UNIBRA

CENTRO UNIVERSITÁRIO BRASILEIRO

CENTRO UNIVERSITÁRIO BRASILEIRO - UNIBRA

CURSO DE GRADUAÇÃO TECNÓLOGO EM

REDES DE COMPUTADORES

JOÃO PAULO FALCÃO

ANÁLISE DE DESEMPENHO ENTRE MÁQUINAS VIRTUAIS E CONTAINERS UTILIZANDO O DOCKER

RECIFE/2022

JOÃO PAULO FALCÃO

ANÁLISE DE DESEMPENHO ENTRE MÁQUINAS VIRTUAIS E CONTAINERS UTILIZANDO O DOCKER

Trabalho de Conclusão de Curso apresentado ao Centro Universitário Brasileiro – UNIBRA, como requisito parcial para obtenção do título de tecnólogo em Redes de Computadores.

Professora Orientadora: Msc Ameliara Freire Santos de Miranda

RECIFE/2022

Ficha catalográfica elaborada pela
bibliotecária: Dayane Apolinário, CRB4- 1745.

F178a Falcão, João Paulo
Análise de desempenho entre máquinas virtuais e containers utilizando o docker / João Paulo Falcão. Recife: O Autor, 2022.
47 p.

Orientador(a): Msc. Ameliara Freire Santos de Miranda.

Trabalho De Conclusão De Curso (Graduação) - Centro Universitário Brasileiro – Unibra. Tecnólogo em Redes de Computadores, 2022.

Inclui Referências.

1. Conteinerização. 2. Docker. 3. Virtualização. I. Centro Universitário Brasileiro - Unibra. II. Título.

CDU: 004

Dedico esse trabalho aos meus familiares,
amigos e educadores.

AGRADECIMENTOS

Agradeço ao nosso Deus por toda força e sabedoria, que nos deu fôlego para que pudéssemos chegar ao resultado final deste trabalho.

À nossa orientadora Ameliara Freire Santos de Miranda por toda paciência e dedicação na orientação deste trabalho, onde sempre se mostrou uma profissional excelente e competente.

Ao nosso coorientador Humberto Caetano Cardoso da Silva pelo seu vasto conhecimento enriquecedor e suas dicas preciosas.

E por fim quero agradecer aos nossos familiares e amigos que sempre nos apoiaram e acreditaram em nossa capacidade, que sempre estiveram conosco e exigiram nosso melhor.

A todos os meus sinceros agradecimentos.

“A emoção mais antiga e mais forte da humanidade é o medo, e o mais antigo e mais forte de todos os medos é o medo do desconhecido.”

(Howard Phillips Lovecraft)

LISTA DE FIGURAS

Figura 1 - Virtualização completa....................15
Figura 2 - Paravirtualização....................16
Figura 3 - Máquinas Virtuais vs Containers....................17
Figura 4 - Containers Linux....................18
Figura 5 - Arquitetura do Docker Engine....................20
Figura 6 - Comando para criação de uma imagem Docker....................21
Figura 7 - Exemplo de um Dockerfile....................22
Figura 8 - Processo de criação de uma imagem....................23
Figura 9 - Comando docker run....................24
Figura 10 - Comando docker images....................24
Figura 11 - Estrutura de um arquivo docker-compose.yml....................27
Figura 12 - Comando docker-compose up....................28
Figura 13 - Comando docker-compose ps....................28
Figura 14 - Comandos para instalação do Apache e MySQL....................30
Figura 15 - Configuração do container para acesso ao MySQL....................30
Figura 16 - Instalação do Apache na máquina virtualiza....................31
Figura 17 - Página padrão do servidor web Apache....................31
Figura 18 - Configuração do MySQL na máquina virtual....................32
Figura 19 - Estrutura para coleta de dados....................32
Figura 20 - Grafana integrado com Zabbix monitorando SRV001 e SRV002....................33
Figura 21 - Instalação do servidor Tomcat 7....................34
Figura 22 - Algoritmo para inserção e seleção no banco de dados....................35
Figura 23 - Algoritmo simulando a criação de múltiplos usuários....................35
Figura 24 - Log da base de dados dos servidores....................36
Figura 25 - Teste do Apache Bench no servidor SRV003....................37
Figura 26 - Código para execução do Apache Bench....................38
Figura 27 - Estrutura geral para os testes....................38
Figura 28 - Teste utilizado para a carga do MySQL....................39
Figura 29 - Consumo de CPU com o MySQL....................40
Figura 30 - Consumo de Memória RAM com o MySQL....................40
Figura 31 - Leitura e escrita de disco com o MySQL....................41

Figura 32 - Entrada e saída de rede com o MySQL...41
Figura 33 - Teste utilizado para a carga do Apache...42
Figura 34 - Consumo de CPU com o Apache...43
Figura 35 - Consumo de Memória RAM com o Apache..43
Figura 36 - Leitura e escrita de disco com o Apache..44
Figura 37 - Entrada e saída de rede com o Apache...44

LISTA DE ABREVIATURAS E SIGLAS

BASH – Bourne Again SHell

CPU - Central Processing Unit

CLI – Command line Interface

HTML - HyperText Markup Language

IBM - International Business Machines

ID - Identity

KVM - Kernel-based Virtual Machine

LXC - Linux Containers

RAM - Random Access Memory

SO - Operating System

TAG - Keywords

TCP - Transmission Control Protocol

VM - Virtual Machine

VMM - Virtual Machine Manager

YAML - YAML Ain't Markup Language

SUMÁRIO

1 INTRODUÇÃO.....12
1.1 Objetivos.....13
1.1.1 Objetivo geral.....13
1.1.2 Objetivos específicos.....13
2 METODOLOGIA DA PESQUISA.....13
2.1 Virtualização.....13
2.1.1 Virtualização completa.....14
2.1.2 Paravirtualização.....15
2.1.3 Virtualização de sistema operacional.....17
2.2 Docker.....20
2.2.1 Control Groups.....21
2.2.2 Namespaces.....21
2.2.3 Dockerfile.....22
2.2.4 Docker Compose.....26
3 TRABALHOS RELACIONADOS.....30
3.1. Desenvolvimento e implementação dos cenários utilizados.....30
3.1.1 Cenário 1 SRV001.....30
3.1.2 Cenário 2 SRV002.....31
3.1.3 Cenário 3 SRV003.....33
3.2 Preparação para os testes.....35
3.2.1 Teste para o MySQL.....35
3.2.2 Teste para o Apache.....37
4 RESULTADOS E ANÁLISES.....40
4.1 Teste de carga com o MySQL.....40
4.2 Teste de carga com o Apache.....43
5 CONSIDERAÇÕES FINAIS.....46
REFERÊNCIAS.....47

ANÁLISE DE DESEMPENHO ENTRE MÁQUINAS VIRTUAIS E CONTAINERS UTILIZANDO O DOCKER

Resumo: A proposta deste trabalho é apresentar o modelo de conteinerização, através da ferramenta Docker e realizar a comparação com a virtualização utilizando hipervisores do tipo 1, bare-metal. Para traçar essa linha comparativa e obter o resultado esperado, foram utilizados três servidores físicos de hardwares idênticos com o sistema operacional GNU/Linux: Um servidor para coleta de dados utilizando o software *Zabbix* e Dois como clientes executando o banco de dados *mysql* e o serviço web *apache2,* mediando o tempo de execução entre ambos os ambientes.

Palavras-chave: Conteinerização, Docker, Virtualização.

PERFORMANCE ANALYSIS BETWEEN VIRTUAL MACHINES AND CONTAINERS USING THE DOCKER

Abstract: The purpose of this work is to present the containerization model through the Docker tool and compare it with virtualization using type 1, bare-metal hypervisors. To draw this comparative line and obtain the expected result, three physical servers with identical hardware with the GNU/Linux operating system were used: A server for data collection using the Zabbix software and Two as clients running the mysql database and the service web apache2, mediating the runtime between both environments.

Keywords: Containerization, Docker, Virtualization.

1 INTRODUÇÃO

Nos últimos anos o constante avanço das tecnologias aplicado à resolução de problemas se tornou crucial para desenvolvedores e empresas buscando sempre equilibrar custo e entrega de resultados (MAZIERO, 2014). Nesse contexto e visando o uso comercial a IBM, entre as décadas de 60 e 70, criou o primeiro sistema operacional com suporte a máquinas virtuais (SILBERSCHATZ et al., 2015).

Esse conceito permitiu que empresas migrassem, de forma gradual, os grandes e caros computadores (mainframes), para servidores de porte menor (CARISSIMI, 2008). Para Tanenbaum e Bos (2015) a utilização de máquinas virtuais (VM) acabou mudando a forma de como se usar vários sistemas operacionais em um único hardware. Porém Maziero (2014) observou que, apesar dos seus benefícios, a virtualização tem seus contratempos. Muito pelo fato da forma como o hardware e software se comportam para prover essa e emulação, chamada de hipervisor (MAZIERO, 2014).

O Hipervisor tem um papel fundamental para as soluções utilizando máquinas virtuais, se tornando o principal agente desta tecnologia (TANENBAUM; BOS, 2015). Carissimi (2008) aponta que quando o hipervisor é executado é feita toda a emulação do hardware do sistema operacional hospedeiro gerando um alto processamento. Por outro lado se torna vantajoso emular todo um ambiente computacional visando custo, desempenho e implantação segura de novas aplicações (CARISSIMI, 2008).

De acordo com Silberschatz et al. (2015) a consolidação da virtualização possibilitou o surgimento da computação em nuvem, e consequentemente, a utilização dos containers. Os containers utilizam o que se chama de virtualização leve dispensando o uso de emulação, utilizando uma imagem que precisa apenas do kernel do hospedeiro para ser executada (SCHEEPERS, 2014). Essa imagem proporciona o isolamento dos containers em execução, algo que foi herdado do sistema operacional UNIX com a funcionalidade chamada chroot (SILVA, 2017). A utilização dessa imagem e realizado através da ferramenta Docker que empacota os binários e dependências facilitando e permitindo o gerenciamento dos containers (SCHEEPERS, 2014).

Percebendo ligeira vantagem que os containers têm sobre os hipervisores, este trabalho visa apresentar uma solução que utiliza a instalação e configuração das mesmas aplicações e as compara utilizando os containers Docker com máquinas virtuais utilizando hipervisor.

1.1 Objetivos

1.1.1 Objetivo geral

Desenvolver uma solução para que seja utilizada com conteinerização e hipervisores, coletando seus respectivos dados e analisando o desempenho entre ambos.

1.1.2 Objetivos específicos

- Implementar três máquinas GNU/Linux com o mesmo hardware que serão definidos como:
    - Um servidor que monitorará todo o consumo de recursos das máquinas clientes, coletando seus dados para posterior análise e comparação;
    - Duas máquinas clientes que simularão várias requisições simultâneas.
- Comparar o desempenho entre a tecnologia de conteinerização e virtualização com hipervisor.

2 METODOLOGIA DA PESQUISA

Este trabalho tem como objetivo embasar os conceitos sobre os tipos de virtualização e suas classificações. Também serão abordadas as diferenças entre containers e máquinas virtuais. Por fim o Docker, ferramenta de gerenciamento de containers, será apresentado, finalizando com avaliação de desempenho para obtenção de resultados na utilização de serviços e aplicações.

