseadas em Docker pela Microsoft, Google, IBM entre outros (VERMA; DHAWAN, 2017, p. 1).

Para aplicações em produção

> “[…] é desejável que se otimize o tamanho de imagens Docker, para obter um tempo de implantação mais rápido logo após o término da codificação da aplicação, o que é muito importante caso a aplicação precise escalar ou atender a um pico de tráfego” (MOELLERING, 2019, acesso em 2020, tradução nossa).

Em uma listagem das linguagens mais usadas em 2014 baseada em métricas fornecidas pelo GitHub, as dez primeiras foram Java, C, C++, C#, Python, JavaScript, PHP, Ruby, SQL e MATLAB nesta ordem (CASS, 2014). Exceto por C++, C, e MATLAB, todas as linguagens listadas são amplamente utilizadas em desenvolvimento de aplicações Web do lado do servidor. JavaScript e Java, em particular, do lado do cliente também. Essas linguagens têm em comum o fato de serem linguagens interpretadas ou parcialmente interpretadas, como o Java, que tem um estágio de compilação do código para *bytecode* antes que ele seja interpretado (LEUN, 2017, p. 8).

O processo de interpretação de uma linguagem ocorre por meio de uma camada de software chamada Máquina Virtual ou Sistema de execução (LEUN, 2017, p. 8). A linguagem Java juntamente com seu sistema de execução, a Java Virtual Machine (JVM) foram lançadas em 1995 (LEUN, 2017, p. 8), uma época em que os computadores pessoais tinham processadores com apenas um núcleo e não tinham gigabytes de memória (LEUN, 2017, p. 10). Desde então, a linguagem provou-se totalmente adaptável aos progressos de hardware e a novos paradigmas que surgiram desde então. Passou a dar suporte aos processadores com mais de um núcleo com a inclusão de threads e adicionou novas classes para facilitar a programação concorrente (LEUN, 2017, p. 10) só para citar um exemplo de adaptação. Outra característica que tornou a JVM popular foi o fato de outras linguagens além do Java, como Kotlin, também poderem usar o formato *bytecode* (LEUN, 2017 p. 25).

Embora seja verdade que as aplicações baseadas em Java já foram consideradas lentas e pouco otimizadas em relação ao consumo de memória, a linguagem progrediu nesses aspectos ao longo dos anos (LEUN, 2017 p. 1). Mas, ainda assim, surgiu um fato novo na história da JVM: em 2019, a Oracle lançou o GraalVM para a comunidade (GRAALVM, 2020)[1]. O GraalVM pode operar como um sistema de execução (GRAALVM, 2020)[2], como as implementações que já existiam desde então no mercado, mas também pode converter uma aplicação feita em Java ou em outras linguagens que usam a JVM, como Kotlin, por exemplo, em uma aplicação nativa por meio de sua ferramenta de geração de imagens nativas, o GraalVM Native.

Para Barrett et al (2017, citado por WIMMER, 2019, p. 2), uma aplicação nativa tende a se comportar de maneira diferente de uma aplicação feita com uma linguagem interpretada: as aplicações em máquinas virtuais podem ter um tempo de inicialização imprevisível.

1.2 OBJETIVOS

Este trabalho tem como propósito avaliar a cena dos ambientes de produção de

1 GRAALVM. **Getting started**. Disponível em: < https://www.graalvm.org/docs/getting-started/ >. Acesso em: 18 out. 2020

2 GRAALVM. **Release notes**. Disponível em: < https://www.graalvm.org/docs/release-notes/19_0/ >. Acesso em: 18 out. 2020

aplicações web do lado do servidor fazendo um escrutínio de questões difíceis de serem respondidas categoricamente, mas que possam pelo menos ser esclarecidas em alguns aspectos. Para começar, vamos levar em conta alguns aspectos já citados na Seção 1.1 e o que podemos inferir até aqui:

- Java está entre as linguagens de programação mais utilizadas dentre todas as linguagens de programação existentes e é usada para o desenvolvimento de aplicações do lado do servidor. Portanto, sua escolha para a realização deste trabalho está justificada;
- Além de Java, outras cinco linguagens de programação que estão entre as dez mais usadas, segundo a lista do GitHub de 2014, também são usadas com a mesma finalidade de construir aplicações do lado do servidor, embora algumas dessas linguagens possam ser usadas para outras finalidades, esse fato mostra o grande tamanho do segmento de aplicações do lado do servidor atualmente e dá a dimensão da importância de lançar mais esclarecimentos a esse segmento;
- A adoção da virtualização de sistemas operacionais com Docker tornou-se praticamente o padrão nos serviços de computação em nuvem oferecidos pelas principais empresas do ramo, portanto é válido testar qual é o impacto do Docker na execução da aplicação;
- Tornar os *containers* das virtualizações mais leves faz com que a entrega e a escalabilidade da aplicação sejam otimizadas e uma maneira de fazer isso é usar aplicações nativas compiladas pelo GraalVM ao invés de aplicações que precisem da instalação de máquinas virtuais.

Demonstrada a relevância de cada elemento que se pretende avaliar neste trabalho, e considerando o fato de que o GraalVM facilita a escalabilidade, pois torna as aplicações em Docker mais leves, o propósito deste trabalho é criar testes com aplicações construídas com padrões de projeto usuais de aplicações em produção e que avaliem principalmente o impacto que cada um desses elementos traz à taxa de transferência de informação. Ou seja, responder, ou pelo menos esclarecer de alguma forma, as seguintes perguntas:

- Como as aplicações executadas do lado do servidor reagem a um grande número de requisições HTTP (*Hyper Text Transfer Protocol*) simultâneas?
- Qual é o impacto de um aumento na quantidade de bytes trafegados na requisição para cada tecnologia?

- Qual é o impacto quando são adotadas aplicações compiladas no lugar de aplicações executadas por sistemas de execução?

Foi avaliado o impacto que a introdução ou a variação de cada elemento dessas perguntas traz aos resultados dos testes. Foi fornecida uma análise comparativa de desempenho de dois frameworks para a construção de aplicações web do lado do servidor com a linguagem Java: Quarkus (QUARKUS, 2020)[3] e Spring Boot (SPRING, 2020)[4].

A ideia é construir um ambiente similar a um ambiente de produção para cada aplicação e ver como o conjunto de escolhas feitas em cada caso desempenha em relação aos outros cenários aqui construídos. Este trabalho não se propõe a analisar funcionalidades específicas de cada linguagem ou framework isoladamente, como, por exemplo, isolar um aspecto específico como a serialização de JSON (JavaScript *Object Notation*) para comparar a eficiência que cada biblioteca ou *framework* tem; o projeto TechEmpower (visitado em 2020) propõe-se a fazer isso e é explicado na Seção 1.3.

O fato de o GraalVM não dar suporte a todas as funcionalidades da linguagem Java é um fator impeditivo para que todos os frameworks de Java possam dar suporte definitivo ao GraalVM. No momento em que este trabalho foi feito, o Quarkus dava suporte definitivo ao GraalVM (QUARKUS, 2020)[5], o suporte do Spring Boot estava em fase experimental (SPRING, 2020)[6].

1.3 TRABALHOS RELACIONADOS

O trabalho de Marr, Daloze e Mössenböck (2016) leva em conta as linguagens Java, JavaScript, Ruby, Crystal, Newspeak, e Smalltalk e tem como objetivo comparar o desempenho de diferentes linguagens quando compiladas pelo mesmo compilador. Marr, Daloze e Mössenböck (2016, p. 122) afirmam que tomaram conhecimento de poucos projetos com uma abordagem para comparar desempenho entre linguagens sistematicamente. Apenas um dos projetos por eles citados é de

3 QUARKUS. **Supersonic Subatomic Java**. Disponível em: < https://quarkus.io/ >. Acesso em: 22 dez. 2020.
4 SPRING. **Overview**. Disponível em: < https://spring.io/projects/spring-boot >. Acesso em: 22 dez. 2020.
5 QUARKUS. **Quarkus - Building a Native Executable**. Disponível em: < https://quarkus.io/guides/building-native-image >. Acesso em: 22 dez. 2020.
6 SPRING. **Spring Native for GraalVM 0.8.3 available now**. Disponível em: < https://spring.io/blog/2020/11/23/spring-native-for-graalvm-0-8-3-available-now >. Acesso em: 22 dez. 2020

artigo acadêmico.

Outro trabalho relevante no domínio do tema desta monografia é o projeto TechEmpower, citado por Marr, Daloze e Mössenböck (2016, p. 122). Segundo o próprio website, TechEmpower (visitado em 2020), o projeto é uma extensa coleção de testes de desempenho que executam tarefas fundamentais como a serialização de JSON, acesso a banco de dados e a geração de HTML do lado do servidor, para citar três exemplos.

O trabalho de Larsson, R. (2020) compara aplicações construídas com o GraalVM com outros JDKs (Java *Development Kit*), mas sem usar frameworks de desenvolvimento como o Quarkus e sem considerar o desempenho das aplicações do ponto de vista da relação cliente-servidor.

