Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Avaliação de Desempenho de Contêineres Docker para Aplicações do Supremo Tribunal Federal

Flávio Henrique Rocha e Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. André Costa Drummond

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Pedro Antonio Dourado Rezende

Banca examinadora composta por:

Prof. Dr. André Costa Drummond (Orientador) — CIC/UnB
Profª Drª Aletéia Patrícia Favacho de Araújo — CIC/UnB
Prof. Dr. Eduardo Adilio Pelinson Alchieri — CIC/UnB

CIP — Catalogação Internacional na Publicação

Silva, Flávio Henrique Rocha e.

Avaliação de Desempenho de Contêineres Docker para Aplicações do Supremo Tribunal Federal / Flávio Henrique Rocha e Silva. Brasília : UnB, 2017.
79 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. Computação em nuvem, 2. Contêineres, 3. Docker, 4. Avaliação de Desempenho, 5. Trabalho de conclusão de curso

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Avaliação de Desempenho de Contêineres Docker para Aplicações do Supremo Tribunal Federal

Flávio Henrique Rocha e Silva

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. André Costa Drummond (Orientador)
CIC/UnB

Profª Drª Aletéia Patrícia Favacho de Araújo
CIC/UnB

Prof. Dr. Eduardo Adilio Pelinson Alchieri
CIC/UnB

Prof. Dr. Pedro Antonio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 20 de março de 2017

Dedicatória

Dedico esse trabalho aos meus filhos Renato, Carla e Miguel. Para que saibam que, em qualquer momento da vida, podemos lutar pelos nossos sonhos e conseguir realizá-los.

Agradecimentos

Agradeço, em primeiro lugar, à minha mãe, Lúcia, que sempre foi para mim um exemplo de luta, garra e determinação. Obrigado mãezinha por me mostrar como é importante o estudo e o trabalho duro para alcançarmos nossos objetivos na vida.

Muito obrigado também ao meu grande amigo Bruno, que me sugeriu e incentivou a cursar Computação na UnB. Valeu figura! Você me convenceu que eu poderia mais quando eu não acreditava ser possível.

Agradeço imensamente ao meu grande amor Ana Paula, que chegou no meio dessa minha caminhada, mas que já me apoiou como nunca, sofreu comigo nas dificuldades e compartilhou as alegrias dessa jornada. Obrigado anjo meu! Essa vitória também é sua.

Obrigado aos colegas do Supremo Tribunal Federal, em especial ao amigo Aurélio, que várias vezes me ofereceu suporte intelectual e acadêmico no desenvolvimento desse trabalho e ao meu chefe à época, Venício, que confiou no meu trabalho e na minha competência para desenvolver o projeto piloto de implantação do Docker no tribunal.

Meus agradecimentos ao meu orientador, professor André Drummond, pela sua paciência e incentivo na elaboração desse trabalho. Agradeço também aos professores Aletéia Araújo, Eduardo Alchieri e Wilson Veneziano pela sua disponibilidade e preciosos conselhos para a concretização da minha graduação.

Por fim, agradeço infinitamente ao meu Deus, sem o qual nada disso seria possível. Pela Sua Graça tudo foi realizado. Obrigado Senhor por sempre cuidar de mim!

Resumo

A Virtualização, na última década, contribuiu de forma decisiva na otimização do uso de recursos computacionais seja em Centros de Dados privados, quanto em ambientes de provedores de serviço de Nuvem. Tradicionalmente dominados pela tecnologia de máquinas virtuais com hipervisores, os ambientes que fazem uso de Contêineres para provisionamento de aplicações têm se popularizado. Considerados como uma alternativa leve ao ambiente de hipervisores, os contêineres trazem diversos benefícios, tais como menor sobrecarga do hardware, redução de falhas em função da configuração, atualizações e erros de softwares e maior velocidade na inicialização. A escalada na adoção dessa tecnologia se deve às facilidades oferecidas pelo Docker, que simplificou a criação e execução de contêineres. A Secretaria de Tecnologia da Informação do Supremo Tribunal Federal viu na utilização da tecnologia de Contêineres o potencial de melhorar o processo de desenvolvimento e o provisionamento de aplicações no tribunal. O presente trabalho se propõe a realizar uma avaliação de desempenho dos Contêineres Docker em relação aos ambientes de instalação nativa (*Baremetal*) e Virtualização com Hipervisores para validar a viabilidade de adoção dessa tecnologia para o provisionamento das aplicações do STF sem degradação dos serviços oferecidos aos usuários da Corte.

Palavras-chave: Computação em nuvem, Contêineres, Docker, Avaliação de Desempenho, Trabalho de conclusão de curso

Abstract

Virtualization over the last decade has contributed decisively to optimizing the use of computing resources in both private Data Centers and cloud service providers' environments. Traditionally dominated by virtual machine technology with hypervisors, environments that make use of Containers for application provisioning have become popular. Considered as a lightweight alternative to the hypervisor environment, the containers bring several benefits, such as lower hardware overhead, reduced configuration failures, software updates and errors, and faster boot speeds. The escalation in the adoption of this technology is due to the facilities offered by Docker, which simplified the creation and execution of containers. The Office of Information Technology of the Federal Supreme Court has seen in the use of container technology the potential to improve the development process and the provision of applications in court. The present work proposes to perform a performance evaluation of the Docker Containers in relation to the native installation environments (Baremetal) and Virtualization with Hypervisors to validate the feasibility of adopting this technology for the provisioning of STF applications without degradation of the services offered to the users of the Court.

Keywords: Cloud computing, Containers, Docker, Performance Evaluation, Thesis

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As tecnologias de virtualização têm desempenhado um papel de destaque no provisionamento de ambientes para execução de aplicações. Com a intenção de reduzir custos e melhorar a eficiência dos centros de dados corporativos, as áreas responsáveis pela infraestrutura de TI investiram pesadamente, na última década, na consolidação de servidores. A consolidação de servidores permite que vários servidores virtuais sejam executados em um mesmo *host*, o que leva a uma melhor utilização da capacidade do hardware reduzindo o espaço e o consumo de energia nos *datacenters*.

Muitas aplicações empresariais que tradicionalmente funcionavam em servidores dedicados são consolidadas em um agregado compartilhado de servidores. Todavia, embora a consolidação de servidores ofereça grande potencial para aumentar a utilização de recursos e melhorar o desempenho da aplicação, também pode trazer nova complexidade no gerenciamento dos servidores consolidados. Isso impõe novos desafios, incluindo a escolha da tecnologia de virtualização e a configuração de consolidação adequadas para um conjunto específico de aplicativos [47]. A solução de virtualização mais utilizada é a tecnologia baseada em hipervisor (*hypervisor*), que tem como principais representantes o VMware [30], Microsoft Hyper-V [22] e Xen [32].

Nos últimos anos tem-se visto uma crescente adoção, especialmente, por provedores de computação em nuvem, da virtualização baseada em contêineres. A virtualização baseada em contêineres fornece um nível diferente de abstração em termos de virtualização e isolamento quando comparado com hipervisores. Em particular, os contêineres podem ser considerados como uma alternativa leve à virtualização baseada em hipervisor. Os hipervisores abstraem o hardware, o que resulta em sobrecarga em termos de virtualização de hardware e *drivers* de dispositivos virtuais.

Um sistema operacional completo (por exemplo, Linux) normalmente é executado no topo deste hardware virtualizado em cada instância de máquina virtual. Em contraste, os contêineres implementam isolamento de processos no nível do sistema operacional,

evitando assim tais sobrecargas. Esses contêineres são executados em cima do mesmo *kernel* do sistema operacional compartilhado da máquina *host* subjacente, e um ou mais processos podem ser executados dentro de cada contêiner [44].

A popularização dessa tecnologia se deve ao Docker [31], que devido ao seu conjunto de recursos e facilidade de uso, rapidamente se tornou para a indústria e desenvolvedores a ferramenta de gerenciamento e o formato de imagem padrão para contêineres. O aumento do investimento nesse tipo de tecnologia se deve a fatores demonstrados pelo seu uso tais como [46]:

- a eficiência no uso de recursos computacionais (processamento, memória e armazenamento) - pois os contêineres compartilham os referidos recursos com o sistema operacional (SO) hospedeiro, permitindo uma maior velocidade na inicialização e na interrupção das aplicações, com pouca ou nenhuma sobrecarga em comparação com aplicativos executados nativamente no SO hospedeiro;
- portabilidade - devido ao encapsulamento da aplicação e suas dependências em um contêiner, bem como a sua distribuição por meio de imagens, a eliminação de diversos erros promovidos por mudanças sutis no ambiente de execução é potencializada;
- produtividade para equipes de desenvolvedores na criação de aplicações - sem preocupações referentes a configuração de ambientes na realização de testes, no qual se pode emular um sistema distribuído em produção com facilidade; e
- adoção do uso da computação em nuvem, na qual provedores de serviços têm adotado o uso massivo da tecnologia de contêineres nas suas soluções de Plataforma como Serviço (PaaS).

Além disso, em muitos casos, as imagens de contêiner Docker requerem menos espaço em disco e operações de Entrada e Saída (E/S) do que imagens de disco VM equivalentes. Isso leva a uma implantação mais rápida na nuvem, uma vez que as imagens geralmente precisam ser copiadas na rede para o disco local antes que a VM ou o contêiner possa ser iniciado.

Por outro lado, o Supremo Tribunal Federal (STF) é o órgão de cúpula do Poder Judiciário do Brasil, e a ele compete a guarda da Constituição, conforme definido no art. 102 da Constituição Federal. O STF acumula competências típicas de Suprema Corte (tribunal de última instância) e Tribunal Constitucional (que julga questões de constitucionalidade independentemente de litígios concretos) [26]. Na área penal, destaca-se a competência para julgar, nas infrações penais comuns, o Presidente da República, o Vice-Presidente, os membros do Congresso Nacional, seus próprios Ministros e o Procurador-Geral da República, entre outros.

Na estrutura organizacional do STF, a Secretaria de Tecnologia da Informação (STI) é a área do Supremo Tribunal Federal responsável pela manutenção da infraestrutura computacional. Dentre as suas atribuições estão:

- Projetar e manter as redes de computadores física e sem fio;
- Gerenciar os ativos de TI como servidores, *switches*, *storages*, etc;
- Gerenciar os ambientes seguros (sala-cofre e sala de contingência);
- Sustentar os servidores que hospedam as aplicações corporativas e de terceiros para utilização pela Corte;
- Desenvolver e sustentar aplicações para realização das atividades laborais;
- Prover e manter o ambiente de microinformática (computadores pessoais e impressoras);
- Realizar aquisições e contratações de serviços de TI;
- Realizar a Govenança de TI.

Atualmente, no modelo de desenvolvimento e hospedagem para aplicações desenvolvidas internamente na STI, cada aplicação é construída para cada ambiente (teste, homologação e produção), assim, enfrentam-se diversos erros nas aplicações em decorrência dessa construção diferenciada. Além disso, ainda pode existir divergências infraestruturais entre ambientes, prejudicando os testes e implantação de novos sistemas ou melhorias nas aplicações já existentes, o que afeta diretamente a qualidade dos serviços providos pela STI. Outro problema relacionado a esse modelo de implantação de sistemas é a dificuldade de provisionar ambientes aos times de desenvolvimento, uma vez que os desenvolvedores devem parametrizar a complexidade dos servidores de aplicações em seus computadores locais, o que acarreta maior tempo para solucionar eventuais erros e implementar novas funcionalidades nos sistemas. Em suma, os problemas técnicos enfrentados pelas equipes de infraestrutura e de desenvolvimento, para manter os serviços de TI desenvolvidos pelo STF, geram indisponibilidades, erros e mal funcionamento.

Diante desses problemas foi observado que o provisionamento de ambientes deveria ser mais simples, rápido e estável. A STI já faz uso de ambientes virtualizados com hipervisores, porém a implantação de uma infraestrutura de contêineres surgiu como uma possibilidade para provisionamento dos serviços de TI por oferecer, em uma primeira análise, a flexibilidade e o aproveitamento de recursos computacionais desejados.

Diante do exposto, este trabalho propõe a realização de uma avaliação de desempenho do uso de contêineres Docker como ambiente de execução de uma aplicação desenvolvida pela STI/STF. O desempenho do Docker será comparado ao uso de máquinas virtuais

com hipervisor e com uma instalação nativa (*baremetal*), essa comparação tem a intenção de averiguar se não haverá alguma degradação de desempenho no uso desse aplicativo utilizando a tecnologia de contêineres.

1.1 Motivação

A utilização da tecnologia de contêineres tem o potencial de melhorar o processo de desenvolvimento e de provisionamento de aplicações do Supremo Tribunal Federal. Dentre essas possíveis melhorias, destaca-se a simplificação da construção de ambientes para as equipes de desenvolvimento e a redução de falhas decorrentes da diferença de configuração entre ambientes de qualidade, homologação e produção. Com isso é possível diminuir o tempo de implantação dos sistemas e correção de erros, bem como aumentar a disponibilidade das aplicações.

Para a adoção de uma plataforma de contêineres para o provisionamento de aplicações desenvolvidas no STF, sem que haja uma degradação dos serviços oferecidos aos usuários, é necessário avaliar o seu desempenho em comparação ao ambiente de virtualização com hipervisor já em uso na infraestrutura do tribunal.

1.2 Objetivos

O objetivo geral deste trabalho é realizar uma avaliação de desempenho do uso de contêineres Docker para o provisionamento de aplicações desenvolvidas pela Secretaria de Tecnologia da Informação do Supremo Tribunal Federal.

Para atingir o objetivo geral, foram definidos objetivos específicos, a saber:

- Desenvolver imagens Docker para uma aplicação desenvolvida pela Secretaria de Tecnologia da Informação do Supremo Tribunal Federal;
- Realizar testes de desempenho da aplicação do STF entre os ambientes de contêineres, virtualização com hipervisor e *baremetal* (nativo).

1.3 Estrutura do Trabalho

Este trabalho está dividido em mais cinco capítulos, além deste introdutório. No Capítulo 2 são apresentados o conceito da tecnologia de contêineres e o seu funcionamento. Também são apresentadas as diferenças entre os contêineres e as máquinas virtuais, bem como os conceitos de computação em nuvem e o uso de contêineres. Esse capítulo é finalizado com um histórico sobre a evolução da tecnologia de contêineres.

O Capítulo 3 apresenta o ecossistema Docker, sua base tecnológica e sua arquitetura. O Capítulo 4 traz uma revisão bibliográfica dos principais artigos que norteiam o desenvolvimento deste trabalho. O Capítulo 5 detalha a avaliação de desempenho realizada sobre o uso de contêineres para o provisionamento de aplicações no Supremo Tribunal Federal.

Por fim, no Capítulo 6 são apresentadas as conclusões acerca do trabalho desenvolvido e as sugestões de trabalhos futuros.

Capítulo 2

Contêineres

2.1 Evolução da Tecnologia de Contêineres

O conceito de contêineres foi iniciado em 1979 com a introdução da funcionalidade chamada *chroot* no sistema operacional UNIX. Essa tecnologia consistia em uma chamada de sistema (*system call*) do SO para alterar o diretório raiz de um processo e seus processos-filhos para um novo local no sistema de arquivos que é apenas visível para aquela hierarquia de processos. Mais tarde, em 1982, este foi adicionado ao sistema operacional BSD (*Berkeley Software Distribution*) baseado no UNIX [12].

A partir de 1998, o FreeBSD passou a fornecer o utilitário chamado *Jail* introduzido por Derrick T. Woolworth na R & D Associates. O FreeBSD Jails é semelhante ao chroot, mas incluiu os recursos de isolamento de processos adicionais para o sistema de arquivos, usuários, redes e outros aspectos do SO. Como resultado, pode fornecer meios de atribuir um endereço IP para cada *jail*, bem como instalações e configurações personalizadas de software.

O Linux VServer, lançado em 2001, é um outro mecanismo de *jail* que pode ser usado para particionar os recursos de forma segura em um ambiente computacional (sistema de arquivos, tempo de CPU, endereços de rede e memória). Isso é realizado por meio de níveis de isolamento do *kernel*. Cada partição é chamada de contexto de segurança (*security contexts*) e o sistema virtualizado dentro dele é chamado de servidor privado virtual (*virtual private server*) [19].

O *Solaris Zones* foi introduzido para sistemas x86 e SPARC na versão 10 do sistema operacional Solaris 10 em 2005. O *Solaris Zones* oferecia uma tecnologia de contêinerização comparativamente completa. As zonas funcionam como servidores virtuais completamente isolados dentro de uma única instância do sistema operacional. Existem dois tipos de *Solaris Zones*: zonas globais (*global zones*) e não-globais (*non-global zones*). A zona global é o ambiente de SO tradicional e é a área onde o SO Solaris está instalado.

Todas as operações do sistema, como instalações, inicializações e desligamentos são feitas na zona global. As zonas não-globais, comumente chamadas somente de zonas, tem seus recurso e limites definidos pela zona global a que pertencem [17].

Ainda em 2005, a empresa Parallels abriu o código do seu projeto de contêineres *Virtuozzo* para a comunidade Linux. O OpenVZ como passou a ser chamado, faz uso do *kernel* Linux para fornecer virtualização, isolamento, gerenciamento de recursos e *checkpointing*. Cada contêiner OpenVZ possui um sistema de arquivos isolado, usuários e grupos de usuários, uma árvore de processos, rede, dispositivos e comunicação entre processos (IPC-*Interprocess Comunication*). Sua adoção em massa possivelmente foi prejudicada pela necessidade de aplicação de correções (*patches*) no *kernel* para seu funcionamento [12].

Engenheiros do Google (principalmente Paul Menage e Rohit Seth) iniciaram no ano de 2006 o desenvolvimento de um recurso sob o nome de *Process Containers*. No final de 2007, a nomenclatura mudou para *Control Groups*, ou simplesmente *cgroups*, para evitar a confusão causada pelos múltiplos significados do termo "contêiner" no contexto do kernel do Linux. Essa funcionalidade foi mesclada na linha principal do *kernel* do Linux na versão 2.6.24 que foi lançado em janeiro de 2008. O *cgroups* é um recurso que limita, contabiliza e isola o uso de recursos (CPU, memória, E / S de disco, rede, etc.) de uma coleção de processos. Devido a sua importância esse recurso será melhor detalhado na Seção 3.1.1.

LXC, abreviação de *LinuX Containers*, é a primeira e mais completa implementação de um gerenciador de contêineres Linux surgida em 2008. Foi desenvolvido tendo como base as funcionalidades *cgroups* e *namespaces* (utilizado para controle de recursos, explicada em detalhes na seção 3.1.2) do *kernel* principal Linux desde a versão 2.6.27. É um projeto de código aberto (*open source*) liderado por Stephane Graber e Serge Hallyn e patrocinado pela Canonical Ltd que é responsável pela distribuição Linux Ubuntu. O LXC fornece ferramentas para gerenciar contêineres, suporte avançado de rede e armazenamento, e modelos mínimos (*templates*) para contêineres. Provê também a biblioteca liblxc - que realiza a interface com as funcionalidades do *kernel* - e uma API (*Application Programming Interface*) em Python3, Python2, Lua, Go, Ruby e Haskell.

No ano de 2011 foi implementado pela CloudFoundry o software Warden tendo como base o código fonte do LXC em sua fase inicial, que mais tarde o substituiu por sua própria implementação. Ao contrário do LXC, o Warden não está estritamente ligado ao Linux. Em vez disso, ele pode trabalhar em qualquer sistema operacional que pode fornecer maneiras de isolar ambientes. Ele é executado como um serviço (*daemon*) no sistema operacional e fornece uma API para gerenciar os contêineres [12].

Docker é o sistema de gerenciamento de contêineres mais popular na atualidade, e amplamente utilizado a partir de janeiro de 2016. Ele foi desenvolvido como um projeto

interno em uma empresa de plataforma como serviço (PaaS) para infraestrutura de computação em nuvem chamada dotCloud, que mais tarde foi renomeada para Docker,Inc. Semelhante ao Warden, o Docker também usou o LXC nos estágios iniciais e depois substituiu a liblxc por sua própria biblioteca chamada *libcontainer*. Ao contrário de qualquer outra plataforma de contêineres, a Docker introduziu um ecossistema inteiro para gerenciamento de contêineres [12]. Por ser a infraestrutura escolhida para o desenvolvimento deste trabalho, a plataforma Docker será detalhada no Capítulo 3.

