UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES
URI ERECHIM
DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO CURSO DE CIÊNCIA DA COMPUTAÇÃO

JESSICA IMLAU DAGOSTINI

AVALIAÇÃO DA APLICABILIDADE DE *CONTAINERS* EM UM SISTEMA DISTRIBUÍDO DE JULGAMENTO DE CÓDIGOS

ERECHIM - RS
2019

JESSICA IMLAU DAGOSTINI

AVALIAÇÃO DA APLICABILIDADE DE *CONTAINERS* EM UM SISTEMA DISTRIBUÍDO DE JULGAMENTO DE CÓDIGOS

Trabalho de Conclusão de Curso, apresentado ao Curso de Ciência da Computação, Departamento de Engenharias e Ciência da Computação da Universidade Regional Integrada do Alto Uruguai e das Missões - URI Erechim.

Orientador: Neilor A. Tonin
Coorientador: Jean Luca Bez

ERECHIM - RS
2019

AGRADECIMENTOS

Inicialmente, agradeço a Deus por toda a proteção e graças concedidas.

Agradeço a minha família por serem minha base e apoio para todos os momentos, principalmente, durante este período tão importante e incrível em minha vida. Sem eles eu não seria nada do que sou hoje e por causa deles que estou aqui.

Agradeço também aos professores do curso com quem tive o privilégio de aprender diversas lições, técnicas ou não, ao longo de todo o período da graduação. Cada um com quem tive a oportunidade de dividir aprendizagens é responsável por uma parte da profissional que aqui conclui uma etapa tão importante da formação acadêmica. Em especial, gostaria de agradecer ao professor Neilor Tonin, que além de me orientar neste trabalho e por quase toda a graduação, se tornou um grande amigo.

Agradeço a todos os colegas da Turma 2015 que dividiram comigo momentos de felicidades, preocupações e de muitas atividades. Em especial, agradeço aos meus parceiros de trabalhos em grupos Cassiane, Daniel Lazarotto e Marisa, que sempre deram o máximo em nossos trabalhos e com quem pude construir uma grande amizade.

Não posso deixar de agradecer também a toda a equipe do URI Online Judge, incluindo todos os bolsistas que passaram pelo projeto durante o período que estive atuando. Cabe um agradecimento muito especial ao coordenador da equipe, meu coorientador neste trabalho e melhor amigo Jean Bez, que junto com professor Neilor acreditaram no meu trabalho desde o início. Sem dúvida alguma, muito da profissional que me tornei se deve a todas as experiências que vivi durante meu período no URI Online Judge.

RESUMO

A tecnologia de *containers* vem sendo cada vez mais utilizada para diversas aplicações. Suas facilidades e benefícios têm tornado esta ferramenta uma alternativa para promover maior tolerância às falhas e à portabilidade de sistemas mais complexos. *Containers* provêm maior isolamento de recursos, de forma mais leve do que máquinas virtuais e permitem que seja possível recriar as mesmas configurações de sistema em qualquer máquina, de forma ágil e simples. Com isso, o presente trabalho visa aplicar e avaliar o uso de *containers* no processo de execução de códigos julgados pelo sistema de julgamento do URI Online Judge. Este sistema de julgamento avalia, em tempo real, códigos-fonte que visam resolver alguns problemas propostos. Neste trabalho serão testados desempenhos e possíveis *overheads* deste sistema, organizado em *containers*, em relação ao tempo de execução dos códigos submetidos e tempo total para julgamento dos códigos. Junto aos testes, também serão observados o tempo da criação e remoção dos ambientes virtuais, a fim de estimar o impacto das execuções no tempo total de execução do sistema todo. Ademais, investigar-se-á possíveis limitações de memória e alocação, segurança e isolamento dos mesmos e facilidades de manutenção (que inclui atualização e instalação de novas linguagens). Ao final, pretende-se concluir se tal adição será de fato adequada ao sistema em questão.

Palavras-chave: Virtualização. Containers. Sistemas Operacionais. Juízes Online.

ABSTRACT

Container technology is increasingly being used for most diverse applications. Its features and benefits have made this tool an alternative to promote greater fault tolerance and portability of more complex systems. Containers provide greater resource isolation, lighter than virtual machines, and allow to quickly and easily recreate the same system configurations on any machine. From these main characteristics, this work aims to apply and evaluate the use of containers for the execution of judged codes from the URI Online Judge judgment system. This judgment system evaluates, in real-time, source codes that aim to solve some proposed problems. In this work, will be tested performances and possible overheads of this system, organized in containers, related to the execution time of the submitted codes and total time for judgment of the codes. Along with testing, the time of creation and removal of these virtual environments will also be watched to estimate the impact of executions on the total system runtime. Also, possible memory and allocation limitations, security and isolation, and maintenance facilities (including updating and installing new languages) will be investigated. In the end, it is intended to conclude whether such an addition will indeed be appropriate to the system in question.

Keywords: Virtualization. Containers. Distributed Systems. Online Judges.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CPU	*Central Processing Unit*
CGROUPS	Controle de Grupos
DP	Desvio Padrão
IaaS	*Infrastructure as a Service*
K.S. Test	*Kolgomorov Smirnov*
KVM	*Kernel-base Virtual Machine*
LXC	*Linux Containers*
SO	Sistema Operacional
PID	*Process Identification*
RAM	*Random Access Memory*
UID	*User Identification*
UOJ	URI Online Judge
VM	*Virtual Machine*

SUMÁRIO

1 INTRODUÇÃO

Container é uma tecnologia que emerge como uma alternativa mais leve e portátil para a virtualização, uma vez que possibilita uma maior escalabilidade para sistemas distribuídos com menores custos computacionais, comparados com máquinas virtuais. Em um *container*, todos os recursos são virtualizados pelo próprio sistema operacional e este é visto como um processo pela máquina hospedeira. Esta tecnologia também se destaca pela possibilidade de criação de diversos ambientes isolados em um mesmo hospedeiro, de forma que estes ambientes não tenham conhecimento de uns aos outros (HARTER et al.,2016).

Isolamento de recursos prevê diversos benefícios. Dentre eles, previne conflitos de concorrência e fornece meios de mitigar execuções maliciosas a fim de melhorar aspectos de segurança em sistemas. A utilização de virtualização para isolamento de aplicações é largamente utilizada atualmente, principalmente, quando se pensa em grandes servidores compartilhados. Pode-se dizer então que a virtualização é a tecnologia-chave, pois é a base da computação em nuvem.

É de alta importância prover segurança em sistemas que permitem a execução de códigos enviados por usuários, como é o caso de juízes online como o URI Online Judge (UOJ). Neste ambiente, usuários submetem diversos códigos-fonte que são executados dentro do sistema em questão, com a finalidade de ser avaliada a sua corretude para solução de determinado problema. Por essa configuração, é importante garantir que possíveis envios de códigos-maliciosos não afetem a segurança do sistema. Ademais, como forma de contemplar uma maior gama de usuários, é interessante que este sistema de correções suporte a execução de códigos em diversas linguagens de programação existentes.

A partir das características apresentadas, o presente trabalho visa aplicar a utilização de *containers* para isolamento das execuções de códigos submetidos para correção pelo sistema de julgamento do URI Online Judge. Além de aperfeiçoar aspectos de segurança, a adição deste componente no contexto de julgamento do UOJ também pretende facilitar à manutenção (que inclui a atualização e instalação de novas linguagens). Para isso, fora planejada toda a arquitetura necessária para a adição dos *containers* no sistema e realizadas avaliações de comportamento, desempenho e possíveis *overheads* com o uso de duas diferentes tecnologias de orquestração de *containers*. Os resultados obtidos foram avaliados através de ferramentas estatísticas que apoiaram as tomadas de decisão de cada etapa de implementação.

Logo, este trabalho está dividido em cinco capítulos, sendo o primeiro a introdução. Em seguida, no capítulo dois, discorrer-se-á sobre conceitos relacionados à virtualização e sistemas operacionais, para embasar o funcionamento de *containers*. Além do mais, abordar-se-á informações relativas ao ambiente do URI Online Judge no capítulo três, e todo o desenvolvimento, bem como às discussões acerca dos resultados obtidos – que basearam as fases seguintes – apresentadas no capítulo quatro. Por fim, o último capítulo, apresentar-se-á as conclusões e propostas para trabalhos futuros.

2 VIRTUALIZAÇÃO

Virtualização é uma simulação via software, de um hardware rodando sobre outro software (SINGH; YIP, 2017). Este conceito de ambiente simulado é comumente chamado de máquina virtual (do inglês, *Virtual Machine*, VM). Cada VM é um ambiente isolado e tem o seu próprio sistema operacional e aplicações rodando dentro dela. A virtualização é peça fundamental para a computação em nuvem, uma vez que ela possibilita a execução de diversos sistemas isolados sob um mesmo hardware, aproveitando assim o máximo da sua capacidade (TRINDADE; COSTA, 2018).

Apesar de sua popularização ser recente, o conceito de virtualização já é discutido desde a década de 1960. Segundo (OLIVEIRA et al., 2015), o tópico surgiu com a introdução do *mainframe* S/370 da IBM, que permitia a criação de máquinas virtuais isoladas para melhor aproveitamento dos recursos da máquina. Com a popularização de microprocessadores de menor custo, a ideia de máquinas virtuais caiu em desuso nas décadas seguintes, voltando à tona no início da década de 2000.

Um dos principais ganhos com o uso de virtualização é a possibilidade de executar aplicações compiladas em qualquer máquina, em qualquer outro sistema. Isso é possível através do uso de interfaces que forneçam aos demais componentes uma a outra interface virtualizada.

> Usando os serviços oferecidos por uma determinada interface de sistema, a camada de virtualização constrói outra interface de mesmo nível, de acordo com as necessidades dos componentes de sistema que farão uso dela. A nova interface de sistema, vista através dessa camada de virtualização, é denominada máquina virtual (AURELIO et al., 2008, p. 158).

Existem diversas possibilidades de implementação de sistemas de máquinas virtuais, variando de acordo com a sua necessidade. Por consequência, existem diversas classificações destas tecnologias, variando de acordo com sistema convidado a ser suportado. Para o contexto do presente trabalho, este capítulo discorre sobre duas classificações em específico: ambientes de máquinas virtuais de sistemas e virtualização em nível de sistema operacional. Ademais, as duas últimas seções do capítulo abordam características e conceitos de dois gerenciadores da virtualização em nível de sistema operacional, ambos de fundamental importância para o trabalho.

2.1 Máquinas virtuais de sistema

Máquinas Virtuais de Sistema são ambientes construídos para suportar a virtualização de sistemas operacionais (SO) convidados completos, com aplicações convidadas rodando sob eles. Mesmo que mais de um sistema convidado esteja sendo executado de forma paralela, cada SO tem a ilusão de estar sendo executado sob um *hardware* físico, e de forma independente. Eles são fortemente isolados uns dos outros (AURELIO et al., 2008).

Para essa virtualização acontecer é necessário o uso de uma camada de *software* que crie uma interface apta para isso. Hipervisor (do inglês, *Hypervisor*), ou Monitor de Máquina Virtual, é uma camada de software entre o hardware e o sistema operacional, responsável por fornecer a abstração necessária para hospedagem de uma ou mais máquinas virtuais. Ele controla como os recursos são acessados pelas máquinas virtuais e aloca dinamicamente os recursos necessários para a VM operar (RIBEIRO; SCHIMIGUEL, 2016). Assim, hipervisores devem oferecer "um ambiente essencialmente idêntico ao da máquina real" (AURELIO et al., 2008, p. 161).

Hipervisores de sistema fornecem ao sistema operacional convidado um conjunto de instruções de *hardware* virtual, que pode ou não ser idêntico ao da máquina física que está hospedando o sistema.

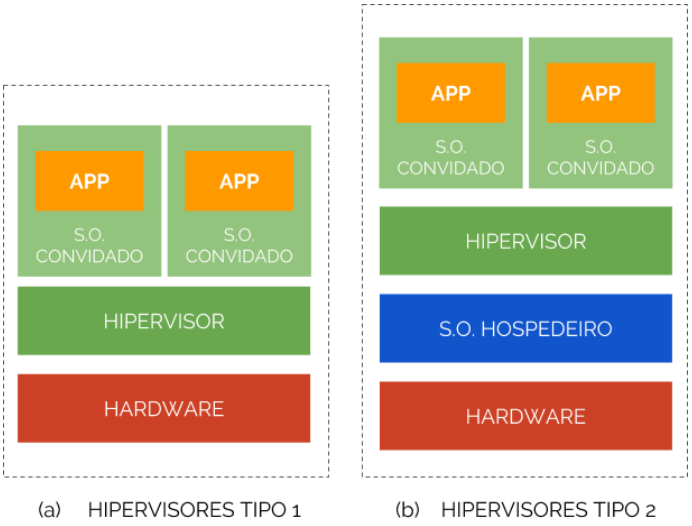
> Além disso, ele virtualiza o acesso aos recursos, para que cada sistema operacional convidado tenha um conjunto de recursos virtuais próprio, construído a partir dos recursos físicos existentes na máquina real. Assim, cada máquina virtual terá sua própria interface de rede, seu próprio disco, sua própria memória RAM, etc. (AURELIO et al.,2008, p. 170)

2.1.1 Classificação dos hipervisores

Conforme demonstrado pela Figura 1, hipervisores de sistema podem ser classificados em dois tipos. O tipo *Bare Metal* ou nativo (Figura 1(a)), é um software que executa diretamente no hardware para fornecer virtualização. Comumente chamado de Hipervisor do Tipo 1, esta forma de organização controla diretamente o hardware hospedeiro e o particiona em múltiplas máquinas virtuais que operam de forma independente (SINGH; YIP, 2017). Dessa forma, cada máquina virtual se comporta como uma máquina física

completa e fisicamente isolada. São exemplos deste tipo de virtualização o VMWare[1] e Xen Server[2].

Figura 1 – Tipo de Hipervisores



Fonte: Adaptado de (BESERRA et al.,2015).

Já o Hipervisor de Tipo 2 (Figura1(b)) é executado sob um sistema operacional e também são chamados de *hosted*. Neste tipo, o sistema operacional convidado é executado sob um hardware virtual, criado sob os recursos do hardware nativo. O sistema operacional hospedeiro é quem organiza as conexões para os recursos deste hardware físico, conversando com o sistema hipervisor para fazer as chamadas entre VM e os recursos como CPU, memória, armazenamento e rede (SINGH; YIP, 2017). São exemplos de hipervisores do tipo 2 o VirtualBox[3] e VMWare Player[4].

2.1.2 Estratégias de virtualização em hipervisores

Para realizar a virtualização é necessário planejar estratégias de como o software hipervisor irá realizá-la, uma vez que um sistema computacional é formado por diversas "peças" que implicam na necessidade do sistema de saber corretamente acessá-las. Uma das

[1] https://www.vmware.com/br.html
[2] https://xenserver.org/
[3] https://www.virtualbox.org/
[4] https://www.vmware.com/br/products/workstation-player.html

estratégias possíveis é a de virtualização total. Nela, “toda a interface de acesso ao hardware é virtualizada, incluindo todas as instruções do processador e os dispositivos de hardware” (AURELIO et al., 2008, p. 173). Uma desvantagem dessa abordagem é que, por todas as instruções terem de ser intermediadas pelo hipervisor, o SO convidado executa de forma mais lenta.

Outra abordagem possível é a de paravirtualização. Nela, o hipervisor fornece uma visualização de um *hardware* virtual similar, mas não idêntico, ao *hardware* real. Para isso, é necessário que o sistema operacional a ser hospedado possua uma adaptação em suas interfaces de conjunto de instruções para executarem sob essa organização. Todavia, essa abordagem permite que o SO convidado acesse diretamente alguns recursos de *hardware*, sem necessidade de intermédio pelo hipervisor. Este comportamento traz benefícios de desempenho (AURELIO et al., 2008).

Por todos estes aspectos apresentados, pode-se concluir que a criação de ambientes virtualizados através de hipervisores é um processo custoso computacionalmente. Sendo assim, essa abordagem de virtualização acaba não sendo adequada para necessidades mais pontuais, como para a execução em ambiente isolado, de um único código em específico. Neste sentido, uma alternativa de virtualização mais leve, como *containers*, é melhor aplicada.

2.2 Virtualização em nível de sistema operacional

Sistemas operacionais são peça-chave de um sistema computacional moderno. Ele é um “dispositivo de software cujo trabalho é fornecer aos programas de usuário um modelo de computador melhor, mais simples” (TANENBAUM,2010, p. 1) e que gerencia de forma adequada o acesso a cada componente de hardware que compõem um computador. A parte responsável por fazer o acesso aos componentes de hardware é chamada de *kernel*. Ele é formado por um conjunto de rotinas que é executada em espaço de núcleo (*kernel space*) do processador, de forma a ter acesso completo a todo hardware e poder executar qualquer instrução. Programas de usuário fazem chamadas ao espaço de núcleo para fazerem uso dos dispositivos que compõem um sistema computacional (TANENBAUM, 2010).

Na virtualização em nível de sistema operacional, uma chamada de sistema (*syscall*) é utilizada para criar um ambiente isolado dentro do próprio SO. Essa *syscall* criará um ambiente que agrupará um conjunto de processos isolados, invisíveis do todo, que farão

chamadas ao *kernel* do sistema operacional para executar. Esse ambiente isolado, com tais características, são popularmente chamados de *containers*. A Figura 2 exemplifica graficamente tais ambientes. Cada *container* é uma virtualização completa, possuindo seu próprio sistema de arquivos e endereço IP, bibliotecas e serviços, que realizam as chamadas de *kernel* através do sistema operacional hospedeiro. Por não terem a necessidade de simular um hardware completo, mas sim fazerem uso do hardware real, *containers* são mais leves e possuem um *boot* mais rápido (BERNSTEIN, 2014).

Figura 2 – Organização Estrutural de Sistema Virtualizados com Containers

Fonte: Adaptado de (BESERRA et al.,2015).

Cada processo dentro de um sistema operacional é identificado por um *process identification*, PID, que é associado ao *user identification*, UID, do usuário que criou e é dono deste processo. Eles estão organizados dentro de uma Tabela de Processos, que é um arranjo de estruturas de cada processo do sistema.

> Cada entrada [da tabela de processos] contém informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, sua informação sobre contabilidade e escalonamento e tudo o mais sobre o processo que deva ser salvo quando o processo passar do estado *em execução* para o estado *pronto* ou *bloqueado* (TANENBAUM, 2010, p. 55).

Processos relacionam-se entre si. A maioria dos processos em sistemas operacionais são criados por outros processos. Assim, cria-se uma estrutura hierarquizada, em que os processos geradores são chamados de processos pai e os novos processos de filhos. Um mesmo "pai" pode possuir diversos filhos. Por conseguinte, também é possível organizar processos em grupos, de forma que o grupo possua alguma similaridade previamente definida. Ao realizar este agrupamento, é possível aplicar operações de forma a afetar todos os processos

pertencentes ao grupo.

Através deste agrupamento é possível criar espaços isolados e independentes, em que os processos recebem identificadores únicos para aquele espaço em específico. Este isolamento de processos pode ocorrer a partir do uso de *chroots* ou *namespaces*. *Chroot* é a forma mais simples de limitar acesso a recursos. "Este método permite a restrição de processos para uma determinada parte do sistema de arquivos" (MAUERER,2010, p. 48). Ele altera o diretório raiz de um processo para um novo, de forma que impede que o processo e seus filhos acessem arquivos ou comandos fora dessa árvore.

Já os *namespaces* são diferentes visões do sistema, que podem lidar com diversos contextos em um mesmo sistema de forma isolada (MAUERER, 2010). Nessa organização, cada espaço possui seu grupo de processos isolados, que são criados a partir de um processo pai (sendo o único visível externamente). Todos os processos filhos dentro de um *namespace* não têm conhecimento de outros processos que estejam fora de seu grupo.

> *Namespaces* podem ser organizados hierarquicamente [...]. Cada um deles possui suas próprias atividades iniciais com PID 0, e os PIDs das outras atividades são designados de forma incremental. Os *namespaces* filhos possuem uma atividade inicial com PID 0 e dois processos com PIDs 2 e 3, respectivamente. Uma vez que os mesmo PIDs aparecem muitas vezes no sistema, os números não são globalmente únicos (MAUERER, 2010, p. 48).

Há diversas implementações de *namespaces* existentes, cada uma isolando uma funcionalidade diferente. Cada um destes espaços possuem os seus próprios objetos internos de *kernel* que os implementam. As descrições a seguir são baseadas de (BIEDERMAN; NETWORX, 2006) e (RESHETOVA et al., 2014).

- ***Mount Pount Namespace,* MNT** - São espaços de pontos de montagem, responsáveis por separar pontos de montagem do *host* e dos espaços virtualizados. Foi o primeiro *namespace* criado;
- ***Process Identification Namespace,* PID** - Espaço responsável pela independência entre identificadores de processos;
- ***Network Namespace,* NET** - Cria pilhas de rede isoladas uma das outras e associadas a um determinado conjunto de processos. É responsável por toda a organização da rede, incluindo endereços IP, firewall, etc.;
- ***InterProcess Communication Namespace,* IPC** - Controla memória, semáforos e

filas de mensagens compartilhadas, isolando processos no estilo SysV IPV[5];

- ***Unix Time Sharing Namespace,* UTS** - Provê uma maneira do sistema aparentar possuir diferentes *hosts* e nomes de usuário, caracterizando e identificando o sistema em que uma aplicação está rodando;
- ***User Namespace*** - Permite o isolamento com a possibilidade de execuções privilegiadas sem a necessidade de dar privilégios para o processo em si. Também permite uma segregação de identificações de usuário entre conjuntos de processos;

Além de fazer uso do espaço de kernel, *containers* fazem uso do controle de grupos (*cgroups*) do Linux. Essa funcionalidade permite que processos sejam "organizados em grupos hierárquicos de forma que os recursos usados por estes possam ser gerenciados e monitorados" (KERRISK, 2019, p. 1). Os *cgroups* fornecem mecanismos para limitar e controlar o uso de recursos de CPU, memória e entrada e saída para um grupo de processos. O *cgroup namespace* também é utilizado, a fim de esconder a identidade do *cgroup* que o processo é membro. Assim, *containers* fazem uso de *cgroups* para gerenciar e controlar acesso a recursos de hardware.

