



CENTRO UNIVERSITÁRIO UNICARIOCA
CIÊNCIA DA COMPUTAÇÃO

MARCOS ANTONIO SOCRATES CARVALHO DA SILVA
RAFAEL VICTOR GUSMÃO DE LIMA

SOLUÇÃO DE PROVISIONAMENTO COM DOCKER

Rio de Janeiro
2021

MARCOS ANTONIO SOCRATES CARVALHO DA SILVA
RAFAEL VICTOR GUSMÃO DE LIMA

SOLUÇÃO DE PROVISIONAMENTO COM DOCKER

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação, do Centro Universitário Carioca, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Fabio Henrique Silva
Coordenador: André Luiz Avelino Sobral

Rio de Janeiro
2021

S586s Silva, Marcos Antonio Socrates Carvalho da
Solução de provisionamento com Docker / Marcos Antonio Socrates
Carvalho da Silva e Rafael Victor Gusmão de Lima. - Rio de Janeiro, 2021.
62 f.

Orientador: Fábio Henrique Silva
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) –
Centro Universitário UniCarioca - Rio de Janeiro, 2021.

1. Docker. 2. Contêiner. 3. Virtualização. 4. Orquestrador. I. Lima, Rafael
Victor Gusmão de. II. Silva, Fábio Henrique, prof. orient. III. Título.

CDD 005.1

MARCOS ANTONIO SOCRATES CARVALHO DA SILVA
RAFAEL VICTOR GUSMÃO DE LIMA

SOLUÇÃO DE PROVISIONAMENTO COM DOCKER

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação, do Centro Universitário Carioca, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Rio de Janeiro, 27 de maio de 2021

BANCA EXAMINADORA

Prof. Fabio Henrique Silva, M.Sc. – Orientador

Prof. André Luiz Avelino Sobral, M.Sc. – Coordenador

Prof. Rogério Malheiros dos Santos, D.Sc. – Convidado

Dedicamos nosso trabalho ao orientador, o Professor Fabio Henrique Silva, cuja experiência foi inestimável na formulação das questões de pesquisa e metodologia. Seu feedback nos impulsionou a aguçar o pensamento em busca do objetivo do trabalho.

AGRADECIMENTOS

Gostaríamos de agradecer aos nossos familiares, pelos conselhos sábios, pelos ouvidos solidários, compreensão, bem como distrações felizes para descansar nossa mente fora da construção desse trabalho.

RESUMO

Os projetos de software podem ter, hoje em dia, infraestruturas extremamente complexas em seu contexto, na forma de bibliotecas e várias outras dependências que precisam ser instaladas nas máquinas em que estão sendo desenvolvidos. Configurar essa infraestrutura em uma nova máquina, manualmente, pode ser um processo tedioso sujeito a erros. Isso pode ser evitado pela automatização do processo usando uma ferramenta de provisionamento de software, que pode transferir automaticamente a infraestrutura entre máquinas com base em instruções que podem ser controladas por versão de maneiras semelhantes ao código-fonte. O Docker é uma ferramenta que pode fazer isso. Ele encapsula projetos em contêineres. Este estudo tem como objetivo analisar três tipos de cenários. O primeiro, sem o sistema de provisionamento em um ambiente local com utilização de servidores físicos (bare-metal), o segundo, com a implantação do Docker e o terceiro, utilizando Docker Swarm com uma ferramenta de escalonamento automática. Ao final do experimento, serão apresentadas as análises com as vantagens de uso das soluções.

Palavras-chave: Docker. Contêiner. Virtualização. Orquestrador.

LISTA DE ILUSTRAÇÕES

Figura 1 — Arquitetura de uma máquina virtual	15
Figura 2 — Comparação das arquiteturas VM e Docker	16
Figura 3 — Camadas de uma imagem Docker	19
Figura 4 — Criação de imagem e contêiner a partir de um <i>dockerfile</i>	19
Figura 5 — Imagens Docker Hub	21
Figura 6 — Ambiente bare-metal ou VM.	24
Figura 7 — Instalação e verificação dos pacotes	25
Figura 8 — Apache HTTPD.....	26
Figura 9 — Integração Apache e PHP	27
Figura 10 — Aplicação PHP	27
Figura 11 — Configuração do MySQL	28
Figura 12 — Aplicação funcional.....	29
Figura 13 — Instalação do Docker	30
Figura 14 — MySQL Dockerfile (adaptado)	32
Figura 15 — Apache + PHP Dockerfile (adaptado)	32
Figura 16 — Criação das imagens Docker	33
Figura 17 — Docker Hub login	34
Figura 18 — Docker push.....	35
Figura 19 — Imagens no Docker Hub	35
Figura 20 — Docker pull.....	36
Figura 21 — Docker run	37
Figura 22 — Cenário Docker Swarm.....	38
Figura 23 — Swarm join-token	39
Figura 24 — Lista dos nós e funções Docker Swarm.....	39
Figura 25 — Arquivo compose da aplicação	40
Figura 26 — Implantação do serviço com stack deploy	41
Figura 27 — Docker Swarm	42
Figura 28 — Docker Swarm apenas local	43
Figura 29 — Docker Swarm apenas AWS	43

LISTA DE ABREVIATURAS E SIGLAS

APT	Advanced Packaging Tool
AWS	Amazon Web Services
CPU	Central Process Unit ou Unidade Central de Processamento
IDE	Integrated Development Environment ou Ambiente de Desenvolvimento Integrado
NAT	Network Address Translation
PHP	Hypertext Preprocessor, originalmente Personal Home Page
SO	Sistema Operacional
SQL	Standard Query Language
TCC	Trabalho de Conclusão de Curso
TI	Tecnologia da Informação
VM	Máquina virtual (VM, pela sigla em inglês)
API	Interface de programação de aplicações (API, pela sigla em inglês)

SUMÁRIO

1	INTRODUÇÃO	9
1.1	VISÃO GERAL	9
1.2	MOTIVAÇÃO	9
1.3	JUSTIFICATIVA	10
1.4	OBJETIVO	11
1.5	METODOLOGIA	12
1.6	ORGANIZAÇÃO DO TRABALHO	12
2	CONTEXTUALIZAÇÃO	14
2.1.1	Bare-metal	14
2.1.2	Máquinas virtuais	14
2.1.3	Contêineres	15
3	DOCKER	17
3.1	O QUE É DOCKER?	17
3.2	IMAGENS DOCKER	17
3.3	DOCKER HUB	20
3.4	DOCKER SWARM	21
4	METODOLOGIA	23
4.1	AMBIENTE EM SERVIDOR BARE-METAL	23
4.1.1	Pré-requisito	24
4.1.2	Instalação dos pacotes MySQL, Apache2 e PHP	25
4.1.3	Integração entre o Apache2 e o PHP	26
4.1.4	Configuração do banco de dados MySQL	28
4.1.5	Integração entre PHP e MySQL	29
4.2	AMBIENTE COM DOCKER	29
4.2.1	Instalação do Docker	30
4.2.2	Criação da imagem Docker personalizada	31
4.2.3	Compartilhamento da imagem criada no Docker Hub	33
4.2.4	Execução do contêiner	36
4.3	AMBIENTE COM DOCKER SWARM	37
4.3.1	Construção do cluster	39
4.3.2	Implantação do serviço	41
4.4	ANÁLISE DOS CENÁRIOS	44
4.4.1	AMBIENTE EM SERVIDOR BARE-METAL	44
4.4.2	AMBIENTE COM DOCKER	45
4.4.3	AMBIENTE COM DOCKER SWARM	46
4.5	DESAFIOS E COMPLEXIDADE	47
5	CONCLUSÃO	49
6	REFERÊNCIAS	51
7	GLOSSÁRIO	53

1 INTRODUÇÃO

1.1 VISÃO GERAL

Com o início da terrível pandemia de COVID-19 o mundo necessitou implementar mudanças drásticas na forma de convivência e nas relações interpessoais. Tudo precisou ser modificado, desde o passeio por lazer até a tão agitada jornada de trabalho, com isso, novas formas de comportamentos e rotinas tiveram de ser aplicadas em todo mundo, uma destas rotinas é a do isolamento social. O isolamento social, fez com que toda cadeia produtiva, principalmente o comércio tivesse que se readaptar a uma nova forma de trabalho, para assim conseguir seu meio de se manter ativo. Dentre essas novas formas estão por exemplo, serviços de *delivery*, lojas virtuais e o *home office*. As empresas de TI (Tecnologia da Informação) mais voltadas em áreas de desenvolvimento, atualmente empregam a forma *home office* para conseguir contornar o problema da pandemia e ao mesmo tempo continuar sua produção de modo integral, quando comparado ao período que o trabalho era presencial. Porém a partir deste ponto é encontrado um problema de compatibilidade.

1.2 MOTIVAÇÃO

Quando um projeto é iniciado localmente em uma máquina, todos os serviços e dependências são instalados apenas nessa máquina. Segundo Pavlovic e Atkins (2020), podem surgir problemas quando o projeto precisa ser movido para uma nova máquina. A partir do momento que o projeto é instalado em um novo computador, todas as dependências devem ser instaladas manualmente, o que pode ser um processo longo e tedioso. Um fluxo de trabalho manual também pode levar a dependências em relação a pessoas específicas, pois pode surgir uma situação em que apenas alguns desenvolvedores sabem como conduzir o processo de instalação. A saída de funcionários (desenvolvedores) com esse conhecimento pode causar problemas para a empresa. Ainda de acordo com Pavlovic e Atkins (2020), as instalações manuais também podem ser uma fonte de problemas, pois as dependências podem ser negligenciadas, resultando em uma configuração

incompleta. Isso representa um risco de *bugs* difíceis, mal-entendidos ou *pachtes* de biblioteca e serviço.

As instalações devem ser realizadas sempre que o ambiente de desenvolvimento for instalado em uma nova máquina. Isso pode ser, por exemplo, ao contratar uma nova equipe ou consultores, ou quando o computador de um desenvolvedor é atualizado ou falha. Da mesma forma, quando a infraestrutura é alterada, a mudança deve ser feita em todas as máquinas de desenvolvimento, o que também é um processo que se beneficiaria por ser automatizado de forma semelhante ao controle de versão do código. Usando o provisionamento baseado em contêiner, a síndrome “funciona na minha máquina” pode ser evitada.

1.3 JUSTIFICATIVA

O *Docker* é uma ferramenta que pode ser usada para provisionar uma infraestrutura, encapsulando em uma estrutura virtual chamada contêiner. (Docker).

Existem também ferramentas usadas para Gerenciamento de Configuração. Essas ferramentas são mais adequadas para automatizar *scripts*, enquanto as ferramentas de provisionamento são mais adequadas para automatizar instalações de infraestruturas inteiras. O *Docker* é usado em um ambiente fechado, o que facilita a migração do ambiente entre as máquinas locais. No entanto, as ferramentas de gerenciamento de configuração podem ser usadas como um complemento para uma solução.

As ferramentas de gerenciamento de configuração não são, portanto, de interesse, pois são projetadas para manter as configurações, não para defini-las. O *Docker* foi selecionado entre outras ferramentas de provisionamento, por ser uma ferramenta popular amplamente utilizada no setor. Isso se deve a vários motivos, incluindo sua eficiência, escalabilidade e portabilidade.

Além disso, os serviços utilizados nos contêineres devem tolerar falhas, ser escaláveis sobre demanda, otimizar recursos, suportar atualizações ou reversões sem qualquer tempo de inatividade. Para isso o *Docker Swarm* fornece essa estrutura de orquestração de contêiner. Neste Trabalho de Conclusão de Curso, em um dos cenários propostos também será utilizado o *Docker Swarm*. O modo *Docker*

Swarm é o mais recente participante em um grande campo de sistemas de orquestração de contêineres, aqui, ele será usado em conjunto com servidores hospedados na *Amazon Web Services* (AWS), uma plataforma de serviços de computação em nuvem, com o objetivo de distribuir uma aplicação web entre um pool de servidores *Apache* com *PHP* e um servidor *MySQL* na nuvem em alta disponibilidade (ROHRER, 2017).

1.4 OBJETIVO

O objetivo deste estudo é analisar a implantação de uma aplicação web comum, baseada em *PHP*, sendo executada por um servidor *Apache* utilizando conexão com banco de dados *MySQL*, numa suposta empresa em diferentes cenários, sendo esses:

- Simulação da aplicação em servidor *bare-metal* ou máquina virtual;
- Simulação da aplicação com o *Docker*;
- Simulação da aplicação com o *Docker Swarm*.

Portanto, as seguintes questões de pesquisa são centrais para atender ao objetivo deste relatório:

- Quais os problemas e dificuldades encontradas na implantação da aplicação em servidores *bare-metal* ou máquinas virtuais?
- Quais os desafios e as vantagens da utilização de *Docker* comparado a um ambiente em servidores *bare-metal* ou máquinas virtuais?
- Quais benefícios da utilização do *Docker* com *clusters* de alta disponibilidade?

