Orquestração de Containers Usando Kubernetes e Docker Swarm

(Versão Definitiva Após Defesa Pública)

João Emanuel Leitão Freire

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Mário Marques Freire

Covilhã, Janeiro de 2021

Dissertação elaborada no Instituto de Telecomunicações - Delegação da Covilhã e no Departamento de Informática da Universidade da Beira Interior e submetida à Universidade da Beira Interior para discussão em provas públicas.

Este trabalho foi financiado pela FCT/MCTES através de fundos nacionais e quando aplicável cofinanciado por fundos comunitários no âmbito do projeto UIDB/ 50008/2020 e foi suportado pela operação CENTRO-01-0145-FEDER-000019 - C4 - Centro de Competências em Cloud Computing, cofinanciada pelo Fundo Europeu de Desenvolvimento Regional (FEDER) através do Programa Operacional Regional do Centro (Centro 2020), no âmbito do Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de IC&DT.

Cofinanciado por:

Dedicatória

Em primeiro lugar dedico esta dissertação aos meu pais por todo o apoio dado ao longo da minha vida e também de todo o meu percurso académico. Em segundo lugar aos meus amigos que sempre me apoiaram nos momentos difíceis. Por último, aos professores do Departamento que me facultaram bastante conhecimento teórico que não possuía de experiências anteriores e, em especial, ao meu professor e orientador Mário Marques Freire por todo o incentivo para a realização desta dissertação.

Agradecimentos

Gostaria de agradecer aos meus pais por tudo o que fizeram e fazem por mim, à Universidade da Beira Interior por me acolher e mostrar me tudo o que a Covilhã pode oferecer de bom os seus alunos e, por fim, ao meu professor e orientador Mário Marques Freire pelos seus excelentes conselhos que foram essenciais para todo o processo de elaboração desta dissertação.

Resumo

Num mundo cada vez mais virtual onde novas tecnologias são criadas para vários propósitos, por vezes é difícil escolher qual é a melhor para uma certa temática. Para resolver esta escolha, existem uma infinidade de programas, aplicações, ou ferramentas diferentes disponíveis. Contudo, como é que um cliente escolhe o software adequado para a sua situação? É preciso realizar um estudo de mercado de modo a verificar qual a melhor ferramenta para o seu caso específico. Nesta dissertação, pretende-se responder a esta questão no que toca a ferramentas de orquestração no *Docker*. O *Docker* é uma tecnologia de virtualização que isola os processos em vez de ser necessário um sistema operativo inteiro, isola os processos em vez de ser necessário um sistema operativo inteiro, o que, por sua vez, o torna mais modular e fácil de trabalhar. Se o cliente quiser utilizar este tipo de tecnologia num ambiente distribuído, é útil saber quais os pontos fortes e fracos de cada ferramenta, pelo que foram comparadas as duas ferramentas de orquestração mais populares do *Docker*, o *Docker Swarm* e o *Kubernetes*. Estas duas ferramentas foram comparadas através de um estudo de literatura onde são apresentados dados qualitativos acerca desta temática bem como a realização de vários testes quantitativos com objetivo de medir os tempos de implementação, escalonamento e tolerância de falhas de containers. Com estes testes, o objetivo é facilitar a escolha dos clientes ou utilizadores de IT acerca da melhor solução para o seu problema num ambiente *Docker*.

Palavras-chave

Docker, Container, Virtualização, *Kubernetes*, *Docker Swarm*

Abstract

In an increasingly virtual world where new technologies are created for various purposes, it is sometimes difficult to choose which is best for a certain issue. To solve this choice, there is a multitude of different programs, applications, or tools available. However, how a customer choose the best software for his/her situation? It is necessary to carry out a market study to verify which is the best tool for your specific case. In this dissertation, we intend to answer this question regarding orchestration tools in Docker. Docker isolates processes instead of requiring an entire operating system, which in turn makes it more modular and easier to work with. If the customer wants to use this type of technology in a distributed environment, it is useful to know the strengths and weaknesses of each tool. Therefore, the two most popular orchestration tools for Docker were compared, Docker Swarm and Kubernetes. These two tools were compared through a literature study where qualitative data about this theme are presented, as well as the performance of several quantitative tests to measure the implementation times, scheduling, and fault tolerance of containers. With these tests, the goal is to make it easier for customers or IT users to choose the best solution for their problem in a Docker environment.

Keywords

Docker, Container, *Virtualization*, *Kubernetes*, *Docker Swarm*

Conteúdos

Lista de Figuras

Lista de Tabelas

Acrónimos

ABI Application Binary Interface

API Application Programming Interface

AWS Amazon Web Services

BPS Basis Points

CE Community Edition

CIDR Classless Inter-Domain Routing

CLI Command Line Interface

CNCF Cloud Native Computing Foundation

CNI Container Network Interface

CPU Central Processing Unit

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

EKS Elastic Kubernetes Service

ERP Enterprise Resource Planning

GPG GNU Privacy Guard

GUI Graphical User Interface

HAL Hardware Abstraction Layer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IBM International Business Machines Corporation

ICMP Internet Control Message Protocol

ID Identity Document

IMAP Internet Message Access Protocol

IOPS Input/Output Operations Per Second

ISA Instruction Set Architecture

IT Information Technology

JSON JavaScript Object Notation

LTS Long Term Support

OS Operating System

PC Personal Computer

POP3 Post Office Protocol 3

SMTP Simple Mail Transfer Protocol

TCP Transmission Control Protocol

UFS Universal Flash Storage

USB Universal Serial Bus

VM Virtual Machine

VMM Virtual Machine Monitor

YAML Yet Another Markup Language

Capítulo 1

Introducão

1.1 Âmbito e Foco da Dissertação

Nos últimos anos tem-se assistido a um forte crescimento das tecnologias de virtualização baseada em containers, sendo o Docker a plataforma que maior atenção tem atraído no mercado atual [FLE19]. Atualmente, muitas empresas pretendem uma solução baseada em containers em detrimento de uma solução baseada em virtualização ao nível de abstração do hardware. O Docker [DOC20B] é usado para implementar aplicações, utilizando containers. Estes processos compartilham o mesmo sistema operativo do host, mas são isolados uns dos outros. O Docker permite uma distribuição de containers muito vasta de várias aplicações independentes ou até mesmo aplicações que colaborem entre si. Para administrar estes containers e realizar as tarefas de implementação, o Docker Swarm [Swm] e o Kubernetes [kub] são ferramentas adequadas para este propósito. De acordo com [XSY18], os recursos da virtualização associada aos containers são partilhados com o host. Assim, é quase um pré-requisito implementar diversos containers com objetivo de alojar aplicações. Por consequência, é fundamental adotar a orquestração de containers e eliminar os hábitos de trabalho manual. A adesão a estas plataformas, como o Kubernetes e o Docker Swarm, é importante para um bom sucesso, melhor segurança e automação dos ambientes das empresas. Apesar de o Kubernetes e Docker cooperarem entre si, o Kubernetes e o Docker Swarm são competidores, pelo que nesta dissertação pretende-se comparar o Kubernetes com o Docker Swarm.

1.2 Definição do Problema e Motivação

Os *containers* são rápidos no sentido de administrar um serviço com facilidade, suavidade e precisão. Não é necessário um sistema operativo para alojar um serviço e isso pode ser uma grande vantagem para os clientes ou utilizadores de IT. Com todas as necessidades de alojar serviços ou aplicações, a escolha da ferramenta de orquestração de *containers* pode não ser óbvia, nomeadamente a escolha entre *Kubernetes* e *Docker Swarm*. O problema abordado nesta dissertação consiste em investigar e mostrar quais os aspetos em que uma ferramenta de orquestração de *containers* é melhor do que a outra e vice-versa, de modo a permitir escolher qual a melhor plataforma de acordo com as necessidades de cada empresa.

1.3 Objetivos da Investigação

O problema abordado nesta dissertação consiste em avaliar e comparar o desempenho de duas plataformas de orquestração de containers, o *Docker Swarm* e o *Kubernetes*. O principal objetivo da investigação consiste em implementar, configurar, testar, avaliar e comparar o desempenho destas duas ferramentas mas também criar soluções à volta das ferramentas para facilitar o trabalho dos utilizadores de IT. Softwares de *benchmarking* e GUI foram implementados e configurados para analisar todo o ambiente em cada ferramenta de orquestração de containers. Existe documentação sobre todo o ambiente que estas duas ferramentas proporcionam. A contribuição técnica desta dissertação aumentará a compreensão de como cada ferramenta se comporta em termos de desempenho nos pontos de medição que foram desenvolvidos e testados, o que por sua vez facilitará a escolha das ferramentas para administradores de IT no futuro.

1.4 Abordagem Adotada para Resolver o Problema

A estratégia para a resolução do problema consiste numa abordagem experimental envolvendo a implementação de um ambiente com as duas plataformas de orquestração de containers, o *Docker Swarm* e o *Kubernetes*, sendo realizadas várias experiências, com o auxílio de um *webserver* para testar os pontos de medição escolhidos. A implementação prática foi toda realizada sobre máquinas virtuais devido à doença COVID-19. É de destacar que embora sejam tipos de virtualização diferentes, verificou-se uma coexistência entre si. Após a realização dos testes, será realizada uma comparação com base nos dados recolhidos com o objetivo de identificar qual a melhor ferramenta de orquestração em função da necessidade dos clientes.

1.5 Limitações do Trabalho Realizado

As limitações desta dissertação foram causadas pela limitação no acesso ao laboratório de investigação da UBI, devido ao confinamento imposto pela pandemia de COVID-19, resultando numa limitação em termos de disponibilidade de máquinas físicas. Tal como anteriormente referido, os testes foram realizados dentro de máquinas virtuais localizadas no mesmo *host*, ou seja numa máquina física, neste caso um computador pessoal. A análise e comparação com soluções pagas, como por exemplo da AWS ou do *Microsoft Azure*, ficaram de fora do âmbito desta dissertação, pelo que o tempo dos testes foi bastante limitado devido ao facto do nível de processamento de dados numa máquina virtual não ser comparável a uma máquina física, no ambiente considerado.

1.6 Organização da Dissertação

A dissertação encontra-se organizada em cinco capítulos principais. O conteúdo dos principais capítulos desta dissertação pode ser resumido da seguinte forma. O primeiro capí-

tulo é dedicado à introdução e descreve o foco principal da dissertação, o problema e os objetivos da investigação, a abordagem utilizada para resolver este problema, as limitações e entraves que apareceram relativamente à realização dos testes e a organização da dissertação.

No segundo capítulo são abordados os temas teóricos mais importantes para a realização desta dissertação. Começa com uma breve descrição da história da virtualização e dos seus níveis, apresentando-se a seguir uma comparação entre o uso das máquinas virtuais tradicionais e os containers, uma comparação teórica entre as ferramentas de orquestração, as principais ferramentas de virtualização, cuja abordagem é importante para identificar qual a plataforma a utilizar para implementar as máquinas virtuais que por sua vez implementam as ferramentas de orquestração. Por último é apresentada uma breve exposição acerca dos mecanismos secundários utilizados.

No terceiro capítulo são descritos os ambientes experimentais. É apresentado o *test bed* experimental, o qual inclui a sua caracterização, especificações do hardware e software. Todos os procedimentos para a instalação e a configuração dos *clusters* das ferramentas de orquestração, das ferramentas de *benchmarking*, dos GUI e do *webserver* NGINX são descritas neste capítulo.

O quarto capítulo é dedicado à implementação dos testes nos *clusters* das ferramentas de orquestração. Estes testes são realizados para medir os tempos de implementação, escalonamento e tolerância a faltas dos containers, os quais, por sua vez, alojam um *webserver*.

No quinto capítulo são apresentadas as principais conclusões acerca da temática da dissertação com base em todas as informações recolhidas nos capítulos anteriores com o objetivo de definir qual é a melhor ferramenta de orquestração para cada caso. São também apresentadas sugestões sobre trabalhos futuros.

Capítulo 2

Background e Estado da Arte

2.1 Introdução

Antes de comparar as ferramentas de orquestração de containers através dos pontos de medição definidos, é fundamental ter o conhecimento teórico através de documentação confiável destas ferramentas e também da temática da virtualização. Esta dissertação baseou –se neste tema tanto na utilização de virtualização ao nível de abstração do *hardware* como ao nível do sistema operativo. Apesar de não ser a abordagem mais correta, foi necessário misturar máquinas virtuais (nível de abstração do *hardware*) com *containers* (nível do sistema operativo) devido às limitações anteriormente mencionadas. Este capítulo descreve uma série de conteúdos essenciais para a compreensão da utilização da virtualização e ainda uma breve comparação teórica das duas ferramentas.

2.2 Breve História da Virtualização

O conceito de máquina virtual não é recente. Os primeiros passos na construção de ambientes de máquinas virtuais começaram na década de 1960, quando a IBM desenvolveu o sistema operacional experimental M44/44X. A partir dele, a IBM desenvolveu vários sistemas comerciais suportando virtualização, entre os quais o *IBM System/360*, que tinha capacidade limitada de virtualização e foi arquitetada pelo lendário *Gene Amdahl* [DM06]. Mas o sistema mais famoso foi o OS/370 criado por *Robert P. Goldberg 1973* e melhorado até 1979. A tendência dominante nos sistemas naquela época era fornecer a cada utilizador um ambiente único, com seu próprio sistema operativo e aplicações, completamente independente e desvinculado dos ambientes dos outros utilizadores. Na década de 1980, com a popularização de plataformas de *hardware* baratas como, por exemplo, o computador, a virtualização perdeu importância. Afinal, era mais barato, simples e versátil fornecer um computador completo a cada utilizador do que investir em sistemas de grande porte e caros. Além disso, o *hardware* do PC tinha desempenho modesto e não fornecia o suporte adequado à virtualização, o que inibiu o uso de ambientes virtuais nessas plataformas. Com o aumento de desempenho e funcionalidades do *hardware* do computador e o surgimento da linguagem *Java*, no início dos anos 90, o interesse pelas tecnologias de virtualização voltou a ser grande [SC06]. Apesar da plataforma *PC Intel* ainda não oferecer um suporte adequado à virtualização, soluções engenhosas como as adotadas pela empresa *Vmware* permitiram a virtualização nessa plataforma, embora com desempenho relativamente modesto. Atualmente, as soluções de virtualização de linguagens e de plataformas vêm despertando grande interesse do mercado. Várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes

trazem um suporte nativo à virtualização [VMw].

2.3 Conceito de Virtualização

A virtualização é extremamente importante para o mundo cada vez mais ”digital” de hoje. Podemos definir o conceito como soluções computacionais que permitem a execução de vários sistemas operativos e seus respetivos *softwares* a partir de uma única máquina, seja ela um computador convencional ou um servidor. O conceito reflete em existir um ou mais computadores distintos dentro de um só. A diferença é que estas máquinas são virtuais. Na prática, elas oferecem resultados como qualquer outro computador, mas existem apenas logicamente, não fisicamente. Cada máquina virtual se traduz num ambiente computacional completo: praticamente todos os recursos do sistema operativo podem ser utilizados, é possível a ligação à rede local, instalação de aplicações, entre outras funcionalidades. Uma das razões para o surgimento da virtualização é que, no passado, na época em que os mainframes dominavam o cenário tecnológico e não havia computadores pessoais, por exemplo, não existia a possibilidade de adquirir, instalar e usar um *software*. Este era acompanhado de bibliotecas e outros recursos que o tornavam quase que exclusivos ao computador para o qual foi desenvolvimento originalmente. Desta forma, muitas vezes quando uma empresa implementava um novo *software*, teria de adquirir um equipamento apenas para executá-lo, em vez de simplesmente aproveitar o equipamento existente, deixando todo processo de implementação cada vez mais caro no final. A virtualização conseguiu resolver este problema. As empresas aproveitam um computador já existente para executar duas ou mais aplicações distintas, cada uma dentro de sua própria máquina virtual. Assim evita-se, gastos com novos equipamentos e aproveita-se os possíveis recursos do computador. Nos dias de hoje, a virtualização permite, por exemplo, que uma empresa execute vários serviços a partir de um único servidor ou até mesmo que um utilizador teste um sistema operativo no seu computador antes de efetivamente instalá-lo na máquina onde o deve fazer posteriormente [Car]. Do ponto de vista corporativo, o uso da virtualização destina-se a várias aplicações, como sistemas de ERP, serviços de *Cloud Computing*, ferramentas de simulação, entre muitos outros. É de salientar uma ferramenta bastante importante nesta temática da virtualização, o VMM. O monitor de máquinas virtuais ou *hipervisor* é uma aplicação que implementa uma camada de virtualização. É o monitor de máquinas virtuais que cria e gere os ambientes de máquinas virtuais, interpretando e simulando o conjunto de instruções entre as máquinas virtuais e a máquina real (*hardware*) [KES20]. As principais funções do monitor de máquinas virtuais são[KH11]:

1. Definir o ambiente de máquinas virtuais.

2. Alterar o modo de execução do sistema operativo convidado *(Guest OS)* de privilegiado para não privilegiado, e vice-versa.