O *Rocket*, também chamado de rkt, é uma iniciativa muito semelhante ao Docker iniciado pela CoreOS em 2014. Segundo a CoreOS, desenvolvedora do aplicativo, o objetivo do *Rocket* é fornecer requisitos de segurança e produção mais rigorosos do que os presentes no Docker. A interface principal do rkt inclui um único aplicativo executável, em vez de um *daemon* executado em segundo plano. O *Rocket* implementa um formato de contêiner aberto - o *App Container* (appc) - e também pode executar outras imagens de contêiner, como as criadas com Docker [6].

Em 2015 a Microsoft também tomou a iniciativa de adicionar suporte a contêineres ao sistema operacional Windows Server. Os *Windows Containers* foram incluídos na versão Microsoft Windows Server 2016, usando as APIs e o cliente do Docker. Os contêineres do Windows incluem dois tipos diferentes: *Windows Server Containers* e contêineres *Hyper-V*. Os contêineres *Windows Server* fornecem isolamento de aplicativos através da tecnologia de isolamento de processos e *namespaces*. Esse tipo de contêiner compartilha o *kernel* do SO com o hospedeiro e todos os contêineres executados na máquina. Já com o tipo Hyper-V, cada contêiner é executado em uma máquina virtual altamente otimizada. Nessa configuração, o *kernel* do *host* não é compartilhado com os contêineres desse tipo [5].

Por fim, com o propósito de criar padrões abertos e comuns para indústria em relação aos formatos e execução (*runtime*) de contêineres, foi criada a *Open Container Initiative* (OCI). Lançada em 22 de junho de 2015 pela Docker e CoreOS, logo obteve a adesão de várias empresas, tais como o Google, a Amazon, a RedHat, a Oracle, dentre outras. O OCI atualmente contém duas especificações: a de execução (*runtime-spec*) e a especificação de imagem (*image-spec*). A especificação de execução descreve como executar um "pacote de sistema de arquivos" que é descompactado no disco. Em alto nível, uma implementação de OCI seria realizar o *download* de uma imagem OCI, e em seguida, desempacotar essa imagem no sistema de arquivos de um ambiente de execução próprio - o *OCI Runtime* - e em seguida executá-la. A especificação de imagem define como criar uma imagem OCI. Esse formato de imagem contém informações suficientes para iniciar o aplicativo na plataforma de destino (como por exemplo, comandos, argumentos e variáveis de ambiente) [25].

2.2 Conceito

A tecnologia de contêineres visa prover uma unidade de computação auto-suficiente para execução de uma aplicação em um ambiente computacional [48]. Um contêiner encapsula o aplicativo e suas dependências, permitindo que ele seja executado de forma isolada das outras aplicações em um mesmo computador [46]. Merkel [43] explica que os contêineres são executados a partir do sistema operacional (SO) hospedeiro em espaços próprios e isolados providos pelo núcleo (*kernel*).

Pahl [48] esclarece que os contêineres são baseados em camadas feitas de imagens individuais construídas sobre uma imagem de base que pode ser estendida. As imagens completas formam contêineres de aplicativos portáteis.

Dessa forma, o ecossistema de contêineres consiste em um mecanismo para construção e execução de imagens, bem como um repositório ou registro para transferir essas imagens de/e para os ambientes de execução instalados em um computador hospedeiro. Os repositórios desempenham um papel central no fornecimento de acesso a, possivelmente, dezenas de milhares de imagens reutilizáveis públicas e privadas de contêineres, como por exemplo, para componentes de plataforma como o MongoDB [16] ou Node.js [24]. A API *(Application Programming Interface)* de contêineres permite operações para criação, definição, composição e distribuição de imagens e também o controle do ciclo de vida dos contêineres executados a partir de imagens por meio de operações de inicialização, parada, parametrização e destruição.

A granularidade dos contêineres, isto é, o número de aplicações em seu interior, é variado. A abordagem de um contêiner por aplicativo é indicada como uma boa prática, pois permite compor facilmente novas pilhas de serviço - como por exemplo, uma aplicação web Java com um proxy HTTP Apache, um servidor de aplicações JBoss e um banco de dados MySQL - ou reutilizar componentes comuns (por exemplo, ferramentas de monitoramento ou um serviço de armazenamento chave-valor como o Redis). Os aplicativos podem ser construídos, reconstruídos e gerenciados facilmente. O armazenamento e o gerenciamento de rede são dois problemas específicos que a tecnologia de contêineres, como mecanismo de encapsulamento de aplicativos para contextos interoperáveis e distribuídos, deve facilitar.

A Figura 2.1 demonstra a estrutura de contêineres formados pela composição de imagens pré-construídas e sua interface com o núcleo do sistema operacional da máquina hospedeira.

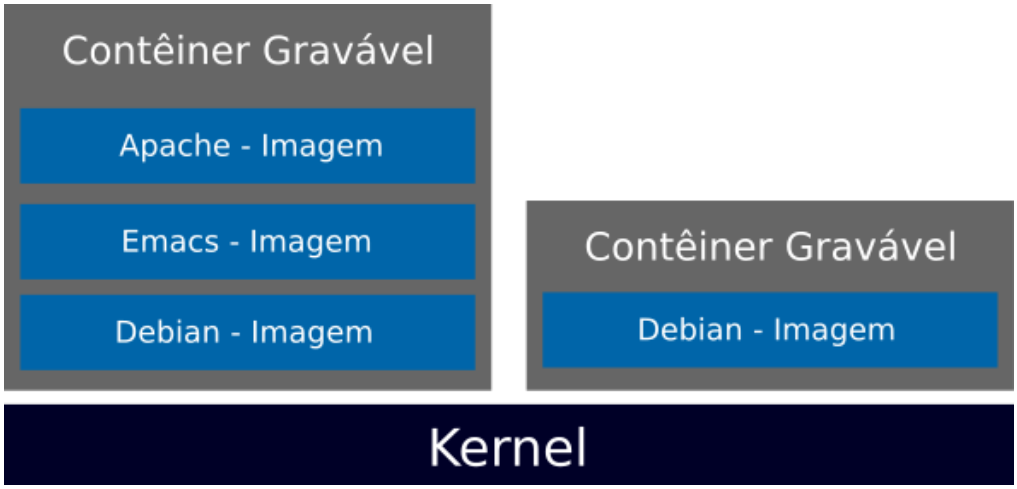

Figura 2.1: Representação de Contêineres Formados por Imagens de Softwares Pré-Construídas, baseado em [31].

.

2.3 Diferenças entre Contêineres e Máquinas Virtuais

Em essência o papel do sistema operacional é oferecer a interface dos programas (softwares) com os recursos físicos (hardware) da máquina que os executam. Com o avanço tecnológico dos componentes de hardware, os recursos físicos se tornaram cada vez mais poderosos, o que levou ao desenvolvimento das técnicas de virtualização para um maior aproveitamento do excesso de poder de processamento, memória, armazenamento ou largura de banda da rede.

Segundo Morais [45], virtualização é a tecnologia que permite a criação de diferentes ambientes computacionais, chamados de virtuais por simular a interface que é esperada por um sistema operacional. A utilização mais usual é a virtualização de hardware que permite a coexistência de diferentes pilhas de softwares sob o mesmo hardware, sendo que estas pilhas contém em seus interiores instâncias de máquinas virtuais (*virtual machines* ou VMs). Um dos principais componentes dessa arquitetura é o software de virtualização chamado de hipervisor (*Hypervisor*). A distribuição dos recursos da máquina física (armazenamento, processamento e memória) entre as VMs é gerenciada pelo hipervisor que fornece também o completo isolamento entre elas.

Na definição de Mouat [46], embora, à primeira vista, os contêineres aparentem ser um tipo de máquina virtual (*virtual machine* ou VM) leve - pois mantém uma instância de um sistema operacional isolada para execução de aplicativos - os objetivos básicos das VMs e dos contêineres são diferentes. A finalidade de uma VM é emular totalmente um ambiente computacional externo, enquanto que a de um contêiner é tornar os aplicativos

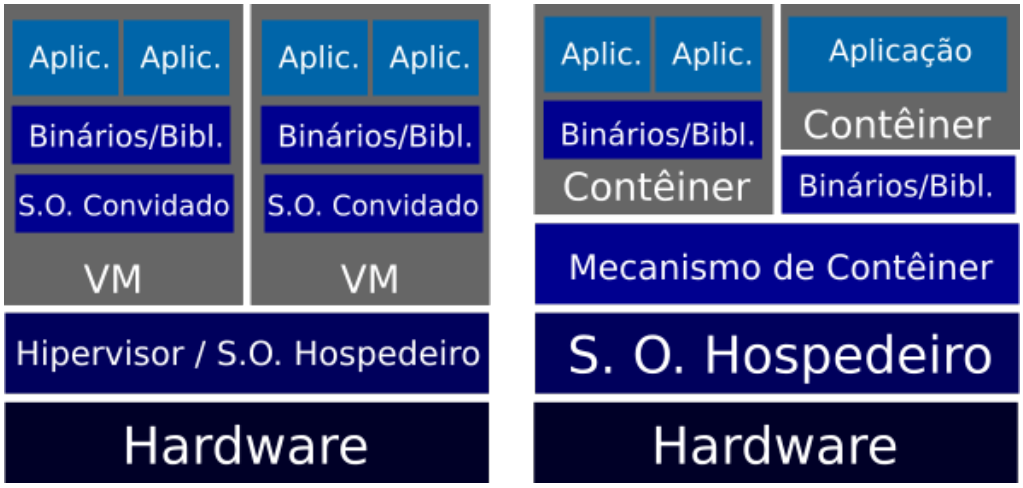

Figura 2.2: Comparação entre implantações de aplicações baseada em máquinas virtuais com hipervisor e com a utilização de contêineres (baseado em [48])

.

portáveis e independentes.

Bernstein [33] aponta que ambos, máquinas virtuais e contêineres, apresentam uma interface de sistema operacional para execução dos programas. No caso das máquinas virtuais é uma implementação completa de qualquer sistema operacional enquanto que os contêineres tem uma visão de "fatias" de um sistema operacional já em execução. Com os contêineres, as aplicações compartilham um sistema operacional (e, quando apropriado, binários e bibliotecas) e, como resultado, essas implantações serão significativamente menores em tamanho do que em uma VM utilizando um hipervisor.

As instâncias da VM usam arquivos grandes e isolados em seu computador hospedeiro (*host*) para armazenar todo o sistema de arquivos e executar normalmente um único e grande processo no *host*. É necessária uma instalação completa do SO para cada VM em funcionamento, que se traduz em maior utilização de memória e requisitos de armazenamento em disco tornando assim mais lenta a sua inicialização [48].

Na Figura 2.2 é possível observar uma comparação entre implantações de aplicações baseada em máquinas virtuais com hipervisor e com a utilização de contêineres. Uma implantação baseada em hipervisor é ideal quando os aplicativos exigem diferentes sistemas operacionais ou sistemas operacionais de diferentes versões. E, por outro lado, nos sistemas baseados em contêineres, as aplicações compartilham o mesmo SO, de modo que essas implantações podem ser significativamente menores em tamanho.

2.4 Contêineres e Computação em Nuvem

Segundo Souza *et al* [53], entende-se por computação em nuvem a provisão de serviços de Tecnologia da Informação (TI) sob demanda com pagamento baseado no uso. Esse modelo tem seu paralelo no consumo de serviços como água, luz e telefone, nos quais o consumidor utiliza o recurso disponibilizado em acordo estabelecido com o provedor. Nessa perspectiva os recursos dos aplicativos de negócios são expostos como serviços sofisticados que podem ser acessados através de uma rede [34]. Assim, o usuário pode ser uma pessoa física que utiliza um serviço de armazenamento *online* ou uma empresa que hospeda seu servidor de correio eletrônico. Em ambos os casos, eles se utilizam de um recurso e é cobrado pelas empresas prestadoras de serviços baseado na sua demanda.

Para Sosinskyv [52], "a computação em nuvem (...) distingue-se pela noção de que os recursos são virtuais e ilimitados e que os detalhes dos sistemas físicos em que rodam o software são abstraídos do usuário". A infraestrutura física que provê estes serviços está instalada em centro de dados (*datacenters*) espalhados pelo globo, acessíveis pela Internet. Por meio de outros paradigmas de computação, tais como a virtualização e a computação orientada a serviços, a infraestrutura de nuvem pode possibilitar tolerância a falhas, alta disponibilidade e, em especial, a elasticidade. A elasticidade é definida pelo *National Institute of Standards and Technology* (NIST) como "(...) capacidade de aprovisionamento de recursos de forma rápida e elástica, em alguns casos automática, para aumentar e diminur o número de recursos. Para o usuário, tais capacidades muitas vezes parecem ser ilimitadas, podendo ser realizadas em qualquer quantidade e a qualquer momento." [51].

Os modelos de implantação tratam sobre o acesso e a disponibilidade de ambientes de computação em nuvem. A restrição ou abertura de acesso depende do processo de negócio, do tipo de informação e do nível de visão. Os modelos de implantação da computação em nuvem podem ser divididos em nuvem pública, privada, comunitária e híbrida. A relação entre esses modelos de implantação, que se diferenciam pelo nível de acesso à infraestrutura [42]. Dessa forma, os principais tipos de nuvem são:

- **Nuvem Privada:** no modelo de implantação de nuvem privada, a infraestrutura de nuvem é utilizada exclusivamente por uma organização. A nuvem pode ser detida, gerenciada e operada pela própria organização, por terceiros ou uma combinação destes; e está localizada dentro da empresa. Quando é utizada uma infraestrutura de nuvem situada remotamente é chamada de **Nuvem Privada Hospedada**.
- **Nuvem Pública:** no modelo de implantação de nuvem pública, a infraestrutura da nuvem é disponibilizada para o público em geral, sendo acessada por qualquer usuário que conheça a localização do serviço. A infraestrutura pode ser gerenciada

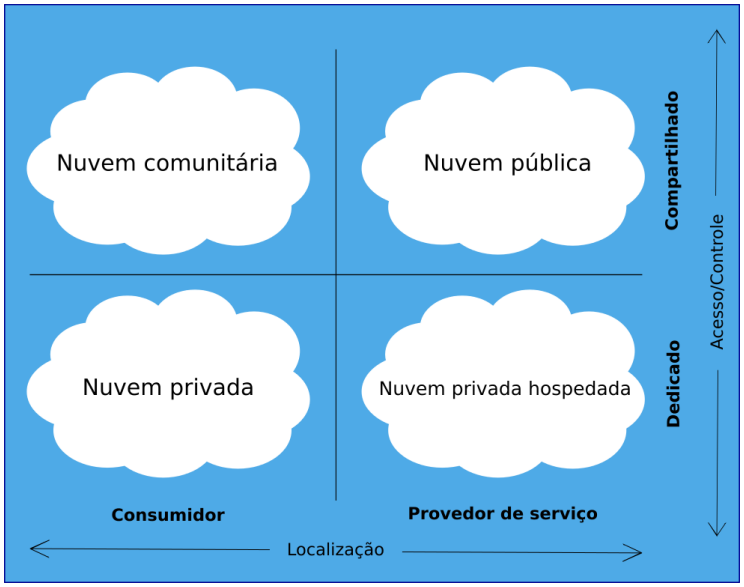

Figura 2.3: Modelos de Implantação da Computação em Nuvem em Relação aos Clientes, Provedores e Controle de Acesso, baseado em [40].

e operada por uma organização empresarial, acadêmica, governamental ou alguma combinação entre estes. A infraestrutura fica localizada internamente no provedor.

- **Nuvem Comunitária:** nesse modelo a infraestrutura de nuvem é compartilhada por várias organizações, e oferece suporte a uma comunidade específica que possui os mesmos interesses (por exemplo, missão, requisitos de segurança, política e considerações de conformidade). Este tipo de modelo de implantação pode existir localmente ou remotamente, e pode ser gerenciada e operada por uma ou mais organizações pertencentes à comunidade, por terceiros ou alguma combinação destes.
- **Nuvem Híbrida:** no modelo de implantação de nuvem híbrida, existe uma composição de duas ou mais nuvens – que podem ser privadas, comunitárias ou públicas – e que permanecem como entidades exclusivas, mas que são agrupadas por tecnologia padrão ou proprietária, possibilitando a portabilidade de dados e de aplicativos.

A Figura 2.3 relaciona os aspectos de hospedagem e controle de acesso da infraestrutura de nuvem com os seus modelos de implantação.

Em ambiente de computação em nuvem pode-se ter três modelos de serviços. Estes modelos são importantes, pois eles definem um padrão arquitetural para soluções de computação em nuvem. Esses modelos são [42]:

- **Software como Serviço (SaaS):** o modelo de SaaS proporciona softwares com propósitos específicos que são disponíveis para os usuários através da Internet. Os softwares são acessíveis a partir de vários dispositivos do usuário por meio de uma

interface *thin client* como um navegador Web. No SaaS, o usuário não administra ou controla a infraestrutura subjacente, incluindo rede, servidores, sistemas operacionais, armazenamento, ou mesmo as características individuais da aplicação, exceto configurações específicas.

- **Plataforma como Serviço (PaaS):** a PaaS oferece uma infraestrutura de alto nível de integração para implementar e testar aplicações na nuvem. O usuário não administra ou controla a infraestrutura subjacente, incluindo rede, servidores, sistemas operacionais ou armazenamento, mas tem controle sobre as aplicações implantadas e, possivelmente, as configurações de aplicações hospedadas nesta infraestrutura. A PaaS fornece um sistema operacional, linguagens de programação e ambientes de desenvolvimento para as aplicações, auxiliando a implementação de softwares, já que contém ferramentas de desenvolvimento e colaboração entre desenvolvedores. Em geral, os desenvolvedores dispõem de ambientes escaláveis, mas eles têm que aceitar algumas restrições sobre o tipo de software que se pode desenvolver, desde limitações que o ambiente impõe na concepção das aplicações, tais como, linguagens de programação, bibliotecas, serviços e ferramentas suportadas pelo provedor.
- **Infraestrutura como Serviço (IaaS):** por meio deste modelo é oferecido ao consumidor os recursos, tais como servidores, rede, armazenamento e outros recursos de computação fundamentais para construir um ambiente de aplicação sob demanda, que podem incluir sistemas operacionais e aplicativos. A IaaS possui algumas características, tais como uma interface única para administração da infraestrutura, API *(Application Programming Interface)* para interação com *hosts*, *switches*, balanceadores, roteadores e o suporte para a adição de novos equipamentos de forma simples e transparente. Em geral, o usuário não administra ou controla a infraestrutura da nuvem, mas tem controle sobre os sistemas operacionais, armazenamento e aplicativos implantados, e, eventualmente, seleciona componentes de rede, tais como *firewalls*.

A Figura 2.4 demonstra a relação dos modelos, a partir dos atores envolvidos no provimento e consumo dos serviços.

2.4.1 PaaS e Contêineres

Na visão de Dua [36], a Plataforma como um Serviço (PaaS) trouxe produtividade para os desenvolvedores, pois com o seu uso foi possível criar uma camada de abstração sobre a IaaS para implantação de serviços de forma fácil e rápida. A PaaS se concentra em fornecer o ambiente de execução e suporte ao desenvolvimento das aplicações, deixando assim, o provisionamento e o gerenciamento de infraestrutura para a camada subjacente.

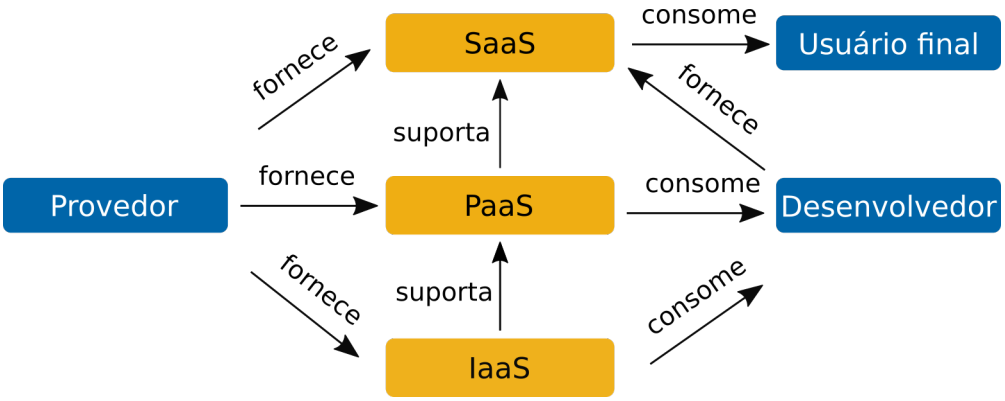

Figura 2.4: Modelos de Serviços e seus Papéis, baseado em [53].

