FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

João Pedro Cayres Bosco

Docker como ferramenta para construção de um ambiente em nuvem

Americana, SP

2019

FACULDADE DE TECNOLOGIA DE AMERICANA

Curso Superior de Tecnologia em Segurança da Informação

João Pedro Cayres Bosco

Docker como ferramenta para construção de ambiente em nuvem

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da informação, sob a orientação do Prof.(º) Me. Rossano Pablo Pinto

Área de concentração: Segurança da Informação

Americana, SP.

2019

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

B753d BOSCO, João Pedro Cayres

Docker como ferramenta para construção de ambiente em nuvem. / João Pedro Cayres Bosco. – Americana, 2019.

54f.

Monografia (Curso Superior de Tecnologia em Segurança da Informação) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Ms. Rossano Pablo Pinto

1. Computação em nuvens I. PINTO Rosano Pablo II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: *681.518*

Fatec Americana
Ministro Ralph Biasi

Centro Paula Souza 50 anos 1969 - 2019

São Paulo Governo do Estado

Faculdade de Tecnologia de Americana

João Pedro Cayres Bosco

Docker como ferramenta para construção de ambiente em nuvem

Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana.
Área de concentração: Segurança da informação

Americana, 12 de junho de 2019.

Banca Examinadora:

Rossano Pablo Pinto
Mestre
Fatec Americana

Benedito Luciano Antunes de França
Mestre
Fatec Americana

Edson Roberto Gaseta
Mestre
Fatec Americana

AGRADECIMENTOS

Em primeiro lugar a minha mãe Erica Simone Cayres, aos professores e alunos da Fatec Americana, em especial ao Gabriel e Silva Botelho e o professor orientador Rossano Pablo Pinto, e a Gabriele Lima, pessoas as quais contribuíram direta e indiretamente no desenvolvimento deste que vos é apresentado.

DEDICATÓRIA

Aos estudantes da área de tecnologia da informação em especial aos que se interessam sobre o tema abordado nas páginas a seguir.

RESUMO

Este Trabalho de conclusão de curso mostra o uso do Docker como uma ferramenta para a construção de um ambiente de nuvem privada. Utilizando containers, um modelo de isolamento de recursos que permite disponibilizar aplicações distribuídas, com uma infraestrutura em cluster, atendendo as necessidades para suportar a condição ideal de execução da aplicação implementada. Existem múltiplos conceitos de isolamento que são tratados ao longo da monografia entre eles estão, *dualboot, hypervisors*, e containers cujo será o foco principal, para desmembrar os fundamentos trabalhados e as ferramentas utilizadas no sistema Linux, a alta disponibilidade e autogerenciamento do estado ideal de execução das aplicações implementadas condizentes com os princípios de disponibilidade. Um orquestrador gerencia dinamicamente as tarefas a serem executadas para manter o ciclo da aplicação ativo e os containers em execução. Cada container possui o seu sistema de arquivos isolado dos sistemas de arquivos do host, não permitindo que os containers modifiquem outro ambiente. O trabalho apresenta as funcionalidades do kernel necessárias à construção de containers, além de diferentes *runtimes* e orquestradores. Por fim, o cenário escolhido testa a eficiência desta tecnologia para prover serviços em nuvem.

Palavras Chave: Isolamento de recursos; Container; Orquestração.

ABSTRACT

This course completion work shows the use of Docker as a tool for building a private cloud environment. Using containers, a resource isolation model that allows distributed applications with a clustered infrastructure to meet the needs to support the ideal execution condition of the implemented application. There are multiple concepts of isolation that are treated throughout the monograph among them are, dualboot, hypervisors, and containers where will be the main focus, to dismember the fundamentals worked and the tools used in the Linux system, the high availability and self-management of the ideal state of execution of implemented applications consistent with the principles of availability. An orchestrator dynamically manages the tasks to be performed to keep the application cycle active and the containers running. Each container has its file system isolated from host file systems, not allowing containers to modify another environment. The work presents the kernel functionalities needed to build containers, as well as different runtimes and orchestrators. Finally, the chosen scenario tests the efficiency of this technology to provide cloud services.

Keywords: Resource isolation; Container; Orchestration.

SUMÁRIO

INTRODUÇÃO.....1

1 CONCEITOS FUNDAMENTAIS 3

1.1 Máquinas Virtuais 3

1.2.1 Hypervisor tipo 1 4

1.2.1 Hypervisor tipo.....5

1.3 Containers.....6

1.4 Containers Linux 8

1.4.1 Namespace.....9

1.4.2 CGroups.....9

1.4.3 Chroot 10

1.5 Tecnologia de contaniers em Linux 10

1.6 Nuvem 12

1.7 Orquestração em nuvem.....15

1.7.1 Kubernetes 16

1.7.2 Apache Mesos 16

1.7.3 Docker Swarm ……17

1.7.4 Orbiter ……..17

2 DOCKER 18

3 SWARM 27

4 CENÁRIO .....32.

CONSIDERAÇÕES FINAIS.....36

REFERÊNCIAS.....37

APÊNDICE.....40

LISTA DE FIGURAS

Figura 1: Hypervisor tipo 1.....5

Figura 2: Hypervisor tipo 2.....6

Figura 3: infraestrutura como serviço 14

Figura 4: Plataforma como serviço.....14

Figura 5: Software como serviço.....15

Figura 6: Docker engine.....19

Figura 7: Arquitetura docker.....20

Figura 8: Arquitetura laboratório Fatec Americana.....33

Figura 9 : Processo de atualização da aplicação 34

Figura 10: app1.0 35

Figura 11: app1.0 35

Figura 12: app1.1.....36

LISTA DE TABELAS

Tabela 1: Instruções Dockerfile.....14

INTRODUÇÃO

A segurança da informação é fruto do desenvolvimento tecnológico. Inicialmente os computadores eram instrumentos de guerra utilizados para comunicação por meio de codificação e decodificação das mensagens enviadas aos aliados e interceptadas dos adversários, entre outras variáveis como o poder de processamento de dados que complementaram as estratégias de combate. Ao final das grandes guerras a tecnologia que fora desenvolvida passou a ser utilizadas no meio corporativo. Posteriormente, com o lançamento dos computadores pessoais, estas tecnologias chegaram aos usuários finais.

A segurança da informação passou a ter um destaque especial após os ataques do 11 de setembro às torres gêmeas, nos Estado Unidos. Nesses ataques algumas empresas do mercado mundial foram destruídas, gerando um ônus não somente financeiro, mas de vidas humanas. Das preocupações que permeiam o meio tecnológico, pode-se destacar na segurança da informação três princípios: a confidencialidade, para os ativos estejam acessíveis apenas aos que tem permissão para acessá-los; a integridade do ativo que só possa ser alterado por quem dispuser da permissão para que o faça; da disponibilidade que o ativo esteja acessível o maior período possível para que todos tenham acesso no momento desejado. Este trabalho aborda o conceito de isolamento de recursos através de sistemas operacionais, no qual se destaca a tecnologia de containers, principalmente nos sistemas operacionais Linux.

O **objetivo geral** foi construir uma nuvem privada utilizando containers.

Como **objetivos específicos** utilizamos o Docker como *runtime* para construir uma nuvem privada utilizando containers em sistemas operacionais Linux, bem como orquestrar a nuvem por meio do Docker Swarm.

‘

A metodologia escolhida para a pesquisa foi descrever as tecnologias de isolamento qualificando o desempenho dos containers em hospedagem de aplicações em um modelo que oferece quase todos os recursos para ser classificado como um PaaS.

O trabalho foi estruturado em 4 capítulos. No Capítulo 1, são apresentados os fundamentos de isolamento de recursos. O Capítulo 2 apresenta o *runtime* de containers que foi utilizado em laboratório. Já no Capítulo 3, é explicado o funcionamento da orquestração de cluster. O Capítulo 4 descreve o cenário construído para testar e expor os dados obtidos com a infraestrutura e modulação da aplicação. Com base nas informações conseguidas a partir dos estudos realizados nos capítulos anteriores, a ultima parte do trabalho se reserva às considerações finais.

1 CONCEITOS FUNDAMENTAIS

Os fundamentos apresentados neste capítulo servem como base para a o desenvolvimento da pesquisa, de modo a esclarecer como a tecnologia da informação permite a construção de um ambiente privado de nuvem. Entre os principais conceitos estão: isolamento de recursos nos sistemas Linux com o uso de cgroups, namespace e isolamento dos sistemas de arquivos. Também se utilizou frameworks para manipulação de containers, bem como as ferramentas disponíveis para gerenciamento e construção da infraestrutura de hospedagem de serviços e orquestração.

1.1 MÁQUINAS VIRTUAIS

O isolamento de recursos é um conceito em sistemas que permite utilizar um mesmo hardware para construir e manipular ambientes distintos, dividindo os recursos entre mais de um sistema. O *dualboot* é uma alternativa mais primitiva, utilizado para executar mais de um sistema em uma mesma máquina, porém em tempos diferentes, permitindo que o usuário manipule os ambientes somente um por vez, embora seja uma opção existente quando a necessidade se dá na variedade de ambiente, quando o uso dos ambientes precisa ser simultâneo este modelo de isolamento não é eficiente. Para atender a este tipo de cenário foram desenvolvidos os *hypervisors* contemplados nas próximas duas seções a seguir, hypervisor tipo 1 e hypervisors tipo 2.