Os resultados da análise entre os cenários podem ser valiosos e auxiliam na tomada de decisão visando o melhor custo-benefício para o negócio. A equipe de desenvolvedores poderá escolher uma estrutura para implantar soluções semelhantes para reduzir o tempo de instalação e propagar as alterações de forma eficiente para outros membros da equipe, o que permite um processo de trabalho mais eficiente.

1.5 METODOLOGIA

O estudo se concentrará na simulação prática dos cenários com o provisionamento da aplicação PHP/MySQL e na análise das soluções propostas, especificamente, as características, vantagens e desvantagens de cada cenário, para implementar um ambiente de desenvolvimento em uma nova máquina, bem como o tempo gasto para o provisionamento, gestão de possíveis erros no processo de inserção da plataforma Docker na aplicação PHP/MySQL e interação entre máquinas rodando servidores apache com um servidor MySQL de alta disponibilidade na AWS. O estudo será realizado numa máquina executando sistema operacional (SO) Linux. É importante notar que o Docker usa recursos do kernel Linux internamente, e que os SO utilizados em testes ou estudos posteriores podem produzir resultados diferentes (Agarwal, 2017).

Além de apontar diferenças entre os cenários, buscou-se arduamente após a apresentação deste trabalho de conclusão de curso, mostrar alternativas, lúcidas e robustas, para equipes de desenvolvedores que utilizam o home office para a forma de desenvolvimento, objetivando-se principalmente a economia de tempo considerável por evitar um longo caminho de incompatibilidade, erros, e outras adversidades que apareceriam naturalmente sem as soluções propostas por este trabalho.

1.6 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está distribuído em cinco capítulos que abordam os principais temas, organizados da seguinte maneira:

O Capítulo 1 (Introdução) apresenta a motivação por trás do trabalho, a declaração do problema, a maneira detalhada como foi conduzido e discute possíveis impactos.

No Capítulo 2 (Conceituação) serão explicados os conceitos das tecnologias que fornecem a base para a análise das soluções no que diz respeito a servidores *bare-metal*, máquinas virtuais e contêineres.

No Capítulo 3 (Docker) são abordados os principais componentes utilizados pela ferramenta de contêiner escolhida.

O Capítulo 4 (Metodologia) apresentará a execução prática dos diferentes ambientes de provisionamento da aplicação e, em seguida, será apresentada uma análise dos cenários, bem como os desafios e complexidade no desenvolvimento.

O Capítulo 5 (Conclusão) serão destacadas as conclusões encontradas com base na análise das construções práticas dos cenários e possíveis trabalhos futuros que podem ser realizados.

2 CONTEXTUALIZAÇÃO

2.1.1 Bare-metal

“A expressão “Bare Metal” (metal nu ou metal puro, em inglês) serve é usada para descrever ambientes de TI em que o sistema operacional é instalado diretamente no hardware, em vez de uma camada de sistema hospedando diversas VM, como é realizado em ambientes virtualizados.” (Under Control, 2017)

Normalmente, são utilizados quando houver cargas de trabalho com muitos dados que priorizam o desempenho e a confiabilidade. (Mincov, 2014)

2.1.2 Máquinas virtuais

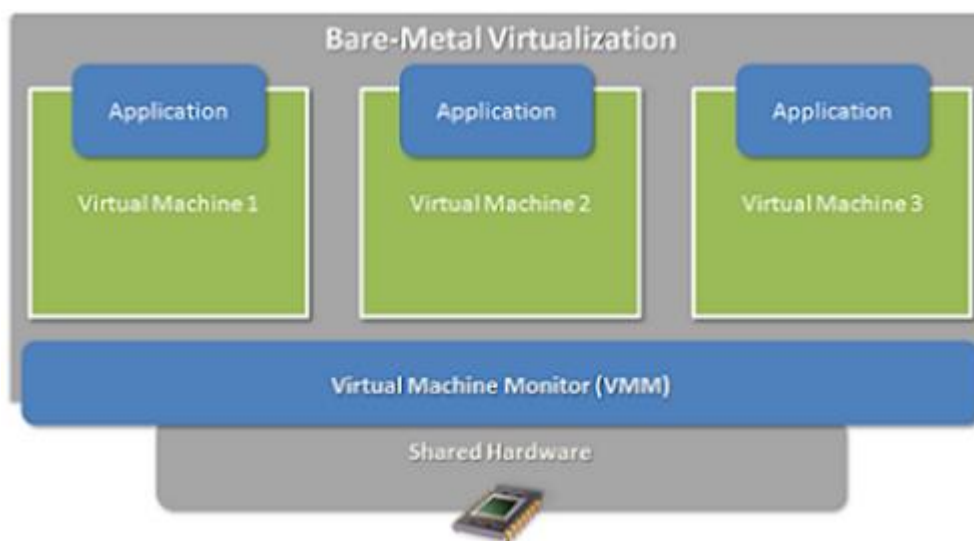
"Uma máquina virtual, normalmente reduzida apenas para VM, não é diferente de qualquer outro computador físico, como um laptop, um smartphone ou um servidor. Ela tem CPU, memória, discos para armazenar seus arquivos e pode se conectar à Internet, se necessário." (Microsoft Azure, 2021)

Embora as partes que compõem o computador (chamadas de *hardware*) sejam físicas e tangíveis, as VM costumam ser consideradas como computadores virtuais ou computadores definidos por software em servidores físicos, existindo apenas como código.

Historicamente, conforme o poder de processamento e a capacidade do servidor aumentavam, os aplicativos que utilizaram servidores *bare-metal* (Os servidores *bare-metal* são conhecidos também como servidores físicos de inquilino único ou servidores dedicados. Em um servidor *bare-metal*, o sistema operacional é instalado diretamente no servidor, eliminando camadas e proporcionando melhor desempenho) não eram capazes de explorar a nova abundância de recursos. Assim, nasceram as VM, projetadas executando *software* em cima de servidores físicos para emular um sistema de *hardware* específico. Um *hipervisor*, ou monitor de máquina virtual, é um *software*, firmware ou *hardware* que cria e executa VM. É o que fica entre o *hardware* e a máquina virtual e é necessário para virtualizar o servidor. (NetApp, 2018)

Em cada máquina virtual é executado um sistema operacional convidado exclusivo. VM com sistemas operacionais diferentes podem ser executados no mesmo servidor físico - uma VM *UNIX* pode ficar ao lado de uma VM *Linux* e assim por diante. Cada VM tem seus próprios binários, bibliotecas e aplicativos que atende, e a VM pode ter muitos *gigabytes* de tamanho. (NetApp, 2018)

Figura 1 — Arquitetura de uma máquina virtual



Fonte: IXC Soft

2.1.3 Contêineres

A virtualização do SO cresceu em popularidade na última década para permitir que o *software* seja executado de maneira previsível e adequada quando movido de um ambiente de servidor para outro. Mas os contêineres fornecem uma maneira de executar esses sistemas isolados em um único servidor ou sistema operacional *host*.

Os contêineres são uma forma de virtualização do sistema operacional. Um único contêiner pode ser usado para executar qualquer coisa, desde um pequeno micro serviço ou processo de software até um aplicativo maior. (NetApp, 2021)

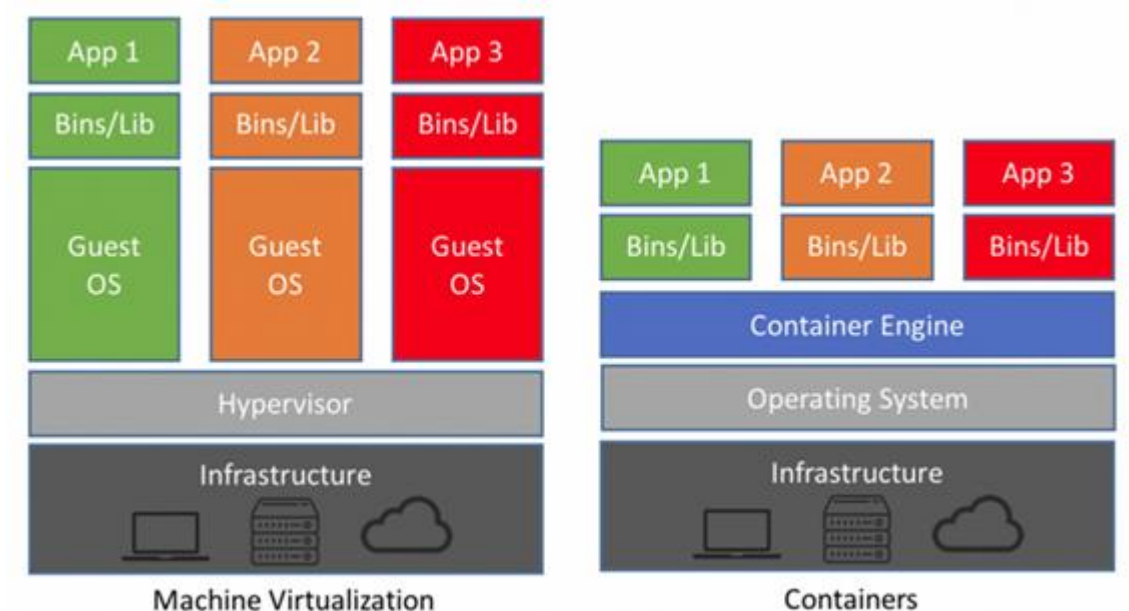
Dentro de um contêiner estão todos os executáveis, código binário, bibliotecas e arquivos de configuração necessários. Em comparação com as abordagens de virtualização de servidor ou máquina, no entanto, os contêineres não contêm imagens do sistema operacional. (NetApp, 2021)

Isso os torna mais leves e portáteis, com significativamente menos sobrecarga. Em implementações de aplicativos maiores, vários contêineres podem ser implementados como um ou mais clusters de contêiner. Esses clusters podem ser gerenciados por um orquestrador de contêineres, na aplicação será usado como orquestrador o Docker Swarm, que será descrito com mais detalhes à frente.

Os contêineres ficam em cima de um servidor físico e seu sistema operacional *host* - por exemplo, *Linux* ou *Windows*. Cada contêiner compartilha o *kernel* do sistema operacional *host* e, geralmente, os binários e bibliotecas também. Os componentes compartilhados são somente leitura. (NetAapp, 2018)

Os contêineres também reduzem a sobrecarga de gerenciamento. Como eles compartilham um sistema operacional comum, apenas um único sistema operacional precisa de cuidados e alimentação para correções de *bugs*, atualizações e assim por diante. Esse conceito é semelhante ao que experimentamos com *hosts* de *hipervisor*, menos pontos de gerenciamento, mas domínio de falha um pouco mais alto. Resumindo, os contêineres são mais leves e mais portáteis do que as VM. (NetAapp, 2018)

Figura 2 — Comparação das arquiteturas VM e Docker



Fonte: AquaSec

3 DOCKER

3.1 O QUE É DOCKER?

O *Docker* é uma plataforma aberta para desenvolvimento, envio e execução de aplicações. Ele permite que você separe suas aplicações de sua infraestrutura para que possa entregar o *software* de forma mais rápida. Com o *Docker*, pode-se gerenciar uma infraestrutura da mesma forma que gerencia aplicações. Tirando proveito das metodologias do *Docker* para envio, teste e implantação de código mais rapidamente, assim você pode reduzir significativamente o atraso entre escrever o código e executá-lo na produção (DOCKER, 2021).

O *Docker* oferece a capacidade de empacotar e executar uma aplicação em um ambiente vagamente isolado denominado contêiner. O isolamento e a segurança permitem que você execute vários contêineres simultaneamente em um determinado *host*. Os contêineres são leves e contêm tudo o que é necessário para executar a aplicação, portanto, você não precisa depender do que está instalado atualmente no *host*. Pode-se compartilhar contêineres facilmente enquanto trabalha e certificar-se que todas as pessoas com quem foi compartilhado receberam o mesmo contêiner. O *Docker* fornece ferramentas e uma plataforma para gerenciar o ciclo de vida de seus contêineres.

3.2 IMAGENS DOCKER

Uma imagem *Docker* é um arquivo que contém dados e informações necessários para criar um grupo de processos com propriedades definidas. Uma imagem da janela de encaixe é composta de camadas somente leitura. Cada camada corresponde a um comando executado durante a criação de uma imagem. Isso torna a criação de uma imagem altamente personalizável, já que imagens pré-existentes podem ser adaptadas ou construídas para servir a um propósito ligeiramente diferente. As informações que uma imagem contém variam de quais portas de rede devem estar disponíveis para quais programas devem ser

executados durante a execução da imagem *Docker*. Quando executado, um ambiente bem isolado denominado contêiner é criado com base nas especificações do arquivo de imagem. As imagens do *Docker* são facilmente compartilháveis, o que torna possível criar ambientes idênticos em diferentes sistemas operacionais, justamente o que se tem como objetivo neste trabalho. A vantagem de ser capaz de executar um determinado contêiner em qualquer sistema operacional é que os aplicativos em sistemas diferentes não encontrarão nenhum problema de compatibilidade, pois estão sendo executados dentro de um ambiente isolado criado a partir da imagem *Docker*.