A virtualização é extremamente importante para o cenário da tecnologia atual. De um ambiente de produção a um provedor de serviços em nuvem, as máquinas virtuais fornecem uma facilidade de gerenciamento e produtividade muito positiva. A conteinerização, através da ferramenta Docker, potencializou esse conceito e é algo que será descrito nos próximos capítulos deste trabalho.

2.1 Virtualização

A origem das máquinas virtuais se iniciou por volta de 1960, quando a IBM desenvolveu o projeto M44/M44X com a intenção de avaliar o recente conceito sobre sistemas compartilhados (SEO, 2009).

Segundo Tanenbaum e Bos (2015), a virtualização permite que um único computador seja hospedeiro de múltiplas máquinas virtuais convidadas.

> Uma máquina virtual provê uma interface de hardware completa para um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente. Cada sistema operacional convidado tem a ilusão de executar sozinho uma plataforma de hardware exclusiva (MAZIERO, 2014, p. 429).

A partir desta definição, Maziero (2014) afirma que as máquinas virtuas utilizam da abstração de hardware para disponibilizar vários sistemas operacionais dentro de um único hipervisor.

De acordo com Seo (2009) o hipervisor (VMM – Monitor de Máquina Virtual) é o responsável pela criação e gerenciamento dos ambientes simulados, possibilitando a execução completa de um sistema operacional.

A consolidação das VMs permitiu criar ambientes seguros para execução e isolamento de aplicações não-confiáveis além de teste de hardware, permitindo a não alteração do ambiente de produção (SEO, 2009).

2.1.1 Virtualização completa

Os estudos de Silberschatz et al. (2015) salientam que na virtualização completa todo o hardware é virtualizado e o sistema operacional é executado sobre um hipervisor. Segundo os autores nessa abordagem não existe alteração do sistema operacional convidado e o hipervisor controla todos os dispositivos do hospedeiro.

A Figura 1 ilustra a estrutura da virtualização completa. Pode-se observar a camada do hipervisor interagindo com o hardware e o sistema operacional, provendo os recursos para camadas acima (MAZIERO, 2014).

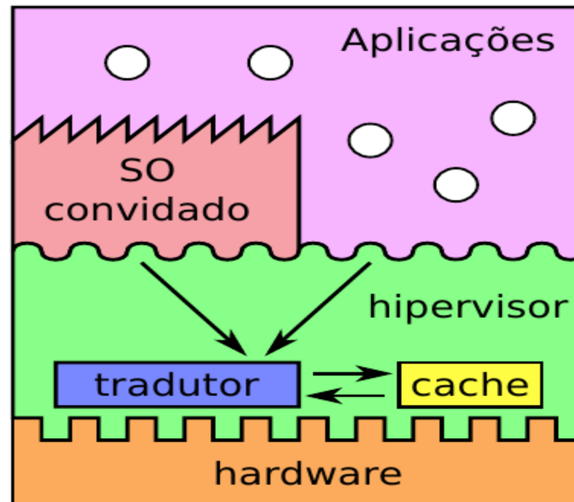
> A virtualização completa consiste em prover uma réplica do hardware subjacente de tal forma que o sistema operacional e as aplicações podem executar como se estivesse executando diretamente no hardware original. (CARISSIMI, 2008, p. 11)

Com base no pensamento de Carissimi (2008) é possível observar que há uma dificuldade no que diz respeito a implementar máquinas virtuais que simulam exatamente cada tipo de dispositivo.

Para Maziero (2014) a virtualização completa pode apresentar um custo extremamente alto referente a desempenho.

Exemplos de virtualização completa: KVM e VMware (TANENBAUM; BOS, 2015).

Figura 1 - Virtualização completa



Fonte – Adaptado de Maziero (2014)

2.1.2 Paravirtualização

A paravirtualização se comporta de um forma diferente se comparada a virtualização completa, oferecendo uma virtualização que não replica o hardware do hospedeiro (SILBERSCHATZ et al., 2015).

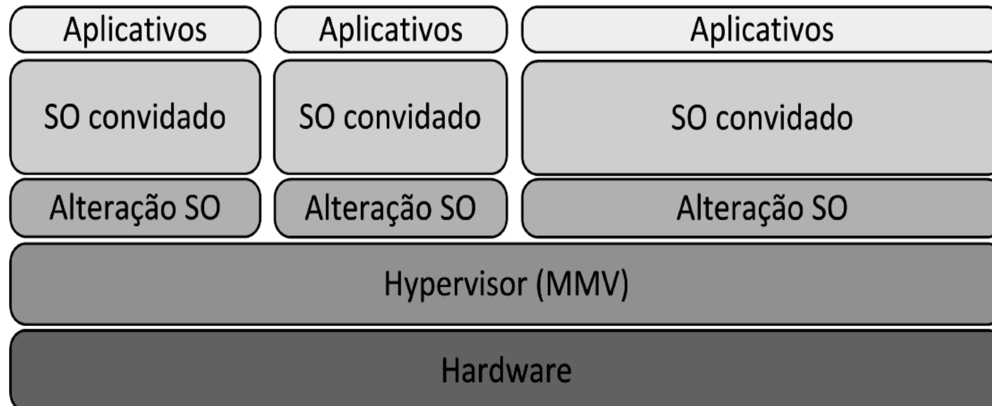
Carissimi (2008) explica que, para este tipo de abordagem, o sistema operacional convidado precisa ser modificado, possibilitando o melhor aproveitamento dos recursos do hardware hospedeiro.

De acordo com Tanenbaum e Bos (2015, p. 329) a paravirtualização apresenta uma interface de software semelhante a uma máquina que expõe explicitamente o fato de que se trata de um ambiente virtualizado.

Com esta afirmação, Tanenbaum e Bos (2015) concluíram que esse tipo de virtualização permite que vários sistemas operacionais diferentes sejam executados com o mesmo hardware.

Na Figura 2 Lange (2013) ilustra como o hipervisor, como camada principal, controla o hardware e a camada de alteração do SO (sistema operacional), recebendo todas as instruções de execução.

Figura 2 - Paravirtualização



Fonte – Adaptado de Lange (2013)

2.1.3 Virtualização de sistema operacional

Também chamada de virtualização baseada em containers, essa tecnologia surgiu como uma alternativa a virtualização que utiliza o hipervisor, proporcionando desempenho e escalabilidade (SILVA, 2017).

Segundo Freire (2021), esse processo permite o compartilhamento do kernel do sistema operacional e o isolamento das aplicações em execução.

> O conceito de containers foi iniciado com a introdução da funcionalidade chamada chroot no sistema operacional UNIX. Essa tecnologia consiste em uma chamada de sistema 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 (SILVA, 2017, p. 6).

Com essa afirmação, Sillva (2017) sugere que os containers apresentam vantagens referente as VMs, dependendo apenas do kernel da máquina hospedeira, já que os sistemas são encapsulados em imagens.

Os containers são leves, portáteis e com baixo consumo de recursos, descartando a necessidade de emulação ou requisito de hardware (SCHEEPERS, 2014). Segundo o autor, um container é executado a partir de uma imagem pronta, que possui os arquivos de configuração, binários, bibliotecas e dependências.

De acordo com Freire (2021, p. 29) o Docker é uma ferramenta open-source criada para facilitar a criação, implementação e execução e gerenciamento de aplicações a partir de containers.

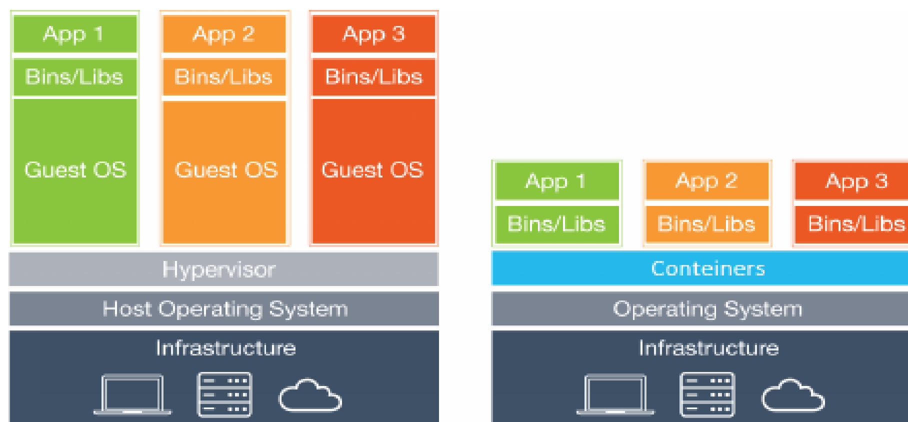
A partir do pensamento de Freire (2021) podemos verificar que os containers precisam da ferramenta Docker, executando sobre uma distribuição GNU/Linux, para fornecer suas funcionalidades.

Porém Bernstein (2014) menciona que o Docker não foi a primeira tecnologia a implementar o conceito de *containeres.* O mesmo autor aponta o LXC (Linux Containers) como responsável pelo surgimento da conteinerização.

Segundo Silva (2017) para conceber a conteinerização, o LXC utiliza o cgroups e namespaces (que serão apresentados no capítulo 2.2.1 e 2.2.2) que são duas funcionalidades especiais do kernel Linux.

A Figura 3 demonstra um comparativo entre máquinas virtuais (esquerda) e containers (direita), onde nota-se o privilégio que o hipervisor exerce sobre os recursos de hardware do host hospedeiro. Nesse processo o hipervisor, como camada principal, emulada todo o hardware do hospedeiro criando seus próprios dispositivos, causado uma possível sobrecarga quando as máquinas hóspedes estiverem em execução. Algo que não acontece no processo de conteinerização por não necessitar de um hipervisor e sim apenas do kernel do hospedeiro.

Figura 3 - Máquinas Virtuais vs Containers



Fonte – Adaptado de Gomes (2017)

Segundo Vitalino e Castro (2016, p. 27):

> Containers são bem similares às máquinas virtuais, porém mais leves e mais integrados ao sistema operacional da máquina host, uma vez que, compartilha o seu kernel, o que proporciona melhor desempenho por conta do gerenciamento único dos recursos.

s o

c uma

g

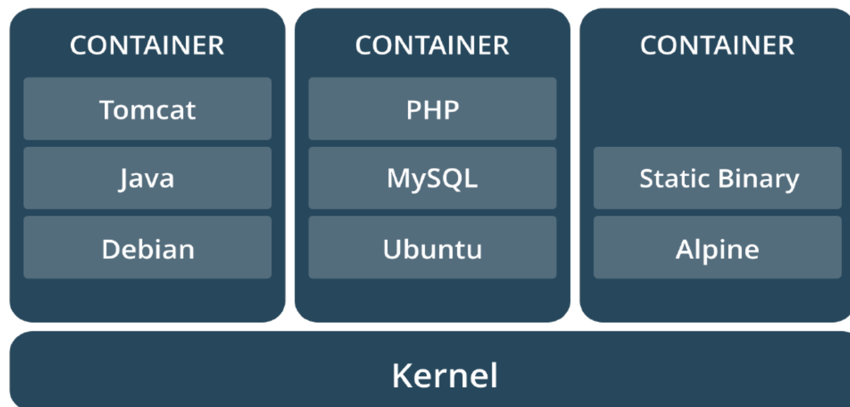
para

AM,

), o

gerenciamento da disponibilidade é crucial no que diz respeito à tolerância que os servidores terão a falhas. O compartilhamento de recursos torna possível o uso dos c*ontainers* em qualquer ambiente devido ao seu isolamento a partir da criação e empacotamento da imagem (DELGADO, 2021).