3. Simular as instruções e escalonar o uso da CPU para as máquinas virtuais.

4. Gerir o acesso aos blocos de memória e disco destinados ao funcionamento das máquinas virtuais.

5. Intermediar as chamadas de sistema e controlar acesso a outros dispositivos como CD-ROM, drives de disquete, dispositivos de rede, dispositivos USB.

Um monitor de máquinas virtuais deve seguir três características principais [KH11]:

1. **Eficiência:** é extremamente importante que um grande número de instruções do processador virtual seja executada diretamente pelo processador real, sem que haja intervenção do monitor. As instruções que não puderem ser tratadas pelo processador real precisam ser tratadas pelo monitor.

2. **Integridade:** todas as requisições aos recursos de *hardware* devem ser alocadas explicitamente pelo monitor.

3. **Equivalência:** o monitor deve prover um comportamento de execução semelhante ao da máquina real para o qual ele oferece suporte de virtualização.

É de salientar que na implementação de um monitor de máquinas virtuais deve-se levar em conta características como: compatibilidade, desempenho e simplicidade. A compatibilidade é importante para a execução do *software*. O desempenho é de extrema importância para a execução do sistema operativo e aplicações na máquina virtual. Na temática de máquinas virtuais, existem 3 tipos [KH11]:

1. **VM nativa (native VM ou bare metal VM)**: instalada através de um VMM designado por hipervisor em modo privilegiado.

2. **VM alojada (hosted VM)**: neste caso, o VMM é executado em modo não privilegiado. O sistema operativo hospedeiro (host OS) não precisa ser modificado.

3. **VM em modo duplo**: parte do VMM é executado no nível do utilizador e outra parte é executado no nível supervisor. Neste caso, o sistema operativo hospedeiro pode ter que ser modificado.

A função principal da camada de *software* para virtualização é virtualizar o *hardware* físico de um host em recursos virtuais para serem usados exclusivamente por VMs. O *software* de virtualização cria a abstração de VMs por interposição de uma camada de virtualização em vários níveis de um sistema de computador. Estas camadas de virtualização comuns incluem os níveis [KH11]:

1. Nível de arquitetura do conjunto de instruções (ISA).

2. Nível de abstração do hardware (HAL).

3. Nível do sistema operativo.

4. Nível de suporte biblioteca.

5. Nível de aplicação.

2.3.1 Nível de arquitetura do conjunto de instruções

No nível ISA, a virtualização é executada através da simulação de uma determinada ISA pela ISA do *host*. Com esta abordagem, é possível executar uma grande quantidade de código binário legado escrito para vários processadores no *hardware* de um novo (*host*). A simulação do conjunto de instruções conduz a ISA virtuais criadas no *hardware* de qualquer máquina. O método de emulação básico é através de interpretação de código. Um programa interpreta as instruções fonte para as instruções alvo, uma a uma. Uma instrução fonte pode necessitar de dezenas ou centenas de instruções alvo nativas para executar a sua função sendo este um processo relativamente lento. Para alcançar um melhor desempenho, é desejável ter tradução binária dinâmica *(dynamic binary translation)*. Esta abordagem traduz blocos básicos de instruções fonte dinâmicas para instruções alvo [SN05a].

2.3.2 Nível de abstração do hardware

A virtualização ao nível do *hardware* explora a semelhança nas arquiteturas das plataformas *guest e host* para reduzir a latência da interpretação. A maioria dos simuladores de PC comerciais do mundo de hoje usa esta camada de virtualização em plataformas x86 populares para torná-lo eficiente o seu uso, viável e prático. Esta camada da virtualização ajuda a mapear os recursos virtuais para recursos físicos e usar o *hardware* nativo para cálculos na máquina virtual. Para que essa tecnologia de virtualização funcione corretamente, a VM deve ser capaz de intercetar todas as instruções de execução privilegiadas e transmita-as ao VMM subjacente para serem tratadas. Isso ocorre porque, dentro de um ambiente de VMM, várias VMs podem ter um sistema operativo em execução que deseja emitir instruções privilegiadas e que chame a atenção do CPU. Quando um problema ocorre durante uma instrução de execução privilegiada, em vez de gerar uma exceção ou falha, a instrução é enviada para o VMM. Isto permite que o VMM consiga o controlo completo da máquina e mantenha cada VM isolada. O VMM, em seguida, executa a instrução no processador ou simula os resultados e é devolvida à VM. Ou seja, resumidamente o objetivo desta camada de virtualização é gerar um ambiente de *hardware* virtual para uma máquina virtual [SN05b].

2.3.3 Nível do sistema operativo

A virtualização ao nível do sistema operativo refere-se a um recurso do sistema operativo no qual o *kernel* permite a existência de várias instâncias isoladas do espaço do utilizador. Estas instâncias designam –se de containers, cujo são nada mais nada menos que um pacote de *software* que contem o código, as configurações e as dependências de uma aplicação, o que proporciona eficiência operacional e produtiva. Assim, o utilizador de sistemas pode saber exatamente como será executado, o que significa que é previsível, repetível e imutável. O aumento de containers tem sido um grande facilitador para o *DevOps* como serviço e pode superar o maior obstáculo de segurança enfrentado atualmente. Este tipo de virtualização é normalmente usado para ambientes de alojamento de máquinas

virtuais, onde é útil para alojar aplicações em diferentes containers para aumentar a segurança e a independência do *hardware*. A virtualização ao nível do sistema operativo não implica grandes sobrecargas no *host* porque os *softwares* que constituem o *container* utilizam todos os recursos do *host*, ou seja não são sujeitos a uma emulação ou na execução de máquinas virtuais intermediárias. Por outro lado, este tipo de virtualização não é tão flexível em relação às outras abordagens porque não é possível hospedar um sistema operativo inteiro ou um *kernel* diferente [SN05c].

2.3.3.1 Contentorização

A contentorização torna as aplicações portáteis virtualizando o seu nível do sistema operacional, criando sistemas encapsulados isolados que são baseados em *kernel*. As aplicações em *containers* podem ser descartados em qualquer lugar e executados sem dependências ou exigindo uma VM inteira, eliminando dependências. Mas quando existem vários *containers* a orquestração de *containers* é necessária. A orquestração de containers é o processo que geralmente implementa vários *containers* para alojar uma aplicação por meio da automação chamada orquestração de *container*. Plataformas como *Kubernetes* e *Docker Swarm* são o mecanismo de gestão e orquestração de *containers* que permitem aos utilizadores de sistemas orientar a implementação de *containers* e automatizar atualizações, monitorização de integridade e procedimentos de failover. A grande vantagem da contentorização é o facto de não ser necessário a criar de uma máquina virtual inteira, com todos os seus recursos associados, para cada aplicação [con].

2.3.3.2 Docker

O *docker* é uma ferramenta *open-source* criada para facilitar a criação, implementação e execução de aplicações a partir de containers. Os containers permitem ao utilizador implementar uma aplicação com todos os requisitos necessários, tais como bibliotecas ou dependências da aplicação. Assim, é possível instalar a aplicação como um pacote de software. Ao fazer isso, o utilizador tem a certeza de que a aplicação será executada em qualquer outra máquina *(Linux)*, independemente das configurações personalizadas que a máquina possa ter. De certa forma, o docker tem uma função semelhante a uma máquina virtual. Contudo, em vez de criar um sistema operativo virtual inteiro, o docker permite que as aplicações usem o mesmo *kernel (Linux)* que o sistema onde estão implementados. Isto proporciona um aumento significativo no desempenho e reduz o tamanho da aplicação [JAS20].

É importante ter conhecimento de tudo o que involve o funcionamento do *docker*. Existem 4 elementos essenciais incluindo o *Docker Client* e *Docker Server/Daemon*, *Docker Images, Docker Registries e os Docker Containers*. Na figura 2.1 é apresentada o designado *Docker Engine* onde são apresentados todos estes elementos [DOC20B]. É de salientar que as ferramentas de orquestração de containers presentes neste documento correm sobre o *docker*, ou seja nenhuma das duas funciona sem o *docker*.

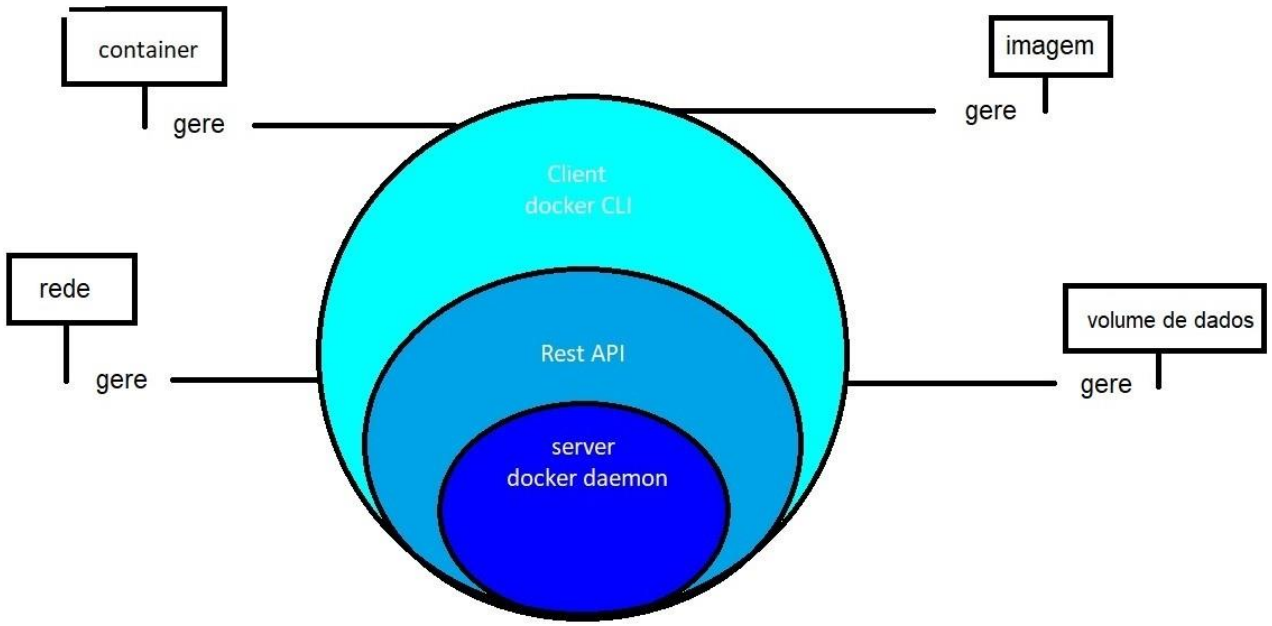

Figura 2.1: *Docker Engine* (figura redesenhada a partir de [DOC20B]).

O *Docker* utiliza uma arquitetura de cliente-servidor. O *Docker Client* comunica com o *Docker Server/Daemon*, que realiza o trabalho "pesado" de construir, executar e distribuir os containers do docker. O *Client e o Daemon do Docker* podem ser executados no mesmo sistema ou podem conectados entre si remotamente. O *client e daemon do Docker* se comunicam por meio de uma *API REST* [DOC20B].
De seguida serão apresentadas todas as funções de cada um dos componentes [DOC20B]:

1. **Docker Server/Daemon**: Espera e escuta as solicitações da API do *Docker* e gere os objetos do Docker, como imagens, *containers*, redes e volumes. Um *daemon* também pode se comunicar com outros *daemons* para gerir estes mesmos objetos.

2. **Docker Client**: É a principal maneira pela qual muitos utilizadores interagem com o *Docker*. Quando o utilizador usa comandos como por exemplo o "docker run", o cliente envia esses comandos para *Docker Daemon* os quais executa. O comando "docker" usa a API do *Docker*. O *Docker Client* pode comunicar com mais de um *daemon*.

3. **Docker Image**: é um dos objetos subjacentes mais importantes que contém instruções necessárias para implementar um *container* do *Docker*. Uma imagem é um modelo apenas de leitura usado para iniciar containers. Cada imagem começa com uma base e outras partes são adicionadas em cima dessa base para executar a tarefa para a qual foi designada. Um exemplo disso pode ser uma imagem que usa o *Ubuntu* como base e uma aplicação *Web* sobre ele com o *Node.js como middleware*. Cada imagem consiste em várias camadas e o *Docker* usa os sistemas de ficheiros *Union* (UFS) para combinar essas camadas num só. O UFS é um serviço de sistema de a ficheiros para sistemas operativos *Linux, FreeBSD e NetBSD* que implementa

uma montagem de modo a unir outros sistemas de ficheiros. Isso significa que ficheiros e registros em diferentes sistemas de armazenamento podem ser reunidos e criar um sistema do mesmo género comum. Por esse motivo, diferentes partes de uma imagem podem ser substituídas ou atualizadas sem que o administrador precise reconstruir uma imagem inteira sem alterar as camadas.

4. **Docker Registries**: É responsável pelo armazenamento das *Docker Images*. *O Docker Hub* é um registro público que qualquer pessoa pode usar, e o Docker está configurado para procurar imagens no *Docker Hub* por predefinição. O utilizador pode até executar seu próprio registo privado. Quando o utilizador usa os comandos "docker pull" ou "docker run", as imagens necessárias são extraídas do registo configurado. Quando se usa o comando "docker push", a imagem do utilizador é enviada para o registo configurado.

5. **Docker Containers**: Um *container* é uma instância executável de uma imagem como anteriormente referido. O utilizador pode criar, iniciar, parar, mover ou excluir um *container* usando a API ou a CLI do *Docker*. É possível também conectar um *container* a uma ou mais redes, anexar armazenamento dentro de si ou até criar uma nova imagem com base no seu estado atual. Por predefinição, um *container* é relativamente bem isolado de outros *containers* e da máquina *host* onde está. O utilizador pode controlar mais objetos como a rede, o armazenamento ou outros subsistemas subjacentes de um *container* estão isolados de outros containers ou da máquina *host*. Um *container* é definido pela sua imagem, bem como pelas opções de configuração que o utilizador fornece quando o cria ou inicia. Quando um *container* é removido, quaisquer alterações no seu estado que não sejam armazenadas no designado "*persistent storage*" desaparecem.

2.3.4 Nível de suporte de biblioteca

Em quase todos os sistemas, as aplicações são programadas utilizando um conjunto de APIs (*Interface* de programação de aplicações) exportadas por um grupo de bibliotecas criadas ao nível do utilizador. Essas bibliotecas são projetadas para ocultar os detalhes relacionados com o sistema operativo, a fim de simplificá-lo para programadores normais. No entanto, isso oferece uma nova oportunidade para a comunidade de virtualização. Existem exemplos de softwares que funcionam acima da camada do sistema operativo e produzem uma configuração do ambiente virtual diferente, tão diferente que pode apresentar uma interface binária diferente. Por outras palavras, as técnicas de virtualização são usadas para implementar uma ABI (*Interface* binária de aplicação) diferente e / ou uma *interface* diferente [SN05d].