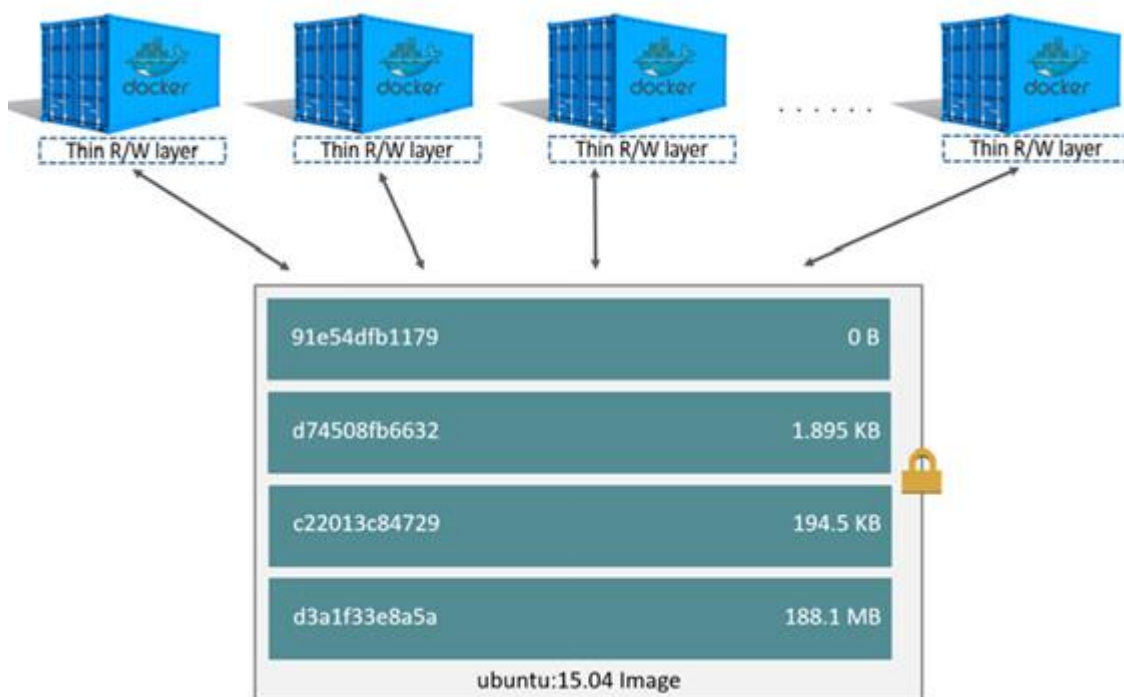
Uma imagem *Docker* é composta de uma coleção de arquivos que agrupam todos os itens essenciais - como instalações, código do aplicativo e dependências - necessários para configurar um ambiente de contêiner totalmente operacional. É possível criar uma imagem *Docker* usando um dos dois métodos:

1. O método interativo: Fazendo a execução de um contêiner a partir de uma imagem prévia já existente, sendo assim, necessitando a alteração manualmente de todos os aspectos desejados pelo desenvolvedor um por um até que este próprio vire uma imagem ideal para este desenvolvedor.
2. Utilizando um *Dockerfile*: Mais à frente no próximo item desse capítulo serão descritos os *Dockerfiles*. Segundo Romero (2015), *Dockerfile* é um script, composto de vários comandos (instruções) e argumentos listados sucessivamente para executar ações automaticamente em uma imagem de base para criar (ou formar) uma nova. Sendo por meio deste é possível utilizar linhas de comando categoricamente fáceis que fornecem especificações diretas para a criação de uma imagem de acordo com o que o desenvolvedor deseja.

De acordo com Rotsaert (2019), "uma imagem Docker consiste em várias camadas. Cada camada corresponde a certas instruções em seu *Dockerfile*. Cada um dos arquivos que compõem uma imagem Docker é conhecido como uma camada. Essas camadas formam uma série de imagens intermediárias, construídas umas sobre as outras em etapas, em que cada camada depende da camada imediatamente abaixo dela. A hierarquia de suas camadas é a chave para o gerenciamento eficiente do ciclo de vida de suas imagens Docker. Portanto, você deve organizar as camadas que mudam com mais frequência o mais alto possível na pilha. Isso ocorre porque, quando você faz alterações em uma camada da imagem, o Docker

não apenas reconstrói essa camada específica, mas todas as camadas criadas a partir dela. Portanto, uma mudança em uma camada no topo de uma pilha envolve o mínimo de trabalho computacional para reconstruir a imagem inteira."

Figura 3 — Camadas de uma imagem Docker

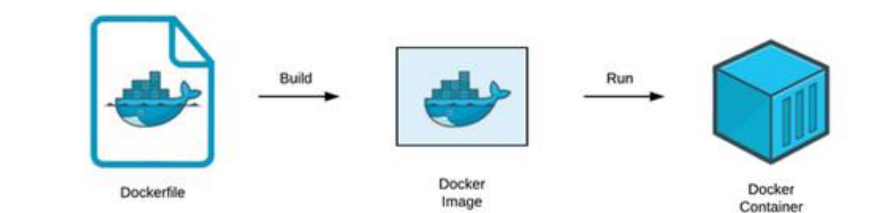


Fonte: Docker

Dockerfile é o conceito básico para a construção de imagens *Docker*, e é um arquivo de texto que contém uma lista de comandos que descreve como uma imagem *Docker* é construída com base nos mesmos. O comando "*docker build*" diz ao *Docker* para construir a imagem seguindo o conteúdo dentro do *Dockerfile*.

O arquivo começa com um comando "*FROM*" que indica a imagem de base. Os comandos subsequentes no arquivo *Docker* são executados na imagem base, que deve ser uma imagem válida.

Figura 4 — Criação de imagem e contêiner a partir de um *dockerfile*



Fonte: Chen

Após a elucidação do conceito à cerca de um *Dockerfile*, será dada uma breve introdução aos comandos de um *Dockerfile*, este conceito será explorado largamente no capítulo 4, onde será estudado a criação do *Dockerfile* utilizado nesta aplicação, sendo contido nele, um Servidor *Apache PHP* e um servidor *MySQL*, neste subitem será esclarecido os principais comandos *Dockerfile* utilizado na aplicação deste estudo, são eles:

- *FROM* - Instrução obrigatória que indica qual imagem vai ser utilizada como ponto de partida.
- *RUN* - Serve para executar comandos no processo de montagem da imagem que estamos construindo no *Dockerfile*, ele é executado durante o *build* (construção da imagem) e não durante a construção do contêiner. Em um *Dockerfile* é possível ter mais de um comando *RUN*.
- *ENV* - Serve para definir variáveis de ambiente, você pode tanto deixar essas variáveis setadas de forma fixa dentro do *Dockerfile* quanto passá-las dinamicamente na hora que você instanciar o container.
- *VOLUME* - Quando adicionamos *VOLUME* no *dockerfile* estamos informando um ponto de montagem, criando uma pasta que ficará disponível entre o contêiner e o *host*.
- *COPY* - Copia arquivos e pastas para dentro da imagem do *Docker*.
- *ENTRYPOINT* - Com esse parâmetro é possível definir se algo deve ser executado na hora da instanciação do contêiner.
- *EXPOSE* - Instrução que informa qual porta deverá ser liberada. *EXPOSE* não é um comando que libera a porta, ele serve para ficar documentado no *Dockerfile* quais portas deverão ser liberadas ao criar o contêiner.
- *CMD* - Diferente do *RUN*, o *CMD* executa apenas na criação do contêiner e não no *build* da imagem. Deve ser único no *Dockerfile*.

3.3 DOCKER HUB

O *Docker Hub* é um repositório em nuvem no qual os usuários e parceiros do *Docker* criam, testam, armazenam e distribuem imagens de contêiner. Por meio do

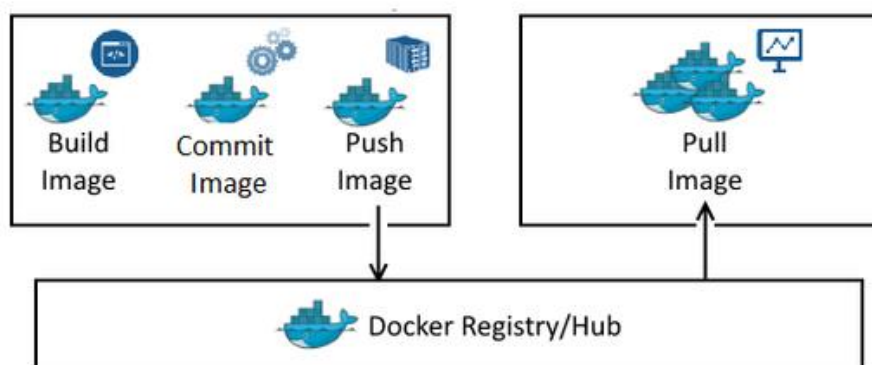
Docker Hub, um desenvolvedor pode acessar repositórios de imagens públicos e de código aberto, bem como usar um espaço para criar seus próprios repositórios privados, funções de construção automatizadas e grupos de trabalho.

Por exemplo, um profissional *DevOps* pode baixar a imagem de contêiner do sistema de gerenciamento de banco de dados relacional de objeto *MySQL* oficial do *Docker Hub* para usar em um aplicativo implantado em contêineres. Ou eles podem escolher um *RDBMS* personalizado no repositório privado de sua empresa.

Neste trabalho, estamos utilizando, como exemplo, uma empresa que possui uma aplicação utilizando *PHP* e *MySQL*, no qual sua equipe de desenvolvimento através da modalidade de trabalho remoto pretende utilizar a ferramenta do *Docker* para vivenciar um ambiente prático e livre de erros de compatibilidade e *bugs* por meio de imagens contidas no repositório.

Docker Hub é uma versão hospedada em nuvem do *Docker Registry*. Um usuário do *Docker* pode optar pelo *Docker Registry*, que é um aplicativo do lado do servidor sem estado, código-fonte aberto e escalonável, se preferir manter o armazenamento e a distribuição das imagens do *Docker* em vez de depender do serviço do *Docker*.

Figura 5 — Imagens Docker Hub



Fonte: Raj (2018)

3.4 DOCKER SWARM

De acordo com Romero (2015), o *Docker Swarm* é um sistema de *cluster* nativo para *Docker* que utiliza a *API* padrão.

Um *Docker Swarm* é um grupo de máquinas físicas ou virtuais que estão executando o aplicativo *Docker* e que foram configuradas para serem unidas em um *cluster*. Depois que um grupo de máquinas foi agrupado, você ainda pode executar os comandos do *Docker* com os quais está acostumado, mas agora eles serão executados pelas máquinas em seu *cluster*. As atividades do *cluster* são controladas por um gerenciador de enxame e as máquinas que se juntaram ao *cluster* são chamadas de nós.

Para que é usado o *Docker Swarm*?

Docker Swarm é uma ferramenta de orquestração de contêineres, o que significa que permite ao usuário gerenciar vários contêineres implantados em várias máquinas *host*.

Um dos principais benefícios associados à operação de um *Docker swarm* é o alto nível de disponibilidade oferecido para os aplicativos. Em um *Docker swarm*, normalmente há vários nós de trabalho e pelo menos um nó de gerenciador que é responsável por lidar com os recursos dos nós de trabalho de forma eficiente e garantir que o *cluster* opere com eficiência.

4 METODOLOGIA

Mostrado na introdução, o objetivo e problemática, respectivamente à cerca deste TCC, neste capítulo será abrangido toda parte prática com possíveis soluções de como pode ser alcançado um sucesso em tempo e custo computacional utilizando a ferramenta *Docker* nos cenários aqui mostrados.

Neste capítulo, será realizado um laboratório com simulação prática dos três cenários apresentados. Os cenários apresentam soluções progressivas que podem fornecer soluções de melhoria de *DevOps* de uma companhia. Após muita análise de todas as ferramentas oferecidas pelo *Docker*, chegou-se à conclusão que neste caso específico onde a problemática central é a resolução de problemas relacionados ao *home-office*, o que consequentemente descreve-se como possíveis falhas de compatibilidade entre computadores em diferentes SO, *bugs* de *IDE* para *IDE*, erros constantes de versionamento, dentre os desenvolvedores de uma possível empresa, aqui será mostrado como fazer a utilização do *Docker* em dois cenários, e uma simulação de um cenário local, sendo estes, o primeiro cenário em um ambiente local, um segundo cenário com a utilização de contêineres *Docker* e finalmente um cenário onde o foco é a alta disponibilidade, fazendo uso de *Docker Swarm* e *AWS*, gerando *clusters* de alta disponibilidade.