Figura 5 - Containers Linux



A Figura 4 apresenta como cada *container* se comporta quando está em execução, a partir de uma imagem, em um sistema operacional GNU/Linux (DELGADO, 2021). Nesse contexto a imagem, já empacotada com tudo que é necessário, precisa apenas do kernel do SO para colocar cada *container* em execução de forma isolada.Fonte - Adaptado de Delgado (2021)

2.2 Docker

O Docker iniciou seu desenvolvimento em meados de 2013 pela Docker Inc e sua base de criação é a linguagem de programação GO, desenvolvida pela Google (SILVA, 2017).

De acordo com Gomes (2017, p. 17), Docker é uma plataforma aberta, criada com o objetivo de facilitar o desenvolvimento, a implantação e a execução de aplicações em ambientes isolados.

A partir dessa definição, Gomes (2017) afirma que o Docker é uma ferramenta que automatiza e facilita o gerenciamento de *containers,* isolando todos os processos que estão em execução.

Segundo Nickoloff e Kuenzli (2019) o funcionamento do Docker utiliza a arquitetura cliente-servidor e tem como base o Docker Engine, que possui os seguintes componentes:

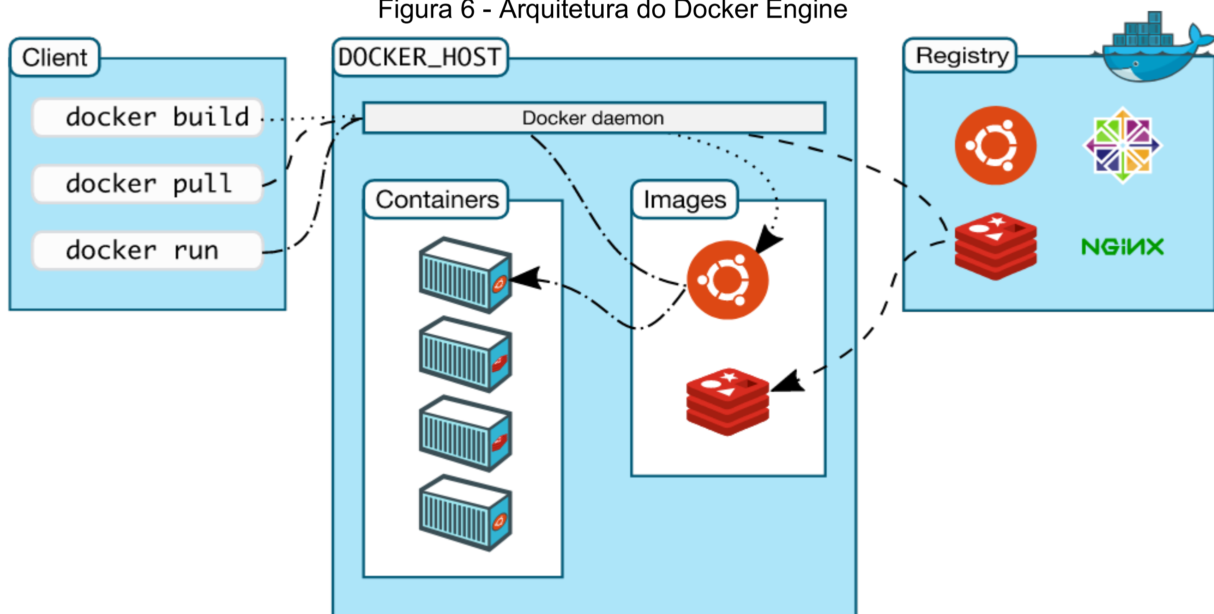
- Docker Daemon: É um processo de sistema, executado em segundo plano, que aguarda os comandos enviados pelo Docker Client para construir, gerenciar e executar as imagens e *containers*;
- Docker Client: Se comunica, através de comandos, diretamente com o Docker Daemon;

- Docker Registry: Repositório onde as imagens estão armazenadas, sendo o Docker Hub o principal.

A Figura 5 demonstra como acontece a interação dos componentes do Docker Engine. Para executar um container o Docker Client se comunica com Docker Daemon, que faz o download da imagem através do Docker Hub (Registry), onde o Docker Daemon, através de comandos do Docker Client, cria e executa os *containers* (NICKOLOFF; KUENZLI, 2019).

Fonte - https://docs.docker.com/get-started/overview/

Figura 6 - Arquitetura do Docker Engine



2.2.1 Control Groups

Os grupos de controle (cgroups) é uma das funcionalidades do kernel Linux que é responsável por gerenciar o quanto de recurso (CPU, Memória RAM, Rede e Dispositivos) será alocado para cada *container* (SCHEEPERS, 2014). Esse limite imposto pelo cgroups permite que o kernel crie hierarquias de grupos (processos) e monitore cada requisição de novos containers (GOMES, 2017).

Os cgroups permitem que o Docker Engine controle como cada container se comporta, executando, parando e retornando sem afetar outro container ou causar uma sobrecarga no host (DELGADO, 2021).

2.2.2 Namespaces

Namespaces é outra funcionalidade do kernel Linux que isola cada container separadamente tornando cada processo parte de um conjunto de namespaces sendo invisíveis para outros processos (BERNSTEIN, 2014).

O isolamento é possível devido a cada container possuir sua própria árvore de processos separando um usuário (ID) que esteja executando um container do usuário da máquina host (NICKOLOFF; KUENZLI, 2019).

Segundo Vitalino e Castro (2016), o Docker Engine utiliza os seguintes namespaces:

- PID namespace: Isola o processo através de um identificador único;
- NET namespace: Faz o gerenciamento das interfaces de rede, permitindo que cada container tenha a sua;
- IPC namespace: Permite a gestão e acesso aos recursos;
- MNT namespace: Define o ponto de montagem e o sistema de arquivos;
- UTS namespace: Provê isolamento do kernel, hotsname, e identificadores de versão.

2.2.3 Dockerfile

O Dockerfile é um arquivo, em modo texto, responsável por criar as imagens seguindo um script que executa, de forma recursiva, cada linha definida neste arquivo (DELGADO, 2021). Este arquivo contém uma lista de instruções (definições) que são executadas através do Docker Client, onde a compilação da imagem é realizada pelo Docker Daemon (GOMES, 2017).

A base de criação das imagens é a partir de um comando interno do Docker (docker build*)* que faz a verificação de toda a estrutura do Dockerfile, analisando sua integridade e se há possíveis erros (de sintaxe ou digitação). Na Figura 6 é apresentado o comando docker build, para construção de imagens.

Figura 7 - Comando para criação de uma imagem Docker

```
[root@centosdocker Dockerfile]# docker build -t <nome da imagem>
```

Fonte – Próprio autor (2022)

O parâmetro “-t” (menos t) do comando *docker build*, indica o nome (TAG) que irá compor a imagem, servindo como contexto de pesquisa, execução e remoção. A criação de uma imagem também gera um número único (IMAGE ID), que identifica cada imagem separadamente, onde com o comando docker images pode-se realizar a consulta. A grande vantagem de utilizar um arquivo em modo texto é que, caso necessite alterar “algo” na imagem, apenas se edita o Dockerfile e executa, novamente, o comando docker build. Isso facilita e otimiza muito o deploy em qualquer ambiente, seja de infraestrutura ou desenvolvimento.

Segundo Gomes (2017, p. 42):

> É importante atentar que o arquivo Dockerfile é uma sequência de instruções lidas do topo à base e cada linha é executada por vez. Se alguma instrução depender de outra instrução essa dependência deve ser descrita mais acima no documento.

A partir desse pensamento Gomes (2017) afirma que, o Dockerfile precisa ser construído com prioridade de execução, mesmo que em sua estrutura não seja obrigatório o uso de case-sensitive.

Mesmo com a não obrigatoriedade do case-sensitive nas definições do Dockerfile, por convenção entre os desenvolvedores e melhor legibilidade, se utiliza letras maiúsculas, algo que será apresentado na Figura 7. Ao ser executado, o aquivo Dockerfile utilizará o sistema operacional Ubuntu 20.04 como base para criar a imagem onde será adicionado a variável de ambiente (PATH) do hospedeiro com o caminho do diretório *“/teste”* no qual o script *./relatorioMaquina.sh* será adicionado para execução.

Figura 8 - Exemplo de um Dockerfile

```
1 FROM ubuntu:20.04
2 CMD ["bash"]
3 ENV PATH=/teste:$PATH
4 COPY relatoriaMaquina.sh /teste/
5 WORKDIR /teste
6 ENTRYPOINT ["./relatorioMaquina.sh"]
```

Fonte – Próprio autor (2022)

As definições utilizadas no aquivo Dockerfile[1] serão detalhadas a seguir:

1 https://docs.docker.com/engine/reference/builder/

- FROM: Obrigatoriamente deverá ser a primeira instrução definida, no qual será a base da imagem utilizada;
- CMD: Comando executado, por padrão, caso nenhum seja informado durante a inicialização de um container;
- ENV: Define variáveis de ambiente que estarão disponíveis para todas as instruções do Dockerfile referente a imagem utilizada;
- COPY: Faz a cópia de novos arquivos ou diretórios adicionando ao sistema de arquivo da imagem;
- WORKDIR: Define o diretório raiz (/) para as instruções no Dockerfile. Caso não exista, é criado automaticamente;
- ENTRYPOINT: Especifica um executável a um container durante sua inicialização.

Podemos verificar na Figura 8 o passo a passo da construção de uma imagem Docker. Após o parâmetro “-t”, do comando *docker build*, foi passado o nome da imagem junto com a sua TAG. O ponto (.) ao final do comando indica que será feita a compilação a partir do diretório corrente e onde também estão o arquivo Dockerfile e o script *./relatorioMaquina.sh*. Observa-se que cada passo executado gera um container “provisório” que após concluído é removido. A mensagem de sucesso ao final apresenta o número e nome da imagem construída.