2.3.5 Nível de aplicação

Neste tipo de virtualização é possível virtualizar diretamente um software. Diferente da virtualização ao nível do hardware onde o hipervisor utiliza uma configuração completa

de *hardware*, a virtualização ao nível das aplicações exige que a aplicação possa ser virtualizado. Este tipo de virtualização normalmente não permite que outras aplicações interajam com a aplicação virtualizada. A virtualização ao nível das aplicações é usada principalmente para permitir que uma aplicação seja executada num sistema sem precisar de ser instalado. Em vez disso, uma aplicação virtual contém seu próprio ambiente virtual no qual é executado [GIO19]. Assim como a virtualização ao nível do hardware requer um hipervisor para criar e gerir máquinas virtuais, a virtualização ao nível das aplicações requer um gestor de aplicações como por exemplo o *Microsoft App-V* [app] ou o *Citrix ZenApp* [Xen].

2.4 Máquinas Virtuais Versus Containers

É essencial para os clientes que atualmente saibam escolher qual é a melhor opção para correr a sua aplicação. Como anteriormente referido, já foi apresentado os conceitos de máquinas virtuais e *containers*. É fundamental estabelecer uma tabela de comparação com vários parâmetros para saber escolher a melhor opção. Contudo, é necessário, em primeiro lugar, apresentar a arquitetura das máquinas virtuais e *containers* [ARQ19]:

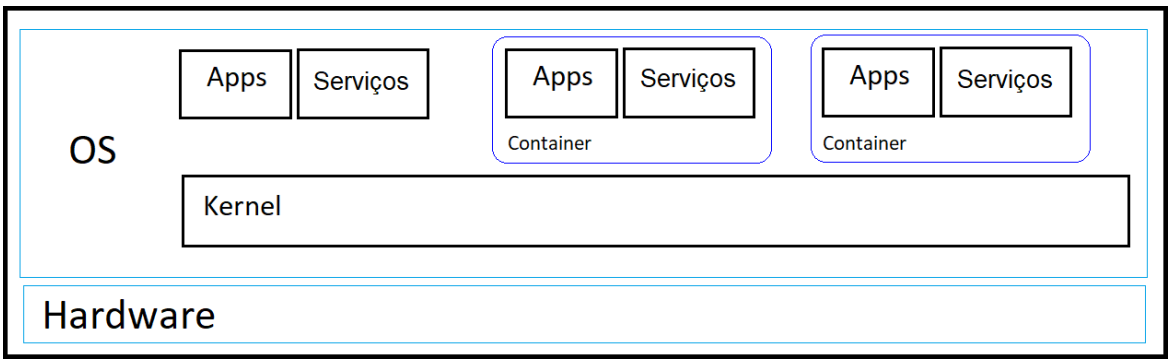

Figura 2.2: Arquitetura do *Container* (figura redesenhada a partir de [ARQ19]).

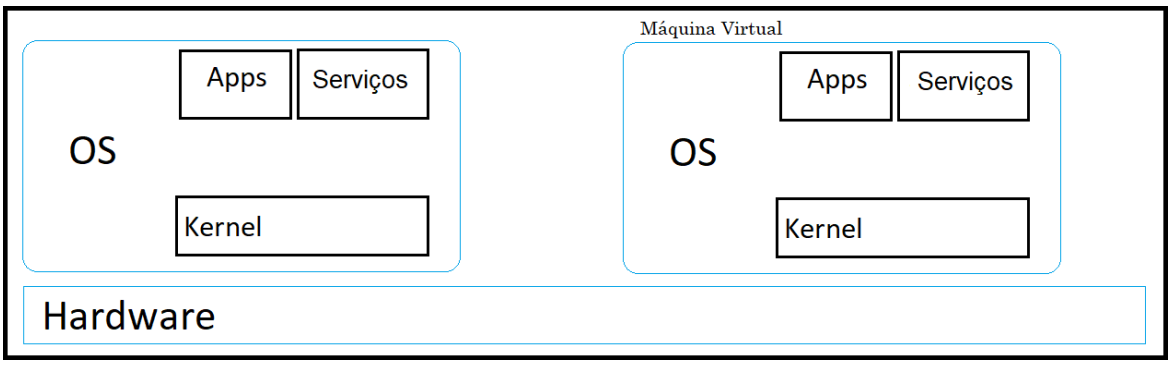

Figura 2.3: Arquitetura da *Virtual Machine* (figura redesenhada a partir de [ARQ19]).

Na figura 2.2, é possível observar que os *containers* não utilizam um sistema operativo próprio e consequentemente o *kernel* é o mesmo do sistema operativo do *host*. Por outro

lado, na figura 2.3 é possível observar que as máquinas virtuais possuem um sistema operativo próprio e um *kernel* próprio. Com as arquiteturas comparadas, é altura para dividir cada um dos componentes em parâmetros de caso de uso e estabelecer uma comparação entre eles. Estes parâmetros são [ARQ19]:

1. *Nível de virtualização.*
2. *Isolamento.*
3. *Sistema operativo.*
4. *Compatibilidade de convidado.*
5. *Implementação.*
6. *Updates e upgrades do sistema operativo.*
7. *Balanceamento de carga.*
8. *Tolerância a faltas.*
9. *Networking.*

Em termos do nível de virtualização, as máquinas virtuais utilizam virtualização ao nível do *Hardware* enquanto os *containers* utilizam virtualização ao nível do sistema operativo.
Em termos de isolamento, as máquinas virtuais fornecem um isolamento completo do sistema operativo e de outras máquinas virtuais. Isto é bastante útil quando os limites de segurança são críticos, como por exemplo hospedar aplicações de empresas rivais no mesmo servidor ou *cluster*. Os containers fornecem um isolamento mais leve entre o *host* e os outros containers mas não fornece mais segurança que as máquinas virtuais.
Em termos do sistema operativo, nas máquinas virtuais é executado um sistema operativo completo incluindo o *kernel*, o que exigi mais recursos do *host* (CPU, memória e armazenamento). Nos *containers* é executado numa pequena porção do sistema operativo do *host* cujo contem os serviços necessários para a execução da aplicação e exige menos recursos do *host*.
Em termos de compatibilidade de convidado, na máquina virtual pode ser executado qualquer sistema operativo dentro da máquina virtual. Por outro lado, num *container* é executado o mesmo sistema operativo que o *host*. Em termos de implementação, uma só máquina virtual é implementada com o uso do *Windows Admin Center* ou *Hyper-V Manager* enquanto múltiplas máquinas virtuais são implementadas com o uso de *PowerShell* ou *System Center Virtual Machine Manager*. No caso dos *containers*, um só *container* é implementado com o uso do *Docker via CLI* enquanto múltiplos *containers* são implementados por um orquestrador, como por exemplo o *Azure Kubernetes Service*.
Em termos de updates e upgrades do sistema operativo, nas máquinas virtuais é necessário fazer o *download* e instalar os *updates* do sistema operativo de cada máquina virtual. Instalar uma nova versão do sistema operativo requer um *upgrade* ou até a criação de uma nova máquina virtual. Este processo pode despender muito tempo principalmente

se houver muitas máquinas virtuais. Nos containers, o processo de *update e upgrade* dos ficheiros do sistema operativo dentro do *container* é o mesmo [ARQ19]:

1. Editar o ficheiro de imagem que construiu o *container* (conhecido por *DockerFile*) para a ultima versão da imagem base do sistema Operativo.
2. Substituir a imagem do *container* com a nova.
3. Enviar a imagem do *container* para o registo do mesmo.
4. Reimplementar o *container* utilizando orquestrador. Este fornece uma grande automação neste processo em escala.

Em termos de balanceamento de carga, no caso das máquinas virtuais, o balanceamento de carga da máquina virtual transfere as máquinas virtuais para outros servidores de um *failover cluster*. Por outro lado os *containers* não podem ser transferidos. Em vez disso, um orquestrador pode automaticamente iniciar ou parar os *containers* dentro *cluster* para assim gerir as operações de balanceamento de carga,.
Em termos de tolerância a faltas, as máquinas virtuais podem fazer o processo de *failover* para outro servidor dentro de um *cluster* enquanto no caso dos *containers*, se um *cluster* falha, qualquer *container* dentro do mesmo pode ser facilmente e rapidamente recriado pelo orquestrador noutro *cluster*.
Por fim, em termos de *Networking*, as máquinas virtuais utilizam adaptadores de rede virtual enquanto os *containers* utilizam uma porção isolada do adaptador de rede virtual, o que traduz numa menor utilização de virtualização. A *firewall* do host é compartilhada com os *containers* [ARQ19].
As máquinas virtuais e *containers* diferem de várias maneiras, mas a principal diferença é que os *containers* fornecem uma maneira de virtualizar um sistema operativo para que várias cargas de trabalho possam ser executadas num única instância do sistema operativo. Isto pouparia muito tempo e dinheiro às empresas que utilizam várias instâncias do mesmo sistema operativo visto que nas máquinas virtuais, o *hardware* é virtualizado para executar várias instâncias do sistema operativo. A velocidade, agilidade e portabilidade dos *containers* tornam – os numa ferramenta para ajudar a otimizar o desenvolvimento de *software* e ajudar os *DevOps* nas suas operações.
Apesar de todas estas diferenças, existe uma pergunta patente nesta temática: Podem coexistir máquinas virtuais e containers no mesmo ambiente?
A resposta é um sim. No nível mais básico da compreensão dos containers, é sempre motivador executar todos os serviços do docker em máquinas virtuais. Este permite ao utilizador uma maior facilidade na adesão de conhecimentos.

Caso haja algum problema na instalação ou configuração dos serviços, a máquina virtual pode ser facilmente descartada para começar novamente o processo todo. Esta tese foi realizada, como anteriormente referido, no *Virtualbox* onde todo o ambiente envolvido continham containers dentro de máquinas virtuais com o sistema operativo *Ubuntu 16.04*. Uma outra questão bastante interessante era saber se os serviços do *container* podem interagir ou não com os serviços da máquina virtual. Novamente, a resposta é um sim. Se, por exemplo, uma aplicação precisar de interagir com uma base de dados que reside noutra máquina virtual é possível que isso aconteça desde que a rede interna entre as Vms esteja instalada e configurada corretamente [MIK16].

2.5 Sistemas de Orquestração de Containers

À medida que um número crescente de aplicações são movidas para a *Cloud*, as suas arquiteturas e distribuições vão continuar a evoluir. Essa evolução requer o conjunto certo de ferramentas e habilidades para gerir efetivamente uma topologia distribuída na *Cloud*. A gestão de micros serviços em máquinas virtuais, cada um com vários containers agrupados, pode-se tornar rapidamente complicado. Para reduzir essa complexidade, a orquestração de *containers* é utilizada. O *Docker Swarm* e o *Kubernetes* são as duas principais ferramentas dos dias atuais para cumprir esta função. Portanto, é necessário fazer uma comparação entre as duas ferramentas para decidir qual é melhor de acordo com as necessidades de cada empresa.

2.5.1 Kubernetes

Kubernetes é uma plataforma *open-source* criada em julho de 2015 [MON19] com objetivo de realizar operações de desenvolvimento de *containers*, escabilidade e automação entre os *cluster*s de *host*s. Esta plataforma é modular e pode ser utilizado em qualquer arquitetura de desenvolvimento. A plataforma foi construída pela *Google* com base na sua experiência na execução de *containers* em produção usando um sistema interno de gestão de *cluster*s chamado *Borg*. A *Kubernetes* também distribui a carga entre os *containers*. O objetivo é aliviar as ferramentas e componentes do problema enfrentado devido à execução das aplicações nas *Clouds* públicas e privadas colocando os *containers* em grupos e nomeando -os como unidades lógicas. O poder desta plataforma está no fácil escalonamento, na portabilidade do ambiente e no crescimento flexível.

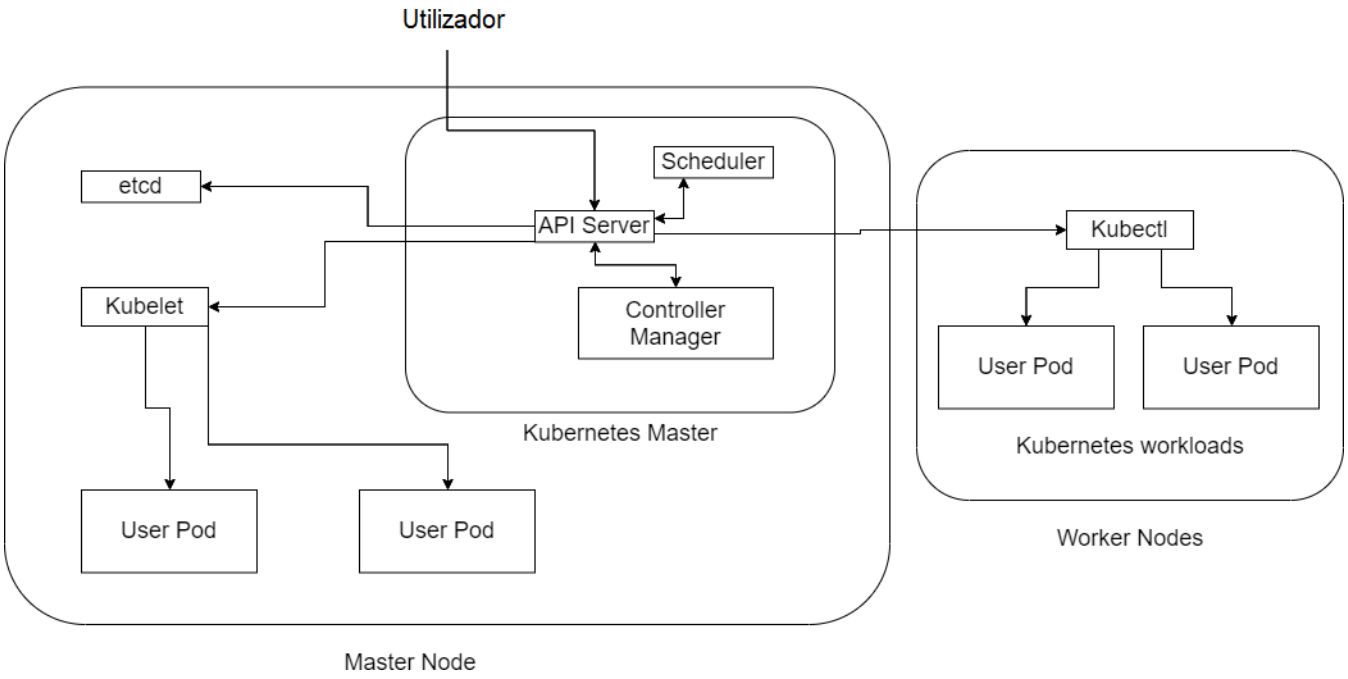

Figura 2.4: Arquitetura do *Kubernetes* (figura redesenhada a partir de [HAR18]).

Na figura 2.4 existem vários termos associados ao *Kubernetes*. Cada um destes componentes é fundamental para o seu funcionamento. Estes componentes são [SPE18]:

1. **etcd**: Armazena os dados de configuração que podem ser acedidos pelo servidor master de API do *Kubernetes* usando HTTP simples ou JSON API.

2. **API Server**: É o hub de gestão do master node do *Kubernetes*, com objetivo de facilitar a comunicação entre os vários componentes, mantendo assim a integridade do *cluster*.

3. **Controller Manager**: Garante que o estado desejado do *cluster* corresponda ao estado atual, escalonando os *workload*s para cima e para baixo.

4. **Scheduler**: Coloca a carga de trabalho no node apropriado - neste caso, todas os *workload*s serão colocados localmente dentro do *host* do utilizador.

5. **Kubelet**: Recebe as especificações de *pods*(grupo de *containers* com recursos partilhados) do API Server e gere os *pods* em execução no *host*. O *Kubernetes* apresenta diversas vantagens acerca da sua utilização entre as quais [SPE18]:

   (a) **É rápido**: quando se trata de implementar continuamente novos recursos sem tempo o *Kubernetes* é uma escolha perfeita. O objetivo do *Kubernetes* é atualizar uma aplicação com um tempo de atividade constante. A sua velocidade é medida através de vários recursos que o utilizador pode enviar por hora, mantendo um serviço disponível.