Para a simulação de todos os cenários, um ambiente de testes virtualizado foi configurado com as seguintes configurações:

- Processador: Intel Core i7-6500U
- Memória RAM: 16GB DDR4
- Armazenamento: SSD 512GB
- Sistema operacional: Ubuntu 20.04.2 LTS
- Virtualização: VMware Workstation Pro e AWS (Amazon Web Services)

4.1 AMBIENTE EM SERVIDOR BARE-METAL

Nesse cenário, tomou-se como base um ambiente cotidiano de trabalho presencial, no qual o desenvolvedor utilizará a máquina de trabalho para

desenvolver uma aplicação em um servidor *bare-metal* em um ambiente local. Será apresentado do início, o processo de implantação de uma aplicação com múltiplas tecnologias com tempo gasto, principais vantagens e desvantagens.

Figura 6 — Ambiente bare-metal ou VM.



Fonte: Os autores (2021)

Para iniciar, algumas ferramentas foram utilizadas para o desenvolvimento do ambiente. Elas serão descritas a seguir:

- *Visual Studio Code* - Essa *IDE* foi utilizada para o desenvolvimento do site na linguagem *PHP* de modo a facilitar todo processo de escrita e execução do código. A ferramenta também foi utilizada para realizar o acesso remoto via terminal (*SSH*) aos *hosts*.
- *Apache HTTP Server* com *PHP* - Essas ferramentas foram utilizadas como servidor *web* para disponibilização da aplicação *PHP*. Optou-se pelo *Apache* por ser o serviço *web* mais utilizado no mundo, além de ser *open source*, o que por si só já justifica essa escolha.
- *MySQL Community Server* - O servidor de banco de dados *MySQL* foi utilizado em conjunto com o *MySQL WorkBench* para implantar o banco de dados utilizado pela aplicação *PHP*. Ele foi selecionado por possuir diversas vantagens quando comparado a outras ferramentas, são elas, ser *Open Source*, multiplataforma, pouco custo computacional, dentre outras.

4.1.1 Pré-requisito

Como pré-requisito para implantação da aplicação em um novo servidor *bare-metal* ou *VM*, é necessária a instalação de um sistema operacional. Para este trabalho, optou-se pela utilização do *Ubuntu Server 20.04 LTS*. Por não se tratar do escopo desse trabalho, não será demonstrado o processo de instalação do *SO*.

Nesta etapa, é importante observar o tempo de instalação do sistema operacional, que leva em torno de 15 a 20 minutos, dependendo das configurações de processamento do servidor.

4.1.2 Instalação dos pacotes MySQL, Apache2 e PHP

Com o SO instalado, o primeiro passo é realizar a instalação dos pacotes que compõem a aplicação. A instalação dos pacotes é relativamente simples, porém com o passar do tempo, novas versões podem ser lançadas, acarretando problemas de compatibilidade. A instalação dos pacotes dos serviços *MySQL*, *Apache 2 HTTPD* *PHP* com suas respectivas dependências foi realizada através do gerenciador de pacotes do *Ubuntu*, o *APT (Advanced Packaging Tool)*, e verificada conforme a figura 7 abaixo. O processo instalação foi realizado em aproximadamente dois minutos.

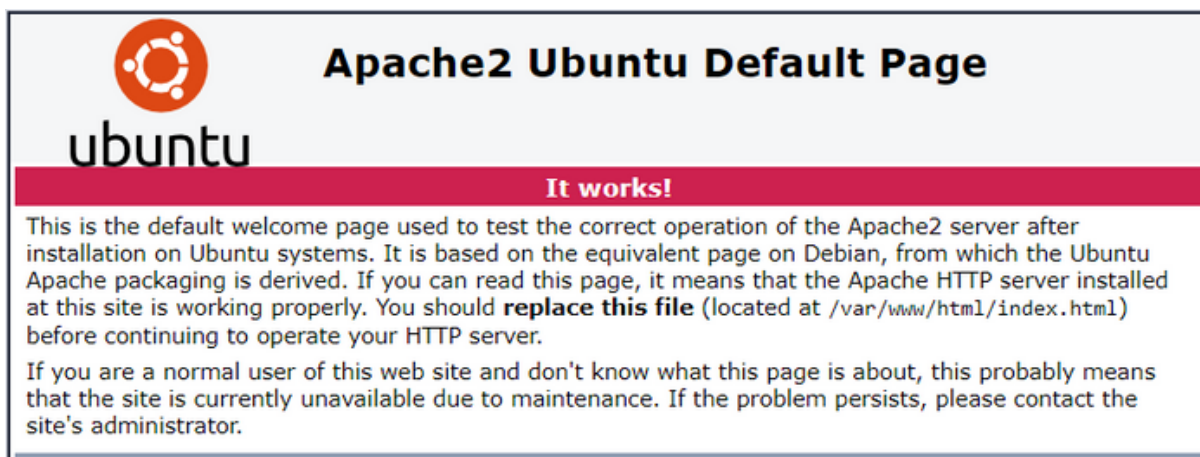
Figura 7 — Instalação e verificação dos pacotes

```
ubuntu@bare-metal:~$ sudo apt-get update
ubuntu@bare-metal:~$ sudo apt-get install apache2
ubuntu@bare-metal:~$ systemctl status apache2.service
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-06-06 14:24:09 UTC; 5min ago
ubuntu@bare-metal:~$ sudo apt-get install mysql-server
ubuntu@bare-metal:~$ systemctl status mysql.service
● mysql.service - MySQL Community Server
   Loaded: loaded (/lib/systemd/system/mysql.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-06-06 14:44:15 UTC; 36s ago
ubuntu@bare-metal:~$ sudo apt-get install php
ubuntu@bare-metal:~$ dpkg --status php
Package: php
Status: install ok installed
Priority: optional
Section: php
Installed-Size: 13
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Architecture: all
Source: php-defaults (75)
Version: 2:7.4+75
```

Fonte: Os autores (2021)

Após a instalação, é possível verificar o funcionamento acessando o endereço *IP* do servidor em um navegador *web*. Após acessar `http://ip_do_servidor` foi exibida a imagem abaixo.

Figura 8 — Apache HTTTD



Fonte: Os autores (2021)

A instalação do sistema operacional e pacotes nesse formato, adicionam diversos outros pacotes e dependências desnecessárias, que aumentam a exigência de capacidade computacional dos servidores.

4.1.3 Integração entre o Apache2 e o PHP

Para realizar a integração entre o *servidor Apache2* e *PHP* é necessário habilitar o módulo "*php*" no *apache*. Além disso, nesse momento deve ser realizada a cópia dos arquivos que contém o código fonte da aplicação para o diretório `/var/www/html/`, raiz do *apache*. A cópia dos arquivos pode ser feita por *SSH*, compartilhamento ou *GitHub* (tipo recomendado para uso pela fácil sincronização para uma equipe de desenvolvimento). A figura 9 demonstra de forma resumida a habilitação do módulo *PHP* no *Apache* e os arquivos no diretório raiz do site.

Figura 9 — Integração Apache e PHP

```

ubuntu@bare-metal:~$ sudo a2enmod php7.4
Considering dependency mpm_prefork for php7.4:
Considering conflict mpm_event for mpm_prefork:
Considering conflict mpm_worker for mpm_prefork:
Module mpm_prefork already enabled
Considering conflict php5 for php7.4:
Enabling module php7.4.
To activate the new configuration, you need to run:
    systemctl restart apache2
ubuntu@bare-metal:~$ systemctl restart apache2
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
Authentication is required to restart 'apache2.service'.
Authenticating as: Ubuntu (ubuntu)
Password:
==== AUTHENTICATION COMPLETE ====
ubuntu@bare-metal:~$ ls -l /var/www/html
total 36
-rw-r--r-- 1 root root 777 Jun 6 16:16 cadastrar.php
-rw-r--r-- 1 root root 2563 Jun 6 16:16 cadastro.php
-rw-r--r-- 1 root root 203 Jun 6 16:16 conexao.php
drwxr-xr-x 2 root root 4096 Jun 6 16:16 css
-rw-r--r-- 1 root root 2432 Jun 6 16:16 index.php
-rw-r--r-- 1 root root 685 Jun 6 16:16 login.php
-rw-r--r-- 1 root root 80 Jun 6 16:16 logout.php
-rw-r--r-- 1 root root 143 Jun 6 16:16 painel.php
-rw-r--r-- 1 root root 90 Jun 6 16:16 verifica_login.php

```

Fonte: Os autores (2021)

Feito isso, é possível visualizar a aplicação ao acessar o endereço `http://ip_do_servidor` conforme a figura 10, exposta abaixo.

Figura 10 — Aplicação PHP



Fonte: Os autores (2021)

4.1.4 Configuração do banco de dados MySQL

Com o pacote instalado anteriormente, algumas configurações adicionais são necessárias para tornar o servidor de banco de dados acessível pela aplicação. Dentre elas, a configuração de autenticação do usuário, popular o banco de dados e permitir o acesso remoto.

O primeiro passo, atribui as permissões de acesso ao usuário *root* a partir de qualquer *host*. A segunda parte realiza a criação da base de dados, tabela e adição de um usuário *admin*, utilizado pela aplicação *PHP*, através da importação e execução de um *script "dump.sql"*. O terceiro bloco altera o endereço de escuta do banco de dados para que seja possível receber solicitações de acesso a partir de qualquer *host*. A figura 11 exibe as principais configurações realizadas.

Figura 11 — Configuração do MySQL

```
ubuntu@bare-metal:~$ mysql -u root -p
Enter password:
mysql> CREATE USER 'root'@'%' IDENTIFIED BY 'root';
ERROR 1396 (HY000): Operation CREATE USER failed for 'root'@'%'
mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION;
Query OK, 0 rows affected (0.00 sec)

ubuntu@bare-metal:~$ cat dump.sql
CREATE DATABASE login;
USE login;

CREATE TABLE `login`.`usuario` (
  `usuario_id` INT NOT NULL AUTO_INCREMENT,
  `usuario` VARCHAR(200) NOT NULL,
  `senha` VARCHAR(32) NOT NULL,
  `nome` VARCHAR(100) NOT NULL,
  `data_cadastro` DATETIME NOT NULL,
  PRIMARY KEY (`usuario_id`));

INSERT INTO `usuario` (`usuario_id`,`usuario`,`senha`,`nome`,`data_cadastro`) VALUES (1,'admin','21232f297a57
a5a743894a0e4a801fc3', 'Admin', '2019-01-11 19:42:12');
ubuntu@bare-metal:~$ mysql -u root -p < ./dump.sql
Enter password:
ubuntu@bare-metal:~$ sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
[mysqld]
bind-address            = 0.0.0.0
ubuntu@bare-metal:~$ sudo systemctl restart mysql
ubuntu@bare-metal:~$ ss -ntap | grep 3306
LISTEN 0      70          127.0.0.1:3306      0.0.0.0:*
LISTEN 0      151         0.0.0.0:3306       0.0.0.0:*
```

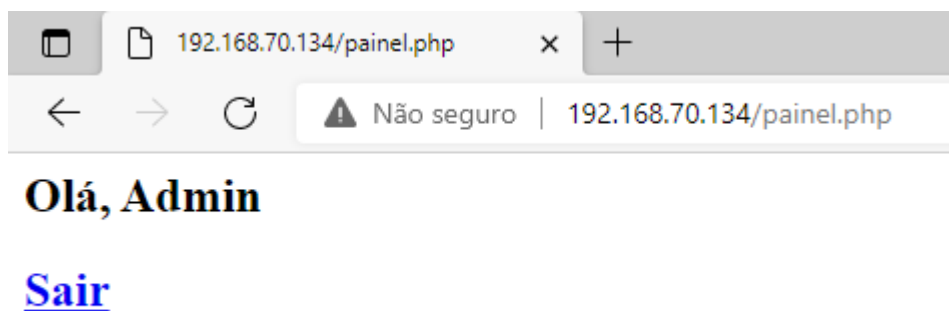
Fonte: Os autores (2021)

4.1.5 Integração entre PHP e MySQL

O último passo para um sistema totalmente funcional é permitir que código *PHP* consiga realizar operações no banco de dados *MySQL*. A conexão com banco de dados integração do *PHP* é realizada no código *PHP* pelo parâmetro "*mysqli_connect*". Para a execução, é necessária a instalação do pacote "*php-mysql*" e habilitação do módulo "*mysqli*" no *PHP* com o comando "*phpenmod mysqli*".

Somente após todos os processos de instalações, configurações e integrações é possível verificar o funcionamento total da aplicação. O servidor *web* é acessado com, exibe a página de *login*, o usuário insere os dados de usuário e senha e é realizada a consulta no banco de dados para validar as credenciais do usuário.

Figura 12 — Aplicação funcional



Fonte: Os autores (2021)

4.2 AMBIENTE COM DOCKER

A principal proposta objetivada neste trabalho, é fornecer uma solução para o provisionamento e gerenciamento de infraestrutura e aplicações totalmente automatizado e profundamente integrado ao ciclo de vida geral dos aplicativos. Com o desenvolvimento remoto (*home-office*), é importante que os desenvolvedores possam construir, testar e implantar repetidamente versões mais recentes de seus aplicativos, independentemente de onde estejam sendo executados. Que as equipes de TI possam atualizar as versões do SO e do *software* sem complicações quando os *pachtes* de segurança e correções de *bugs* são lançados.