Máquina virtual (TANENBAUM, 2015) é uma tecnologia que vem sendo desenvolvida há mais de meio século. A partir da década de 60 tivemos muitas mudanças na tecnologia dos *hypervisors,* discutidos mais na próxima seção.

Segundo Tanenbaum (2015, p.) :

> [...]Com as máquinas virtuais agora disponíveis, um desenvolvedor de software pode construir cuidadosamente uma máquina virtual, carregá-la com o sistema operacional exigido, compiladores, bibliotecas e

‘

> código de aplicação, e congelar a unidade inteira, pronta para executar. Essa imagem de máquina virtual pode então ser colocada em um CD-ROM ou um website para os clientes instalarem ou baixarem. Tal abordagem significa que apenas o desenvolvedor do software tem de compreender todas as dependências. Os clientes recebem um pacote completo que funciona de verdade, completamente independente de qual sistema operacional eles estejam executando e de que outros softwares, pacotes e bibliotecas eles tenham instalado.

Segundo Tanenbaum (2015, p.341), uma grande vantagem das máquinas virtuais é o fato da sua portabilidade e modularidade sem perder a consistência no trabalho desenvolvido sobre o sistema e suas aplicações, bibliotecas e serviços, permitindo compartilhar uma imagem virtual confiável e segura para que possa funcionar em outras máquinas hospedeiras.

1.2.1 Hypervisor tipo 1

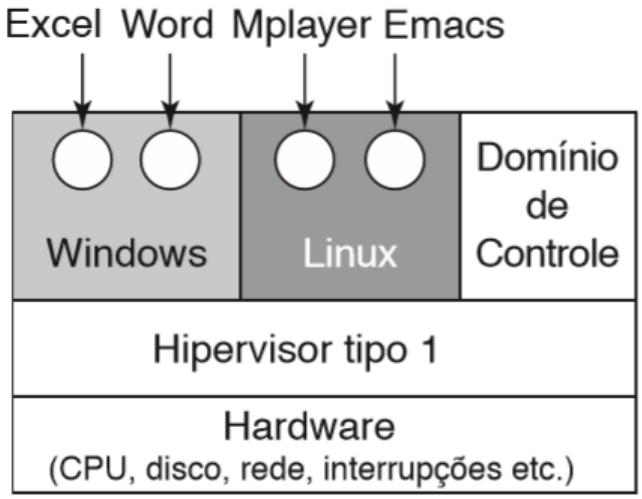
O modelo de operação do *hypervisor* tipo 1 permite disponibilizar múltiplos sistemas operacionais distintos em uma mesma máquina *host*. Neste modelo, o *hypervisor* está na camada que antecede os sistemas operacionais, portanto independe dos sistemas utilizados.

Tanenbaum (2015) cita Goldber (1972) em seu fundamento de *hypervisors (a figura* 7.1(a), citada no texto, refere-se a Figura 1*)*:

> Goldberg (1972) distinguiu entre duas abordagens para a virtualização. Um tipo de hipervisor, chamado de hipervisor tipo 1 está ilustrado na Figura 7.1(a). Tecnicamente, ele é como um sistema operacional, já que é o único programa executando no modo mais privilegiado. O seu trabalho é dar suporte a múltiplas cópias do hardware real, chamadas máquinas virtuais, similares aos processos que um sistema operacional normalmente executa. (GOLDER, 1972, Aprid TANENBAUM, 2015, p.)

‘

Figura 1 – Hypervisor tipo 1



Fonte: Tanenbaum (2015, p.)

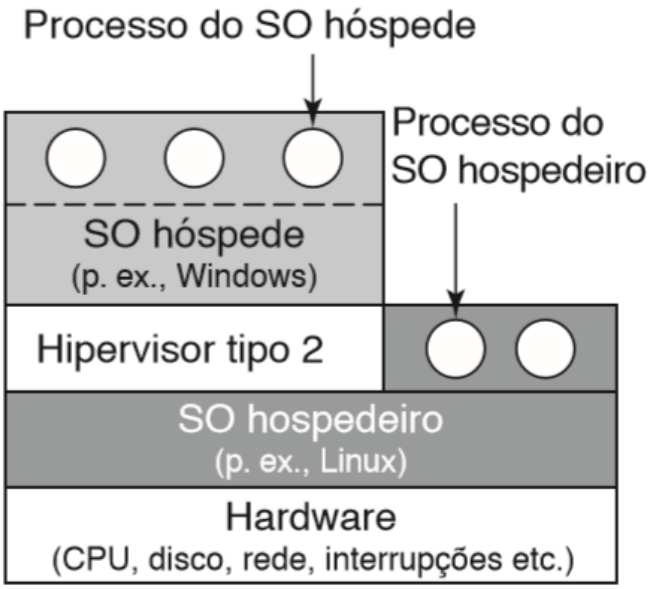
1.2.2 Hypervisor tipo 2

O *hypervisor* do tipo 2 é um modelo de virtualização que permite utilizar múltiplos sistemas operacionais distintos simultaneamente por meio de um virtualizador na camada que sucede o sistema operacional. Portanto é uma aplicação instalada em um sistema *host* que possibilita instanciar outros sistemas operacionais. Tanenbaum (2015) descreve o segundo modelo de *hypervisor* (a figura 7.1(b), citada no texto, refere-se a Figura 2) :

> Em comparação, um hipervisor tipo 2, mostrado na Figura 7.1(b), é um tipo diferente de animal. Ele é um programa que depende do, digamos, Windows ou Linux para alocar e escalonar recursos, de maneira bastante similar a um processo regular. É claro, o hipervisor tipo 2 ainda finge ser um computador completo com uma CPU e vários dispositivos. Ambos os tipos de hipervisores devem executar o conjunto de instruções da máquina de uma maneira segura. Por exemplo, um sistema operacional executando sobre o hipervisor pode mudar e até bagunçar as suas próprias tabelas de páginas, mas não as dos outros.(TANENBAUM, 2015, p.)

‘

Figura 2 - Hypervisor tipo 2



Fonte: Tanenbaum (2015, p.)

O *hypervisor* tipo 2, assim como os containers tratados na próxima seção, utilizam frameworks sobre um sistema operacional, enquanto o *hypervisor* tipo 1 fica uma camada abaixo do sistema operacional.

1.3 Containers

Segundo Petazzoni (2019, p.) containers são parecidos com máquinas virtuais, possui espaço de PIDs independente, bem como interface de rede isolada. Ambos podem executar ações como *root*, instalar pacotes, executar serviços, alterar as relações de roteamento e *iptables*. Embora tenham particularidades em comum, máquinas virtuais e containers não são a mesma coisa e por conta disso logo possuem diferenças elementares, tais como: containers utilizam o *kernel* do *host*, não podem *bootar* um *kernel* diferente do *host,* não possuem permissão para carregar módulos do *kernel* independentes

‘

do host, não precisam de que o primeiro processo seja o *init,* não precisam de *syslog* e *cron*.

De acordo com a descrição disposta no site oficial da documentação do Docker (2019a) :

> "Um contêiner é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de maneira rápida e confiável de um ambiente de computação para outro. Uma imagem de contêiner do Docker é um pacote de software leve, autônomo e executável que inclui tudo o que é necessário para executar um aplicativo: código, tempo de execução, ferramentas do sistema, bibliotecas do sistema e configurações. As imagens de contêiner se tornam contêineres no *runtime*, no caso de contêineres do Docker, as imagens se tornam contêineres quando são executadas no Docker Engine [...]"

O grupo Redhat (2019a) também disponibiliza sua definição de container:

> Um container Linux é um conjunto de um ou mais processos organizados isoladamente do sistema. Todos os arquivos necessários à execução de tais processos são fornecidos por uma imagem distinta. Na prática, os containers Linux são portáteis e consistentes durante toda a migração entre os ambientes de desenvolvimento, teste e produção. Essas características os tornam uma opção muito mais rápida do que os pipelines de desenvolvimento, que dependem da replicação dos ambientes de teste tradicionais.

Atualmente, A Amazon Web Services (AWS, 2019b) é o mais popular provedor de serviços em nuvem tem sua definição de containers:

> Os containers proporcionam uma maneira padrão de empacotar código, configurações e dependências de seu aplicativo em um único objeto. Eles compartilham um sistema operacional instalado no servidor e são executados como processos isolados de recursos. Isso permite fazer implantações rápidas, confiáveis e consistentes, independentemente do ambiente. A Nuvem AWS oferece recursos de infraestrutura otimizados para a execução de containers, além de um conjunto de serviços de orquestração que facilitam a criação e execução de aplicativos conteinerizados em produção.

Containers estão em diversas plataformas, como em nuvem, datacenters, sistemas Windows e Linux. A próxima seção trata de containers Linux.

‘

1.4 Containers Linux

De acordo com os relatos históricos na documentação do Docker (2019a) e na página oficial do Redhat (2019a) a história dos containers Linux se iniciou no FreeBSD no ano 2000 por meio do conceito de “*jails*”. Eles particionam o sistema *host* isolando recursos para que outros usuários não administradores utilizem o sistema de maneira mais segura. Esse conceito utiliza o isolamento do sistema de arquivos para dividir a raiz dos usuários da raiz do sistema de arquivos principal do *host*.