   (b) **Cumpre os princípios da infraestrutura imutável**: de maneira tradicional, se algo der errado com várias atualizações, o utilizador de sistema não

terá registo de quantas atualizações implementou e em que ponto ocorreu o erro. Em infraestruturas imutáveis, se o utilizador de sistema deseja atualizar qualquer aplicação, é necessário criar uma imagem do *container* com uma nova *tag* e implementá-la, eliminando o *container* antigo com a versão antiga da imagem. Desta forma, o utilizador de sistema terá um registo e terá uma visão do que fez. No caso, se houver algum erro ele poderá reverter facilmente para a imagem anterior.

(c) **Fornece uma configuração declarativa**: O utilizador de sistema pode saber em que estado o sistema deve estar para evitar erros. Controlo de origem, testes de unidade, etc., que são ferramentas tradicionais, não podem ser usados com configurações imperativas, mas podem ser usados com configurações declarativas.

(d) **Implementação e automação do software em escala**: o dimensionamento é fácil devido à natureza imutável e declarativa do *Kubernetes*. O *Kubernetes* oferece vários recursos úteis para fins de dimensionamento:

(e) **Escala de infraestrutura horizontal**: as operações são feitas no nível do servidor individual para aplicar o dimensionamento horizontal. Os servidores mais recentes podem ser adicionados ou desconectados sem esforço.

(f) **Escalonamento automático**: com base no uso de recursos da CPU, o utilizador de sistema pode alterar o número de *containers* em execução.

(g) **Escala manual**: o utilizador de sistema pode dimensionar manualmente o número de *containers* em execução por meio de um comando ou da *interface*.

(h) **Controlador de replicação**: o controlador de replicação garante que o *cluster* tenha um número especificado de *pods* equivalentes a uma condição de execução. Se no caso de existirem muitos *pods*, o controlador de replicação pode remover os *pods* extras ou vice-versa.

(i) **Lida com a disponibilidade da aplicação**: o *Kubernetes* verifica a integridade dos nodes e *containers*, além de fornecer auto-recuperação e substituição automática, caso o conjunto de *pods* do computador trave devido a um erro. Além disso, distribui a carga em vários *pods* para equilibrar os recursos rapidamente durante o tráfego acidental.

(j) **Volume de armazenamento**: no *Kubernetes*, os dados são compartilhados entre os *containers*, mas, se os *pods* forem mortos, o volume será removido automaticamente. Além disso, os dados são armazenados remotamente, portanto, se o *pod* for movido para outro *node*, os dados permanecerão intatos até que sejam excluídos pelo utilizador de sistema.

Em todas as plataformas existentes existem tanto vantagens como desvantagens visto que nada é perfeito. No que toca as desvantagens de seguida serão apresentadas algumas delas [SPE18]:

1. **O processo inicial leva tempo**: quando um novo processo é criado, o utilizador de sistema precisa aguardar o início da aplicação antes de estar disponível para os

utilizadores aos quais o serviço será servido. Se ocorrer uma migração para o *Kubernetes*, é necessário fazer modificações no código para tornar o processo de início mais eficiente para que os utilizadores da aplicação não tenham nenhum problema.

2. **A migração para *stateless* requer muitos esforços**: se a aplicação estiver em *cluster* ou sem estado, os *pods* extras não serão configurados e terão que refazer as configurações das aplicações do utilizador de sistema.

3. **O processo de instalação é tedioso**: é difícil configurar o *Kubernetes* no *cluster* se o utilizador de sistema não estiver a usar nenhuma plataforma baseado em *Cloud* como o *Azure, o Google ou o Amazon*.

2.5.2 Docker Swarm

O *Docker Swarm* ou simplesmente *Swarm* é uma plataforma *open-source* criada em março de 2013 [MON19] de orquestração e gestão de *containers* e é o mecanismo de *cluster* nativo para e pelo Docker. Todos os *softwares*, serviços ou ferramentas que são executados com *containers* do *Docker* são executados igualmente no *Swarm*. Além disso, o *Swarm* utiliza a mesma linha de comando do *Docker*. Quem faz a orquestração e gestão do *cluster* são os chamados *Manager Nodes*. Os *Worker Nodes* recebem e executam tarefas dos *Manager Nodes*. Um serviço, que pode ser especificado declarativamente, consiste em tarefas que podem ser executadas nos *nodes do Swarm*. Os serviços podem ser replicados para serem executados em vários *nodes*. No modelo de serviços replicados, o balanceamento de carga de ingresso e o DNS interno podem ser usados para fornecer pontos de extremidade de serviço altamente disponíveis. O *Swarm* transforma um *pool* de *hosts do Docker* num único *host* virtual. O *Swarm* é especialmente útil para pessoas que estão a tentar familiarizar-se com um ambiente orquestrado ou que precisam aderir a uma técnica de implementação simples, mas também ter mais apenas um ambiente em *Cloud* ou uma plataforma específica para executá-lo.

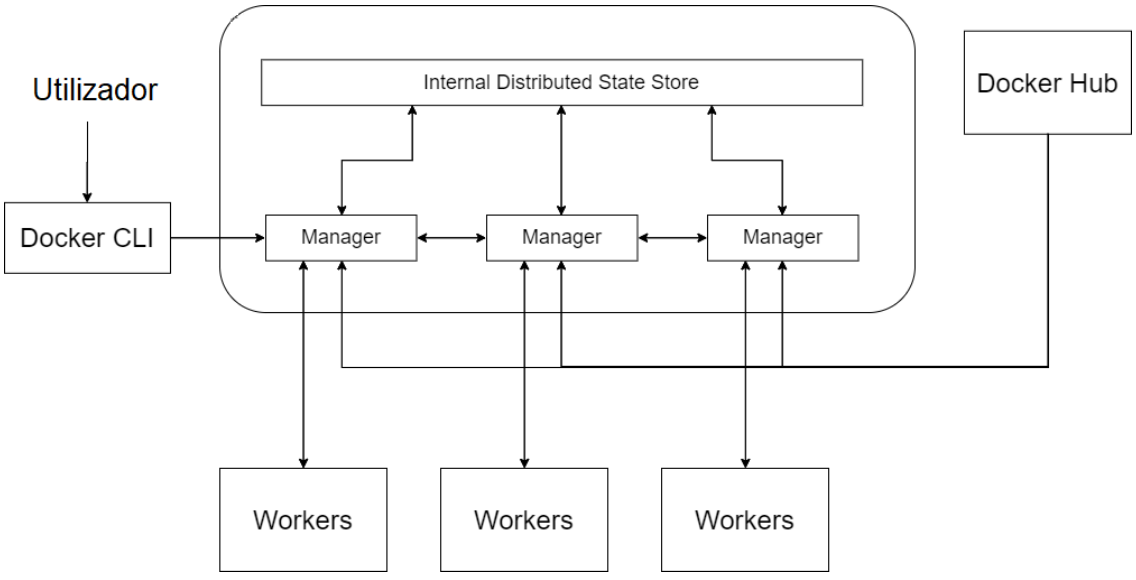

Figura 2.5: Arquitetura do *Docker Swarm* (figura redesenhada a partir de [DSA20]).

Na figura 2.5 existem vários termos associados à arquitetura do *Docker Swarm*. Cada um destes componentes é fundamental para o seu funcionamento. Estes componentes são [SPE18]:

1. **Node**: Um *node* é uma instância de um *Swarm*. Os *nodes* podem ser distribuídos localmente ou em *Clouds* públicas.

2. **Swarm**: Um *cluster* de *nodes (ou Docker Engines)*. No modo *Swarm*, o utilizador de sistema orquestra os serviços, em vez de executar comandos do *container*.

3. ***Manager Nodes***: Estes *nodes* recebem definições de serviço do utilizador e enviam o trabalho para os *Worker Nodes*. Os *Manager Nodes* também podem executar as tarefas dos *Worker Nodes*.

4. ***Worker Nodes***: Colecionam e executam as tarefas dadas pelos *Manager Nodes*.

5. **Service**: Um serviço especifica a imagem do container e o número de réplicas.

6. **Task**: Uma tarefa é uma unidade atômica de um serviço agendado dentro de um *Worker Node*.

Tal como o *Kubernetes*, o *Docker Swarm* apresenta diversas vantagens acerca da sua utilização entre as quais [SPE18]:

1. **É executado num ritmo mais rápido**: quando um utilizador de sistema usa um ambiente virtual, talvez apercebe-se que este ambiente leva muito tempo e inclui o tedioso procedimento de inicializar e inicialização da aplicação que o utilizador de sistema deseja executar. Com o *Docker Swarm*, isso não é um problema. O *Docker Swarm* elimina a necessidade de inicializar uma máquina virtual completa e permite que a aplicação seja executada num ambiente virtual e definido por software rapidamente e ajuda na implementação do *DevOps*.

2. **A documentação fornece todas as informações**: a equipa do *Docker* destaca-se quando o assunto se trata de documentação. O *Docker* está evoluindo rapidamente e recebeu grande reconhecimento por toda a plataforma. Quando uma nova versão é lançada num curto espaço de tempo, algumas plataformas não cuidam de atualizar a documentação. Contudo, neste caso, o *Docker* nunca compromete isso. Se, no caso de as informações se aplicarem apenas a determinadas versões, a documentação garante que todas as informações sejam atualizadas.

3. **Fornece uma configuração simples e rápida**: um dos principais benefícios do *Docker Swarm* é que simplifica o assunto. O *Docker Swarm* permite que os utilizadores das aplicações assumam a sua própria configuração. O *Docker Swarm* pode ser usado em vários ambientes em que os requisitos não são limitados pelo ambiente das aplicações.

4. **Garante que a aplicação é isolada**: o *Docker Swarm* cuida para que cada *container* seja isolado dos outros *containers* e tenha seus próprios recursos. Vários *containers*

podem ser implantados para executar a aplicação separado em diferentes *stacks*. Além disso, o *Docker Swarm* limpa a remoção das aplicações à medida que cada aplicação é executada dentro do seu próprio *container*. Se a aplicação não for mais necessária, o utilizador de sistema poderá excluir do seu *container*.

5. **Controlo de versão e reutilização de componente**: Com o *Docker Swarm*, o utilizador de sistema pode encontrar versões consecutivas de um *container*, examinar diferenças ou retroceder para as versões anteriores.

Relativamente às desvantagens do *Docker Swarm* [SPE18]:

1. **O Docker depende da plataforma**: o *Docker Swarm* é uma plataforma do *Linux*. Embora o *Docker* suporte o *Windows* e o *Mac OS X*, utiliza máquinas virtuais para serem executadas em uma plataforma *não-Linux*. Uma aplicação projetada para ser executada no *container do Docker no Windows* não pode ser executado no *Linux* e vice-versa.

2. **Não oferece opção de armazenamento**: o *Docker Swarm* não oferece uma maneira fácil de conectar os *containers* ao armazenamento, e essa é uma das principais desvantagens. Os seus volumes de dados exigem muita improvisação no *host* e configurações manuais. O *Docker Swarm* soluciona vários problemas de armazenamento, mas não de maneira eficiente e amigável.

3. **Má monitorização**: O *Docker Swarm* fornece as informações básicas sobre o *container* e, se cliente estiver à procura pela solução de monitorização básica, o comando *Stats* será suficiente. Se o cliente procura uma monitorização avançada, o *Docker Swarm* nunca é uma opção. Embora existam ferramentas de terceiros disponíveis como *CAdvisor, prometheus ou grafana* que oferecem mais monitorização, não é possível colecionar mais dados sobre *containers* em tempo real com o próprio *Docker* .

2.5.3 Comparação dos sistemas de orquestração de containers

Embora ambas as ferramentas tenham sido utilizadas para economizar recursos limitando o uso do hardware para atender aos requisitos de recursos de negócios, existem algumas diferenças entre elas que exigem uma análise abrangente de o cliente tomar a sua decisão acerca do melhor caminho a tomar. Estas diferenças dividem-se em [PLA17]:

1. Definição de aplicações.
2. Rede.
3. Escalabilidade.
4. Elevada disponibilidade.
5. *Container Setup*.
6. Balanceamento de carga.

2.5.3.1 Definição de aplicações

Kubernetes: uma aplicação pode ser implementada no *Kubernetes* utilizando uma combinação de serviços (ou microsserviços), implementações e *pods*.
Docker Swarm: As aplicações podem ser implementadas como micro-serviços ou serviços dentro de um swarm *cluster* no *Docker Swarm*. Os ficheiros YAML podem ser utilizados para identificar vários *containers*. Além disso, o *Docker Compose* pode instalar a aplicação.

2.5.3.2 Rede

Kubernetes: O modelo de rede é uma rede plana, permitindo que todos os *pods* interajam uns com os outros. As políticas de rede especificam a maneira de como os *pods* interagem entre si. A rede plana é implementada normalmente como uma sobreposição. O modelo precisa de dois CIDRs: um para os serviços e outro para o qual os *pods* adquirem um endereço IP.
Docker Swarm: O node que une um *Swarm cluster* gera uma rede de sobreposição para serviços que abrangem todos os *hosts* no *Docker Swarm* e uma rede de ponte do *Docker* somente de *hosts* para *containers*. Os utilizadores de sistemas têm a opção de criptografar o tráfego de dados do *container* enquanto criam uma rede de sobreposição por conta própria no *Docker Swarm*.

2.5.3.3 Escalabilidade

Kubernetes: Para sistemas distribuídos, o *Kubernetes* é mais um *framework all-in-one*. É um sistema complexo porque fornece fortes garantias sobre o estado do *cluster* e um conjunto unificado de APIs. Isso diminui o dimensionamento e a implementação do container.
Docker Swarm: Quando comparado ao *Kubernetes*, o *Docker Swarm* pode implementar *container* muito mais rápido e isso permite que tempos de reação mais rápidos aumentem conforme a demanda.

2.5.3.4 Elevada disponibilidade

Kubernetes: Todos os *pods* em *Kubernetes* são distribuídos entre *nodes* e isso oferece alta disponibilidade, tolerando a falha da aplicação. Os serviços de *Load Balancing* em *Kubernetes* detectam *pods* não íntegros e eliminam-nos, suportando alta disponibilidade.
Docker Swarm: Como os serviços podem ser replicados nos *nodes do Swarm*, o *Docker Swarm* também oferece alta disponibilidade. Os *manager nodes* no *Docker Swarm* são responsáveis pelo *cluster* inteiro e lidam com os recursos dos *Worker Nodes*.

2.5.3.5 Container Setup

Kubernetes: Utiliza suas próprias definições de YAML, API e cliente, e cada uma delas difere das suas correspondentes no *Docker*. Ou seja, o utilizador de sistemas não pode

utilizar o *Docker Compose* nem a CLI do *Docker* para definir *containers*. Ao alternar de plataformas, as definições e comandos do YAML precisam ser reescritos.

Docker Swarm: A API do *Docker Swarm* não engloba totalmente todos os comandos do , mas oferece grande parte da funcionalidade normal do *Docker*. Suporta a maioria das ferramentas que são executadas com o *Docker*. No entanto, se a API do *Docker* estiver com algum problema numa operação específica, não haverá uma maneira fácil de contorná-la utilizando o *Swarm*.

2.5.3.6 Balanceamento de carga

Kubernetes: Os *pods* que são expostos podem ser utilizados como balanceadores de carga no *cluster*. Geralmente, um ingresso é utilizado para balanceamento de carga.

Docker Swarm: O modo Swarm consiste num elemento DNS que pode ser utilizado para distribuir solicitações recebidas para um nome de serviço. Os serviços podem ser atribuídos automaticamente ou podem ser executados em portas especificadas pelo cliente.

2.5.4 Comparação dos sistemas no mercado atual

De acordo com o relatório da *RightScale* de 2018 e 2019 [FLE18] e [FLE19] sobre adoção de tecnologias para a *Cloud, o Docker e o Kubernetes* têm taxas de adoção muito elevadas.