Plataformas como Serviço, tradicionalmente, fornecem mecanismos para desenvolvimento e implantação de aplicativos em ambiente de nuvem computacional, quais sejam serviços para migração de bancos de dados, mapeamento de domínios próprios, *plugins* para ambientes integrados de desenvolvimento (IDE) ou ferramentas para construção e empacotamento da aplicação. As PaaS possuem recursos como agrupamento de máquinas, camadas de roteamento de rede e agendadores para distribuição de cargas de trabalho em ambientes provisionados com máquinas virtuais. Alguns dos principais requisitos de um PaaS para hospedar aplicativos são [36]:

- Isolamento de aplicativos em nível de rede, processamento, memória e armazenamento;
- Menor sobrecarga de desempenho;
- Suporte a várias linguagens de progração e ambientes de execução;
- Compartilhamento justo do uso de CPU;
- Gerenciamento fácil dos eventos do ciclo de vida do aplicativo;
- Persistência e migração do estado do aplicativo.

Segundo Pahl [48], embora as máquinas virtuais sejam o meio mais comum para fornecer a infraestrutura para aplicações em um ambiente PaaS, os contêineres aparecem como uma tecnologia mais adequada para o empacotamento e gerenciamento de aplicativos em PaaS. Uma solução de contêineres resolve as questões elencadas acima por meio de empacotamento virtualizado leve provido por funcionalidades próprias do núcleo do sistema operacional hospedeiro; uma API para gerência do ciclo de vida dos contêineres; um catálogo de modelos pré-construídos para as mais diversas linguagens de programação; e a interoperabilidade - que por meio dos seus próprios mecanismos de execução torna o contêiner agnóstico em relação à mudanças no hospedeiro.

Os contêineres têm uma vantagem inerente sobre VMs devido a melhorias de desempenho e tempo de inicialização reduzido. A PaaS poderia usar o próprio sistema operacional da máquina hospedeira, uma máquina virtual ou um contêiner como ambiente de hospedagem para aplicativos [36]. Dessa forma, tem-se a possibilidade de três cenários:

1. Cada aplicativo é executado em um contêiner no nível de aplicativo diretamente no sistema operacional do hospedeiro, na qual a camada PaaS é executada;
2. Cada aplicativo é executado em uma máquina virtual própria no PaaS;
3. Cada aplicativo é executado em um contêiner que compartilha uma VM comum ou VMs para hospedar aplicativos.

A terceira opção é a opção de implementação para a maioria dos fornecedores PaaS, uma vez que a combinação da máquina virtual e da tecnologia de contêineres satisfaz os requisitos de gerenciamento do ciclo de vida da aplicação, compartilhamento justo do tempo da CPU e isolamento. Um aspecto a ser melhorado é a segurança dos contêineres, tanto do ponto de vista do sistema de arquivos, quanto de rede e do isolamento da memória.

Bernstein [33] afirma que se for necessária alta segurança, pode valer a pena sacrificar o desempenho de uma implantação de contêiner puro, introduzindo uma VM para obter isolamento mais experimentado e verdadeiro. Aponta também que como em qualquer outra tecnologia, é necessário conhecer os requisitos de segurança da implantação e tomar as decisões apropriadas.

A Plataforma como Serviço tem um uso importante para VMs, pois abstrai o sistema operacional de execução do aplicativo do hardware subjacente. A necessidade de cada aplicativo ter sua própria VM é o ponto de discórdia, ainda na visão de Pahl [48]. Como o contêiner tende a ser muito mais leve no uso de memória, ele é a arquitetura preferida em comparação com uma máquina virtual para cada aplicativo. As VMs são usadas para hospedar os contêineres e os outros serviços em PaaS como autenticação, roteamento, controladores de nuvem e armazenamento para persistência de dados é executado diretamente em máquinas virtuais.

O uso de contêineres em PaaS está se expandindo, porém alguns aspectos necessitam ser desenvolvidos para aumentar a adoção de contêineres em plataformas PaaS. Sejam eles a padronização de formato de arquivo de contêiner, permitindo a interoperabilidade em mecanismos de execução, na linha do que está sendo feito pela *Open Container Initiative* (OCI) (mais detalhes na seção 2.1) - e a independência do sistema operaracional hospedeiro, pois os contêineres devem ter uma camada de abstração adicional para que não estejam vinculados a um *kernel* específico ou seu espaço de usuário [36].

Capítulo 3

Docker

O Docker [31] é uma plataforma aberta para desenvolvimento, entrega e execução de aplicativos por meio de contêineres criada e mantida pela empresa Docker Inc. O seu ecossistema abrange um mecanismo de criação de imagens e de execução de contêineres chamado de *Docker Engine*, os repositórios para armazenamento e a distribuição de imagens conhecidos como registros (*Registries*) - o que inclui um serviço em nuvem chamado *Docker Hub* [7] - e uma ferramenta de gerenciamento de *clusters* e orquestração de contêineres, o *Swarm*. Quando nos referimos ao Docker de forma geral trata-se do *Docker Engine*.

A popularização da tecnologia de contêineres deve-se em grande parte ao Docker, que simplificou o seu uso para os desenvolvedores por meio de uma interface de usuário amigável e imagens portáteis. A execução de um contêiner com o uso de uma tecnologia como o LXC requeria conhecimento especializado e trabalho manual significativo [46]. O *Docker Hub* disponibiliza um grande número de imagens de contêineres públicas para *download*, permitindo que os usuários comecem a fazer uso rapidamente da tecnologia.

3.1 Base Tecnológica

Docker é escrito em *Go* [13], que é uma linguagem de programação imperativa de gramática compacta e regular. Essa linguagem é fortemente tipada e possui coleta de lixo (*garbage collector*) para gerenciamento automático de memória. Ela possui também suporte explícito para programação concorrente e os programas são construídos a partir de pacotes, cujas propriedades permitem o gerenciamento eficiente de dependências. As implementações existentes usam um modelo de compilação e ligação tradicional para gerar binários executáveis [14].

O Docker faz uso de diversos recursos do núcleo do sistema operacional para prover a tecnologia de contêineres, essas tecnologias serão detalhadas nas próximas subseções.

3.1.1 Grupos de Controle

Os grupos de controle, mais comumente conhecidos como *cgroups*, são um recurso do *kernel* do sistema operacional Linux que permite que os processos sejam organizados em grupos hierárquicos cuja utilização de vários tipos de recursos pode ser limitada e monitorada [4].

A interface *cgroup* do kernel é fornecida através de um pseudo-sistema de arquivos chamado *cgroupfs*. Um *cgroup* é um conjunto de processos que estão vinculados a um conjunto de limites ou parâmetros definidos através de um sistema de arquivos *cgroup*. Um subsistema é um componente do *kernel* que modifica o comportamento dos processos em um *cgroup*.

Vários subsistemas foram implementados, tornando possível limitar a quantidade de tempo de CPU e de memória disponível, contabilizar o tempo de CPU utilizado, bem como congelar e retomar a execução dos processos em um determinado *cgroup*. Os subsistemas são às vezes também conhecidos como controladores de recursos (ou simplesmente, controladores). Os *cgroups* para um controlador são organizados em uma hierarquia. Essa hierarquia é definida pela criação, remoção e renomeação de subdiretórios dentro do sistema de arquivos *cgroup*. Em cada nível da hierarquia, atributos (por exemplo, limites) podem ser definidos. Os limites e controles fornecidos pelos *cgroups*, geralmente, têm efeito em toda a sub-hierarquia sob o *cgroup* onde os atributos são definidos. Assim, por exemplo, os limites colocados em um *cgroup* em um nível mais alto na hierarquia não podem ser excedidos pelos *cgroups* descendentes [4].

Assim, um *cgroup* limita um aplicativo a um conjunto específico de recursos. Os grupos de controle permitem que o *Docker Engine* compartilhe recursos de hardware disponíveis em contêineres e, opcionalmente, imponha limites e restrições. Dessa forma, um dos usos possíveis é limitar a memória disponível para um contêiner específico [9].

3.1.2 Namespaces

Namespaces são um recurso do *kernel* Linux que permite que um sistema isole uma coleção de processos para que não possam ver certas partes do sistema geral. Um *namespace* envolve um recurso do sistema global em uma abstração, que faz parecer para os processos dentro daquele *namespace* que eles têm sua própria instância isolada do recurso global. As alterações desse recurso global são visíveis para processos que são membros do *namespace*, mas são invisíveis para outros processos fora do *namespace*.

Dessa forma, exemplos de recursos que podem ser abstraídos incluem identificações (ID) de processos, nomes de *host*, IDs de usuário, acesso à rede, comunicação entre processos e sistemas de arquivos [23].

Com *namespaces* é possível, por exemplo, utilizando o isolamento de identificação do processo (PID, namespace), fazer um processo pensar que é o único processo em execução no servidor. Ele não teria acesso para saber que existem outros processos, e não teria a capacidade de enviar sinais para qualquer um desses processos. Dessa forma, um contêiner pode executar seu próprio processo de inicialização como PID 1, enquanto o sistema hospedeiro vê esse processo como um PID inteiramente diferente [39].

O Docker utiliza a tecnologia de *namespaces* para fornecer o espaço de trabalho isolado que é chamado de contêiner. Quando um contêiner é executado o Docker cria um conjunto de *namespaces* para esse contêiner. Esses *namespaces* fornecem uma camada de isolamento. Cada aspecto de um contêiner é executado em um *namespace* separado e seu acesso é limitado a esse *namespace*. O *Docker Engine* usa os seguintes *namespaces* no Linux:

- *pid namespace* para isolamento do processo (PID - *Process ID*);
- *net namespace* para gerenciamento de interfaces de rede (NET - *Networking*);
- *ipc namespace* para gerenciamento de acesso aos recursos IPC (IPC - *InterProcess Communication*);
- *mnt namespace* para gerenciamento de pontos de montagem do sistema de arquivos (MNT - *Mount*);
- *uts namespace* para isolamento do kernel e identificadores de versão (UTS - *Unix Timesharing System*).

3.1.3 Union Filesystem

O *Unionfs* é um sistema de arquivos empilhável que permite aos usuários especificar um conjunto de diretórios que são apresentados aos usuários como um único diretório virtual, mesmo que esses diretórios possam vir de diferentes sistemas de arquivos. O *Unionfs* cria camadas de forma simultânea em cima de vários sistemas de arquivos ou em diretórios diferentes dentro do mesmo sistema de arquivos. Essa técnica de camadas é conhecida como empilhamento. Como o Unionfs intercepta operações vinculadas a sistemas de arquivos de nível inferior, ele pode modificar operações para apresentar a visão unificada ao *kernel* do sistema operacional. Dessa forma, o referido sistema de arquivos virtual pode ser utilizado de maneira transparente para as aplicações em execução [50].

Uma coleção de diretórios mesclados é chamada de união, e cada diretório físico é chamado de ramo. No *Unionfs* é atribuído uma precedência para cada ramo. Um ramo com uma precedência maior substitui um ramo com uma precedência mais baixa. O *Unionfs*

opera por meio de diretórios. Se um diretório existe em dois ramos subjacentes, o conteúdo e os atributos do diretório *Unionfs* são a combinação dos dois diretórios inferiores. São removidas automaticamente quaisquer entradas de diretório duplicadas para que os usuários não sejam confundidos por nomes de arquivos ou diretórios duplicados. Assim, se um arquivo existir em dois ramos, o conteúdo e atributos do arquivo *Unionfs* são iguais ao arquivo no ramo de prioridade mais alta, e o arquivo no ramo de prioridade inferior é ignorado [54].

O *Docker Engine* usa UnionFS para fornecer os blocos de construção para contêineres. O Docker pode utilizar diversas variantes do *UnionFS*, incluindo AUFS, btrfs, vfs e DeviceMapper [9].

Docker Engine combina os *namespaces*, grupos de controle e UnionFS em um *wrapper* chamado de formato de contêiner. O formato de contêiner padrão é *libcontainer*. No futuro, Docker pode suportar outros formatos de contêiner, integrando-se com tecnologias como o BSD Jails ou Solaris Zones.

3.2 Arquitetura do Docker

3.2.1 Docker Engine

O *Docker Engine* usa uma arquitetura cliente-servidor composta de três componentes: um processo executado em segundo plano (*daemon*), um cliente e uma API REST. O cliente é a principal interface do usuário com o Docker. Trata-se de uma aplicação binária na qual são executados comandos para interagir com o *daemon* do Docker. O cliente faz uso da API REST para executar essas operações com o *daemon*. Uma API REST define um conjunto de funções que os desenvolvedores podem executar solicitações e receber respostas via protocolo HTTP, como GET e POST. A API REST do Docker especifica as interfaces que os programas podem utilizar para interagir com o *daemon*. O *daemon* cria e manipula os objetos Docker tais como imagens, contêineres, rede e volumes de dados [9].

Em resumo, o cliente conversa com o *daemon*, que realiza o trabalho de construir, executar e distribuir as imagens e os contêineres. O cliente e o *daemon* podem ser executados no mesmo sistema ou um cliente pode se conectar a um *daemon* Docker remoto. O cliente Docker e o *daemon* se comunicam usando a API REST, por meio de soquetes UNIX ou uma interface de rede. A Figura 3.1 apresenta a interação entre os elementos da arquitetura do *Docker Engine*.

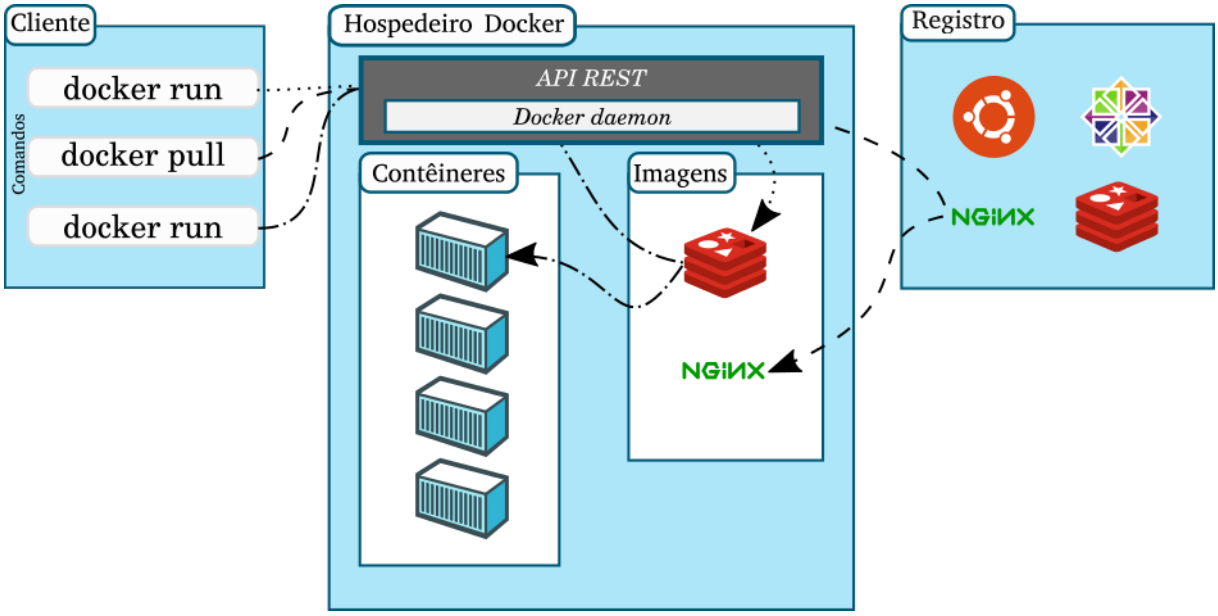

Figura 3.1: Relacionamento entre os Elementos da Arquitetura do *Docker Engine*, baseado em [9].

3.2.2 Imagens Docker

A imagem Docker é um modelo de somente leitura a partir do qual os contêineres são instanciados. Pode, por exemplo, conter o sistema operacional Ubuntu com servidor web Apache e uma aplicação instalada. Cada imagem consiste em uma série de camadas (*layers*). Essas camadas são combinadas em um único filesystem consistente por meio da tecnologia *UnionFS*. Quando uma imagem é atualizada ou recriada, somente as camadas afetadas são atualizadas. Esse comportamento se repete quando as imagens são distribuídas, ou seja, é necessário transferir somente as camadas atualizadas de uma imagem já presente em um *host*. O mecanismo do Docker determina quais camadas precisam ser atualizadas em tempo de execução.

Toda imagem é construída a partir de uma imagem base. Essas imagens base podem ser construídas a partir de uma imagem vazia (*scratch*) ou de outras imagens pré-construídas. As imagens Docker são construídas usando um conjunto simples, descritivo de passos chamados de instruções. Cada instrução cria uma nova camada e a adiciona à imagem. As instruções incluem ações como executar um comando de linha, adicionar um arquivo ou pasta, criar uma variável de ambiente e qual o processo a ser executado ao iniciar um contêiner criado a partir desta imagem. Estas instruções são armazenadas em um arquivo chamado de *Dockerfile*. Quando é solicitada uma compilação de uma imagem, o *Docker Engine* lê esse arquivo, executa as instruções e retorna uma imagem final.

A solicitação de criação de uma imagem é feita, via cliente do Docker, com o comando "*docker build*", conforme demonstrado no Quadro 3.1.

Quadro 3.1: Comando para Criação de uma Imagem Docker.

```
docker build [-t <nome da imagem>] .
```

No comando apresentado no Quadro 3.1, após a palavra chave *build* é informado o parâmetro "-t", no qual é indicado o nome da imagem e em seguida o contexto de construçao. O contexto de construçao é o diretório da máquina física do qual o Docker iniciará a construção, e onde estão os arquivos que serão utilizados nela (inclusive o *Dockerfile*), nesse caso, é o diretório corrente (indicado pelo "."). A compilação é executada pelo *daemon* Docker, não pelo cliente.

Assim, a primeira coisa que um processo de construção faz é enviar todo o contexto (recursivamente) para o *daemon*. Antes que o *daemon* Docker execute as instruções no *Dockerfile*, ele executa uma validação preliminar e retorna um erro se a sintaxe estiver incorreta. Sempre que possível, o Docker irá reutilizar as imagens intermediárias (*cache*), para acelerar, significativamente, o processo de criação da imagem.

As instruções em um *Dockerfile* são escritas uma por linha, sendo que primeiro deve estar a instrução, e em seguida seus argumentos. Na Tabela 3.1 estão relacionadas algumas instruções mais relevantes na composição desses arquivos.

Tabela 3.1: Instruções Relevantes para Composição de *Dockerfiles* [11].

Instrução	**Objetivo**
FROM	A instrução FROM define a imagem base a partir da qual a nova imagem será criada. Essa deve ser a primeira instrução no Dockerfile.
RUN	A instrução RUN irá executar quaisquer comandos em uma nova camada criada na parte superior da imagem atual e gravar o resultado final. A imagem resultante será usada pela próxima instrução do Dockerfile.

CMD	O objetivo principal da instrução CMD é fornecer o processo padrão para um contêiner de execução. O CMD define o comando a ser processado ao executar a imagem. Esse processo padrão pode ser um *software* executável. Caso o contêiner execute o mesmo executável sempre, é recomendado utilizar a instrução ENTRYPOINT em combinação com CMD. Se o usuário especificar argumentos para a execução do Docker, eles substituirão o padrão especificado no CMD.
EXPOSE	A instrução EXPOSE informa ao Docker que o contêiner escuta nas portas de rede especificadas em tempo de execução. Essa instrução não torna as portas do contêiner acessíveis ao *host*. Para fazer isso, deve-se usar o sinalizador "-p" para publicar um intervalo de portas ou o sinalizador "-P" para publicar todas as portas expostas. Pode ser exposto um número de porta e publicá-lo externamente em outro número.
ENV	A instrução ENV define uma variável de ambiente por meio de um nome e um valor. Este valor estará disponível no ambiente para todos as instruções subsequentes, *Dockerfiles* que tiverem essa imagem como base e no contêiner executado a partir dela. Os valores dessas variáveis podem substituídos em tempo de construção e de execução.
ADD/COPY	As instruções ADD e COPY copiam novos arquivos ou diretórios e os adiciona ao sistema de arquivos da imagem no caminho especificado. Vários recursos podem ser especificados, mas se forem arquivos ou diretórios, eles devem ser relativos ao diretório de origem que está sendo construído (o contexto de compilação). O que diferencia as duas instruções é que o COPY é mais performático e o ADD pode também fazer cópias a partir de URLs de arquivos remotos.