Existem diversas soluções implementadas que realizam a virtualização baseada em *containers*. Eles diferenciam-se entre si, principalmente, na forma como os recursos são gerenciados para cada *container* e em como o isolamento entre as instâncias acontece.

A LinuxVServer (PöTZL, 2011) é uma das mais antigas implementações de *containers*, que introduziu a capacidade de isolamento de rede, CPU e processos, fazendo uso do padrão POSIX e chamadas de sistema *chroot*. O isolamento de processo acontece através do espaço global de PID, escondendo todos os processos de fora do escopo de cada *container* em execução. *Containers* criados por esse gerenciador não possuem rede própria – todos compartilham o mesmo subsistema de rede. O controle de acesso a recursos é feito por chamadas de sistema *rlimit root* (XAVIER et al., 2013).

A OpenVZ (OPENVZL, 2018) foi construída no topo do espaço de *kernel*, a fim de garantir que cada *container* possua fontes isoladas de recursos. OpenVZ faz uso de *namespaces* para o isolamento, em que cada *container* criado por esse gerenciador possui o seu próprio espaço de identificador de processos (PIDs), fazendo com que cada processo interno do *container* possua um identificador único dentro dele. *Containers* OpenVZ também possuem o seu próprio espaço compartilhado de memória, semáforos e mensagens, bem como a sua própria pilha de rede ao fazer uso da *network namespace* (XAVIER et al., 2013;

[5] SysV é uma abreviação para System V, que é uma das primeiras versões comerciais do sistema operacional Unix.

Dua; Raja; Kakadia, 2014).

Uma vez que ambos foram utilizados no presente trabalho, as seções a seguir dedicam-se a apresentar maiores detalhes de funcionamento e características de outros dois gerenciadores, Linux *Containers* (LXC) e Docker.

2.3 Linux *Containers*

O início do projeto Linux *Containers* (CANONICAL, 2019) é baseado na adição dos *cgroups* ao *Linux kernel* 2.6.24. (CROSBY, 2017). Este gerenciador faz uso de diversos *namespaces* para garantir o isolamento entre os *containers*. PIDs, IPCs e pontos de montagem são virtualizados através de seus respectivos *namespaces*. LXC também faz uso do *network namespace* para isolar a rede do *container* perante ao *host*, e utiliza *cgroups* para limitar o acesso dos recursos.

> *Containers* LXC são considerados um meio-termo entre *chroot* e uma máquina virtual completa. O principal objetivo do LXC é criar um ambiente o mais próximo o possível da instalação Linux padrão, mas sem a necessidade de um *kernel* separado (CANONICAL, 2019).

Como forma de melhorar o serviço prestado e fornecer uma melhor experiência ao usuário, foi adicionado um novo gerenciador de sistema ao LXC, o LXD. Criado no topo da implementação LXC, LXD faz uso da *liblxc* para a criação dos *containers*, fornecendo uma *Application Programming Interface* (API) que visa provir um fácil gerenciamento dos ambientes virtuais. Pode-se considerar o LXD uma nova geração de *containers* LXC (GRABER, 2015).

A partir do LXD, Linux *containers* são baseados em imagens – anteriormente chamados de *templates* – e possuem um grande número de distribuições Linux adaptadas para a execução nestes ambientes virtualizados. Estas imagens são arquivos que guardam configurações que são aplicadas a um *container* quando ele executar. Ademais, com essa atualização, o controle de recursos, rede e arquivos é feito de forma mais eficaz e intuitiva para o usuário.

O foco dos *containers* LXC/LXD é o da execução de sistemas. Ou seja, um *container* LXC/LXD é voltado para a execução de uma cópia limpa de uma distribuição Linux ou de uma aplicação completa, não importando para o gerenciador o que está sendo executado dentro do *container*. Esta abordagem é um pouco diferente dos *containers*

Docker, que tem como foco distribuir aplicativos como *containers*, o que faz com que estes gerenciadores levem mais em conta o que está sendo executado dentro do ambiente virtual (GRABER, 2015). Abordar-se-á mais sobre o pressuposto na seção a seguir.

2.4 Docker

Docker (DOCKER, 2019) é uma plataforma *open-source* para gerenciamento de *containers*. Ele representa uma evolução na tecnologia de *containers*, estendendo as já existentes tecnologias ao prover um mecanismo de camadas subjacentes independentes, que unem-se em uma API para criar e gerenciar *containers* (TRINDADE; COSTA, 2018). Diferentemente das tecnologias pioneiras de *containers*, o Docker agrega diversas funcionalidades e módulos, que permitem que o usuário final obtenha uma abstração alta das chamadas de sistema e isolamentos necessários para a execução desta virtualização.

A Docker *engine*, que teve como base a *liblxc* em suas primeiras versões, hoje executa sob o *containerd daemon* (CONTAINERD, 2019). O *containerd* é um *container runtime* que abstrai chamadas de sistema para a criação de *containers* por *engines* de maior nível, podendo assim ser possível criar *containers* em sistemas operacionais de fora da família *UNIX*. Este *daemon* gerencia *containers* através do espaço de nomes do *kernel* do sistema operacional, utilizando também os mesmos *namespaces* utilizados pelo LXC: PID, NET, IPC, *Mount* e UTS. Para fazer o gerenciamento de recursos de hardware, o *containerd* também faz uso dos *ccgroups* para isso (ISMAIL et al., 2015).

Containers Docker são criados a partir de imagens. Assim como apresentado nos *containers* LXC, imagens são *templates* com instruções para criar um *container* Docker. Normalmente, uma imagem é baseada em uma outra imagem.

> Cada camada [de uma imagem] contem as modificações feitas para o sistema de arquivos da camada anterior, iniciando a partir da imagem base (geralmente uma versão leve de uma distribuição Linux). Dessa forma, imagens são organizadas em árvores e cada imagem tem um pai, com exceção das imagens que são raízes das árvores (COMBE; MARTIN; PIETRO, 2016, p. 3).

Há diversos componentes que fazem parte da arquitetura Docker. O Docker *client* é a interface de linha de comando que faz a comunicação com o Docker *daemon* através de uma API. O *daemon*, por sua vez, aceita conexões de clientes via API e expõem as funcionalidades da Docker *engine* além de realizar o monitoramento dos processos, *containers* e organização geral de imagens. A *engine* faz a execução por baixo do *daemon*,

criando os *namespaces* e controles de acesso a recursos. Por fim, diversos objetos são criados para um *container* Docker ser executado. Além da imagem e do próprio *container*, pode-se criar um volume para o *container* – que é uma espécie de ponto de montagem onde os arquivos manipulados dentro do *container* podem ser persistidos – e um objeto de rede, que cria uma interface de rede para o *container* (SPARKS, 2018).

3 AMBIENTE

Programas de origem algorítmica normalmente possuem um preceito de comportamento: recebem uma entrada devidamente formatada em determinado padrão, realizam o processamento destes dados e retornam um resultado, ao final do processamento, também padronizado. Por esse padrão de comportamento, que também segue um sequência algorítmica, é possível que a avaliação da corretude destes programas ocorra através de um sistema automático que forneça os dados de entrada ao programa-fonte e compare a saída gerada por este programa em avaliação com uma saída esperada. Estes sistemas são conhecidos como juízes online (CHAVES et al.,2013).

Juízes online possuem um repositório de exercícios pré-estabelecidos, que são compostos por uma descrição e pelos arquivos de entradas de dados e de saídas esperadas, a fim de ser possível, assim, realizar a avaliação dos códigos submetidos para a resolução dos problemas. Para enviar sua solução codificada para avaliação, o usuário deve submeter seu código-fonte na plataforma, para que ele seja então julgado e sua avaliação retornada (KURNIA; LIM; CHEANG, 2001).

O URI Online Judge (UOJ) (TONIN; BEZ, 2012) é um juiz online desenvolvido no Departamento de Engenharias e Ciência da Computação da URI Erechim. Inicialmente desenvolvido com o intuito de prover um ambiente de prática para os estudantes do próprio campus, o projeto acabou ganhando maior notoriedade. Hoje, o URI Online Judge é utilizado por mais de 400.000 usuários do mundo todo.

Com o intuito de melhor compreender o ambiente em que este trabalho foi aplicado, as seções a seguir transcorrem às características gerais da ferramenta, bem como estatísticas e detalhes técnicos. Além disso, na última seção, discutir-se-á os aspectos relacionados ao funcionamento do sistema de correção, linguagens suportadas e respostas para as avaliações.

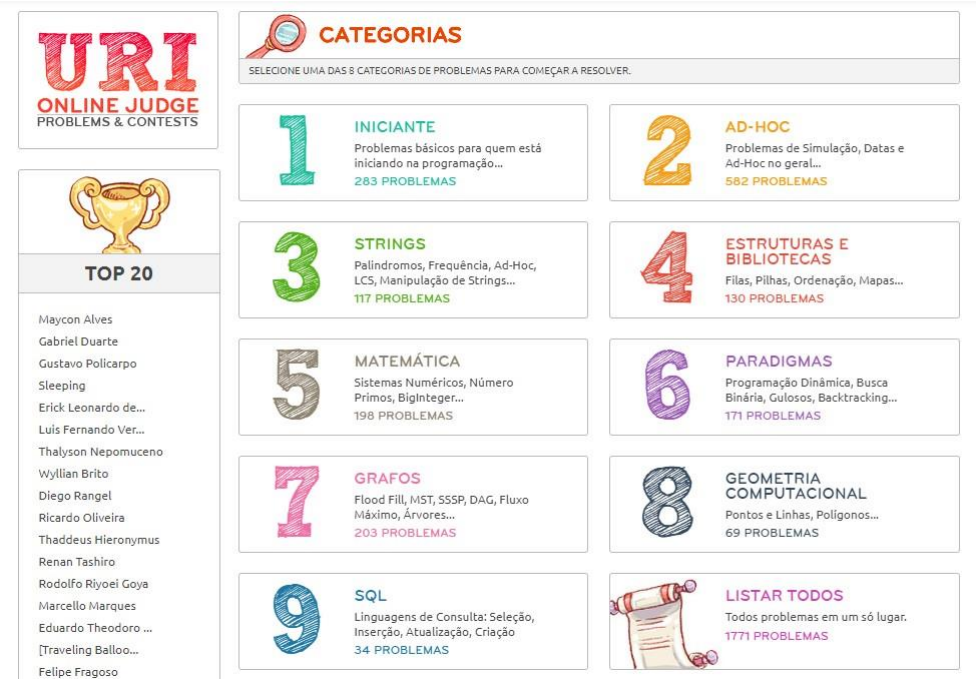
3.1 Características gerais

Com o projeto iniciado em 2012, diversas funcionalidades foram agregadas ao UOJ ao longo dos anos de existência do projeto. O módulo principal da plataforma – URI Online Judge – realiza a correção automática de submissões que visam resolver um dos mais de 2000 problemas disponíveis em seu repositório. Estes problemas estão divididos em 9 grandes categorias.

Conforme mostrado pela Figura 3, as 8 primeiras categorias são referentes a problemas que devem ser resolvidos com algoritmos estruturados, utilizando linguagens de

programação. A categoria 9, por sua vez, possui problemas que devem ser resolvidos através de linguagens de consulta. Este é o módulo URI SQL (LIMA et al., 2017), que conta com um juiz diferenciado para essas correções.

Figura 3 – Divisão de Categorias de problemas do URI Online Judge



Fonte: (AUTORA, 2019).

Além do ambiente de correção de códigos-fonte em tempo real, o UOJ também fornece outras funcionalidades. O URI Online Judge Academic (BEZ; TONIN; RODEGHERI, 2014) é um módulo voltado para professores, em que podem utilizar os exercícios do repositório do UOJ em suas aulas. Através deste módulo, o professor monta uma lista com exercícios selecionados para os seus estudantes, e a partir desta lista, ter acesso às submissões de seus alunos para poder acompanhar o progresso e possíveis dificuldades.

Seguindo o contexto educacional, em 2017 foi adicionado ao URI Online Judge o módulo Blocks (DAGOSTINI et al., 2018). O URI Blocks é um módulo que permite a montagem de soluções para os problemas do repositório através de uma linguagem visual, em que blocos com determinadas funções podem ser conectados para resolverem os problemas propostos. O módulo é especialmente utilizado por professores para aplicação com estudantes em fase de iniciação à programação.

Há também dois módulos de competições dentro da plataforma: o módulo de Contests e o de Torneios. O primeiro é voltado para a realização de competições de programação com problemas de origens externas ao URI Online Judge, ou inéditos. O

segundo, por sua vez, é voltado à criação de competições com problemas já presentes no repositório do UOJ, que podem ser usados facilmente por qualquer usuário registrado no ambiente.

Estatisticamente, os números do URI Online Judge são bastante expressivos. Os dados

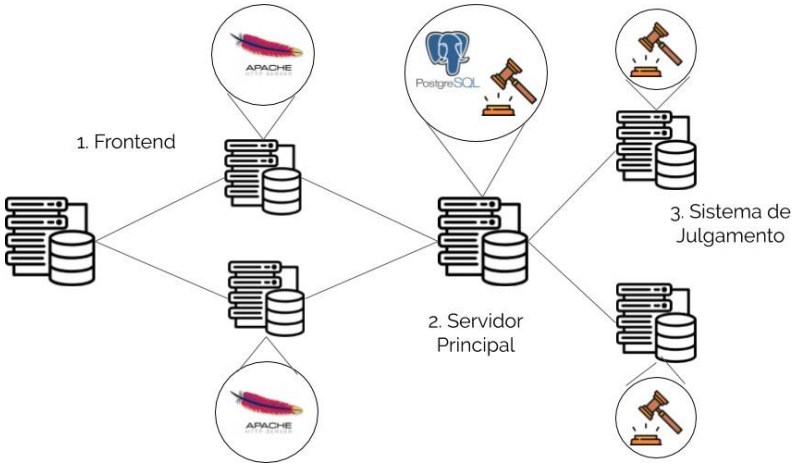
abaixo foram compilados a partir da sua base de dados e publicamente divulgados no projeto URI Online Judge - Year In Numbers[6] edição 2/2019:

- Mais de 15.000.000 de submissões corrigidas;
- 435 competições realizadas;
- 2082 professores utilizando o *URI Academic*;
- 2273 universidades cadastradas, indicadas pelos usuários.

3.2 Características técnicas

A partir dos dados anteriormente apresentados, é perceptível a necessidade de uma boa organização técnica, de forma a fornecer um serviço altamente disponível aos usuários da ferramenta. O URI Online Judge está hospedado em máquinas virtuais dentro da DigitalOcean[7], uma plataforma que oferece Infraestrutura como Serviço (*IaaS*). A DigitalOcean faz uso da virtualização *Kernel-base Virtual Machine* (KVM) e fornece uma interface *web* e uma API bastante amigável e que permite a criação de instâncias de forma simples, rápida e intuitiva.

Figura 4 – Distribuição dos componentes que compõem a arquitetura do URI Online Judge



Fonte: (AUTORA, 2019).

[6] https://www.urionlinejudge.com.br/year-in-numbers/2019/2/
[7] https://www.digitalocean.com/

Quadro 1 – Características técnicas das máquinas servidores do URI Online Judge

	Máquinas *Frontend*	**Máquina Principal**	**Máquinas Juiz**
Sistema Operacional	Ubuntu 18.04 X86_64	Ubuntu 14.04 X86_64	Ubuntu 14.04 i386
Processadores	4 vCPUs	6 vCPUs	1vCPU
Memória RAM	8GB	16GB	2GB
Armazenamento	160GB	320GB	50GB

Fonte: (AUTORA, 2019).

A arquitetura do URI Online Judge está organizada de forma completamente distribuída. Ela é composta por três principais componentes: *frontend*, servidor principal e sistema de julgamento, conforme mostra a Figura 4. As características técnicas de cada máquina envolvida nesta arquitetura é detalhada pelo Quadro 1.

O componente *frontend* é responsável por servir as interfaces e serviços *web*. Na ponta da distribuição fica um balanceador de carga, sendo este um serviço também oferecido pela DigitalOcean para seus usuários. Este balanceador distribui a carga de acessos entre duas máquinas que hospedam as plataformas *web* do UOJ. As máquinas que servem as plataformas web estão configuradas com o servidor *web* Apache.

O servidor principal é o responsável por armazenar os arquivos principais do sistema e o banco de dados. Ele também hospeda o serviço mestre do sistema de julgamento, que é composto por máquinas virtuais dedicadas às correções dos códigos (“Máquina Juiz”). Essas organizações permitem que o sistema de julgamento seja escalável, podendo ser adicionadas tantas instâncias quantas forem necessárias para distribuírem a carga de trabalho.

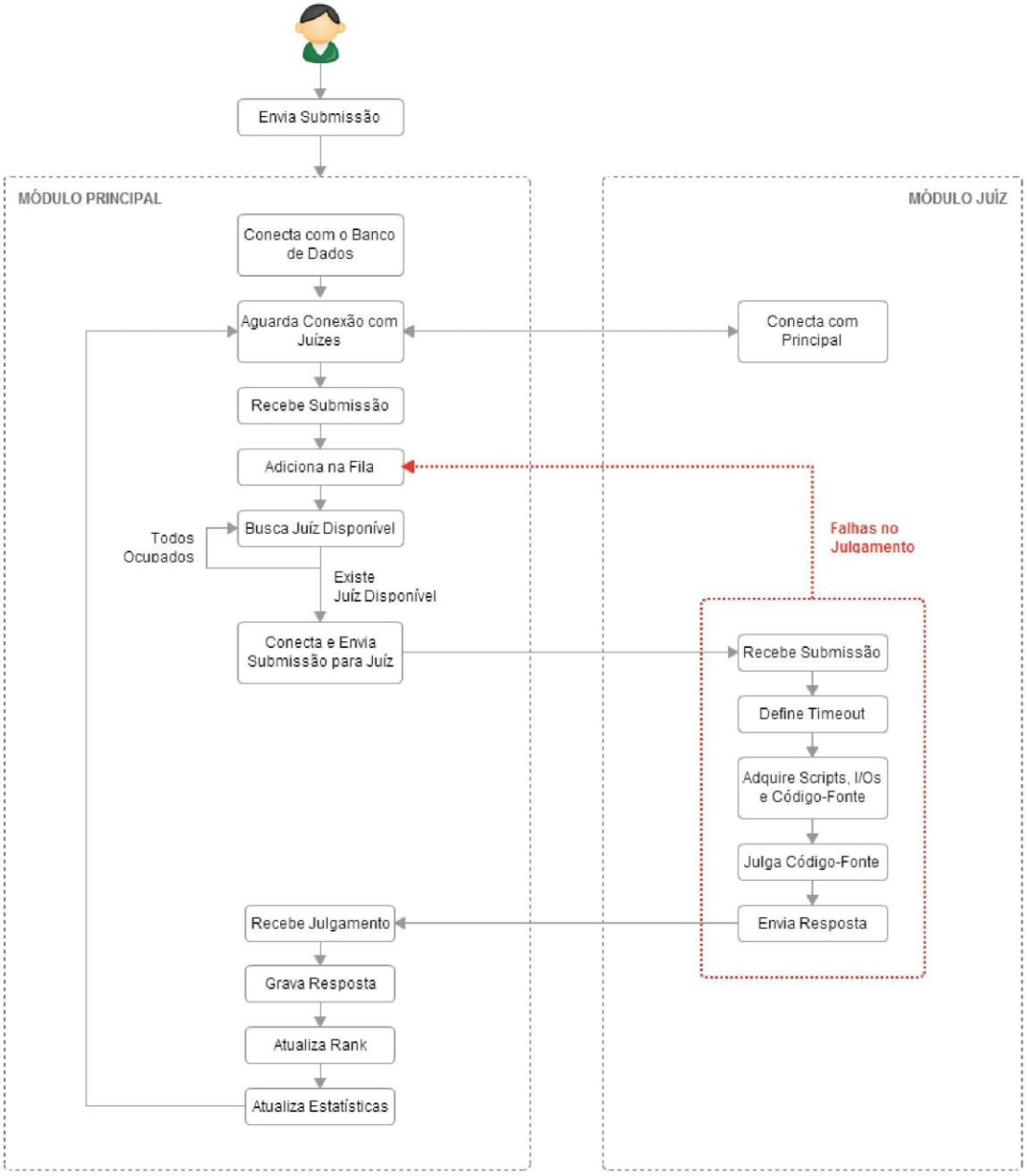
3.3 Sistema de correção

Conforme inicialmente comentado, o comportamento do julgamento de códigos segue uma sequência algorítmica:

1. O juiz recebe o código-fonte a ser avaliado;
2. O código é compilado (para linguagens que requerem compilação);
3. Executa-se o código, passando como entrada os arquivos de entrada do problema e gravam-se as saídas em outros arquivos, respectivos para cada entrada;
4. A saída gerada pelo código submetido é então comparada com a saída esperada, cadastrada com o problema;
5. Define-se a resposta da avaliação.

Esse comportamento é a base para o processo de julgamento de códigos, porém não o compõem por completo. Uma vez que este procedimento deve ser executado para diversos códigos enviados em curtos intervalos de tempo entre um envio e outro, é necessário organizar o sistema de forma que ele possa atender adequadamente todas essas solicitações.

Figura 5 – Processo do sistema de julgamento



Fonte: (BEZ, 2014, p. 48).

O sistema de julgamento do URI Online Judge é composto por dois módulos, sendo eles o módulo principal e o módulo juiz, como mostra a Figura 5. O módulo mestre é o responsável por distribuir as submissões entre os juízes e por gravar no banco de dados os

resultados dos julgamentos. O sistema faz uso de *sockets* para realizar as conexões entre banco de dados, juiz mestre e juízes distribuídos. Toda a estrutura de julgamento aplica técnicas de tolerância a falhas, possibilitando que o sistema detecte e recupere falhas e assim mantenha alta disponibilidade (BEZ, 2014).

A cada nova submissão gravada no banco de dados, uma *trigger* é disparada e executa uma *procedure*, que seleciona os dados essenciais para o julgamento e os retorna em uma *string*. O módulo mestre possui um *listener* que é responsável por receber o *payload* gerado pela *procedure* de notificação. A partir disso, o módulo mestre busca por um juiz em sua lista de conexões disponíveis, e encaminha as mesmas informações para que este realize o procedimento de julgamento do código (BEZ, 2014).

De forma a garantir que possíveis execuções errôneas não travem o sistema e interrompam o fluxo de julgamento, *signals* e *alarms* foram configurados ao longo da execução do processo de avaliação e são disparados caso o processamento ultrapasse determinado intervalo de tempo programado. Terminado o processamento, o juiz retorna para o mestre a avaliação do código, que grava essas informações no banco de dados.