> "[...] comparamos o desempenho das versões community edition e enterprise edition do GraalVM ao OpenJDK e OracleJDK, usando Java 8 e Java 11 […]. Descobrimos que o desempenho dos diferentes JDKs variam significativamente dependendo do teste, o que torna difícil fazer qualquer conclusão definitiva." Larsson, R. (2020, p. 1, tradução nossa).

O desafio apontado por Larsson (2020) na variação dos tempos de medição foi notado neste trabalho também, por isso optamos por fazer testes com várias iterações, ou, quando as limitações de hardware impossibilitaram mais de uma iteração, pelo menos manter uma grande quantidade de requisições simultâneas. Esse aspecto será melhor abordado na Seção 3.7.

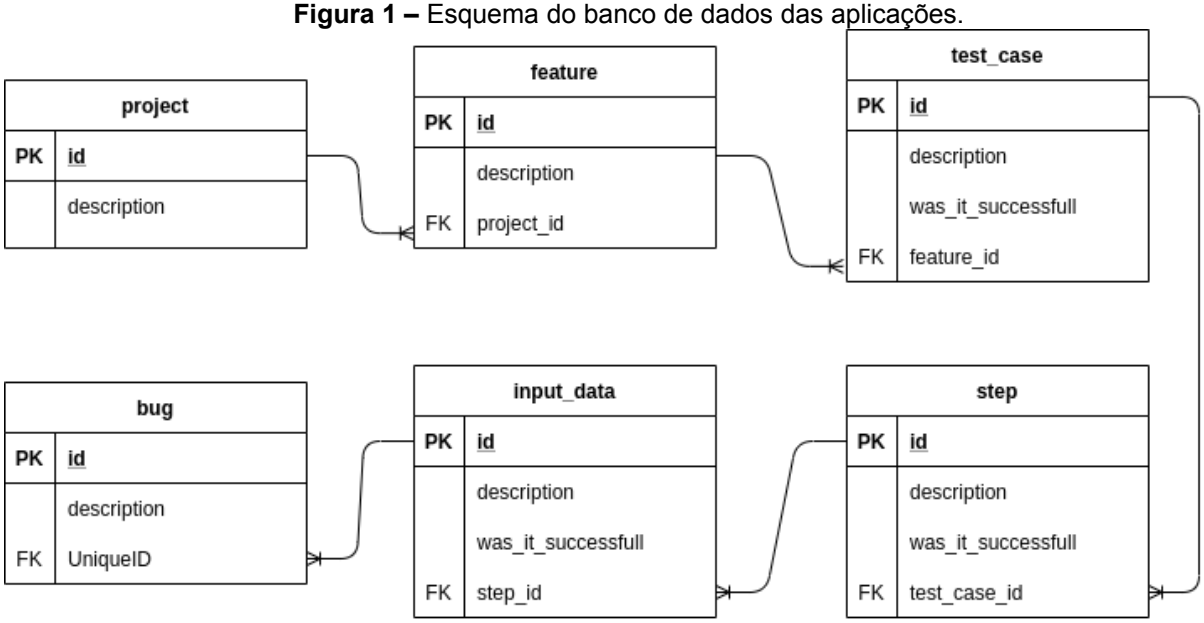
2 METODOLOGIA UTILIZADA

2.1 CONSTRUÇÃO DAS APLICAÇÕES

Foram criadas duas aplicações para realizar medições de desempenho. A primeira, com o framework Spring Boot e a linguagem Java. A segunda com o framework Quarkus e também com a linguagem Java. Ambas as aplicações podem ser encontradas em Almeida, M. S. D. (2020). Todas as aplicações, sejam em versão para ser executada pela JVM, sejam em código nativo, foram compiladas com o GraalVM. A JVM utilizada para executar as aplicações foi a Azul Zulu JDK, exceto no Docker, onde foi usado o próprio GraalVM para executar a JVM. Essa escolha se deu pelo fato de o Azul Zulu ser um sistema baseado no OpenJDK e por ser uma implementação da JVM bem aceita pela comunidade (AZUL, 2020), mas poderia ter sido escolhida outra versão da JVM, não foi notada uma diferença significativa de desempenho com outras implementações da JVM, inclusive com a própria JVM do GraalVM.

As duas aplicações comunicam-se com o banco de dados representado na Figura 1. Para acessar o banco de dados, ambas as aplicações funcionam como uma API (*Application Program Interface*) com um método HTTP do tipo GET para consulta e um método HTTP do tipo POST para fazer uma inserção de um conjunto de valores em cada tabela, totalizando assim seis métodos HTTP GET e seis HTTP POST. As médias gerais para os tempos de resposta para cada cenário de todas essas requisições estão nas Tabelas 3, 4, 10 e 11. A escolha de um esquema com mais de uma tabela foi feita para simular um pequeno sistema em produção, aplicações *front-end* geralmente precisam fazer várias requisições e buscar dados de diferentes APIs para preencher as telas.

Figura 1 – Esquema do banco de dados das aplicações.



Fonte: Elaborada pelo autor.

O sistema de gerenciamento de banco de dados utilizado para o sistema representado na Figura 1 foi o MySQL versão 8. Em todas as aplicações foi utilizado o mesmo banco de dados, porque variar o sistema de gerenciamento de banco de dados pode acarretar em modificações no desempenho da aplicação e este trabalho tem por objetivo apenas avaliar as linguagens usadas em conjunto com os *frameworks* seguindo padrões comumente utilizados em sistemas em produção. A escolha do MySQL 8 não seguiu nenhum critério especial, trata-se de um sistema de gerenciamento de banco de dados amplamente utilizado, os experimentos poderiam ter usado o PostgreSQL, por exemplo, que ainda serviriam ao mesmo propósito.

Foi criado um método HTTP para os dois projetos que faz a ordenação de elementos de um vetor passado pelo corpo da requisição para testar o desempenho da aplicação quando submetida apenas a operações lógicas, aritméticas e de processamento de *strings* sem que ela acesse o banco de dados. Nesse cenário, cada usuário virtual faz quatro requisições HTTP POST, cada uma com três, 300, 3 mil e 30 mil elementos respectivamente.

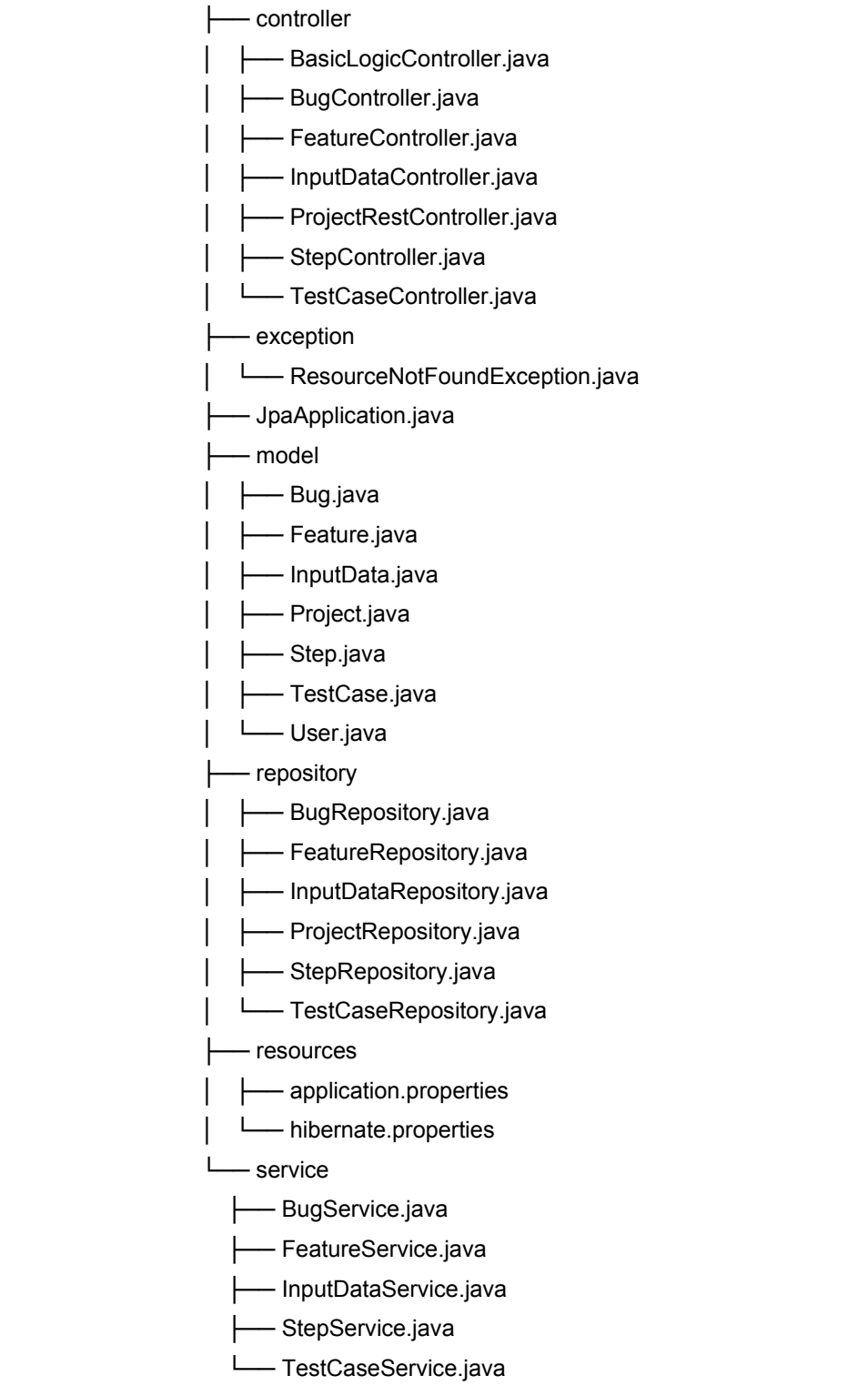
Tanto na aplicação feita com Spring Boot quanto na aplicação feita com Quarkus, foi usado o método *sort* da biblioteca *java.util.Arrays.* Na documentação da Oracle (2020) é especificado que não é obrigatório o uso de um algoritmo específico de ordenação para a implementação desse método, porém é especificado que o algoritmo a ser utilizado deve ser estável, como o *Merge sort*, por exemplo.