Em 2001 Jacques Gélinas introduziu o conceito para Linux com seu projeto VServer onde definiu a execução como “vários servidores Linux de uso geral em uma única caixa com grau elevado de independência e segurança”(REDHAT, 2019a). Em pouco tempo depois o cgroups e o systemd passaram a integrar a tecnologia de containers. O cgroups (seção 2.4.2) é responsável por estabelecer os limites dos recursos que serão disponibilizados para os containers (cpu, memória e rede) e o systemd, sistema de inicialização de serviços que controla processos. A segunda evolução no conceito de contêiner é de responsabilidade do namespace (seção 2.4.1), funcionalidade do kernel Linux responsável por isolar a árvore de processos dos containers e os objetos do sistema, possibilitando que um usuário tenha privilégios na execução de containers sem ter esses privilégios no sistema host. O próximo passo foi a criação de uma biblioteca oficial de containers em Linux denominada LXC (seção 2.5).

‘

1.4 1 Namespace

Namespace é uma das funcionalidades do kernel Linux que possibilita a construção dos containers. Este é responsável por gerenciar o isolamento dos objetos do host e dos objetos dos containers. Uma das funções é dividir a árvore de processos do host e do container.
Segundo o manual do programador Linux (MAN PAGE PROJECT, 2019d), namespace é definido como:

> Um namespace envolve um recurso de sistema global em uma abstração que faz parecer aos processos dentro do namespace que eles têm sua própria instância isolada do recurso global. Alterações para os recursos global são visíveis para outros processos que são membros do namespace, mas são invisíveis para outros processos. Um uso de namespaces é implementar contêineres [...]
>
> User_namespaces
>
> Pid_namespaces
>
> Network_namespaces .

1.4.2 Cgroups

O CGroups é uma outra funcionalidade importante do kernel Linux para a funcionalidade dos containers. Responsável por dimensionar o hardware disponível no host, é quem determina a quantidade de memória RAM que cada container terá disponível para uso, assim como o uso da CPU para o processamento e da memória para armazenar os dados.

Segundo o manual do programador Linux (MAN PAGE PROJECT, 2019a) que define o fundamento do CGROUPS

> “ cgroups - grupos de controle do Linux
>
> Grupos de controle, geralmente chamados de cgroups, são um kernel do Linux recurso que permite que os processos sejam organizados em grupos hierárquicos cujo uso de vários tipos de recursos pode ser limitado e monitorado. A interface cgroup do kernel é fornecida por meio de pseudo-sistema de arquivos chamado cgroupfs. O agrupamento é implementado no código núcleo do cgroup, enquanto o rastreamento de recursos e os limites são implementado em um conjunto de subsistemas por tipo de recurso (memória, CPU,
> e assim por diante).[...] “

‘

1.4.3 Chroot

Chroot é responsável por isolar a raiz do sistema de arquivos do host do diretório raiz dos containers.

De acordo com as especificações do manual do programador Linux (MAN PAGE PROJECT 2019b) define o chroot como :

> chroot - muda o diretório raiz
>
> **#include <unistd.h>**
>
> **int chroot (** *caminho* **const char *);**
>
> Requisitos de macro de teste de recurso para a glibc (consulte feature_test_macros (7)):
>
> **chroot** ():
> Desde glibc 2.2.2:
> _XOPEN_SOURCE &&! (_POSIX_C_SOURCE> = 200112L)
> || / * Desde glibc 2.20: * / _DEFAULT_SOURCE
> || / * Versões da Glibc <= 2.19: * / _BSD_SOURCE
> Antes da glibc 2.2.2: nenhum
>
> **chroot** () muda o diretório raiz do processo de chamada para aquele especificado no *caminho* . Este diretório será usado para nomes de caminho começando com / . O diretório raiz é herdado por todos os filhos de o processo de chamada. [...]“

Estas três funcionalidades dos sistemas Linux trabalhando em conjunto criam o conceito de contêiner, tema da próxima seção.

1.5 Tecnologias de container em Linux

Existem diversas formas de implementar a tecnologia de containers em uma infraestrutura, a mais independente seria utilizar os conceitos tratados anteriormente e desenvolver um modelo de construção de um container manualmente. Porém, existem algumas bibliotecas para facilitar o desenvolvimento de uma solução utilizando containers no Linux as quais serão apresentadas a seguir.

‘

1.5.1 LXC /LXD

Esta é a primeira biblioteca disponibilizada pelo Linux para containers utilizada inicialmente pelos demais *runtimes*. Segundo o Manual do programador Linux (MAN PAGE PROJECT 2019c) quem define o funcionamento das bibliotecas do LXC:

> A tecnologia de contêiner está sendo ativamente empurrada para o kernel Linux. Ele fornece gerenciamento de recursos por meio de grupos de controle e isolamento de recursos por meio de namespaces. **lxc** , pretende usar essas novas funcionalidades para fornecer um objeto contêiner do espaço do usuário que forneça isolamento total de recursos e controle de recursos para um aplicativo ou um sistema completo. **O lxc** é pequeno o suficiente para gerenciar facilmente um contêiner com linhas de comando simples e completo o suficiente para ser usado para outras finalidades. (LINUX MAN, lxc)

LXD

O modelo de execução do LXD é muito semelhante a uma máquina virtual, como destaca os desenvolvedores:

> O LXD é um gerenciador de contêineres de próxima geração. Ele oferece uma experiência de usuário semelhante às máquinas virtuais, mas usando contêineres Linux. Sua imagem é baseada em imagens pré-criadas disponíveis para um grande número de distribuições Linux e é construída em torno de uma API REST muito poderosa, mas bastante simples. [...] O projeto LXD foi fundado e atualmente é liderado pela Canonical Ltd com contribuições de uma série de outras empresas e colaboradores individuais. (LXD)

1.5.2 Docker

Principal *runtime* de containers, utiliza o Containerd para criar as aplicações desenvolvidas utilizando containers. De acordo com a visão geral dos desenvolvedores o Docker (2019d) pode ser definido como:

> [...] uma plataforma aberta para desenvolvimento, envio e execução de aplicativos. O Docker permite separar seus aplicativos de sua infraestrutura para que você possa entregar software rapidamente. Com o Docker, você pode gerenciar sua infraestrutura da mesma maneira que gerencia seus aplicativos. Aproveitando as metodologias do Docker para enviar, testar e implantar código

‘

> rapidamente, você pode reduzir significativamente o atraso entre a gravação do código e sua execução na produção.

1.5.3 Sytemd-nspawn

Segundo a análise feita pelos desenvolvedores do CoreOS (REDHAT 2019b). O Systemd-nspawn é um *runtime* desenvolvido com o propósito de executar processos dentro de containers Linux, não tem uma *daemon* centralizada, não tem autonomia para fazer *download* de imagens e não tem controle do uso de recursos como *cpu*, memória e armazenamento pois utiliza apenas as funcionalidades do namespace.

1.5.4 RKT

Segundo a documentação do CoreOS, o RKT (REDHAT, 2019b) é um *runtime* sem uma *daemon* centralizada para facilitar a sua integração com sistemas init como systemd e *upstart*, faz download e executa imagem de containers, inclusive containers Docker.

As tecnologias de containers são base para diversos serviços, entre eles o serviço de nuvem, tópico trabalhado na seção a seguir.

1.6 Nuvem

Segundo a descrição dos provedores de serviços em *cloud* (MICROSOFT AZURE 2019) Nuvem é a disponibilidade de recursos de infraestrutura, serviços, e gerenciamento inteligente de aplicações via rede de modo a garantir a alta disponibilidade.

Segundo Pedrosa e Nogueira (2019) a computação em nuvem pode ser definida como:

> [...] um novo modelo de computação que permite ao usuário final acessar uma grande quantidade de aplicações e serviços em

‘

> qualquer lugar e independente da plataforma, bastando para isso ter um terminal conectado à “nuvem”. [...] (PEDROSA; NOGUEIRA, ano, p.)

Infraestrutura como Serviço - IaaS

Ainda seguindo a ideia fundamentada por Pedrosa e Nogueira (PEDROSA; NOGUEIRA, 2019, p) a classe IaaS ilustrado na Figura 3 :

> “[...] são oferecidos os serviços de infra-estrutura sob demanda, isto é, oferece recursos “de hardware” virtualizados como computação, armazenamento e comunicação. Este tipo de serviço prove servidores capazes de executar softwares customizados e operar em diferentes sistemas operacionais. Possui uma aplicação que funciona como uma interface única para a administração da infra-estrutura, promovendo a comunicação com hosts, switches, roteadores e o suporte para a inclusão de novos equipamentos. Por se tratar da camada inferior, esta também é responsável por prover a infra-estrutura necessária para as camadas intermediária e superior. “ [computação em nuvem]

Figura 3 : Infraestrutura como serviço

Fonte: Microsoft Azure, 2019

Plataforma como Serviço - PaaS

‘

Tendo como base o modelo de serviço em nuvem Infraestrutura como serviço, Pedrosa e Nogueira (PEDROSA; NOGUEIRA, 2019, p) definem outro modelo de serviço em nuvem mais abrangente ilustrado na Figura 4:

> Esta é a camada intermediária. É oferecido como serviço um ambiente no qual o desenvolvedor pode criar e implementar aplicações sem ter que se preocupar em saber quantos processadores ou o quanto de memória esta sendo usada para executar a tarefa. Utilizando-se da camada inferior, fornece uma infra-estrutura com alto nível de integração compatível com diversos sistemas operacionais, linguagens de programação e ambientes de desenvolvimentos. [computação em nuvem]

Figura 4 : Plataforma como serviço

Fonte: Azure, 2019

Software como Serviço – SaaS

Pedrosa e Nogueira (2019) definem um terceiro modelo caracterizando a classe de serviços disponível para computação em nuvem ilustrado na Figura 5:

> A camada mais alta da arquitetura da computação na nuvem tem a responsabilidade de disponibilizar aplicações completas ao usuário final. Este acesso é provido pelos prestadores de serviço através de portais web, sendo completamente transparente ao usuário, o que permite a execução de programas que executam na nuvem a partir de uma máquina local. Para oferecer esta transparência, o SaaS utiliza-se das duas camadas inferiores, o PaaS e o IaaS. (PEDROSA; NOGUEIRA, 2019, p)

‘

Figura 5: Software como serviço

Fonte: Azure, 2019.