No ambiente deste subcapítulo será mostrado como utilizar a ferramenta *Docker* para resolver os problemas enfrentados com soluções *bare-metal* para que a equipe de *DevOps* continue realizando seu trabalho de forma prática mesmo não estando presencialmente no seu local de trabalho com a máquina da empresa (quando for o caso).

4.2.1 Instalação do Docker

O pacote de instalação do *Docker* disponível no repositório oficial do *Ubuntu* pode não ser a versão mais recente. Para garantir que se obtenha a versão mais recente, deverá ser instalado o *Docker* do repositório oficial. Antes de instalar o *Docker Engine* pela primeira vez em uma nova máquina, é necessário configurar o repositório do *Docker*. Para fazer isso, é necessário adicionar uma nova fonte de pacote, a chave GPG do *Docker* garantirá que os downloads sejam válidos, e então será prosseguida a instalação do pacote. Com o repositório configurado, é possível instalar e atualizar o *Docker* a partir do repositório. Após a instalação completa, utilizando o comando abaixo, poderá ser averiguado que o *Docker* está ativo na máquina:

Figura 13 — Instalação do Docker

```
ubuntu@docker-host:~$ sudo apt-get update
ubuntu@docker-host:~$ sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-release
ubuntu@docker-host:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /usr/share/keyrings/docker-archive-keyring.gpg
ubuntu@docker-host:~$ echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d
/docker.list > /dev/null
ubuntu@docker-host:~$ sudo apt-get update
ubuntu@docker-host:~$ sudo apt-get install docker-ce docker-ce-cli containerd.io
ubuntu@docker-host:~$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2021-06-07 17:12:27 UTC; 6min ago
```

Fonte: Os autores (2021)

4.2.2 Criação da imagem Docker personalizada

Uma vez que o *Docker* foi instalado e está pronto para uso, é necessário a migração da aplicação ou conjunto de aplicações para o ambiente *Docker*. Esse processo é realizado através da criação de uma imagem *Docker* que replica os recursos utilizados no ambiente local descrito no primeiro cenário. Essa imagem pode ser construída a partir de modelos pré-definidos ou personalizada de acordo com a necessidade da aplicação. Conforme mencionado no capítulo 3, para construir o aplicativo, é necessário usar um *Dockerfile*. "Um *Dockerfile* é simplesmente um *script* de instruções baseado em texto que é usado para criar uma imagem de contêiner", (Docker). Nesta seção, será abordada a customização de duas imagens necessárias para esta aplicação, uma para o banco de dados *MySQL*, e outra para o servidor *Apache* com *PHP*.

Neste processo, foi encontrado o principal desafio, que é a migração das aplicações para uma imagem *Docker* adequada às necessidades, evitando a utilização de ferramentas comuns e desnecessárias que acompanham uma aplicação executada em um sistema operacional de *host*. Por isso, optou-se por criar imagens personalizadas, enxutas, somente com os recursos e dependências necessárias para a aplicação. Imagens enxutas são mais rápidas de puxar pela rede e mais rápido para carregar na memória ao iniciar contêineres ou serviços.

Os *Dockerfiles* completos das aplicações podem não serão fornecidos diretamente neste material devido aos seus tamanhos, porém na demonstração desse bloco serão demonstradas versões resumidas (sem os comandos de instalação de dependências) dos arquivos.

Figura 14 — MySQL Dockerfile (adaptado)

```

1 # cria uma camada a partir da imagem Docker debian:buster-slim.
2 FROM debian:buster-slim
3
4 # Cria um usuario e um grupo de sistema chamados "mysql", atribuindo o usuário ao grupo.
5 RUN groupadd -r mysql && useradd -r -g mysql mysql
6
7 # Determina a versão MySQL utilizada na imagem.
8 ENV MYSQL_MAJOR 8.0
9 ENV MYSQL_VERSION 8.0.25-1debian10
10
11 # Adiciona o repositório MySQL à source list do APT para instalação.
12 RUN echo 'deb http://repo.mysql.com/apt/debian/ buster mysql-8.0' > /etc/apt/sources.list.d/mysql.list
13
14 O comandos e parâmetros abaixo eliminam pré-configurações desnecessárias, assim como bases de dados
15 que acompanham a aplicação
16 # Altera as permissões do /var/run/mysqld (usado para arquivos de socket e lock) para que seja
17 gravável independentemente do UID em tempo de execução.
18 RUN
19     && apt-get update \
20     && apt-get install -y \
21         mysql-community-client="${MYSQL_VERSION}" \
22         mysql-community-server-core="${MYSQL_VERSION}" \
23     && rm -rf /var/lib/apt/lists/* \
24     && rm -rf /var/lib/mysql && mkdir -p /var/lib/mysql /var/run/mysqld \
25     && chown -R mysql:mysql /var/lib/mysql /var/run/mysqld \
26     && chmod 1777 /var/run/mysqld /var/lib/mysql
27
28 # A instrução VOLUME expõe uma área transitória utilizada pelo contêiner. É usada para partes
29 mutáveis possam ser reparadas pelo usuário da imagem.
30 VOLUME /var/lib/mysql
31
32 # A instrução informa ao Docker que o contêiner escutará na porta de rede 3306 no tempo de execução.
33 EXPOSE 3306 3306
34
35 # Execuda o daemon "mysqld" (Servidor MySQL)
36 CMD ["mysqld"]

```

Fonte: Os autores (2021)

Figura 15 — Apache + PHP Dockerfile (adaptado)

```

1 # cria uma camada a partir da imagem Docker debian:buster-slim.
2 FROM debian:buster-slim
3
4 # Especifica e cria o diretório do arquivo de configuração "php.ini"
5 # Cria e atribui permissão ao diretório raiz do site
6 ENV PHP_INI_DIR /usr/local/etc/php
7 RUN set -eux; \
8     mkdir -p "$PHP_INI_DIR/conf.d"; \
9     [ ! -d /var/www/html ]; \
10    mkdir -p /var/www/html; \
11    chown www-data:www-data /var/www/html; \
12    chmod 777 /var/www/html
13
14 # Especifica o diretório dos arquivos de configuração e o arquivo de variáveis do
15 ambiente do Apache
16 # instala o Apache2
17 ENV APACHE_CONFDIR /etc/apache2
18 ENV APACHE_ENVVARS $APACHE_CONFDIR/envvars
19 RUN apt-get update; \
20     apt-get install -y --no-install-recommends apache2; \
21
22 # Copia o docker-php-source (provê diversas extensões utilizadas pelo PHP) para a pasta /
23 usr/local/bin/
24 COPY docker-php-source /usr/local/bin/
25
26 # Copia os scripts de extensões docker-php-ext-configure, docker-php-ext-install,
27 docker-php-ext-enable e docker-php-entrypoint para a imagem.
28 COPY docker-php-ext-* docker-php-entrypoint /usr/local/bin/
29
30 # Habilita o módulo sodium e mysqli (módulo compartilhado)
31 RUN docker-php-ext-enable sodium \
32     && docker-php-ext-install mysqli
33
34 # Define o diretório de trabalho para quaisquer instruções do Dockerfile
35 WORKDIR /var/www/html
36
37 # A instrução informa ao Docker que o contêiner escutará na porta de rede 80 no tempo de
38 execução.
39 EXPOSE 80

```

Fonte: Os autores (2021)

Após a criação dos *Dockerfiles*, o comando "*docker build*" poderá ser executado para iniciar o processo de construção da imagem *Docker*. Por padrão, ele procurará um *Dockerfile* na raiz do diretório de construção.

O comando foi executado com o parâmetro "-t", que atribui uma *tag* à imagem. Além disso, foi definido o parâmetro "-q" (*quiet*), que suprime a saída da compilação e imprime o *ID* da imagem em caso de sucesso. A figura 16 demonstra a execução dos comandos para criação das imagens com os nomes "*mysql-lab*" e "*php-apache-lab*", e a listagem das imagens contidas do dispositivo. É importante notar que além das imagens criadas, há uma imagem "*debian buster-slim*", utilizada como fonte de criação das demais.

Figura 16 — Criação das imagens Docker

```
ubuntu@docker-host:~/mysql$ docker build -q -t mysql-lab ./
sha256:a9e97b84cb67d5410a4234d8c49561916101032f4d0bf3d10b3023de50f2522b
ubuntu@docker-host:~/php-apache$ sudo docker build -q -t php-apache-lab ./
sha256:697c014b6209828c52f5801a2ae446d62dac12d5aa44be84a226502c1341efec
ubuntu@docker-host:~/php-apache$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
php-apache-lab	latest	697c014b6209	About a minute ago	435MB
mysql-lab	latest	a9e97b84cb67	20 minutes ago	556MB
debian	buster-slim	80b9e7aadac5	3 weeks ago	69.3MB

Fonte: Os autores (2021)

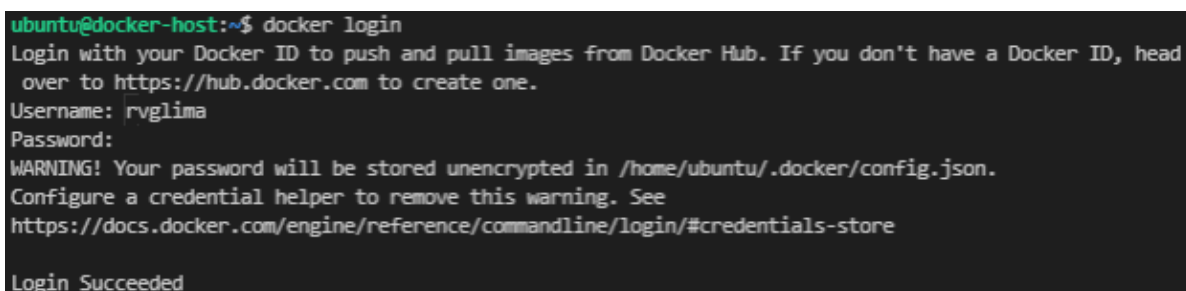
4.2.3 Compartilhamento da imagem criada no Docker Hub

Com a migração das aplicações para uma imagem *Docker*, os próximos passos demonstram o processo de disponibilizar a imagem criada na nuvem do *Docker Hub*. Primeiramente, é necessário ter uma conta criada no *Docker Hub*, o que pode ser realizado de forma trivial ao acessar o site do hub.docker.com (link válido em junho/2021). Ao criar a conta, ela será vinculada em um *Docker ID*, o que mais a frente será requisitada para a conclusão deste subcapítulo.

Com a conta no *Hub* devidamente criada, o próximo passo será criar um repositório para poder fazer o *upload* da imagem, o que também pode ser feito de forma intuitiva ao obter sucesso após logar-se no *Hub*.

Uma vez criado o repositório que irá receber as imagens, faz-se necessário realizar o login no *Hub* pelo terminal. Para isso, deve ser executado o comando "*docker login*". A figura 17 exibe processo de *login*.

Figura 17 — Docker Hub login

A terminal window with a dark background and light green text. The prompt is 'ubuntu@docker-host:~\$'. The command 'docker login' is entered. The output shows instructions to login with a Docker ID, followed by the username 'rvglima' and a password prompt. A warning message is displayed, and the login is successful.

```
ubuntu@docker-host:~$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head
over to https://hub.docker.com to create one.
Username: rvglima
Password:
WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Fonte: Os autores (2021)

O *upload* de uma imagem local para a nuvem *Docker Hub* é executado através do comando "*docker push*", mas antes é necessário atribuir uma *tag* remota à imagem. A figura abaixo demonstra o processo atribuição da *tag* através do comando "*docker tag <existing-image> <hub-user>/<repo-name>[:<tag>]*" e *upload* das imagens.