ENTRYPOINT	Uma instrução ENTRYPOINT permite configurar um contêiner que será executado como um executável. Os argumentos passados para o comando "docker run <imagem>" serão anexados após todos os elementos do ENTRYPOINT e substituirão todos os elementos especificados usando CMD. Isso permite que os argumentos sejam passados para o executável definido no ENTRYPOINT. A instrução ENTRYPOINT pode ser substituído usando a opção "–entrypoint" na inicialização do contêiner. Somente a última instrução ENTRYPOINT no Dockerfile terá efeito.
VOLUME	A instrução VOLUME cria um ponto de montagem com o nome especificado e marca-o como um volume que pode ser montado externamente no *host* e por outros contêineres.
USER	A instrução USER define o nome de usuário ou identificação de usuário (UID) para execução de quaiquer instruções RUN, CMD e ENTRYPOINT que virão a seguir no *Dockerfile*. Esse será o usuário de execução do contêiner.
WORKDIR	A instrução WORKDIR define o diretório de trabalho para quaisquer instruções RUN, CMD, ENTRYPOINT, COPY e ADD que o sigam no Dockerfile. Se o WORKDIR não existir, ele será criado.

Outros comandos para manipulação de imagens são para listagem das imagens presentes no *host* e para exclusão de imagens. O uso desses comandos estão mostrados no Quadro 3.2

Quadro 3.2: Comandos para Listagem e para Exclusão de uma Imagem Docker.

```
# Lista as imagens presentes no host
docker images

# Remove imagem do host
docker rmi <id ou nome>
```

3.2.3 Registros Docker

Registro (*registry*) é um repositório para armazenamento e distribuição de imagens previamente criadas. Depois das imagens construídas pode ser realizado o *upload* para o *registry*. O cliente Docker é utilizado para procurar por imagens já publicadas e realizar o *download* para executar contêineres a partir delas.

Os registros podem ser públicos ou privados. A Docker Inc fornece um registro público chamado *Docker Hub* [1]. Esse registro público possui uma enorme coleção de imagens criadas por desenvolvedores e empresas para *download*. Os registros privados tem acesso permitido somente a usuários autorizados e residem normalmente em uma rede própria protegida por um *firewall*. Os registros privados são excluídos por padrão dos resultados de pesquisa e permitem controlar com precisão onde as imagens estão sendo armazenadas e quem pode ter acesso a elas. Os registros Docker são usados com os objetivos de integrar o armazenamento e a distribuição de imagens no fluxo de trabalho interno de desenvolvimento, e operacionalizar o processo de entrega contínua das aplicações em contêineres.

Para fazer o *download* de uma determinada imagem do registro é feito uma operação de (*pull*) via cliente do Docker. As imagens são identificadas por um nome e uma etiqueta (*tag*). A *tag* é indicada após o nome da imagem divididas pelo sinal ":". Se nenhuma *tag* for fornecida, o *Docker Engine* usa a *tag* "*latest*". Por padrão, a operação de *pull* do Docker baixa as imagens do *Docker Hub*. Para realizar a operação de um registro privado é preciso indicar o endereço desse registro antes do nome da imagem (separados pelo caracter "/"). Um endereço de registo é semelhante a uma URL, mas não contém um especificador de protocolo (https: //).

O *download* de uma imagem a partir de um registro Docker é realizado conforme comando apresentado no Quadro 3.3.

Quadro 3.3: Comando para *Download* de uma Imagem de um *Docker Registry*.

```
docker pull [OPÇÕES] [REGISTRO PRIVADO/] NOME [:TAG]
```

O *upload* de uma imagem construída localmente para um registro é feito de forma análoga. Por meio do cliente Docker é realizada uma operação de *push* como indicado no Quadro 3.4.

[1] `http://hub.docker.com`

Quadro 3.4: Comando para subir uma imagem para um registro Docker

```
docker push [OPÇÕES] [REGISTRO PRIVADO/] NOME [:TAG]
```

3.2.4 Contêineres Docker

Um contêiner Docker mantém tudo o que é necessário para uma aplicação ser executada. Um contêiner consiste em um sistema operacional, os arquivos adicionados pelo usuário e os metadados. Na imagem, da qual o contêiner é instanciado, é definido o processo a ser executado quando ele é iniciado, e uma variedade de outros dados de configuração. A imagem Docker é somente leitura. Quando o Docker inicializa um contêiner a partir de uma imagem, ele adiciona uma camada de leitura e escrita na parte superior da imagem (usando *UnionFS*), no qual a aplicação pode ser executada. Diversar operações podem ser efetuadas em um contêiner, tais como iniciar, parar, mover e excluir [9]. As seguintes operações são efetuadas pelo *Docker Engine* para a execução de um contêiner.

1. o *Docker Engine* verifica a presença da imagem solicitada. Se a imagem já existir, então o mecanismo do Docker a utiliza para executar o novo contêiner. Se ela não existir localmente na máquina, então o *Docker Engine* realiza o *download* do registro;
2. Uma vez que o Docker Engine possui a imagem, ele a utiliza para criar o contêiner. O contêiner é criado no sistema de arquivos, e a camada de leitura e escrita é adicionada a imagem;
3. É criada uma interface de rede que permite que o contêiner Docker se comunique com o *host* hospedeiro. O Docker localiza e vincula um endereço IP disponível na pilha do Docker;
4. Executa-se o programa definido na imagem e o mecanismo do Docker passa a capturar as informações da entrada e da saída padrão do contêiner, e erros da aplicação em execução.

A execução de um contêiner é feita por meio do cliente Docker como indicado no Quadro 3.5.

Quadro 3.5: Comando para execuçao de um contêiner a partir de uma imagem Docker

```
docker run [OPÇÕES] [REGISTRO PRIVADO/] IMAGEM[:TAG] [
   COMANDO] [ARG...]
```

Capítulo 4

Trabalhos Relacionados

Os benefícios oferecidos pela adoção de uma infraestrutura de contêineres, tais como a otimização dos recursos computacionais, a redução de falhas em função da configuração, as atualizações e os erros de *softwares*, e a rapidez na inicialização têm levado à realização de estudos comparativos de desempenho entre essa tecnologia e outras plataformas de provimento de aplicações, em especial no ambiente de computação em nuvem.

Esses estudos têm como objetivo, de maneira geral, identificar a viabilidade da adoção dos contêineres confrontando sua performance em relação à aspectos de uso de CPU, memória, entrada e saída (E/S) de disco, e E/S de rede com a virtualização por hipervisor e instalação nativa na máquina (chamada de *baremetal*). Para o presente trabalho foi realizada uma revisão da literatura sobre o tema, com destaque para a utilização do Docker como plataforma de execução de contêineres.

4.1 Análise de Desempenho de Contêineres com Uso de Aplicações Sintéticas

No artigo [44], a realização da análise de desempenho, também chamada de *benchmarking*, fez uso de diferentes ferramentas para quantificar o nível de sobrecarga introduzido pelas soluções baseadas em hipervisor e contêineres em relação à um ambiente não virtualizado. A avaliação foi realizada utilizando as ferramentas KVM (*Kernel-based Virtual Machine*) como ambiente de virtualização com hipervisor, LXC e Docker como tecnologias de contêineres e o OSv, que é uma solução de sistema operacional projetada exclusivamente para a nuvem, executado em cima de um hipervisor. Nesse trabalho foi definido o desempenho nativo (não virtualizado) como base para medir a sobrecarga da virtualização. As ferramentas de *benchmark* medem (usando cargas de trabalho genéricas) CPU, memória, E/S de disco e desempenho de E/S. Cada medição individual foi repetida 15 vezes.

O desempenho de CPU foi testado por meio da ferramenta Y-cruncher que calcula o valor Pi e outras constantes, executando como um aplicativo de teste de estresse para CPU. Como resultado foi verificado que ambas as soluções baseadas em contêiner apresentam melhor desempenho do que o KVM. Considerando apenas o tempo de computação, os contêineres exibem desempenho quase semelhante ao ambiente nativo. O teste de CPU foi realizado também com o *software* NBENCH, que diferentemente do Y-cruncher é uma ferramenta de *single thread*, e o seu algoritmo produz três índices diferentes para avaliação: Índice Inteiro, Índice de Ponto Flutuante e Índice de Memória. Exceto para o Índice de Memória, onde a KVM tem uma degradação de desempenho em comparação com as outras plataformas (cerca de 30 por cento), todas as tecnologias analisadas atingem desempenhos quase iguais. Os autores validaram os dados obtidos com a realização de outros testes com as ferramentas Geekbench e Linpack.

A avaliação do desempenho de disco foi realizada por meio da ferramenta Bonnie++, que é um software de código aberto de referência para essas aferições. Usando um tamanho de arquivo de teste de pelo menos o dobro do tamanho da memória do sistema foram realizadas operações de escrita e de leitura sequencial. As duas plataformas baseadas em contêineres oferecem um desempenho muito similar em ambos os casos, que são bastante próximos do nativo. O *throughput* (taxa de transferência) de escrita do KVM é aproximadamente um terço, e o de escrita quase um quinto do nativo. Da mesma forma, foram realizadas operações de leitura e de escrita aleatórias que tiveram comportamente similar ao sequencial. Uma observação relevante foi de que nas medições de desempenho de disco com o sistema nativo, KVM e LXC, foram obtidos quase sempre o mesmo resultado sem qualquer desvio significativo.

Todavia, diferente comportamento foi observado com Docker, que estava realizando, para algumas execuções, desempenho ainda melhor do que o nativo. Nos testes o sistema nativo e LXC usavam o formato de arquivo padrão para o Linux, o KVM usava um formato de imagem qcow2, enquanto o Docker utilizava o sistema de arquivos AUFS, que suporta camadas e habilita o controle de versão de imagens. Os autores ainda indicam que foram encontradas imcompatibilidades entre os resultados de Bonnie++ e outras ferramentas como Sysbench, sugerindo que a estimativa de desempenho de E/S de disco pode ser complicada.

Para testar o desempenho de memória foi utilizada a ferramenta STREAM, que mede o desempenho usando operações de *kernels* de vetores muito simples. O desempenho medido pela ferramenta tem uma forte dependência do tamanho da *cache* da CPU. Por esta razão, o tamanho do input para a ferramenta deve ser, pelo menos, 4 vezes o tamanho da memória *cache* disponível. Seguindo esta condição, os resultados obtidos indicam que os ambientes KVM, Docker e LXC alcançaram desempenho semelhante ao da execução

nativa. O desempenho de OSv é aproximadamente a metade dos outros de acordo com estes *benchmarks*.

Para o *benchmark* de rede os autores, por meio da ferramenta Netperf - ferramenta de referência com vários testes pré-definidos para medir o desempenho da rede entre dois *hosts* - foi realizada uma transferência de dados unidirecional de pedido e resposta com o protocolo TCP e UDP entre duas máquinas idênticas, diretamente conectadas com uma interface Ethernet de 10 Gigabit. Os resultados representam a média em 15 execuções. Usando TCP, LXC e Docker foram alcançandos desempenhos quase iguais em comparação com o nativo, e para o KVM foi obtido um desempenho 28,41% mais lento em relação ao nativo. OSv funcionou melhor do que o KVM e introduz um hiato igual a 26,46% em relação ao sistema nativo. Todas as plataformas ofereceram menor *throughput* com UDP. LXC e Docker tiveram um desempenho comparável entre eles, mas respectivamente 42,14% e 42,97% inferior ao nativo. No KVM o *overhead* é o maior (54,35%) e OSv foi 46,88% pior do que nativo. Após os experimentos relatados em [44], os autores chegaram a conclusão de que o nível de sobrecarga introduzido por contêineres pode ser considerado quase insignificante.

No trabalho [49], o desempenho dos contêineres Docker é avaliado com base na utilização de recursos do sistema. Este *benchmarking* foi implementado em um sistema Linux, no qual o Docker foi instalado diretamente sem qualquer *hypervisor*. Diferentes ferramentas foram usadas para realização dos experimentos. O desempenho baseado no sistema de arquivos é avaliado usando Bonnie++. Outros recursos do sistema, como a utilização da CPU e da memória foram avaliados por meio de um *script* desenvolvido na linguagem Python utilizando a biblioteca psutil [1]. O psutil é uma biblioteca multi-plataforma para recuperar informações sobre processos de roteamento e utilização do sistema (CPU, memória, discos, rede) em Python. Ele é usado principalmente para monitoramento do sistema, criação de perfil, e limitação de recursos de processo e gerenciamento de processos em execução. Ele suporta muitos sistemas operacionais como LINUX, Windows, FreeBSD, Sun Solaris etc, em arquiteturas de 32 e 64 bits.

Para os testes de desempenho das operações em discos foi fixado em 40 MB o tamanho dos arquivos para medição da velocidade de leitura e de escrita pelos contêineres. Os resultados obtidos demonstraram uma variação de desempenho um pouco melhor do sistema operacional nativo em relação ao Docker (a leitura e a escrita do contêiner foram de 1351Kb/s e 507Kb/s respectivamente, e do S.O nativo foram 4388Kb/s e 536Kb/s). A utilização de memória e da CPU utilizada no experimento foi aferida de forma absoluta e os números foram comparados entre o ambiente nativo e em Docker. O uso de memória foi semelhante em ambos os casos, bem como o uso de CPU. No cenário de uso de rede foi

[1] `https://github.com/giampaolo/psutil`

identificada uma diferença entre os contadores de E/S do Docker e o sistema operacional hospedeiro, onde o desempenho foi melhor. A partir das avaliações feitas usando a ferramenta de desempenho Bonnie ++ e psutil, os autores concluíram que o Docker tem um desempenho que pode ser comparado ao desempenho de um sistema operacional rodando em *baremetal*.

Por fim, dentro da perspectiva de avaliação com utilização de aplicações sintéticas, no artigo [38] os autores exploraram o desempenho das implementações de máquinas virtuais tradicionais e as contrastaram com o uso de contêineres Linux. Foi utilizado um conjunto de cargas de trabalho que enfatizam a CPU, a memória, o armazenamento e os recursos de rede. Os experimentos foram realizados utilizando o KVM como hipervisor e o Docker como um plataforma de contêineres. A motivação nasceu da ideia de que no ambiente de computação em nuvem, os níveis extras de abstração envolvidos na virtualização com hipervisor reduzem o desempenho da carga de trabalho, que é repassado aos clientes na relação preço/desempenho, ou seja, o valor pago não entrega toda a performance possível.

É possível melhorar a qualidade do serviço prestado com a utilização da tecnologia de contêineres, pois simplificam a implantação de aplicativos e continuam permitindo o controle dos recursos alocados para diferentes aplicativos. O artigo examinou as duas formas diferentes de conseguir o controle de recursos - contêineres e máquinas virtuais - e comparou o desempenho de um conjunto de cargas de trabalho em ambos os ambientes com a execução nativa. Foram objeto do estudo um conjunto de *benchmarks* que enfatizam aspectos diferentes como computação, largura de banda de memória, latência de memória, largura de banda de rede e largura de banda de E/S, bem como o desempenho de duas aplicações reais, a saber, Redis e MySQL.

O objetivo foi isolar e compreender a sobrecarga introduzida por máquinas virtuais (KVM) e contêineres (Docker) em relação ao Linux não virtualizado. Os autores destacam que é esperado que outros hipervisors como Xen, VMware ESX e Microsoft Hyper-V forneçam desempenho semelhante ao KVM, uma vez que eles usam os mesmos recursos de aceleração de hardware. Da mesma forma, outras ferramentas contêiner devem ter desempenho igual ao Docker quando usam os mesmos mecanismos. O foco dado da análise de desempenho foi na questão da sobrecarga em comparação com a execução nativa e não virtualizada porque reduz os recursos disponíveis para o trabalho produtivo. Foram utilizadas ferramentas de *benchmarks* para medir individualmente a CPU, a memória, a rede e a sobrecarga de armazenamento.

Também foi medido o uso de duas aplicações de servidor real: Redis e MySQL. Em relação ao uso de CPU foram realizados dois testes. No primeiro foi utilizada a ferramenta PXZ para compressão de dados sem perdas. A compressão é um componente frequentemente usado em cargas de trabalho em nuvem. Com o PXZ foi comprimido 1 GB de

dados da Wikipedia utilizando 32 threads. O *throughput* foi avaliado e o sistema nativo e o Docker tiveram desempenho são muito semelhantes. Enquanto KVM é 22% mais lento. Ainda foi destacado que o ajuste do KVM por vCPU fixando e expondo a topologia de *cache* faz pouca diferença no desempenho. No segundo experimento foi usado o Linpack que resolve um sistema denso de equações lineares.

O desempenho do Linpack é quase idêntico tanto no Linux como no Docker. No entanto, o desempenho KVM é visivelmente pior, mostrando os custos de abstrair ou esconder detalhes de hardware de uma carga de trabalho que pode tirar proveito dela. Ao ser incapaz de detectar a natureza exata do sistema, a execução emprega um algoritmo mais geral com conseqüentes penalidades de desempenho. O uso de memória foi avaliado utilizando a ferramenta STREAM. O *benchmark* tem quatro componentes: COPY, SCALE, ADD e TRIAD que executam acessos sequenciais a memória onde todos os dados dentro da página de memória são acessados antes de passar para a próxima página. Foi observado que o desempenho no Linux, Docker e KVM é quase idêntico, com os dados médios exibindo uma diferença de apenas 1,4% nos três ambientes de execução. No caso de acesso aleatório a memória tivemos os mesmos *overheads* para ambientes virtualizados e não-virtualizado.

Para o desempenho de rede foi utilizado o nuttcp para medir o *throughput* de uma transferência de dados unidirecional em massa sobre uma única conexão TCP com o padrão MTU de 1500 bytes entre o sistema em teste e uma máquina idêntica conectada usando uma conexão Ethernet direta de 10 Gbps entre duas interfaces. O Docker anexa todos os containers no *host* a uma ponte e conecta a ponte à rede via NAT. A configuração do KVM usou virtio e vhost para minimizar a sobrecarga de virtualização. Em um cenário ligado a E/S, foi determinada a sobrecarga medindo a quantidade de ciclos de CPU necessários para transmitir e receber dados. Foi observado que o uso de pontes e NAT por Docker aumenta consideravelmente o comprimento do caminho de transmissão elevando assim a sobrecarga no lado de recebimento. Os contêineres que não usam NAT têm desempenho idêntico ao Linux nativo.

O *benchmark* realizado com o Redis foi realizado pelo autores em função de o armazenamento de chave-valor baseado em memória ser comumente usado na nuvem para armazenar de *cache*, informações de sessão e outros conjuntos de dados não estruturados. O modelo de uso desse tipo de aplicação geralmente é sensível à latência da rede. O teste consistiu em um número de clientes que emitem pedidos ao servidor, sendo que metade das requisições era de leitura e a outra metade de escrita. Cada cliente mantinha uma conexão TCP persistente com o servidor e podia encaminhar até 10 solicitações simultâneas sobre essa conexão. As chaves tinham 10 caracteres de comprimento e os valores foram gerados para uma média de 50 bytes. Para cada execução, o conjunto de dados é limpo e,

em seguida, é emitida uma seqüência determinística de operações, resultando na criação gradual de 150 milhões de chaves. O consumo de memória do servidor Redis atingiu um máximo de 11 GB durante a execução.

A métrica utilizada foi o *throughput* (em solicitações por segundo) em relação ao número de conexões de cliente para os diferentes modelos de implantação. Na implementação nativa, o subsistema de rede foi bastante suficiente para lidar com a carga. Assim, à medida que se escalava o número de conexões de clientes, o principal fator que limitava o throughput de um servidor Redis era a saturação da CPU. Ao se usar o Docker com a pilha de rede do *host*, foi verificado que tanto a taxa de transferência quanto a latência são praticamente as mesmas do caso nativo. Porém o comportamento contrário pode ser observado quando se usa o Docker com NAT ativado, neste caso, a latência introduzida cresceu com o número de pacotes recebidos pela rede. Além disso, o NAT consome ciclos de CPU, impedindo assim o Redis de alcançar o desempenho máximo observado nas implantações com pilhas de rede nativas. Da mesma forma, ao executar no KVM, é adicionado aproximadamente 83ms de latência a cada transação. Vimos que a VM tem baixa taxa de transferência em baixa simultaneidade, mas assintoticamente aproxima o desempenho nativo à medida que a concorrência aumenta. Além de 100 conexões, o *throughput* de ambas as implementações são praticamente idênticos.