No ano de 2018, com o aumento acentuado no uso de *containers*, o *Docker* continua a apresentar um forte crescimento. A adoção geral do *Docker* aumentou de 35 % para 49% em relação a 2017 (uma taxa de crescimento de 40%). O *Kubernetes*, ferramenta de orquestração de containers que alavanca o *Docker*, viu o crescimento mais rápido, quase o dobro para atingir 27% de adoção. Muitos utilizadores também escolhem ofertas de *containers-as-a-service* fornecidos pelas soluções *Cloud*. O serviço de *container da AWS (ECS / EKS)* seguiu de perto atrás do *Docker* com adoção de 44% (taxa de crescimento de 26%). A adoção do *Azure Container Service* atingiu 20% devido a um forte crescimento de 82%, e o *Google Container Engine* também cresceu fortemente (75%) para alcançar adoção de 14% [FLE18] .

No ano de 2019, o uso de *containers do Docker* continua a crescer, com a taxa de adoção aumentando de 49% em 2018 para 57% em 2019. O *Kubernetes* mais uma vez apresenta um crescimento mais rápido, aumentando de 27% para 48% de adoção, de 2018 para 2019. A adoção por empresas é ainda maior, com 66% a usar o *Docker* e 60% a usar o *Kubernetes*. Por outro lado, os maiores players na oferta de serviços na *Cloud* oferecem este tipo de serviços: O serviço de *containers da AWS (ECS / EKS)* tem 44% de adoção em 2019 (a partir de 2018), enquanto a adoção do *Azure Container Service* atinge 28% (acima dos 20% em 2018) e o *Google Container Engine* cresce ligeiramente, atingindo uma taxa de adoção de 15% [FLE19].

2.6 Principais Ferramentas de Virtualização

2.6.1 VMware

O *VMware* é uma plataforma que suporta *hosted* virtualização x86 que consegue correr um sistema operativo *guest* com alguma perda de performance. A arquitetura de x86 apresenta 4 níveis de privilégio conhecido como *Ring* 0,1,2 e 3 para o sistema operativo e aplicações conseguirem o acesso ao *hardware* do computador. Enquanto as aplicações normalmente correm no nível *Ring* 3, os sistemas operativos precisam de ter acesso direto à memória e hardware e isso exige instruções privilegiadas no nível *Ring* 0. Virtualizar arquiteturas de x86 requer colocar a camada de virtualização abaixo do sistema operativo (à qual é espectável que sejam executadas instruções no nível *Ring* 0) para criar e gerir máquinas virtuais que apresentem recursos partilhados. As instruções que não se encontrem no nível *Ring* 0 podem ser comprometer o desempenho da virtualização. Por ser bastante complicado esta situação das instruções de virtualização correrem no nível *Ring* 0, parecia impossível criar arquiteturas x86 de virtualização. O *VMware* veio resolver este desafio ao criar técnicas de tradução binária que permitem que o VMM corra dentro do nível *Ring* 0 no que toca no isolamento e performance, enquanto o sistema operativo era movido para outro nível com maior privilégio que as aplicações no nível *Ring* 3 mas com menos privilégios que o VMM no nível *Ring* 0 [VAS12].

2.6.2 Oracle VM Virtualbox

O *Oracle VM Virtualbox* é uma plataforma x86 *open-source* que oferece soluções de virtualização ao nível do software criada pela *Oracle Corporation*. O *Virtualbox* é o chamado hipervisor "*hosted*" e suporta todas as plataformas *host* tal como os formatos de imagens que são usados. Isto permite ao utilizador criar máquinas virtuais onde o *host OS é Windows* e a *guest OS* da VM é corrida em *Linux*. Tal como dito anteriormente, o *Virtualbox* utiliza virtualização ao nível do *software* para criar e correr máquinas virtuais. Este é o procedimento normal para qualquer VM (com excepção dos *guest OS* de 64-bit) criada dentro de qualquer ambiente no *Virtualbox*. Contudo, esta plataforma fornece a opção de ativar a virtualização ao nível do *hardware* [VAS12].

2.6.3 Comparação

É importante conhecer melhor estas duas plataformas para os utilizadores decidirem quais as melhores para o seu negócio e necessidades.
Em termos de preço e licenças, tal como referido o *Oracle VM Virtualbox* é uma plataforma *open-source* com uma licença GNU V2. Por outro lado, o *VMware* apresenta também uma versão grátis tal como versões pagas com funcionalidades ilimitadas. A versão paga do *VMware* é designada *VMware Workstation Pro para o Windows e VMware Fusion para sistemas Linux e macOS*. Já a versão grátis, é conhecida por *VMware Workstation Player* e é bastante limitada em termos de funcionalidades. Portanto conclui –se que em termos de preços o *Virtualbox* é o vencedor.

Em termos de *performance*, tem havido muitas discussões acerca da comparação entre os ambientes de virtualização destas duas plataformas. Em termos de CPU e Memória, o *VMware* supera bastante o *Virtualbox*. Ao nível de I/O estão bastante similares. Contudo, existem vários parâmetros aos quais muda da necessidade de cada utilizador e isso é o desafio maior dos engenheiros, saber qual é a melhor solução para certo problema. Para casos de uso de x64 *guests OS em x64 hosts OS, o VMware ultrapassa o Virtualbox*.
Na perspetiva das funcionalidades, ambas as plataformas inspiram –se uma na outra. Por exemplo, o *Virtualbox* apresenta os *snapshots e o VMware* apresenta os *rollback points* para reverter todo os processo que esteja a ser feito na máquina virtual caso exista algum problema interno nesta. Ambas têm integração de aplicações virtualizadas no *desktop* nativo. No *VMware* é designado de *Unity mode* e no *Virtualbox* é designado de *seamless mode* e ambos permitem ao utilizador abrir aplicações do *Windows* na máquina *host*, enquanto uma máquina virtual suporta essa mesma aplicação que está a correr em *background*. A maior parte das funcionalidades no caso do *VMware* apenas são apresentadas nas versões pagas enquanto na *Virtualbox* é completamente grátis.
Em termos da interface do utilizador, o *Virtualbox* apresenta uma interface mais simples e fácil de compreender. Todas as funcionalidades estão à visto do utilizador, entre as quais as mais básicas como criar, modificar, começar, parar ou eliminar uma máquina virtual. Outras funcionalidades como gerir o armazenamento, o CPU e a rede interna também são funcionalidades bastante importantes e fáceis de gerir nesta interface. Por outro lado, o *VMware* apresenta uma interface mais complicada, onde os nomes das funcionalidades do menu são mais técnicos e mais difícil de entender para utilizadores menos entendidos. Isto acontece porque no início, esta plataforma foi desenvolvida mais para profissionais de *Cloud* e gestão de servidores de virtualização, ou seja não utilizadores comuns. Neste tópico, no que toca à *interface, o Virtualbox* mais uma vez supera o *VMware* [RAN18].
Em suma, pode –se concluir que o *Oracle VM Virtualbox* apresenta melhores condições. No presente documento, todas as instalações e testes foram realizadas em máquinas virtuais no *Virtualbox*. Isto deve –se ao facto de todas as funcionalidades apresentadas neste capitulo tal como experiências no passado com esta plataforma. Apesar do *VMware* dominar o mercado em termos de empresas [RAN18], o *Virtualbox* é melhor solução para utilizadores que, por exemplo, pretendam fazer testes.

2.7 Mecanismos Secundários Utilizados

Com o objetivo de utilizar várias ferramentas para realizar vários testes nos os *clusters* tanto do *Kubernetes e Docker Swarm* é necessário apresentar estas mesmas ferramentas.

2.7.1 NGINX

O *NGINX* é um *software open-source para web serving, reverse proxy, cache, load balacing*, entre outras funcionalidades. Tudo começou como um *web server* projetado para desempenho e estabilidade máximos. Além dos recursos do servidor HTTP, o *NGINX* também pode funcionar como servidor *proxy para email (IMAP, POP3 e SMTP) e*

como reverse proxy e load balacing para servidores HTTP, TCP e UDP [ngi].

2.7.2 GUI

Uma GUI é um sistema de componentes visuais interativos para ajudar à melhor compreensão de um software. Uma GUI exibe objetos que transmitem informações e representam ações que podem ser executadas pelo utilizador. Os objetos mudam de cor, tamanho ou visibilidade quando o utilizador interage com eles [COM19]. Nesta perspetiva, uma GUI não é algo necessário aos testes mas ajuda os utilizadores que não consigam perceber a CLI de cada uma das ferramentas de orquestração. No caso do *Kubernetes* foi configurado e instalado uma GUI associada ao próprio *software* de orquestração designada de *kubernetes dashboard*. No *Docker Swarm* foi configurado e instalado uma GUI externa ao Docker designada de *Portainer.io*. Cada uma das GUI apenas podem ser instaladas no *node* que gere todo o *cluster*.

2.7.2.1 Kubernetes Dashboard

O *Kubernetes Dashboard* é uma GUI que ajuda os utilizadores fazer o *deploy de containers* dentro de um *cluster no Kubernetes* e gerir todos os recursos desse mesmo *cluster*. É possível verificar todos os estados relacionados com cada recurso do *cluster* entre os quais mais importantes os *deployments, pods e nodes*. Em termos práticos, é possível fazer *scale de um deployment, iniciar um update, reiniciar um pod e também fazer o deploy* de algumas aplicações com base em *templates* de imagens [KDA20]. O *kubernetes dashboard* apenas pode ser instalado e configurado no *master node*.

2.7.2.2 Portainer.io

O *Portainer.io* é uma GUI que permite aos utilizadores configurar e gerir os ambientes no qual é instalado. Neste caso particular, esse ambiente é o *Docker Swarm*. É uma ferramenta bastante simples tanto de usar como gerir. Com a ajuda do *Docker Engine*, esta ferramenta permite aos utilizadores que não entendem a CLI do *docker*, organizarem o *cluster* de uma maneira simples, eficaz e rápida. Permite gerir todos os recursos dos *cluster do Docker Swarm* tais como *containers, images, volumes e redes* [POR20B]. O *portainer.io*, tal como acontecia no *kubernetes dashboard*, apenas pode ser instalado e configurado no *manager node*.

2.7.3 Benchmarking

O *Benchmarking* surgiu como uma tentativa de melhoria de práticas empresariais e de alcance de desempenhos superiores. Trata-se de ferramentas de comparação empresarial e de gestão de empresas, que começa com uma ávida pesquisa e termina com a implementação de ações específicas [EDD02]. Neste caso em específico, as ferramentas de *benchmarking* servem para fazer uma avaliação de tudo o que está acontecer em cada um dos *clusters* e os comportamentos que se deve fazer para melhorar o *cluster*.

2.7.3.1 Prometheus

O *Prometheus* é uma ferramenta *open-source* de monitorização originalmente criado pela *SoundCloud*. Desde o seu começo em 2012, muitas empresas e organizações adotaram o *Prometheus* e o projecto é uma comunidade de *developers* e utilizadores muito ativa [PRO20]. Esta ferramenta coleciona métricas extraídas dos sistemas a monitorizar por *HTTP endpoints*. Dando um exemplo de uma máquina virtual ou até mesmo uma aplicação, são executados processos na máquina *host* e portanto existem métricas específicas que precisam ser monitoradas, como memória e armazenamento usados, bem como relatórios gerais sobre o estado da máquina ou aplicação. Convenientemente, o *Prometheus* expõe uma ampla variedade de métricas que podem ser facilmente monitorizadas. Contudo, o *Prometheus* não é melhor ferramenta no que toca a exposição dos dados, daí a ser necessário utilização de outros programas opcionais para executar essa tarefa, como exemplo o *Grafana* [GRA17].

2.7.3.2 Grafana

O *Grafana é um software open-source* de análise de dados. A função do *Grafana* serve para receber os dados e expô –los visualmente para que o utilizador faça uma avaliação rigorosa. São apresentados vários componentes gráficos para analisar os dados, tais como gráficos e listas [GRA20]. A combinação entre o *Prometheus e o Grafana* é cada vez mais comum entre as equipas *DevOps* para o armazenamento e visualização dos dados. O *Prometheus* atua com um *backend* de armazenamento de dados e o *Grafana uma interface* para visualizar esses dados. Apesar de existirem gráficos no *Prometheus*, a informação é bastante mais explorada e exposta de uma maneira mais prática com a ajuda do *Grafana*. De modo a facilitar o trabalho dos seus utilizadores, existem *templates* dos serviços já previamente criados com todos os gráficos e listas que distribuem os dados que vem do *cluster* tanto do *Kubernetes ou do Docker Swarm* [GRA17].

2.8 Conclusão

Neste capítulo foram abordados vários temas importantes acerca da virtualização. É importante adquirir informação referente a uma comparação teórica das ferramentas de modo a testar na parte prática presente no terceiro capítulo e posteriormente a análise dos dados no quarto capítulo. A escolha da utilização do *Oracle VM Virtualbox* foi também importante devido a experiências anteriores. É importante também destacar os mecanismos secundários devido à sua importância numa perspetiva de utilizador de IT, ou seja estas ferramentas podem ser fundamentais na tarefa de administração e suporte dos *clusters* das empresas.

Capítulo 3

Implementação do Ambiente Experimental

3.1 Introdução

Neste capítulo é apresentado todo o ambiente experimental, os recursos de hardware e software utilizados, as versões de cada elemento utilizado e todas as instalações e configurações de cada ferramenta. A partir da demonstração da implementação dos *clusters e do web-server NGINX* é possível realizar os testes para medir quais os pontos fortes de cada uma das ferramentas de orquestração de *containers*. É vital destacar que o facto de a experiência ser realizada em máquinas virtuais, permitiu uma elevada possibilidade de correção de erros que apareciam ao longo de todo o processo de instalação e configuração.

3.2 Test Bed Experimental

3.2.1 Caracterização do Test Bed Experimental

O *test bed* é constituído por seis máquinas virtuais, três das quais serviram para o *cluster* do *Kubernetes* e as outras três para o *cluster do Docker Swarm*. Todas as máquinas têm as mesmas características e estão conectadas à mesma rede (192.168.1/24). Apesar do *host* não ter qualquer influência na preparação dos *clusters* além de alojar as máquinas que servem para o efeito, foi decidido conectar as máquinas à rede local do *host* através da opção "*Bridged Adapter*". Com esta opção, o *virtualbox* recebe todas as especificações da rede do *host* e coneta a rede virtual à rede local. Posteriormente, o *virtualbox* atribui os endereços IPs dentro da mesma rede (192.168.1/24) a partir do protocolo DHCP [BRI12]. Os *clusters* criados são compostos por um *master node* (*manager node no Docker Swarm*) e dois *worker nodes*. Cada node foi instalado em cada máquina virtual que contêm o sistema operativo *Ubuntu 16.04 LTS*. A escolha de um sistema operativo de *Linux* baseia – se no facto de que o *Linux* é um sistema operativo melhor que o *Windows*, na sua arquitetura, especialmente o *Kernel* e o sistema de ficheiros. Os *containers* aproveitam o isolamento do processos no *Linux*, juntamente com os *namespaces*, para criar processos isolados [WVS18]. O storage dos recursos é todo realizado localmente pelos *nodes*, devido à pequena dimensão deste ambiente experimental. Na figura 3.1 está representado o esquema de rede do ambiente experimental.