1.7 Orquestrações na Nuvem

De acordo com a documentação, orquestração (DOCKER, 2019c) é a capacidade de gerenciar os recursos disponíveis de acordo com a demanda, de modo automático, seja no nível de aplicação instanciando mais ou menos containers nos *hosts* ou no nível estrutural instanciando mais ou menos nós para o cluster.

Segundo a descrição dos desenvolvedores (DOCKER, 2019c; KUBERNETES, 2019) das ferramentas Docker e Kubernetes, o Docker Swarm é um orquestrador para ambientes construídos em containers com menos enfoque na infraestrutura fazendo a orquestração do ciclo de vida da aplicação, seu número de réplicas, garantindo que estará funcionando no nível ideal. Enquanto o Kubernetes é um orquestrador com foco em dimensionar e gerenciar a infraestrutura física que hospeda os containers.

1.7.1 Kubernetes

Os desenvolvedores do Kubernetes (2019) o definem como sendo uma plataforma *open source* de orquestração de containers que automatiza a implantação, monitoramento, dimensionamento e gerenciamento de aplicações

‘

em containers. A ferramenta foi desenvolvida pelo Google, funciona com uma estrutura padrão de cluster, de modo que os nós gerenciadores delegam as cargas de trabalho para os nós agentes ou trabalhadores.

1.7.2 Apache Mesos

Segundo a documentação oficial, o Mesos (2019) trabalha com uma hierarquia de mestre e agente onde o mestre determina os recursos e as tarefas de cada nó do cluster. O framework do Mesos é subdividido em duas partes fundamentais, a primeira é o planejador responsável por se registrar no gerenciador para receber os recursos e a segunda é iniciar os nós agentes para executar as tarefas delegadas no framework.

Após inicialização do cluster, o gerenciador determina o valor dos recursos de cada agente, enquanto o planejador seleciona os recursos de acordo com o serviço que o agente executará.

1.7.3 Docker swarm

Segundo a documentação oficial do Docker (2019c), o Swarm é um modelo de orquestrador nativo do Docker *engine.* Funciona com uma hierarquia *master* e *workers*, que permite a criação de um cluster para executar aplicações em containers. A arquitetura disponibiliza todas as ferramentas do Docker *engine,* como dockerfile, compose. O Swarm é responsável por inicializar o cluster e gerenciar a quantidade de nós *workers* a aplicação instanciará para atender o ambiente ideal de funcionamento. O Docker Swarm não possui um gerenciamento vertical referente aos nós do cluster, portanto não adiciona dinamicamente novos nós ao cluster.

1.7.4 Orbiter

A documentação do Orbiter (ARBEZZANO, 2019) está disponível no Github. O Orbiter é um orquestrador que disponibiliza o auto-escalonamento na

‘

infraestrutura de sistemas que executam o Docker, já que o Docker Swarm carece desta funcionalidade. O Orbiter possui uma API web, portanto a interação acontece via protocolo HTTP.

Os orquestradores operam sobre uma estrutura de containers que são executados por algum *runtime* escolhido. Neste caso o Docker, *runtime* que será melhor apresentado no Capítulo 3.

‘

2 Docker

O Docker (DOCKER, 2019d) é um *runtime* de containers. Atualmente é a maior plataforma em escala global de desenvolvimento, suporte e soluções para aplicações envolvendo containers em todas as fases, desde os testes iniciais até a produção. Os desenvolvedores expõem a versatilidade da plataforma em funcionamento:

> O Docker fornece a capacidade de empacotar e executar um aplicativo em um ambiente isolado vagamente chamado de contêiner. O isolamento e a segurança permitem que você execute muitos contêineres simultaneamente em um determinado host. Os contêineres são leves porque não precisam da carga extra de um hipervisor, mas são executados diretamente no kernel da máquina host. Isso significa que você pode executar mais contêineres em uma determinada combinação de hardware do que se estivesse usando máquinas virtuais. Você pode até mesmo executar contêineres do Docker em máquinas host que são, na verdade, máquinas virtuais! (DOCKER, 2019d)

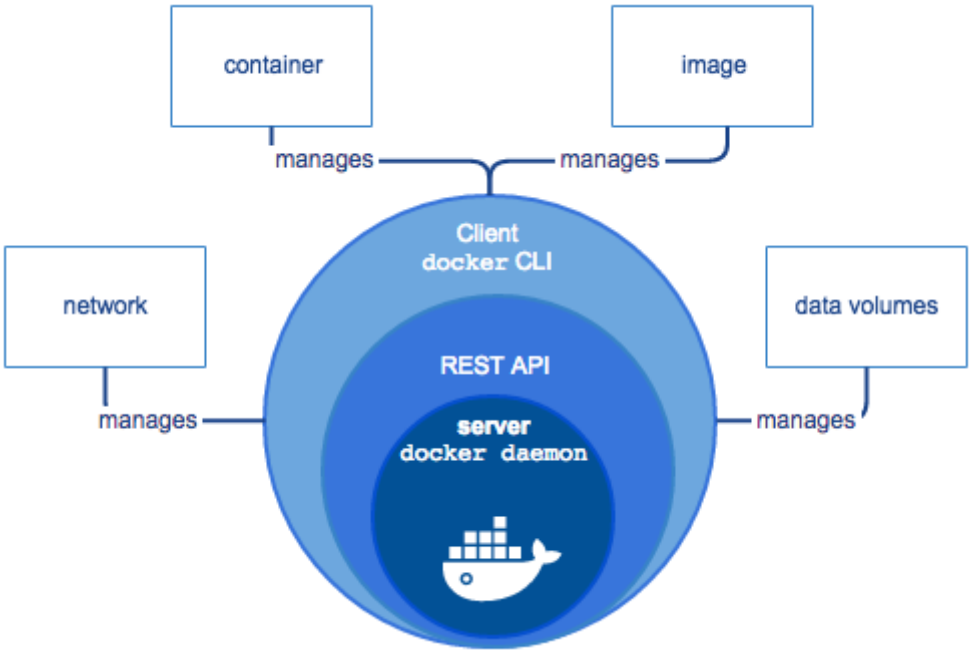
Motor Docker (Docker Engine)

Silva (2017) descreve o Docker *engine,* ilustado na figura 6 :

> O Docker Engine usa uma arquitetura cliente-servidor composta de três componentes: um processo executado em segundo plano (daemon), um cliente e uma API REST. O cliente é a principal interface do usuário com o Docker. Trata-se de uma aplicação binária na qual são executados comandos para interagir com o daemon do Docker. O cliente faz uso da API REST para executar essas operações com o daemon. Uma API REST define um conjunto de funções que os desenvolvedores podem executar solicitações e receber respostas via protocolo HTTP, como GET e POST. A API REST do Docker especifica as interfaces que os programas podem utilizar para interagir com o daemon. O daemon cria e manipula os objetos Docker tais como imagens, contêineres, rede e volumes de dados (SILVA, 2107, p)

‘

Figura 6: Docker engine



Fonte : Docker, 2019d

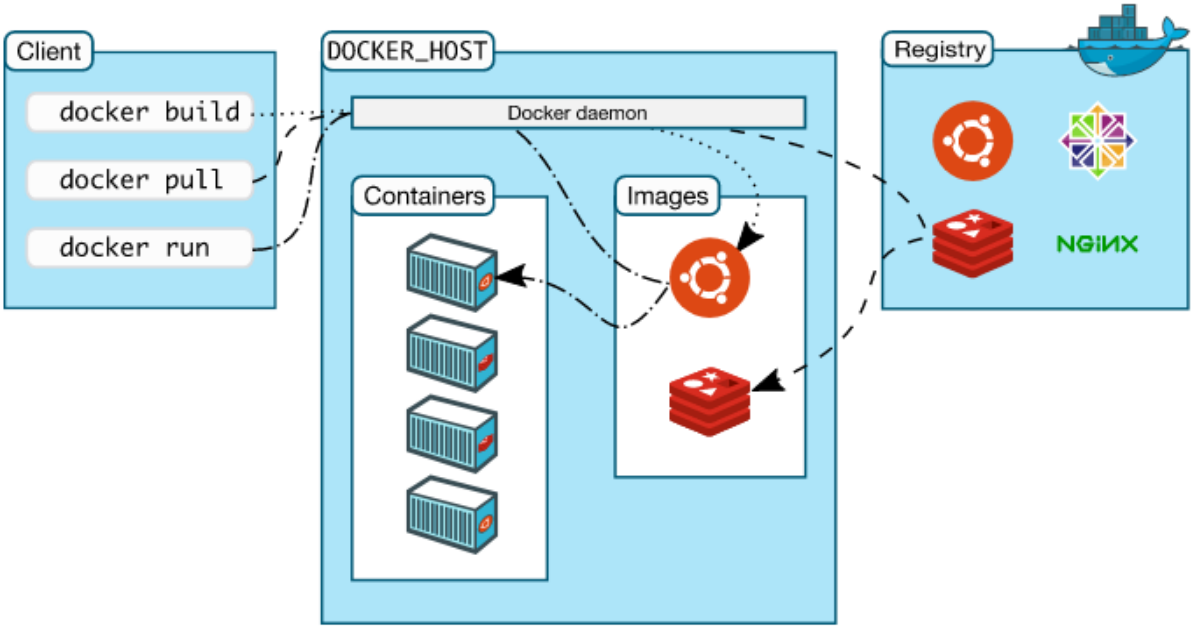
Arquitetura do Docker

O Docker (2019d) foi arquitetado sobre um modelo cliente-servidor ilustrada na Figura 7. O cliente é responsável por solicitar um comando ao Dockerd que executa a função de gerenciar os objetos do Docker, e quem realiza a construção, execução e distribuição dos containers solicitados pelo cliente. Este modelo de execução permite que o cliente e o *daemon* sejam executados no mesmo host ou remotamente.