Nas Figuras 2 e 3 são mostradas as estruturas de classes dos projetos em Spring Boot e em Quarkus. A classe JpaApplication é a classe principal, por onde começa a execução da aplicação. Ambos os projetos tem os pacotes “model” e “controller”, o primeiro contém classes que representam as tabelas da Figura 1, o segundo tem métodos com as assinaturas dos *endpoints* das requisições HTTP.

Apenas para o projeto feito com Spring Boot, foram criados os pacotes “repository” e “service”, o primeiro contém implementações da interface JpaRepository da biblioteca *org.springframework.data.jpa.repository*, que serve como interface para acessar as classes do banco de dados quando seus métodos são implementados, como por exemplo *findBy<X>Id*, onde <X> é substituído pela classe pai. A classe “service” contém os métodos com a lógica das requisições HTTP, esse métodos instam os métodos das classe do pacote “repository”.

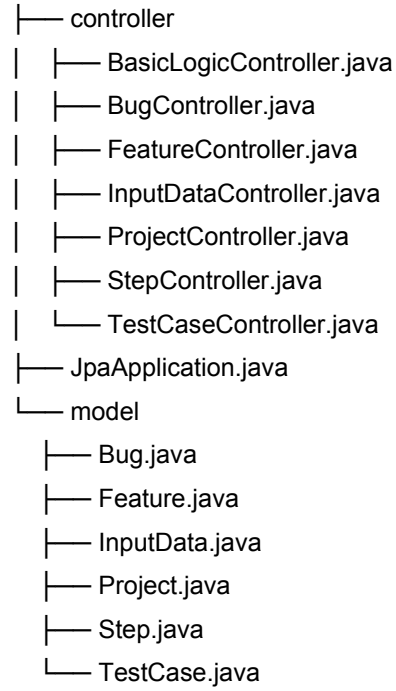
Para o projeto feito com Quarkus, o acesso ao banco de dados foi feito através da classe javax.persistence.NamedQuery, que permite fazer consultas em SQL dentro do código.

Figura 2 – Classes criadas no projeto feito com Spring Boot.



Fonte: Elaborada pelo autor.

Figura 3 – Classes criadas no projeto com Quarkus.



Fonte: Elaborada pelo autor.

Todas as requisições HTTP feitas pelas aplicações em Quarkus e em Spring Boot foram passadas com um JSON no corpo. E em ambos os casos foi usada a mesma biblioteca para a serialização, que é o processo de converter cada chave do JSON com seu respectivo valor em variáveis da classe Java que representa a tabela do banco de dados e para a desserialização, que é o processo inverso, também. A biblioteca utilizada foi a FasterXML/Jackson, que faz o processo de serialização e desserialização com o uso de uma propriedade da linguagem Java: a “reflexão” (FASTERXML/JACKSON, 2020).

> Reflexão é uma funcionalidade da linguagem de programação Java que permite examinar ou fazer uma “introspecção” do programa em si mesmo e manipular propriedades internas do programa. Por exemplo, a possibilidade de uma classe Java obter nomes de todos os seus membros e mostrá-los. (MCCLUSKEY, 1998, acesso em 2020, tradução nossa).

Durante a confecção das aplicações em Java, todas as bibliotecas que causaram erro de execução na aplicação nativa devido à reflexão foram incluídas em um arquivo de configuração em cada projeto.

Também foram feitos testes com as aplicações em Spring Boot e Quarkus sendo executadas por meio de um ambiente virtualizado com Docker. Para esses testes,

foi usado apenas o método HTTP que faz a ordenação do vetor passado pelo corpo da requisição sem que haja acesso ao banco de dados. Nesse cenário, foram removidas as classes com os métodos que acessam o banco de dados para ver qual seria o impacto na taxa de transferência das aplicações com uma configuração mais leve, não houve evidência de que a simples remoção das classes, que deixou a aplicação mais leve, interferiu de alguma forma na taxa de transferência.

A escolha de todas essas especificações foi feita por serem cenários viáveis ou comumente usados para ambientes de produção, exceto pelo caso em que é feita a ordenação do vetor, na qual o propósito é avaliar o desempenho dos frameworks sem o gargalo do acesso ao banco de dados.

2.2 COMPARAÇÃO DE DESEMPENHO ENTRE AS APLICAÇÕES

As duas aplicações foram testadas em duas máquinas distintas que funcionaram como servidor. Enquanto uma máquina operava como servidor e executava a aplicação e o banco de dados, a outra máquina funcionava como cliente e executava as requisições à máquina que funcionava como servidor.

Para a máquina que funcionava como cliente executar várias requisições por segundo e gerar relatórios de desempenho, foi utilizado o Apache JMeter. Para cada cenário de teste, foi especificado um número de usuários virtuais (*threads*), um tempo de inicialização para todos os usuários virtuais (*ramp up*) e uma quantidade de iterações (APACHE JMETER, 2020).

Um fato importante é que todos os testes começam com todas as tabelas do banco de dados vazias e elas são preenchidas conforme os POST são chamados. Portanto, as primeiras requisições GET são executadas mais rapidamente e trazem menos dados, enquanto as últimas demoram mais porque trazem mais dados.

Não houve uma tendência divergente entre as aplicações com relação a proporção do tempo de cada requisição, por isso foi mostrado só a média geral de cada cenário nas Tabelas 3, 4, 10 e 11. Esse cenário misto, com várias requisições de diferentes naturezas foi criado por ser um cenário similar a um cenário de uma aplicação em produção. Portanto, nos casos em que há acesso ao banco de dados, a quantidade total de requisições executadas em cada cenário é:

$$T = U \times R \times I$$

Onde T é o total de requisições feitas, U é a quantidade de usuários virtuais, I é a

quantidade de iterações e R é a quantidade de requisições executadas por um usuário virtual.

As tabelas com o resultado dos testes são apresentadas com os valores médio, mediana e máximo tempo de resposta, taxa de transferência em KB/s enviados e recebidos, embora no decorrer do trabalho seja discutido somente o tempo de resposta médio. Isso ocorre porque é esperado que esses valores sejam diretamente proporcionais à média, no caso da mediana, máximo tempo de resposta – ou inversamente proporcionais, no caso da taxa de transferência, portanto corroboram com a integridade da média.

Para padrões detectados entre medições correlatas, foram feitos cálculos das médias para podermos aferir como a adição ou a variação de algum fator afetou a aplicação. Todas as médias foram apresentadas com intervalo de confiança para atestar suas validades. Para tal, foi usada a distribuição t de Student, porque foram usadas apenas amostras consideradas pequenas no presente trabalho.

> "Quando o tamanho da amostra é grande, n > 30, a tabela t de Student e a distribuição normal z são idênticas […]. A vantagem da distribuição t de Student é que ela faz uma compensação para amostras de tamanho pequeno esticando a calda da distribuição de probabilidade." (PAULSON, 2008, p. 10, tradução nossa).

Seguem as fórmulas utilizadas para calcular o desvio padrão da média seguindo a distribuição t de Student:

A média simples, $\bar{x}$, dos valores de uma amostra:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Onde n é o tamanho da amostra.

O desvio padrão s é para essa média é:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}.$$

Por fim, temos o valor C, que é a média da nossa amostra com a margem obtida a partir do intervalo de confiança:

$$c = \bar{x} \pm z^* \frac{s}{\sqrt{n}}$$

Onde z^* é o valor da tabela t de Student, que é a mais usada com amostras pequenas, quando n < 30 , para o intervalo de confiança de 95%.