Um código-fonte pode receber uma das seguintes respostas após a sua avaliação pelo sistema de correção do URI Online Judge:

- *Accepted*: o código gerou as saídas esperadas para todos os casos de teste do problema;
- *Wrong Answer*: o código não gerou a saída esperada para um ou mais casos de teste;
- *Presentation Error*: a apresentação da saída do código julgado se difere em um ou mais casos de teste da apresentação esperada;
- *Runtime Error*: houveram erros durante a execução do código em avaliação (normalmente acessos indevidos de memória);
- *Compilation Error*: houveram erros durante a compilação do código em avaliação;
- *Time limit exceeded*: o código em avaliação levou um tempo maior que o configurado para o problema para executar;
- *Memory Limit Exceeded*: o código em avaliação usou mais memória que o configurado para o problema para executar;
- *Closed*: o sistema de julgamento não pode encontrar o arquivo do código-fonte ou dos casos de teste para executar a avaliação.

Estes códigos-fonte podem ser codificados em determinadas linguagens de programação. Atualmente, o sistema de julgamento do URI Online Judge possui suporte para

a avaliação de códigos escritos em 14 linguagens de programação/versões diferentes.

O Quadro 2 sumariza as linguagens e suas respectivas versões suportadas, bem como os compiladores/interpretadores das mesmas.

Quadro 2 – Linguagens de Programação Atualmente Suportadas pelo UOJ

Linguagem/Versão	**Compilador**
C	gcc 4.8.5
C99	gcc 4.8.5
C++11	g++ 4.8.5
C++17	g++ 7.3.0
C#	mono 5.10.1.20
Go	go 1.8.1
Haskell	ghc 7.6.3
Java 7	OpenJDK 1.7.0
Java 8	OpenJDK 1.8.0
JavaScript	nodejs 8.4.0
Kotlin	1.2.10
Lua	5.2.3
Ocaml	4.01.0
Pascal	fpc 2.6.2
Python 2	python 2.7.6
Python 3	python 3.4.4
Ruby	ruby 2.3.0
Scala	scalac 2.11.8

Fonte: (AUTORA, 2019).

Cada linguagem de programação suportada pelo juiz do UOJ possui três *scripts* para o processo de julgamento: um é responsável pela compilação do código-fonte, outro pela execução do mesmo, e, por fim, um terceiro *script* é responsável por realizar a comparação da saída gerada pelo código do usuário com a saída esperada pelo problema. Estes *scripts* também são responsáveis por converterem os códigos de retorno dessas execuções para os padrões esperados pelo código do juiz propriamente dito.

4 DESENVOLVIMENTO E CONSIDERAÇÕES

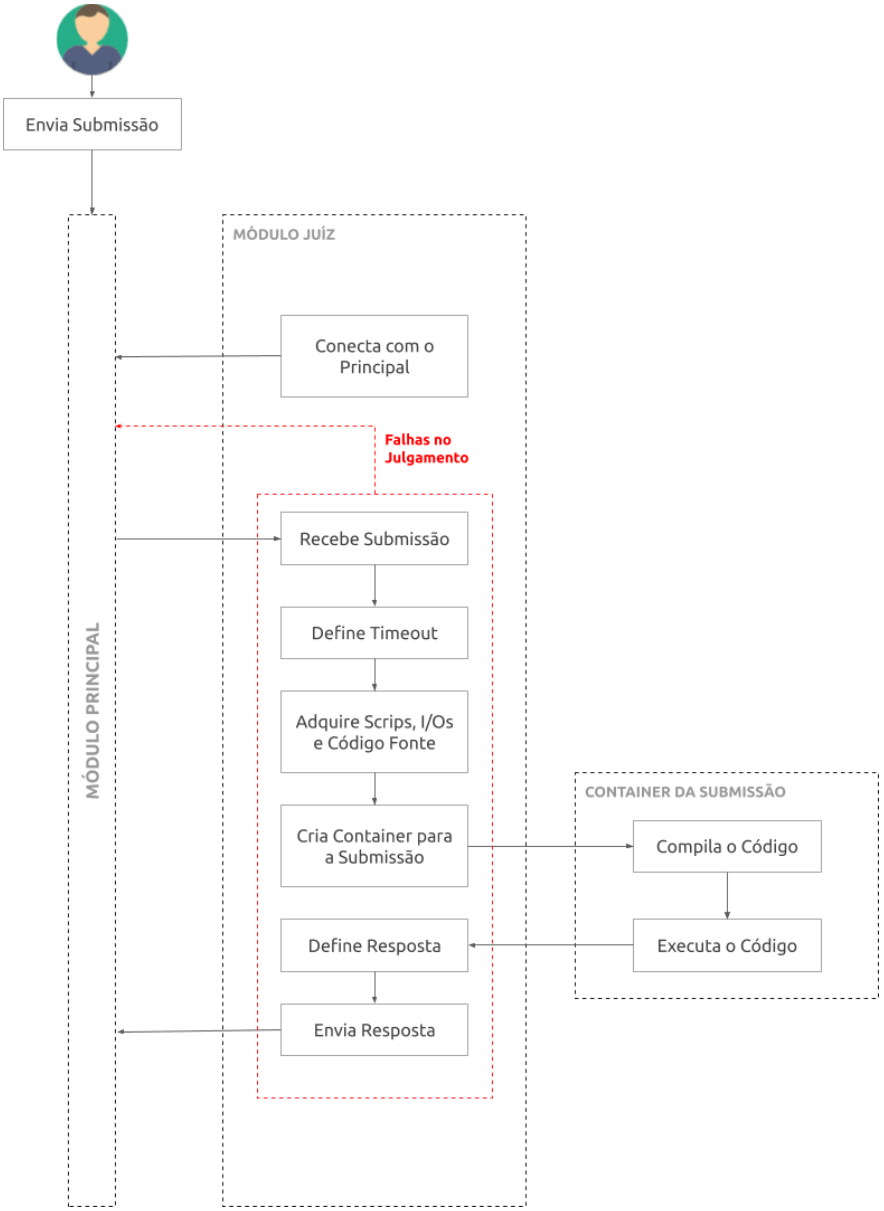
Por proverem uma forma leve e rápida de instanciar ambientes isolados para execuções de programas de usuário, a adição de *containers* na hierarquia de julgamento do URI Online Judge vem com o objetivo de garantir maior isolamento e segurança para as execuções de códigos-fonte a serem julgados pelo sistema. Além disso, o uso desta virtualização permitirá facilidades na manutenção do sistema, em relação à adição e atualização de linguagens e versões de compiladores suportados para julgamento.

O presente capítulo irá apresentar o desenvolvimento da proposta, juntamente com resultados obtidos. Optou-se por essa organização, pois a discussão dos resultados obtidos em cada fase do projeto serviu como base para a tomada de decisões que definiram as etapas posteriores. As seções a seguir discutem os testes realizados e relacionados ao comportamento de execuções dentro de *containers*, também do comportamento do *container* em si, além das arquiteturas de organização propostas e atualizações realizadas no sistema de julgamento para a adequação a esta nova organização.

4.1 Organização do ambiente

A arquitetura inicialmente projetada para essa adição é mostrada pela Figura 6. A cada envio, o *script* de compilação da linguagem cria um novo *container* – dedicado exclusivamente para o isolamento daquela execução em específico – sendo montado dentro de seu sistema de arquivos o código-fonte do código em julgamento, os casos de teste do problema e o *script* que realiza o controle de uso de recursos das submissões. Cada *container* é identificado com um nome temporário, que também é utilizado para o diretório temporário que armazena os arquivos necessários para o julgamento, descritos acima.

Figura 6 – Arquitetura proposta para a adição de *containers* no processo de julgamento



Fonte: (AUTORA, 2019).

Conforme comentado no Capítulo 2, os Docker *containers* acabaram tornando-se uma alternativa altamente popular para a criação e gerenciamento de *containers*, por proverem uma API bastante amigável e robusta. Por essas razões, inicialmente, optou-se por utilizar esse orquestrador para o gerenciamento dos ambientes virtualizados deste trabalho.

Pela sua popularização, diversas linguagens de programação fornecem imagens Docker oficiais de seus compiladores, para diversas versões. Ao instanciar um *container* com a imagem Docker oficial da linguagem, o mesmo estará corretamente configurado com a versão do compilador requerida. Por essa praticidade e segurança – uma vez que imagens

oficiais são constantemente atualizadas pelos seus mantenedores – o emprego destas imagens neste projeto foi avaliado.

Quadro 3 – Relação das Imagens Docker utilizadas para os primeiros testes

Linguagem	**Imagem**	**Versão Compilador**	**Compilador Atual**
C/C99/C++11	gcc:5	gcc 5	gcc 4.8.5
C++17	gcc:7.3	gcc 7.3.4	gcc 7.3.0
C#	mono:5.10	mono 5.10	mono 5.10.1.20
Go	golang:1.8.7	go 1.8.7	go 1.8.1
Haskell	haskell:7.8	ghc 7.8	ghc 7.6.3
Java 7	openjdk:7	OpenJDK 1.7.0	OpenJDK 1.7.0
Java 8	openjdk:8	OpenJDK 1.8.0	OpenJDK 1.8.0
JavaScript	node:8.4	nodejs 8.4	nodejs 8.4.0
Kotlin	urioj-kotlin:1.2	kotlinc 1.3	kotlinc 1.2.10
Lua	urioj-lua:5.2	lua 5.2.4	lua 5.2.3
Ocaml	urioj-ocamlc:4.05	ocamlc 4.05	ocamlc 4.01.0
Pascal	urioj-fpc:2.6	fpc 2.6.2	fpc 2.6.2
Python 2	python:2.7	python 2.7.16	python 2.7.6
Python 3	python:3.4	python 3.4.10	python 3.4.4
Ruby	ruby:2.3	ruby 2.3	ruby 2.3.0
Scala	urioj-scalac:2.13	scalac 2.13	scalac 2.11.8

Fonte: (AUTORA, 2019).

A relação de imagens Docker oficiais utilizadas é mostrada pelo Quadro 3. A primeira coluna identifica as linguagens de programação e suas versões. A coluna “Imagem” identifica a imagem Docker utilizada, seguida pela coluna “Versão do Compilador”, que corresponde a versão instalada na imagem Docker. A última coluna apresenta a versão atualmente suportada pelo juiz do UOJ para cada linguagem (“Compilador Atual”). Todas as imagens Docker oficiais tem como base o sistema operacional Debian, variando entre as versões 9 e 10 do mesmo.

Algumas linguagens não possuem versão Docker oficial, o que fez ser necessário criar imagens próprias. Estas tiveram como base a imagem do Ubuntu 18.04, em que fez-se a instalação da versão do compilador mais próxima à versão necessária, atualmente suportada pelo UOJ. Optou-se por essa versão de sistema operacional, pois é a mais atualizada da distribuição Ubuntu, e inicialmente pareceu ser mais adequado à proposta. Somente a linguagem Pascal foi exceção, tendo como base o Ubuntu 14.04, já que não foi possível instalar uma versão do compilador próxima à instalada nos servidores do URI Online Judge. Definiu-se também, como forma de padronização, que as imagens personalizadas para o julgamento teriam a nomenclatura urioj-compilador:versão. É possível também observar que

não foi encontrada uma opção com a exata versão do compilador atualmente suportada a todas as linguagens nesta configuração, porque são versões antigas e por vezes não mais atualizadas pelos mantenedores da linguagem.

Todavia, após inicial discussão com a equipe de desenvolvimento do UOJ, alguns problemas nesta abordagem foram detectados de forma empírica. A combinação entre sistema operacional hospedeiro, sua respectiva arquitetura e versão do compilador utilizado pode influenciar nos resultado das execuções dos códigos. Levantou-se a hipótese de que mudanças em qualquer um desses aspectos da combinação poderiam acarretar na variação da resposta final da solução, além do tempo de execução. Estes dois pontos são cruciais em um serviço de julgamento, uma vez que um define a sentença da submissão e o outro a posição da solução em um *ranking*.

Dado que o URI Online Judge já corrigiu mais de 16 milhões de códigos, realizar o rejulgamento se torna inviável. Dessa forma, foi necessário garantir o mínimo de discrepância entre os resultados obtidos na atual organização do UOJ e no ambiente com o uso de *containers*. Para isso, realizou-se a avaliação do desempenho dos códigos-fonte das submissões sendo executados dentro de *containers* criados a partir das imagens oficiais das linguagens, a fim de verificar se os *runtimes* seriam iguais ou similares aos tempos atuais e assim confirmar ou não a hipótese levantada.

4.2 Análise do comportamento das submissões em *containers* oficiais

Definida esta análise, foram selecionadas 500 submissões *Accepted* da base de dados do URI Online Judge, de forma aleatória, para cada linguagem de programação suportada pela ferramenta. Coletou-se como informação o número de identificação da solução, arquivo-fonte, o seu tempo de *runtime* e também informações relativas ao problema, como seu número de identificação, tempo limite e memória limite. Com essas informações, foi possível reproduzir o processo de compilação e execução destas submissões, a fim de verificar os seus comportamentos dentro de *containers*.

As submissões selecionadas foram reexecutadas quatro vezes dentro dos ambientes virtuais Docker, sendo coletado o tempo de execução em cada repetição e também informações para identificação da submissão executada, como seu número de identificação e arquivo-fonte. Para as análises aqui apresentadas, os tempos de cada submissão representam a média destas quatro repetições. Esta medida foi comparada ao *runtime* salvo para a submissão e coletado na fase de criação da amostra em análise, com o objetivo de observar

possíveis diferenças nos resultados. Assim, a diferença observada foi definida por

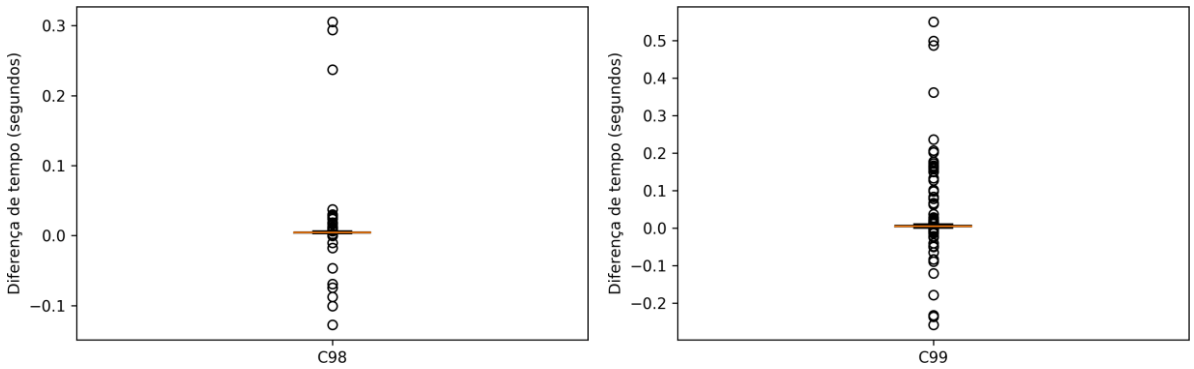
$$diferenca = media\_runtime_{container} - runtime_{host} \quad (1)$$

Em que a $media\_runtime_{container}$ é a média coletada pelas execuções nos ambientes Docker e $runtime_{host}$ é o tempo de execução que o código obteve ao ser julgado na atual configuração do juiz do UOJ. *Host* será a nomenclatura utilizada ao longo deste trabalho para indicar a atual organização das máquinas responsáveis pelo julgamento do URI Online Judge.

Iniciou-se a análise pelo desempenho das submissões codificadas na linguagem C. Além de ser a linguagem-base para os tempos de acréscimo nos *timelimits* dos problemas, esta é uma das linguagens mais utilizadas para submissões pelos usuários, juntamente com C++, Java e Python. A Figura 7 apresenta o *boxplot* da distribuição das diferenças nos tempos de execução encontradas entre as execuções dentro de *containers* e no *host* para as duas versões da linguagem. O *boxplot* apresenta uma caixa com subdivisões, que representam os quartis dos dados em análise. Estes quartis demonstram a distribuição dos valores dos resultados. Hastes superiores e inferiores indicam o maior e o menos valor da distribuição, respectivamente. A linha colorida representa a mediana dos dados, que também representa o segundo quartil, e os círculos exteriores indicam os *outliers*, que são valores que excedem os limites da distribuição.

Figura 7 – Distribuição da diferença do tempo de execução das submissões entre *host* e *container* da linguagem C

(a) C98 (b) C99



Fonte: (AUTORA, 2019).

A partir da análise dos gráficos supracitados, pode-se perceber que as diferenças não apresentam uma distribuição constante, justificado pela grande quantidade de *outliers*. Também nota-se que há certo grau de diferença entre os tempos de execução, mesmo que

pequenos, demonstrando que há uma variabilidade entre os ambientes. As medianas destas distribuições é de 0,0045 segundos para C98 e 0,0047 segundos para C99.

A fim de fazer uma análise estatisticamente correta dos dados coletados, aplicou-se um teste de hipótese. O primeiro ponto a ser observado para aplicar teste de hipótese em uma avaliação é relacionado ao formato da distribuição dos dados. A correta aplicação do teste depende muito de como os dados se distribuem e se eles obedecem a uma distribuição específica, podendo tender ou não à distribuição normal. Para definir se a distribuição dos dados analisados aqui seguem uma normal, o teste *Kolgomorov Smirnov (K.S. Test)* foi utilizado. As hipóteses levantadas para este teste foram:

- H_0: As diferenças nos tempos de execução seguem uma distribuição normal
- H_1: As diferenças dos tempos de execução não seguem uma distribuição normal

Também é necessário definir um nível de significância, denotado pela letra grega α, que tem por objetivo definir a probabilidade que a hipótese nula tem de ser rejeitada quando ela é verdadeira (MORETTIN; BUSSAB, 2017). Este valor de significância é então comparado com o *p-value*, que é a probabilidade de se obter um determinado resultado esperado na amostra observada. Se *p-value* for menor que o valor de α, rejeitamos a hipótese nula. Um valor popularmente atribuído em aplicações estatísticas para $\alpha = 0{,}05$, mesmo aplicado neste teste.

O *K.S. Test* retornou um *p-value* = $1{,}8540 \cdot 10^{-109}$ para C98 e *p-value* = $2{,}4553 \cdot 10^{-99}$ para C99, valores estes bem menores que o atribuído a α, o que fez com que H_0 fosse rejeitada para ambas as versões. Assim, provou-se que a distribuição dos dados em análise não segue a distribuição normal. Essa informação indica que, na amostra em análise, há muitos valores destoantes, podendo ser muito maiores ou menores que a maioria. Isso faz com que a média dos dados não corresponda de fato ao ponto médio da distribuição, já que estes máximos e mínimos alteram o seu resultado. Por isso, foi necessário utilizar um teste não-paramétrico para essa avaliação. Quando à distribuição da amostra em análise não segue uma normal, é necessário utilizar este tipo de teste para a análise de hipóteses, que tem como medida de referência à mediana (ponto médio da ordenação de valores de uma distribuição).

Sabendo informações quanto ao formato da distribuição, o próximo passo foi definir o teste de hipótese a ser aplicado para a análise dos resultados. Optou-se pelo uso do teste de *Wilcoxon* (WILCOXON, 1945), pois analisa duas amostras e tem como sua hipótese nula (H_0) a avaliação de que não há diferença entre elas. Dado que este é o objetivo da análise aqui

realizada, isso justifica sua escolha. Assim, as hipóteses definidas foram:

- H_0 : Os tempos das amostras não apresentam diferenças entre os valores
- H_1 : Os tempos das amostras apresentam diferença

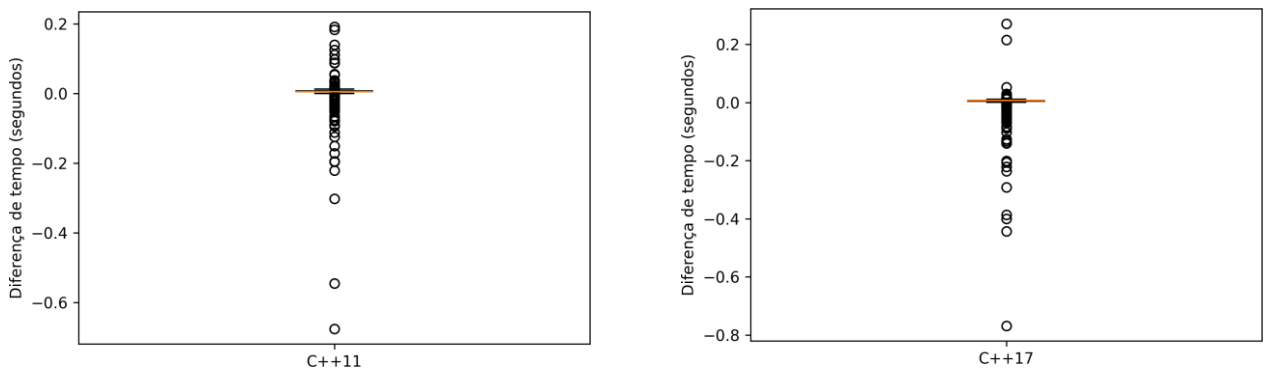
O valor de significância também foi definido em $\alpha = 0{,}05$. Esta avaliação foi empregada em todas as análises realizadas neste trabalho em relação as diferenças nos tempos de execução, uma vez que a aplicação do *K.S. Test* provou que todas as distribuições analisadas não seguem a distribuição normal.

Aplicado o teste de *Wilcoxon* nos resultados para a versão 98 da linguagem C, obteve-se um *p-value* = $1{,}15416 \cdot 10^{-74}$, e para a versão 99 *p-value* = $3{,}67635 \cdot 10^{-57}$. Isto permite concluir, com 95% de confiança, de que há uma diferença entre os tempos de execução e ela pode ser entendida como uma diferença considerável, já que é quase improvável encontrar um caso em que a diferença foi nula.

As submissões na linguagem C++ também apresentaram diferenças nos tempos de execução. Por sua vez, a versão 17 apresentou uma menor diferença em relação às analisadas até então. É possível perceber pelos gráficos da Figura 8 uma maior concentração de valores próximos a 0,0 segundos de diferença, porém não em sua totalidade. A mediana destas distribuições foi de 0,0055 segundos para C++11 e 0,007 segundos para C++17. Os resultados da aplicação do teste de *Wilcoxon* também negam as hipóteses de não haver diferença de desempenho entre os ambientes, sendo *p-value* = $2{,}18932 \cdot 10^{-41}$ para C++11 e *p-value* = $1{,}17436 \cdot 10^{-30}$ para C++17.