Para a medição da performance com o banco de dados relacional MySQL foi executado o *software* SysBench oltp em uma única instância. O *benchmark* oltp usa um banco de dados pré-carregado com 2 milhões de registros e executa um conjunto fixo de transações de leitura e gravação, escolhendo entre cinco consultas SELECT, duas consultas UPDATE, uma consulta DELETE e um INSERT. As medições fornecidas pelo SysBench são estatísticas de latência de transação e *throughput* em transações por segundo. O número de clientes foi variado até saturação e dez execuções foram utilizadas para produzir cada ponto de dados. Cinco configurações diferentes foram medidas: MySQL executando normalmente em Linux (nativo), MySQL em Docker usando rede de *host* e um volume (Docker net = host volume), usando um volume e a configuração padrão de rede do Docker (Docker NAT volume) armazenando o banco de dados dentro do sistema de arquivos de container (Docker NAT AUFS) e MySQL rodando em KVM. Foi analisado o rendimento da transação em função do número de usuários simulados pelo SysBench. O Docker teve desempenho semelhante ao nativo, com a diferença assintótica aproximando 2% em maior concorrência. A KVM tem uma sobrecarga muito superior, sendo superior a 40% em todos os casos medidos. O AUFS introduz uma sobrecarga significativa uma vez que a E/S passou por várias camadas. Foram testados diferentes protocolos de armazenamento KVM e observou-se que eles não fazem diferença no desempenho de uma carga de trabalho em *cache* como neste teste.

Os autores, observaram que dadas suas implementações, contêineres e VMs impuseram quase nenhuma sobrecarga no uso de CPU e memória. O impacto maior impacto sofrido foi em operações de E/S. Essa sobrecarga veio na forma de ciclos extras para cada operação de E/S, de modo que as operaçõs de E/S menores sofrem muito mais do que as maiores. Essa sobrecarga aumentou a latência de E/S e reduziu os ciclos de CPU disponíveis para o trabalho útil, limitando a taxa de transferência. O Docker acrescenta vários recursos, como imagens em camadas e NAT que o torna mais fácil de se usar do que LXC, mas esses recursos vêm com um custo de desempenho. Assim o mecanismo Docker usando configurações padrão pode não ser mais rápido do que o KVM. Os aplicativos que tem uso intensivo de sistema de arquivos ou disco deve evitar o uso do AUFS e utilizar volumes. O *overhead* inserido pelo uso de NAT na camada de rede pode ser facilmente eliminado usando a opção "-net = host", mas com isso não se fará uso dos benefícios dos namespaces da rede. Os autores indicam que o modelo de um endereço IP por contêiner, conforme proposto pelo projeto Kubernetes, pode fornecer flexibilidade e desempenho.

Por fim os autores concluem que o Docker igualou ou excedeu o desempenho do KVM em todos os casos testados. Os resultados mostram que tanto o KVM quanto o Docker introduziram uma sobrecarga desprezível para desempenho de CPU e memória (exceto em casos extremos). Para cargas de trabalho com intensivas operações de E/S ambas as formas de virtualização devem ser usadas com cuidado.

4.2 Análise de Desempenho de Contêineres em Aplicações de Bioinformática

Uma outra perspectiva é trazida pelo artigo [37], no qual os custos de desempenhos dos contêineres foi analisado sob a ótica da Bioinformática. A motivação dos autores foi a possibilidade de trazer para este ramo da TI os benefícios de uma gestão mais simples da infra-estrutura e da acessibilidade oferecidas pela computação em nuvem por meio da avaliação dos custos de desempenho envolvidos. Esse artigo investigou o desempenho da computação em nuvem para Bioinformática através da medição do desempenho de aplicações de alinhamento em três diferentes ambientes de virtualização (o hipervisor KVM, o hipervisor Xen e o Linux Containers - LXC) em comparação a um servidor físico. Nos experimentos foram medidos os tempos de execução e uso de recursos computacionais dos ambientes a partir da execução de duas aplicações de Bioinformática para o alinhamento genômico, quais sejam, o Burrows-Wheeler Aligner (BWA) e o Novoalign.

Nesse trabalho foi escolhido o BWA por ser um pacote popular de código aberto e Novoalign por ser amplamente considerado na comunidade de Bioinformática como o programa de alinhamento mais preciso disponível. Do ponto de vista de sistemas de

computadores, essas aplicações são interessantes, pois levam a mesma entrada e trabalham para um objetivo similar, mas usando dois algoritmos diferentes. Os experimentos usaram uma cópia do mesmo conjunto de dados de entrada (chamado de chr1), para assegurar que as diferenças observadas não fossem causadas pela variação nos dados de entrada.

Os autores apontaram que quando aplicações de *benchmarking* que usam uma grande quantidade de dados, é importante estar atento como essas máquinas mantem em *cache* esses dados. Se um está usando imagens de máquina virtual que são camadas em cima de um sistema de arquivos existente (como poderia ser esperado em um ambiente em nuvem usando o hipervisor KVM), então tanto o sistema operacional VM quanto o sistema operacional *host* poderia armazenar dados em *cache*. Portanto, é importante limpar a *cache* do *buffer* entre todas as amostras de dados para garantir que cada experiência comece em um estado semelhante. O *cache* do *buffer* foi limpo antes de cada execução de cada *benchmark* ou aplicativo. Para caracterizar o desempenho de cada ambiente de virtualização os autores analisaram o desempenho de E/S de disco (devido à emulação de dispositivo) e o acesso à memória (devido ao gerenciamento de memória virtualizado). A ferramenta Bonnie ++ foi utilizada para caracterizar o desempenho do disco de cada plataforma.

Assim, os resultados obtidos demonstraram que as diferenças no desempenho de E/S de disco não foram estatisticamente significantes para leituras ou gravações sequenciais. Para aferição do desempenho da memória foi utilizado o STREAM com um tamanho de matriz de 40M elementos, ou um requisito de memória total de 0.9GiB. A CPU tem um tamanho de *cache* de 20MiB, que esse teste excedeu em quase duas ordens de grandeza. Os resultados da execução de STREAM 30 vezes (cada execução em si repetiu o benchmark 30 vezes e relatou o melhor resultado). Segundo as observações relatadas no artigo, embora não haja diferenças substanciais, a tendência parece ser consistente de que o servidor físico e LXC oferecem o mesmo desempenho, depois o KVM, seguido por Xen.

Nos testes de execução do BWA em todas as quatro plataformas, um alinhamento final emparelhado com BWA foi executado 30 vezes no conjunto de dados chr1 e foi obtida uma média do tempo de execução. Para este conjunto de dados, o aplicativo foi executado *single-threaded*. Dessa forma, foi verificado que o desempenho do LXC se aproxima do servidor físico, mas Xen e KVM têm sobrecarga significativa. Cada execução foi iniciada ao mesmo tempo, e o tempo de conclusão relatado é quando todos os trabalhos terminaram. Este cenário foi concebido como um "pior caso"uma vez que todas as VMs devem estar competindo pelos mesmos recursos ao mesmo tempo. Aqui, a tarefa mais lenta torna-se a tarefa crítica. Essa métrica foi escolhida porque em muitos fluxos de trabalho todas as tarefas precisam ser concluídas antes de passar para a próxima etapa (por exemplo, alinhar todos os cromossomos antes de passar para a variante de genoma

completo). Para a realização do experimento com a ferramenta Novoalign foi utilizada a mesma configuração utilizada pelo BWA. A única comparação do hipervisor que parece ser estatisticamente significante foi o Xen que é ligeiramente mais lento.

As conclusões relatadas em [37] são de que o desempenho para um determinado ambiente de virtualização depende da aplicação sendo executada, e que as alterações sutis num sistema (como por exemplo afinidade da CPU) podem ter um efeito drástico no tempo de conclusão do programa. Essa observação demonstra que o comportamento de um aplicativo em um ambiente não é totalmente indicativo de como outro aplicativo se comportaria no mesmo ambiente ou como esse mesmo aplicativo seria executado em um ambiente de virtualização diferente. Outra conclusão é de que aplicações que dependem mais do uso de CPU funcionam em velocidades próximas do *baremetal* em ambientes virtualizados, e que aparentemente as buscas em disco são dispendiosas nesses ambientes e de que ambientes configurados corretamente têm uma sobrecarga de aproximadamente 5 a 8% em todos os casos, não apenas nos ambientes virtualizados. Sobre os contêineres Linux foi concluído que têm baixa sobrecarga de desempenho e que, portanto, seria a melhor escolha entre as opções apresentadas no artigo.

O artigo [35] é outro exemplo do interesse da área de Bioinformática no uso de contêineres. Os autores indicam que os *pipelines* genômicos, em geral, são uma combinação de diversas partes de *softwares* de terceiros. Esses aplicativos tendem a ser protótipos acadêmicos que muitas vezes são difíceis de instalar, configurar e implantar. Um programa implementado em um determinado ambiente, normalmente, tem muitas dependências implícitas em programas, bibliotecas e outros componentes presentes nesse ambiente. Como conseqüência, um fluxo de trabalho computacional construído em um ambiente tem pouca chance de funcionar corretamente em outro ambiente sem esforço significativo. Os contêineres do Docker, na sua visão, aparecem como uma solução possível para muitos destes problemas, porque permitem o empacotamento dos *pipelines* em uma maneira isolada e autocontida por meio de imagens pré-construídas. Isso facilita a distribuição e a execução de *pipelines* de forma portátil em uma ampla gama de plataformas de computação.

Outra vantagem da utilização de contêineres Docker elencadas pelos autores é que cada processo em um contêiner seja executado de forma isolada, evitando assim conflitos com outros programas instalados no ambiente de hospedagem e garantindo que cada processo é executado em uma configuração de sistema previsível que não pode mudar ao longo do tempo devido a softwares mal configurados, atualizações do sistema ou erros de programação. Ainda é destacado que o benefício de rápida inicialização dos contêineres, bem como a vantagem de que muitas instâncias podem ser executadas no mesmo ambiente de hospedagem. O foco do artigo é de validar em que medida o uso de contêineres Docker pode afetar o desempenho de um fluxo de trabalho computacional quando comparado com

a execução nativa. No trabalho os autores avaliaram o impacto dos contêineres Docker sobre o desempenho de *pipelines* genômicos usando um cenário de uso de biologia computacional realista baseado no re-cálculo de subconjuntos selecionados da análise ENCODE - *Encyclopedia of DNA elements*, é um projeto de anotação genômica - em ratos.

Para avaliar o impacto do uso de Docker no desempenho de execução de ferramentas de Bioinformática, foram comparados três diferentes *pipelines* genômicos. Uma comparação dos tempos de execução foi feita executando os *pipelines* com e sem Docker junto com o mesmo conjunto de dados. Todos os três *pipelines* são desenvolvidos com Nextflow, uma ferramenta que é projetada para simplificar a implantação de *pipelines* computacionais em diferentes plataformas de forma reproduzível. O Nextflow fornece suporte integrado para Docker, permitindo que tarefas de *pipeline* sejam executadas de forma transparente em contêineres Docker. A vantagem dessa característica é que foi possível executar o mesmo *pipeline* nativamente ou executá-lo com Docker sem ter que modificar o código do *pipeline*, mas simplesmente especificando a imagem Docker a ser usada no arquivo de configuração Nextflow.

A sobrecarga introduzida pela tecnologia de contêineres no desempenho de *pipelines* foi estimada comparando o tempo de execução médio de 10 instâncias executando com e sem Docker. Como o *pipeline* executou tarefas paralelas, o tempo de execução foi normalizado somando o tempo de execução de todas as tarefas em cada instância. A primeira avaliação de desempenho foi realizada usando um *pipeline* simples para análise de dados RNA-Seq. O tempo médio de execução do *pipeline* no ambiente nativo foi de 1.156,9 min (19 h 16 m 55 s), enquanto que o tempo médio de execução quando executado com Docker foi de 1,158.2 min (19 h 18 m 14 s).

Assim, a execução dos contêineres introduziu uma sobrecarga virtualmente zero de 0,1%. O segundo *benchmark* foi executado usando um *pipeline* de chamada de variante baseada em montagem, parte de um fluxo de trabalho de genotipagem compatível para aplicações de histocompatibilidade, imunogenética e imunogenômica. O tempo médio de execução do *pipeline* no ambiente nativo foi de 1.254,0 minutos (20 horas 53 minutos 58 segundos), enquanto o tempo médio de execução quando executado com Docker foi de 1,283.8 minutos (21 horas 23 minutos 50 segundos). Isto significa que ao executar com Docker a execução foi abrandada por 2.4%.

O último experimento foi realizado usando Piper-NF, um *pipeline* genômico para a detecção e mapeamento de RNAs longos não codificantes. O tempo médio de execução do *pipeline* no ambiente nativo foi de 58,5 minutos, enquanto o tempo médio de execução ao executá-lo com Docker foi de 96,5 minutos. Nessa experiência, a execução com Docker introduziu um abrandamento significativo do tempo de execução do *pipeline*, cerca de 65%. Esse resultado pode ser explicado pelo fato de que o *pipeline* executou muitas

tarefas de curta duração - o tempo médio de execução da tarefa foi de 35,8 segundos e o tempo de execução médio foi de 5,5 segundos. Assim, a sobrecarga adicionada pelo Docker para inicializar o ambiente de contêiner e montar o sistema de arquivos do *host* tornou-se significativo quando comparado à curta duração da tarefa.

Os resultados dos experimentos realizados em [35] mostram que a utilização da plataforma de contêineres Docker tem uma sobrecarga negligenciável no desempenho do *pipeline* quando é composta de tarefas de execução de médio e de longo prazo, que é o cenário mais comum em *pipelines* genômicas computacionais. Foi destacado pelo autores que para essas tarefas o desvio padrão observado é menor quando executado com Docker, sugerindo que a execução com contêineres é mais "homogênea", presumivelmente devido ao isolamento proporcionado pelo ambiente do contêiner. A degradação do desempenho é mais significativa para as execuções onde a maioria das tarefas tem uma granularidade fina ou muito fina (alguns segundos ou milissegundos).

Nesse caso, o tempo de instanciação do contêiner, embora pequeno, não pode ser ignorado e produz uma perda perceptível de desempenho. Foi levantado também que existiam à época da escrita do artigo algumas limitações e potenciais problemas de segurança no Docker que devem ser levados em consideração. Apesar do Docker tirar proveito da habilidade do *kernel* do Linux de criar ambientes isolados nos quais cada contêiner recebe sua própria pilha de rede, espaço de processo e sistema de arquivos, essa separação não é tão forte quanto a das máquinas virtuais que administram um SO independente em cima da camada de virtualização em nível de hardware. Por fim, os autores concluíram com base nas evidências coletadas que a perda de desempenho mínima introduzida pelo mecanismo Docker é compensada pelas vantagens de executar uma análise em um ambiente de tempo de execução autônomo e controlado com precisão.

Capítulo 5

Avaliação de Desempenho de Contêineres Docker

5.1 Desenvolvimento de Imagens Docker para Aplicação do STF

Como premissa para realização de avaliação de desempenho foi definido que seria utilizada uma aplicação desenvolvida pela Secretaria de Tecnologia da Informação do Supremo Tribunal Federal. Para isso, foi considerado que para validar nossa análise seria necessário que essa aplicação fosse relevante nos processos de trabalho do Tribunal. Assim sendo, foi escolhida a aplicação "Supremo Autuação", na qual é realizado o procedimento de autuação de um processo judicial. A autuação de um processo judicial é o procedimento no qual é realizado o registro de informações básicas de uma ação judicial, tais como classe processual, número, nome e categoria das partes, informações de representante judicial (advogados ou procuradores), origem, confidencialidade, assunto e preferências definidas em lei (maior de 65 anos, réu preso, etc). Assim, após finalizar esse procedimento o processo autuado é distribuído entre os Ministros da Corte seguindo as regras estabelecidas no Regimento Interno do STF.

Para a criação da imagem Docker da aplicação foi realizada uma análise de quais softwares (sistema operacional, servidor de aplicação e aplicação) estão envolvidos na operação dos serviços e de como se relacionam. A partir dessa análise foi possível identificar a divisão mais apropriada dos elementos que compõem as imagens.

Conforme visto na Seção 3.2.2, o Docker pode construir imagens automaticamente lendo as instruções de um *Dockerfile*. A composição desse arquivo, baseada nas instruções e na ordem em que são executadas, determinam diversos aspectos da imagem final e, consequentemente, dos contêineres instanciados a partir dela. Alguns aspectos relevantes

são o tamanho da imagem, a complexidade, os processos executados e suas dependências.

Baseado na característica de montagem do sistema de arquivos em camadas (*layers*) da tecnologia de contêineres, optou-se pela divisão das imagens pela instalação e configuração dos componentes, bem como pela sua frequência de atualização. Essa abordagem permite as seguintes vantagens:

- Menor tempo de construção ou de reconstrução das imagens, uma vez que estas atividades que envolvem imagens logicamente em um nível superior não demandam o mesmo processo das imagens abaixo delas;
- Maior estabilidade das aplicações, pois mudanças de aspectos próprios da mesma, tais como bibliotecas ou configurações específicas de ambientes, afetam somente as imagens a ela relacionadas;
- Padronização de ambiente, devido a inclusão de softwares e configurações gerais em imagens hierarquicamente inferiores, o que permite a propagação destes elementos a outras imagens que forem construídas a partir destas; e
- Maior aproveitamento das camadas (*layers*) já construídas, favorecendo a utilização do cache e minimizando o crescimento de utilização do espaço de armazenamento.

Dessa forma, optou-se pelo armazenamento dos binários dos *softwares* a serem instalados nas imagem em um servidor web Apache para *download*, instalação e exclusão em uma mesma instrução no *Dockerfile* ao invés de se utilizar uma das instruções de cópia (ADD ou COPY). Essa prática diminui o tamanho da camada adicionada à imagem final.

A imagem base selecionada para a arquitetura foi a do sistema operacional CentOS [8] na sua versão 6, por ser sistema análogo ao Red Hat Enterprise Linux versão 6, utilizado nas máquinas servidoras dos ambientes tradicionais do STF. Assim, três níveis foram definidos a partir da imagem base do sistema operacional: Softwares e configurações de uso geral; Servidor de Aplicações e Aplicação conforme demonstrado na Figura 5.1. Cada uma dessas camadas e sua construção está detalhada abaixo.

5.1.1 Imagem de Softwares e Configurações de Uso Geral

A estratégia de imagem mínima em imagens-base de sistemas operacionais indica que se mantenha por padrão, em imagens dessa natureza, somente bibliotecas estritamente necessárias para o funcionamento do sistema. O uso dessa estratégia visa diminuir ao máximo o tamanho da imagem do sistema operacional e, consequentemente, o tamanho das imagens construídas a partir dela. Por conta da abordagem descrita, nesse nível da arquitetura de imagens, é realizada a instalação das bibliotecas adicionais do sistema operacional.

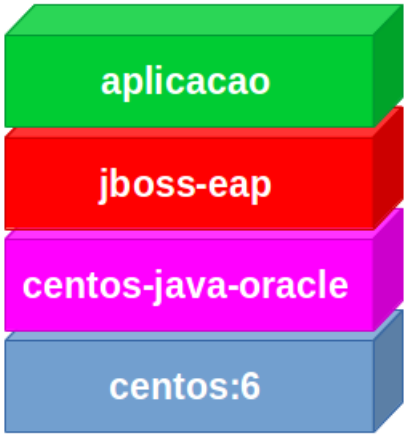

Figura 5.1: Visão conceitual da Arquitetura das Imagens.

Nessa camada foram instaladas as bibliotecas *glibc-common (Common binaries and locale data for glibc)* para suporte do sistema ao fuso horário brasileiro e *libaio (Linux-native asynchronous I/O access library)* necessária para o cliente do banco de dados. Outros softwares de uso geral foram adicionados, os quais são: *Unzip* e *Tar* para manipulação de arquivos compactados; *cURL* que é uma ferramenta para criar requisições em diversos protocolos (nesse caso em particular o HTTP) e obter conteúdo remoto; *passwd*, utilitário para definição de senhas; e o *sudo* que possibilita acessos restritos de super-usuário a usuários específicos.

Nesse cenário, optou-se também pela inclusão de softwares não diretamente ligados ao sistema operacional, mas que são necessários para o funcionamento das aplicações desenvolvidas pelo STF. Um desses softwares é o cliente de banco de dados Oracle.