A comunicação acontece via API REST e interface de rede. Existe um terceiro elemento na arquitetura do Docker chamada de *registry* ou repositório de imagens de containers. Na configuração *default* o Docker vem pronto para quando executar um container cuja imagem não esteja armazenada no host buscar uma imagem base no Docker hub (o repositório online onde existe um compartilhamento coletivo de imagens para serem utilizadas abertamente). Entretanto é possível utilizar um repositório local para imagens confidencias de uso restrito.

‘

Figura 7 : Arquitetura docker



Fonte: Docker, 2019d

Objetos Docker

Os objetos do Docker (2019d) são os elementos que o Docker faz uso para construir os serviços, como imagens, containers e volumes, tratados a seguir, e também redes e plug-ins que serão abordados no Capítulo 4.

- **Imagens -** Imagens são objetos *read only* onde estão as instruções para instanciar um container. O Docker hub dispõe de diversas imagens oficiais que servem de base para outras imagens personalizadas que são criadas no arquivo dockerfile onde estão as etapas de construção de uma imagem. Cada etapa adiciona uma camada à imagen. Ao editar o arquivo dockerfile a aplicação em execução somente adiciona as novas, camadas o que mantem a imagen compacta e leve.

SILVA (2017) descreve o processo executado pelo Docker na criação de uma imagem:

> Cada instrução cria uma nova camada e a adiciona à imagem. As instruções incluem ações como executar um comando de linha, adicionar um arquivo ou pasta, criar uma variável de ambiente e qual

‘

> o processo a ser executado ao iniciar um contêiner criado a partir desta imagem. Estas instruções são armazenadas em um arquivo chamado de Dockerfile. Quando é solicitada uma compilação de uma imagem, o Docker Engine lê esse arquivo, executa as instruções e retorna uma imagem final.(SILVA, 2017, p)

- **Container -** Container é uma instancia em execução que tem como base uma imagem. O Docker permite iniciar, parar, mover e excluir os containers, manipulados pela API ou via cliente, tem permissão para serem conectados em rede, armazenar arquivos, e servirem de base para novas imagens em função do estado atual do container.

- **Volumes -** Volume é o espaço reservado para o armazenamento do conteúdo produzido ou recebido por um container. Pode ser configurado na criação da imagem, posteriormente ou na inicialização do container. Representa um espaço no sistema de arquivos isolado dos arquivos principais do host. O modelo de armazenamento persistido o qual pertence os volumes é o que mantem os dados do container mesmo após a sua parada, por outro lado o modelo não persistido é o que disponibiliza os dados enquanto o container esta em execução sem direito a gravar no disco (modelo conhecido como tpmfs).

- **Registry -** Registry é um repositório de imagens de containers. Os repositórios pode ser público ou privado. No caso de uso de repositórios públicos o Docker tem o Dockerhub, seu repositório oficial.É o diretório em que comando de inicialização de um container " docker run " vai buscar uma imagem caso não exista nenhuma arquivada no host. É possível fazer upload de uma imagem após criá-la. É preciso criar uma conta no Dockerhub para conseguir fazer o *pull* da imagem. É possível criar uma divisão privada dentro do Docker hub isolando suas imagens das pesquisas, porém é possível configurar um diretório compartilhado protegido por firewall e separado da rede externa para garantir a confidencialidade da imagem ,Silva (2017) descreve o funcionamento do registry público e privado no Docker(2019a):

> "Para fazer o download de uma determinada imagem do registro é feito uma operação de (pull) via cliente do Docker. As

‘

> imagens são identificadas por um nome e uma etiqueta (tag). A tag é indicada após o nome da imagem divididas pelo sinal “:”. Se nenhuma tag for fornecida, o Docker Engine usa a tag “latest”. Por padrão, a operação de pull do Docker baixa as imagens do Docker Hub. Para realizar a operação de um registro privado é preciso indicar o endereço desse registro antes do nome da imagem (separados pelo caracter “/”). Um endereço de registo é semelhante a uma URL, mas não contém um especificador de protocolo (https: //).” (SILVA, 219, p)

- **Compose -** O composse (DOCKER, 2019)é uma ferramenta para construir e iniciar aplicações utilizando quantidades variadas de containers de modo que interajam em conjunto isolado de outras aplicações no mesmo host, entre as principais funcionalidades estão :

- Persistência dos volumes após os containers serem criados;
- quando um containers é atualizado copia os dados produzidos na antiga versão permitindo continuar de onde a antiga versão parou;
- Reinicia apenas os containers que forma alterados permitindo rápidas atualizações;
- Interpreta variáveis o que permite que um único container se comporte de maneiras distintas de acordo com a variável em que ele se aplica no momento da inicialização e execução.

o arquivo composse é escrito em linguagem YALM e utiliza *tags* para determinar os parâmetros de execução das aplicações abaixo estão alguns dos parâmetros aceitos pelo composse:

- Version : versão do composse que será utilizada 3.7 é a mais atualizada;
- Réplicas : quantidades de containers em execução;
- Context : caminho para o arquivo dockerfile ou repositório de imagens
- Labels : metadados ;
- Command : substitui o comando padrão do arquivos dockerfile.

Segundo a descrição de Mesias (2014) sobre a funcionalidade e praticidade do compose como ferramenta para o Docker:

‘

> Esse serviço fornece aos desenvolvedores a capacidade de montar aplicativos a partir de contêineres Docker discretos e interoperáveis, completamente independentes de qualquer infraestrutura subjacente, permitindo que pilhas de aplicativos distribuídos sejam implantadas em qualquer lugar e movidas a qualquer momento. Definir uma pilha de aplicativos distribuídos e suas dependências por meio de um arquivo de configuração simples do YAML converte o que era um processo incrivelmente complexo em um processo simples que pode ser executado com apenas alguns toques no teclado. Essa poderosa capacidade significa que os novos aplicativos distribuídos podem ser desenvolvidos em minutos, em contraste com as abordagens convencionais, em que o tempo de colocação no mercado de um aplicativo era medido em meses, senão anos. (MESIAS, 2014)

- **Dockerfile -** O Dockerfile é um arquivo onde estão os parâmetros de construção de uma imagem, utilizando instruções para determinar as características da imagem. Por exemplo, copiar arquivo ou pasta do host, qual processo a ser inicializada ao executar a imagem a porta que será mapeada, qual será o diretório raiz da imagem, as instruções são escritas uma por linha, primeiro a instrução depois o argumento, cada instrução cria uma cada adicional na imagem segue abaixo tabela com as instruções aceitas pelo Dockerfile.

Tabela 1 - Instruções Dockerfile

INTRUÇÃO	OBJETIVO
RUN	A instrução RUN irá executar quaisquer comandos em uma nova camada criada na parte superior da imagem atual e gravar o resultado final. A imagem resultante será usada pela próxima instrução do Dockerfile.
CMD	O objetivo principal da instrução CMD é fornecer o processo padrão para um contêiner de execução. O CMD define o comando a ser processado ao executar a imagem. Esse processo padrão pode ser um software executável. Caso o contêiner execute o mesmo executável sempre, é recomendado utilizar a instrução ENTRYPOINT em combinação com CMD. Se o usuário especificar argumentos para a execução do Docker, eles substituirão o padrão especificado no CMD.
EXPOSE	A instrução EXPOSE informa ao Docker que o contêiner escuta nas portas de rede especificadas em tempo de execução. Essa instrução não torna as portas do contêiner acessíveis ao host. Para

‘

	fazer isso, deve-se usar o sinalizador “-p” para publicar um intervalo de portas ou o sinalizador “-P” para publicar todas as portas expostas. Pode ser exposto um número de porta e publicá-lo externamente em outro número.
ENV	A instrução ENV define uma variável de ambiente por meio de um nome e um valor. Este valor estará disponível no ambiente para todos as instruções subsequentes, Dockerfiles que tiverem essa imagem como base e no contêiner executado a partir dela. Os valores dessas variáveis podem substituídos em tempo de construção e de execução.
ADD/COPY	As instruções ADD e COPY copiam novos arquivos ou diretórios e os adiciona ao sistema de arquivos da imagem no caminho especificado. Vários recursos podem ser especificados, mas se forem arquivos ou diretórios, eles devem ser relativos ao diretório de origem que está sendo construído (o contexto de compilação). O que diferencia as duas instruções é que o COPY é mais performático e o ADD pode também fazer cópias a partir de URLs de arquivos remotos. 2
ENTRYPOINT	Uma instrução ENTRYPOINT permite configurar um contêiner que será executado como um executável. Os argumentos passados para o comando “docker run ” serão anexados após todos os elementos do ENTRYPOINT e substituirão todos os elementos especificados usando CMD. Isso permite que os argumentos sejam passados para o executável definido no ENTRYPOINT. A instrução ENTRYPOINT pode ser substituído usando a opção “–entrypoint” na inicialização do contêiner. Somente a última instrução ENTRYPOINT no Dockerfile terá efeito
VOLUME	A instrução VOLUME cria um ponto de montagem com o nome especificado e marca-o como um volume que pode ser montado externamente no host e por outros contêineres.
USER	A instrução USER define o nome de usuário ou identificação de usuário (UID) para execução de quaiquer instruções RUN, CMD e ENTRYPOINT que virão a seguir no Dockerfile. Esse será o usuário de execução do contêiner
WORKDIR	A instrução WORKDIR define o diretório de trabalho para quaisquer instruções RUN, CMD, ENTRYPOINT, COPY e ADD que o sigam no Dockerfile. Se o WORKDIR não existir, ele será criado
FROM	A instrução FROM define a imagem base a partir da qual a nova imagem será criada. Essa deve ser a primeira instrução no Dockerfile.