Figura 8 – Distribuição da diferença do tempo de execução das submissões entre *host* e *container* da linguagem C++

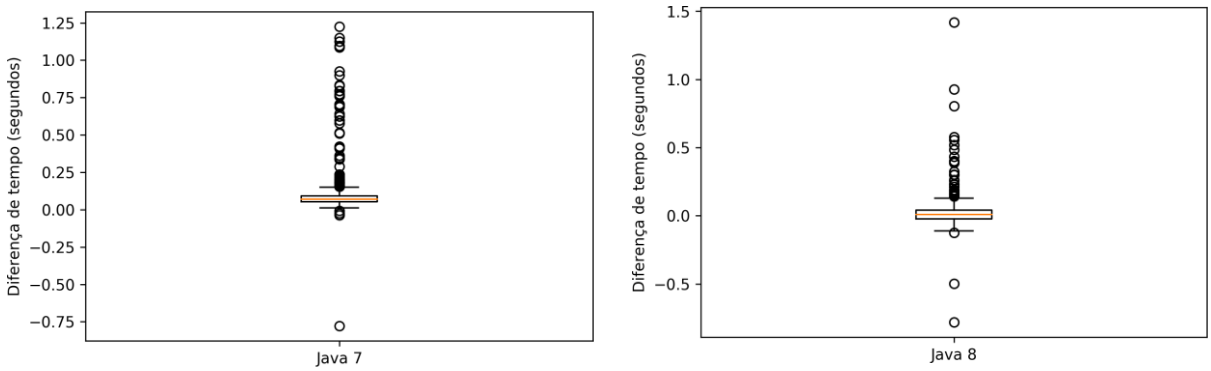
(a) C++11 (b) C++17



Fonte: (AUTORA, 2019).

Figura 9 – Distribuição da diferença do tempo de execução das submissões entre *host* e *container* da linguagem Java

(a) Java 7 (b) Java 8

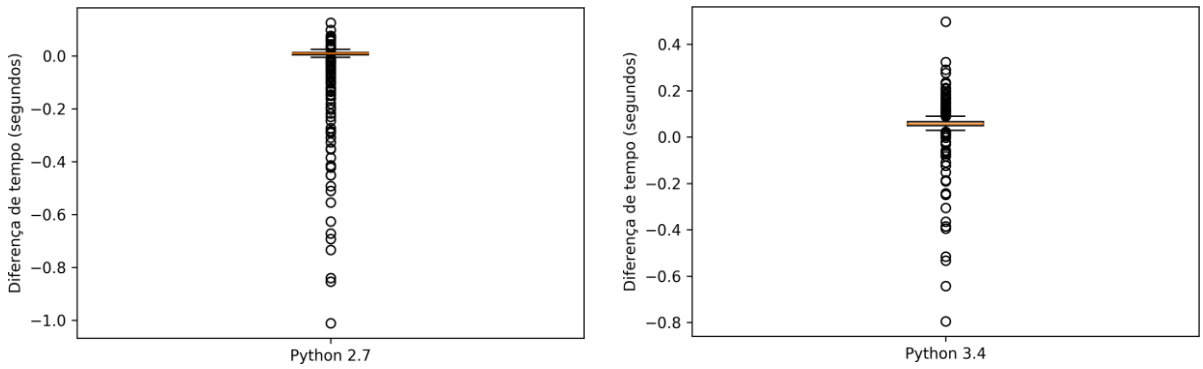


Fonte: (AUTORA, 2019).

Os testes com a linguagem Java (Figura 9) também apresentaram diferenças nos tempos de execução, com *outliers* de valores maiores que os demonstrados pelas linguagens analisadas anteriormente. Também observa-se pelos gráficos que as diferenças distanciaram-se mais do valor mediano da distribuição. A hipótese de não haver diferença nos tempos de execução também foi negada, mas os valores *p* resultantes foram menores que os apresentados anteriormente, sendo eles $2{,}87511 \cdot 10^{-82}$ para Java 7 e $3{,}22751 \cdot 10^{-06}$ para Java 8. Isso demonstra que há uma maior probabilidade de não haver diferença nos tempos de execução desta linguagem sendo executada em *containers* em relação a C e C++.

Figura 10 – Distribuição da diferença do tempo de execução das submissões entre *host* e *container* da linguagem Python

(a) Python 2.7 (b) Python 3.4

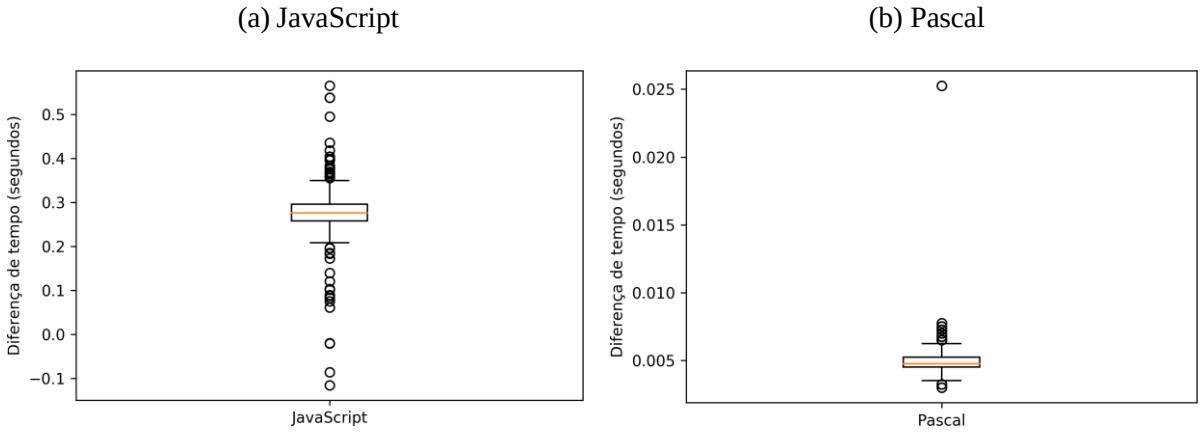


Fonte: (AUTORA, 2019).

As diferenças na linguagem Python demonstraram que os tempos de execução dentro do *container* foram menores que os registrados pelo sistema atual, uma vez que a

maioria dos *outliers* apresentados pelos gráficos da Figura 10 marcam valores abaixo de zero. Ademais, pode-se visualizar uma grande quantidade de *outliers*, o que ajuda a demonstrar a oscilação da distribuição dos valores. O teste de hipótese relativo a diferença nos tempos de execução ser nula também foi negado, com *p-values* iguais a $1{,}82697 \cdot 10^{-15}$ para Python 2.7 e $1{,}28032 \cdot 10^{-61}$ para Python 3.4. A mediana da distribuição das diferenças na versão 2.7 foi igual a 0,00975 e na versão 3.4 igual a 0,05650.

Figura 11 – Distribuição da diferença do tempo de execução das submissões entre *host* e *container* das linguagens JavaScript e Pascal



Fonte: (AUTORA, 2019).

As linguagens JavaScript e Pascal apresentaram, respectivamente, o pior e o melhor resultado de todas as análises. Como mostra o gráfico da Figura 11a, as diferenças entre os ambientes de execução para a linguagem JavaScript foram altas, sendo a mediana das diferenças igual a 0,27575 segundos. Há poucos valores que apresentaram diferença nula, e eles são apresentados como *outliers* no gráfico. O teste de *Wilcoxon* ajuda a provar ainda mais que a probabilidade de selecionarmos um valor nulo da amostra é baixa, sendo o *p-value* = $1{,}42548 \cdot 10^{-83}$. Pascal, por outro lado, apresentou a maior estabilidade entre todos os testes. Como mostra a Figura 11b, a maioria das submissões apresentou uma diferença muito próxima a 0,0 segundos, mas nenhuma chegou a diferença nula, justificado pelo valor ínfimo de p *p-value* = $7{,}30717 \cdot 10^{-84}$. Metade dos valores da distribuição apresentaram valores menores ou iguais há 0,00475 segundos.

Tabela 1 – Valores médios das diferenças no tempo de execução em *containers* com imagens oficiais

	C#	Go	Haskell	Kotlin	Lua	Ruby	Scala
Média	0,017	0,270	0,015	0,069	0,011	0,031	0,105
Mediana	0,011	0,223	0,013	0,017	0,006	0,032	0,082
D.P.	0,075	0,224	0,028	0,239	0,069	0,018	0,192

Fonte: (AUTORA, 2019).

A Tabela 1 sumariza os valores de médias, medianas e desvio padrão (D.P.) do restante das linguagens. Optou-se por não realizar discussões acerca das mesmas uma vez que seus comportamentos assemelham-se com as análises supracitadas. Os gráficos destas podem ser encontrados no Anexo A. É possível concluir, através de todas as considerações feitas até então, que as diferenças de tempo de execução das submissões nos *containers* criados a partir das imagens Docker oficiais das linguagens são substanciais e instáveis, dado que apresentam variabilidade entre as linguagens.

Além dos tempos de execução serem próximos, foi necessário garantir que as soluções receberiam a mesma resposta que receberam em suas execuções primárias. É possível que sejam apresentadas respostas diferentes, dado que as características técnicas da máquina em que os códigos são executados também influenciam neste comportamento. Conforme relatado anteriormente, a amostra destes testes é composta por 500 submissões, por linguagem, que foram aceitas no processo de julgamento. Por todas submissões em análise possuírem a mesma resposta, torna-se fácil detectar anomalias nas avaliações.

Houveram variações de respostas nas linguagens C98, C99, C++17, Go, Haskell, Java 7, JavaScript, Kotlin e Python 2.7. Com exceção das linguagens C++17, JavaScript e Python 2.7, todas estas apresentaram poucos casos em que a submissão estourou o limite de tempo para execução (cerca de 1,2% do total das submissões). C99, C++17 e JavaScript apresentaram alguns casos com erros em tempo de execução. Estes resultados podem indicar que as diferenças encontradas na combinação SO + compilador apresentadas pelas imagens oficiais em relação ao sistema atual do UOJ influenciaram nestes resultados. Por fim, Python 2.7 apresentou uma submissão que levou *Memory Limit Exceeded*. Os comportamentos apresentados nesta observação não apresentaram o desempenho esperado, pois mesmo que em baixo número, alterações nas respostas aconteceram.

4.3 Análise do comportamento das submissões em *containers* personalizados

Uma vez que as imagens oficiais apresentaram certas diferenças nos tempos de execução dos códigos da amostra e também nas respostas esperadas, optou-se por realizar este mesmo teste de desempenho em imagens Docker configuradas com as mesmas características técnicas das atuais máquinas de julgamento do URI Online Judge. Este teste tem por principal objetivo, além de visar à garantia de baixa variabilidade entre os tempos de execução das submissões, testar a afirmação relacionada à necessidade de manter o ambiente de execução com características similares para apresentar os mesmos resultados.

As novas imagens Docker foram criadas a partir da imagem do Ubuntu 14.04, versão 32 bits, e buscou-se por instalar as exatas versões de compiladores atualmente suportadas pelo sistema de julgamento do UOJ e já discutidas pelo Quadro 2. O Código 1 exemplifica o *dockerfile* responsável pela criação de uma imagem Docker personalizada e o Código 2 a chamada para criação da imagem, a partir do *dockerfile* correspondente.

Código 1 – *Dockerfile* da imagem personalizada para a linguagem C

```
1 FROM i386/ubuntu:14.04
2
3 RUN apt-get update
4 RUN apt-get install -y software-properties-common
5 RUN sudo add-apt-repository ppa:ubuntu-toolchain-r/test
6 RUN apt-get update
7 RUN apt-get install –y gcc g++ bc
```

Código 2 – Chamada Docker para criação de uma imagem

```
docker build –t urioj-gcc:4.8 –f gcc-4.8.dockerfile .
```

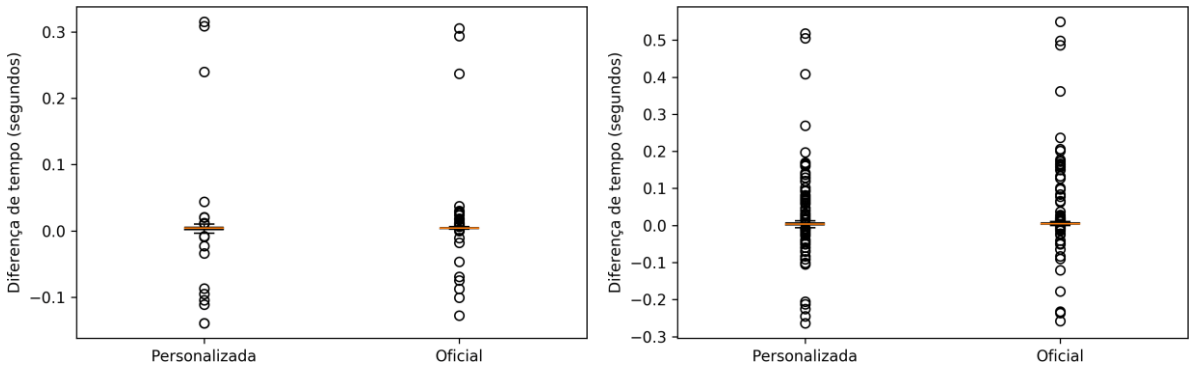
Onde $-t$ identifica a tag da imagem e $-f$ o arquivo *dockerfile* a ser utilizado.

A análise dos gráficos comparativos das duas versões da linguagem C (Figura 12) mostra que o desempenho das execuções na imagem personalizada apresentou menor variabilidade em relação à imagem oficial, principalmente em C98 (Figura 12a). Aplicando o teste de *Wilcoxon* para a hipótese nula de que não haveriam diferenças nas execuções entre a imagem personalizada e o *host*, a mesma permaneceu negada com um *p-value* = $3{,}21880 \cdot 10^{-71}$ para C98 e *p-value* = $4{,}47532 \cdot 10^{-42}$ para C99. Esses valores demonstram que a há uma probabilidade menor que 0,1% de selecionarmos um dado aleatório na amostra e este não

apresentar diferenças. Entretanto, é possível considerar o resultado apresentado pela imagem Docker personalizada mais adequado em relação ao apresentado pela imagem oficial, dado que a probabilidade de encontrarmos diferenças nulas aumentou nesta organização, apesar de ainda demonstrar valores pequenos.

Figura 12 – Comparação da distribuição de diferenças nos tempos de execução entre imagem Docker oficial e personalizada para a linguagem C

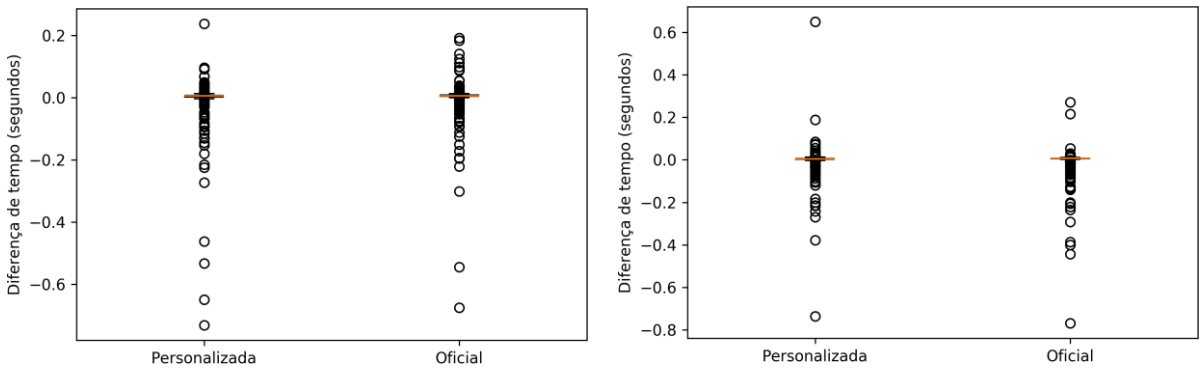
(a) C98 (b) C99



Fonte: (AUTORA, 2019).

Figura 13 – Comparação da distribuição de diferenças nos tempos de execução entre imagem Docker oficial e personalizada para a linguagem C++

(a) C++11 (b) C++17



Fonte: (AUTORA, 2019).

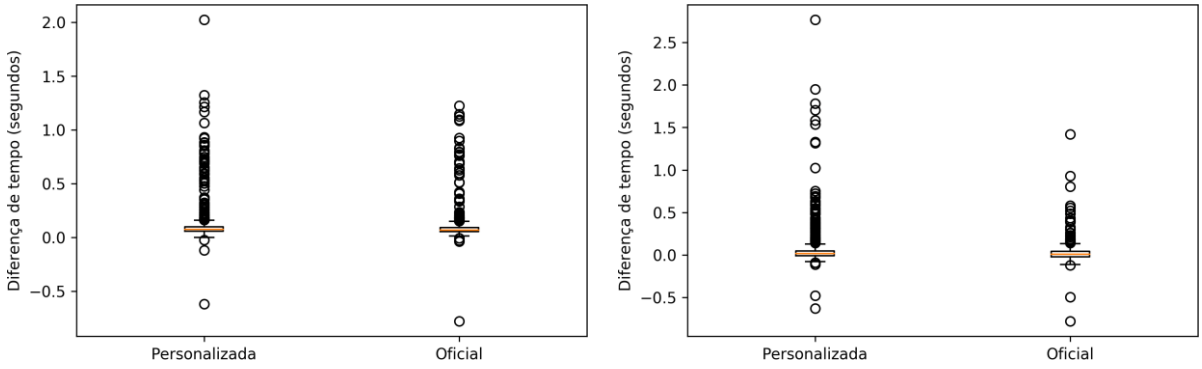
O desempenho apresentado pela linguagem C++ em suas duas versões também apresentou menor variabilidade na imagem personalizada. Apesar de também não negarem a hipótese nula no teste de *Wilcoxon* – apresentando *p-value* = $8{,}43686 \cdot 10^{-37}$ na versão 11 e *p-value* = $8{,}80145 \cdot 10^{-32}$ na versão 17 – as medianas das distribuições diminuíram. A mediana em C++11 passou de 0,00550 segundos para 0,00524 segundos, e em C++17 de 0,0700 segundos para 0,005 segundos, o que permite concluir que as execuções no *container* criado

a partir da imagem personalizada demonstraram comportamentos mais similares ao encontrado atualmente em comparação com o uso da imagem Docker oficial da linguagem.

O desempenho da linguagem Java nas imagens personalizadas demonstrou resultados diferentes dos até então analisados. A mediana da distribuição da diferença observada em Java 7 (Figura 14a) foi de 0,074751 segundos para a imagem personalizada e de 0,06975 segundos na imagem oficial. Com a versão 8 da linguagem (Figura 14b) os resultados não foram muito diferentes, também foi possível perceber um aumento no valor da mediana de 0,00862 segundos para 0,01412 segundos. Estes números demonstram-se mais altos do que as diferenças até então analisadas, que demonstravam diferenças na escala dos milissegundos. Ademais, os *outliers* apresentaram valores mais discrepantes e altos, chegando a uma diferença de mais de 2 segundos.

Figura 14 – Comparação da distribuição de diferenças nos tempos de execução entre imagem Docker oficial e personalizada para a linguagem Java

(a) Java 7 (b) Java 8

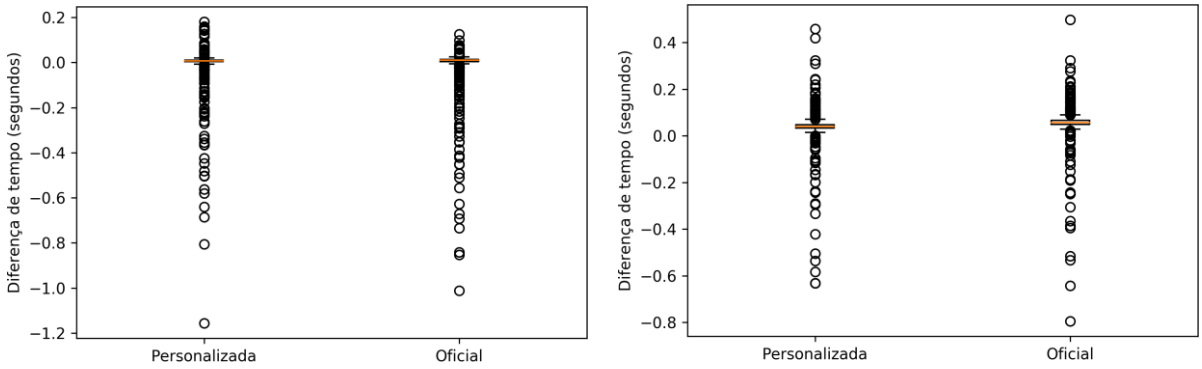


Fonte: (AUTORA, 2019).

A Figura 15 apresenta os resultados obtidos nos testes com a linguagem Python. Os *outliers* apresentados pelos *boxplots* das imagens personalizadas para ambas versões estão mais concentrados e próximos dos valores dos quartis das distribuições das diferenças. A mediana das versões caiu de 0,00975 segundos para 0,00775 segundos para a versão 2.7 e de 0,05650 segundos para 0,0395 segundos na versão 3.4. Os valores *p* para as versões foi de *p-value* = $5{,}16922 \cdot 10^{-19}$ para a versão 2 e *p-value* = $2{,}14420 \cdot 10^{-58}$ para a versão 3, demonstrando que o uso das imagens personalizadas adéqua-se melhor as necessidades do projeto uma vez que apresentou menor variabilidade nos tempos de execução.