Neste trabalho, sempre que é calculada uma média, é mostrado o valor c.

Tabela 1 – Tabela t de Student para um intervalo de confiança de 95%.

Graus de liberdade	Valor
1	12,71
2	4,3
3	3,18
4	2,78
5	2,57
6	2,45
7	2,37
8	2,31
9	2,26

Fonte: Adaptada de PAULSON (2008, p. 164)

A partir da Tabela 1, o valor de z^* é escolhido em função do grau de liberdade da quantidade de amostras n, o grau de liberdade é sempre igual a n – 1. Por exemplo, para o caso em que temos 4 amostras, o grau de liberdade é 3, portanto escolhemos z^* = 3,18.

2.3 VERSÕES DOS ARTEFATOS UTILIZADOS NESTE TRABALHO

Os nomes das versões apresentadas na Tabela 2 foram obtidos a partir do comando “<nome do artefato> --version” executado via terminal, ou na pasta onde os programas estão instalados ou ainda no arquivo “pom.xml”, presente na raiz da pasta de cada projeto, que serve de base para para o Maven gerenciar as versões dos pacotes.

Tabela 2 – Versões dos artefatos utilizados neste trabalho

Artefato	**Versão**
GraalVM	20.2.0.r11
Azul Zulu JDK	11.0.9
Spring Boot	2.4.0.M3
Quarkus	1.8.2.Final
Apache JMeter	5.3
Docker (MacOS)	2.5.0.1
MySQL	8.0.20
Maven	3.6.3

Fonte: Elaborada pelo autor.

3 ANÁLISE DE DESEMPENHO

Neste capítulo são apresentados os resultados das medições de desempenho das aplicações em cada cenário com análises dos resultados e correlações encontradas entre os resultados.

3.2 TESTES FEITOS NO MAC MINI

Máquina utilizada: Apple Mac Mini (Late 2012) com 16GB de memória DDR3 1600MHz, SSD 240GB, processador Intel Core i5 dual core 2,5GHz, sistema operacional MacOS 10.15.3 Catalina.

Para o primeiro cenário, Tabelas 3 e 4, foram incluídas as médias gerais de todas as requisições GET e POST executadas por todos os usuários virtuais, porque, como já foi dito anteriormente, ao avaliar as requisições individualmente, não foram encontradas tendências particulares que justificassem mostrá-las aqui. O total de requisições feitas é verificado pele a fórmula T = U x R x I mostrada na Seção 2.2. Por exemplo, para a Tabela 3, T = 100 usuários virtuais x 12 requisições por usuário x 10 iterações = 12000.

Tabela 3 – Primeiro cenário: 100 usuários virtuais, tempo de inicialização igual a 10s e 10 iterações. Cada usuário virtual executa seis requisições HTTP GET e seis HTTP POST.

Tecnologias utilizadas para construir a aplicação	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)		Requisições mal sucedidas
		Médio	Mediana	Máximo		Recebido	Enviado	
Spring Boot – *jar*	12000	492.46	316.00	11432	163.46	2664.65	29.72	0
Spring Boot – nativo	12000	413.16	293.00	14382	157.63	2562.52	28.66	0
Quarkus – *jar*	12000	586.00	334.00	15733	142.49	2305.86	25.90	0
Quarkus – nativo	12000	431.43	281.00	12127	156.57	2521.11	28.47	0

Fonte: Elaborada pelo autor com base nos relatórios do teste feito pelo JMeter.

No cenário da Tabela 3, observamos como a aplicação se comporta com cada usuário virtual fazendo seis requisições GET e seis requisições POST ao banco de dados. Com 100 usuários simultâneos iniciados a cada dez segundos, observamos uma vantagem das aplicações nativas em relação às aplicações compiladas no formato *jar* executadas pela JVM Azul Zulu. O tempo médio de resposta das aplicações nativas foi menor (um tempo de resposta menor é o desejado) e as taxas

de envio e recebimento de dados pela rede foram maiores.

Tabela 4 – Segundo cenário: 1000 usuários virtuais, tempo de inicialização igual a 10s e 2 iterações. Cada usuário virtual executa seis requisições HTTP GET e seis HTTP POST.

Tecnologias utilizadas para construir a aplicação	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)		Requisições mal sucedidas
		Médio	Mediana	Máximo		Recebido	Enviado	
Spring Boot – *jar*	24000	12615.58	9293.50	384399	56.32	1890.50	10.24	5
Spring Boot – nativo	24000	10976.35	7496.50	313830	61.80	2036.05	11.23	2
Quarkus – *jar*	24000	12901.63	4111.50	391058	49.85	1645.20	9.06	2
Quarkus – nativo	24000	11877.59	4414.00	403443	53.14	1726.58	9.66	4

Fonte: Elaborada pelo autor com base nos relatórios dos testes feitos pelo JMeter.

No cenário da Tabela 4, com mil usuários simultâneos iniciados a cada dez segundos, observa-se a mesma vantagem descrita na análise da Tabela 3 das aplicações nativas em relação às aplicações executadas pela JVM.

Outro fator a ser analisado é a proporção no aumento do tempo de resposta em relação à quantidade de usuários em relação ao cenário da Tabela 4, ou seja, ao passar de 100 para 1000 usuários simultâneos:

- Um aumento de 25,62 vezes para a aplicação feita com Spring Boot executada pela JVM;
- Um aumento de 26,57 vezes para a aplicação feita com Spring Boot compilada pelo GraalVM;
- Um aumento de 22,01 vezes para a aplicação feita com Quarkus executada pela JVM;
- Um aumento de 27,53 vezes para a aplicação feita com Quarkus compilada pelo GraalVM

Logo, para um aumento de 100 para 1000 usuários, o tempo de resposta é em média 25,43 ± 3,84 vezes maior para todas as aplicações feitas com Java.

As Tabelas 5, 6, 7 e 8 seguem as especificações para o terceiro cenário: Ordenação de vetores com números inteiros: 100 usuários virtuais, tempo de inicialização igual a 10s e 10 iterações. Cada usuário virtual executa quatro requisições HTTP POST. As tabelas representam respectivamente as aplicações feitas em Spring Boot executada pela JVM, Spring Boot compilada com GraalVM, Quarkus executada pela JVM e Quarkus compilada pelo GraalVM.

Tabela 5 – Terceiro cenário. Aplicação feita em Java 11 com Spring Boot. Aplicação *jar* executada com o sistema de execução Azul Zulu 11.0.9.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3	1000	364.39	298.00	23415	12.18	2.30	2.33
300	1000	362.01	298.00	3585	12.29	13.08	20.33
3000	1000	871.12	659.00	2847	12.29	110.75	182.73
30 mil	1000	5417.56	5023.00	3899	12.28	1085.43	1804.28
Total	4000	1753.77	555.00	23415	48.65	1199.45	1989.58

Fonte: Elaborada pelo autor com base nos relatórios dos testes feitos pelo JMeter.

Tabela 6 – Terceiro cenário. Aplicação feita em Java 11 com Spring Boot. Aplicação nativa compilada pela versão 20.2.0.r11 do GraalVM.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3	1000	548.89	8062.00	4441	8.24	1.55	1.58
300	1000	529.52	437.00	3804	8.26	8.79	13.66
3000	1000	1411.23	429.00	9625	8.25	74.37	122.72
30 mil	1000	8273.87	1029.00	27671	8.24	727.89	1209.95
Total	4000	2690.88	818.00	27671	32.85	809.91	1343.43

Fonte: Elaborada pelo autor com base nos relatórios dos testes feitos pelo Jmeter.

Tabela 7 – Terceiro cenário. Aplicação feita em Java 11 com Quarkus. Aplicação *jar* executada com o sistema de execução Azul Zulu 11.0.9.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3	1000	454.78	348.00	3466	9.23	0.82	1.77
300	1000	434.65	342.00	4479	9.25	8.92	15.30
3000	1000	1128.88	832.00	38263	9.25	82.58	137.49
30 mil	1000	6418.43	5931.00	28840	9.23	814.89	1355.80
Total	4000	2109.19	3640.00	38263	36.82	904.49	1505.82

Fonte: Elaborada pelo autor com base nos relatórios dos testes feitos pelo Jmeter.

Tabela 8 – Terceiro cenário. Aplicação feita em Java 11 com Quarkus. Aplicação nativa compilada pela versão 20.2.0.r11 do GraalVM.

Quantidade de números do vetor	Total de requisi-ções	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebi-do	Envia-do
3	1000	323.56	282.00	2001	12.87	1.14	2.46
300	1000	329.41	284.00	2410	12.92	12.46	21.37
3000	1000	810.37	618.00	4766	12.91	115.32	192.01
30 mil	1000	5065.27	4765.00	20931	12.90	1138.44	1894.12
Total	4000	1632.15	507.00	20931	51.37	1261.85	2100.77

Fonte: Elaborada pelo autor com base nos relatórios dos testes feitos pelo Jmeter.