Fonte: Silva, 2017

Segundo Silva (2017) o processo que o Docker executa ao construir uma imagem a partir de um dockerfile acontece da seguinte maneira:

> A compilação é executada pelo daemon Docker, não pelo cliente. Assim, a primeira coisa que um processo de construção faz é enviar todo o contexto (recursivamente) para o daemon. Antes que o daemon Docker execute as instruções no Dockerfile, ele executa uma validação preliminar e retorna um erro .

‘

> Se a sintaxe estiver incorreta. Sempre que possível, o Docker irá reutilizar as imagens intermediárias (cache), para acelerar, significativamente, o processo de criação da imagem.”(SILVA, 2017, p)

O modelo eficiente de manipulação dos seus objetos faz do Docker esta ferramenta de alto nível em execução e gerenciamento de containers. Sem nenhuma dificuldade dependendo do uso não é necessária nenhuma configuração é apenas instalar e iniciar a *engine* e já será possível subir um cluster com aplicações de anto nível de execução e disponibilidade, trazendo alternativas diversas para segurança, controle de atualização, e manutenção da aplicação, fazendo uso de recursos importantes para melhor implantar as soluções de acordo com as especificidades de cada etapa otimizando o processo de funcionamento pleno sem dificuldades técnicas, amparado por uma ampla comunidade que opera em prol do desenvolvimento e manutenção deste *runtime*s.

O Docker *engine* é uma ferramenta completa, além de configurar e gerenciar os containers, também tem o seu próprio orquestrador de cluster nativo que será retratado com mais detalhes no Capítulo 4.

‘

3 SWARM

O Swarm (DOCKER, 2019) é um gerenciador e orquestrador de cluster nativo do Docker. Faz parte do Swarmkit, projeto que implementa a camada de orquestração no Docker. O modo Swarm é quando diversos hosts estão executando em conjunto via rede. A arquitetura consiste em *workers* e *managers*, sendo que um mesmo host pode desempenhar ambas as funções. Ao disponibilizar um serviço é estabelecido um estado ideal de execução definido pelo compose. A inserção de novos nós no cluster é feito utilizando o modelo de *token*, onde ao criar o cluster também são criados dois *tokens*, um para acrescentar novos *managers* e outro para novos *workers* mantendo o cluster seguro. Mesias (2014) complementa sobre o Swarm e suas diversas funcionalidades como ferramenta para o Docker *engine*:

> O Docker Swarm é um serviço de cluster nativo do Docker que funciona com os Docker Engines, provisionado pelo novo serviço Docker Machine, e cria um pool de recursos dos hosts nos quais os aplicativos distribuídos são executados. Ao agendar automaticamente cargas de trabalho de contêineres e alocar recursos, o Docker Swarm fornece aos usuários alto desempenho e disponibilidade, eliminando o gerenciamento de recursos manuais ineficiente e propenso a erros. O Docker Swarm é único no setor, pois é projetado especificamente para um ciclo de vida contínuo, desde o desenvolvimento até as operações. Ele é estruturado de forma que um desenvolvedor possa testar seu aplicativo em cluster em algumas máquinas locais, enquanto uma equipe de operações pode usar o mesmo conjunto de ferramentas para dimensionar esse mesmo aplicativo em centenas de hosts em diversas infraestruturas.(MESIAS, 2014)

Nós do Cluster

Nó é uma instância de Docker que participa do cluster. O Swarm trabalha com dois tipos de nós: os *managers* e os *workers*. Obrigatoriamente todo Swarm deve ter um *manager*, por isso ao iniciar o Swarm o host que executa o comando “ docker Swarm init” é designado como *manager*. A sua função no

‘

Swarm é delegar as tarefas para os nós *workers* que são responsáveis por executar essas tarefas e manter o estado do serviço como ideal.

Serviços e Tarefas

Serviços são um conjunto de tarefas a serem executadas em um nó do Swarm. Estrutura central de interação do usuário com o Swarm, ao definir um serviço são selecionados suas especificidades como imagem e programas e comandos que dever ser executados. Para isso existem dois modelos de serviços: os serviços replicados, onde o *manager* define um número específico de réplicas da tarefa entre os nós, tomando como base os parâmetros passados na configuração do estado ideal da aplicação; e os serviços globais, tarefas as quais são executadas em todos os nós ativos no Swarm.

Funcionalidades

O Swarm (DOCKER, 2019c) se destaca pela praticidade, sobre tudo na sua escalabilidade permitindo que uma solução atenda demandas distintas, garantindo o estado ideal de execução. Os serviços são separados por DNS, o que permite balancear a carga dos containers para designar prioridade no tráfego entre as aplicações. A inserção de novos nós é feita de maneira a manter o nível de segurança, utilizando criptografia TLS para proteger a comunicação do cluster. Ao atualizar um container o processo ocorre de forma incremental, método que permite retroceder a versões anteriores em caso de alguma divergência com novas atualizações, o escalonamento no Swarm acontece de maneira parcial. Ocorre em nível de aplicação de modo a instanciar automaticamente novos containers nos *workers* já inseridos no cluster, o que significa que para adicionar novos nós ao cluster de acordo com a demanda somente é possível utilizando um outro orquestrador que ofereça o escalonamento automático. O Swarm utiliza diversos drivers de conexão via rede (Seção 4.2) disponíveis no Docker para manipular e gerenciar da melhor maneira as aplicações e serviços.

‘

3.2 REDES

O Docker (DOCKER, 2019b) dispõe de diversas opções quanto as configurações de modelos de conexão via rede, o que em conjunto com o Swarm e as demais ferramentas, personalizam a aplicação conforme a necessidade e demanda específica de cada solução produzida utilizando containers.

Bridge

É o modelo de onde o tráfego é encaminhado pela camada de enlace dentro de um segmento de rede (DOCKER, 2019b), podendo ser via hardware ou software em execução no kernel do host. É o modelo de conexão que vem selecionado por padrão pelo Docker, útil para ambientes em que os containers precisem se comunicar diretamente.

> Em termos de Docker, uma rede de ponte usa uma ponte de software que permite que contêineres conectados à mesma rede de ponte se comuniquem, ao mesmo tempo em que fornece isolamento de contêineres que não estão conectados a essa rede de ponte. O driver de ponte do Docker instala automaticamente regras na máquina host para que os contêineres em diferentes redes de ponte não possam se comunicar diretamente entre si.(DOCKER, 2019b)

Overlay

A rede de sobreposição constrói uma rede distribuída entre os hosts Docker. Esta rede está hierarquicamente acima das demais redes do host (DOCKER, 2019b), permitindo que os containers se comuniquem com segurança. Este modelo é feito com transparência de roteamento passando pelo host até chegar no container de destino. Ao inicializar o Swarm, ou adicionar um host a um Swarm em execução, duas novas redes são criadas neste host adicionado:

‘

> [...] uma rede de sobreposição chamada ingress, que lida com o tráfego de controle e dados relacionado aos serviços de enxame. Quando você cria um serviço de enxame e não o conecta a uma rede de sobreposição definida pelo usuário, ele se conecta à ingress rede por padrão.uma rede de bridge chamada docker_gwbridge, que conecta o daemon Docker individual aos outros daemons que participam do swarm.

Hostdrive

O modelo de rede host não isola a rede dos containers da rede do host (DOCKER, 2019b), assim, se uma aplicação utiliza a porta 80 para se comunicar instanciada em um container o endereço de acesso será o IP do host. Este modelo de rede está disponível somente na plataforma Linux , a partir da versão 17.06 do Docker esta disponível para o Swarm também esta opção de funcionamento da rede, porém está atrelado a algumas limitações neste modo citado pelos desenvolvedores.

> Nesse caso, o tráfego de controle (tráfego relacionado ao gerenciamento do swarm e do serviço) ainda é enviado por uma rede de sobreposição, mas os contêineres de serviço de enxame individuais enviam dados usando as portas e portas do host do daemon do Docker. Isso cria algumas limitações extras. Por exemplo, se um contêiner de serviço se vincular à porta 80, apenas um contêiner de serviço poderá ser executado em um determinado nó de enxame.