Figura 15 – Comparação da distribuição de diferenças nos tempos de execução entre imagem Docker oficial e personalizada para a linguagem Python

(a) Python 2.7 (b) Python 3.4

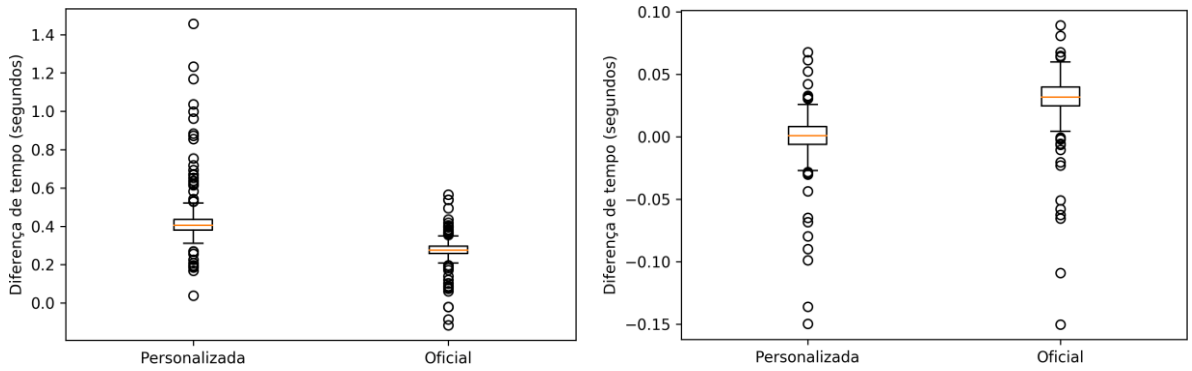


Fonte: (AUTORA, 2019).

Novamente, a linguagem JavaScript apresentou o pior resultado na análise. Destoando da maioria das linguagens, JavaScript (Figura 16a) apresentou acréscimo significativos nas diferenças de execução entre *host* e imagem Docker personalizada. É visivelmente perceptível que esta nova configuração apresentou valores ainda mais altos que aqueles demonstrados pelo uso da imagem Docker oficial. Não houveram diferenças nulas, o que fez com que o teste de *Wilcoxon* retornasse um *p-value* = $1{,}2644 \cdot 10^{-83}$, e a mediana da distribuição foi de 0,40562 segundos, quase o dobro do valor apresentado anteriormente (que foi de 0,27575 segundos).

Figura 16 – Comparação da distribuição de diferenças nos tempos de execução entre imagem Docker oficial e personalizada para as linguagens JavaScript e Ruby

(a) JavaScript (b) Ruby



Fonte: (AUTORA, 2019).

Diferentemente do apresentado nas imagens Docker oficiais, Ruby foi o melhor resultado observado (Figura 16b). A linguagem apresentou um valor de mediana igual a

0,00087 segundos, demonstrando que pelo menos 50% das execuções apresentaram valores extremamente próximos a zero. Pode-se perceber pela caixa dos quartis do gráfico que toda a distribuição fica em torno da diferença nula, e que apresentou menos *outliers* comparado a outras linguagens. É notório também o ganho de desempenho em relação a imagem oficial. O *p-value* do teste de *Wilcoxon* foi igual a 0,03727.

Tabela 2 – Valores médios das diferenças no tempo de execução em *containers* com imagens oficiais

	C#	Go	Haskell	Kotlin	Lua	Pascal	Scala
Média	0,018	0,079	0,013	0,094	0,011	0,0001	0,121
Mediana	0,010	0,027	0,011	0,027	0,006	0,0045	0,072
D.P.	0,089	0,234	0,034	0,270	0,075	0,0170	0,247

Fonte: (AUTORA, 2019).

Os resultados das demais linguagens são sumarizados na Tabela 2. Assim como nos testes com as imagens oficiais, os gráficos podem ser encontrados no Anexo B deste trabalho. Apesar de os testes ainda mostrarem que haverá uma diferença nos tempos de execução dos códigos a serem executados em *containers*, as diferenças apresentadas pelas imagens personalizadas, em sua maioria, são valores mais condizentes para o contexto do projeto. Há uma variabilidade nos tempos de execução existente no próprio sistema atualmente utilizado para o julgamento do URI Online Judge e atestado por (SELIVON, 2016). Assim, os valores apresentados nos testes aqui discutidos estão adequados às necessidades do sistema, o que permitiu avançar para uma segunda fase da proposta abordada.

Em relação as respostas das submissões, novamente houveram linguagens que apresentaram alterações. C98, Go, Kotlin e Haskell apresentaram submissões que excederam o tempo limite de execução. Essa alteração é explicada pelos pequenos mas existentes acréscimos no tempo de execução apresentados nas execuções dos códigos em *containers*. Por representarem menos de 1,8% dos casos, é possível inferir que não apresentará um impacto significativo no julgamento. C++, por sua vez, apresentou em suas duas versões uma quantidade significativa de submissões que receberam *Runtime Error* como resposta. Analisando mais atentamente estas submissões, se percebeu que elas de fato apresentavam erros que deveriam ser detectados em tempo de execução. Os erros detectados eram relacionados a acessos indevidos de memória, com incorreto acesso de endereços de variáveis e ponteiros, o que deveria ter feito com que a submissão recebesse *Runtime Error* como resposta desde a sua primeira execução. Assim, pode-se considerar essas alterações de

respostas uma correção e melhoria no sistema.

4.4 Atualizações no sistema de julgamento

Com as imagens prontas, partiu-se para a atualização dos *scripts* responsáveis pela compilação e execução dos códigos, de forma a realizarem estas ações dentro de *containers*. Os *scripts* de compilação ficaram responsáveis por também realizar a chamada de criação do *container*, passando como parâmetros o caminho a ser montado dentro do *container* e também o nome do mesmo para identificação. A API Docker permite o gerenciamento de alguns aspectos relacionados ao uso de recursos da máquina hospedeira por parte deste *container*, possibilita montar arquivos do *host* no mesmo, durante todo o seu "ciclo de vida". Nas linguagens que não necessitam de compilação, o *script compile* ficou somente responsável pela instanciação do *container* para execução.

Atualmente, o URI Online Judge utiliza um subsistema para realizar o controle das execuções dos códigos-fonte enviados para correção. Todavia, logo nos primeiros testes de execução realizados com *containers*, notou-se incompatibilidades deste subsistema com este novo ambiente, por ambos usarem *namespaces* para o isolamento das execuções. Conforme relatado no Capítulo 2, não é possível criar um *namespace* a partir de outro sem que o espaço hospedeiro possua privilégios no sistema. Dado que fornecer acesso a privilégios de *kernel* pode gerar brechas de segurança, optou-se por atualizar a forma do controle de execuções e não fazer mais uso deste subsistema em questão, a fim de resolver esta incompatibilidade.

Assim, a partir desta conclusão, fez-se necessário a criação de uma nova forma de controle dos tempos de execução para as submissões que permitisse a execução dos códigos dentro de *containers* e que mantivesse o mesmo padrão já utilizado pelo juiz do UOJ. Para isso, foi criado um *script* em *bash* para ser utilizado em todas as execuções, para todas as linguagens. Este *script* é responsável por calcular o tempo de execução do programa, e define o tempo limite que este pode ser executado, além de converter os códigos de retorno das execuções para os padrões esperados pelo código de julgamento do UOJ.

Para o controle do tempo limite do problema, optou-se por fazer uso da função *timeout*, do *bash*, e o tempo de execução do programa é calculado através da função *date*. A chamada para a execução do código submetido é feita como um subprocesso controlado pela chamada *timeout*. Esta chamada de sistema controla o tempo de execução, porém não retorna o tempo despendido em si, retornando apenas códigos indicando se o tempo foi ou não excedido. Assim, a captura do tempo de início e fim da execução realizada por este *script*

também leva em conta o tempo de execução da chamada *timeout*. Considerar este tempo seria incorreto para o contexto de julgamento, já que deve-se retornar para o usuário o tempo que o código submetido por ele levou para executar, desconsiderando *overheads* do sistema. Para contornar este ponto, optou-se por descontar um valor padronizado de todas as execuções, de forma a desconsiderar os tempos adicionados pelo sistema.

Este valor foi determinado a partir da média de execuções da chamada *timeout* dentro de *containers* criados a partir da imagem de cada uma das linguagens suportadas pelo UOJ. Foram executadas 500 vezes a chamada *timeout 1 sleep 1* dentro de *containers* criados a partir de cada uma das 14 imagens, que resultou na média de 1,0038076016565332 segundos. Optou-se por arredondar esta média para 5 casas decimais, dado que esta é a precisão que o juiz salva as informações de *runtime* para as execuções. Assim, o valor de *overhead* do *timeout* foi assumido como 0,00381 segundos. Cabe ressaltar que, por se tratar de uma média de execução, o desconto desse valor pode gerar tempos de execução negativos. Quando estes casos são detectados, o tempo de execução é mostrado como 0,00 segundos.

Cada problema cadastrado no *Judge* possui, além dos limites de tempo de execução, limites de memória que o programa-fonte pode fazer uso para resolvê-lo. Conforme citado anteriormente, a API Docker permite a limitação do uso de recursos por parte do *container* em relação a máquina hospedeira. Essa característica foi um ponto positivo para essa nova organização, pois permitiu limitar a memória disponível para as execuções de forma padronizada. Assim, logo após a compilação do fonte, o limite de memória RAM que pode ser utilizado pelo *container* é atualizado para o limite definido pelo problema. Dessa forma, caso o processo do programa-fonte ultrapasse o limite de memória do *container*, o mesmo será abortado e terminado, e a solução receberá *Memory Limit Exceeded* (MLE).

Esse controle de memória será um adicional ao UOJ. Hoje, a resposta MLE não é retornada aos usuários pois não há uma forma padronizada de realizar o controle do uso de memória para todas as linguagens disponíveis. Assim, apesar do estouro de memória ser detectado para algumas linguagens, ele não é salvo com essa resposta em banco devido a essa falta de padronização. No atual cenário em produção, quando o limite de memória é extrapolado, o usuário recebe *Wrong Answer* ou *Runtime Error* para seu código, dependendo da linguagem. A partir do uso de *containers* para a execução dos códigos, será possível limitar de forma padronizada o uso de memória de cada submissão, o que virá a permitir retornar a resposta *Memory Limit Exceeded* para os usuários.

Este novo formato de organização relacionado às linguagens permitiu uma outra melhoria. Como discutido inicialmente, grande parte das versões dos compiladores

atualmente configurados nos servidores de julgamento do UOJ não são mais atualizadas pelos mantenedores. Fazer o *upgrade* destas versões na arquitetura atual, em que todos os compiladores encontram-se instalados na mesmo máquina, demandaria diversas configurações e nem seria garantido que estas mudanças não acarretariam em problemas entre as linguagens. Agora, com a adição destes ambientes virtualizados isolados para cada versão, será possível fazer atualizações das linguagens de forma controlada. Não obstante, de depreciar versões não mais atualizadas – como Java 7, por exemplo – e facilmente adicionar novas e atualizadas, podendo também manter estas antigas versões para possíveis necessidades de rejulgamento de códigos. Esta foi uma importante vantagem adicionada pelo uso de *containers* no UOJ.

4.5 Análise de *overheads* adicionados pelos *containers*

Com o código do juiz adaptado para as chamadas de *containers*, realizaram-se testes referentes aos possíveis *overheads* causados pelas chamadas a estes ambientes. Dado que está sendo adicionada uma camada extra de virtualização no processo, um certo grau de *overhead* já era previamente esperado, mas este seria aceitável caso apresentasse baixos valores, dado a gama de benefícios que a adição dos *containers* trará ao processo de julgamento.

Primeiro observou-se o tempo de execução das chamadas Docker em específico. Marcadores de tempo foram adicionados antes e depois das chamadas para os processos de compilação e execução dos códigos, a fim de obter os tempos destas. Para estes testes, se fez uso dos mesmos conjuntos de submissões utilizados nos testes de desempenho das linguagens em *containers*. Não foi constatada necessidade de ser selecionado um novo conjunto de submissões para este experimento, uma vez que o objetivo desta etapa foi de avaliar o desempenho do *container* dentro do sistema de julgamento.

As Tabelas 3 e 4 apresentam as médias, medianas e desvio padrão (D.P.) do *overhead* dos *containers* para todas as linguagens, sendo avaliados separados os processos de compilação e execução. Optou-se por apresentar estes dados em duas tabelas, uma agregando os resultados das linguagens que são compiladas (Tabela 3) e outra das linguagens interpretadas (Tabela 4) a fim de melhor distribuir estes dados e também facilitar a interpretação dos mesmos, já que nas linguagens interpretadas o processo de compilação envolveu somente a criação do *container*, enquanto nas compiladas o processo envolve a criação e execução de comandos.

Esses dados foram coletados a partir da realização de quatro repetições de 500

execuções de compilação e execução dentro de *containers*. Dado que os tempos para diferentes versões da mesma linguagem foram bem aproximados, estes foram unidos e feito suas médias, a fim de melhor apresentar os resultados. A unidade de tempo demonstrada é em segundos.

Tabela 3 – Valores médios dos resultados em linguagens compiladas.

		C	C++	C#	Go	Haskell	Java	Kotlin	Pascal	Scala
Compilação	Média	1,575	2.001	1,780	1,741	2,198	2,103	5,276	1,562	6,153
	Mediana	1.549	1,866	1,737	1,707	2,152	2,065	5,228	1,535	4,791
	D.P.	0,127	0,468	0,187	0,175	0,173	0,163	0,312	0,182	2,402
Execução	Média	0,571	0,574	0,585	0,586	0,595	0,587	0,564	0,598	0,586
	Mediana	0,574	0,578	0,594	0,596	0,605	0,596	0,556	0,599	0,600
	D.P.	0,036	0,036	0,031	0,031	0,031	0,031	0,034	0,020	0,035

Fonte: (AUTORA, 2019).

Tabela 4 – Valores médios dos resultados em linguagens interpretadas.

		Javascript	Lua	Python Ruby
Compilação	Média	1,081	0,929	0,991 0,909
	Mediana	1,050	0,915	0,967 0,890
	D.P.	0,136	0,075	0,112 0,093
Execução	Média	0,663	0,581	0,618 0,593
	Mediana	0,683	0,597	0,620 0,600
	D.P.	0,050	0,037	0,047 0,028

Fonte: (AUTORA, 2019).

A partir da análise das tabelas supracitadas, é possível perceber valores discrepantes entre tempos de compilação e execução, principalmente os apresentados pela Tabela 3. Conforme anteriormente comentado, estes tempos mais altos podem ser justificados pelo processo de compilação englobar a criação do *container* e a execução de comandos, enquanto no processo de execução das submissões somente são feitas chamadas de execução.

Nota-se também que as linguagens Kotlin e Scala apresentaram médias e medianas demasiada altas para o processo de compilação. Kotlin acaba sendo um caso ainda mais preocupante, pois o seu desvio padrão apresentou um valor baixo, indicando que os tempos variam pouco em relação à média apresentada por esta tabela.

Nas linguagens interpretadas, o tempo de compilação apresentado é mais baixo. No caso destas linguagens, os resultados demonstrados pela subdivisão Compilação da Tabela 4

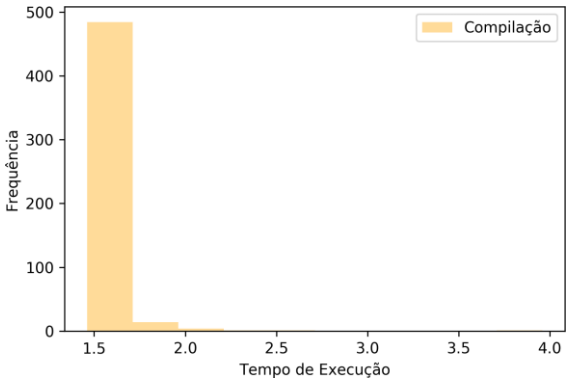
são exclusivamente da criação do *container* para cada submissão. Por ela, é possível inferir que o tempo de criação de *containers* Docker seja próximo há 1,25 segundos. Assim, pode-se dizer que o *overhead* adicionado ao processo de julgamento pela criação de *containers* para cada submissão será maior ou igual a este valor. Considerando-se que o processo de julgamento deve ser o mais otimizado o possível, este acréscimo mínimo de um segundo pode impactar de forma consideravelmente negativa o processo completo de julgamento.

Em contrapartida aos tempos de compilação, os tempos de execução apresentaram-se bastante estáveis. O tempo médio geral das linguagens, tanto compiladas quanto interpretadas, foi de 0,594 segundos, e todas apresentaram um desvio padrão bem próximo e estável, tendo ele um valor médio igual a 0,036.

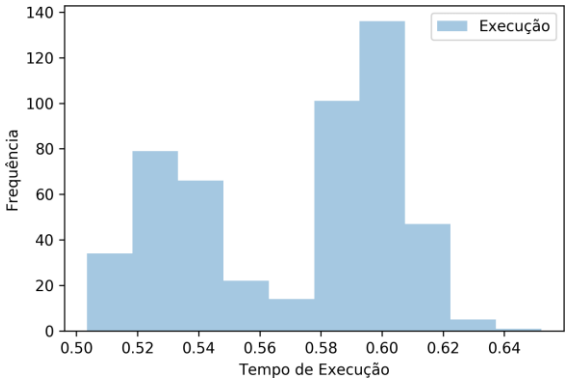
A fim de melhor discutir estes testes, a seguir, serão apresentados alguns histogramas destes. Eles demonstram as distribuições dos tempos médios dos processos de compilação e execução para as linguagens C, C++17, Java, JavaScript, Python 3, Kotlin e Scala. Optou-se por não discutir os gráficos de todas as linguagens, pois alguns resultados são bastante similares, como pode-se atestar pelas Tabelas elucidadas. Os histogramas não discutidos aqui podem ser encontrados nos anexos do trabalho.

Figura 17 – Histogramas do tempo médio das chamadas de execução dos *containers* Docker para linguagem C

(a) *Overheads* na compilação



(b) *Overheads* na execução



Fonte: (AUTORA, 2019).

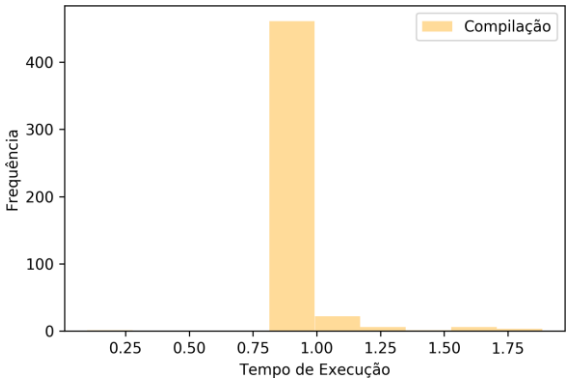
A Figura 17 demonstra a distribuição dos tempos médios dos testes realizados com *containers* para a linguagem C. A diferença de tempo entre os processos de compilação e execução demonstradas é evidente ao realizar a análise dos gráficos, conforme já apresentado de forma resumida pelas tabelas dos tempos médios das linguagens. Pode-se avaliar que o tempo despendido pelos processos de compilação trouxeram resultados mais estáveis

comparados aos processos de execução. Em contrapartida, os tempos médio de execução apresentam menores valores, e podem ser considerados adequados para as necessidades do julgamento, não sendo o mesmo mostrado pelos tempos de compilação.

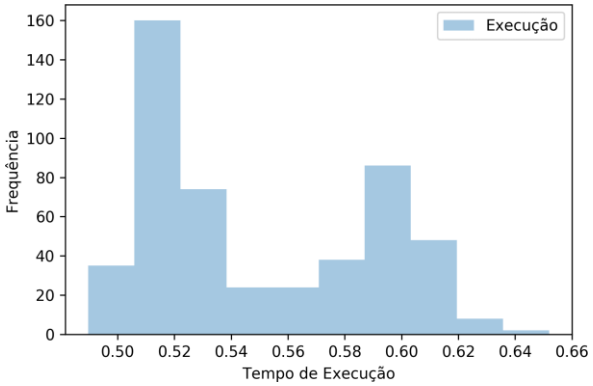
Python também apresentou resultados estáveis, mas altos, para o processo de compilação. Conforme já citado anteriormente, seus resultados de *overhead* na compilação representam o tempo de criação dos *containers* para cada submissão. Os tempos de *overhead* na execução (Figura18b) apresentam distribuição similar a linguagem C.

Figura 18 – Histogramas do tempo médio das chamadas de execução dos *containers* Docker para linguagem Python 3

(a) *Overheads* na compilação



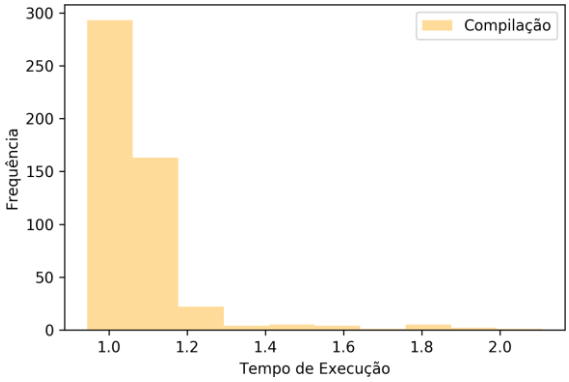
(b) *Overheads* na execução



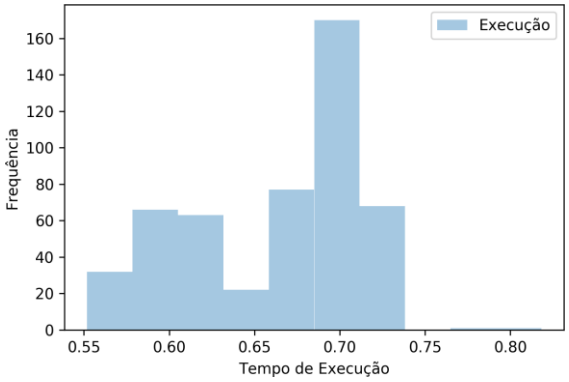
Fonte: (AUTORA, 2019).

Figura 19 – Histogramas do tempo médio das chamadas de execução dos *containers* Docker para linguagem JavaScript

(a) *Overheads* na compilação



(b) *Overheads* na execução



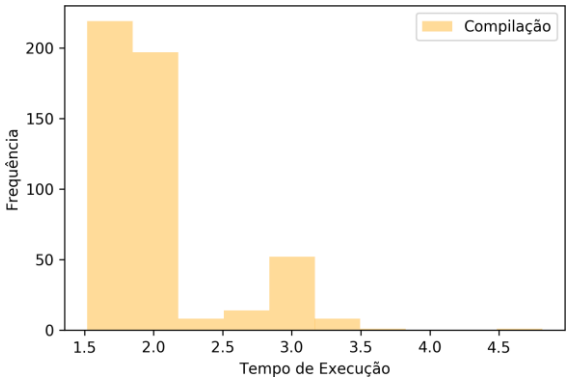
Fonte: (AUTORA, 2019).