Figura 9 - Processo de criação de uma imagem

```
[root@centosdocker Dockerfile]# docker build -t ubuntu:relatoriomaquina .
Sending build context to Docker daemon  3.584kB
Step 1/6 : FROM ubuntu:20.04
20.04: Pulling from library/ubuntu
8e5c1b329fe3: Already exists
Digest: sha256:115822d64890aae5cde3c1e85ace4cc97308bb1fd884dac62f4db0a16dbddb36
Status: Downloaded newer image for ubuntu:20.04
 ---> 1a437e363abf
Step 2/6 : CMD ["bash"]
 ---> Running in 5db2ca78d76b
Removing intermediate container 5db2ca78d76b
 ---> b30e94e8fd3b
Step 3/6 : ENV PATH=/teste:$PATH
 ---> Running in b6550e5d126a
Removing intermediate container b6550e5d126a
 ---> db2777d1182c
Step 4/6 : COPY relatorioMaquina.sh /teste/
 ---> 46502eb5c332
Step 5/6 : WORKDIR /teste
 ---> Running in c4980618d480
Removing intermediate container c4980618d480
 ---> 90bf0ee79c9c
Step 6/6 : ENTRYPOINT ["./relatorioMaquina.sh"]
 ---> Running in 08254c1af64b
Removing intermediate container 08254c1af64b
 ---> 6893e361a2f3
Successfully built 6893e361a2f3
Successfully tagged ubuntu:relatoriomaquina
```

Fonte – Próprio autor (2022)

A Figura 9 apresenta a imagem “relatoriomaquina” iniciando o container no qual tem a finalidade de executar o script *./relatorioMaquina.sh*. Esse script foi desenvolvido em shell bash e apresenta as principais características de uma host com sistema operacional GNU/Linux. Com o comando *docker run* podemos iniciar o *container* passando como parâmetro o sistema operacional *ubuntu* e o nome da imagem *relatoriomaquina.* A opção *“–rm”* (menos 2x) informa ao Docker Daemon que após executar o container ele seja encerado.

Figura 10 - Comando docker run

```
[root@centosdocker Dockerfile]# docker run ubuntu:relatoriomaquina --rm
============================================================================
Relatório da Máquina: 6893e361a2f3
Data/Hora: 21-04-2022 01:39
============================================================================

Máquina Ativa desde: 2022-04-21 01:03:00

Versão do Kernel: 3.10.0-1160.66.1.el7.x86_64

CPUs:
Quantidade de CPUs/Core: 1
Modelo da CPU: AMD Opteron 63xx class CPU

Memória Total: 1.8G

Partições:
Filesystem                          Size  Used Avail Use% Mounted on
overlay                             4.8G  1.4G  3.2G  30% /
shm                                  64M     0   64M   0% /dev/shm
/dev/mapper/centos_docker-var       4.8G  1.4G  3.2G  30% /etc/hosts
============================================================================
[root@centosdocker Dockerfile]#
```

Fonte – Próprio autor (2022)

Com o comando *docker images* podemos verificar, na Figura 10, todas as informações, divididas em colunas, referentes imagem que foi criada, apresentando qual foi o sistema operacional utilizado, o nome da imagem, o número que identifica a imagem, há quanto tempo ele foi criada e o seu tamanho total.

Figura 11 - Comando docker images

```
[root@centosdocker Dockerfile]# docker images
REPOSITORY    TAG                IMAGE ID       CREATED          SIZE
ubuntu        relatoriomaquina   6893e361a2f3   4 minutes ago    72.8MB
```

Fonte - Próprio autor (2022)

2.2.4 Docker Compose

De acordo com Delgado (2021), o Docker Compose é uma ferramenta para gerenciamento de múltiplos containers, que tem como sua principal funcionalidade iniciar vários containers, através de um único arquivo definição.

Segundo Gomes (2017), este arquivo de definição é do tipo YAML[2], o que torna obrigatório o uso da indentação, algo que pode invalidar todo o seu conteúdo caso não sejam respeitadas as regras de indentação.

2 https://yaml.org/

> Com o Docker Compose, você consegue, em um único arquivo, indicar as instruções para a criação de diversos containers e tudo o que a aplicação precisar. Com isso você não cria somente um Dockerfile para criar uma imagem, e sim, você monta um arquivo YAML, mesmo que seja em vários containers, para criar seu ambiente (VITALINO; CASTRO, 2016, p. 251)

Baseado no pensamento de Vitalino e Castro (2016), a composição desse arquivo pode ser desenvolvida através de instruções e parâmetros referentes a cada ambiente que se deseja implementar. Como Docker Compose trata todos os containers definidos como serviços, isso permite inúmeras possibilidades de dimensionamento do ambiente onde será utilizado, evitando o excesso de Dockerfiles (VITALINO; CASTRO, 2016).

O arquivo docker-compose.yml[3] possui várias definições, porém as mais relevantes serão descritas a seguir:

- version: Esta definição é obrigatória em todo início de arquivo e serve para indicar a versão do compose que será utilizada;
- services: Informa quais serviços (containers) serão definidos;
- build: Informa um arquivo (Dockerfile) que servirá de base para construção de uma imagem;
- image: Imagem que será utilizada para iniciar o container;
- volumes: Define o caminho onde os dados persistentes serão armazenados;
- networks: Define as redes que serão anexadas aos containers;
- restart: Define uma ação de como o container irá se comportar caso ocorra falha de execução;
- environment: Variáveis de ambiente utilizadas dentro dos containers;
- depends_on: Define, basicamente, que para um serviço ser inicializado ele depende que um outro seja iniciado primeiro;
- ports: Define quais portas dos containers devem ser expostas para acesso de um host hospedeiro;
- expose: Informa quais portas, para comunicação entre containers, serão expostas.

A grande vantagem em utilizar o Docker Compose, além da portabilidade e simplicidade, é poder executar múltiplos serviços com apenas um arquivo e alguns comandos.

---

3 https://docs.docker.com/compose/compose-file/

O Docker Compose possui muitos comandos CLI[4] (Interface de Linha de Comando), onde alguns serão apresentados a seguir:

- docker-compose up: Cria e inicia os containers;
- docker-compose down: Encerra todos os containeres e remove imagem, rede e volume;
- docker-compose start: Inicia os containers;
- docker-compose stop: Paralisa os containers;
- docker-compose restart: Reinicia os containers;
- docker-compose build: Realiza, apenas, a construção de imagens;
- docker-compose ps: Lista os containers em execução;
- docker-compose logs: Visualiza os logs dos containers.

Seguindo as boas práticas, como já mencionado, de um arquivo tipo yml, um docker-compose.yml tem a seguinte estrutura, apresentada na Figura 11.

É possível notar que as definições respeitam uma hierarquia em cada seção definida, seguindo, linha a linha, a construção de cada serviço. A proposta do referido docker-compose.yml foi criar um exemplo de uma pilha *LAMP*[5] onde na primeira linha foi escolhido utilizar a versão “3” do Docker compose. Na terceira linha, no mesmo nível de indentação, serão definidos os serviços utilizados. A partir da quarta até a décima terceira linha foi defino o banco de dados utilizado, no qual foi escolhido o *mariadb*[6] em sua última versão (latest) disponível no repositório Docker Hub, a senha do usuário administrador, a base de dados, o usuário “padrão” e sua senha e onde serão armazenados os arquivos persistentes de dentro do container para a máquina host. Seguindo da décima quarta até a vigésima terceira linha foi utilizado o *php*[7], em sua versão 8.1 com o módulo *apache2*[8], ambos dependentes da inicialização bem-sucedida do banco de dado*s* e com seu volume e portas definidas, onde o container utilizará a mesma porta TCP da máquina host. Por fim, entre a vigésima quarta até a trigésima terceira linha, o *phpmyadmin*[9] foi utilizado com

---

4 https://docs.docker.com/compose/reference/

5 https://pt.wikipedia.org/wiki/LAMP

6 https://mariadb.org/

7 https://www.php.net/

8 https://apache.org/

a mesma dependência de inicialização, com suas portas expostas e suas variáveis de ambientes utilizadas dentro do container.

Figura 12 - Estrutura de um arquivo docker-compose.yml

```
 1 version: '3'
 2
 3 services:
 4   mariadb:
 5     image: mariadb:latest
 6     restart: always
 7     environment:
 8       MYSQL_ROOT_PASSWORD: rootpasswd
 9       MYSQL_DATABASE: lamp
10       MYSQL_USER: user
11       MYSQL_PASSWORD: userpasswd
12     volumes:
13       - "./mariadb:/var/lib/mysql"
14   webserver:
15     image: php:8.1-apache
16     restart: always
17     volumes:
18       - ".:/var/www/html"
19     depends_on:
20       - mariadb
21     ports:
22       - 80:80
23       - 443:443
24   phpmyadmin:
25     image: phpmyadmin:latest
26     restart: always
27     depends_on:
28       - mariadb
29     ports:
30       - 8080:80
31     environment:
32       - PMA_HOST=mariadb
33       - PMA_PORT=3306
```

Fonte – Próprio autor (2022)

A Figura 12 demonstra como o comando docker-compose up cria cada container, definindo um número, ao final do nome, indicando que para aquele tipo de serviço específico apenas um container foi criado. O parâmetro “-d” (menos d) indica que, após a execução do comando, o terminal estará disponível para uso. Isso possibilita realizar outras tarefas durante a execução dos serviços, não havendo necessidade de iniciar outra instância de terminal.

9 https://www.phpmyadmin.net/

Figura 13 - Comando docker-compose up

```
[root@centosdocker lamp]# docker-compose up -d
Creating network "lamp_default" with the default driver
Creating lamp_mariadb_1 ... done
Creating lamp_phpmyadmin_1 ... done
Creating lamp_webserver_1  ... done
[root@centosdocker lamp]#
```

Fonte – Próprio autor (2022)

Na Figura 13 é possível verificar, com o comando docker-compose ps, os serviços em execução. Nota-se a coluna com os nomes dos containers, o comando que os “subiu”, o estado de execução, e as portas que foram expostas para acesso externo (Host).

Figura 14 - Comando docker-compose ps

```
[root@centosdocker lamp]# docker-compose ps
      Name                     Command               State                           Ports
---------------------------------------------------------------------------------------------------------------------------
lamp_mariadb_1      docker-entrypoint.sh mariadbd    Up      3306/tcp
lamp_phpmyadmin_1   /docker-entrypoint.sh apac ...   Up      0.0.0.0:8080->80/tcp,:::8080->80/tcp
lamp_webserver_1    docker-php-entrypoint apac ...   Up      0.0.0.0:443->443/tcp,:::443->443/tcp, 0.0.0.0:80->80/tcp,:::80->80/tcp
```

Fonte – Próprio autor (2022)

3 TRABALHOS RELACIONADOS

O objetivo deste trabalho é apresentar uma comparação, utilizando serviços e configurações idênticas, entre *containers* e máquinas virtuais. Referente a essa motivação, o artigo a seguir apresenta uma proposta que foi a base para os resultados obtidos dessa comparação.

3.1. Desenvolvimento e implementação dos cenários utilizados

Para realizar os testes e comprovar os resultados obtidos neste artigo, Fell (2018) utilizou três servidores Dell modelo PowerEdge M710HD que continham um processador Intel Xeon E5-2450, disco rígido de 120GB e memória RAM de 130GB.

Os servidores foram nomeados de SRV001, SRV002 e SRV003, facilitando a identificação de cada cenário separadamente, onde no cenário 1 foi implementado a ferramenta Docker e o cenário 2 o hipervisor do tipo 1 VMware ESXI[10] v6 (FELL, 2018).

Fell (2018) optou por utilizar o sistema operacional GNU/Linux Ubuntu 18.04, no qual foram instalados em ambos os servidores como uma forma de padronização, evitando coleta de dados imprecisas.