Essa imagem terá como base a imagem do sistema operacional (centos:6) disponível no *Docker Hub*.

O *Dockerfile* desenvolvido está presente no Apêndice A.1. O comando para criação da imagem é apresentado no Quadro 5.1.

Quadro 5.1: Comando para Criação de Imagem da Camada de Softwares e Configurações de Uso Geral (*centos-java-oracle*).

```
docker build -t registro/centos-java-oracle .
```

5.1.2 Imagem de Servidor de Aplicações

A aplicação permanecerá sendo executada por meio do servidor de aplicação já em uso no ambiente do STF. Portanto, no presente nível da arquitetura das imagens será instalado o servidor de aplicação *Red Hat Jboss Enterprise Application Plataform* (JBoss EAP) [28] na sua versão 6.4. Em contraposição à arquitetura tradicional do JBoss no STF na qual se faz uso de um Domínio Gerenciado (*Managed Domain*), que fornece gerenciamento centralizado de várias instâncias de aplicação e *hosts* físicos, foi utilizado o servidor autônomo (*Standalone Server*) que permite uma única instância do servidor de aplicação.

A instalação do JBoss é efetuada a partir do arquivo de instalação disponibilizado pela Oracle em seu sítio [27]. São definidas nesse momento as configurações para o funcionamento do JBoss: a criação da váriável de ambiente *JBOSS_HOME*, com a indicação da pasta na qual o servidor de aplicação foi instalado; e a inclusão dessa variável no *PATH* para que o sistema operacional possa localizar os executáveis necessários para operação do JBoss. Para o funcionamento das aplicações, também são instalados nessa imagem os módulos requeridos por elas. Assim, nessa etapa será criado um usuário específico para execução dos contêineres provenientes da arquitetura de imagens em questão. Esta é uma boa prática [2] indicada para processos que não necessitam de usuários privilegiados, o que reforça a proteção do ambiente numa eventual falha de segurança, que possa ser explorada por usuários mal intencionados.

Serão expostas na imagem as portas 8080 - para receber as requisições HTTP da aplicação - e 9990, que é utilizada para se ter acesso a console de gerenciamento do servidor de aplicações. Essas portas serão mapeadas para portas da máquina hospedeira do Docker. A inicialização da instância de aplicação no modo *standalone server* do JBoss é feita por meio do *script* ”standalone.sh” localizado na pasta ”JBOSS_HOME/bin”. Será definido nessa imagem o *Entrypoint* com esse comando para inicialização do JBoss. Essa imagem terá como base a imagem de softwares e configurações de uso geral (registro/centos-java-oracle) descrita na Seção 5.1.1. O *Dockerfile* desenvolvido está presente no Apêndice A.2. O comando para criação da imagem é apresentado no Quadro 5.2.

Quadro 5.2: Comando para Criação de Imagem da Camada do Servidor de Aplicações (*jboss-eap:6.4*)

```
docker build -t registro/jboss-eap:6.4 .
```

5.1.3 Imagem de Aplicação

A imagem da aplicação ficará restrita à cópia do artefato de implantação da aplicação (arquivo *war*). Essa abordagem permite um reaproveitamento das imagens (e consequentemente das camadas) na construção das imagens Docker de outras aplicações. Essa imagem terá como base a imagem de Servidor de Aplicações (registro/jboss-eap:6.4) descrita na Seção 5.1.2.

O *Dockerfile* desenvolvido está presente no Apêndice A.3. O comando para criação da imagem é apresentado no Quadro 5.3.

Quadro 5.3: Comando para criação de imagem da camada do Servidor de Aplicações (*jboss-eap:6.4*)

```
docker build -t registro/supremo-autuacao .
```

5.2 Metodologia de Avaliação

No presente trabalho foram avaliados os efeitos de sobrecarga do uso da tecnologia de contêineres Docker na execução de cargas de trabalho da aplicação "Supremo Autuação" do Supremo Tribunal Federal. Foram realizadas duas avaliações (Seções 5.4 e 5.5) do uso de recursos computacionais (CPU e memória) em relação ao desempenho nativo (instalação *Baremetal*) e ao ambiente de virtualização com o hipervisor Hyper-V (referido a partir daqui somente como máquinas virtuais ou VMs). A metodologia utilizada foi inspirada nos artigos presentes no Capítulo 4 desse trabalho.

O teste de desempenho de uma aplicação web tem por objetivo avaliar o comportamento de um sistema submetido a uma determinada carga de trabalho. Um dos motivos para a realização desses testes é comparar características de desempenho de vários sistemas ou configurações de um sistema. A carga de trabalho (*workload*) é o estímulo aplicado a um sistema, aplicação ou componente para simular um padrão de uso, em relação à concorrência e/ou entradas de dados. A carga de trabalho inclui o número total de usuários ativos, usuários ativos simultâneos ou concorrentes, volume de dados e volume de transações [41].

A funcionalidade do sistema Supremo Autuação executada para realização dos testes é a autuação de um processo judicial. A execução dessa funcionalidade será repetida continuamente durante o intervalo de 10 minutos. Para a automatização dessa tarefa foi utilizado o aplicativo Apache JMeter [1] na versão 3.0. O Apache JMeter é uma ferramenta

popular desenvolvida na linguagem Java para avaliação de desempenho de aplicações web. O JMeter só funciona no nível de protocolo e não renderiza imagens como um navegador comum.

Concomitante à execução da aplicação foi disparado um *script* desenvolvido em Python, que utiliza a biblioteca de *benchmark* psutil, já discutida no Capítulo 4, para aferição das informações de uso de CPU e de memória. Na avaliação feita, optou-se por não avaliar a E/S de disco por conta do perfil da aplicação, na qual o uso desse recurso não é significativo, pois se trata de uma aplicação em que o armazenamento das informações é feito exclusivamente em banco de dados. Além disso, a utilização de rede não foi incluída no escopo deste trabalho.

A métrica de uso de CPU capturada foi o percentual de uso de CPU (*psutil.cpu_percent*) e para memória foi a quantidade de memória total utilizada (*psutil.virtual_memory*). O *script* de *benchmark* coleta os dados a cada 1 segundo na máquina hospedeira do ambiente em que estava sendo executado o teste (*Baremetal*, VM e Docker). Neste trabalho foi escolhida essa forma de captura de dados devido ao uso de sistemas operacionais diferentes nas máquinas hospedeiras (nos ambientes *Baremetal* e Docker é utilizado o Linux CentOS 7 e na virtualização com Hyper-V é utilizado o Windows Server 2012 R2). Como se trata de uma biblioteca multi-plataforma ela possibilita uma uniformidade na obtenção das métricas a serem analisadas. O código do *script* de *benchmark* é apresentado em detalhes no Apêndice B na Seção B.1.

Cada medição individual foi replicada 10 vezes de forma independente, gerando assim 6000 amostras para cada cenário de teste. Nas amostras obtidas aplicou-se a média aritmética simples e foram calculados intervalos de confiança com nível de confiabilidade de 95%.

5.3 Ambiente de Avaliação

Os experimentos foram realizados em máquinas conforme especificação de *hardware* listada na Tabela 5.1. As plataformas e versões de *software* utilizadas estão listadas na Tabela 5.2.

Tabela 5.1: Especificações Técnicas de Hardware para Realização dos Experimentos.

Recurso	Descrição	Quantidade
Servidor	IBM BladeCenter HS23	02

Processador	Intel Xeon ES-2620 64 bits 2.00 GHz (6 cores, 12 threads, Cache L1 64KB, Cache L2 256KB, Cache L3 15MB)	02 por servidor
Memória	DDR III SDRAM	24 GB por servidor
Disco	Western Digital SAS 10k	300GB por servidor
Rede	Broadcom	1 Gbps por servidor

Tabela 5.2: Plataformas e Versões de Software Utilizadas nos Experimentos.

Plataforma	**Versão**
Linux	CentOS Linux release 7.1.1503
Hyper-V	2012 R2
Docker	1.12.3

Para a criação de máquinas virtuais é necessário definir sua configuração no software de gerenciamento do Hyper-V. A configuração das máquinas virtuais está demonstrada na Tabela 5.3.

Tabela 5.3: Configuração das Máquinas Virtuais no Ambiente de Virtualização Hyper-V.

Recurso	**Quantidade**
vCPU	02 (2GHz com 02 cores cada)
Memória	5 GB
Disco	50 GB Virtual Disk VHDX
Rede	Virtual Switch
Sistema Operacional	CentOS Linux release 7.1.1503

O Docker será executado diretamente no sistema operacional do *host* físico. Embora o Docker permita a definição de limites de uso de CPU e de memória por meio do *cgroups*, não será definida nenhuma limitação para esse ambiente.

A aplicação Supremo Autuação é executada a partir do servidor de aplicações *Red Hat Jboss Enterprise Application Plataform* (JBoss EAP) versão 6.4 com modo de servidor autônomo (*Standalone Server*), que permite uma única instância do servidor de aplicação. Para o ambiente *Baremetal* e VM será realizada a instalação dos binários do JBoss no SO,

enquanto que para o Docker será utilizada a imagem da aplicação conforme descrita na Seção 5.1. O limite máximo de memória a ser utilizado para a aplicação nos três ambientes será definido pelas configurações da *Java Virtual Machine* (JVM) listada na Tabela 5.4. Essas configurações correspondem aos valores utilizados pela aplicação Supremo Autuação no ambiente de Produção do STF.

Tabela 5.4: Configuração de Memória da *Java Virtual Machine* para execução da aplicação Supremo Autuação.

Parâmetro	**Valor**
Heap Size	512 MB
Max Heap Size	4096 MB
Permgen Size	512 MB
Max Permgen Size	512 MB

5.4 Avaliação de Sobrecarga em Relação ao Ambiente de Provisionamento da Aplicação

O objetivo desta avaliação é identificar a sobrecarga imposta ao sistema pelo uso do Docker como ambiente para o provisionamento de aplicações em comparação com os ambientes nativo (*Baremetal*) e virtualizado com hipervisor (VM). Como esclarecido por Felter [38], essa avaliação de desempenho é relevante pois uma pior utilização reduz os recursos disponíveis para o trabalho produtivo.

Segundo a classificação definida por Meier [41], o Teste de Carga (*Load Test*) é um subtipo dos testes de performance, que tem por objetivo verificar o comportamento da aplicação em condições de carga normal e de pico. Assim sendo, para a realização do experimento foi necessária a definição de uma carga normal e do pico de utilização baseado no uso da aplicação Supremo Autuação.

Para a definição de uma carga de trabalho são necessárias três informações: o cenário de uso da aplicação (qual funcionalidade será testada); a frequência de execução (quantas vezes será executada a aplicação); e a quantidade de usuários concorrentes (quantos usuários estarão executando a aplicação ao mesmo tempo). As duas primeiras informações foram definidas na Seção 5.2. Para definir o número de usuários concorrentes para as cargas normal e o pico de utilização foi utilizada a fórmula apresentada no Quadro 5.4.

Quadro 5.4: Fórmula para Determinar a Quantidade de Usuários Concorrentes para Definição de Carga de Trabalho [3].

```
NUC = (NSH x DMS / 3.600) / TU

NUC: Número de Usuários Concorrentes
NSH: Número de Sessões por Hora
DMS: Duração Média da Sessão (em segundos)
TU: Tempo de Utilização (em horas)
```

Para obter os dados necessários para o cálculo de usuários concorrentes foram utilizadas as estatísticas da aplicação coletadas por meio da ferramenta Google Analytics [15]. O Google Analytics é um sistema gratuito de monitoramento de tráfego para sites e aplicações web. Essa ferramenta capta diversas estatísticas sobre o uso da aplicaçao, tais como quantidade e frequência de acessos, comportamento de navegação, dados geográficos e páginas mais acessadas.

As informações são exibidas por meio de relatórios e painéis configuráveis pelo usuário. Foi avaliado um período de três meses em que houvessem mais dias úteis, o que em consequência significaria uma maior utilização da aplicação Supremo Autuação. Assim, definiu-se o período entre os meses de Setembro e Novembro de 2016, o que totalizou 52 dias úteis com 7 horas de trabalho (carga horária diária do STF). As estatísticas coletadas pelo Google Analytics para esse período podem ser vistas na Figura 5.2.

Ao aplicar a fórmula para calcular a quantidade de usuários concorrentes, conforme demonstrada no Quadro 5.5, foi atingido o número de 25,8.

Quadro 5.5: Cálculo do Número de Usuários Concorrentes da Aplicação Supremo Autuação Baseado nas Estatísticas do Período.

```
NUC = (47033 x 719 / 3.600) / (52 x 7)
    = 25,8 Usuários Concorrentes
```

Dessa forma, foi definido que para a realização dos experimentos a quantidade de usuários concorrentes para a carga normal seria de 26. Considerando que a carga normal já se trata de um período de intensa utilização da aplicação, definiu-se um valor hipotético

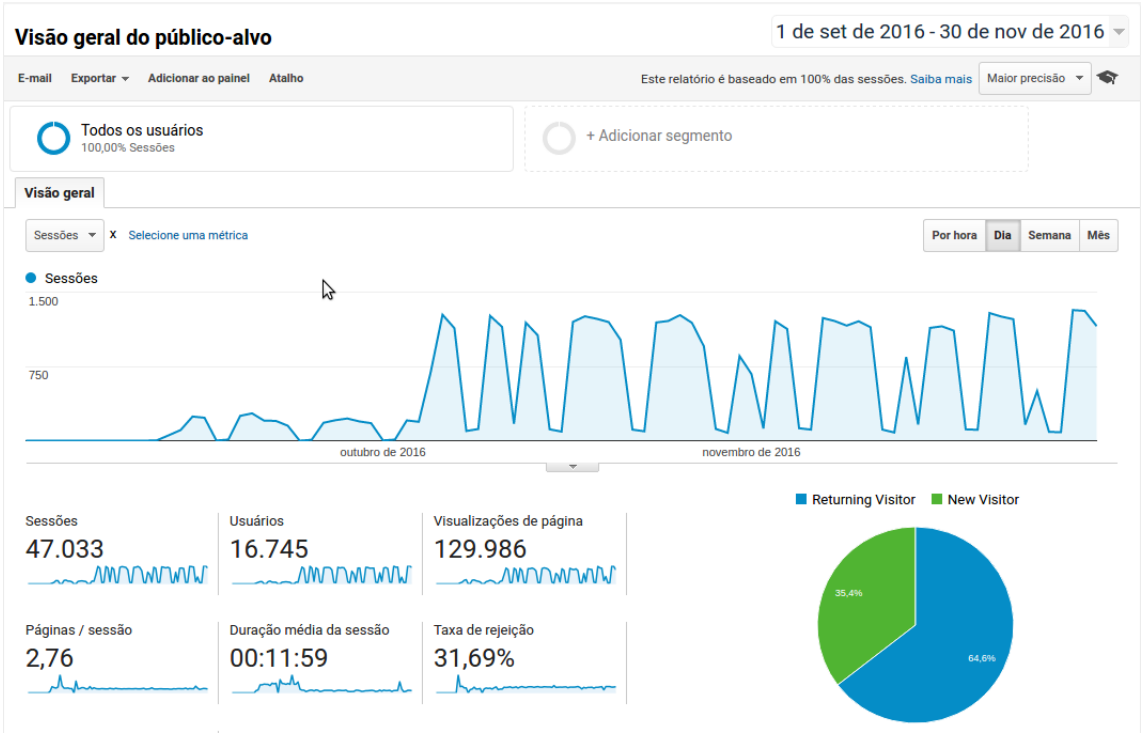

Figura 5.2: Estatísticas de Uso da Aplicação Supremo Autuação no Período.

para o pico de utilização para efeito deste estudo. Logo, estipulou-se que o pico de utilização seria o dobro da carga normal, portanto, o número de 52 usuários concorrentes.

Definidas as cargas de trabalho, foram definidos dois cenários para a realização do experimento. No primeiro cenário foram colhidas as informações de CPU e de memória para cada um dos ambientes em que a aplicação foi provisionada (*Baremetal*, VM e Docker), com uma carga de trabalho normal. No segundo cenário foi realizado o mesmo procedimento com a carga de trabalho de pico de utilização. Ao final foi comparado o uso de recursos médio pelos diferentes ambientes em relação a carga de trabalho aplicada.

Conforme esperado, é possível observar na Figura 5.3 que o uso de CPU do ambiente *Baremetal* e Docker ficaram mais próximos enquanto que o uso no ambiente de VM foi muito superior. Destaca-se nos resultados obtidos, um melhor uso de CPU do Docker em relação ao *Baremetal* no cenário de carga normal, credita-se esse comportamento possivelmente ao *cgroups*. O Docker usa *cgroups* para agrupar processos em execução no contêiner. Esta abordagem permite flexibilidade no gerenciamento dos recursos, já que pode se gerenciar cada grupo individualmente. Em um sistema operacional que usa *systemd* como gerenciador de serviços, que é o caso do Linux CentOS 7, todos os processos (e não apenas os que estão dentro de contêineres) serão associados a um *slice* específico dentro da hierarquia *cgroups*. O *slice* é a estrutura interna do *cgroups* que permite que os recursos sejam restritos ou atribuídos a qualquer processo associado ao

slice [20]. Os limites para o *slice* são aplicados somente quando necessário. O *cgroups* não limita os processos imediatamente (não permitindo que eles sejam executados mesmo se houver recursos livres). Em vez disso, libera o máximo de recursos que pode e limita apenas quando necessário (por exemplo, quando muitos processos começam a usar a CPU fortemente ao mesmo tempo) [29].

No ambiente *Baremetal*, quando o JBoss é executado os seus processos são incluídos em um *slice* para processos de usuário geral (*user.slice*). Por outro lado, com o Docker os contâineres têm seus processos associados ao *slice* Docker. Infere-se que a alocação de recursos para o *user.slice* tem um limite maior do que o *slice* Docker, permitindo assim, um maior uso de CPU pelo ambiente *Baremetal* em relação ao Docker. Não é simples dizer o quanto dos recursos será atribuído ao processo, pois depende realmente de como os outros processos pertencentes ao *slice* se comportarão e de quanto dos recursos compartilhados serão atribuídas a eles [29]. O comportamento no cenário de pico de utilização corrobora essa explicação, na medida em que, com o incremento da carga é observada uma maior sobrecarga aplicada ao sistema pelo uso do Docker.

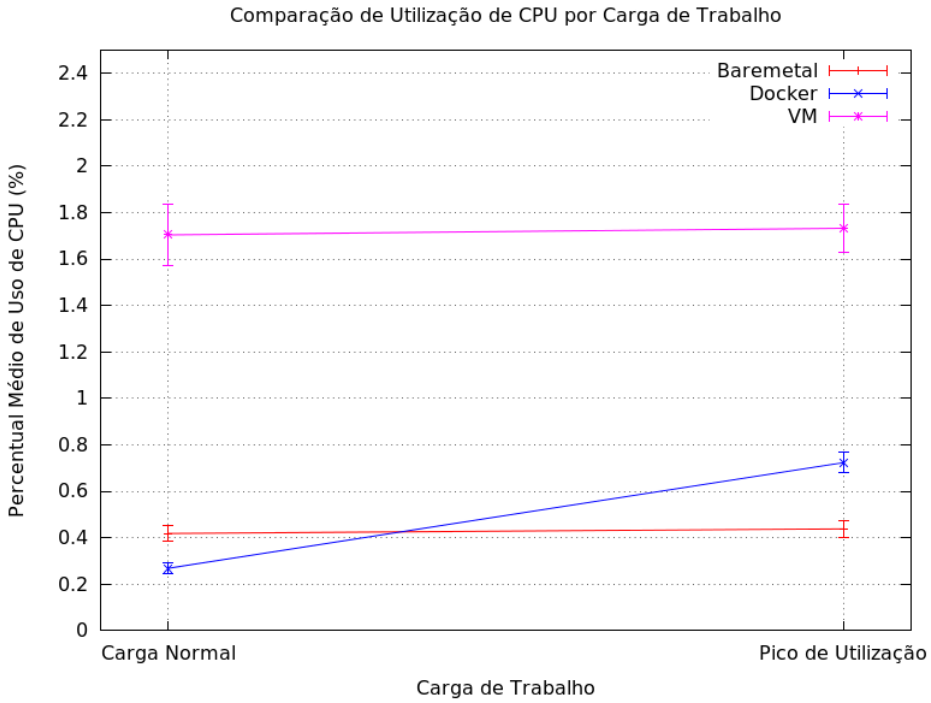

Figura 5.3: Comparação de Uso de CPU em Relação à Carga de Trabalho Aplicada.