Para esse terceiro cenário, obtivemos um resultado surpreendente com a aplicação nativa do Spring Boot que teve um desempenho inferior a versão executada pela JVM. Cabe lembrar que o Spring Boot oferece suporte experimental à geração de aplicações nativas com o GraalVM, o plugin de compilação utilizado foi o Spring GraalVM Native versão 0.8.1, esse assunto é tratado na Seção 3.7. Para o Quarkus ocorreu o esperado, um desempenho superior da aplicação nativa.

Na Tabela 9 é possível verificar que há uma tendência de crescimento grande a partir do salto de 300 para 3 mil números. A proporção mostrada para os saltos é a média de cada teste feito

Tabela 9 – Proporção entre a o aumento da média dos tempos médios de resposta entre a ordenação com menos números para a ordenação com mais números para o cenário 3 das Tabelas 5, 6, 7 e 8.

Salto	Proporção
De 3 para 300 números	0,9830 ± 0,046
De 300 para 3 mil números	2,5322 ± 0,190
De 3 mil números para 30 mil números	6,005 ± 0,440

Fonte: Elaborada pelo autor.

Como já foi mencionado anteriormente, a ordenação foi feita com o método *java.util.Arrays.sort.* O limite de 30 mil números aqui estabelecido foi justamente devido a limitações nos testes, um aumento na quantidade de números de 30 mil para 300 mil tornariam as requisições suscetíveis a erros e seria impossível traçar uma tendência. A análise desse cenário feita na Seção 3.3 mostra a mesma relação de aumento no tempo em função do salto na quantidade de usuários.

3.3 TESTES FEITOS NO LAPTOP ACER

Máquina utilizada: laptop Acer Aspire E 14 com 12GB de memória DDR3, um disco rígido de 500GB, um processador Intel Core i3 5005U (2,0GHz, 3MB L3 cache), sistema operacional Linux Lubuntu Eoan Ermine 19.10.

Nas Tabelas 10 e 11 são mostradas apenas as médias gerais da soma de todas as requisições GET e POST feitas por cada usuário em cada cenário. O motivo é o mesmo que foi explicado na Seção 3.2.

Tabela 10 – Primeiro cenário: 100 usuários virtuais, tempo de inicialização igual a 10s e 10 iterações. Cada usuário virtual executa seis requisições HTTP GET e seis HTTP POST.

Tecnologias utilizadas para construir a aplicação	**Total de requisições**	**Tempo de Resposta (ms)**			**Transações por segundo**	**Rede (KB/s)**		**Requisições mal sucedidas**
		Médio	**Mediana**	**Máximo**		**Recebido**	**Enviado**	
Spring Boot – *jar*	12000	960.49	964.00	4489	95.34	1560.82	17.61	0
Spring Boot – nativo	12000	1063.08	1012.00	9808	85.30	1395.41	15.76	0
Quarkus – *jar*	12000	807.53	706.00	5715	108.60	1753.27	20.06	0
Quarkus – nativo	12000	666.38	648.00	4009	128.90	2081.29	23.81	0

Fonte: Elaborada pelo autor.

Tabela 11 – Segundo cenário: 1000 usuários virtuais, tempo de inicialização igual a 10s e uma iteração. Cada usuário virtual executa seis requisições HTTP GET e seis HTTP POST.

Tecnologias utilizadas para construir a aplicação	**Total de requisições**	**Tempo de Resposta (ms)**			**Transações por segundo**	**Rede (KB/s)**		**Requisições mal sucedidas**
		Médio	**Mediana**	**Máximo**		**Recebido**	**Enviado**	
Spring Boot – *jar*	12000	9620.87	9161.50	85686	68.74	1506.87	12.70	0
Spring Boot – nativo	12000	10088.27	9262.50	83321	73.36	1565.70	13.55	2
Quarkus – *jar*	12000	7544.83	4782.50	191347	50.91	931.44	9.40	40
Quarkus – nativo	12000	5984.28	4439.50	88724	92.84	1755.79	17.15	36

Fonte: Elaborada pelo autor.

Para as Tabelas 10 e 11 foram executados os mesmos testes das Tabelas 3 e 4. Como era esperado, um servidor com configurações de hardware inferiores fez o desempenho das aplicações piorarem. A aplicação feita em Quarkus ainda teve seu

desempenho melhorado em sua versão compilada com GraalVM tanto no primeiro como no segundo cenário. Em contrapartida, a versão nativa da aplicação feita em Spring Boot teve um desempenho pior que a versão executada pela JVM em ambos os cenários.

Cabe aqui fazer a mesma análise que foi feita na seção anterior em relação à proporção no aumento do tempo de resposta em relação à quantidade de usuários simultâneos, que saltou de 100 para 1000, entre os cenários das Tabelas 10 e 11:

- Um aumento de 10,02 vezes para a aplicação feita com Spring Boot executada pela JVM;
- Um aumento de 9,49 vezes para a aplicação feita com Spring Boot compilada pelo GraalVM;
- Um aumento de 9,34 vezes para a aplicação feita com Quarkus executada pela JVM;
- Um aumento de 8,98 vezes para a aplicação feita com Quarkus compilada pelo GraalVM.

Portanto, o aumento ocasionado no tempo de resposta pelo acréscimo de usuários simultâneos foi de 9,46 ± 0,70 vezes. É interessante notar que esse aumento é menor que o aumento entre os tempos das Tabelas 4 e 5. Embora o cenário da Tabela 11 tenha uma iteração a menos que o cenário da tabela 4, ambos contam com 1000 usuários simultâneos. Essa redução para uma iteração foi feita após algumas tentativas frustradas de executar as duas iterações com as aplicações sendo executadas pelo Laptop Acer, a segunda iteração fez as aplicações pararem de funcionar no servidor.

As Tabelas 12, 13, 14 e 15 seguem as especificações para o terceiro cenário: Ordenação de vetores com números inteiros: 100 usuários virtuais, tempo de inicialização igual a 10s e 10 iterações. Cada usuário virtual executa três requisições HTTP POST.

Tabela 12 – Terceiro cenário. Aplicação feita em Java 11 com Spring Boot. Aplicação *jar* executada com o sistema de execução Azul Zulu 11.0.9.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3 números	1000	477.10	490.00	3714	11.54	2.18	2.24
300 números	1000	458.03	342.50	3584	11.59	12.33	19.20
3000 números	1000	977.81	335.00	6064	11.59	104.42	172.33
30 mil números	1000	5532.62	655.00	25910	11.57	1022.41	1699.55
Total	4000	1861.39	490.00	25910	46.06	1135.54	1883.69

Fonte: Elaborada pelo autor.

Tabela 13 – Terceiro cenário. Aplicação feita em Java 11 com Spring Boot. Aplicação nativa compilada pela versão 20.2.0.r11 do GraalVM.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3 números	1000	668.85	455.00	6713	8.79	1.66	1.71
300 números	1000	627.27	445.00	5269	8.79	9.36	14.57
3000 números	1000	1293.46	766.50	11980	8.79	79.21	130.72
30 mil números	1000	7203.38	6027.00	38572	8.78	776.01	1289.97
Total	4000	2448.24	675.00	38572	35.09	865.19	1435.22

Fonte: Elaborada pelo autor.

Tabela 14 – Terceiro cenário. Aplicação feita em Java 11 com Quarkus. Aplicação jar executada com o sistema de execução Azul Zulu 11.0.9.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebido	Enviado
3 números	1000	608.57	470.50	3954	8.10	0.72	1.58
300 números	1000	604.78	457.00	4906	8.11	7.83	13.44
3000 números	1000	1500.25	894.50	31135	8.11	72.44	120.63
30 mil números	1000	8162.02	7052.50	36980	8.11	715.79	1190.94
Total	4000	2718.90	696.50	36980	32.37	795.21	1323.99

Fonte: Elaborada pelo autor.

Tabela 15 – Terceiro cenário. Aplicação feita em Java 11 com Quarkus. Aplicação nativa compilada pela versão 20.2.0.r11 do GraalVM.

Quantidade de números do vetor	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)	
		Médio	Mediana	Máximo		Recebi-do	Enviado
3 números	1000	458.85	336.50	5123	10.64	0.95	1735.41
300 números	1000	452.80	331.00	8906	10.65	10.27	2.07
3000 números	1000	1067.81	653.00	7523	10.64	95.05	17.65
30 mil números	1000	6119.57	5219.50	31411	10.62	937.80	158.29
Total	4000	2024.76	502.00	31411	42.431	1042.31	1560.34

Fonte: Elaborada pelo autor.

Cabe notar que a aplicação feita em Spring Boot compilada (Tabela 13) teve um desempenho pior que a aplicação executada pela JVM (Tabela 12), o mesmo ocorreu com essas aplicações quando executadas no Mac Mini (Tabelas 7 e 8). Para as Tabelas 12, 13, 14 e 15 também cabe a mesma análise feita na Tabela 9 em relação ao aumento no tempo de resposta provocado pelo aumento na quantidade de usuários, ela foi feita na Tabela 16.