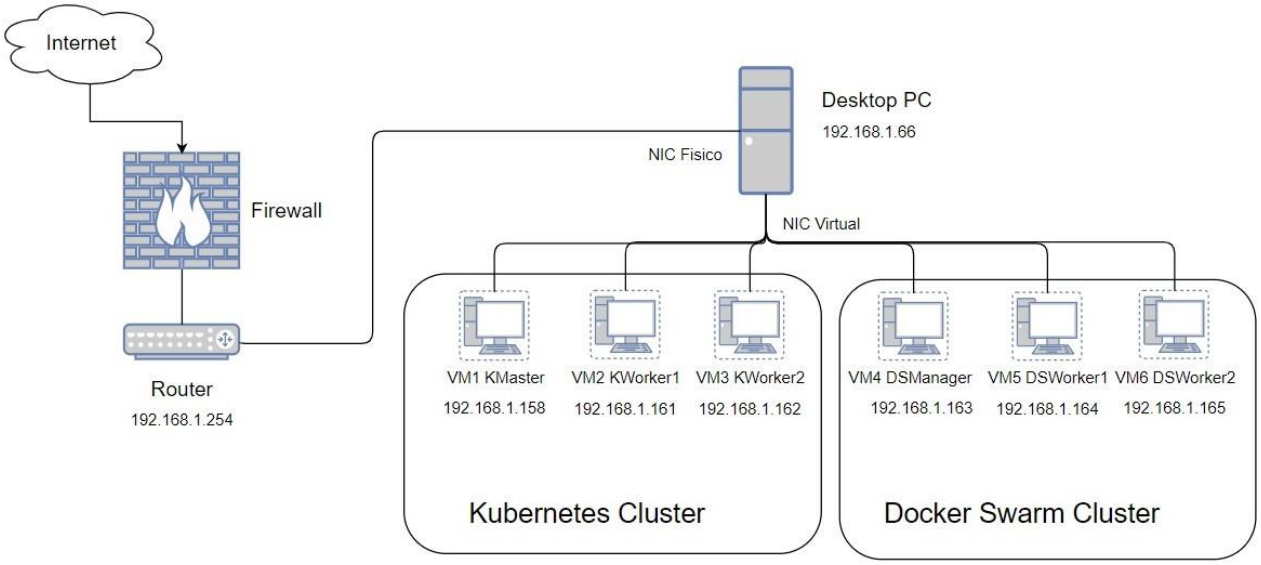

Figura 3.1: Esquema de rede do ambiente experimental.

3.2.2 Especificações de Hardware e Software do Test Bed

Host	
CPU	Intel(R) Core (TM) i7-7700 CPU @ 3.60GHZ, 4 Núcleos, 8 Processadores lógicos
Memória	G.SKILL DDR4-2400 PC4-19200 8192MB (x2)
Disco	Samsung SSD 960 EVO 250 GB ST2000DM001-1ER164
Motherboard	STRIX H270F GAMING
Sistema Operativo	Windows 10 Pro

Tabela 3.1: Especificações de Hardware e Software da máquina host.

Nodes dos Clusters	
CPU	Intel(R) Core (TM) i7-7700 CPU @ 3.60GHZ, 4 Núcleos, 8 Processadores lógicos 1 (Docker Swarm) e 2 (Kubernetes) vCPU
Memória	Virtualbox Memory 4 GB
Disco	Virtualbox Hard disk (40GB)
Motherboard	Virtualbox Motherboard
Sistema Operativo	Ubuntu 16.04 LTS

Tabela 3.2: Especificações de Hardware e Software dos nodes dos clusters.

Versões dos Softwares	
Docker	V19.03.12
Docker Swarm	V19.03.12
Portainer.io	V1.24.0
Mozilla Firefox	V78.0.1
Prometheus	V2.5.0
Grafana	v5.3.4
NGINX	V1.19.1

Tabela 3.3: Versões dos *Software* do *cluster no Docker Swarm*.

Versões dos Softwares	
Docker	V19.03.12
Kubernetes	V1.18.0
Calico	v3.8.0
CoreDNS	V1.6.7
Kubernetes Dashboard	V2.0.3
Mozilla Firefox	V78.0.1
Helm	v3.3.0
Prometheus	V2.19.0
Grafana	v7.0.3
NGINX	V1.19.1

Tabela 3.4: Versões dos *Software* do *cluster no Kubernetes*.

3.2.3 Instalação e configuração dos Clusters

Para a instalação e configuração dos *clusters* de cada uma das ferramentas de orquestração de *containers* presentes neste documento, foi necessário procurar guias e documentação adequada de modo a seguir todos os passos para a instalação dos *clusters*. Sem qualquer tipo de documentação oficial, foi feita uma pesquisa de sites mais "confiáveis". No caso do *Docker Swarm*, os sites encontrados foram [BHO18] e [COH18] . No caso do *Kubernetes* foram [CLO18], [MEN19] e [BOS19].

3.2.3.1 Docker Swarm

Para criar um *cluster* no *Docker Swarm* é necessário instalar o docker em todas as máquinas. Para isso, é necessário instalar o repositório do docker e posteriormente o *Docker Community Edition*. Em primeiro lugar, instala –se os pacotes que permitem utilizar este repositório sobre o HTTPS, em segundo lugar, é necessário fazer o *download* da chave GPG oficial do *docker* e do repositório estável do *docker*. Por último, instala –se o *docker CE* com o auxílio da chave e do repositório e após a conclusão da instalação, ativar o serviço do *docker* nos sistemas das máquinas virtuais. Com o objetivo de testar o *docker*, é executado o comando "sudo docker run hello-world". A figura 3.2 representa a verificação do comando.

Figura 3.2: Docker Hello World.

Após todos os passos anteriores, é apenas necessário correr o código que identifica uma das máquinas como *manager do cluster*. É gerado um "**token swarm**", presente na figura 3.3, para adicionar as outras máquinas como *workers no cluster*. Na figura 3.4 está representada a confirmação da entrada de um dos *workers no cluster*.

Figura 3.3: Docker Swarm Join Token.

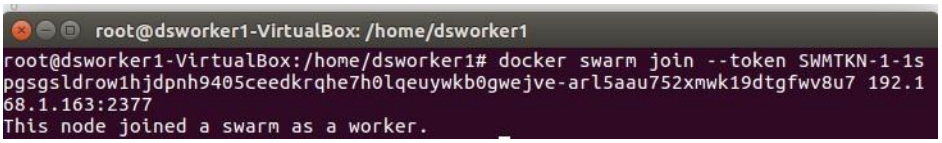

Figura 3.4: Confirmação da entrada do worker no cluster.

Utilizando o comando "sudo docker node ls", são apresentados todos os elementos do *cluster* bem como o identificador dos *hostnames*, o estado e a versão do *docker* de cada um, como se pode verificar na 3.5.

Figura 3.5: Lista de nodes do cluster do Docker Swarm.

Todos os comandos para a instalação do *cluster do Docker Swarm* encontram-se no anexo A1.

3.2.3.2 Kubernetes

Tal como aconteceu com a instalação do *Docker Swarm*, o *docker* é instalado exatamente da mesma maneira. Contudo existe uma diferença antes desse procedimento. É necessário

desativar a memória *swap* visto que o *Kubernetes* não suporta o ficheiro *swap* presente nos sistemas das máquinas. Após a instalação do docker é necessário instalar os componentes do Kubernetes (kubeadm,kubectl e kubelet). Para isso, tal como acontecia no *docker*, é necessário fazer *download* da chave GPG oficial e do repositório do *Kubernetes*. De destacar que o *Kubernetes* não pode ser instalado se as máquinas não possuírem pelo menos 2 vCPU. Por fim, instalam –se os todos os componentes do *Kubernetes* anteriormente referidos. Com a conclusão de todos os passos anteriores, agora é fundamental configurar o *master node do cluster*. Para isso, é preciso inicializar o *cluster* utilizando o endereço IP privado da máquina que servirá de *master*. É gerado um **token** de associação que servirá para as outras máquinas se juntarem ao *cluster como workers* e três comandos específicos que precisam ser inseridos no *master* para adicionar permissões aos ficheiros de configuração do sistema. Na figura 3.6 é representada a confirmação da inicialização do *master no cluster*.

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as
 root:

kubeadm join 192.168.1.107:6443 --token s8yopf.dw271vrd57676vzb \
    --discovery-token-ca-cert-hash sha256:7da6a1f43581bb1cbcd5ce4437aafc567dfdfe
e885c03804438da8f071ed4b29
root@kubernetesmasternode-VirtualBox:/home/kubernetes-masternode#
```

Figura 3.6: Inicialização do *master node no cluster*.

Contudo, o *master node* ainda não está preparado. Isto acontece devido ao facto de ainda ser necessário instalar um CNI. O CNI escolhido foi o *Calico*, apesar de haver outras opções. Quando o *Calico* estiver instalado e configurado no *cluster*, o *master node* está preparado para executar as suas funções. Da parte dos *workers*, é preciso utilizar o *token* para entrarem no *cluster*. A confirmação está presente na figura 3.7.

```
root@kworker2-VirtualBox:/home/kworker2# kubeadm join 192.168.1.158:6443 --token b6e6ue.vgwg9919i2
9x6xq3 --discovery-token-ca-cert-hash sha256:fd796d01dfa91d6e9990a91a68fa49c995669eecb44a4adddd219
5c8460a6219
W0713 19:58:57.296249   18541 join.go:346] [preflight] WARNING: JoinControlPane.controlPlane setti
ngs will be ignored when control-plane flag is not set.
[preflight] Running pre-flight checks
        [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The recom
mended driver is "systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-conf
ig -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.18" ConfigMap
 in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flag
s.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Figura 3.7: Associação do Worker Node no cluster.

Utilizando o comando "sudo kubectl get nodes", são apresentados todos os elementos do *cluster* bem como o identificador, o estado e a versão do *docker* de cada um, como se pode verificar na figura 3.8.

```
root@kmaster-VirtualBox: /home/kmaster
Every 2,0s: kubectl get nodes                          Mon Jul 13 21:16:34 2020

NAME                 STATUS   ROLES    AGE   VERSION
kmaster-virtualbox   Ready    master   14d   v1.18.0
kworker1-virtualbox  Ready    <none>   14d   v1.18.0
kworker2-virtualbox  Ready    <none>   77m   v1.18.0
```

Figura 3.8: Lista de *nodes do cluster do Kubernetes*.

É importante referir que após a instalação do *Calico*, existem vários componentes que são instalados para o funcionamento deste CNI, com principal destaque o *CoreDNS*. Este elemento é um servidor DNS flexível que serve como DNS da rede interna no *Kubernetes* [KUB20B]. O *CoreDNS* também é fundamental para que o *Prometheus* tenha acesso aos dados do *cluster*. Na figura 3.9 são apresentados todos os elementos da rede interna do *cluster* bem como o node onde está cada componente instalado.

```
NAMESPACE              NAME                                                READY   STATUS    RESTARTS   AGE     IP               NODE
default                pod/nginx-f89759699-dpkt6                           1/1     Running   0          4m28s   192.168.119.7    kworker2-virtualbox
kube-system            pod/calico-kube-controllers-75d555c48-bg48n         1/1     Running   27         14d     172.17.0.2       kmaster-virtualbox
kube-system            pod/calico-node-fg4bj                               1/1     Running   1          86m     192.168.1.166    kworker2-virtualbox
kube-system            pod/calico-node-gvfhm                               1/1     Running   14         14d     192.168.1.161    kworker1-virtualbox
kube-system            pod/calico-node-m5cz9                               1/1     Running   12         14d     192.168.1.158    kmaster-virtualbox
kube-system            pod/coredns-66bff467f8-57cpt                        1/1     Running   6          54m     172.17.0.6       kworker1-virtualbox
kube-system            pod/coredns-66bff467f8-r5qvj                        1/1     Running   0          52m     172.17.0.6       kmaster-virtualbox
kube-system            pod/coredns-66bff467f8-spfwx                        1/1     Running   0          54m     192.168.119.6    kworker2-virtualbox
kube-system            pod/etcd-kmaster-virtualbox                         1/1     Running   22         14d     192.168.1.158    kmaster-virtualbox
kube-system            pod/kube-apiserver-kmaster-virtualbox               1/1     Running   21         14d     192.168.1.158    kmaster-virtualbox
kube-system            pod/kube-controller-manager-kmaster-virtualbox      1/1     Running   28         14d     192.168.1.158    kmaster-virtualbox
kube-system            pod/kube-proxy-hwzmf                                1/1     Running   13         14d     192.168.1.161    kworker1-virtualbox
kube-system            pod/kube-proxy-lsld9                                1/1     Running   1          86m     192.168.1.166    kworker2-virtualbox
kube-system            pod/kube-proxy-nnsdp                                1/1     Running   12         14d     192.168.1.158    kmaster-virtualbox
kube-system            pod/kube-scheduler-kmaster-virtualbox               1/1     Running   24         14d     192.168.1.158    kmaster-virtualbox
```

Figura 3.9: Componentes do *Calico* instalados em cada *node do cluster*.

Todos os comandos para a instalação do *cluster do Kubernetes* encontram-se no anexo A2.

3.2.4 Instalação e configuração das ferramentas de Benchmarking

O *Prometheus e o Grafana* foram ferramentas fundamentais para monitorizar o estado dos *clusters*. Enquanto o *Prometheus* captura os dados do *cluster* e do sistema, o *Grafana* recebe esses mesmos dados e apresenta –os de uma maneira interativa e compreensiva. Apesar de não ter qualquer influência nos testes, é fundamental aprender a configurar este tipo de ferramentas que dão ao utilizador informações vitais para a antecipação de erros ou problemas. Para a realização da instalação foram utilizados projetos de instalação previamente criados para cada um dos casos. No *Docker Swarm* o projecto é designado de "*Swarmprom*" e no Kubernetes designado de "*Prometheus Operator*". Mais uma vez sem documentação oficial, foi feita uma pesquisa de sites mais "confiáveis". No caso do *Docker Swarm* o site encontrado foi [SWP17]. No caso do *Kubernetes* o site encontrado foi [KPR17] e [DES20] .

3.2.4.1 Docker Swarm

A criação de projetos para facilitar a instalação de certos *softwares* é cada vez mais frequente nos dias de hoje. Como referido anteriormente, no *Docker Swarm*, o projeto designado de "*Swarprom*" foi criado em 2017 por parte de *Stefan Prodan*. Este projeto permitiu instalar os serviços necessários que permitem monitorizar o *cluster*. Além dos mais importantes, os já referidos *Prometheus e Grafana*, são instalados outros componentes que são importantes para a recolha de informações dentro do *cluster*. São eles:

- **Prometheus** - utilizado como base de dados para as métricas recolhidas.
- **Grafana** - utilizado para visualizar as métricas recolhidas.
- **Node-Exporter** – utilizado para recolher métricas do sistema de onde o *cluster* está instalado.
- **Cadvisor** – utilizado para recolher métricas dos *containers do cluster*.
- **Blackbox-Exporter** – utilizado para permitir ao utilizador acesso à monitorização a partir dos *endpoints HTTP, DNS, TCP and ICMP*.
- **Alertmanager**- utilizado para alertas quando algo está de errado no *cluster*.

Todos estes serviços são utilizados em *background*, mas o *Prometheus e o Grafana* são os mais importantes e aqueles que serão destacados. A instalação do projeto é bastante simples. Para isso é necessário fazer *download* da pasta do projeto que se encontra no *GitHub*, entrar na pasta onde foi guardada e correr o comando "*Docker stack deploy -c Docker-compose.yml mon*". O objetivo deste comando é criar uma *stack no cluster do Docker Swarm* com nome "*mon*". O "*Docker-compose.yml*" é o ficheiro de instalação dos serviços anteriormente referidos na *stack* "mon". Com o comando "*docker stack ps mon*" é possível verificar os serviços do projeto bem como os *nodes do cluster* onde se encontram. Os serviços encontram-se na figura 3.10. É importante destacar que o *Docker Swarm* distribuiu os serviços pelos *nodes* de forma equitativa.

Figura 3.10: Lista de serviços em funcionamento na *stack "mon"*.