JavaScript também apresentou tempos de *overhead* para o processo de compilação semelhantes aos da linguagem Python (Figura 19). Isso também se deve ao fato de que os códigos nessa linguagem são interpretados, e estes valores apresentados demonstram o tempo de criação do *container* para cada submissão. Desta forma, a linguagem demonstra tempos maiores nos *overheads* de execução, comparado às linguagens supracitadas. É possível perceber uma maior concentração de execuções com tempos maiores que 0,65 segundos, sendo este valor maior que os apresentados pelas linguagens anteriores.

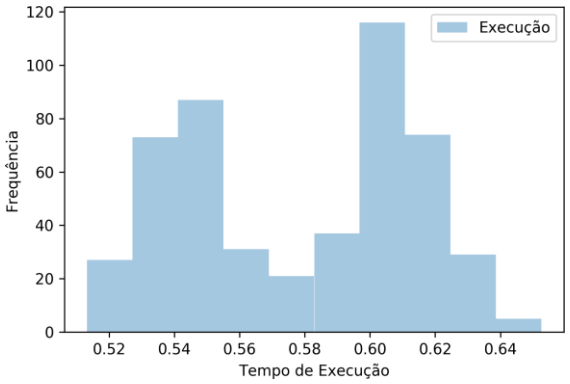
Os tempos médios dos processos de compilação e execução para a linguagem C++, demonstrados pela Figura 20, já demonstram valores maiores. Os *overheads* no processo de compilação apresentaram valores altos e instáveis comparados a linguagem C (que também é compilada). É possível observar que obtiveram-se *overheads* maiores que três segundos, o que pode vir a comprometer o tempo total de julgamento do sistema. Os tempos médios de execução apresentaram os mesmos intervalos que os da linguagem C, entretanto, eles estão mais igualmente distribuídos entre os extremos.

Figura 20 – Histogramas do tempo médio das chamadas de execução dos *containers* Docker para linguagem C++17

(a) *Overheads* na compilação



(b) *Overheads* na execução

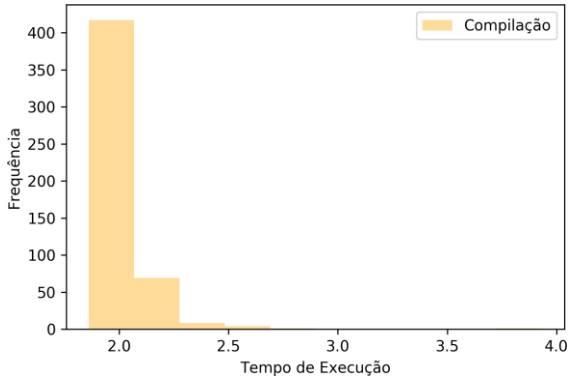


Fonte: (AUTORA, 2019).

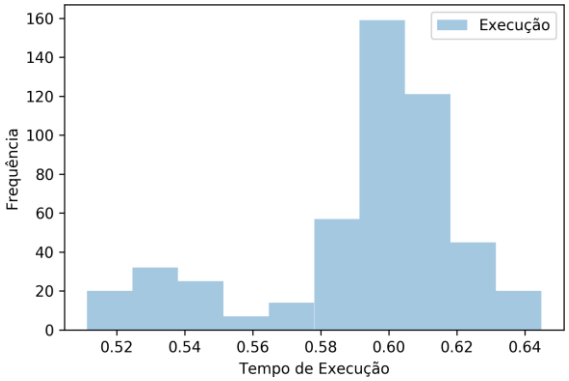
A linguagem Java (Figura 21) apresentou acréscimos no tempo de compilação maior que dois segundos, e nos tempos de execução este acréscimo se encontra em um intervalo entre 0,52 segundos e 0,64 segundos. A partir dessa análise, pode-se levantar a hipótese de que a criação dos *containers* contida dentro do processo de julgamento adicionará um grande *overhead* durante o processo completo.

Figura 21 – Histogramas do tempo médio das chamadas de execução dos *containers* Docker para linguagem Java

(a) *Overheads* na compilação



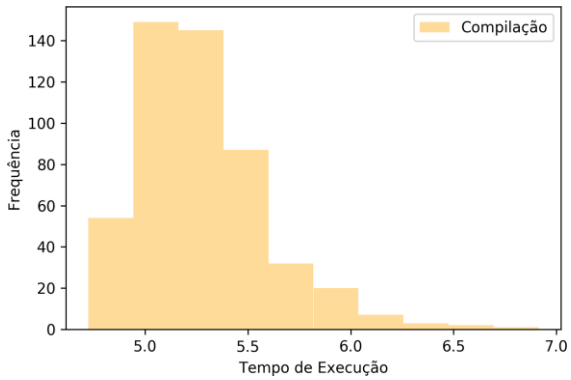
(b) *Overheads* na execução



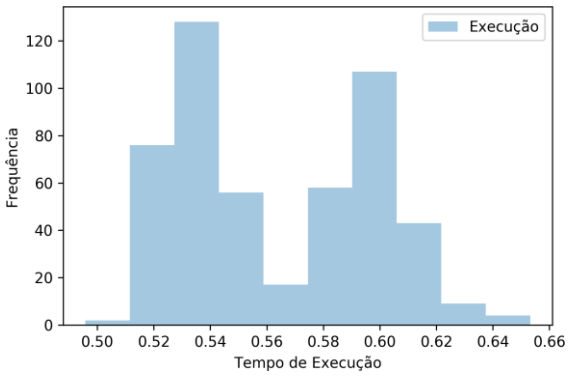
Fonte: (AUTORA, 2019).

Figura 22 – Histogramas dos tempos médios das chamadas de execução dos *containers* Docker para as linguagens Kotlin e Scala

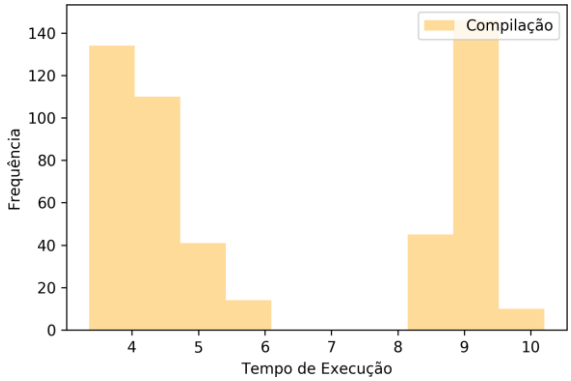
(a) *Overheads* na compilação em Kotlin



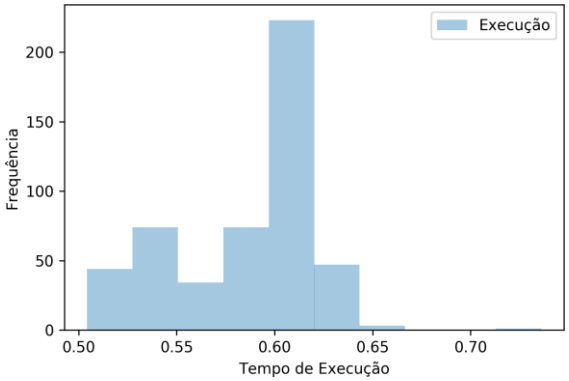
(b) *Overheads* na execução em Kotlin



(c) *Overheads* na compilação em Scala



(d) *Overheads* na execução em Scala



Fonte: (AUTORA, 2019).

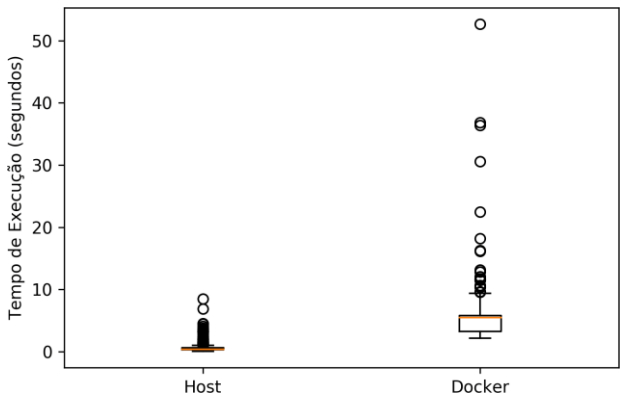
As linguagens Kotlin e Scala demonstraram os piores resultados destes testes (Figura 22). Seus tempos de compilação apresentaram valores exorbitantemente altos e

instáveis. O gráfico demonstrado pela Figura 22a auxilia na confirmação da interpretação feita a partir dos dados da Tabela 3 de que os tempos de compilação da linguagem Kotlin apresentaram pouca variação da média. Pode-se perceber também, pela Figura 22c, que Scala apresentou um número alto de ocorrências com tempo médio maior que 8 segundos. A análise destes resultados reforça a hipótese anteriormente levantada em relação a não adequação da organização inicialmente pensada para a adição de *containers* dentro do *pipeline* do processo de julgamento do UOJ. Os tempos de *overhead* nos processos de execução de ambas as linguagens agora analisadas se aproximam dos dados já apresentados neste trabalho.

4.6 Análise do tempo de execução do processo completo de julgamento

A próxima etapa foi definida pela realização de testes de desempenho do processo de julgamento como um todo. Esta tem por objetivo confirmar ou não a hipótese levantada em relação aos altos *overheads* de compilação. Aqui, marcadores de tempo foram adicionados no início e ao final de todo o processo de julgamento do sistema (que envolve, além dos processos de compilação e execução, a comparação das saídas geradas e a definição da sentença para o código avaliado). Para este teste, um novo conjunto de submissões foi selecionado. Desta vez, a amostra de teste foi composta por 1000 submissões, selecionada de forma randômica, não divididas por linguagens. Estes dados foram ressubmetidos, com um intervalo também randômico entre um e cinco segundos, a fim de simular o mais próximo o possível o real comportamento dos juízes em produção.

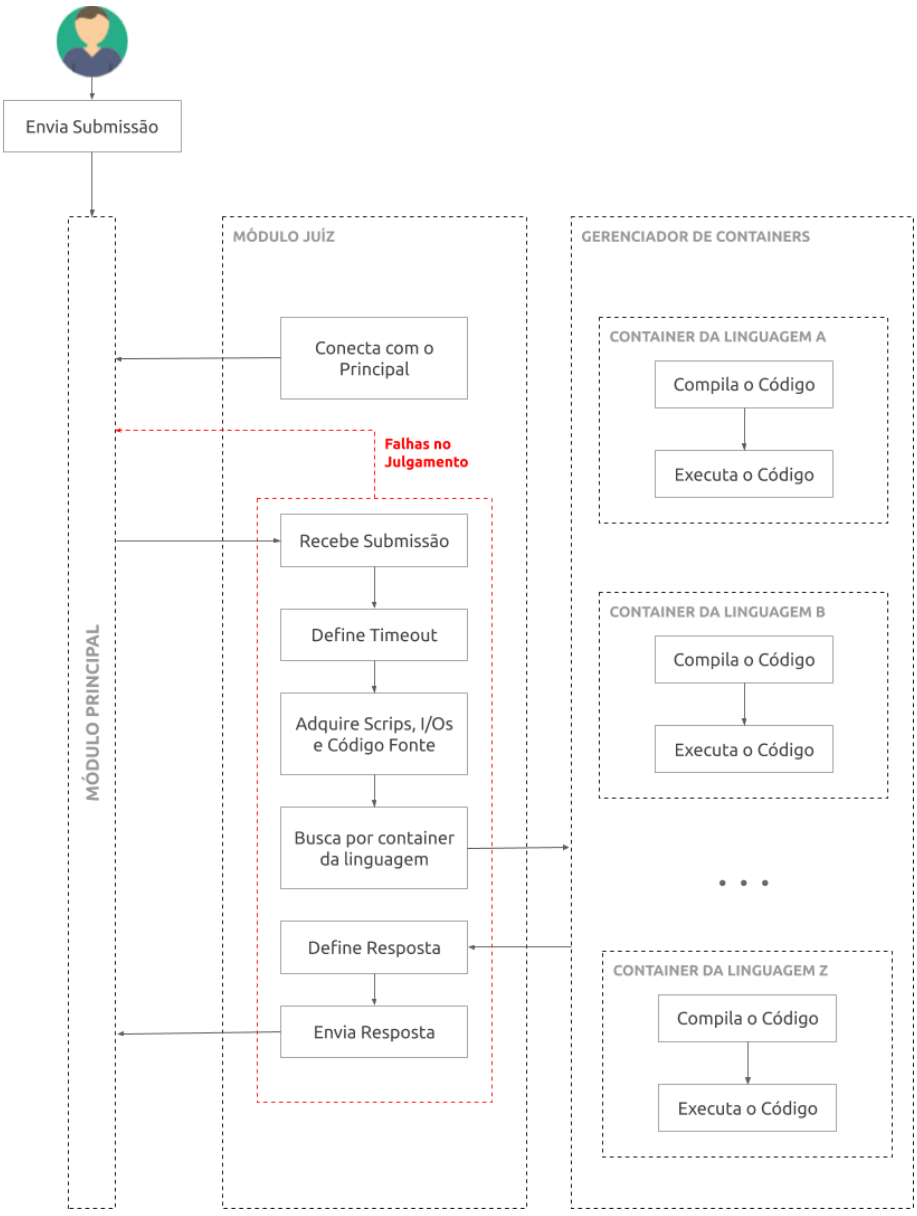
Figura 23 – Comparação entre o tempo total de julgamento no *host* e em *containers* por submissão



Fonte: (AUTORA, 2019).

A discrepância entre os resultados apresentados é visivelmente alta. O sistema com *containers* apresentou uma média de 5,04117 segundos para o julgamento completo, mediana 5,35470 segundos e um desvio padrão de 2,90173 segundos. Pode-se observar também que a distribuição das execuções em *containers* apresenta *outliers* altos, com máximo valor igual a 53,80013 segundos. Estes valores representam um acréscimo médio de 766,99% no tempo total de julgamento, porcentagem que torna esta organização inviável para o processo.

Figura 24 – Nova arquitetura proposta para a adição de *containers* no processo de julgamento



Fonte: (AUTORA, 2019).

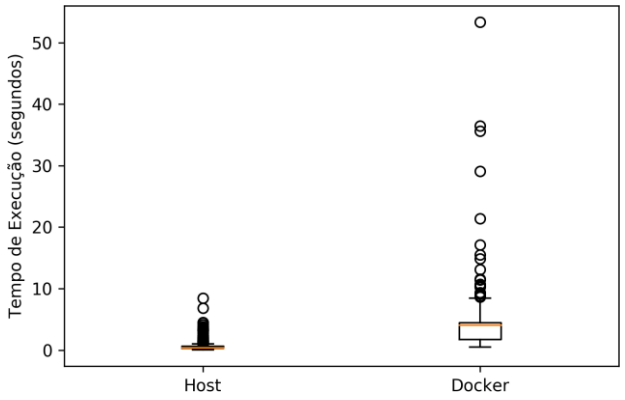
A partir disso, foi planejada uma nova arquitetura. Nesta nova abordagem, mostrada

pela Figura 24, são criados *containers* dedicados para cada compilador/linguagem. Como pôde ser observada nos testes de desempenho dos *containers*, a tarefa que mais leva tempo de processamento é a criação do *container*. Uma vez que o objetivo maior da adição deste ambiente virtualizado no processo de julgamento é o de isolar a execução do código propriamente dito, é possível fornecer um ambiente que seja compartilhado somente entre códigos de mesma linguagem – dado que o processo do julgamento acontece em formato sequencial dentro de um único juiz

Comenta-se em compilador, pois o compilador gcc-4.8 suporta a compilação e execução das linguagens C e C++, em suas versões C98, C99 e C++11. Estes *containers* são criados conforme sua necessidade, i.e. submissões na linguagem sejam enviadas, e ficam disponíveis enquanto estejam sendo utilizados. *Containers* ociosos são removidos, a fim de poupar recursos da máquina hospedeira, e criados novamente conforme apresentada necessidade.

Planejada esta nova arquitetura, realizou-se o teste de desempenho da mesma. Repetiu-se o mesmo procedimento utilizado com a organização anterior, a fim de poder realizar um comparativo adequado. Os resultados obtidos são demonstrados pela Figura 25.

Figura 25 – Comparação entre o tempo total de julgamento no *host* e em *containers* por linguagem



Fonte: (AUTORA, 2019).

Os tempos apresentados pela nova organização com *containers* Docker continuaram altos. Este novo teste apresentou uma média de 3,51514 segundos, mediana de 3,96019 com desvio padrão de 3,08653 segundos. Mesmo sendo menores que a organização anterior, estes valores representam um acréscimo médio de 505% em comparação a atual organização do sistema de julgamento, valor este que acaba por tornar inviável a aplicação de *containers* no processo de julgamento.

4.7 Adicionando um novo gerenciador de *containers*

A partir dos resultados apresentados pelos testes de sistema supracitados, partiu-se para uma nova análise. Conforme citado no Capítulo 2, existem diversos gerenciadores de *containers*, cada um com uma forma e características próprias. Concluir que a aplicação de *containers* para o processo de julgamento do URI Online Judge é inviável apenas com base nos resultados obtidos com o uso do Docker como gerenciador destes ambientes virtualizados não parecia correto. Assim, o processo de julgamento *containerizado* foi novamente analisado, agora com o uso de um outro gerenciador.

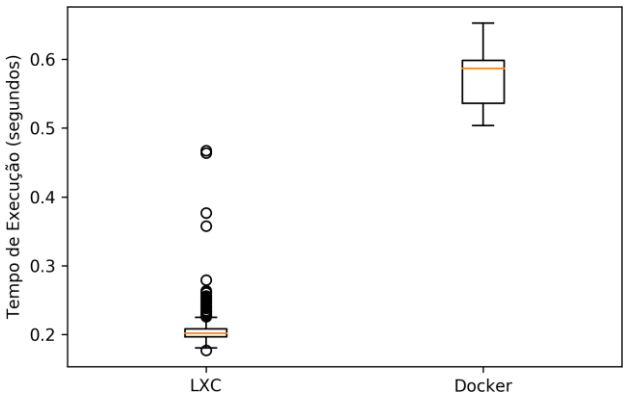
Para isso, optou-se pelos *Linux Containers* (LXC) para estes novos testes. Dado que cada gerenciador possui sua organização própria de imagens-base para os *containers*, a estrutura de virtualização teve de ser recriada para os padrões deste gerenciador. Aspectos de configuração e nomenclatura obedeceram o mesmo padrão já discutido na Seção 4.1, apenas sendo adaptado para o LXC.

Visto que o aspecto crítico para a não adoção dos *containers* Docker foi em relação aos tempos de execução das chamadas do próprio gerenciador, o primeiro teste realizado com o LXC foi relacionado a isso. Estes testes seguiram o mesmo protocolo descrito na Seção 4.5, a fim de poder realizar correta comparação dos resultados. Foram realizadas quatro repetições de 500 execuções de códigos nas linguagens C, C++, Java, JavaScript e Python. Estas linguagens foram utilizadas como base para a avaliação do uso deste gerenciador dado que C, C++, Java e Python são as linguagens mais utilizadas pelos usuários do UOJ e JavaScript por sempre ter apresentado os piores resultados nas avaliações com Docker.

O primeiro tempo de execução observado nos Linux *containers* foi em relação a sua criação. Assim como observado nos *containers* Docker, a inicialização do ambiente virtual é a operação mais custosa dentro de seu processo. A média de quatro criações de *containers* para cada uma das linguagens envolvidas neste teste foi de 2,386 segundos. Apesar de ser um número baixo – quando consideramos que trata-se do tempo de *boot* de um ambiente virtualizado – ter este tempo adicional por submissão é inviável.

Assim, o uso do LXC segue a organização mostrada pela Figura 24, em que são criados *containers* dedicados ao julgamento de cada linguagem suportada pelo juiz do URI Online Judge. Por conseguinte, as comparações apresentadas a seguir entre os desempenhos dos *containers* LXC e Docker foram feitas somente em relação aos tempos de execução das chamadas aos gerenciadores no processo de execução das submissões, uma vez que o processo de compilação com Docker envolve também o tempo de criação dos *containers*.

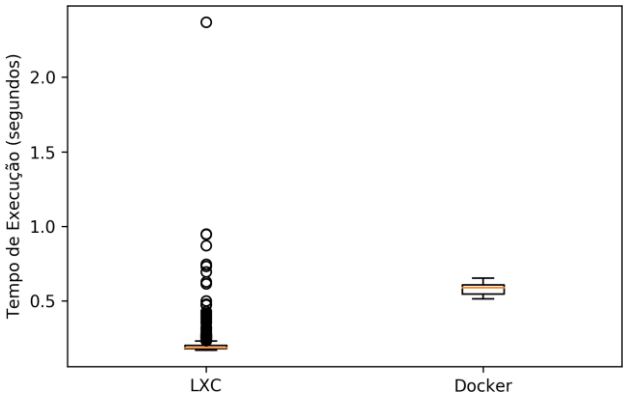
Figura 26 – Comparação entre os tempos de chamadas de execução de *container* Docker e LXC para a linguagem C



Fonte: (AUTORA, 2019).

Já nos primeiros resultados foi possível perceber o notório ganho de desempenho que o LXC oferece. A Figura 26 demonstra a comparação dos tempos de execução entre os dois gerenciadores para um *container* da linguagem C. Conforme é possível visualizar pela própria Figura, os tempos de chamadas de execução apresentados pelo LXC são 63% menores que em Docker, apresentam uma média de 0,206 segundos, com um desvio padrão de 0,081 segundos. Mesmo apresentando *outliers*, estes foram menores do que a distribuição total de tempo do Docker.

Figura 27 – Comparação entre os tempos de chamadas de execução de *container* Docker e LXC para a linguagem C++17

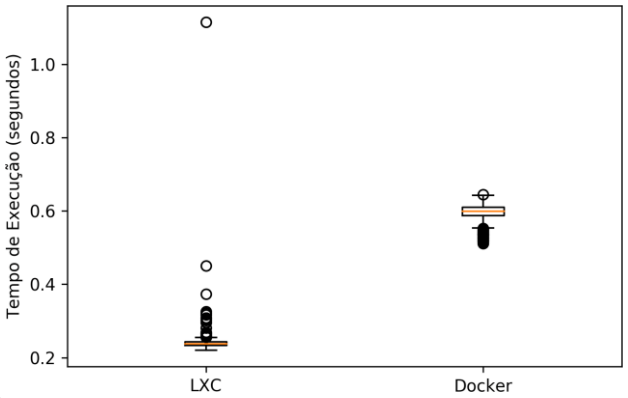


Fonte: (AUTORA, 2019).