Com relação ao desempenho no uso da memória (veja a Figura 5.4) verificou-se um comportamento similar ao do uso de CPU. Porém, nesse caso, tem-se uma proximidade maior dos valores entre os ambientes Docker e *Baremetal*, sobretudo no pico de utilização. Em relação ao ambiente de VM, a utilização de memória pelo Docker foi 3,5 vezes menor.

Em ambos os cenários o maior uso de recursos pela VM demonstra o maior custo imposto pela utilização do hipervisor nesse ambiente.

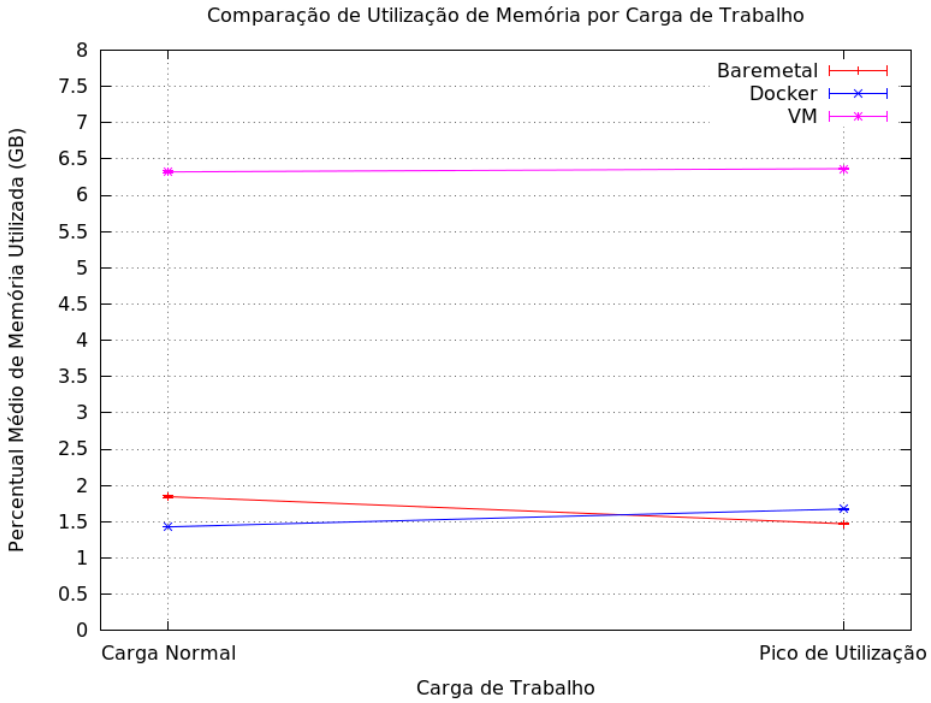

Figura 5.4: Comparação de Uso de Memória em Relação à Carga de Trabalho Aplicada.

Com relação à sobrecarga individual entre os cenários (normal e pico de utilização), com exceção do uso de CPU para o Docker, observou-se que o uso de recursos permaneceu constante. Esse comportamento foi atribuído a uma subestimação do pico de utilização da aplicação, o que não permitiu definir qual o seu perfil de uso, se é *CPU bound* (uso intensivo da CPU) ou *Memory bound* (maior uso da memória).

5.5 Avaliação de Uso de Recursos em Função do Número de Unidades Computacionais

A presente avaliação procura identificar o comportamento do Docker em oposição ao ambiente de VM, a medida que se adiciona mais unidades computacionais à máquina hospedeira. É chamada de unidade computacional cada contêiner Docker ou máquina virtual em execução no *host*. Assim, fixou-se para o presente experimento a carga de trabalho como normal, conforme explicado na Seção 5.4. Para isso, foi aplicada uma carga de trabalho completa em cada ambiente para o número de Unidades Computacionais de um a cinco. Esse número máximo de Unidades Computacionais foi definido em função da divisão da memória total do *host* hospedeiro (24GB)pela quantidade máxima de memória

que pode ser alocada pela aplicação (4GB), chegando ao resultado de 6. Porém, como é preciso considerar que existe uso de memória por outros processos, tanto no hospedeiro quanto nas máquinas virtuais, limitou-se a cinco o número máximo de unidades computacionais por *host*. Além disso, foi realizada a coleta de dados de CPU e de memória para cada uma das execuções no hospedeiro. Finalizadas as mensurações, passou-se à análise do padrão de uso dos recursos computacionais para cada ambiente. Os dados relacionados ao uso de CPU são apresentados na Figura 5.5.

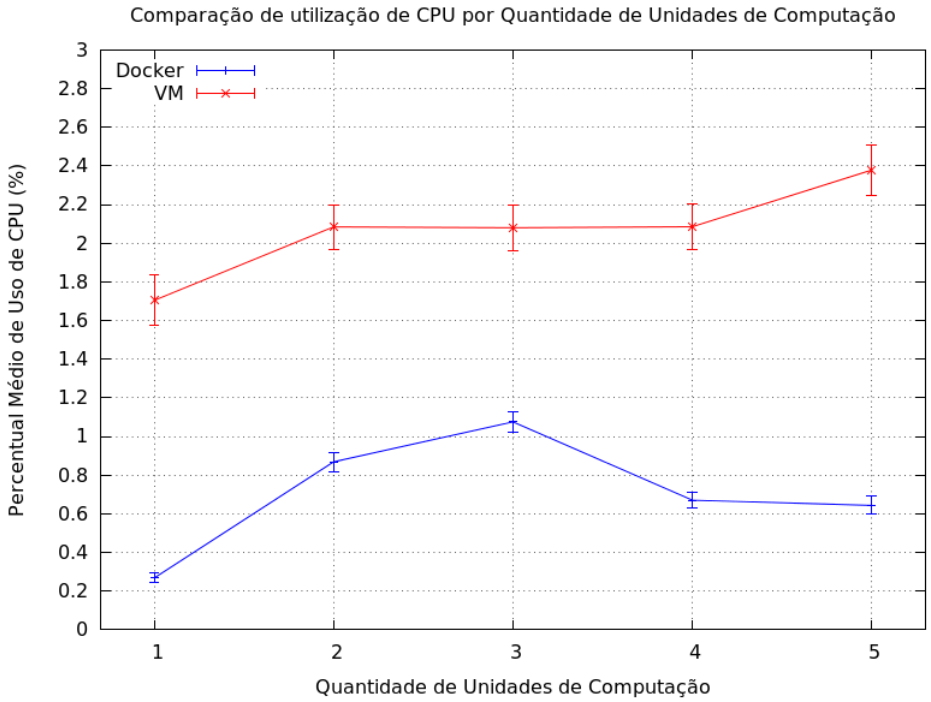

Figura 5.5: Comparação de Utilização Média da CPU em Relação a Quantidade de Unidades de Computação Alocadas no *Host* Hospedeiro.

Na Figura 5.5 é possível observar que o uso de CPU do ambiente Docker é substancialmente menor em relação ao ambiente de VM. Esse comportamento confirma o que é demonstrado na literatura (Capítulo 4). Mais uma vez atribui-se esse *overhead* ao uso do hipervisor para emulação de hardware, o que torna necessário um uso mais intenso da CPU do *host* hospedeiro. Curiosamente, o gráfico do ambiente Docker tem uma rápida ascensão com uso de até três contêineres, para em seguida diminuir com quatro e cinco unidades. O motivo que pode explicar essa diminuição é a forma como o Docker distribui o uso de CPU entre os contêineres em execução. Cada novo contêiner recebe 1024 partes de CPU por padrão. É possível entender essas partes como um valor variável utilizado para aumentar ou reduzir o peso do contêiner, e dar acesso a uma maior ou menor proporção dos ciclos de CPU da máquina host. Isso só é imposto quando os ciclos de CPU são limitados. Quando há muitos ciclos de CPU disponíveis, todos os contêineres usam o

máximo de CPU que precisam. Um aspecto interessante é que os tempos ociosos de um contêiner podem ser utilizados por outro contêiner que precisar [29]. Sendo assim, para a carga de trabalho aplicada, a partir de três contêineres, o mecanismo do Docker passou a otimizar o desempenho do uso da CPU substituindo os tempos ociosos por processamento efetivo de outros contêineres.

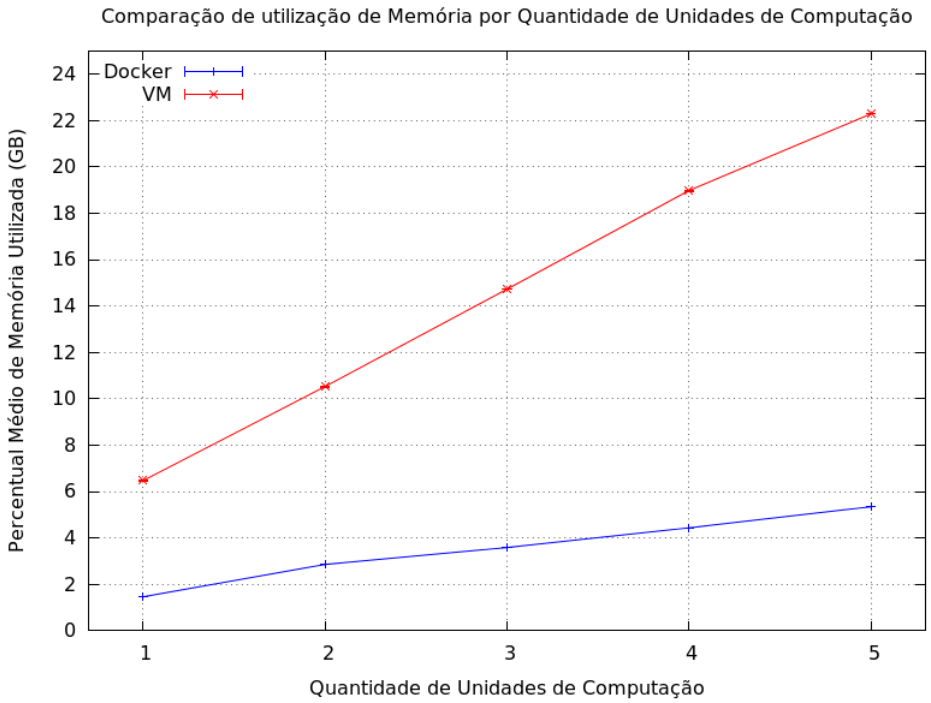

Figura 5.6: Comparação de Utilização Média de Memória em Relação a Quantidade de Unidades de Computação Alocadas no *Host* Hospedeiro.

.

Os resultados relativos à memória (veja a Figura 5.6) demonstraram que no ambiente em que foram utilizadas máquinas virtuais o consumo desse recurso foi muito maior. À medida em que foram sendo adicionadas outras VMs, a alocação de memória não só manteve o patamar anterior como aumentou, causando, no seu pico de utilização, a ocupação de 90,67% da memória física do *host* hospedeiro. Enquanto que para o ambiente com Docker chegou a 21,75% da memória total do *host*. Diante disso, percebe-se a vantagem no provisionamento da aplicação utilizando contêineres ao invés de máquinas virtuais, que consomem mais memória em função dos outros processos que rodam no sistema operacional convidado.

Além disso, foi observado o tempo de resposta da aplicação Supremo Autuação com relação ao seu tempo de resposta durante o experimento, pois essa métrica é a mais sensível sob o ponto de vista do usuário final. Conforme pode ser observado nas Figuras

5.7 e 5.8, o tempo de resposta médio foi aproximadamente o mesmo no ambiente Docker e no provisionamento com uso de máquinas virtuais (VM).

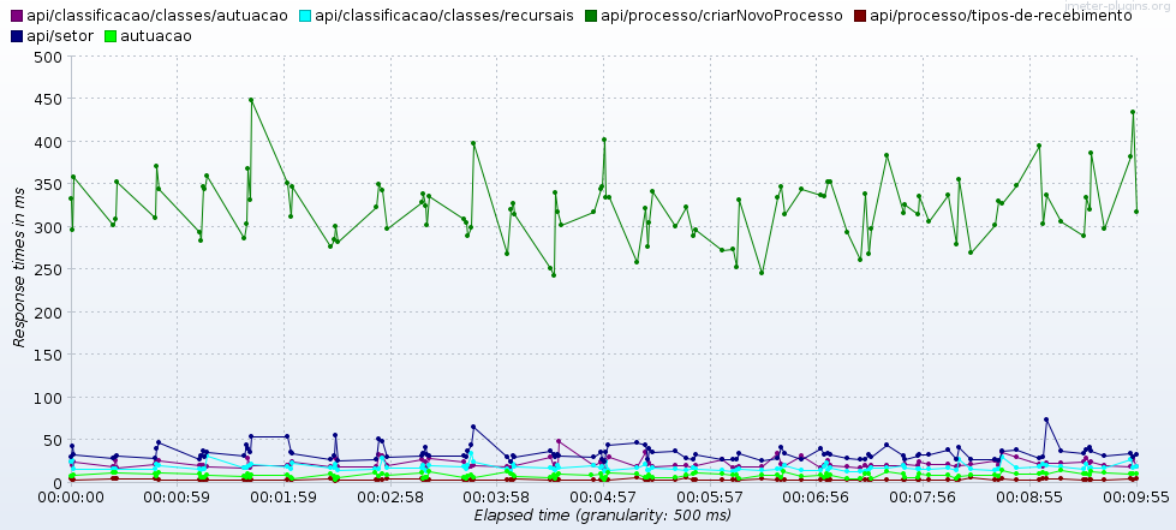

Figura 5.7: Tempo de Resposta Médio da Aplicação Supremo Autuação no Ambiente Docker.

.

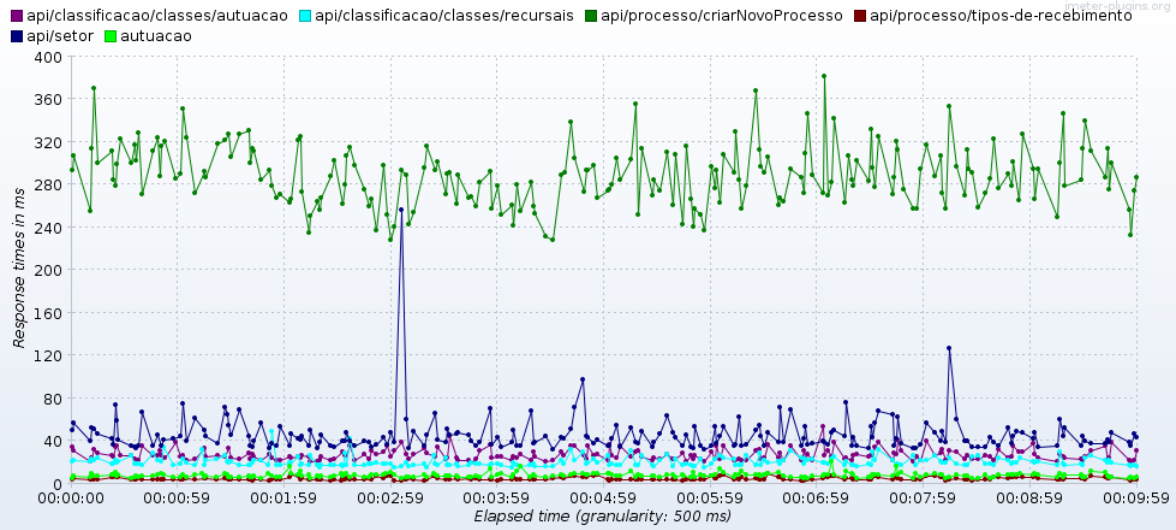

Figura 5.8: Tempo de Resposta Médio da Aplicação Supremo Autuação no Ambiente VM.

.

5.6 Considerações Finais

Nesse capítulo foram apresentados os testes realizados para avaliação de desempenho de contêineres Docker para provisionamento da aplicação Supremo Autuação do STF. Para

essa avaliação foi utilizada uma imagem Docker desenvolvida neste trabalho.

Assim, foram propostos dois objetivos para os testes, o primeiro era provar que a plataforma Docker utilizaria melhor os uso de recursos computacionais (CPU e memória) em comparação ao ambiente virtualizado com hipervisor (VM), e o segundo objetivava determinar que a performance da aplicação Supremo Autuação não sofreria perdas significativas de performance sob a ótica do usuário final.

Portanto, diante dos resultados apresentados, os dois objetivos foram atingidos neste trabalho, pois foi demonstrado que o desempenho do ambiente com utilização de contêineres Docker foi superior ao ambiente virtualizado com hipervisor (VM) e muito próximo ao ambiente nativo (*Baremetal*). Também foi comprovado que, o tempo de resposta da aplicação foi similar, tanto com o uso do Docker como com o uso de VMs, o que demonstra, portanto, não haver deterioração de performance da aplicação com o uso dessa plataforma de contêineres.

Capítulo 6

Conclusão

Assim, este trabalho se propôs a realizar uma análise de desempenho dos contêineres Docker como ambiente de provisionamento para aplicações do Supremo Tribunal Federal, dado que a utilização de contêineres oferece simplicidade, agilidade e estabilidade aos processos de desenvolvimento, de construção e de entrega de aplicações.

Inicialmente neste estudo, preocupou-se em conhecer mais a fundo do que se trata a tecnologia de contêineres, seu funcionamento e sua evolução. As principais diferenças entre os contêineres e máquinas virtuais, e o uso da tecnologia de contêineres na computação em nuvem foram explorados. Em seguida, foi aprofundado o estudo no ecossistema do Docker, detalhando sua base tecnológica e arquitetura. Todos esses conhecimentos apreendidos, juntamente com as informações obtidas nos artigos da revisão bibliográfica efetuada, compuseram o embasamento científico e metodológico para a realização da avaliação de desempenho subsequente.

Os experimentos fizeram uso de contêineres instanciados a partir da imagem Docker da Aplicação Supremo Autuação. O referido sistema é utilizado para o cadastramento inicial de um processo judicial no STF. Diante disso, foram investigados efeitos de sobrecarga do uso da tecnologia de contêineres Docker em relação ao desempenho nativo (instalação *Baremetal*), e o ambiente de virtualização com o hipervisor Hyper-V. As métricas utilizadas foram de uso de CPU e memória do *host* hospedeiro.

A partir dos resultados obtidos nas avaliações ficou comprovado que a sobrecarga imposta pelo uso de contêineres Docker é inferior a do ambiente utilizando máquinas virtuais com hipervisor nos dois cenários analisados. Na primeira avaliação foram aplicadas duas cargas de trabalho diferentes (normal e pico de utilização) para três ambientes de provisionamento diferentes (Docker, *Baremetal* e VM), e o desempenho do Docker foi melhor que VM e muito próximo ao *Baremetal*. Na segunda avaliação, na qual foram alocadas até 5 Unidades Computacionais rodando uma carga de trabalho normal, o desempenho do Docker foi consideravelmente melhor. As evidências empíricas confirmam o que estava

posto na literatura. Dessa forma, é possível concluir que o desempenho dos contêineres Docker não são um obstáculo para a adoção dessa tecnologia para provisionamento das aplicações desenvolvidas pela Secretaria de Tecnologia da Informação do STF.

Como trabalhos futuros é considerada a avaliação do desempenho dos contêineres Docker trabalhando em *cluster* utilizando ferramentas como o Docker Swarm [10] ou Kubernetes [18], pois o presente trabalho foi realizado com a aplicação rodando em contêiner único. Também é relevante aferir o desempenho de aplicações utilizando o paradigma de Microsserviços [21]. Microsserviços são uma maneira de desenvolver e compor sistemas de software de tal forma que a construção se dá a partir de componentes pequenos e independentes que interagem um com os outros pela rede [46]. Isso é o oposto da abordagem monolítica, utilizada pela Aplicação Supremo Autuação, na qual há um único programa grande escrito em linguagem Java.