A estrutura dos cenários 1, 2 e 3 serão descritas a seguir:

- Cenário 1 SRV001 – Sistema operacional GNU/Linux Ubuntu 18.04, com o a ferramenta Docker e os serviços Apache e MySQL instalados;
- Cenário 2 SRV002 – O hipervisor do tipo 1 VMware ESXI virtualizando o sistema operacional GNU/Linux Ubuntu 18.04 com os serviços Apache e MySQL instalados.
- Cenário 3 SRV003 – Servidor para coleta de dados.

3.1.1 Cenário 1 SRV001

A instalação do Docker Engine[11], no SRV001, foi realizada seguindo a documentação oficial conforme apresentado em seu site. Após instalado o Docker, foi necessário criar um *container* com o banco de dados MySQL, em sua versão 5.7, e outro *container* com o servidor web Apache na versão 2.4, apresentados na Figura 14.

10 https://www.vmware.com/br/products/esxi-and-esx.html

11 https://docs.docker.com/engine/install/ubuntu/

Figura 15 - Comandos para instalação do Apache e MySQL

```
1  // COMANDO PARA BAIXAR ÚLTIMA VERSÃO DO APACHE
2  sudo docker pull httpd
3
4  // COMANDO PARA CRIAR O CONTAINER APACHE
5  sudo docker run -dit --name apache -p 80:80 -v /home/user/
   website/:/usr/local/apache2/htdocs/ httpd:2.4
6
7  // COMANDO PARA INICIAR O CONTAINER APACHE
8  sudo docker start apache
9
10 // COMANDO PARA BAIXAR ULTIMA VERSÃO DO MYSQL
11 sudo docker pull mysql/mysql-server:latest
12
13 // COMANDO PARA CRIAR O CONTAINER MYSQL
14 sudo docker run --name mysql -e MYSQL_ROOT_HOST=% -e
   MYSQL_ROOT_PASSWORD=1234 -p 3307:3307 -d mysql/mysql-server
15
16 // COMANDO PARA INICIAR O CONTAINER MYSQL
17 sudo start mysql
```

Fonte – Adaptado de Fell (2018)

Conforme apresentado na Figura 15, Fell (2018) executou os comandos para criar o usuário de dentro do container, já iniciado, pelo fato de que, por padrão, o MySQL bloqueia todos os acessos externos.

Figura 16 - Configuração do container para acesso ao MySQL

```
1  //ACESSAR LOCALMENTE O BANCO DE DADOS
2  sudo docker exec -it mysql mysql -p
3
4  // CRIAR UM USUARIO COM ACESSO EXTERNO
5  GRANT ALL ON *.* TO 'gerson'@'%' IDENTIFIED BY '29PQ71ij@@' WITH GRANT OPTION;
```

Fonte – Adaptado de Fell (2018)

3.1.2 Cenário 2 SRV002

A instalação do VMware ESXI 6.0[12], como hipervisor do tipo 1, foi executada seguindo o tutorial em seu site oficial. O sistema operacional GNU/Linux Ubuntu foi instalado e virtualizado com os mesmos serviços utilizados no cenário 1 (FELL, 2018).

12 https://kb.vmware.com/s/article/2130503?lang=pt_PT

Com a máquina virtual Ubuntu inicializada foi realizado a instalação do servidor web Apache, conforme mostra os comandos ilustrados na Figura 16.

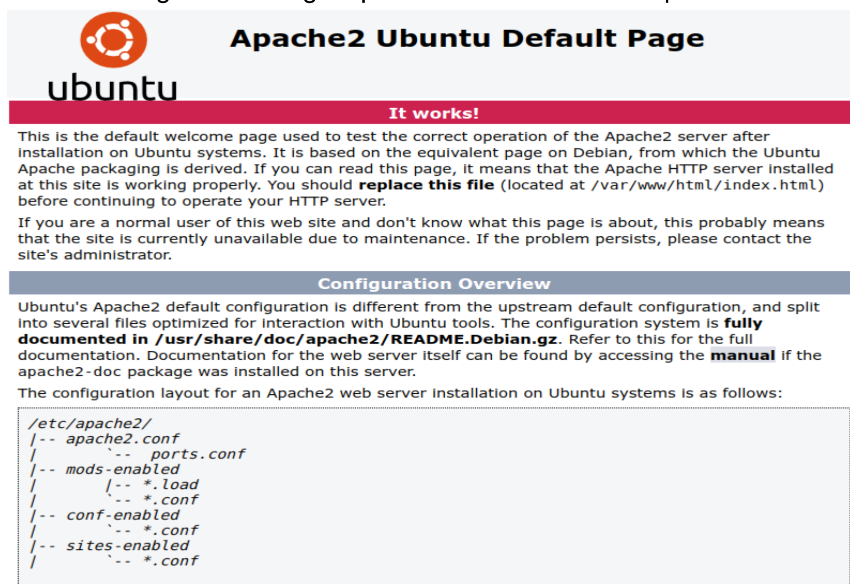
Figura 17 - Instalação do Apache na máquina virtualiza

```
1 # ATUALIAR OS REPOSITÓRIOS
2 sudo apt update
3
4 # ATUALIZAR TODOS OS PACOTES DO SISTEMA
5 sudo apt upgrade
6
7 # INSTALAR O PACOTE DO APACHE
8 sudo apt install apache2
9
10 # INICIAR O SERVIÇO DO APACHE
11 sudo systemctl start apache2.service
12
13 # VERIFICAR O STATUS DO SERVIÇO DO APACHE
14 sudo systemctl status apache2.service
```

Fonte – Próprio autor, adaptado de Fell (2018)

Após inicialização do serviço foi verificado, através da página de teste padrão do Apache, o sucesso obtido, apresentado na Figura 17.

Figura 18 - Página padrão do servidor web Apache



Fonte – Próprio autor, adaptado de Fell (2018)

Prosseguindo com a configuração do cenário 2, podemos observar na Figura 18 que foi instalado o bando de dados MySQL e logo em seguida criado um usuário para acesso externo, da mesma forma que o cenário 1 (FELL, 2018).

Figura 19 - Configuração do MySQL na máquina virtual

```
1  //Instalação do MYSQL
2  sudo apt update
3  sudo apt install mysql-server
4  // Configuração do acesso externo
5  mysql -uroot -p
6  mysql> GRANT ALL ON *.* TO 'gerson'@'%' IDENTIFIED BY '29PQ71ij@@' WITH GRANT OPTION;
```

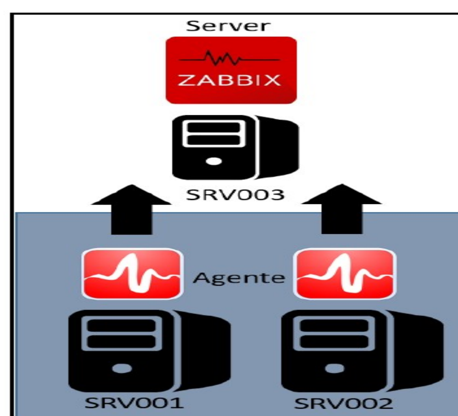
Fonte – Adaptado de Fell (2018)

3.1.3 Cenário 3 SRV003

Fell (2018) utilizou o software de código aberto Zabbix[13] para realizar o monitoramento dos agentes (SRV001 e SRV002) e obter os resultados, algo primordial para a proposta de seu artigo. A grande vantagem em utilizar o Zabbix é que sua essência é monitorar redes, servidores e serviços, permitindo analisar diversos componentes de TI (FELL, 2018).

A estrutura idealizada por Fell (2018), ilustrada na Figura 19, apresenta o Zabbix Server 3.x instalado no SRV003 e um agente instalado para o SRV001 e outro para o SRV002, seguindo o tutorial[14] de instalação e configuração no site oficial.

Figura 20 - Estrutura para coleta de dados



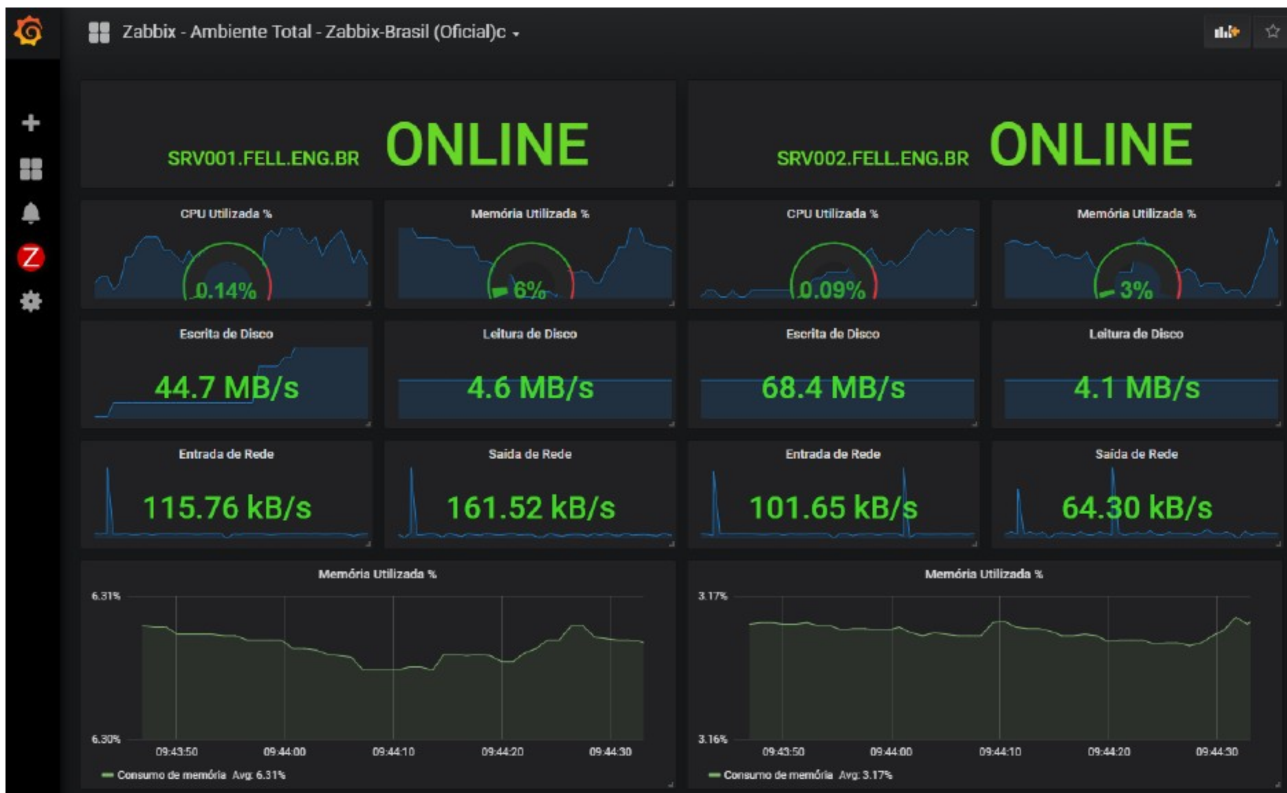
Fonte – Adaptado de Fell (2018)

13 https://www.zabbix.com/

14 https://www.zabbix.com/documentation/3.4/en/manual/installation/install_from_packages/debian_ubuntu

Porém foi observado por Fell (2018) que o Zabbix não permitia o compartilhamento dos gráficos em tempo real, forçando a procura por outro software que atendesse essa necessidade. O software de código aberto Grafana[15] foi a escolha do autor, permitindo além de melhor legibilidade o acompanhamento dos resultados em tempo real. A instalação foi realizada no SRV003 e a Figura 20 mostra o layout final após configuração feita pelo autor. É possível visualizar a organização de todos os componentes (CPU e RAM utilizadas; leitura e gravação dos discos rígidos; entrada e saída dos pacotes de redes) para cada agente configurado (FELL, 2018).