Figura 18 — Docker push

```

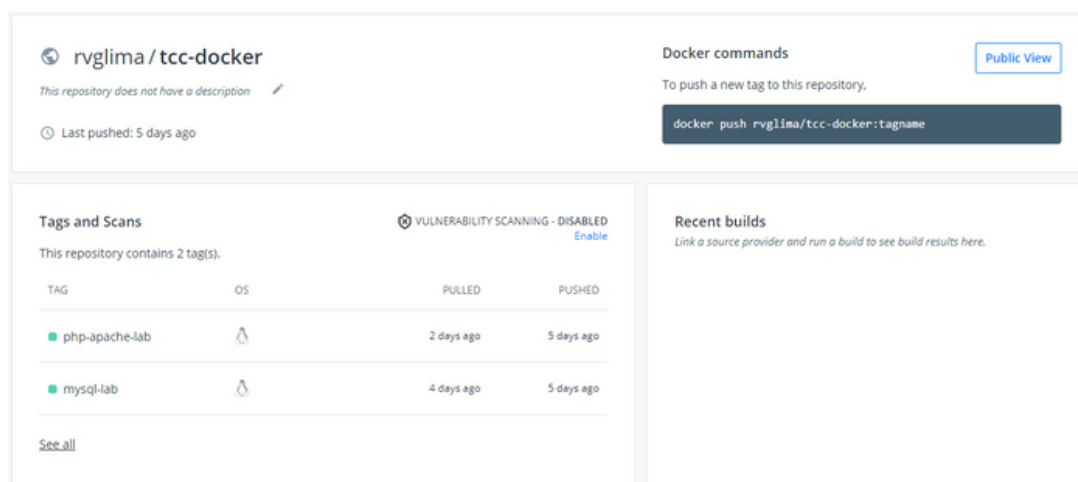
ubuntu@docker-host:~$ docker tag <hub-user>/<repo-name>[:<tag>]
debian:buster-slim mysql-lab:latest php:apache php-apache-lab:latest
ubuntu@docker-host:~$ docker tag mysql-lab:latest rvglima/tcc-docker:mysql-lab
ubuntu@docker-host:~$ docker tag php-apache-lab:latest rvglima/tcc-docker:php-apache-lab
ubuntu@docker-host:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
php-apache-lab      latest             697c014b6209       About an hour ago  435MB
rvglima/tcc-docker  php-apache-lab    697c014b6209       About an hour ago  435MB
rvglima/tcc-docker  mysql-lab         a9e97b84cb67       About an hour ago  556MB
mysql-lab           latest            a9e97b84cb67       About an hour ago  556MB
debian              buster-slim       80b9e7aadac5       3 weeks ago        69.3MB
ubuntu@docker-host:~$ docker push rvglima/tcc-docker:mysql-lab
The push refers to repository [docker.io/rvglima/tcc-docker]
1fd6f1fcf995: Pushing 1.536kB
a05ddf022465: Pushing 17.92kB
a24f6900a54a: Pushing 5.632kB
80be15d2ada6: Pushing 7.782MB/420.4MB
730aed4a4701: Pushing 3.584kB
f50ed34e94d7: Waiting
e8385f1d3de2: Waiting
ubuntu@docker-host:~$ docker push rvglima/tcc-docker:php-apache-lab
The push refers to repository [docker.io/rvglima/tcc-docker]
04b2807a31f7: Pushing 185.3kB/17.8MB
5334826963fb: Pushing 185.3kB
8d85ccbc089b: Preparing
66d41002e650: Preparing
c7631fad7e28: Preparing
2e9d04783e40: Waiting
d90aa8ab19ed: Waiting

```

Fonte: Os autores (2021)

Através do site do *Docker Hub* é possível visualizar as imagens no repositório. Com as imagens disponibilizadas, o usuário das aplicações tem seu trabalho facilitado, sendo necessário apenas o *download (pull)* da imagem e a execução.

Figura 19 — Imagens no Docker Hub



Fonte: Os autores (2021)

4.2.4 Execução do contêiner

Finalmente após todo processo de criação dos *DockerFiles* e com o *upload* das imagens descritas no subtítulo, as aplicações estão prontas para serem executadas as ferramentas da imagem já criada e personalizadas. Neste subtítulo será abordada a execução dos contêineres das aplicações *PHP/MySQL*.

Inicialmente, será demonstrado o *download* das imagens em um sistema apenas com o *Docker* instalado. É importante frisar que para realizar o *download* da imagem em um repositório privado, é necessário realizar o *login* no *Docker Hub* conforme mostrado anteriormente. O *download* das imagens é realizado com o comando "*docker pull*"

Figura 20 — Docker pull

```
ubuntu@docker-host:~$ docker pull rvglima/tcc-docker:mysql-lab
mysql-lab: Pulling from rvglima/tcc-docker
69692152171a: Already exists
ef156a82cf48: Pulling fs layer
9345714aa155: Downloading 42.89kB/4.179MB
d374df720031: Pulling fs layer
2f78b89ffa87: Waiting
a8a2782af2e9: Waiting
4f1e8171ad56: Waiting
Digest: sha256:997ef422809b66ab5ed27a8a728be3581ae8c326914fb9758367a9160effd65f
Status: Downloaded newer image for rvglima/tcc-docker:mysql-lab
docker.io/rvglima/tcc-docker:mysql-lab
ubuntu@docker-host:~$ docker pull rvglima/tcc-docker:php-apache-lab
php-apache-lab: Pulling from rvglima/tcc-docker
69692152171a: Already exists
2040822db325: Pull complete
9b4ca5ae9dfa: Downloading 43.08MB/76.68MB
ac1fe7c6d966: Download complete
5b26fc9ce030: Download complete
3492f4769444: Download complete
1dec05775a74: Download complete
519598d7b825: Download complete
Digest: sha256:377b5051185ef43570f86fe6c4c4417c260f2bb303fa3e85d903eaaf02448fa3
Status: Downloaded newer image for rvglima/tcc-docker:php-apache-lab
docker.io/rvglima/tcc-docker:php-apache-lab
```

Fonte: Os autores (2021)

A execução dos contêineres *MySQL* e *Apache-PHP* são executadas através do comando "*docker run*". O comando "*docker run*" executa processos em contêineres isolados. Um contêiner é um processo executado em um *host*. O *host* pode ser local ou remoto. Ao executar o comando, o processo do contêiner

executado é isolado para ter seu próprio sistema de arquivos, sua própria rede e sua própria árvore de processo isolada separada do *host*. A figura abaixo demonstra a execução dos contêineres.

Figura 21 — Docker run

```
ubuntu@docker-host:~$ docker run --name mysql-app -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 -d rvglima/tcc-docker:mysql-lab
27e510d3ec6b234d3dccc485f0f1a6d3ca95d59190f53d2793c64cc20278c73c5
ubuntu@docker-host:~$ docker run -d -p 80:80 --name php-apache-app -v /home/ubuntu/login-php:/var/www/html rvglima/tcc-docker:php-apache-lab
29c963ffaccfa6ea0991712fbb74ca1797705e65d6a6053a0b7c794d7b3f6831
```

Fonte: Os autores (2021)

Os parâmetros utilizados possuem as seguintes funções:

- **--name** - Atribui um nome ao contêiner. Nesse caso, foram utilizados "*mysql-app*" e "*php-apache-app*".
- **-e** - O parâmetro define variáveis de ambiente no contêiner. Para o contêiner *MySQL* foi passada a variável *MYSQL_ROOT_PASSWORD* com o valor *root*, configurando a senha do usuário administrador do banco de dados.
- **-v** - Atribui um volume ao contêiner. O parâmetro foi especificado no contêiner *Apache PHP* para criar uma ligação da pasta local que contém o site *PHP* com o diretório raiz do apache */var/www/html* configurado no contêiner.
- **-d** - Diz ao contêiner para ser executado em segundo plano.
- **-p** - A opção publica todas as portas para as interfaces do *host*, permitindo que a conexão externa com o *host* seja direcionada ao contêiner. Foram publicadas as portas 3360 para o *MySQL* e 80 para o servidor *web Apache*.
- **image[:tag]** - Parâmetro obrigatório, que indica uma imagem da qual derivar o contêiner.

Com os contêineres em execução, as aplicações já estarão funcionando e prontas para uso pelo desenvolvedor.

4.3 AMBIENTE COM DOCKER SWARM

Esta seção demonstra o desenvolvimento simulado do cenário anterior, baseado em *Docker* contêiner, porém com a construção de um *cluster* de alta

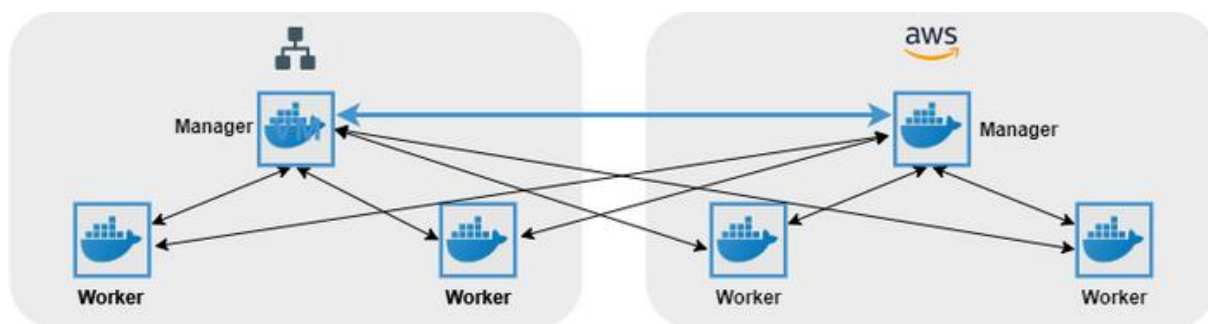
disponibilidade com *Docker Swarm* em ambiente misto (local e nuvem) para a aplicação web.

A demonstração desse cenário tem como propósito destacar os benefícios da utilização de ferramentas de orquestração de contêineres, o que permite que a equipe de infraestrutura gerencie vários contêineres implantados em várias máquinas hospedeiras, tornando a aplicação com alto nível de disponibilidade.

Além disso, a utilização de um ambiente misto, composto pela infraestrutura local da empresa e o uso da infraestrutura de computação em nuvem da *Amazon*, a *AWS* (*Amazon Web Services*), fornece uma infraestrutura de *backup* em caso de falha do ambiente local.

Nesta simulação experimental, o contêiner com o servidor web *Apache2* e a aplicação web *PHP* é disponibilizado em uma estrutura de cluster com seis nós *Swarm* (2 *Managers* e 4 *Workers*) criando seis máquinas virtuais. A divisão das máquinas virtuais é demonstrada na figura 22. No entanto, todas essas VM e, posteriormente, os nós do *Docker Swarm* podem ser criados em diferentes ambientes de computação em nuvem.

Figura 22 — Cenário Docker Swarm



Fonte: Os autores (2021)

Após a criação das máquinas virtuais, o *Docker* foi instalado conforme os passos descritos na seção anterior. Os dois nós *managers* do Docker Swarm são chamados "*manager-local*" e "*manager-aws*" e os quatro nós *workers* são chamados "*worker1-local*", "*worker2-local*", "*worker1-aws*" e "*worker2-aws*".

4.3.1 Construção do cluster

Ao instalar o *Docker*, o modo swarm é desabilitado por padrão. Ao executar o comando "*docker swarm init*", o *Docker Engine* começa a ser executado no modo swarm. O "*docker swarm init*" gera um token aleatório para adicionar um nó *worker* ao *swarm*. Para obter o *token* de adição do nó *manager* é necessário executar o comando "*docker swarm join-token manager*". A figura 23 demonstra a saída dos comandos.

Quando um novo nó é adicionado ao *swarm*, o nó se associa como um *worker* ou *manager* com base no token utilizado com o comando "*docker swarm join*". Mesmo após configurado é possível visualizar os *join-tokens* para *manager* ou *worker* com o comando "*docker swarm join-token (worker|manager)*" conforme a figura abaixo.

Figura 23 — Swarm join-token

```
ubuntu@manager-local:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-55zd1zkmgswm9uxd36x94vvvs2j22ulbsnsbzmz3li6hriwg03-ddm5m1z2tb3n5glngxg6je72qm 192.168.70.51:2377

ubuntu@manager-local:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-55zd1zkmgswm9uxd36x94vvvs2j22ulbsnsbzmz3li6hriwg03-9broqcgankduo6brr74u7rrpt 192.168.70.51:2377
```

Fonte: Os autores (2021)

Após a adição dos nós é possível visualizá-los com o comando "*docker node ls*" (DOCKER).