Após a conclusão da instalação, é importante verificar se a base de dados do *Prometheus* está a funcionar corretamente. Para isso é necessário abrir o *browser*, utilizando o *Mozilla Firefox*, e navegar na página do *Prometheus* a partir do endereço "https://localhost:9090". Na página principal existe um menu que contêm várias opções. De modo a verificar o bom funcionamento do *Prometheus* é necessário entrar em *Status -> Targets*.
Os outros serviços são também apresentados, mas não têm grande influência no que toca à visualização dos dados no *Grafana*. Para visualizar os dados é necessário entrar na página do *Grafana*. Utilizando o endereço "*https://localhost:3000*", é possível entrar na página de *login do Grafana*. Dentro do projeto, as credenciais foram predefinidas como utilizador "*admin*" e password "*admin*". Sendo uma experiência, as credenciais não são algo necessário em ter em conta, contudo em casos de monitorização de *softwares* dentro de empresas, a segurança destes serviços é fundamental e deve ser bem reconfigurada.
O *Grafana* tem uma particularidade muito importante que facilita o trabalho aos engenheiros de IT. Existem várias *dashboards* de dados já previamente criadas facilmente acessíveis na internet que servem como menus de elementos que fornecem dados do que é pretendido. Por exemplo, existem *dashboards* disponibilizadas na internet acerca de informações tanto dos *clusters no Docker Swarm* como no *Kubernetes*. Em cada *dashboard*, são apresentados elementos de visualização como gráficos ou tabelas dos dados recolhidos. A única exigência que é necessário na implementação destas *dashboards* é a base de dados de onde são retirados os dados a visualizar. Dentro deste projeto existem duas *dashboards para o Grafana*.
Na primeira *dashboard* são apresentadas as métricas fundamentais para a monitorização dos recursos usados pelos serviços e *stacks do Docker Swarm* e podem ser filtrados pelos IDs dos *nodes*. São elas:

- Número de *nodes, stacks,* serviços e *containers* em funcionamento.
- Gráfico das tarefas do *Docker Swarm* ordenadas pelos nomes dos serviços.
- Gráfico de verificação de estado do *cluster* (todas as verificações em funcionamento ou falhadas).
- Gráfico dos Top 10 de serviços e *containers* que representa a utilização de CPU.
- Gráfico dos Top 10 de serviços e containers que representa a utilização de memória.
- Gráfico da utilização de rede por partes dos serviços (recebido e transmitido).
- Tráfego de rede no *cluster* e gráficos de IOPS.

- Ações por parte dos containers e da rede do *Docker Engine por node*.
- Lista de informações (versão, *ID do node, OS, kernel* e drive do gráfico) do *Docker Engine*.

Na segunda *dashboard* são apresentadas as métricas fundamentais para a monitorização dos recursos usados pelos *nodes do Docker Swarm* e podem também ser filtrados pelos *IDs dos nodes*. São elas:

- O *up-time do cluster*, número de *nodes*, número de CPUs, e o medidor de percentagem de *CPU idle*.
- Gráficos de carga média do sistema e utilização de CPU por *node*.
- Número utilizado de Memória total , medidor de percentagem de memória disponível, número de espaço de disco total e medidor de percentagem de armazenamento disponível.
- Gráfico de memória utilizada por *node*.
- Gráfico de operações I/O utilizadas.
- Gráficos de IOPS e informação de quanto tempo o CPU precisa de esperar para que as operações I/O a estarem concluídas.
- Gráfico dos containers em funcionamento por serviços e *nodes*.
- Gráfico de rede utilizada (*inbound Bps, outbound Bps*).
- Lista de informações (instância, id e o nome) dos *nodes*.

3.2.4.2 Kubernetes

Os "Operators" foram criados pelo *CoreOS* como classes de *software* que operam outro *software*, introduzindo o conhecimento operacional coletado pelos engenheiros de IT em *software*. O projeto *Prometheus-Operator* serve para utilizar a execução do *Prometheus* sobre o *Kubernetes* o mais simples possível, preservando sempre as configurações nativas do Kubernetes [COR20]. Tal como acontecia no *Docker Swarm*, os elementos instalados mais importantes continuam a ser o *Prometheus e Grafana*, porem são instalados outros componentes que são importantes na execução do processo de monitorização. São eles:

- **Prometheus** - utilizado como base de dados para as métricas recolhidas.
- **Grafana** - utilizado para visualizar as métricas recolhidas.
- **Node-Exporter** – utilizado para recolher métricas do sistema de onde o *cluster* está instalado.
- **Kube-state-metrics** - serviço simples que escuta o servidor da API do *Kubernetes* e gera métricas sobre o estado dos objetos do *cluster*.

- **Alertmanager** - utilizado para alertas quando algo está de errado no *cluster*.

Para a instalação do projeto é necessária uma ferramenta que simplifica a instalação e gestão de aplicações no *Kubernetes*. Esta ferramenta é designada por *Helm*. Criado pela CNCF, o *Helm* fornece os "*charts*" que são um conjunto de ficheiros ou recursos pré-configurados que ajudam os utilizadores instalarem as aplicações pretendidas. Para instalar o *Helm* é necessário fazer *download* da versão desejada que se encontra em [HVE20]. Descompactar o ficheiro com o comando "**tar -zxvf 'ficheiro'**" e alterar o diretório do ficheiro de configuração com o comando "*mv linux-amd64/helm /usr/local/bin/helm*" [HSI20]. Após a configuração bem-sucedida do *Helm* é possível prosseguir à instalação do *Prometheus-Operator*. O primeiro passo é criar um *namespace no Kubernetes*. O benefício do *namespace* é que permite agrupar todos os elementos do projeto e se algo correr mal, será possível eliminar tudo apenas eliminando o *namespace*. O nome do *namespace* é obrigatório ser "monitoring" devido ao facto de alguns ficheiros de configuração presentes no *chart* dependerem de criar os recursos dentro de um *namespace* com este nome. Correndo o comando "*helm install prometheus-operator stable/prometheus-operator –namespace monitoring*" é possível instalar o *chart do Prometheus-Operator*. Caso o repositório dos *charts* não seja encontrado no *Helm*, é necessário ser adicionado manualmente a partir do comando "*helm repo add stable https://kubernetes-charts.storage.googleapis.com/*" e posteriormente atualizar o repositório do *Helm* com o comando "*helm repo update*". A figura 3.11 representa todos os elementos instalados com sucesso no *namespace "monitoring"*.

```
monitoring              service/alertmanager-operated                        ClusterIP   None             <none>        9093/TCP,9094/TCP,9094/UDP
monitoring              service/prometheus-operated                          ClusterIP   None             <none>        9090/TCP
monitoring              service/prometheus-operator-alertmanager             ClusterIP   10.105.201.16    <none>        9093/TCP
monitoring              service/prometheus-operator-grafana                  ClusterIP   10.105.109.230   <none>        80/TCP
monitoring              service/prometheus-operator-kube-state-metrics       ClusterIP   10.96.50.255     <none>        8080/TCP
monitoring              service/prometheus-operator-operator                 ClusterIP   10.98.143.35     <none>        8080/TCP
monitoring              service/prometheus-operator-prometheus               ClusterIP   10.102.130.144   <none>        9090/TCP
monitoring              service/prometheus-operator-prometheus-node-exporter ClusterIP   10.101.170.117   <none>        9100/TCP
```

Figura 3.11: Lista de serviços em funcionamento no *namespace "monitoring"*.

Tal como no *Docker Swarm*, é importante verificar o bom funcionamento do *Prometheus e do Grafana*. No *Kubernetes* para aceder às *dashboards* de cada um, é necessário correr um comando de *port-forwarding*. Este comando "*kubectl port-forward -n monitoring 'pod da app' 'porta'*" permite ao utilizador redirecionar a porta de uma aplicação para ter acesso no computador pessoal. Como acontecia no *Docker Swarm*, a porta do *Prometheus* é 9090 e 3000 para o *Grafana*. Posto isto, é agora necessário abrir o *browser*, utilizando o *Mozilla Firefox*, e navegar na página do *Prometheus* a partir do endereço "*https://localhost:9090*". Apesar de as versões do *Prometheus* serem diferentes, o acesso ao menu é exatamente realizado da mesma maneira. Com o endereço "*https://localhost:3000*", é possível entrar na página de *login do Grafana*. Tal como acontecia no projecto *Swarmprom*, no projeto *Prometheus-Operator*, as credenciais também foram predefinidas. Neste caso, como utilizador "*admin*" e password "*prom-operator*". Ao contrário do que acontecia no *Docker Swarm*, as *dashboards* não foram configuradas no projeto. Portanto, foi feita uma pesquisa minuciosa da melhor *dashboard* acessível no *website oficial do Grafana*. A *dashboard* escolhida encontra-se em [REF20]. De modo a ser utilizada na página *online do*

Grafana é necessário importar a *dashboard*. Existem duas alternativas para esse efeito. A primeira alternativa é fazer download do ficheiro *json da dashboard* e posteriormente fazer o *upload no Grafana*. A segunda alternativa é mais simples e consiste em copiar o url do *website ou o id dashboard* e colar no painel de *import*. Após este processo, é necessário escolher o nome da *dashboard*, a pasta onde fica instalada e a base de dados de onde vêem os dados a visualizar, neste caso o *Prometheus*. Na *dashboard* escolhida são apresentadas as métricas fundamentais para a monitorização dos recursos usados no *Kubernetes* e podem ser filtrados pelos *IDs dos nodes*. São elas:

- Número de *nodes no cluster*.
- Medidores de percentagem de *pods*, CPU, memória e espaço de disco utilizadas no *cluster*.
- Número de *nodes* não disponíveis.
- Gráficos de capacidade de *pods*, CPU, memória e espaço de disco utilizadas no *cluster*.
- Gráfico de rede utilizada.
- Número de *deployment* réplicas total, disponíveis, atualizadas e não disponíveis.
- Número de *pods* em funcionamento, em espera, desconhecidos, falhados e sucedidos.
- Número de *containers* em funcionamento, preparados, em espera, reiniciados e terminados.

Em suma, é de destacar a importância destas ferramentas para a realização desta tese. Foi, muitas vezes, a partir das ferramentas de monitorização que foram descobertos erros no *cluster*. A monitorização deve ser algo importante que tem de ser tomada em consideração pelas empresas para permitir aos seus clientes segurança em tempo real.

3.2.5 Instalação e configuração dos GUI

Existem diversas maneiras de implementar objetos dentro dos *clusters* tanto do *Docker Swarm como o Kubernetes*. Todo o processo de instalação e gestão dos *clusters* foi feita pela linha de comandos. Contudo, é importante saber gerir os recursos através de ferramentas que facilitem esse trabalho. Ou seja, ferramentas interativas que permitam a utilizadores inexperientes com a CLI administrar os elementos de cada um dos *clusters*. No caso do *Docker Swarm*, tal como referido anteriormente, foi escolhido um programa exterior designado de *Portainer.io*. Apesar de existirem outras opções, o *Portainer.io* é um dos softwares mais utilizados para administrar recursos no Docker Swarm. No caso do Kubernetes, existe mesmo uma *dashboard* própria para a gestão dos recursos designada de *Kubernetes Dashboard*. Em termos de instalação das ferramentas, foi encontrada documentação oficial em cada um dos casos. No caso do *Portainer.io* encontra –se em

[POR20A] e no caso do *Kubernetes Dashboard* em [KUB20A]. É importante destacar que as ferramentas não foram inteiramente exploradas devido ao facto de não atenderem diretamente ao tema principal do presente documento.

3.2.5.1 Docker Swarm

O *Portainer.io* é composto por dois elementos, o *Portainer Server e o Portainer Agent*. Ambos os elementos são executados como *containers* dentro do *cluster no Docker Swarm*. O processo de implementação é bastante simples. O objetivo é instalar diretamente o *Portainer.io* como um serviço dentro do *cluster* . Este método implementa automaticamente uma única instância do *Portainer Server no Manager node* e o *Portainer Agent* como um serviço global em todos os *nodes do cluster*. Em primeiro lugar é necessário fazer *download* do ficheiro de instalação do *Portainer.io* por meio do comando "*curl -L https://downloads.portainer.io/portainer-agent-stack.yml-oportainer-agent-stack.yml*". Por último, é necessário instalar um *stack* designado de "*portainer*" onde sejam armazenados todos os serviços anteriormente referidos dentro do Docker Swarm. O comando para esse efeito é "*docker stack deploy –compose-file=portainer-agent-stack.yml portainer*". Por predefinição, o *Portainer.io* utiliza a porta 9000 do *Docker Engine* para aceder à *dashboard no browser*. Com a ajuda do comando "*https://localhost:9000/*", é possível aceder à *dashboard*. Em primeiro lugar, é necessário fazer a autenticação. As credenciais para aceder ao menu principal estão predefinidas como utilizador "*admin*" e *password "admin"*. Sempre que necessário será possível alterar a *password* posteriormente. Após o processo de autenticação, é necessário verificar se os *Portainer Agents* implementados em cada *node* estão a funcionar corretamente.
O *Portainer.io* têm acesso aos dados de todos os *nodes* através dos *Portainer Agents*. A lista é bastante detalhada na perspetiva em que dá informações essenciais acerca dos *containers* tais como:

- Nome.
- Estado.
- A *stack* onde se encontra.
- A imagem de instalação.
- A data de criação.
- O *node* onde se encontra.

3.2.5.2 Kubernetes

A Kubernetes Dashboard foi criada para fornecer aos utilizadores uma administração e gestão de *cluster* mais simples e interativa. Esta permite a implementação de aplicações no *cluster do Kubernetes*, solucionar problemas que possam aparecer nas aplicações e gerir os recursos do *cluster*. É possível ter uma visão geral de todos os recursos em funcionamento do *cluster* bem como modificar esses mesmos recursos. Também fornece

informação acerca do estado dos recursos no *Kubernetes* bem como erros que possam eventualmente aparecer. A implementação da *dashboard* não é feita juntamente com a instalação e configuração do *cluster*. É necessário correr o comando "*kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.3/aio/deploy/recommended.yaml*". O objetivo deste comando é fazer *download* do ficheiro YAML correspondeste a todos os elementos necessários à configuração da Kubernetes *dashboard* e aplicar a partir do comando "*kubectl apply -f*". Para aceder à *dashboard no browser*, é necessário, tal como aconteceu na configuração das ferramentas de *benchmarking no Kubernetes*, correr um comando de *port-forwarding*. O comando em questão é "*kubectl port-forward -n kubernetes-dashboard svc/kubernetes-dashboard 8080:443*" e permite a utilização da porta 8080 para o acesso do serviço da *dashboard* no computador pessoal. Com o endereço "*https://localhost:8080*" é possível entrar na página de *login*. A autenticação na *Kubernetes Dashboard* pode ser realizado de duas maneiras. A primeira é a partir de um ficheiro de configuração de credenciais. A segunda é a partir da criação de um *token* e aquela que foi escolhido. Para isso, o primeiro passo é a criação de um *service account no namespace "default"* a partir do comando "*kubectl create serviceaccount dashboard -n default*". Em segundo lugar, adicionar regras de ligação do *cluster ao service account* por meio do comando "*kubectl create clusterrolebinding dashboard-admin -n default –clusterrole=cluster-admin –serviceaccount=default:dashboard*". Por último, é necessário ter acesso ao *token* que foi criado com as regras para aceder à *dashboard*. Utilizando o comando "*kubectl get secrets*" é possível ver os *secrets* cujo são objetos do *Kubernetes* que são usados para armazenar dados sensíveis como nome de utilizador ou *passwords* encriptadas. Os *secrets* presentes no *cluster* podem ser verificados na figura 3.12. Para aceder ao *token* do *secret* referente à *dashboard* é necessário utilizar o comando "*kubectl describe secret 'nomedosecret'*" demonstrado na figura 3.13 e copiar o valor do *token* e colar na página de *login* da *Kubernetes Dashboard*.

```
root@kmaster-VirtualBox:/home/kmaster# kubectl get secrets
NAME                  TYPE                                  DATA   AGE
dashboard-token-qb5n2   kubernetes.io/service-account-token   3      14d
default-token-rfsgn     kubernetes.io/service-account-token   3      14d
root@kmaster-VirtualBox:/home/kmaster#
```

Figura 3.12: Lista de *secrets no cluster do Kubernetes*

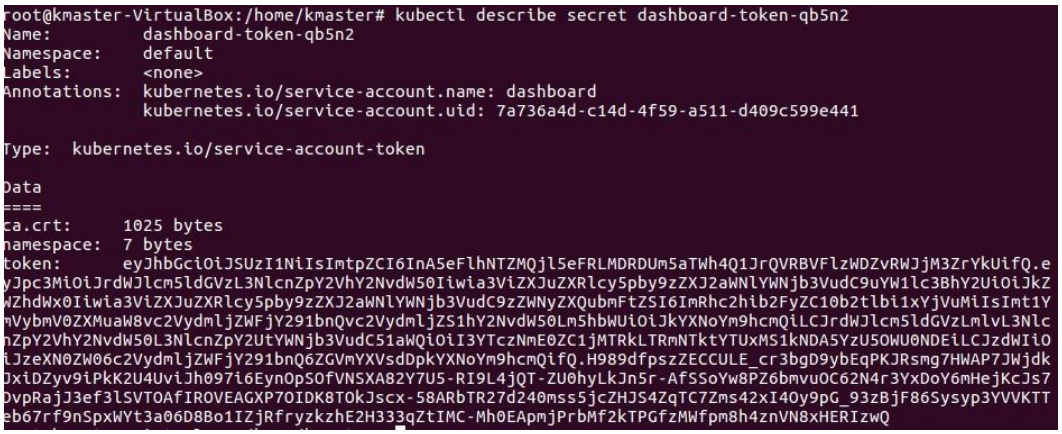

Figura 3.13: Valor do *token* referente à autenticação na *Kubernetes Dashboard*

Após concedido o acesso à *dashboard*, é apresentado uma série de opções relativas aos recursos do *cluster* bem o seu estado em tempo real presentes.