Tabela 16 – Proporção entre a o aumento da média dos tempos médios de resposta entre a ordenação com menos números para a ordenação com mais números para o cenário 3 das Tabelas 12, 13, 14 e 15.

Salto	Proporção
De 3 para 300 números	0,970 ± 0,041
De 300 para 3 mil números	2,259 ± 0,311
De 3 mil números para 30 mil números	5,600 ± 0,201

Fonte: Elaborada pelo autor.

Observa-se que as proporções da Tabela 16 são muito próximas das proporções da Tabela 9, as proporções podem ser consideradas iguais, se considerarmos os intervalos de confiança.

3.4 TESTES FEITOS COM SISTEMA OPERACIONAL VIRTUALIZADO

Esta seção contém as Tabelas dos testes feitos com virtualização. Para isto, foi usada uma imagem Docker do Oracle Linux com GraalVM 20.2.0.r11 instalada no Mac Mini cujas especificações foram descritas no primeiro parágrafo da Seção 3.2.

Foram executadas a partir dela uma versão nativa e uma versão no formato *jar* das aplicações feitas para Spring Boot e Quarkus. Para esse caso, foi removido todo o código que acessava o banco de dados, restando somente o código que faz a

ordenação de vetor – vide os testes das Tabelas 5, 6, 7 e 8 –, porém como o código ficou mais leve devido às classes removidas, para esta seção, foram refeitos os testes de ordenação das Tabelas 5, 6, 7 e 8 com a versão com menos classes de ambas as aplicações para compará-los aos testes feitos com Docker.

Tabela 17 – Aplicação executada executada com o GraalVM 20.2.r11 e Docker com o sistema operacional Oracle Linux no Mac Mini. Ordenação de vetores com números inteiros: 100 usuários virtuais, tempo de inicialização igual a 10s e 10 iterações. Cada usuário virtual executa quatro requisições HTTP POST: ordenações de vetores com três, 300, 3 mil e 30 mil elementos.

Tecnologias utilizadas para construir a aplicação	**Total de requisições**	**Tempo de Resposta (ms)**			**Transações por segundo**	**Rede (KB/s)**	
		Médio	**Mediana**	**Máximo**		**Recebido**	**Enviado**
Spring Boot – *jar*	4000	1879.11	606.50	20544	45.74	1127.80	1870.72
Spring Boot – nativo	4000	1958.69	641.00	21976	44.33	1093.07	1813.12
Quarkus – *jar*	4000	1912.03	611.00	23919	45.99	1129.81	1880.96
Quarkus – nativo	4000	1742.19	587.00	20719	48.05	1180.34	1965.08

Fonte: Elaborada pelo autor.

Tabela 18 – Mesma situação da tabela 17, exceto pelo fato de a aplicação ter sido executada no Mac Mini fora da virtualização.

Tecnologias utilizadas para construir a aplicação	**Total de requisições**	**Tempo de Resposta (ms)**			**Transações por segundo**	**Rede (KB/s)**	
		Médio	**Mediana**	**Máximo**		**Recebido**	**Enviado**
Spring Boot – *jar*	4000	2934.39	881.00	31642	31.04	765.36	1269.54
Spring Boot – nativo	4000	2160.29	659.00	21959	40.00	986.12	1635.72
Quarkus – *jar*	4000	2254.55	638.00	27141	39.56	971.80	1617.89
Quarkus – nativo	4000	1894.09	573.00	26224	45.32	1113.36	1853.57

Fonte: Elaborada pelo autor.

Tabela 19 – Aplicação executada com o GraalVM 20.2.r11 a partir de um virtualização Docker com o sistema operacional Oracle Linux no Mac Mini. Ordenação de vetores com números inteiros: 1000 usuários virtuais, tempo de inicialização igual a 10s e uma iteração. Cada usuário virtual executa quatro requisições HTTP POST: ordenações de vetores com três, 300, 3 mil e 30 mil elementos.

Tecnologias utilizadas para construir a aplicação	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)		Requisições mal sucedidas
		Médio	Mediana	Máximo		Recebido	Enviado	
Spring Boot – *jar*	4000	17006.30	14459.00	155873	24.04	592.15	983.11	1
Spring Boot – nativo	4000	15241.75	9854.50	156626	23.03	567.70	941.66	0
Quarkus – *jar*	4000	17747.05	11358.00	254330	15.12	369.07	614.34	7
Quarkus – nativo	4000	18313.45	10972.50	148184	26.13	634.59	1056.17	14

Fonte: Elaborada pelo autor.

Tabela 20 – Mesma situação da tabela 19, exceto pelo fato de a aplicação ter sido executada no Mac Mini fora da virtualização.

Tecnologias utilizadas para construir a aplicação	Total de requisições	Tempo de Resposta (ms)			Transações por segundo	Rede (KB/s)		Requisições mal sucedidas
		Médio	Mediana	Máximo		Recebido	Enviado	
Spring Boot – *jar*	4000	16917.64	12489.00	136140	22.75	553.35	930.33	15
Spring Boot – nativo	4000	18441.53	11679.50	133085	28.74	703.53	1175.39	12
Quarkus – *jar*	4000	19553.84	6804.50	132715	26.93	657.89	1094.99	12
Quarkus – nativo	4000	18293.47	11089.50	123346	29.69	724.10	1205.19	12

Fonte: Elaborada pelo autor.

As medições das Tabelas 17, 18, 19 e 20 apontam uma diminuição no tempo de resposta de todas as aplicações feitas com Quarkus executadas no Docker, mas não é possível dizer que o Docker causou isso, uma vez que o sistema operacional fora da virtualização era o MacOS e a imagem da virtualização usou o Oracle Linux. O resultado da Tabela 18 inclusive mostra que a aplicação com menos classes teve um tempo de resposta total médio pior do que a aplicação com todas as classes em algumas versões, vide Tabelas 5, 6, 7, e 8. A Tabela 19 mostra a única medição deste trabalho em que a versão nativa da aplicação feita com Quarkus teve um tempo de resposta médio maior do que a versão executada pela JVM. Na Seção 3.7 são discutidas possíveis origens de fenômenos como esses que podem ameaçar a validade deste trabalho.

3.5 TEMPO DE INICIALIZAÇÃO DAS APLICAÇÕES

Nesta seção é apresentada a tabela com as médias dos tempos de inicialização de cada aplicação. Em todos os casos, as versões compiladas em imagem nativa tiveram uma ampla vantagem em relação às versões executadas pela JVM.

Tabela 21 – Tempo de inicialização médio de cada aplicação.

Tecnologias usadas na aplicação	Sistema operacional / Máquina	Tempo de inicialização médio em segundos (10 amostras)
Java e Spring Boot executado com o Azul Zulu	Mac Mini / Mac OS 13.15	13,594 ± 0,250
	Laptop Acer /Lubuntu 19	22,30 ± 0,65
Java, Spring Boot compilado com GraalVM	Mac Mini / Mac OS 13.15	5,515 ± 0,016
	Laptop Acer /Lubuntu 19	14,70 ± 0,80
Java e Quarkus executado com o Azul Zulu	Mac Mini / Mac OS 13.15	7,39 ± 0,16
	Laptop Acer /Lubuntu 19	20,75 ± 0,40
Java, Quarkus compilado com GraalVM	Mac Mini / Mac OS 13.15	0,301 ± 0,02
	Laptop Acer /Lubuntu 19	16,97 ± 0,59

Fonte: Elaborada pelo autor.

3.6 DISCUSSÃO DOS RESULTADOS

As respostas para as perguntas da Seção 1.2 foram respondidas durante as análises dos dados. A primeira pergunta:

- Como as aplicações executadas do lado do servidor reagem a um grande número de requisições HTTP (*Hyper Text Transfer Protocol*) simultâneas?

Teve sua resposta constatada com os testes feitos com mil usuários simultâneos, vide Tabelas 4 e 11, a aplicação passou a retornar erro para algumas requisições. Não se evidenciou que a compilação por GraalVM possa otimizar isso de alguma forma.

Sobre esta pergunta:

- Qual é o impacto de um aumento na quantidade de bytes trafegados na requisição para cada tecnologia?

Para responder essa pergunta, em primeiro lugar devemos avaliar se o aumento dos tempos mostrados nas Tabelas 9 e 16 tem alguma relação com a complexidade do algoritmo de ordenação. Como foi citado na Seção 2.1, tudo o que sabemos sobre o algoritmo utilizado pelo método que faz a ordenação é que ele é estável. As maiores complexidades no tempo possíveis para algoritmos de ordenação estáveis são O(N

log N) ou O(N²) (DAVE, P. H., p. 409), portanto é importante verificar qual a proporção do salto no tempo que apenas a ordenação provocaria, caso fosse ela o único fator a ser considerado.