Figura 21 - Grafana integrado com Zabbix monitorando SRV001 e SRV002



Fonte – Adaptado de Fell (2018)

15 https://grafana.com/docs/grafana/latest/setup-grafana/installation/debian/

3.2 Preparação para os testes

Fell (2018) decidiu que o modelo de testes serão realizados através de múltiplas requisições simultâneas para ambos os servidores agentes. O intuito do autor é fazer a captura dos dados em tempo real e analisar os gráficos gerados. O autor utilizou a linguagem Java Server Pages[16] (JSP), tecnologia que é utilizada no desenvolvimento de páginas HTML dinâmicas, para desenvolver uma aplicação que utilizou o framework ZK[17]. Essa aplicação foi hospedada no servidor de páginas Tomcat 7[18] instalado no servidor SRV003. Os comandos executados para instalar essa ferramenta serão apresentados na Figura 21 (FELL, 2018).

Figura 22 - Instalação do servidor Tomcat 7

```
1 # ATUALIAR OS REPOSITÓRIOS
2 sudo apt update
3
4 # ATUALIZAR TODOS OS PACOTES DO SISTEMA
5 sudo apt upgrade
6
7 # INSTALAR O PACOTE DO TOMCAT
8 sudo apt install tomcat7
9
10 # INICIAR O SERVIÇO DO TOMCAT
11 sudo systemctl restart tomcat7.service
12
13 # VERIFICAR O STATUS DO SERVIÇO DO TOMCAT
14 sudo systemctl status tomcat7.service
```

Fonte – Próprio autor, adaptado de Fell (2018)

3.2.1 Teste para o MySQL

O modelo de algoritmo desenvolvido por Fell (2018) tem o objetivo de realizar a inserção e seleção de dados, que, em cada ciclo, efetuará a inserção e após finalizar vai realizar a consulta.

É possível verificar, na Figura 22, que o código espera o número de usuários e a quantidade de vezes que cada usuário vai inserir e selecionar as tabelas per_1 e per_2 através de dois parâmetros (FELL, 2018).

---

16 https://pt.wikipedia.org/wiki/JavaServer_Pages

17 https://en.wikipedia.org/wiki/ZK_(framework)

18 https://tomcat.apache.org/tomcat-7.0-doc/index.html

Figura 23 - Algoritmo para inserção e seleção no banco de dados

```
String sql = "insert into per_1 values ( MD5(RAND()), MD5(RAND()), MD5(RAND()), MD5(RAND()), MD5(RAND()), ?, ?, ?, ?, ? )";

LogUtilities.getInstance().write( sql );

PreparedStatement ps = db.getPreparedStatement( sql );

ps.setBinaryStream( 1, ImageProvider.IMG.get( 1 ) );
ps.setBinaryStream( 2, ImageProvider.IMG.get( 2 ) );
ps.setBinaryStream( 3, ImageProvider.IMG.get( 3 ) );
ps.setBinaryStream( 4, ImageProvider.IMG.get( 4 ) );
ps.setBinaryStream( 5, ImageProvider.IMG.get( 5 ) );

try
{
    ps.execute();
}

finally
{
    ps.close();
}

sql = "select * from per_1";
```

Fonte – Adaptado de Fell (2018)

O autor também desenvolveu um algoritmo para criação de múltiplos usuários, que podemos visualizar na Figura 23.

Para isto é criado *threads* para cada usuário adicionado possibilitando a simulação de múltiplos usuários inserindo e selecionado ao mesmo tempo, simulando uma concorrência de acessos nos servidores de testes (FELL, 2018, p. 67).

A partir dessa afirmação o autor informa que esse mesmo código também espera por um parâmetro para o número de usuários.

Figura 24 - Algoritmo simulando a criação de múltiplos usuários

```
private void doStatment( Event evt ) {
    StatmentData data = statmentPane.getData();
    for ( int i = 0; i <= data.getUser(); i++ ) {
        fireStatment( i, Base.DOCKER );
        fireStatment( i, Base.VIRTUAL_MACHINE ); }
    Prompts.info( "Executando..." );
}
private void fireStatment( int user, Base base )  {
    StatmentData data = statmentPane.getData();
    Thread t = new Thread(() ->
    { try  {
            Database db = Database.getInstance( base );
            LogUtilities.getInstance().write( "Conectando ao " + base + "...." );
            try {
                switch ( data.getType() ) {
                    case MIXED:
                        StatmentController.doStatment( data, db );
                    break;
                    case HTTP:
                        RequestController.doRequest( data, base );
                    break; } }
            finally
            { LogUtilities.getInstance().write( "Disconenctando ao " + base + "...." );
             db.release();
            } }
        catch ( Exception e )
        {
           e.printStackTrace( System.err );
        } } );
    t.setName( base.toString() + " - " + ( user + 1 ) );
    t.setDaemon( true );
    t.start(); }
```

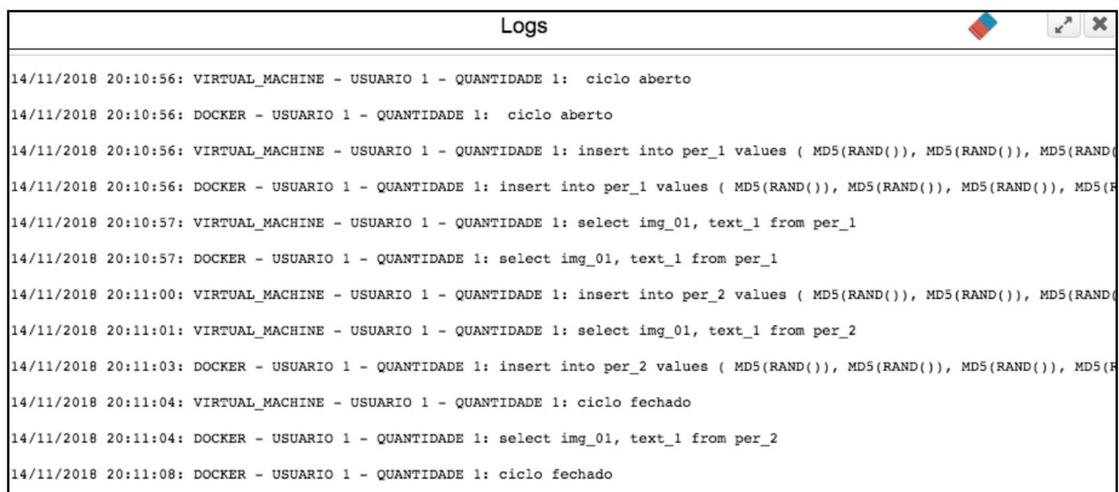
Fonte – Adaptado de Fell (2018)

O intuito do autor, com o algoritmo da figura anterior, é repetir uma quantidade “x” de vezes que cada usuário executará as operações no banco de dados.

A Figura 24 apresenta um melhor entendimento, analisando os logs, sobre como a aplicação se comporta quando o código é executado.

É possível visualizar que os comandos são executados paralelamente entre os servidores, inserindo e consultando as tabelas, nessa ordem.

Figura 25 - Log da base de dados dos servidores



Fonte – Adaptado de Fell (2018)

3.2.2 Teste para o Apache

A ferramente escolhida por Fell (2018) para este modelo, onde será testado e analisado o desempenho dos servidores, foi o Apache Bench software executado por linha de comando que permite a inserção de três parâmetros. O autor menciona que esses parâmetros podem ser respectivamente: números de páginas, números de usuários e o nome do servidor responsável pela execução dos testes.

Na Figura 25 podemos verificar que o teste foi realizado com 80 usuários simulando o acesso a 90 páginas no servidor web. É possível visualizar que o Apache Bench retorna algumas informações sobre a execução dos testes no SRV003.

Figura 26 - Teste do Apache Bench no servidor SRV003

```
gerson@srv003:~$ ab -n 90 -c 80 -k srv005.fell.eng.br/
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking srv005.fell.eng.br (be patient).....done


Server Software:        Apache/2.4.35
Server Hostname:        srv005.fell.eng.br
Server Port:            80

Document Path:          /
Document Length:        1883 bytes

Concurrency Level:      80
Time taken for tests:   0.009 seconds
Complete requests:      90
Failed requests:        0
Keep-Alive requests:    90
Total transferred:      195020 bytes
HTML transferred:       169470 bytes
Requests per second:    10252.90 [#/sec] (mean)
Time per request:       7.803 [ms] (mean)
Time per request:       0.098 [ms] (mean, across all concurrent requests)
Transfer rate:          21696.20 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    3   1.1      3       4
Processing:     1    2   0.7      2       4
Waiting:        1    2   0.7      2       4
Total:          1    5   1.5      5       8

Percentage of the requests served within a certain time (ms)
  50%      5
  66%      6
  75%      6
  80%      6
  90%      6
  95%      7
  98%      8
  99%      8
 100%      8 (longest request)
```

Fonte – Adaptado de Fell (2018)

O código, ilustrado na Figura 26, desenvolvido por Fell (2018) demostra uma função que inicia, no servidor SRV003, o software Apache Bench, através da linha de comando e permite que os testes nos servidores SRV001 e SRV002 sejam executados. O objetivo do autor com esta função é que seja simulado o acesso de alguns usuários a vários serviços (Ex: 1 usuário acessa 3 páginas) simultaneamente em ambos os servidores de testes.