MCvlan

O modelo de Macvlan permite conectar os containers como se estivessem conectados diretamente na rede física (DOCKER, 2019b), através da atribuição de um endereço MAC as interfaces de rede virtuais vinculadas a cada container. Para que isso seja possível é necessário apontar uma interface física no host do Docker para ser utilizado tal como a sub-rede o *gateway* da Macvlan. Existem dois modelos de atuação deste modo: bridge e trunk, descrito na documentação.

> No modo bridge, o tráfego Macvlan passa por um dispositivo físico no host.No modo de porta-tronco 802.1q, o tráfego passa por uma sub-interface 802.1q que o Docker cria em tempo real. Isso permite controlar o roteamento e a filtragem em um nível mais granular.

‘

None

É possível desativar a rede para os containers (DOCKER, 2019b). Caso a aplicação não utilize nenhuma comunicação via rede, é uma das opções para manter a aplicação segura. Neste modelo apenas a interface lógica de loopback é colocada em funcionamento.

Docker machine

O Docker machine é uma ferramenta de virtualização que provisiona hosts de maneira dinâmica para executar as soluções desenvolvidas com os containers Docker. É muito útil para testes que podem ser realizados em uma única máquina como se estivesse rodando em um cluster físico de múltiplos hosts Docker físicos. MESIAS (2014) comenta sobre esta ferramenta da *engine* no modo Swarm:

> Esse serviço expande ainda mais os recursos de portabilidade dos aplicativos distribuídos, oferecendo ao usuário a flexibilidade de provisionar qualquer host com o Docker Engine, seja um laptop, uma VM do datacenter ou um nó na nuvem. Isso poupa um desenvolvedor de um tempo significativo na configuração manual e no script personalizado, resultando em iterações mais rápidas e compactando o ciclo de desenvolvimento para implantação.(MESIAS, 2014)

‘

4 Cenário

O cenário atenderá a modelo de nuvem que se aproxima do PaaS, disponibilizando o controle parcial do ciclo de vida da aplicação, de modo foi possível manter o estado ideal de execução da aplicação. O modelo contruido deixou de fora a autenticação de usuários, bem como o suporte ao desenvolvimento da aplicação. Para realizar os testes do cenário construído foi utilizado 1 laboratório da Fatec Americana.

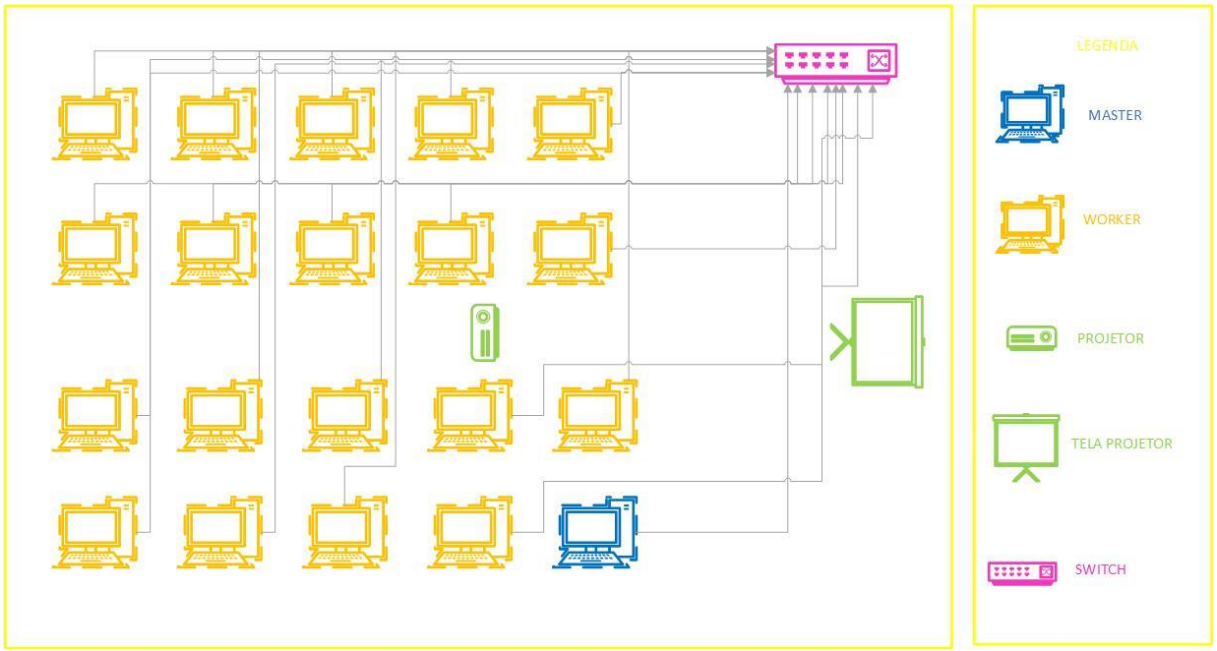
O objetivo deste cenário será analisar a eficiência do Docker *engine* em dois fatores: (1) gerenciar os recursos para manter a disponibilidade das aplicações com base no ambiente de execução ideal, e também (2) a atualização da aplicação acrescentando duas funcionalidades da primeira versão que foi executada alterando o arquivo docker-compose.yml. A primeira versão da aplicação identificará o container em execução que disponibilizou aquele acesso, expondo o ID do container na página web e reconhecendo conexão com o redis (um banco de dados). Posteriormente, na atualização, foi acrescentado um serviço via web chamado *visualizer*. Ele disponibiliza um modelo gráfico. Ilustrando os nós do cluster em execução e identificando os containers e algumas características como: imagem, id, estado atual, tag e a data da última alteração.

Arquitetura do laboratório

O laboratório ilustrado na Figura 8 possui máquinas conectadas em bridge. Na infraestrutura temos um proxy e certificado de autenticação para manter conexão com a rede externa. Cada máquina host dispõe de 8GB de RAM, 500 GB de armazenamento, processador intel i5 de 4 núcleos.

‘

Figura 8 : Arquitetura laboratório Fatec Americana



Procedimentos

Cada máquina no laboratório utilizado para teste está com o Slakware instalado como sistema base. Utilizando o virtual box, instanciamos uma máquina virtual em cada host com Ubuntu e o *daemon* do Docker instalado. Em cada host virtual construiu-se um diretório Dockerfile e um diretório compose na raiz do usuário Docker. Que foi criado para manipular a ferramenta sem a necessidade de executar as tarefas como root. O Dockerfile é o arquivo contido dentro do diretório Dockerfile. Ele armazena as especificações da imagem construída com base em uma imagem Python2.7-Slim oficial, que foi utilizada para os testes, o arquivo docker-compose.ylm que foi armazenado no diretório compose contem os dados de implantação da aplicação. Estes dados foram responsáveis pelo funcionamento da aplicação no estado ideal de funcionamento. Para iniciar a infraestrutura do cluster, que suportará o modelo de nuvem privada, utilizamou-se o Docker para concatenar as informações do arquivo dockerfile construindo a imagem e iniciando os containers. O Docker Swarm, ferramenta que gerencia o cluster e adiciona os nós, faz uso do arquivo compose para inicializar a quantidade de containers necessária para a

‘

aplicação. O nó master inicia o cluster e incluí três *wokers* para iniciar os testes.

App

A aplicação retornara a identificação do container que a executou. Esta aplicação foi construída utilizando o comando “docker build” concatenando aos arquivos da aplicação com os parâmetros estruturais contidos nos arquivo Dockerfile retornando uma imagem como saida. Esta imagem sera executada pelo master após inicializar o cluster com o comando “docker swarm initi” e retornar o token de saída para inserção de mais nós ao cluster, o máster também é responsável por delegar a carga de trabalho para os workers que executam à aplicação conforme descrito nos arquivos de construção da magem, as informações armazenadas no arquivo composse docker-compose.ylm delimitam o ambiente a ser monitorado, número de instancias e as informação do funcionamento geral da aplicação, após verificar os dados redimensionamos a aplicação para executar um numero maior de instancias alterando o parâmetro replicas exemplificado a baixo, e depois reduziu-se o numero de instancias da mesma aplicação para testar a modularidade da aplicação, a infraestrutura sobre qual a aplicação foi executada documentada no apêndice 1 foram removidos três containers ilustrado na figura 10, e o ambiente teria de inicias mais três ilustrado na figura 11, para voltar a ter o nível de disponibilidade adequado a configuração .