Figura 24 — Lista dos nós e funções Docker Swarm

```
ubuntu@manager-local:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
h78yiv6i5y65o6hboeiee8pzm	manager-aws	Ready	Active	Reachable	20.10.6
x9xst0fvavjx4omlfk89oi5pp *	manager-local	Ready	Active	Leader	20.10.6
xqqmlt6unqwb4xfo9o4v4wmp	worker1-aws	Ready	Active		20.10.6
swgc642so1rj3ziv3sy6qkndh	worker1-local	Ready	Active		20.10.6
vvdxv70gptsb090nn8ix4zzrh	worker2-aws	Ready	Active		20.10.7
ivil7uz1i3sxtxv79i7t6q1bq	worker2-local	Ready	Active		20.10.6

Fonte: Os autores (2021)

O *Swarm* nunca cria contêineres individuais como feito no cenário anterior. Em vez disso, todas as cargas de trabalho do *Swarm* são agendadas como serviços,

que são grupos escalonáveis de contêineres com recursos de rede adicionais mantidos automaticamente pelo *Swarm*. Além disso, todos os objetos *Swarm* podem e devem ser descritos em manifestos chamados *stack files* (arquivos de pilha). Esses arquivos *YAML* descrevem todos os componentes e configurações do *Swarm app* e podem ser usados para criar e destruir facilmente o aplicativo em qualquer ambiente *Swarm*.

Neste cenário, a aplicação *web* com *Apache2* e *PHP* executada como um serviço *Swarm*. A aplicação foi escrita em um *stack file services.yml*:

Figura 25 — Arquivo compose da aplicação

```
ubuntu@manager-local:~$ cat services.yml
version: '3.3'

services:
  php-apache:
    image: rvglima/tcc-docker:php-apache-lab
    deploy:
      replicas: 4
      placement:
        constraints: [node.role == worker]
    ports:
      - "80:80"
```

Fonte: Os autores (2021)

Neste arquivo *YAML Swarm*, temos apenas um objeto: *service*, descrevendo um grupo escalável de contêineres idênticos. Nesse caso, será utilizado apenas um contêiner e esse contêiner será baseado na imagem *php-apache-lab* gerada no cenário anterior. Além disso, o *Swarm* é configurado para encaminhar todo o tráfego que chega na porta 80 da máquina hospedeira para a porta 80 do contêiner com a aplicação.

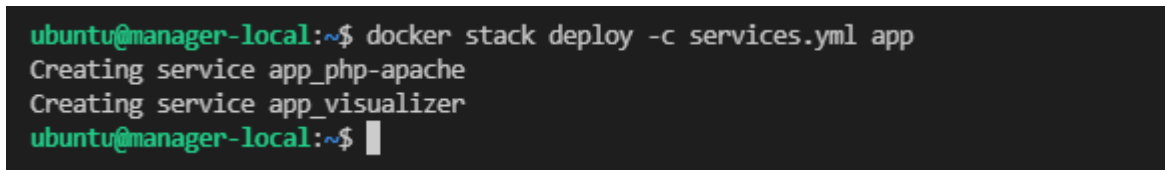
No *deploy* são descritos respectivamente as réplicas e a colocação deste objeto. O sinalizador *--replicas* serve para definir o número de tarefas de réplica para o serviço replicado. No comando descrito na figura 25 a tarefa é replicada 4 vezes. O parâmetro *constraints* restringe os nós nos quais o serviço pode ser atribuído, neste caso aos nós *workers*.

4.3.2 Implantação do serviço

A implantação da pilha de aplicativos pode ser realizada com o comando "*docker stack deploy*", que será responsável por propagar o serviço nos nós *Swarm* especificados. O comando aceita uma descrição de pilha na forma de um arquivo *compose*, criado na etapa anterior. Este comando deverá ser executado em um dos nós *managers*.

É importante salientar que se a imagem ainda não estiver disponível para cada nó, ela será automaticamente extraída do *Docker Hub*. Portanto, para que este exemplo funcione, todos os nós devem ter acesso direto à Internet ou devem ser configurados para usar um *proxy*. Em um ambiente de produção, as imagens normalmente seriam hospedadas em um registro local e cada nó poderia extrair a imagem do registro local.

Figura 26 — Implantação do serviço com stack deploy

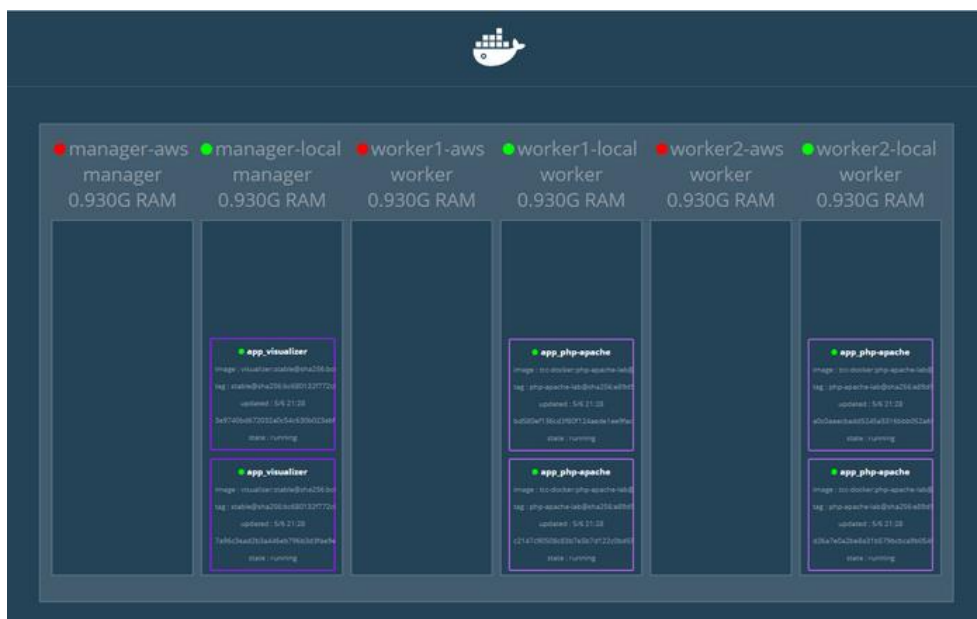
A terminal window with a dark background and light green text. The prompt is 'ubuntu@manager-local:~\$'. The command entered is 'docker stack deploy -c services.yml app'. The output shows 'Creating service app_php-apache' and 'Creating service app_visualizer'. The prompt returns to 'ubuntu@manager-local:~\$' with a cursor.