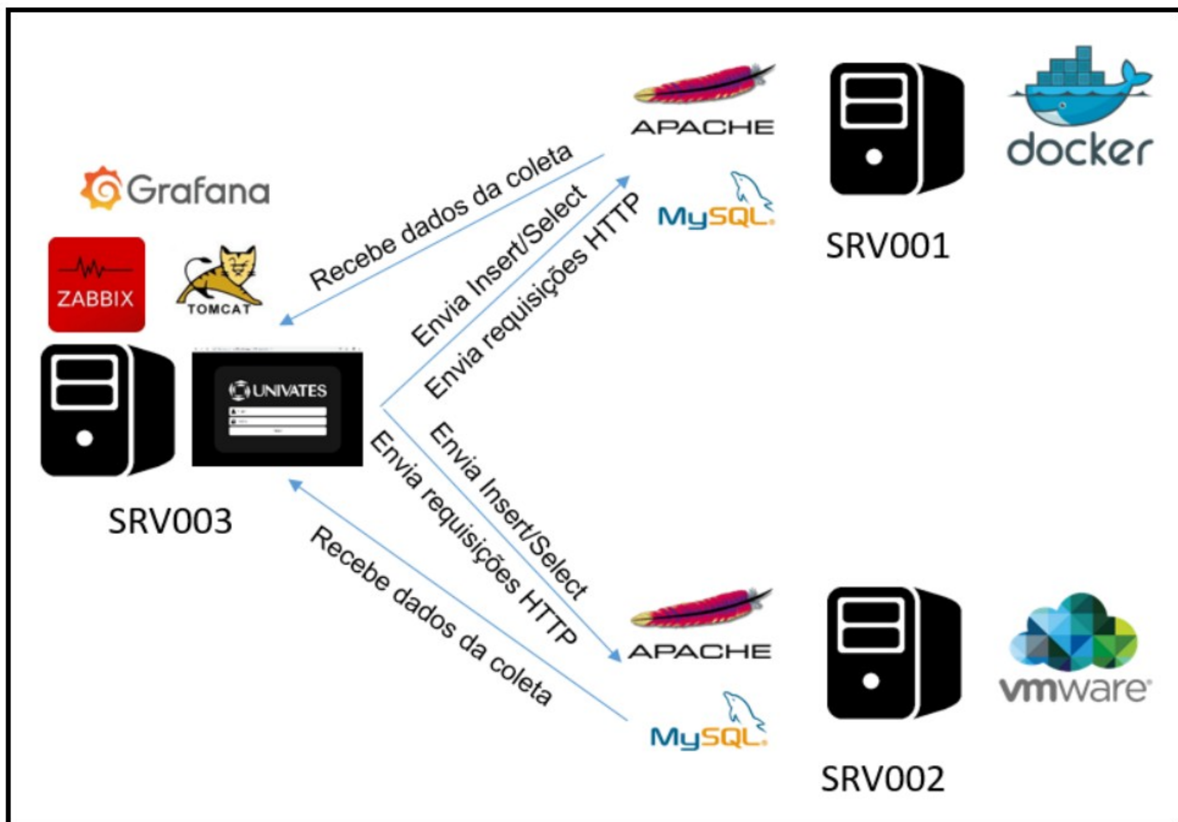
Figura 27 - Código para execução do Apache Bench

```
public static void doRequest( StatmentData data, Base base )
  {
      try {
      LogUtilities.getInstance().write( -1, "Requisitando " + base.host() + ":" + base.port() );
           ProcessBuilder builder = new ProcessBuilder( "ab", "-n", String.valueOf( data.getPaginas() * data.getUser() ) +
          "-c", String.valueOf( "50" ), "-k", base.host() + ":" + base.host() );
          builder.redirectErrorStream( true );
          BufferedReader input = new BufferedReader(new InputStreamReader(builder.start().getInputStream()));
          String line, lines = "\n";
          try {
              while ((line = input.readLine()) != null) {
                  lines += line + System.lineSeparator();
              }
              LogUtilities.getInstance().write( -1, lines );
                 } catch (IOException e) {
              LogUtilities.getInstance().write( -1, "ERROR: " + e.getMessage() );
          }
      }
      catch ( Exception e ) {
          try  {
              LogUtilities.getInstance().write( -1, "ERROR: " + e.getMessage() ); } catch ( Exception ex ) {
              ApplicationContext.getInstance().logException( ex );
          }  } } }
```

Fonte – Adaptado de Fell (2018)

A visão geral, idealizada por Fell (2018), de toda a arquitetura proposta para a demonstração dos testes e validação do seu artigo, pode ser visualizada na Figura 27.

Figura 28 - Estrutura geral para os testes



Fonte – Adaptado de Fell (2018)

4 RESULTADOS E ANÁLISES

Os testes e análises serão realizados com todos os cenários que foram apresentados por Fell (2018) a partir do capítulo 3. A avaliação será com base no consumo dos seguintes itens:

- SRV001 – Será analisado o uso da tecnologia de conteinerização através do consumo de CPU, Memória RAM, disco rígido e rede;
- SRV002 – Será analisado o uso da tecnologia de máquina virtual do tipo 1 através do consumo de CPU, Memória RAM, disco rígido e rede;

Todas as análises serão disponibilizadas através de gráficos seguindo a ordem:

- Gráficos da direita SRV001
- Gráficos da esquerda SRV002

4.1 Teste de carga com o MySQL

Fell (2018) decidiu que os parâmetros de testes para a carga do bando de dados dos servidores SRV001 e SRV002 foram os apresentados da Figura 28. O autor chegou a esses parâmetros pelo fato de que foi realizado inúmeros testes com o tempo máximo de 5 minutos para o processamento dos servidores, tornando esse tempo satisfatório para sua análise e coleta dos dados.

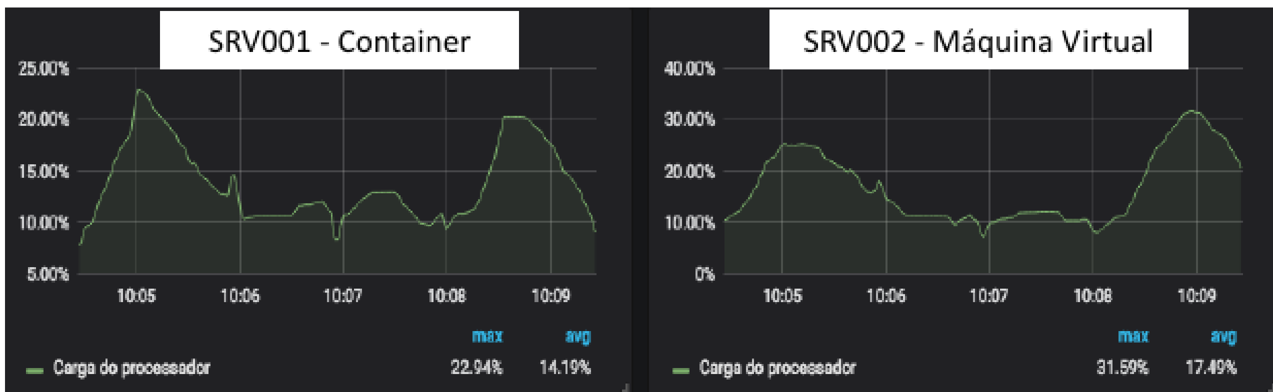
Figura 29 - Teste utilizado para a carga do MySQL

Fonte – Adaptado de Fell (2018)

O teste com carga para o MySQL foi configurado por Fell (2018) para 2200 usuários efetuarem 100 repetições cada um.

As figuras a seguir apresentam como cada servidor se comportou.

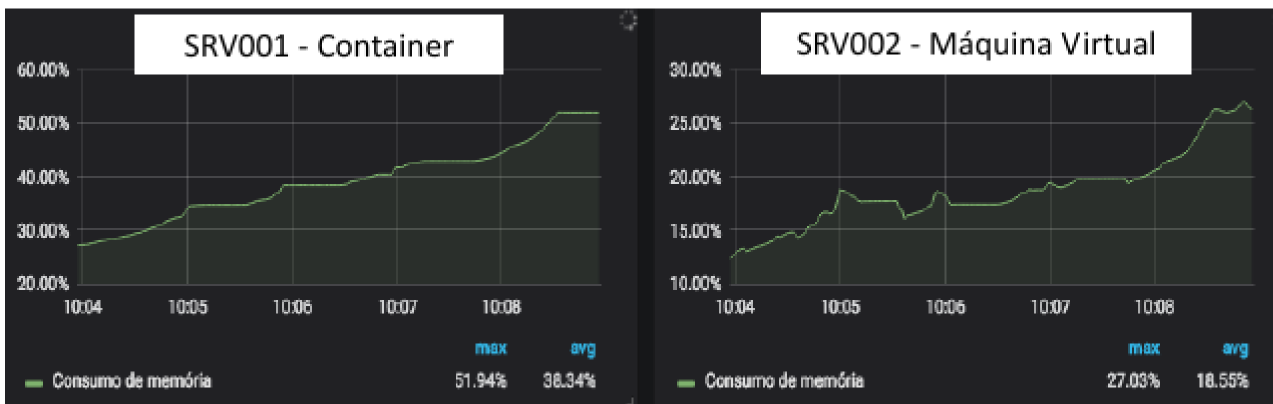
Figura 30 - Consumo de CPU com o MySQL



Fonte – Adaptado de Fell (2018)

É possível verificar na Figura 29 que o comportamento é similar entre ambos os servidores, com uma diferença de 3,3% na média de 5 minutos de utilização (FELL, 2018).

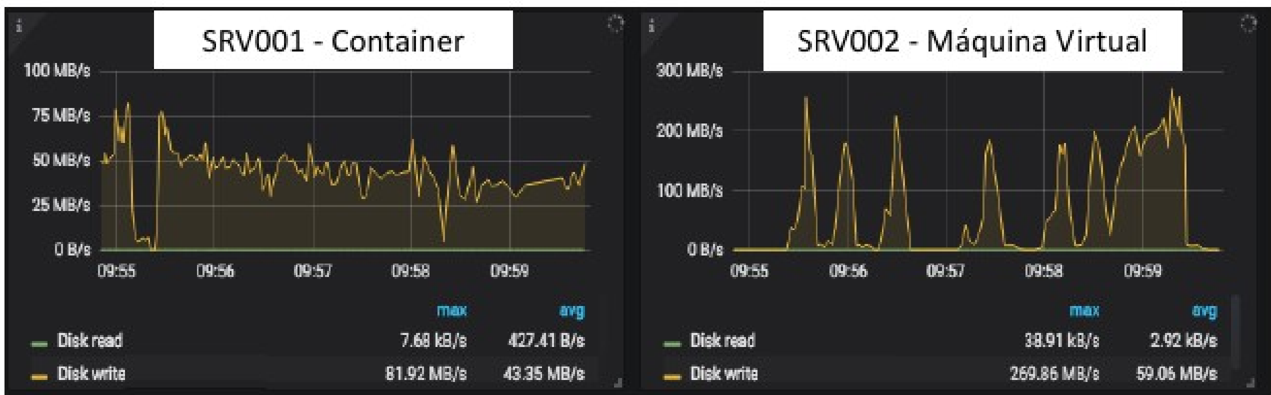
Figura 31 - Consumo de Memória RAM com o MySQL



Fonte - Adaptado de Fell (2018)

A Figura 30 apresenta uma diferença de consumo entre as duas tecnologias com o SRV001 19,79% a mais que o SRV002 (FELL, 2018)

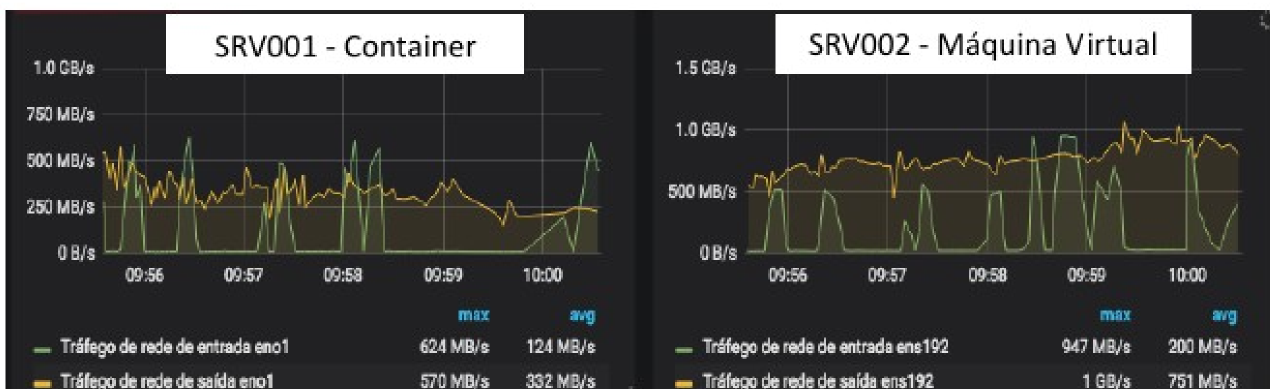
Figura 32 - Leitura e escrita de disco com o MySQL



Fonte – Adaptado de Fell (2018)

O resultado da Figura 31 demonstra que a diferença entre as duas tecnologias foi de 187,94 MB/s quando atingido o pico máximo de gravação e uma diferença média de 15,71 entre os servidores (FELL, 2018)

Figura 33 - Entrada e saída de rede com o MySQL



Fonte – Adaptado de Fell (2018)

Na Figura 32 nota-se uma diferença expressiva entre o SRV001 e SRV002 com uma variação, dentre os 5 minutos de testes, de 419 MB/s de saída e 76 MB/s de entrada. Referente ao pico máximo essa diferença chega a 430 MB/s na saída e 323 MB/s na entrada (FELL, 2018).