Figura: 10 controle de replicas 1

```
root@docker:/Dockerfile/compose# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
vg8o8mf8ozud        getsatedlab_web     replicated          2/5                 9ine6ix/cloud:1.0
*:80->80/tcp
```

Figura 11: Controle de replicas 2

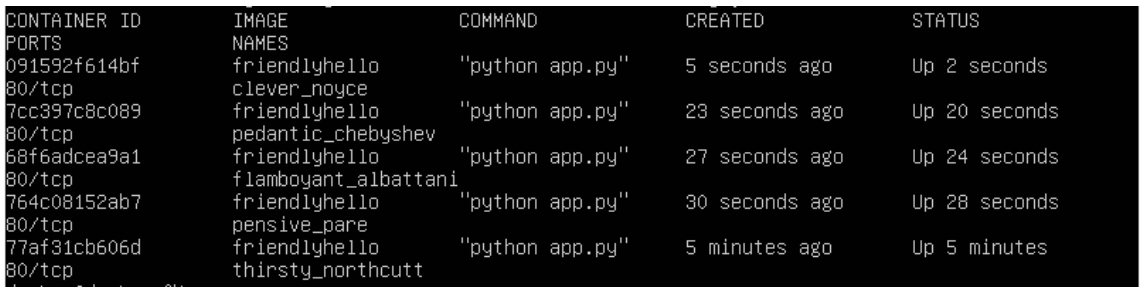
```
root@docker:/Dockerfile/compose# docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
vg8o8mf8ozud        getsatedlab_web     replicated          5/5                 9ine6ix/cloud:1.0
*:80->80/tcp
```

‘

Arquivo dockerfile da aplicação (Apêndice 2) esta contemplada na integra no apêndice, a aplicação tem como base uma imagem python oficial, expões a porta 80 para comunicação e interação via protocolo HTTP, ao iniciar o container ira buscar as bibliotecas para que a aplicação possa ser inicializada, posteriormente será executada a aplicação que é composta por dois arquivos, app.py e o requirements.txt, inicialmente a aplicação ira apenas retornar o ID do containers que esta executando aquela instancia apresentada da aplicação via web. Saída da pagina web em execução no container ilustrada na Figura 10

Figura 12: app

Figura 13: Replicas em execução



‘

CONSIDERAÇÕES FINAIS

Este Trabalho construiu um ambiente de nuvem, utilizando o Docker como ferramenta para manipular as principais funcionalidades do kernel, que são necessárias para construir os containers. Para isso o *runtime* isolou a árvore de processos dos containers, da árvore de processos do sistema host utilizando as funções do namespace, bem como o sistema de arquivos e o diretório raiz em que as imagens forram construídas, utilizando os fundamentos do chroot. Além de dimensionar os recursos de hardware para cada container em execução, função gerenciada pelo cgroups. Estes conceitos anteriormente apresentados no primeiro Capítulo são a parte central da construção desta tecnologia de isolamento. Apresentado junto as demais tecnologias de isolamento como: dualboot e os tipos de *hypervisors.* Para construção do ambiente em nuvem após os containers serem definidos, precisavam ser agrupados para trabalhar em função da disponibilidade. Para isso utilizou-se o Docker Swarm, o orquestrador nativo do Docker apresentado no Capítulo 3, ferramenta que orquestrou os containers nesta tarefa de executar e manter a aplicação ativa em seu estado ideal. Para isso o Docker Swarm iniciou o cluster. Infraestrutura na qual os nós workers foram conectados em modo brigde e tiveram suas funções delegas pelo máster, para atender aos testes realizados para analisar a capacidade do cluster em manter o estado ideal da aplicação., Para isso foram removidos alguns containers e o cluster iniciou mais containers para manter o numero de replicas dentro do padrão definido no arquivo composse da aplicação executada em laboratório. Após a análise dos dados coletados nos testes executados no cenário apresentado no capítulo 5 controlando o estado ideal de atividade da aplicação desenvolvida em python, foi possível observar a dinâmica eficiente na implementação e estabilidade dos containers, em manter e disponibilizar um ambiente para execução de serviços em nuvem, atendendo as delimitações impostas nos testes realizados. Distribuindo a demanda entre os nós *workers*. Bem como gerenciando o nível ideal de execução para a aplicação escolhida como esperado, reiniciando os containers quando necessário seguindo a condição definida no arquivo composse de que somente em caso de falha o container deve ser reiniciado,

‘

como o modelo de armazenamento é compartilhado o novo container segue as atividades de onde o antigo parou.

Outra questão importante diz respeito à os containers, é que esta ferramenta pode ser utilizada em todos os processos da aplicação, desde o desenvolvimento até a implementação prática.

‘

REFERÊNCIAS BIBLIOGRÁFICA

AMAZON WEB SERVICE, **O que é computação em nuvem,** 2019(a) . Disponível em: https://aws.amazon.com/pt/what-is-cloud-computing/. Acessado em: 06 abr. 2019

AMAZON WEB SERVICE**O que é um container,** 2019(b). Disponível em: https://aws.amazon.com/pt/containers/ Acessado em :06 abr.2019

ARBEZZANO, Gianluca **Orbiter** 12 de abr.2018 [S.I] Disponível em: https://github.com/gianarb/orbiter Acesso em : 06 abr. 2019

DOCKER **O que é um container ?,** In.: Docker ,2019(a).Disponível em: https://www.docker.com/resources/what-container Acessado em : 06 abr. 2019

DOCKER **Redes** In.: Docker, 2019(b) [S.I] Disponivel em: https://docs.docker.com/network/
Acessado em : 3 maio 2019

DOCKER **Swarm** In.: Docker, 2019(c) [S.I] Disponível em :https://docs.docker.com/engine/swarm/Acesso em : 06 abr.2019

DOCKER **Visão geral do docker** In.: Docker, 2019(d) [S.I] Disponível em: https://docs.docker.com/engine/docker-overview/Acesso em : 06 abr.2019

KUBERNETES **Gerenciamento de cluster** [S.I] Disponível em:https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/ Acesso em: 06 abr. 2019

LINUX MAN PROJECTS **Cgroups** 2019(a) [S.I]Disponível em:http://man7.org/linux/man-pages/man7/cgroups.7.htmlAcesso em : 06 abr. 2019

‘

KERRISK, Michael. **Chroot,** The Linux Programming Interface, 2019 Disponível em: http://man7.org/linux/man-pages/man2/chroot.2.htmlAcesso em: 09 abr. 2019

LEZCANO,Daniel; BRAUNER, Christian; HALLYN,Serge; GRABER, Stéphane. **Lxc,** 2019c In.: KERRISK, Michael. The Linux Programming Interface. Disponível em: http://man7.org/linux/man-pages/man7/lxc.7.html Acesso em : 06 abr. 2019

KERRISK, Michael (2013); BIEDERMAN, Eric W. **Namespace** Linux Programmer's Manual 2019. Disponível em : http://man7.org/linux/man-pages/man7/namespaces.7.html Acessado em: 06 abr.2019

MESIAS david **Docker anuncia orquestração para aplicativos distribuídos em vários contêineres** Disponivel em : https://blog.docker.com/2014/12/docker-announces-orchestration-for-multi-container-distributed-apps/ Acessado em: 25 arb. 2019 Publicado em : 4dez. 2014

MESOS **Mesos arquitetura** [S.l] Disponível em:http://mesos.apache.org/documentation/latest/architecture/ Acesso em : 06 abr.2019

MESSINA, David **Docker anuncia orquestração para aplicativos distribuídos em vários containers** 4 dez. 2014 [S.l] Disponível em :https://blog.docker.com/2014/12/docker-announces-orchestration-for-multi-container-distributed-apps/ Acesso em : 06 abr. 2019

MICROSOFT AZURE **O que é computação em nuvem ?,**In.: Microsoft, 2019. Disponível em:https://azure.microsoft.com/pt-br/overview/what-is-cloud-computing/Acesso em : 06 abr.2019

‘

PEDROSA, Paulo H.C; NOGUEIRA, Tiago **Computação em Nuvem** 2011 Instituto de computação, Universidade estadual de campinas (Unicamp) Disponível em : http://www.ic.unicamp.br/~ducatte/mo401/1s2011/T2/Artigos/G04-095352-120531-t2.pdf Acesso em : 06 abr.2019

PETAZZONI, Jerome **Container's Anatomy** [S.I] Disponivel:http://rossano.pro.br/fatec/cursos/soiiads/Anatomy-of-a-container.pdf Acesso em: 04 fev.2019

REDHAT **O que é um containers Linux** 2017 Disponível em: https://www.redhat.com/pt-br/topics/containers/whats-a-linux-containerAcessado em : 06 abr.2019

REDHAT **Rkt vs outros projetos** [S.I]Disponível em :https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html Acesso em: 04 mar. 2019

SILVA F. H. R **Avaliação de Desempenho de Contêineres Docker para Aplicações do Supremo Tribunal Federal** 2017 [S.I] Disponível em : http://bdm.unb.br/bitstream/10483/17796/1/2017_FlavioHenriqueSilva_tcc.pdf Acesso em : 30 abr. 2019

TANENBAUM, Andrew Sthuart; BOS, Herrert. **Sistemas operacionais modernos** 4 ed. Local: São Paulo Pearson, 2015.

‘

APÊNDICE 1 - infraestrutura

Dockerfile

```
FROM python:2.7-slim

WORKDIR /app

COPY . /app

RUN pip install --trusted-host pypi.python.org -r requirements.txt

EXPOSE 80

ENV NAME World

CMD ["python","app.py"]
```

docker-compose.ylm

```
version: "3"
services:
 web:
  image: 9ine6ix/cloud:1.0
  deploy:
    replicas: 5
    restart_policy:
      condition: on-failure
    resources:
     limits:
       cpus: "0.1"
       memory: 50M
  ports:
    - "80:80"
  networks:
    - webnet
```

‘

APÊNDICE 2 – aplicação

App.py

```
docker@docker:/Dockerfile$ cat app.py
from flask import Flask
from redis import Redis, RedisError
import os
import socket

redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
        try:
                visits = redis.incr("counter")
        except RedisError:
                visits= "<i> cannont connect to Redis, counter disable</i>"

        html = "<h3>Hello {name}!</h3>" \
                "<b>Hostname </b> {hostname}<br/>" \
                "<b>Visits:</b> {visits}"
        return html.format(name=os.getenv("NAME","world"), hostname=socket.gethostname(), visits=vis
its)
if __name__=="__main__":
        app.run(host='0.0.0.0', port=80)
```

‘