Tabela 22 – Cálculo das relações aritméticas das possíveis duas maiores complexidades do método utilizado para ordenar os vetores.

N	$N \log_2 N$	N^2
3	4,75	9,00E+00
300	2468,64	9,00E+04
3 mil	34652,24	9,00E+06
30 mil	446180,25	9,00E+08

Fonte: Elaborada pelo autor.

Tabela 23 – Cálculo das relações aritméticas da proporção de aumento do tempo ocasionada por um algoritmo de ordenação.

Salto	Proporção do aumento da complexidade produzido pelo salto para $N \log_2 N$	Proporção do aumento da complexidade produzido pelo salto para N^2
3 para 300	519,18	10000
300 para 3 mil	14,04	100
3 mil para 30 mil	12,88	100

Fonte: Elaborada pelo autor com base na Tabela 22.

Ao analisar a Tabela 23, podemos concluir que a tendência de proporcionalidade no aumento do tempo de resposta para todos os frameworks, independentemente das configurações de hardware, apontadas pela Tabela 9 e pela Tabela 16 não teve como principal influência a proporcionalidade do aumento na complexidade, porque o maior aumento proporcional registrado para as Tabelas 9 e 16 foi de 6 vezes quando há um salto de 3 mil para 30 mil números no vetor, menor do que qualquer aumento registrado na Tabela 23.

E por fim:

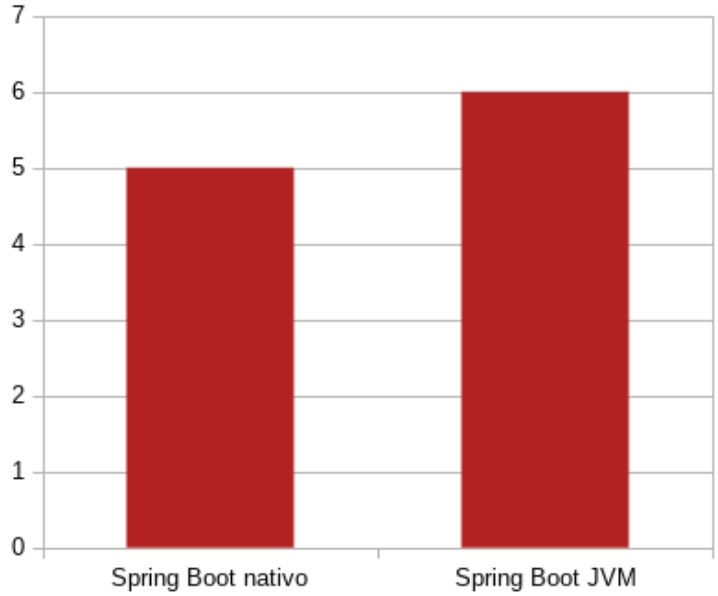
- Qual é o impacto quando são adotadas aplicações compiladas no lugar de aplicações executadas por sistemas de execução?

O maior impacto positivo observado foi no tempo de inicialização, que foi consideravelmente menor. Mas a possibilidade citada na Seção 1.1 de que o tempo de inicialização de máquinas virtuais pode ser imprevisível (Barrett et al., 2017, citado por WIMMER, 2019, p. 2) não se concretizou nos testes realizados no

presente trabalho. A Tabela 21 mostra que o tempo de inicialização das aplicações tem uma estreita margem de confiança, sejam elas pacotes *jar* executados pela JVM ou aplicações nativas compiladas pelo GraalVM.

Para avaliar melhor a taxa de transferência, o Gráfico 1 e o Gráfico 2 mostram uma relação da quantidade de vezes em que a versão compilada em código nativo teve vantagem sobre a versão executada pela JVM em cada cenário.

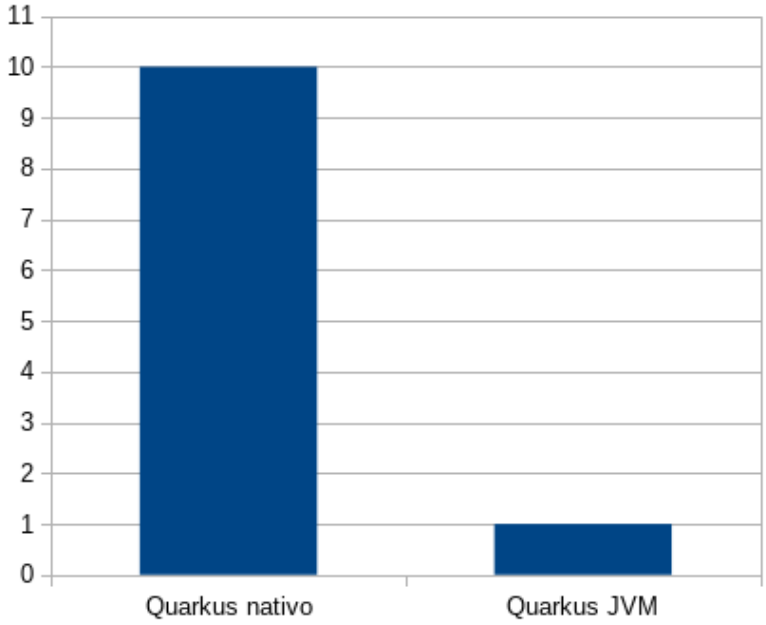
Gráfico 1 – Total de vezes em que cada versão da aplicação feita com Spring Boot (código nativo e executada pela JVM) teve um tempo de resposta menor em cada cenário.



Fonte: Elaborado pelo autor.

O Gráfico 1 mostra que em relação ao tempo de resposta, não houve vantagem da aplicação nativa. Como no momento em que os testes deste trabalho foram realizados, o plugin para compilação de aplicações feitas com Spring Boot encontrava-se em versão experimental, é possível que a versão do *plugin* para a construção de aplicações nativas aqui utilizada – Spring GraalVM Native 0.8.1 – ainda sofra correções e melhorias.

Gráfico 2 – Total de vezes em que cada versão da aplicação feita com Quarkus (código nativo e executada pela JVM) teve um tempo de resposta menor em cada cenário.



Fonte: Elaborado pelo autor.

O Gráfico 2 mostra que houve desempenho superior da aplicação Quarkus compilada para código nativo em todos os cenários, exceção feita ao terceiro cenário, registrado nas Tabelas 14 e 15.

3.7 AMEAÇAS À VALIDADE

Para esta seção, vamos considerar os quatro tipos de ameaça à validade mencionados por Jedlitschka (2008, p. 222): ameaça à validade da formulação, à validade interna, à validade externa e à validade da conclusão. É importante não ignorar as ameaças à validade para que não sejam tiradas conclusões equivocadas a partir do trabalho (JEDLITSCHKA, 2008, p. 222).

Sobre a ameaça à validade da formulação, deve-se discorrer sobre até que ponto pode-se aplicar os resultados de um determinado trabalho em outros contextos (JEDLITSCHKA, 2008, p. 223). Este trabalho tem um escopo bem definido: aplicações feitas com Java 11 compiladas em código nativo e em *bytecode* para serem executadas pela JVM. Cada aplicação aqui criada, em Spring Boot e Quarkus, tem particularidades na implementação, portanto, evitou-se compará-las entre si. Este trabalho não compara o desempenho entre frameworks, mas sim o desempenho da versão nativa e da versão executada pela JVM de cada framework. Ao considerar que o mesmo código fonte gera as duas versões: nativa e em

bytecode, é justo compará-las desde que sejam testadas sob as mesmas circunstâncias.

Ameaças à validade interna são os fatores não considerados em um trabalho e que podem influenciar no resultado do mesmo (JEDLITSCHKA, 2008, p. 223). É conhecido que existem inúmeros fatores alheios a uma aplicação que podem influenciar em seu desempenho, como a temperatura do *hardware*, que segundo SAUCIUC, I. (2005, p. 2, tradução nossa) é gerida com o objetivo de "[…] manter a temperatura da junção da matriz dos microcomponentes abaixo dos limites aceitáveis para garantir o desempenho e a confiabilidade do dispositivo". Também é possível que outros processos executados em segundo plano pelo sistema operacional possam competir por recursos de *hardware*, para citar outro exemplo. Portanto, os valores para os tempos de resposta aqui apresentados podem variar quando executados mais de uma vez no mesmo *hardware*, para mitigar esse problema, as aplicações foram testadas sob várias circunstâncias diferentes, em relação à carga de usuários simultâneos e implementações, para que pudéssemos chegar às conclusões apresentadas neste trabalho.

Uma ameaça à validade externa ocorre quando um experimento é conduzido em um ambiente muito artificial, então, quando é reproduzido em situações do cotidiano, do "mundo real", apresenta resultados diferentes (JEDLITSCHKA, 2008, p. 223). Aplicações como a deste trabalho são comumente executadas em sistemas distribuídos em computação em nuvem.