As chamadas para o *container* LXC da linguagem C++17 também apresentaram notório ganho de desempenho. Mesmo apresentando *outliers* de maior tempo, a maior parte

dos valores ficou bem abaixo de toda a distribuição de valores de chamadas Docker. Houve uma melhoria de 62% nas chamadas com o uso do novo gerenciador. A média dos tempos foi de 0,219 segundos, com desvio padrão de 0,131.

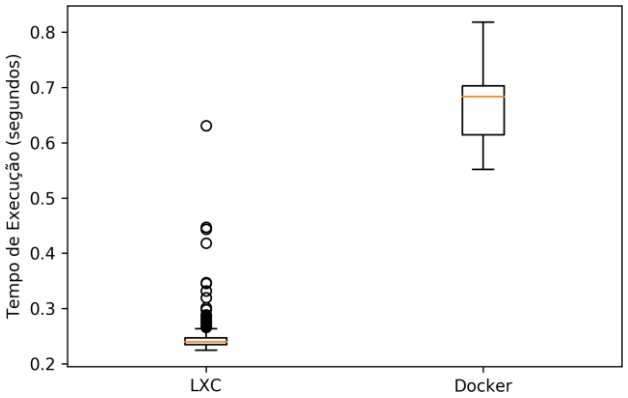
Figura 28 – Comparação entre os tempos de chamadas de execução de *container* Docker e LXC para a linguagem Java



Fonte: (AUTORA, 2019).

O ganho de desempenho também pôde ser atestado nas execuções com a linguagem Java (Figura 28). Apesar de ter apresentado uma execução com tempo elevado, a concentração dos resultados encontra-se em valores pequenos e bem abaixo dos resultados do Docker. A média dos tempos de chamadas de execução LXC para a linguagem foi de 0,241 segundos, com um desvio padrão de 0,079 segundos. Estes resultados apresentam um ganho de 59% em comparação com *container* Docker.

Figura 29 – Comparação entre os tempos de chamadas de execução de *container* Docker e LXC para a linguagem JavaScript



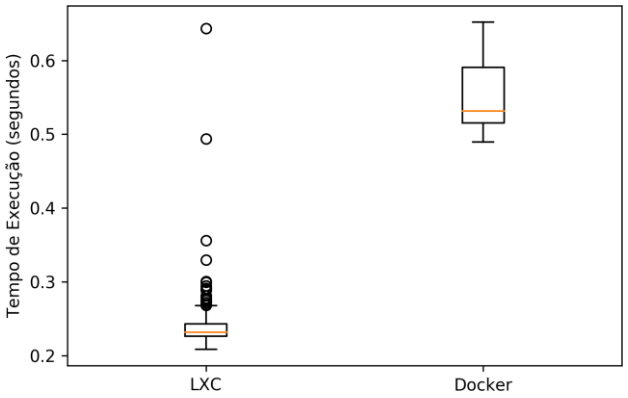
Fonte: (AUTORA, 2019).

Container LXC para JavaScript também apresentou melhor desempenho. O maior

tempo de chamada de execução apresentado pelo foi menor que a média geral do *container* Docker, como pode ser observado na Figura 29. A média para as chamadas ao *container* da linguagem JavaScript foi de 0,246 segundos, com desvio padrão de 0,140 segundos, representando 63% de ganho de desempenho.

O *container* LXC para a linguagem Python seguiu o padrão de resultados até então demonstrados pelas outras linguagens. Conforme mostra a Figura 30, apesar de apresentar um maior número de *outliers* comparado as outras linguagens, os tempos apresentados são cerca de 56% menores em relação ao Docker. A média dos tempos de chamada de execução foi de 0,232 segundos, com desvio padrão de 0,067 segundos.

Figura 30 – Comparação entre os tempos de chamadas de execução de *container* Docker e LXC para a linguagem Python 3



Fonte: (AUTORA, 2019).

4.8 Análise de desempenho das submissões em *containers* LXC

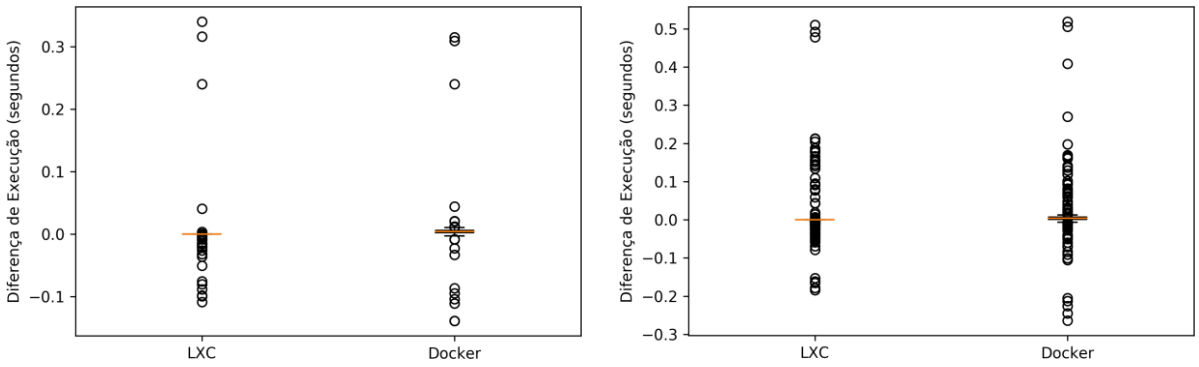
Com os resultados apresentados pelo desempenho nas chamadas de execução dos *containers* LXC, seguiu-se para a averiguação do desempenho das submissões propriamente ditas dentro destes novos *containers*. Estes testes também seguiram o mesmo protocolo utilizado para avaliação dentro de *containers* Docker, descrito na Seção 4.2, em que foi avaliada a diferença nos tempos de execução das submissões dentro do *container*, comparando-as com os tempos registrados no *host*.

O *K.S. Test* foi utilizado para verificar se a distribuição dos dados seguia uma distribuição normal, hipótese também negada para todos os testes realizados no LXC. Com isso, o teste de *Wilcoxon* também foi aplicado nesta análise, a fim de testar hipótese de diferença nula entre o ambiente LXC e o *host*. A Equação (1), apresentada para os testes com

Docker, também define como o cálculo da diferença foi feito para os testes realizados no LXC, que tem seus resultados apresentados a seguir.

Figura 31 – Comparação das diferenças no tempo de execução entre *containers* Docker e LXC para a linguagem C

(a) C98 (b) C99



Fonte: (AUTORA, 2019).

As diferenças nos tempos de execução das submissões em *container* LXC em comparação ao *host* apresentaram resultados satisfatórios. Apesar de ainda não apresentar diferença nula para todos os casos, a mediana da distribuição das diferenças em LXC foi de 0,0 segundos para as duas versões da linguagem C, o que indica que pelo menos 50% dos resultados ou não apresentaram diferenças nos tempos de execução ou apresentaram diferenças negativas.

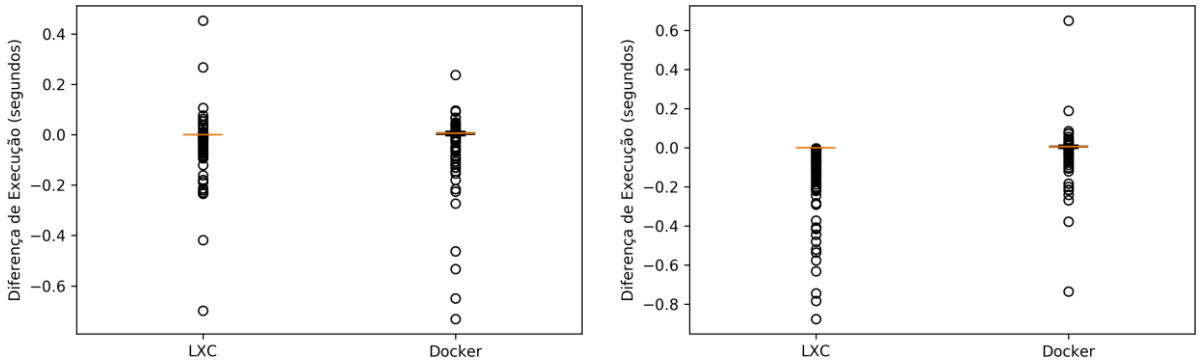
O *p-value* dessa distribuição, ao aplicar o teste de *Wilcoxon*, resultou em *p-value* = 0,00107 pra C98 e *p-value* = 0,00022 para C99. Isso demonstra que a probabilidade de selecionarmos uma amostra ao acaso e ela ser nula ainda é baixa, mas comparado ao resultado para o Docker há um aumento de cerca de 3,28%. A Figura 31 apresenta os *boxplots* comparativos da distribuição das diferenças entre LXC e Docker para as duas versões da linguagem. Através de suas análises, é possível visualizar tais melhorias discutidas.

Melhorias nos resultados repetiram-se nos testes realizados com a linguagem C++. Assim como em C, ambas versões apresentaram mediana igual a 0,0, indicando que a maioria dos códigos apresentou mesmo tempo de execução nos dois ambientes (*container* e *host*). Os valores p resultantes do teste de *Wilcoxon* para as duas versões da linguagem C++ foram *p-value* = $1{,}32091 \cdot 10^{-11}$ para a versão 11 e *p-value* = $4{,}04197 \cdot 10^{-21}$ para a versão 17. Pela

análise dos gráficos da Figura 32, nota-se que os *outliers* concentram-se em maior valor na parte inferior do gráfico, o que demonstra que os tempos de execução nos *linux containers* foram menores que no *host*. É possível perceber também que houve uma melhor distribuição das diferenças no LXC em comparação as apresentadas no Docker.

Figura 32 – Comparação das diferenças no tempo de execução entre *containers* Docker e LXC para a linguagem C++

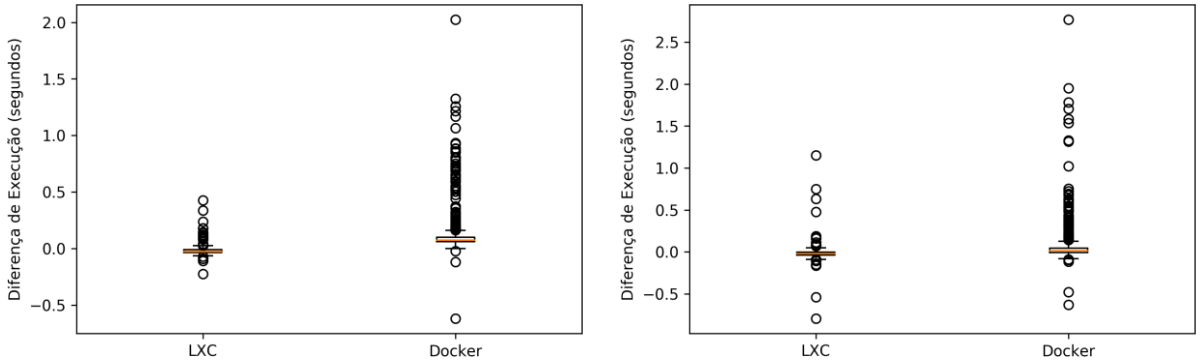
(a) C++11 (b) C++17



Fonte: (AUTORA, 2019).

Figura 33 – Comparação das diferenças no tempo de execução entre *containers* Docker e LXC para a linguagem Java

(a) Java 7 (b) Java 8



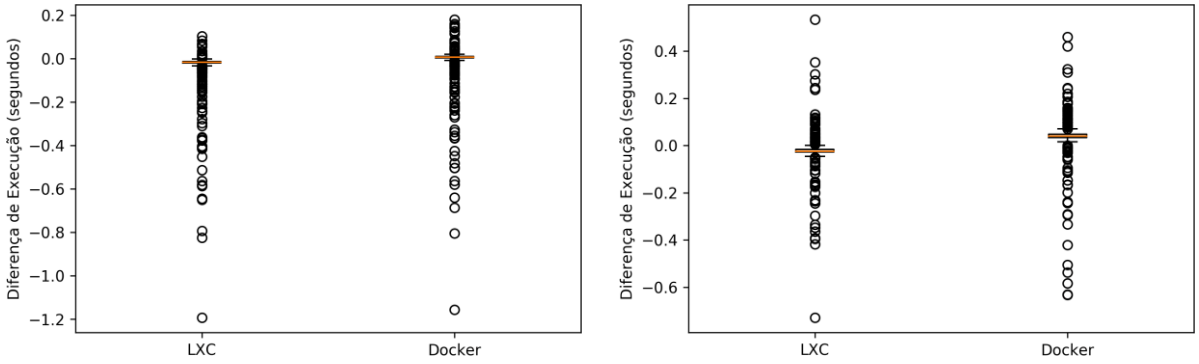
Fonte: (AUTORA, 2019).

A execução das submissões no *container* LXC para Java (Figura 33) também apresentou melhor resultado em comparação ao *container* Docker. A mediana da linguagem não foi zerada, mas resultou em valores muito próximos a zero. Para Java 7, a mediana das diferenças foi de $-0{,}02662$ segundos, com *p-value* = $1{,}67910 \cdot 10^{-55}$ e para Java 8 a mediana foi de $-0{,}02162$ segundos com *p-value* = $2{,}37444 \cdot 10^{-37}$, demonstrando que os tempos de

execução nos *linux containers* apresentaram-se mais curtos que no *host* e maior probabilidade de apresentarem diferenças nulas. Pela análise dos gráficos, também é possível observar que, apesar de ainda haver *outliers*, sua concentração é bem próxima a distribuição geral, demonstrando assim maior estabilidade nas execuções.

Figura 34 – Comparação entre diferenças no tempo de execução entre *containers* Docker e LXC para a linguagem Python

(a) Python 2.7 (b) Python 3.4



Fonte: (AUTORA, 2019).

As diferenças nos tempos de execução entre *container* e *host* com o uso dos *Linux Containers* também apresentaram melhorias nas execuções da linguagem Python, conforme atestam os gráficos da Figura 34. A distribuição dos valores de diferença concentrou-se em valores muito próximos a zero. A mediana das diferenças para a versão 2.7 da linguagem foi de −0,016 segundos e para a versão 3.4 foi de −0,024 segundos. Os valores p resultantes do teste de *Wilcoxon* foram de *p-value* = $4{,}03111 \cdot 10^{-70}$ para Python 2.7 e *p-value* = $9{,}24358 \cdot 10^{-45}$ para Python 3.4.

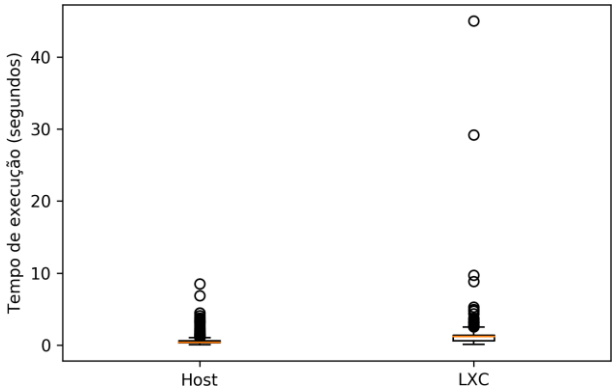
4.9 Análise do desempenho do processo completo de julgamento com *containers* LXC

A partir das análises já apresentadas, é possível perceber que o uso dos *containers* LXC se mostra adequado para o isolamento das execuções das submissões dos usuários do URI Online Judge a serem julgadas. Tanto no âmbito do *overhead* das suas chamadas de execução quanto no contexto das execuções das submissões dentro deste novo ambiente virtual, o gerenciador apresentou resultados satisfatórios e que permitem a aplicação dos *containers* no processo de julgamento do UOJ.

Para devidamente provar esta adequação, são apresentados a seguir os resultados do

teste de tempo de execução do julgamento completo, organizado com *containers* LXC. Os parâmetros e números de execuções também seguiram os padrões utilizados para a avaliação dos *containers* Docker, com o diferencial de que para o LXC só foi considerada a organização de um *container* por linguagem.

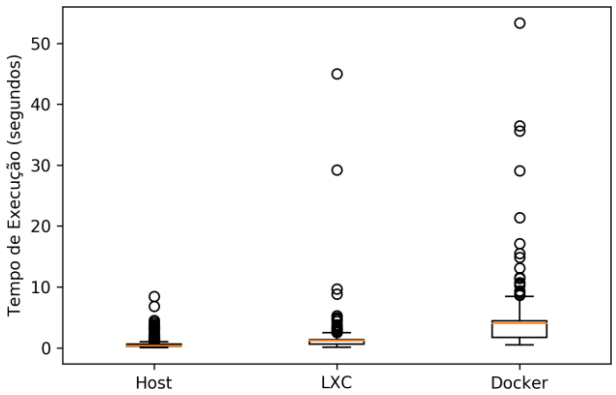
Figura 35 – Comparação entre os tempos totais de julgamento entre *containers* LXC e *Host*



Fonte: (AUTORA, 2019).

A Figura 35 apresenta *boxplots* comparativos entre os tempos totais de julgamento no *host* e no LXC. Percebe-se, com exceção de alguns *outliers*, que as execuções com LXC apresentaram um menor acréscimo no tempo total de julgamento comparado com o Docker, de cerca de 101,64%. Apesar de ser uma porcentagem consideravelmente alta, ela é aceitável dado que o aumento é na escala dos milissegundos. O tempo médio de execução no *host* é igual a 0,525 segundos. O tempo médio de execução do LXC foi de 1,170 segundos, com desvio padrão de 2,195 segundos. Este alto desvio pode ser justificado por duas execuções que apresentaram tempos altos. A mediana da distribuição foi de 1,20137 segundos.

Figura 36 – Comparação entre os tempos totais de julgamento das configurações observadas



Fonte: (AUTORA, 2019).

A Figura 36 apresenta, por fim, um comparativo entre as três configurações

observadas. Através dela, é ainda mais notória a diferença entre os tempos de execução apresentados entre os dois gerenciadores de *containers* e a proximidade das distribuições *Host* e LXC. Conforme anteriormente já relatado, esperava-se a adição de *overheads* no tempo de execução do sistema de julgamento, uma vez que está sendo adicionada uma camada de virtualização extra neste processo. Os resultados demonstrados pelo LXC apresentam tempo adicional adequado para as características necessárias no projeto, dado a quantidade de outros benefícios que o mesmo trás com seu uso, o que permite a sua adição ao sistema de julgamento.

5 CONCLUSÃO

O presente trabalho teve como objetivo aplicar a virtualização por *containers* para melhorar aspectos de segurança, manutenção e do processo de julgamento de soluções do URI Online Judge. Para isso, foram avaliados aspectos relacionados ao comportamento das submissões dentro destes ambientes, seus tempos de execução, tempos de execução das chamadas para os *containers* e tempo do julgamento completo, que foram base para as tomadas de decisões realizadas.

Pode-se provar, através das análises estatísticas apresentadas, que a aplicação de *containers* Docker não foi a mais adequada para esta organização. Apesar desta afirmação parecer ir na contra mão da maioria das aplicações que fazem uso de *containers*, o uso deste gerenciador apresentou altos *overheads* ao sistema de julgamento. Mesmo com a mudança da arquitetura inicialmente projetada para sua aplicação, o Docker apresentou um acréscimo médio de cerca de 500% no tempo total de julgamento de submissões, o que acabou por inviabilizar o uso deste gerenciador neste contexto.

O uso dos Linux *Containers* (LXC) demonstrou o comportamento inicialmente esperado para a adição destes ambientes virtualizados no contexto do julgamento. Além de *overheads* menores no âmbito de suas chamadas de execução, os tempos de execução de códigos dentro dos *containers* LXC apresentaram menores diferenças em relação as execuções na atual configuração de servidores do UOJ. Como referenciado diversas vezes ao longo do trabalho, essa baixa diferença é crucial para a adição de *containers* no julgamento, a fim de não comprometer o sistema de *ranking*.

O desenvolvimento deste trabalho foi bastante dinâmico, já que os resultados de cada etapa ditavam os rumos das etapas posteriores. Por esta razão, necessidades não inicialmente planejadas – como a reestruturação do subsistema responsável pelo controle dos recursos para a execução das submissões e a mudança da arquitetura inicialmente projetada para a adição dos *containers* no processo de julgamento – aumentaram consideravelmente a complexidade do trabalho desenvolvido.

Todavia, provou-se que a adição de *containers* no sistema de correções do UOJ é adequada e apresentará diversos ganhos. Será possível, a partir de agora, retornar a resposta *Memory Limit Exceeded* para o usuário, retorno este que não era possível até então e explicado em maiores detalhes no Capítulo 4; o processo de manutenção e atualização das versões de compiladores, bem como da adição de suporte para novas linguagens, será mais

simples e de fácil execução; além, claro, dos aspectos de isolamento e segurança adicionados.

Como trabalhos futuros, sugere-se o desenvolvimento de um escalonador de *containers*, já que não há submissões constantes para todas as linguagens, a todo momento. Manter todos os *containers* das linguagens em execução a todo tempo pode consumir recursos computacionais excessivos, entre outros comportamentos não previamente projetados. A realização de um escalonamento entre os *containers* ativos irá auxiliar na prevenção de possíveis problemas.

REFERÊNCIAS

AURELIO, M. et al. Virtualização: Conceitos e aplicações em segurança. In: XVII SIMPóSIO BRASILEIRO EM SEGURANçA DA INFORMAçãO E DE SISTEMAS COMPUTACIONAIS. **Minicursos do Simpósio Brasileiro de Segurança da Informação e Sistemas - SBSeg**. [S.l.], 2008. cap. 4, p. 151–200. ISBN 978-85-7669-190-7.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE Cloud Computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

BESERRA, D. et al. Performance analysis of lxc for hpc environments. In: IEEE. **2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems**. [S.l.], 2015. p. 358–363.

BEZ, J. L. **Implementação de Técnicas de Tolerância a Falhas no URI Online Judge**. 61 p.
— Universidade Regional Integrada do Alto Uruguai e das Missões - URI Erechim, 2014. Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação.

BEZ, J. L.; TONIN, N. A.; RODEGHERI, P. R. Uri online judge academic: A tool for algorithms and programming classes. In: IEEE. **2014 9th International Conference on Computer Science & Education**. [S.l.], 2014. p. 149–152.