4.2 Teste de carga com o Apache

Com a conclusão dos testes com o serviço MySQL, Fell (2018) iniciou os testes com o Apache, definindo os parâmetros para as simulações dos testes apresentados na Figura 33. O autor decidiu que a execução dos testes seguiria com 1125 usuários efetuando 800 requisições cada um, nos servidores SRV001 e SRV002.

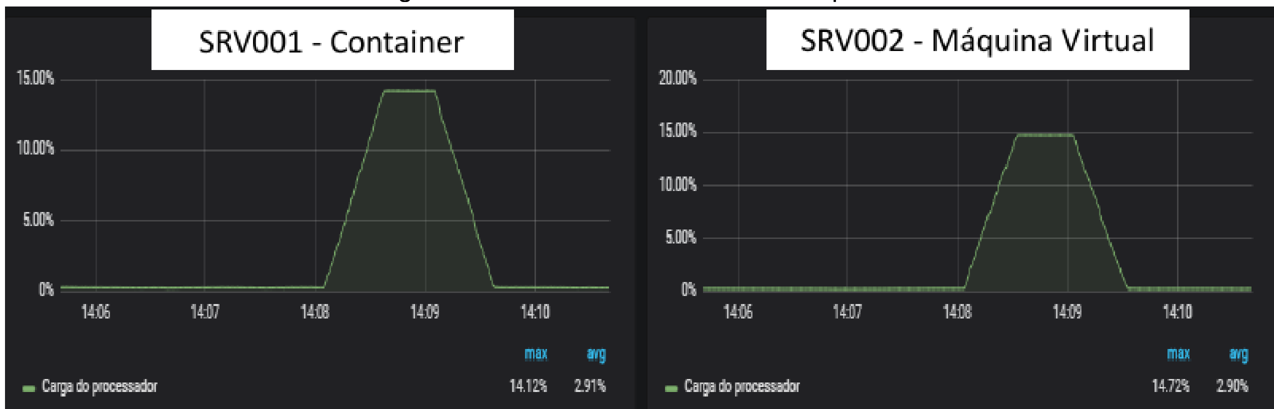
Figura 34 - Teste utilizado para a carga do Apache

Fonte – Adaptado de Fell (2018)

Fell (2018) observou que não seria possível a duração dos testes durante 5 minutos, pelo fato do servidor de coleta dos dados SRV003 não possuir uma capacidade de hardware suficiente.

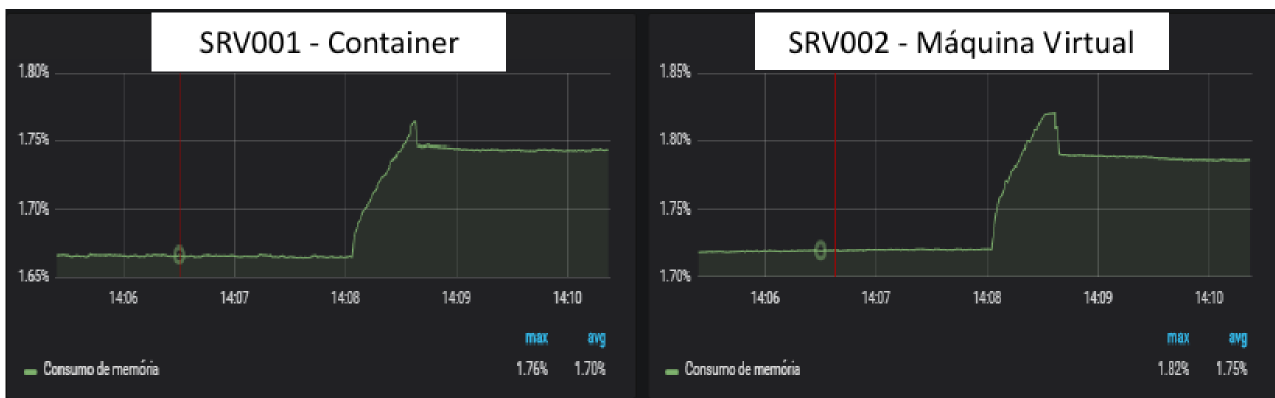
As figuras a seguir apresentam o comportamento de cada servidor.

Figura 35 - Consumo de CPU com o Apache



Fonte – Adaptado de Fell (2018)

Os testes entre os servidores, observados na Figura 34, apresentaram muita similaridade referente ao consumo de CPU, algo que ficou entre 0,01% na média de 5 minutos de coleta (FELL, 2018).



Fonte – Adaptado de Fell (2018)

A Figura 35 também apresenta um comportamento muito similar com o consumo de memória que, em 5 minutos de testes, obteve uma média de 0,05% (FELL, 2018).

Figura 37 - Leitura e escrita de disco com o Apache

Fonte – Adaptado de Fell (2018)

Conforme ilustrado na Figura 36 o teste de leitura e escrita de disco tem um resultado similar, apresentando uma diferença média na taxa de escrita de 26,08 MB/S (FELL, 2018).

Figura 38 - Entrada e saída de rede com o Apache

Fonte – Adaptado de Fell (2018)

Por fim, o teste de rede da Figura 37 apresentou uma estabilidade, no SRV002, tanto de entrada quanto de saída de dados e mesmo com essa variação, a diferença média, dentre os 5 minutos de testes, foi de 1MB/S na entrada e 67MB/s na saída (FELL, 2018).

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma análise de desempenho entre a tecnologia de conteinerização e máquinas virtuais com hipervisor do tipo 1, comparando seu consumo de recursos de hardware.

Foi utilizado o artigo do Fell (2018) como referência para obtenção dos resultados, no qual foi bastante satisfatório ao objetivo proposto.

Toda a estrutura para efetuar a análise se baseou em simular acessos simultâneos nos servidores de testes utilizando o servidor de páginas web Apache e o bando de dados MySQL. O algoritmo desenvolvido com o objetivo de realizar *benchmark* obteve êxito e possibilitou a análise através da coleta de dados realizada pelo servidor principal, utilizando a ferramenta Zabbix.

Iniciando com a análise de consumo de CPU, os testes realizados no servidor web e banco de dados apresentaram semelhança tanto no *container* quando no hipervisor.

Quanto ao teste de memória RAM o comportamento entre as duas tecnologias foram um pouco diferentes, beneficiando o hipervisor que apresentou um gerenciamento de memória mais eficiente se comparado com o *container* quando os serviços estavam em execução.

Os testes de disco apresentaram algumas variações no momento de gravação no serviço de banco de dados, no qual o hipervisor obteve uma maior velocidade de gravação se comparado com o *container.* Já no servidor web houve semelhança. Possivelmente estes testes poderiam seguir resultados diferente se um disco com a tecnologia SSD (Disco de alto desempenho) estivesse sendo utilizado.

O monitoramento de entrada e saída de rede demostrou que o hipervisor se manteve em constante estabilidade enquanto os serviços estavam em execução, se comparado com o *container.*

Referente a trabalhos futuros seria interessante utilizar o hipervisor do tipo 1 KVM, como uma solução de software livre, e a adição do dispositivo de armazenamento SSD.

REFERÊNCIAS

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

CARISSIMI, Alexandre. Virtualização: da teoria a soluções. **Minicursos do Simpósio Brasileiro de Redes de Computadores–SBRC**, v. 2008, p. 173-207, 2008.

DELGADO, Caio. **Docker - Do básico à Certificação Docker DCA**: certificação docker dca. 2021. Disponível em: https://leanpub.com/dockerdca. Acesso em: 16 abr. 2022.

Dockerfile. Disponível em: https://docs.docker.com/engine/reference/builder/. Acessado em: 21 abr. 2022.

Docker Compose. Disponível em: https://docs.docker.com/compose/compose-file/. Acessado em: 29 abr 2022.

Docker Compose CLI. Disponível em: https://docs.docker.com/compose/reference/. Acessado em: 30 abr 2022.

FELL, Gerson. **Análise de desempenho entre máquinas virtuais e containers para aplicação web.** 2018. Trabalho de Conclusão de Curso.

FELTER, et al. **An updated performance comparison of virtual machines and linux containers**. In: 2015 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, 2015. p. 171-172.

FREIRE, João Emanuel Leitão. **Orquestração de Containers Usando Kubernetes e Docker Swarm.** 2021. Tese de Doutorado.

GOMES, Rafael. **Docker para desenvolvedores**, 2017. Disponível em: https://leanpub.com/dockerparadesenvolvedores. Acesso em: 17 abr. 2022.

LANGE, Timoteo Alberto Peters. **Avaliação dos impactos de um novo paradigma de virtualização de banco de dados.** 2013. Master's Thesis. Pontifícia Universidade Católica do Rio Grande do Sul.

MAZIERO, Carlos A. **Sistemas operacionais: conceitos e mecanismos**. Livro aberto, 2014.

NICKOLOFF, Jeffrey; KUENZLI, Stephen. **Docker in action**. Simon and Schuster, 2019.

SCHEEPERS, Mathijs Jeroen. **Virtualization and containerization of application infrastructure: A comparison**. In: 21st twente student conference on IT. 2014.

SEO, Carlos Eduardo. Virtualização–Problemas e desafios. **IBM Linux Technology Center**, 2009.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de Sistemas Operacionais**. Grupo Gen-LTC, 2015.

SILVA, Flávio Henrique Rocha. **Avaliação de desempenho de Contêineres Docker para aplicações do Supremo Tribunal Federal,** 2017.

TANENBAUM, A.S.; BOS, H. **Sistemas Operacionais Modernos**. PEARSON BRASIL, 2015.

VITALINO, Jeferson Fernando Noronha; CASTRO, Marcus André Nunes. **Descomplicando o Docker 2a edição**. Brasport, 2016.

YAML. Disponível em: https://yaml.org/. Acesso em: 29 abr 2022.