> Funções em nuvem, um serviço no qual a AWS é pioneira com o AWS Lambda, ganham cada vez mais popularidade como método de executar aplicações distribuídas. Elas formam um novo paradigma, frequentemente chamado de *Function-as-a-Service* (FaaS) ou computação *serverless*. Funções em nuvem permitem aos desenvolvedores lançarem seus códigos em forma de função ao provedor da nuvem, a infraestrutura é responsável pela execução fornecendo os recursos, ela também escala automaticamente o ambiente do sistema de execução (MALAWSKI, M.; FIGIELA, K.; GAJEK, A.; ZIMA, A. p. 1, 2017, tradução nossa).

Portanto, em um cenário de *Function-as-a-Service*, muitos outros fatores devem ser levados em consideração, mas principalmente como a infraestrutura do serviço de computação em nuvem lida com o sistema e execução. Também é possível que uma aplicação em produção tenha cenários muito diferentes dos que foram apresentados ao longo deste trabalho, portanto podem ter um desempenho diferente também. Outro fator para considerar é que foi usado o GraalVM *Comunity Edittion*

na versão 20.2.r11. A versão *Enterprise* oferece mais otimizações, portanto pode ter um desempenho diferente.

Sobre a validade da conclusão, valem como ressalva para este trabalho em particular os mesmos pontos citados como ameaça à validade interna: se fatores que nem foram notados pelo autor podem interferir nos resultados, como podemos estar certos das relações de causas e consequências aqui apresentadas? Para mitigar esse problema, os testes foram executados sob circunstâncias diferentes nos aspectos que o autor teve controle: quantidade de usuários virtuais e um cuidado para que o mínimo de recursos da máquina e do sistema operacional fossem utilizados por outras aplicações durante os testes. As tendências detectadas foram registradas aqui de forma que o experimento possa ser repetido por qualquer pessoa que tenha os artefatos da Tabela 2. Temos de considerar que o autor do trabalho também conduziu o experimento, o que pode acarretar em algum viés na forma de implementação das aplicações, dos testes ou até mesmo algum erro não detectado.

4 CONCLUSÃO

Este trabalho tem como objetivo contribuir para que a comunidade tenha mais um parâmetro na escolha das tecnologias a serem adotadas para criar aplicações *web* do lado do servidor. Duas aplicações foram desenvolvidas e testadas em diversas situações que contribuíram para formar um juízo sobre como cada cenário pode afetar o desempenho do servidor do ponto de vista do cliente.

Um desafio encontrado foi configurar a aplicação em Spring Boot para gerar imagens nativas, pois, como já foi mencionado, seu suporte ao GraalVM é experimental, por isso demandou uma série de configurações e comandos automatizados para gerar a imagem nativa. Houve também a dificuldade inerente da metodologia a que este trabalho se propôs a seguir para obter parâmetros confiáveis de comparação e decidir quais testes deveriam ser feitos.

Como o suporte oferecido ao GraalVM pelo Spring Boot no momento em que os testes foram realizados era experimental, é possível vislumbrar melhorias no desempenho de aplicações nativas feitas com Spring Boot para as próximas versões. Em contrapartida, o suporte que o Quarkus oferecia já era definitivo, o que justifica a superioridade de seu desempenho nas versões nativas em relação às executadas pela JVM. Esses fatores foram discutidos na Seção 3.7.

Ao discutir sobre as ameaças à validade externa na Seção 3.7, foi trazido o contexto do estado da arte dos serviços de computação em nuvem. Para que se tome uma decisão sobre qual tecnologia adotar, deve-se considerar primeiramente o que esses serviços oferecem. Por exemplo, se for indiferente publicar uma aplicação nativa ou uma aplicação para ser executada na JVM para um serviço de computação em nuvem em particular, e o framework escolhido for o Quarkus, recomenda-se publicar a aplicação nativa, porque foi demonstrado neste trabalho que ela apresenta um tempo de resposta menor em quase todos os cenários. Como o Spring Boot ainda não dá suporte definitivo à geração de aplicações nativas, ainda não é recomendável publicá-las em ambientes de produção.

REFERÊNCIAS

ALMEIDA, M. S. D.. **Agregador de Repositórios**: Trabalho de Graduação, 2020. Disponível em: < https://matheusdealmeida.github.io/landing-page >. Acesso em: 24 dez. 2020.

APACHE JMETER. **3. Elements of a Test Plan**. Disponível em: < https://jmeter.apache.org/usermanual/test_plan.html >. Acesso em: 25 nov. 2020.

AZUL. **Zulu OpenJDK Advantages**. Disponível em: < https://www.azul.com/downloads/zulu-community/?package=jdk >. Acesso em: 22 dez. 2020.

CASS, S. **The top 10 programming languages spectrum's 2014 ranking [dataflow]**. IEEE Spectrum, Vol.51(7), p. 68-68. Jul. 2014.

DAVE, P. H. **Design and Analysis of Algorithms**. Pearson Education: Gujarat, India, 2008.

FASTERXML/JACKSON. **Jackson Project Home @github**. Disponível em: < https://github.com/FasterXML/jackson >. Acesso em 24 dez. 2020.

GRAALVM. **Getting started**. Disponível em: < https://www.graalvm.org/docs/getting-started/ >. Acesso em: 18 out. 2020.

GRAALVM. **Release notes**. Disponível em: < https://www.graalvm.org/docs/release-notes/19_0/ >. Acesso em: 18 out. 2020.

JEDLITSCHKA, A. et al. Reporting Experiments in Software Engineering**.** In: **Guide to Advanced Empirical Software Engineering**. SHULL, F.; SINGER, J.; SJØBERG, D. I. K. Londres, Reino Unido: Springer, 2008.

LARSSON, R. **Evaluation of GraalVM Performance for Java Programs**. Bachelor Degree Project, Linnaeus University, Faculty of Technology, Department of computer science and media technology (CM), 2020. Disponível em: <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1457592&dswid=-5138>. Acesso em: 12 dez. 2020.

LEUN, V. V. D. **Introduction to JVM Languages**. Birmingham, Reino Unido: Packt. Publishing Ltd, 2017.

MALAWSKI, M.; FIGIELA, K.; GAJEK, A.; ZIMA, A. **Benchmarking Heterogeneous Cloud Functions**. Cracóvia, Polônia: Springer, 2017. Disponível em: < http://galaxy.agh.edu.pl/~malawski/CloudFunctionsHeteroPar17InformalProceedings.pdf >. Acesso em: 22 dez. 2020.

MARR, S et al. Cross-Language Compiler Benchmarking: Are We Fast Yet?. **DLS 2016:** Proceedings of the 12th Symposium on Dynamic Languages, p. 120-131, Áustria: Johannes Kepler University Linz, 2016.

MCCLUSKEY, G. **Using Java Reflection**. Jan. 1998. Disponível em: < https://www.oracle.com/technical-resources/articles/java/javareflection.html > Acesso em 3 dez. 2020.

MOELLERING, S. **Using GraalVM to Build Minimal Docker Images for Java Applications**. Jun. 2019. Disponível em: < https://aws.amazon.com/pt/blogs/opensource/using-graalvm-build-minimal-docker-images-java-applications/ >. Acesso em: 2 dez. 2020.

ORACLE. **Class Arrays**. Disponível em: < https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html >. Acesso em: 3 dez. 2020

PAULSON, D S. **Biostatistics and Microbiology**: A Survival Manual. Bozeman, Montana, EUA: Springer, 2008.

QUARKUS. **Quarkus - Building a Native Executable**. Disponível em: < https://quarkus.io/guides/building-native-image >. Acesso em: 22 dez. 2020.

QUARKUS. **Supersonic Subatomic Java**. Disponível em: < https://quarkus.io/ >. Acesso em: 22 dez. 2020.

SAUCIUC, I. et al. THERMAL PERFORMANCE AND KEY CHALLENGES FOR FUTURE CPU COOLING TECHNOLOGIES. **Proceedings of IPACK2005**: ASME InterPACK '05, San Francisco, California, USA, 17-22 jul. 2005.

SPRING. **Overview**. Disponível em: < https://spring.io/projects/spring-boot >. Acesso em: 22 dez. 2020.

SPRING. **Spring Native for GraalVM 0.8.3 available now**. Disponível em: < https://spring.io/blog/2020/11/23/spring-native-for-graalvm-0-8-3-available-now >. Acesso em: 22 dez. 2020

TECH EMPOWER. **Introduction**. Disponível em: < https://www.techempower.com/benchmarks/#section=intro&hw=ph&test=plaintext >. Acesso em 2 dez. 2020.

VERMA, M.; DHAWAN, M. **Towards a More Reliable and Available Docker-based Container Cloud**. Ithaca, Nova Iorque, EUA: Cornell University, 28 ago. 2017.

WIMMER, C. et al. Initialize Once, Start Fast: Application Initializationat Build Time. **Proc. ACM Program. Lang.**, vol. 3, No. OOPSLA, Article 184, out. 2019.