BIEDERMAN, E. W.; NETWORX, L. Multiple instances of the global linux namespaces. In: CITESEER. **Proceedings of the Linux Symposium**. [S.l.], 2006. v. 1, p. 101–112.

CANONICAL. **Linux Containers**. 2019. Disponível em<https://linuxcontainers.org/pt_br/lxc/ introduction/>. Acesso em 08 Abr. 2019.

CHAVES, J. O. et al. Uma ferramenta baseada em juízes online para apoio às atividades de programação de computadores no moodle. **RENOTE**, v. 11, n. 3, 2013.

COMBE, T.; MARTIN, A.; PIETRO, R. D. To docker or not to docker: A security perspective.
IEEE Cloud Computing, v. 3, n. 5, p. 54–62, 2016.

CONTAINERD. **containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability**. 2019. Disponível em<https://containerd.io/>. Acesso em 27 Set. 2019.

CROSBY, M. **What is containerd ?** 2017. Disponível em<https://www.docker.com/blog/ what-is-containerd-runtime/>. Acesso em 27 Set. 2019.

DAGOSTINI, J. et al. Uri online judge blocks: Construindo soluções em uma plataforma online de programação. In: . [S.l.: s.n.], 2018. p. 168.

DOCKER. **Docker Engine - Enterprise & Community**. 2019. Disponível em<https://www. docker.com/products/docker-engine>. Acesso em 08 Abr. 2019.

Dua, R.; Raja, A. R.; Kakadia, D. Virtualization vs containerization to support paas. In: **2014**

IEEE International Conference on Cloud Engineering. [S.l.: s.n.], 2014. p. 610–614.

GRABER, S. **WGetting started with LXD – the container lightervisor**. 2015. Disponível em
<https://stgraber.org/2015/04/21/lxd-getting-started/>. Acesso em 18 Nov. 2019.

HARTER, T. et al. Slacker: Fast distribution with lazy docker containers. In: **14th *{*USENIX*}***
Conference on File and Storage Technologies (*{*FAST*}* 16). [S.l.: s.n.], 2016. p. 181–195.

ISMAIL, B. I. et al. Evaluation of docker as edge computing platform. In: IEEE. **2015 IEEE Conference on Open Systems (ICOS)**. [S.l.], 2015. p. 130–135.

KERRISK, M. **cgroups - Linux man-pages**. 2019. Disponível em<http://man7.org/linux/man-pages/man7/cgroups.7.html>. Acesso em 27 Set. 2019.

KURNIA, A.; LIM, A.; CHEANG, B. Online judge. **Computers & Education**, Elsevier, v. 36, n. 4, p. 299–315, 2001.

LIMA, M. V. de M. et al. Uma ferramenta online para execução de scripts em sql. In: SBC. **Anais da XIII Escola Regional de Banco de Dados 2017 (ERBD 2017)**. [S.l.], 2017.

MAUERER, W. **Professional Linux kernel architecture**. [S.l.]: John Wiley & Sons, 2010.

MORETTIN, P. A.; BUSSAB, W. O. **Estatística básica**. 9ª. ed. [S.l.]: Editora Saraiva, 2017.

OLIVEIRA, I. C. d. et al. Aprimorando a elasticidade de aplicações de banco de dados utilizando virtualização em nível de sistema operacional. Pontifícia Universidade Católica do Rio Grande do Sul, 2015.

OPENVZL. **Open source container-based virtualization for Linux**. 2018. Disponível em <https://openvz.org/>. Acesso em 08 Abr. 2019.

PöTZL, H. **Linux-VServer**. 2011. Disponível em<http://www.linux-vserver.org/Paper>. Acesso em 08 Abr. 2019.

RESHETOVA, E. et al. Security of os-level virtualization technologies. In: SPRINGER. **Nordic Conference on Secure IT Systems**. [S.l.], 2014. p. 77–93.

RIBEIRO, H. A. C.; SCHIMIGUEL, J. Análise de desempenho de hipervisores no contexto dos sistemas operacionais windows e linux. **Revista Engenho**, v. 12, 2016.

SELIVON, M. **Análise Comparativa Entre Ambiente Virtualizado e Não Virtualizado em Relação ao Tempo de Execução de Submissões no Portal URI Online Judge**. 84 p. — Universidade Regional Integrada do Alto Uruguai e das Missões - URI Erechim, 2016. Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação.

SINGH, H.; YIP, M. **Next-Gen Virtualization for Dummies**. 111 River St., Hoboken: John Wiley & Sons, Inc, 2017.

SPARKS, J. Enabling docker for hpc. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e5018, 2018.

TANENBAUM, A. **Sistemas operacionais modernos**. 3ª. ed. [S.l.]: Prentice-Hall do Brasil, 2010. ISBN 9788576052371.

TONIN, N. A.; BEZ, J. L. Uri online judge: a new classroom tool for interactive learning. **Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)**, p. 1–5, 2012.

TRINDADE, L. V. P.; COSTA, L. H. M. Análise do desempenho da virtualização leve para ambientes com edge computing baseada em nfv. In: SBC. **Anais do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [S.l.], 2018.

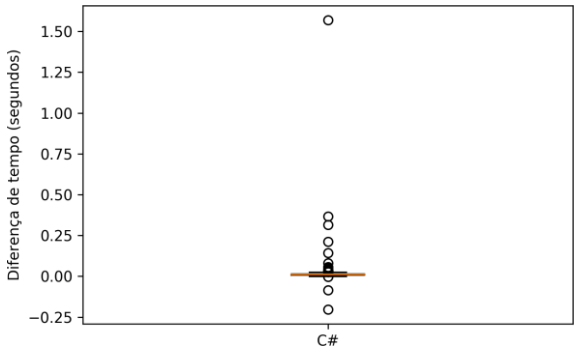
WILCOXON, F. Individual comparisons by ranking methods. **Biometrics Bulletin**, [International Biometric Society, Wiley], v. 1, n. 6, p. 80–83, 1945. ISSN 00994987. Disponível em:<http://www.jstor.org/stable/3001968>.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: IEEE. **2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing**. [S.l.], 2013. p. 233–240.

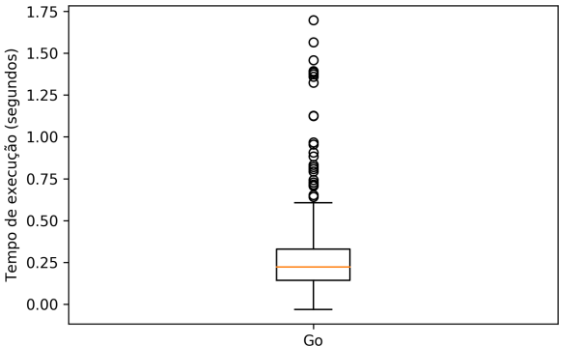
Anexos

ANEXO A – Diferenças de *runtime* entre *host* e *containers* Docker oficiais

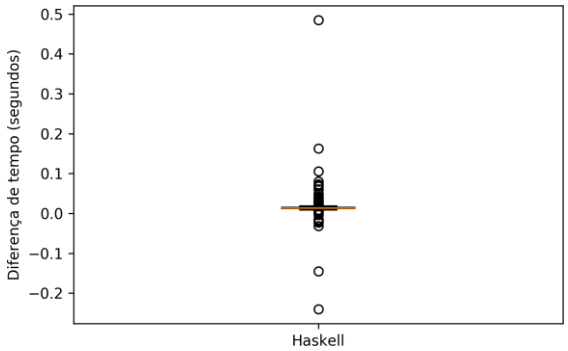
(a) C#



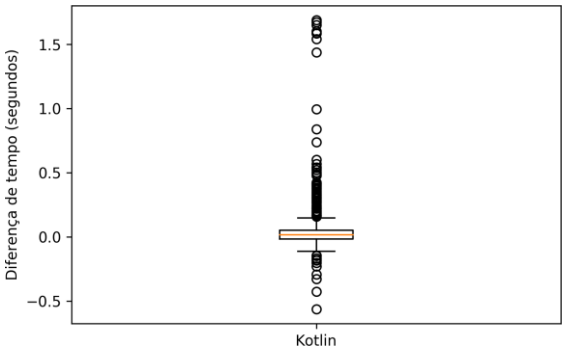
(b) Go



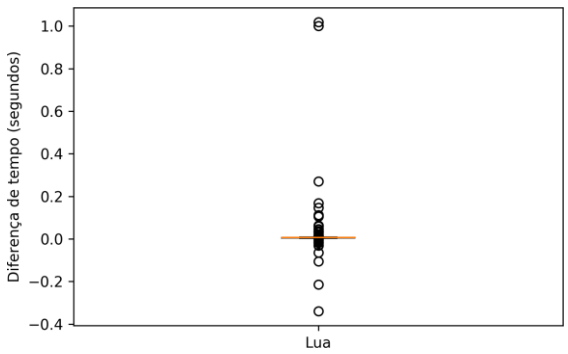
(c) Haskell



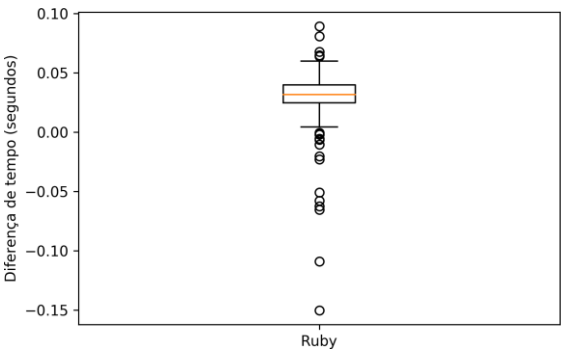
(d) Kotlin



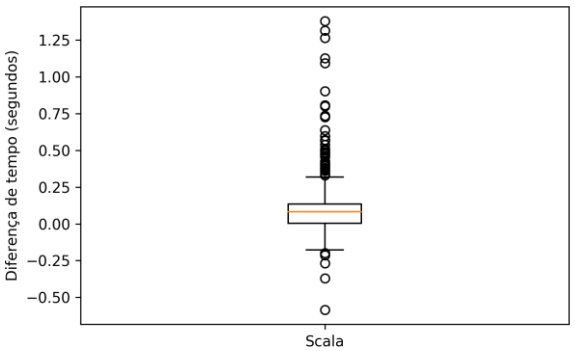
(e) Lua



(f) Ruby

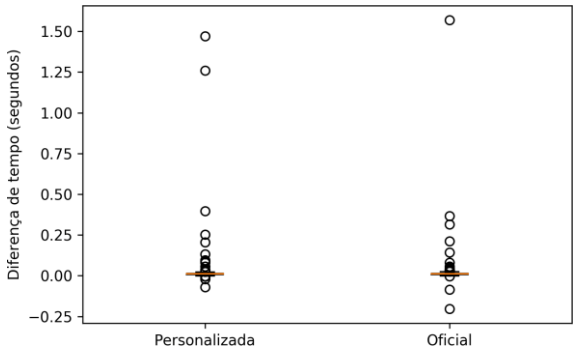


(g) Scala

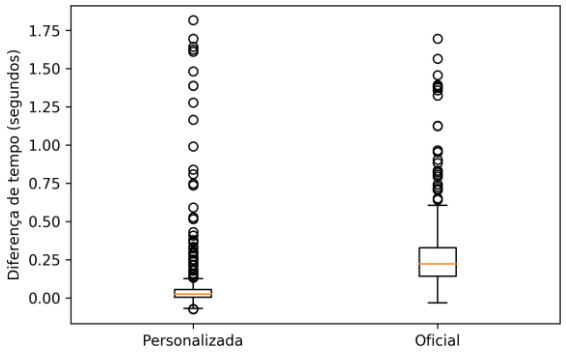


ANEXO B – Diferenças de *runtime* entre *host* e *containers* Docker customizados

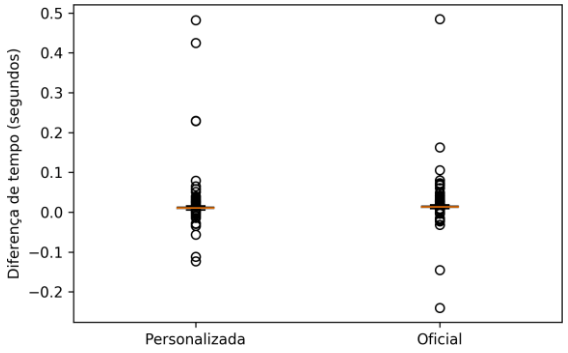
(a) C#



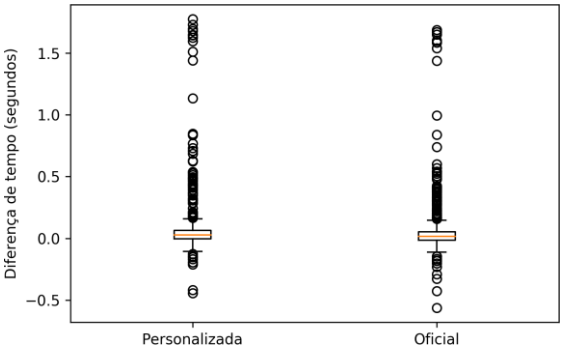
(b) Go



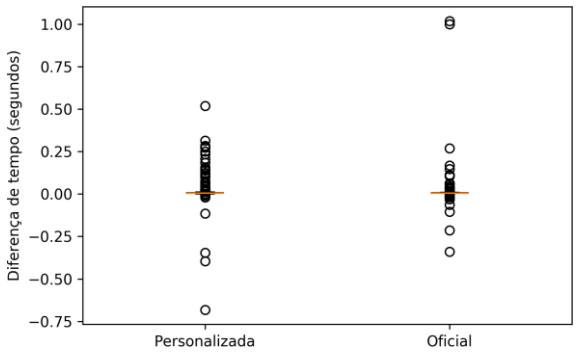
(c) Haskell



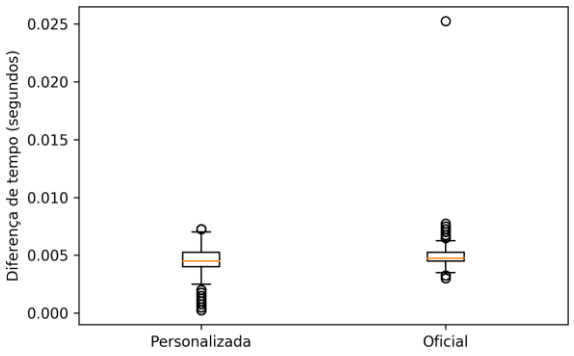
(d) Kotlin



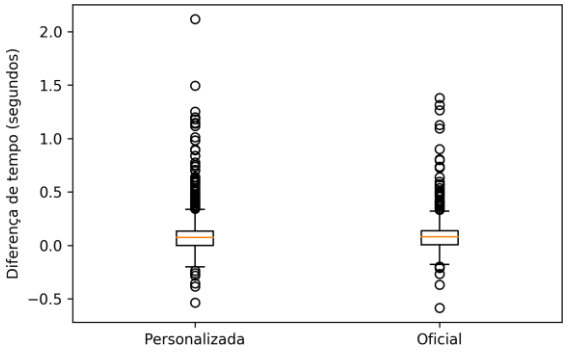
(e) Lua



(f) Pascal

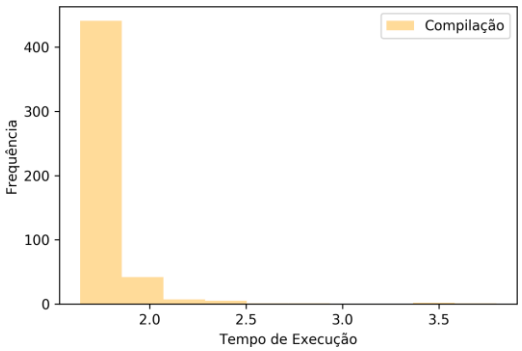


(g) Scala

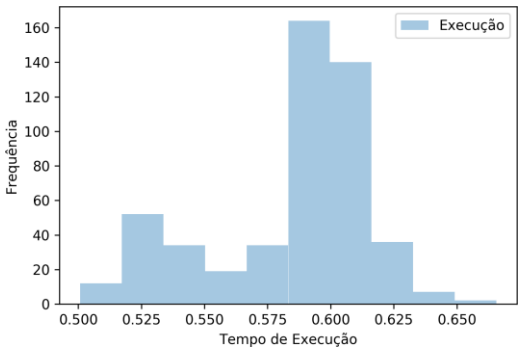


ANEXO C – Histogramas do tempo médio de execução dos *containers* Docker

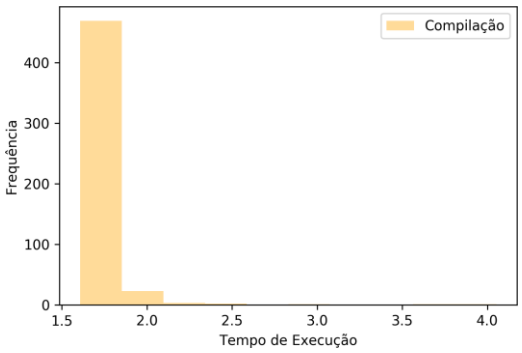
(a) *Overheads* na compilação C#



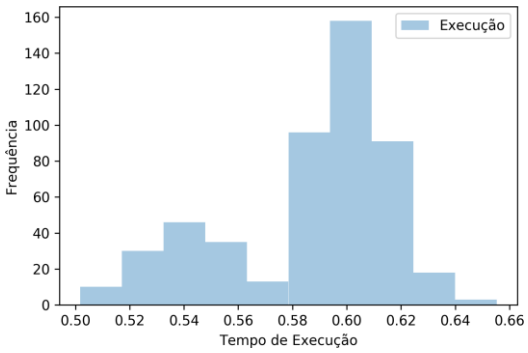
(b) *Overheads* na execução C#



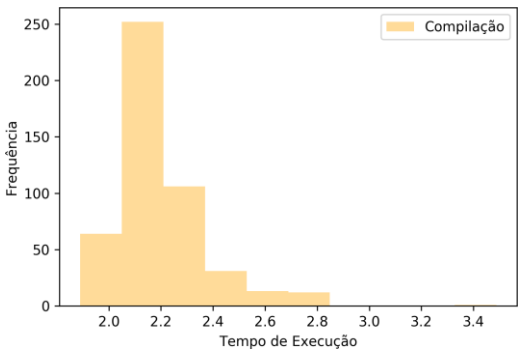
(a) *Overheads* na compilação Go



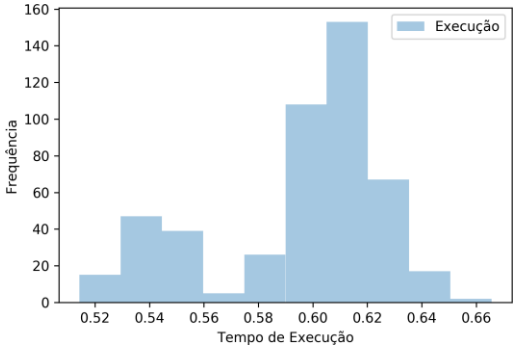
(b) *Overheads* na execução Go



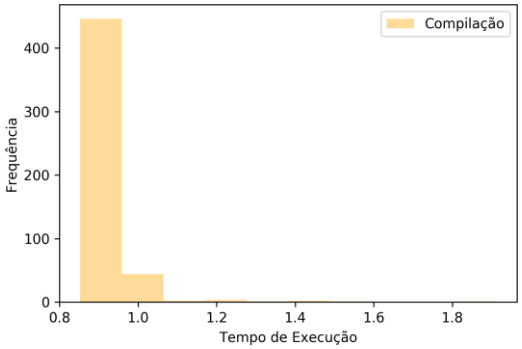
(a) *Overheads* na compilação Haskell



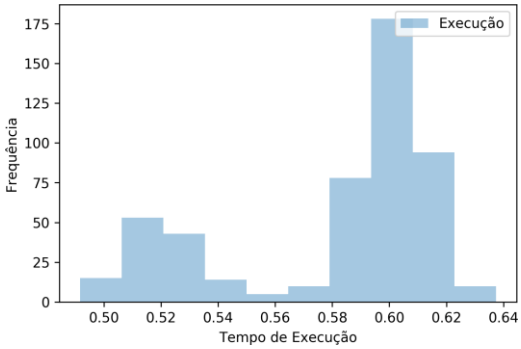
(b) *Overheads* na execução Haskell



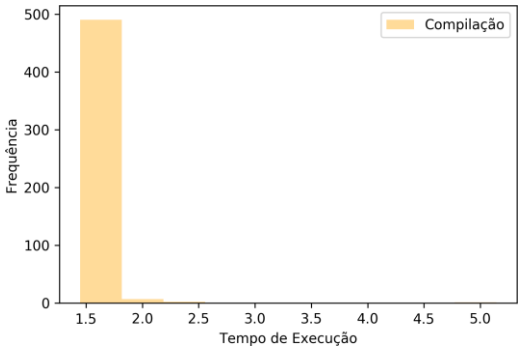
(a) *Overheads* na compilação Lua



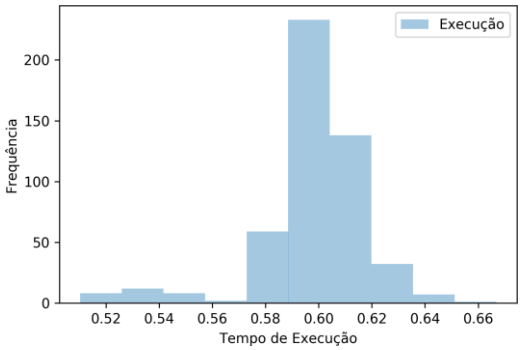
(b) *Overheads* na execução Lua



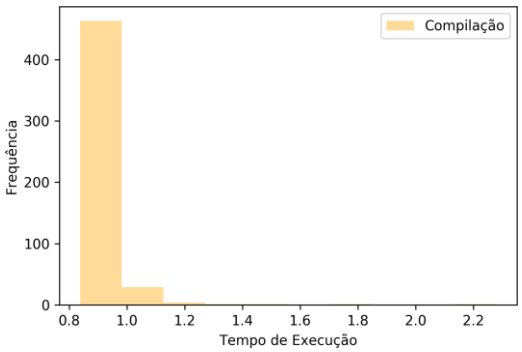
(a) *Overheads* na compilação Pascal



(b) *Overheads* na execução Pascal



(a) *Overheads* na compilação Ruby



(b) *Overheads* na execução Ruby