Referências

[1] Apache jmeter. http://jmeter.apache.org/. Acessado em: 26-02-2017. 43

[2] Best practices for writing dockerfiles. https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/. Acessado em: 01-11-2016. 42

[3] Calculating the number of virtual users (concurrent users) to test. http://support.loadimpact.com/knowledgebase/articles/265461-calculating-the-number-of-virtual-users-concurren. Acessado em: 14-03-2017. 47

[4] cgroups - linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html. Acessado em: 23-02-2017. 18

[5] Containers: Docker, windows and trends. https://azure.microsoft.com/pt-br/blog/containers-docker-windows-and-trends/. Acessado em: 27-01-2017. 8

[6] Coreos rkt overview. https://coreos.com/rkt/. Acessado em: 27-01-2017. 8

[7] Docker hub. https://hub.docker.com/. Acessado em: 09-10-2016. 17

[8] Docker hub centos official repository. https://hub.docker.com/_/centos/. Acessado em: 09-10-2016. 40

[9] Docker overview. https://docs.docker.com/engine/understanding-docker/. Acessado em: 22-02-2017. x, 18, 20, 21, 26

[10] Docker swarm mode overview. https://docs.docker.com/engine/swarm/. Acessado em: 21-02-2017. 56

[11] Dockerfile reference. https://docs.docker.com/engine/reference/builder/. Acessado em: 27-02-2017. xi, 22

[12] The evolution of linux containers and their future. https://dzone.com/articles/evolution-of-linux-containers-future/. Acessado em: 19-01-2017. 6, 7, 8

[13] The go programming language. https://golang.org/. Acessado em: 23-02-2017. 17

[14] The go programming language specification. https://golang.org/ref/spec. Acessado em: 23-02-2017. 17

[15] Google analytics. https://analytics.google.com/. Acessado em: 26-02-2017. 47

[16] Introduction to mongodb. https://docs.mongodb.com/manual/introduction/. Acessado em: 21-02-2017. 9

[17] Introduction to solaris zones. https://docs.oracle.com/cd/E19455-01/817-1592/zones.intro-1/index.html. Acessado em: 21-02-2017. 7

[18] Kubernetes - production-grade container orchestration. https://kubernetes.io/. Acessado em: 21-02-2017. 56

[19] Linux vserver overview. http://linux-vserver.org/Overview. Acessado em: 21-02-2017. 6

[20] Man page - systemd.slice. http://www.dsm.fordham.edu/cgi-bin/man-cgi.pl?topic=systemd.slice&ampsect=5. Acessado em: 15-03-2017. 49

[21] Microservices. https://martinfowler.com/articles/microservices.html. Acessado em: 21-02-2017. 56

[22] Microsoft hyper-v - sítio oficial. https://www.microsoft.com/pt-br/cloud-platform/server-virtualization?ocid=otc-c-br-loc--wikivirt. Acessado em: 21-02-2017. 1

[23] namespaces - overview of linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html. Acessado em: 24-02-2017. 18

[24] Node.js - sítio oficial. https://nodejs.org/en/. Acessado em: 21-02-2017. 9

[25] Open container initiative. https://www.opencontainers.org/. Acessado em: 27-01-2017. 8

[26] Portal jusbrasil - supremo tribunal federal. https://stf.jusbrasil.com.br/. Acessado em: 24-02-2017. 2

[27] Red hat customer portal software downloads - red hat jboss enterprise application platform 6.4.0. https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=appplatform&version=6.4. Acessado em: 01-11-2016. 42

[28] Red hat jboss enterprise application platform. https://developers.redhat.com/products/eap/overview/. Acessado em: 26-02-2017. 42

[29] Resource management in docker. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/. Acessado em: 15-03-2017. 49, 52

[30] Vmware - sítio oficial. http://www.vmware.com/br/solutions/virtualization.html. Acessado em: 21-02-2017. 1

[31] What is docker? https://www.docker.com/what-docker. Acessado em: 21-02-2017. x, 2, 10, 17

[32] Xen project. `https://www.xenproject.org/`. Acessado em: 21-02-2017. 1

[33] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014. 11, 16

[34] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009. 12

[35] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame. The impact of docker containers on the performance of genomic pipelines. *PeerJ*, 3:e1273, September 2015. 36, 38

[36] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614, March 2014. 14, 15, 16

[37] Z. J. Estrada, Z. Stephens, C. Pham, Z. Kalbarczyk, and R. K. Iyer. A performance evaluation of sequence alignment software in virtualized environments. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 730–737, May 2014. 34, 36

[38] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172, March 2015. 31, 46

[39] M. Hayden and R. Carbone. Securing linux containers. *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License*, 2015. 19

[40] B. Loeffler. Cloud computing: What is infrastructure as a service. *TechNet Magazine*, (10):45, 2011. x, 13

[41] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. *Performance testing guidance for web applications: patterns & practices*. Microsoft Press, 2007. 43, 46

[42] P. Mell and T. Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011. 12, 13

[43] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. 9

[44] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393, March 2015. 2, 28, 30

[45] N. S. Morais. Proposta de modelo de migração de sistemas de ambiente tradicional para nuvem privada para o polo de tecnologia da informação do Exército brasileiro. Master's thesis, Departamento de Ciência da Computação - Universidade de Brasília, 2015. 10

[46] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*. "O'Reilly Media, Inc.", 2015. 2, 9, 10, 17, 56

[47] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K.G. Shin. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007. 1

[48] C. Pahl. Containerisation and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015. x, 9, 11, 15, 16

[49] E. N. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri. Evaluation of docker containers based on hardware utilization. In *2015 International Conference on Control Communication Computing India (ICCC)*, pages 697–700, Nov 2015. 30

[50] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006. 19

[51] R. Righi. Elasticidade em cloud computing: conceito, estado da arte e novos desafios. *Revista Brasileira de Computação Aplicada*, 5(2):2–17, 2013. 12

[52] B. Sosinsky. *Cloud computing bible*, volume 762. John Wiley & Sons, 2010. 12

[53] F. R.C. Sousa, L. O. Moreira, and J. C. Machado. Computação em nuvem: Conceitos, tecnologias, aplicações e desafios. In *III Escola Regional de Computação Ceará, Maranhão e Piauí - ERCEMAPI*, pages 150–175. EDUFPI, 2009. x, 12, 15

[54] C. P. Wright and E. Zadok. Unionfs: Bringing file systems together. *Linux Journal*, 2004(128):24–29, December 2004. 20

Apêndice A

Dockerfiles

Este apêndice contém os Dockerfiles para a construção da imagem da aplicação Autuação do Supremo Tribunal Federal, utilizada para realização da análise de desempenho da plataforma de contêineres Docker.

A.1 Softwares e configurações de uso geral

```
FROM centos:6
MAINTAINER Flavio Henrique Rocha <flavioh@stf.jus.br>
#Repositorio dos softwares para instalacao
ENV REPO "https://[REGISTRO_STF]/recursos/"
#Configuracao do ambiente
ENV TERM xterm
RUN echo "alias ll='ls -lh'" >> /etc/bashrc;\
    echo "alias vim='vi'" >> /etc/bashrc
#Instalacao das bibliotecas necessarias
RUN yum reinstall -y glibc-common;\
    yum install -y unzip curl tar libaio passwd sudo
ENV LANG pt_BR.iso88591
#Instalacao do cliente Oracle
ENV REPO_PATH "sistema_operacional/oracle/"
ENV INSTALADOR [ORACLE_BIN].rpm
ENV ORACLE_HOME "[CAMINHO_BINARIO_ORACLE]"
ENV TNS_ADMIN "${ORACLE_HOME}/network/admin"
ENV LD_LIBRARY_PATH="${ORACLE_HOME}/lib"
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    rpm -ivh $INSTALADOR;\
    rm -f $INSTALADOR;\
    curl -LOkf "${REPO}${REPO_PATH}tnsnames.ora";\
    mkdir -p $TNS_ADMIN;\
    mv tnsnames.ora  $TNS_ADMIN
#Instalacao do Java JDK
ENV REPO_PATH "sistema_operacional/java/"
ENV INSTALADOR [JAVA_BINARIO].bin
ENV JAVA_HOME "[CAMINHO_BINARIO_JAVA]"
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    ./$INSTALADOR;\
```

```
    rm -f $INSTALADOR;\
ENV PATH $JAVA_HOME/bin:$PATH
ENV JAVA_OPTS ""
#Define o bash como comando default
CMD ["/bin/bash"]
```

A.2 Servidor de Aplicações

```
FROM registro/centos-java-oracle
MAINTAINER Flavio Henrique Rocha <flavioh@stf.jus.br>
#Repositorio dos softwares para instalacao
ENV REPO "https://[REGISTRO_STF]/recursos/"
#Criacao do usuario JBoss com permissao de sudo
ENV JBOSS_HOME /usr/jboss-eap-6.4/
RUN useradd -c "JBoss" -m jboss -s /bin/bash -d $JBOSS_HOME
#Instalacao do JBoss
ENV REPO_PATH "servidor_aplicacao/jboss-eap/6.4/"
ENV INSTALADOR jboss-eap-6.4.0.zip
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    unzip $INSTALADOR -d /usr ; \
    rm -f $INSTALADOR; \
    chown -R jboss:jboss $JBOSS_HOME
ENV PATH $JBOSS_HOME/bin:$PATH
#Cria usuario de gerenciamento para utilizacao da console
RUN $JBOSS_HOME/bin/add-user.sh -s -u jboss -p [SENHA]
#Instalacao dos modulos
ENV INSTALADOR modulos.tar.gz
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    tar -xvf $INSTALADOR -C $JBOSS_HOME/modules/system/layers/base/ ; \
    rm -f $INSTALADOR; \
    chown -R jboss:jboss $JBOSS_HOME/modules/system/layers/base/
#Alteracao do standalone.conf para receber parametros de memória
RUN echo $'JAVA_OPTS="-Xms$JAVA_XMS -Xmx$JAVA_XMX -XX:MaxPermSize=$JAVA_MAXPERMSIZE" \n \
JAVA_OPTS="$JAVA_OPTS"' >> $JBOSS_HOME/bin/standalone.conf
#Variaveis para configuracao de memoria do java
ENV JAVA_PERMSIZE 256m
ENV JAVA_MAXPERMSIZE 256m
ENV JAVA_XMS 1303m
ENV JAVA_XMX 1303m
#Variavel do repositorio remoto do standalone.xml
ENV STANDALONE_XML "${REPO}${REPO_PATH}"standalone.xml
RUN curl -LOkf STANDALONE_XML
#Define o usuario jboss para execucao
USER jboss
#Expoe as portas utilizadas
EXPOSE 8080 9990
#Define como diretorio de trabalho a pasta de configuracao
WORKDIR $JBOSS_HOME/standalone/configuration
ENTRYPOINT ["standalone.sh","-c","standalone.xml","-Djboss.bind.address=0.0.0.0","-Djboss
    .bind.address.management=0.0.0.0"]
```

A.3 Aplicação Supremo Autuação

```
FROM registro/centos-java-oracle
```

```
MAINTAINER Flavio Henrique Rocha <flavioh@stf.jus.br>
#Repositorio dos softwares para instalacao
ENV REPO "https://[REGISTRO_STF]/recursos/"
#Criacao do usuario JBoss com permissao de sudo
ENV JBOSS_HOME /usr/jboss-eap-6.4/
RUN useradd -c "JBoss" -m jboss -s /bin/bash -d $JBOSS_HOME
#Instalacao do JBoss
ENV REPO_PATH "servidor_aplicacao/jboss-eap/6.4/"
ENV INSTALADOR jboss-eap-6.4.0.zip
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    unzip $INSTALADOR -d /usr ; \
    rm -f $INSTALADOR; \
    chown -R jboss:jboss $JBOSS_HOME
ENV PATH $JBOSS_HOME/bin:$PATH
#Cria usuario de gerenciamento para utilizacao da console
RUN $JBOSS_HOME/bin/add-user.sh -s -u jboss -p [SENHA]
#Instalacao dos modulos
ENV INSTALADOR modulos.tar.gz
RUN curl -LOkf "${REPO}${REPO_PATH}${INSTALADOR}";\
    tar -xvf $INSTALADOR -C $JBOSS_HOME/modules/system/layers/base/ ; \
    rm -f $INSTALADOR; \
    chown -R jboss:jboss $JBOSS_HOME/modules/system/layers/base/
#Alteracao do standalone.conf para receber parametros de memória
RUN echo $'JAVA_OPTS="-Xms$JAVA_XMS -Xmx$JAVA_XMX -XX:MaxPermSize=$JAVA_MAXPERMSIZE" \n \
JAVA_OPTS="$JAVA_OPTS"' >> $JBOSS_HOME/bin/standalone.conf
#Variaveis para configuracao de memoria do java
ENV JAVA_PERMSIZE 256m
ENV JAVA_MAXPERMSIZE 256m
ENV JAVA_XMS 1303m
ENV JAVA_XMX 1303m
#Variavel do repositorio remoto do standalone.xml
ENV STANDALONE_XML "${REPO}${REPO_PATH}"standalone.xml
RUN curl -LOkf STANDALONE_XML
#Define o usuario jboss para execucao
USER jboss
#Expoe as portas utilizadas
EXPOSE 8080 9990
#Define como diretorio de trabalho a pasta de configuracao
WORKDIR $JBOSS_HOME/standalone/configuration
ENTRYPOINT ["standalone.sh","-c","standalone.xml","-Djboss.bind.address=0.0.0.0","-Djboss
    .bind.address.management=0.0.0.0"]
```

Apêndice B

Código fonte dos scripts utilizados

Este apêndice contém os códigos fontes dos scripts desenvolvidos em Python para realização dos *benchmarks*.

B.1 Código do script de Benchmark

A seguir é apresentado o código do script para captura das informações de uso de CPU e memória nos *hosts* hospedeiros dos ambientes de execução dos experimentos. O script foi adaptado a partir do código fonte desenvolvido por Mauricio Vergara Ereche. Disponível em `https://github.com/mave007/scripts/blob/master/psutil-process-benchmark.py`.

```
1  #!/usr/bin/env python2.7
2  #
3  import os
4  import sys
5  import csv
6  import psutil
7  import time
8  from datetime import datetime, timedelta
9  from optparse import OptionParser
10
11 now = int(time.time())
12
13 def is_empty(any_structure):
14     if any_structure:
15         return False
16     else:
17         return True
18
```

```
19 def query_yes_no(question, default="yes"):
20     """Ask a yes/no question via raw_input() and return their answer.
21 
22     "question" is a string that is presented to the user.
23     "default" is the presumed answer if the user just hits <Enter>.
24         It must be "yes" (the default), "no" or None (meaning
25         an answer is required of the user).
26 
27     The "answer" return value is one of "yes" or "no".
28     """
29     valid = {"yes": True, "y": True, "ye": True,
30              "no": False, "n": False}
31     if default is None:
32         prompt = " [y/n] "
33     elif default == "yes":
34         prompt = " [Y/n] "
35     elif default == "no":
36         prompt = " [y/N] "
37     else:
38         raise ValueError("invalid default answer: '%s'" % default)
39 
40     while True:
41         sys.stdout.write(question + prompt)
42         choice = raw_input().lower()
43         if default is not None and choice == '':
44             return valid[default]
45         elif choice in valid:
46             return valid[choice]
47         else:
48             sys.stdout.write("Please respond with 'yes' or 'no' "
49                              "(or 'y' or 'n').\n")
50 
51 def bytes2human(n):
52     """
53     >>> bytes2human(10000)
54     '9K'
55     >>> bytes2human(100001221)
56     '95M'
57     """
58     symbols = ('K','M')
59     prefix = {}
60     for i, s in enumerate(symbols):
61         prefix[s] = 1 << (i + 1) * 10
62     for s in reversed(symbols):
63         if n >= prefix[s]:
```

```
64              value = int(float(n) / prefix[s])
65              return '%s%s' % (value, s)
66      return "%sB" % n
67
68  def poll(interval, csvfile):
69      time.sleep(interval)
70
71      cpu_percent = psutil.cpu_percent()
72      cpu_time_percent = psutil.cpu_times_percent()
73      memoria_virtual = psutil.virtual_memory()
74      memoria_swap = psutil.swap_memory()
75      rede_io_counters = psutil.net_io_counters()
76
77      bench = (cpu_percent, cpu_time_percent.user, cpu_time_percent.system
            , cpu_time_percent.idle, memoria_virtual.total, memoria_virtual.
            available, memoria_virtual.used, memoria_virtual.free,
            memoria_swap.total, memoria_swap.used, memoria_swap.free,
            memoria_swap.percent, rede_io_counters.bytes_sent,
            rede_io_counters.bytes_recv, rede_io_counters.packets_sent,
            rede_io_counters.packets_recv)
78
79      return (bench, csvfile)
80
81  def writecsv(bench, csvfile):
82      # If procs is not set, it must be a header
83      if is_empty(bench):
84          line = (
85                  "num",
86                  "hora",
87                  "cpu_percent",
88                  "cpu_time_percent_user",
89                  "cpu_time_percent_system",
90                  "cpu_time_percent_idle",
91                  "memoria_virtual_total",
92                  "memoria_virtual_available",
93                  "memoria_virtual_used",
94                  "memoria_virtual_free",
95                  "memoria_swap_total",
96                  "memoria_swap_used",
97                  "memoria_swap_free",
98                  "memoria_swap_percent",
99                  "rede_io_counters.bytes_sent",
100                 "rede_io_counters.bytes_recv",
101                 "rede_io_counters.packets_sent",
102                 "rede_io_counters.packets_recv"
```

```
103             )
104             if csvfile != "__no_output_just_verbose":
105                 with open(csvfile, 'a') as f:
106                     writer = csv.writer(f)
107                     writer.writerows([line])
108                 f.close
109             else:
110                 print line
111         # Normal data
112         else:
113             line = (
114                     int(time.time()) - now, # Iteration number
115                     time.strftime("%H:%M:%S", time.localtime(time.time())),
                            # Hora
116                     bench[0],    # CPU Percent
117                     bench[1],    # CPU Time Percent User
118                     bench[2],    # CPU Time Percent System
119                     bench[3],    # CPU Time Percent Idle"
120                     bytes2human(bench[4]), # Memoria Virtual Total
121                     bytes2human(bench[5]), # Memoria Virtual Available
122                     bytes2human(bench[6]), # Memoria Virtual Used
123                     bytes2human(bench[7]), # Memoria Virtual Free
124                     bytes2human(bench[8]),  # Memoria Swap Total
125                     bytes2human(bench[9]),  # Memoria Swap Used
126                     bytes2human(bench[10]), # Memoria Swap Free
127                     bytes2human(bench[11]), # Memoria Swap Percent
128                     bytes2human(bench[12]), # Rede Bytes Sent
129                     bytes2human(bench[13]), # Rede Bytes Receive
130                     bytes2human(bench[14]), # Rede Packets Sent
131                     bytes2human(bench[15])  # Rede Packets Receive
132             )
133             if csvfile != "__no_output_just_verbose":
134                 with open(csvfile, 'ab') as f:
135                     writer = csv.writer(f)
136                     writer.writerows([line])
137                 f.close
138             else:
139                 print line
140 
141 #Main
142 def main():
143     try:
144         # We process the options and flags given from the command line
145         parser = OptionParser(usage="usage: %prog [-o filename] [-n num]
                [-v] ", version="%prog 1.0")
```

```
146         parser.add_option("-o", "--output",
147                           action="store", # optional because action
                                defaults to "store"
148                           dest="filename",
149                           default="output.csv",
150                           metavar="FILE",
151                           help="CSV file to save the output (default:
                                output.csv)"
152                           )
153         parser.add_option("-v", "--verbose",
154                           action="store_true",
155                           dest="verbose",
156                           help="Show output instead of writing CSV")
157         parser.add_option("-n",
158                           action="store",
159                           dest="iterations",
160                           help="Number of seconds to record")
161
162         (options, args) = parser.parse_args()
163
164         csvfile = options.filename
165         seconds = int(options.iterations)
166
167         if options.verbose == True:
168             csvfile="__no_output_just_verbose"
169         else:
170             csvfile = time.strftime("%d_%m_%Y_%H_%M_%S_", time.localtime
                    (time.time())) + csvfile
171             if os.path.isfile(csvfile):
172                 if query_yes_no ("File " + csvfile + " already exists.
                        Overwrite?"):
173                     os.remove(csvfile)
174             print "Writing CSV file into " + csvfile
175             print " ...press CTRL + C to stop..."
176
177         now = int(time.time())
178         cont = 0
179         interval = 0.1
180         args = ([], csvfile)
181         while 1:
182             writecsv(*args)
183             interval = 1
184             args = poll(interval, csvfile)
185             cont = cont + 1
186             if cont > seconds:
```

```
187                     sys.exit(0)
188         except (KeyboardInterrupt, SystemExit):
189             print "\nFinished."
190             pass
191 
192 
193 if __name__ == '__main__':
194     main()
```