```
ubuntu@manager-local:~$ docker stack deploy -c services.yml app
Creating service app_php-apache
Creating service app_visualizer
ubuntu@manager-local:~$
```

Fonte: Os autores (2021)

Após a implantação do serviço é possível identificar que os contêineres, em forma de serviço, foram replicados em todos os nós *workers*. Para auxiliar a visualização da distribuição dos contêineres também foi implantado um outro serviço chamado *Visualizer*. O *Visualizer* é um contêiner de demonstração que exibe os serviços *Docker* em execução em um *Docker Swarm* na forma de diagrama. Na figura 27, é possível ver a distribuição dos contêineres entre os nós. A aplicação com *PHP* e *Apache2* foi implantada em quatro réplicas nos quatro nós *workers*, por sua vez, o *visualizer*, foi implantado nos dois nós *managers*.

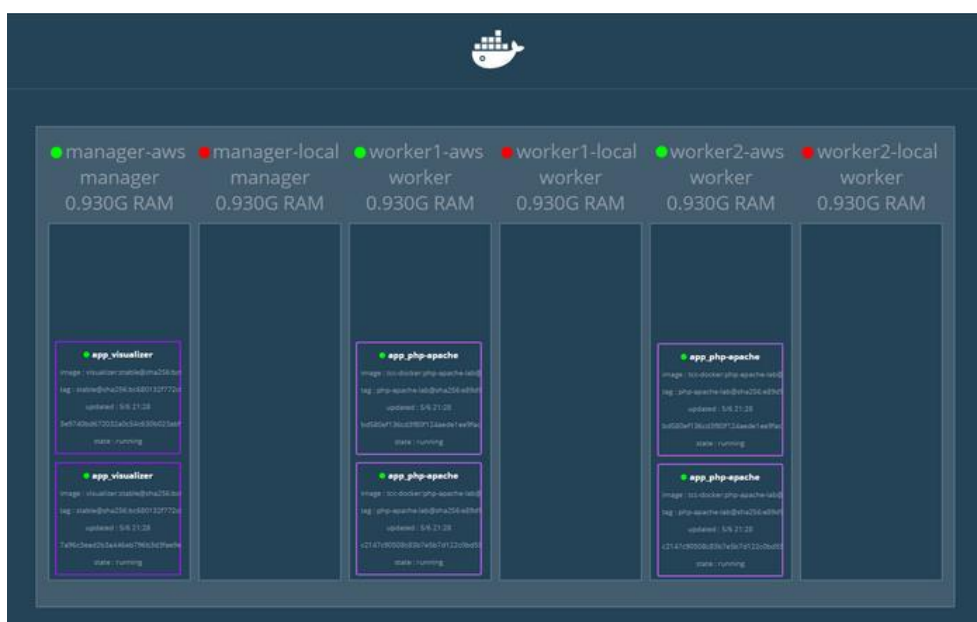
Figura 28 — Docker Swarm apenas local



Fonte: Os autores (2021)

Em caso do comprometimento da rede local, é possível perceber na figura 29, que os nós posicionados na nuvem AWS assumem a execução dos contêineres, fornecendo alta disponibilidade ao cenário. Enquanto o problema não for resolvido a infraestrutura de nuvem alugada manterá o sistema sempre em funcionamento.

Figura 29 — Docker Swarm apenas AWS



Fonte: Os autores (2021)

A economia de tempo com a configuração e operação de um *cluster* no modo *Swarm*, combinada com sua experiência do usuário e interface pontuais, define o *Docker* como a principal ferramenta no segmento e realmente permite o gerenciamento de contêiner em grande escala.

4.4 ANÁLISE DOS CENÁRIOS

Após a demonstração prática de cada cenário, nada se faz tão necessário quanto a análise comparativa das vantagens e desvantagens em cada.

4.4.1 AMBIENTE EM SERVIDOR BARE-METAL

No primeiro cenário, foi mostrada a parte prática de criação de uma infraestrutura local de desenvolvimento e disponibilização de recursos em um servidor *bare-metal*. O objetivo do cenário foi a construção da infraestrutura da aplicação *Apache PHP/MySQL* em um novo servidor com todos os *softwares* (sistema operacional, aplicativos e dados) reconstruído do zero. Essa simulação visa avaliar o tempo despendido para uma atualização de *hardware* ou uma recuperação de falhas.

Como resultado, constatou-se que todo o processo de provisionamento do servidor demoraria cerca de um dia de trabalho para um profissional com conhecimento das ferramentas utilizadas. Isso se deve ao fato dos seguintes processos:

- Instalação do sistema operacional
- Configurações de acesso à rede
- Instalação de aplicativos e dependências
- Atualizações de *softwares*
- *Hardening*
- Modificações dos processos de configuração (versionamento)
- Testes de funcionamento

Portanto, há um grande problema com o gerenciamento de um servidor *bare-metal*. Não há uma solução completa que forneça um modelo operacional

gerenciado contínuo que automatize o provisionamento, a implantação e o gerenciamento de servidores *bare-metal*. Seja em qualquer ambiente, as empresas estão procurando resolver o problema de automatizar e orquestrar o ciclo de vida de servidores *bare-metal* em uma maneira que seja tão simples quanto operar a nuvem. As mudanças nas necessidades de negócios também exigem agilidade. O *bare-metal* deve acelerar o negócio, e não o retardar como acontece hoje.

Podem ser destacadas como vantagens do primeiro cenário o maior desempenho computacional, quando comparado ao uso de máquinas virtuais ou contêineres, que lidam com camadas adicionais de processamento através do *hypervisor* ou contêiner *engine*. Além do desempenho, medidas de segurança e controle total do *hardware* e SO são mais fáceis de gerenciar.

4.4.2 AMBIENTE COM DOCKER

No segundo cenário, que se tem como objetivo explicitar o uso prático de contêineres *Docker*. Neste cenário é observado uma imensa vantagem comparativa em relação ao primeiro cenário, pois com *Docker*, além do desenvolvimento remoto poder ser livre de erros de compatibilidade (caso a equipe conte com funcionários que utilizem *Windows* e funcionário que utilizem *Linux*), também se tem uma praticidade ímpar, por simplesmente realizar *download* de uma imagem já pronta sem necessitar de dependências externas para o uso tanto do *PHP*, quanto do *MySQL*.

Já no terceiro cenário, o emprego do *Swarm* leva a aplicação e o desenvolvimento a um nível acima, neste cenário, além da utilização do *Docker*, o que por si só já facilita muito o trabalho do desenvolvedor, soma-se também o uso de *Clusters* de alta disponibilidade, com a inserção do *AWS*, o que na prática, caso o servidor de uma máquina venha a cair a *AWS* mantém o mesmo ativo, sem interrupção de serviço para o cliente.

O segundo cenário, tem como objetivo explicitar a migração de um conjunto de aplicações para um ambiente com contêineres *Docker* de forma prática, personalizada e de rápido provisionamento. Neste cenário é observado uma imensa vantagem comparativa em relação ao primeiro cenário, pois com *Docker*, além do

desenvolvimento remoto ser livre de erros de compatibilidade (caso a equipe conte com funcionários que utilizem sistemas operacionais *Windows* e funcionário que utilizem *Linux* ou até mesmo *Mac OS*), também é obtido uma praticidade ímpar, por simplesmente realizar o *download* das imagens já criadas, disponíveis na nuvem *Docker Hub* para atender as necessidades da aplicação, sem demandar de dependências externas para o uso tanto do *PHP*, quanto do *MySQL* presentes neste trabalho. Pode-se destacar também como ponto positivo para este cenário a considerável economia de *hardware* quando comparado ao primeiro, pois, as imagens do *Apache PHP* e do banco de dados *MySQL* totalizam aproximadamente 500 MB cada. No primeiro cenário, apenas o sistema operacional *Ubuntu* (mesmo sistema utilizado para executar as aplicações dos cenários descritos) com as aplicações e dados totalizam cerca de 7 GB de armazenamento, sem as atualizações do sistema operacional, aplicativos e dependências.

Como resultado neste cenário, constatou-se que o processo de provisionamento da aplicação demoraria cerca de 15 minutos para um profissional com conhecimento das ferramentas utilizadas. Isso se deve ao fato dos seguintes processos:

- Instalação do *Docker*
- *Pull* das imagens
- Execução do contêiner

Como desvantagens, destacam-se o maior custo computacional, já mencionado, as vulnerabilidades de segurança da aplicação, menor flexibilidade em relação ao SO (Em sistemas hospedeiros *Linux*, não é possível executar contêineres *Windows*), diferente de um sistema de virtualização.

4.4.3 AMBIENTE COM DOCKER SWARM

Já no terceiro cenário, o emprego do *Swarm* leva a aplicação e o desenvolvimento um nível acima, pois, além da utilização do *Docker*, o que por si só já facilita muito o trabalho da equipe de *DevOps*, soma-se o auto provisionamento através do uso de *clusters* de alta disponibilidade, o que na prática, caso o servidor de uma máquina venha a apresentar falha, as outras máquinas “*workers*” ou

“*managers*” mantem o sistema ativo, sem interrupção de serviço para o cliente. Outra vantagem deste cenário é a possibilidade de uso de um sistema híbrido, com a infraestrutura privada e nuvem. As equipes de TI podem provisionar instantaneamente mais servidores sob demanda quando a empresa exigisse que a infraestrutura seja dimensionada ou quando os servidores precisarem ser provisionados novamente para diferentes casos de uso/cargas de trabalho.

Como resultado deste cenário, constatou-se que o processo de provisão é iniciado automaticamente em caso de falha de um nó. Portanto, a migração do contêiner para outro nó *Swarm*, demoraria cerca de poucos segundos. É importante destacar que diversos nós em diferentes regiões podem compor *cluster*, aumentando ainda mais a disponibilidade da aplicação.

4.5 DESAFIOS E COMPLEXIDADE

Este TCC, criado por dois alunos do curso de Ciência da Computação, apresentou inúmeros desafios até a conclusão da aplicação. Nos cenários descritos neste capítulo, foram demandados horas e até mesmo dias para solucionar um único erro, por exemplo. O propósito desta seção, é descrever toda problemática enfrentada ao longo do desenvolvimento deste TCC, desde o primeiro arquivo com extensão .php, até o funcionamento da aplicação utilizando Docker *Swarm* em conexão direta com um servidor *MySQL* na nuvem *AWS*, vale ressaltar, que ambos os alunos não possuem conhecimento avançado à cerca dos temas aqui abordados. Marcos, possui conhecimento intermediário em desenvolvimento de aplicações *Web*, básico em redes e básico em banco de dados, enquanto Rafael, possui conhecimento intermediário em redes, básico em aplicações *Web* e básico em banco de dados. Vale salientar também, que ambos não possuíam nenhum conhecimento em *Docker* e *AWS*, mas sentiam-se atraídos pelo tema. Após ter em vista a ótica de habilidades dos criadores deste TCC, serão mostrados alguns dos principais desafios (foram possivelmente dezenas ou quase centenas deles, o que ficaria impossível de mostrar nesta sessão) enfrentados para a criação deste trabalho, seguindo uma ordem de plataforma, mensagem de erro/desafios encontrados, possível solução e tempo utilizado, respectivamente, para resolução de cada problema:

- *PHP*, *Warning: Undefined array key "status_cadastro"*, uso do *isset* ajudou a solucionar esse erro, foram utilizadas cerca de 3 horas até encontrar a solução.
- *PHP/MySQL*, *Warning: mysqli_connect(): (HY000/2002): Nenhuma conexão pôde ser feita porque a máquina de destino as recusou ativamente*. Verificou-se a extensão *mysqli* e a versão do *php*, uma vez que essa função não está disponível para algumas versões, para sanar este erro foram demandado cerca de 3 dias, pois além da versão do *php* a porta padrão no código não era a mesma que chamava o *root* no *WorkBench*.
- *MySQL*, senha incorreta do *root*., foi criado uma senha do *root* na fase de pesquisa para criação deste TCC, a mesma foi acidentalmente esquecida, para recuperar esta senha e assim prosseguir com o desenvolvimento da aplicação foi demandado cerca de uma semana, pois mesmo instalando e desinstalando o *WorkBench*, restaurando sistema, alterando registro, a senha do *WorkBench* continua inalterada, para isso, fez-se necessário a alteração de um arquivo *my.ini* que encontrava-se numa pasta oculta do sistema, após a alteração deste arquivo, foi possível criar uma nova senha.
- *Docker*, foram dezenas de erros encontrados no geral com o *Docker*, pois este conceito foi totalmente novo para a dupla de alunos, desde a introdução até a resolução e execução da aplicação fazendo uso de servidores *Apache PHP* e *MySQL* utilizando unicamente a ferramenta *Docker*, foram demandados cerca de 2 árduos meses.
- No *Docker Swarm* com *AWS*, foram encontrados dois desafios principais. O primeiro, foi alterar o endereço de conexão padrão utilizado pelo *Swarm* da rede privada do *AWS* para a rede pública. O item foi solucionado com a inclusão do parâmetro "*--advertise-addr*" comando *swarm init*. O segundo, foi a necessidade de realizar um redirecionamento de porta no roteador de borda da infraestrutura local, uma vez que as máquinas virtuais utilizadas na simulação estão em uma rede privada atrás de um *NAT*, não visível pela *AWS*.

5 CONCLUSÃO

Como descrito nos capítulos anteriores, este trabalho teve como objetivo geral propor cenários progressivos de melhorias com o uso da ferramenta *Docker*, paralelos ao desenvolvimento e execução de uma aplicação local (com ênfase em servidores *Apache PHP* e *MySQL*). Essa avaliação se fez necessária perante a importância do desenvolvimento remoto, principalmente pela necessidade de isolamento social decorrente da pandemia de COVID-19, além claro, da praticidade e facilitação de utilizar-se de contêineres *Docker* para implantação de sistemas em vias não locais (remotas).

Para realizar o objetivo deste trabalho, foram executados diversos passos, primeiro, foi apresentada toda a estrutura da aplicação, pois sem ela, este documento não faria sentido. Em seguida, foram explicados os três cenários abordados nesta temática, incluindo as ferramentas utilizadas para tornar real o êxito de cada um.

Houve muita dificuldade para realização descritiva e clara deste objetivo, pois além de se tratar de uma aplicação *PHP* conectada diretamente a um banco de dados *MySQL*, a problemática central localizou-se em migrar a experiência da aplicação para um ambiente de provisionamento e configurá-la em alta disponibilidade com o *Docker Swarm* entre o ambiente local e a nuvem *AWS*. Após muita pesquisa e força de vontade, alcançou-se o objetivo final deste trabalho.

Com este estudo, foi concluído que cada cenário possui uma vantagem em especial, sendo no cenário de ambiente local, a vantagem de custo computacional, segurança e controle, no segundo cenário, foi concluído que com *Docker*, o provisionamento, desenvolvimento e execução da aplicação pode ser feita de forma rápida e integral, com a vantagem de todo time de desenvolvedor conseguir trabalhar na mesma aplicação sem atribuições geradas por sistemas operacionais distintos, *bugs*, e erros de versões. Finalmente no terceiro cenário, foi visto que essa aplicação se torna muito mais funcional sendo realizada com *clusters* de alta disponibilidade trazidos pelo *Swarm mode*, um componente do *Docker*.

Como trabalhos futuros podemos citar uma análise de segurança e do isolamento de recursos providos pelo *Docker*. Pois em atualização recente, o *Docker* implementou novos mecanismos de segurança e isolamento. Com esses

isolamentos é possível mitigar o ataque *Fork Bomb* que é um ataque de negação de serviço onde um processo cria cópias de si, indefinidamente, causando lentidão e até travamento de servidores.

6 REFERÊNCIAS

AGARWAL, Nitin. **Understanding the Docker Internals**. 2017. Disponível em: <https://medium.com/@BeNitinAgarwal/understanding-the-docker-internals-7ccb052ce9fe>. Acesso em: 15 mai. 2021.

AQUASEC. **Docker Containers vs. Virtual Machines**. Disponível em: <https://www.aquasec.com/cloud-native-academy/docker-container/docker-containers-vs-virtual-machines/>. Acesso em: 15 mai. 2021.

CHEN, Rocky. **Understand Dockerfile**. Disponível em: <https://medium.com/swlh/understand-dockerfile-dd11746ed183>. Acesso em: 15 mai. 2021.

DEPLOY to Swarm: Describe apps using stack files. Disponível em: <https://docs.docker.com/get-started/swarm-deploy/>. Acesso em: 15 mai. 2021.

DOCKER. **Command-line reference**: docker swarm join. Disponível em: https://docs.docker.com/engine/reference/commandline/node_ls/. Acesso em: 15 mai. 2021.

DOCKER. **Docker overview. docker docs**. Disponível em: <https://docs.docker.com/get-started/overview/>. Acesso em: 15 mai. 2021.

DOCKER. **docker swarm init**: Referência de linha de comando. Disponível em: https://docs.docker.com/engine/reference/commandline/swarm_init/. Acesso em: 15 mai. 2021.

IXC SOFT. **O que é um Hypervisor para Virtualização Profissional**. Disponível em: http://wiki.ixcsoft.com.br/index.php/O_que_%C3%A9_um_Hypervisor_para_Virtualiza%C3%A7%C3%A3o_Profissional. Acesso em: 15 mai. 2021.

MICROSOFT AZURE. **O que é uma VM (máquina virtual)?**. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-a-virtual-machine/>. Acesso em: 15 mai. 2021.

PAVLOVIC, Uros; ATKINS, Gregory. **Docker: Top 7 Benefits of Containerization**. 2020. Disponível em: <https://hentsu.com/docker-containers-top-7-benefits/>. Acesso em: 15 mai. 2021.

UNDER CONTROL. **O que é bare-metal?**. 2017. Disponível em: <https://under.com.br/o-que-e-bare-metal/>. Acesso em: 10 abr. 2021.

Raj. **Working with Docker Images**. 2018. Disponível em: <https://www.itzgeek.com/how-tos/linux/working-with-docker-images-building-docker-images.html>. Acesso em: 15 mai. 2021.

ROHRER, Jim. **Docker Swarm Mode on AWS**. 2017. Disponível em: <https://stelligent.com/2017/02/21/docker-swarm-mode-on-aws/>. Acesso em: 15 mai. 2021.

ROMERO, Daniel. **Containers com Docker: Do desenvolvimento à produção**. Editora Casa do Código, v. 3, f. 64, 2015. 127 p.

RO TSAERT, Gunter. **Docker Layers Explained**. 2019. Disponível em: <https://dzone.com/articles/docker-layers-explained>. Acesso em: 15 mai. 2021.

NETAPP. **What are containers?**. 2021. Disponível em: <https://www.netapp.com/devops-solutions/what-are-containers/>. Acesso em: 10 abr. 2021.

NETAPP. **Containers vs. Virtual Machines (VMs): What's the Difference?**. 2018. Disponível em: <https://blog.netapp.com/blogs/containers-vs-vms/>. Acesso em: 10 abr. 2021.

MINCOV, Reinaldo. **Bare metal vs. virtual servers: Which choice is right for you?**. 2014. Disponível em: <https://www.ibm.com/blogs/cloud-computing/2014/07/25/bare-metal-vs-virtual-servers-choice-right/>. Acesso em: 10 abr. 2021.

7 GLOSSÁRIO

AWS	Plataforma de serviços de computação em nuvem da Amazon.
Patches	Atualizações ou correções de software que visam melhorar sua usabilidade, segurança ou performance.
Bugs	Falhas que ocorrem na execução algum software ou hardware.
PHP	Linguagem de script open source de uso geral.
MySQL	Sistema de gerenciamento de banco de dados, que utiliza a linguagem SQL como interface.
Nuvem	Rede global de servidores que fornece recursos de computação sob demanda por meio da Internet.
Framework	Conjunto de códigos prontos que podem ser usados no desenvolvimento de aplicativos e sites.
Apache	Servidor web HTTP de código aberto.
Banco de dados	Coleção organizada de informações - ou dados - estruturadas, normalmente armazenadas eletronicamente.
Sistema operacional	Principal software do computador, responsável pelo gerenciamento do uso dos dispositivos e programas.
Script	Conjunto de instruções para que uma função seja executada em determinado aplicativo.
Banco de dados	Coleção organizada de informações - ou dados - estruturadas, normalmente armazenadas eletronicamente.
Kernel	Núcleo do sistema operacional.

CPU	Principal item de hardware do computador, também conhecido como processador.
Bare-metal	Servidor físico, que possui um sistema operacional instalado diretamente em seu disco.
Hypervisor	Software que cria e executa máquinas virtuais
DevOps	Conjunto de ideias e práticas mais colaborativa e produtiva entre equipes de desenvolvimento e operações.
Open Source	Software de código aberto com o código fonte disponibilizado e licenciado livremente para qualquer um e para qualquer finalidade.
Windows	Sistema operacional criado pela empresa Microsoft
Linux	Sistema operacional que utiliza kernel Linux
Ubuntu	Sistema operacional construído a partir do núcleo Linux baseado em Debian
Cluster	Sistema de computadores que trabalham da mesma forma para prover redundância
NAT	Técnica utilizada para traduzir endereços IP
Software	Aplicação comum qualquer
Hardware	Componente físico da máquina