3.2.6 Instalação e configuração das imagens do NGINX

Com o objetivo de realizar testes acerca do comportamento de cada um dos *clusters*, é sempre fundamental ter uma ferramenta ou aplicação que ajude a concretizar este objetivo. Em vez de utilizar uma aplicação inteira, foi optado por um *web-server* bastante conhecido no mundo das tecnologias, o *NGINX*. Portanto, para realizar os testes, é necessário instalar o *NGINX nos clusters*. Existem duas formas de implementar o *NGINX* ou outra aplicação nas ferramentas de orquestração de *containers*. A primeira é a partir da linha de comandos e a alternativa é a partir dos GUI anteriormente referidos. Ao nível da instalação e configuração do *NGINX*, foi encontrada documentação oficial em cada um dos casos. No caso do ambiente do *Docker Swarm* encontram –se em [DOC20A] e [DER20]. No caso ambiente do *Kubernetes* encontra –se em [KRS20].

3.2.6.1 Docker Swarm

A instalação do *NGINX no Docker Swarm* é um processo simples. Apesar de ser simples, existem diversos fatores importantes relativos ao *NGINX* em ter em conta tais como:

- Nome do serviço, neste caso '*NGINXAPP*', onde se encontra.
- A porta, neste caso 8081, onde irá ser hospedado.
- O número de réplicas do serviço, neste caso apenas uma réplica.
- A imagem utilizada, neste caso foi utilizada a imagem da última versão do *NGINX*.

Com o comando "*docker service create –name NGINXAPP –publish 8081:80 –réplicas 1 nginx* " é possível criar o serviço do *NGINX* a partir da linha de comandos. A segunda forma de implementar é a partir do *Portainer.io*. Na página principal, existe a opção de criar um serviço e posteriormente é necessário preencher um formulário com o objetivo de proceder à instalação e configuração do *NGINX*.

Por último, sendo necessário verificar o bom funcionamento do *NGINX no cluster*, utilizando o comando "*https://localhost:8081/*" é possível verificar o *NGINX* ativo na porta 8081 como é demonstrado na figura 3.14.

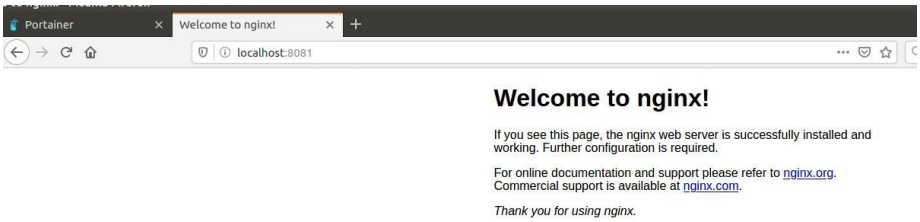

Figura 3.14: Página de verificação do *NGINX no Docker Swarm*.

3.2.6.2 Kubernetes

No caso do ambiente do *Kubernetes* o processo de instalação também é simples mas diferente no caso da linha de comandos. Tal como anteriormente referido, no *Kubernetes* é bastante comum utilizar ficheiros YAML para configurar vários serviços. E na instalação do *NGINX* não é diferente. No anexo B1 é apresentado o ficheiro YAML responsável pela instalação. Utilizando o ficheiro YAML, é necessário utilizar o comando " *kubectl apply -f 'diretóriodoficheiroyaml*"'. Para verificar o funcionamento, é necessário saber em porta está a correr o serviço. A partir do comando "*kubectl get endpoints -o wide*" é possível verificar o IP e a porta aos quais o serviço do *NGINX* está associado. Com esta informação, é possível aceder à página de verificação do *NGINX* presente na figura 3.15.

Figura 3.15: Página de verificação do *NGINX no Kubernetes*.

A segunda forma é implementar a partir da *Kubernetes Dashboard*. Na página principal, existe uma opção de criar um novo recurso. Tal como acontecia no *Portainer.io*, também é necessário preencher um formulário com objetivo de proceder à instalação e configuração do *NGINX*.

3.3 Conclusão

Neste capítulo foram descritas todas as instalações e configurações de todos os mecanismos utilizados nesta dissertação. Apesar de as ferramentas de GUI e *benchmarking* não terem qualquer influência nos testes, foi importante adquirir conhecimento sobre as mesmas devido à importância que podem desempenhar numa empresa. As ferramentas de GUI podem facilitar muito o trabalho de utilizadores com menos conhecimentos da CLI de cada plataforma. Por outro lado, as ferramentas de *benchmarking* são fundamentais nas atividades de suporte, com propósito na resolução de problemas em tempo real.

Capítulo 4

Análise dos Resultados Experimentais

4.1 Introdução

Neste capítulo são apresentados os resultados experimentais a partir dos pontos de medição definidos, o tempo de implementação e escalonamento de um web-server e ainda o tempo de tolerância a faltas dos containers. Os dados foram realizados com base em tempos médios com 3 casos em cada experiência. Através da análise dos dados, foi possível averiguar quais os pontos fortes e fracos de cada ferramenta de orquestração.

4.2 Implementação dos Testes

Com o objetivo de testar e comparar o *Docker Swarm e o Kubernetes*, optou-se por realizar três testes diferentes. Os testes são selecionados para avaliar três aspetos diferentes das ferramentas de orquestração:

1. Tempo necessário para a implementação de várias réplicas de um serviço *(NGINX) nos clusters* com vários cenários. São eles:
   - *(a)* Implementação de várias réplicas de um serviço *(NGINX)* com todos os *nodes* disponíveis.
   - *(b)* Implementação de várias réplicas de um serviço *(NGINX)* com um dos *workers* em baixo. .
   - (c) Implementação de várias réplicas de um serviço *(NGINX)* com todos os *workers* em baixo.
2. Tempo necessário para o escalonamento para cima e para baixo dos *containers*.
3. Tempo necessário para verificar a tolerância a faltas dos *containers*.

Para calcular o tempo dos testes foi utilizado um comando designado de "*watch*". Este comando é usado para executar qualquer comando arbitrário em intervalos de tempo regulares e exibe a saída do comando arbitrário na janela do terminal. Por exemplo, o utilizador pode usar o comando "*watch*" para monitorar o tempo de atividade do sistema ou o uso do disco [WCO19]. Nestes casos serve para monitorizar as listas atualizadas, de dois em dois segundos, dos serviços criados. Para medir o tempo desde o começo dos comandos de implementação de cada teste até a sua conclusão foi utilizado um cronómetro. O processo de implementação foi realizado 3 vezes para cada teste e ainda o cálculo da média dos tempos. Assim, foi possível comparar a diferença de tempo que uma ferramenta demora menos que a outra dependendo do teste realizado.

O primeiro teste consiste em implementar o serviço do *NGINX nos clusters* com várias réplicas. Quantas mais réplicas são implementadas maior o tempo que dura o processo de implementação. Optou –se por fazer testes com a criação de dez, trinta e cinquenta réplicas. Com o objetivo de testar o comportamento do *cluster* ainda foram realizados testes com os mesmos propósitos, anteriormente referidos, mas com um dos *workers* desligado de cada *cluster* e ainda outros testes sem *workers*.

O escalonamento de containers consiste em aumentar (escalonamento para cima) ou diminuir (escalonamento para baixo) o número de réplicas de um serviço. Para a realização deste processo, no *Docker Swarm* é utilizado o comando "*docker service scale nomedoserviço = númeroderéplicas*" e no *Kubernetes* o comando "*kubectl scale nomedodeployment –réplicas= númeroderéplicas*". Os testes realizados constituíam em implementar um serviço do *NGINX* com apenas uma réplica e escalar o serviço em dez, trinta e cinquenta réplicas. O processo oposto também foi realizado e o objetivo da experiência é medir o tempo de escalonamento em cada ferramenta. O processo de escalonamento foi apenas realizado pela linha de comandos, contudo é possível escalar os *containers* através das ferramentas de GUI.

O teste da tolerância a faltas é talvez o mais importante. Isto porque, é importante para os utilizadores escolherem a melhor ferramenta que lhes fornece rapidez e eficácia quando algo acontece de mal, ou seja, se um dos *workers* está em baixo, é expectável que todos os serviços atribuídos a esse *worker* continuem em funcionamento. Para isso, as ferramentas transportam os serviços do *worker* em baixo para um *worker* funcional, ou no caso do *Docker Swarm* para o *manager*. No *Kubernetes, o master* é apenas responsável pela gestão e administração do *cluster* e quando uma aplicação é implementada, apenas os *workers* são responsáveis pelo seu funcionamento enquanto no *Docker Swarm o manager* ajuda os *workers* a suportar este tipo de serviços. Este facto foi verificado nas experiências realizadas da implementação do serviço do*NGINX sem workers* em funcionamento. A tolerância a faltas foi testada num ambiente com o serviço do *NGINX* em funcionamento com dez, trinta e cinquenta réplicas.

4.3 Comparação Experimental das Ferramentas de Orquestração de Containers

Os resultados são apresentados abaixo nos histogramas, que mostram o número de resultados dentro de um determinado intervalo de tempo.

4.3.1 Implementação de um serviço NGINX no Kubernetes e no Docker Swarm

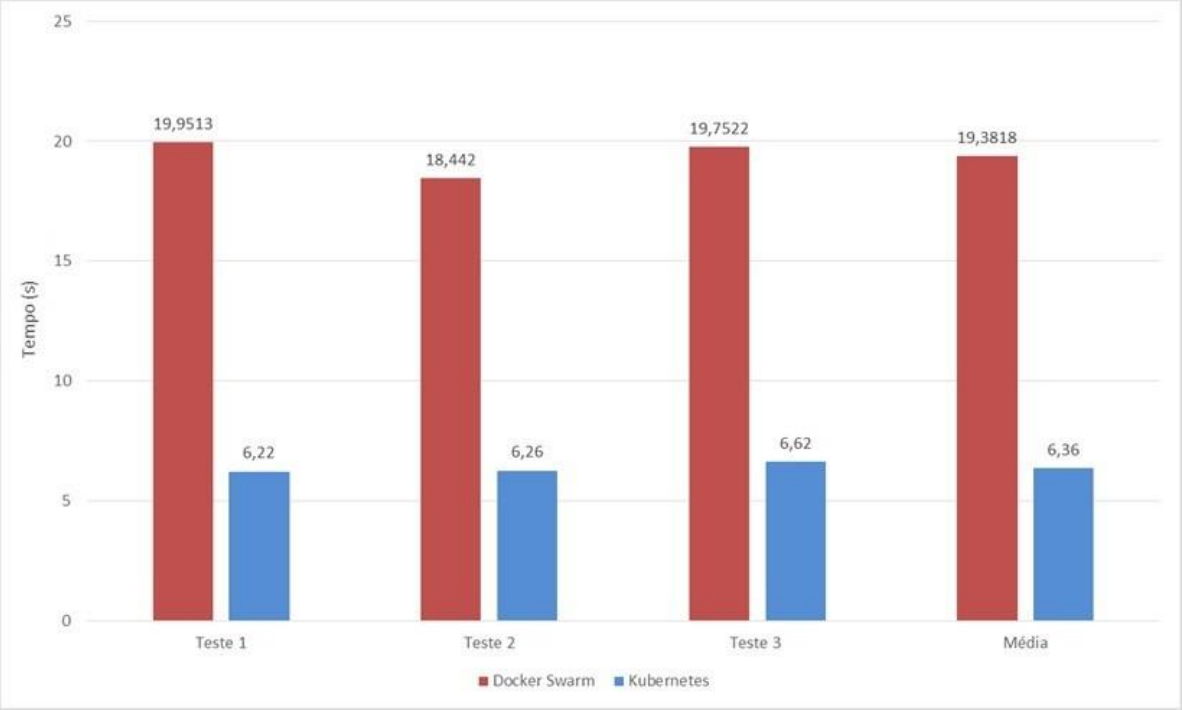

Figura 4.1: Tempo necessário à implementação de um serviço *NGINX* com 10 réplicas com todos os *nodes* disponíveis em cada ferramenta.

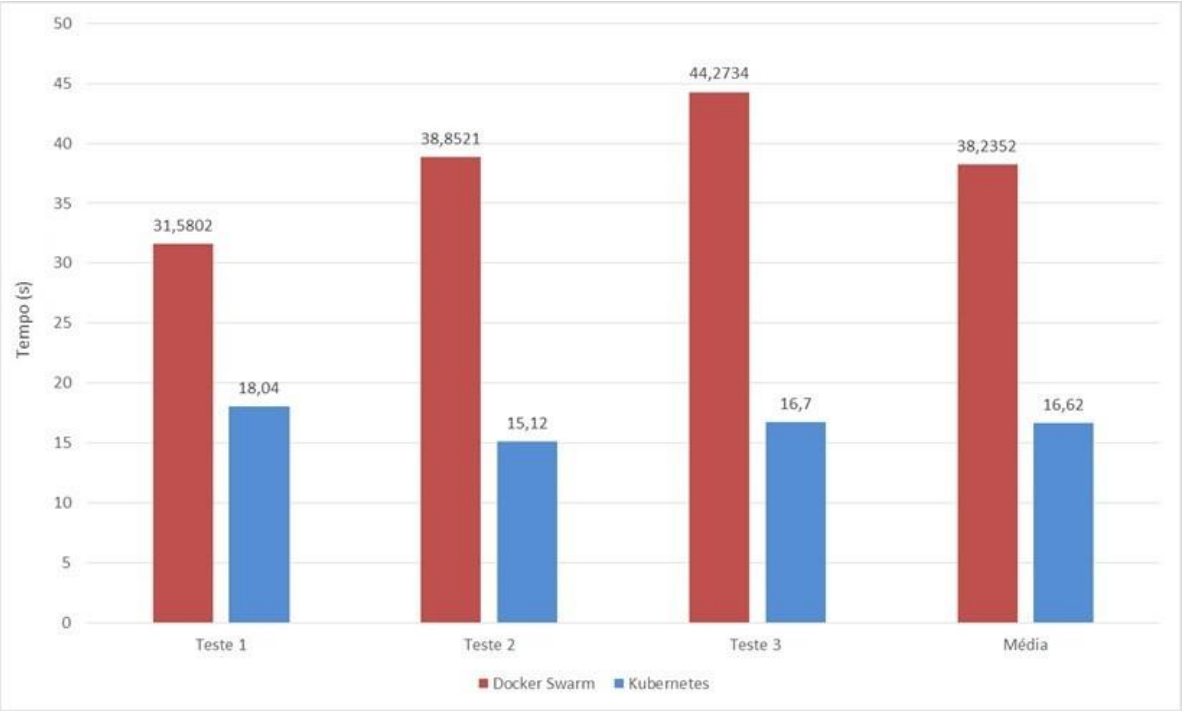

Figura 4.2: Tempo necessário à implementação de um serviço *NGINX* com 30 réplicas com todos os *nodes* disponíveis em cada ferramenta.

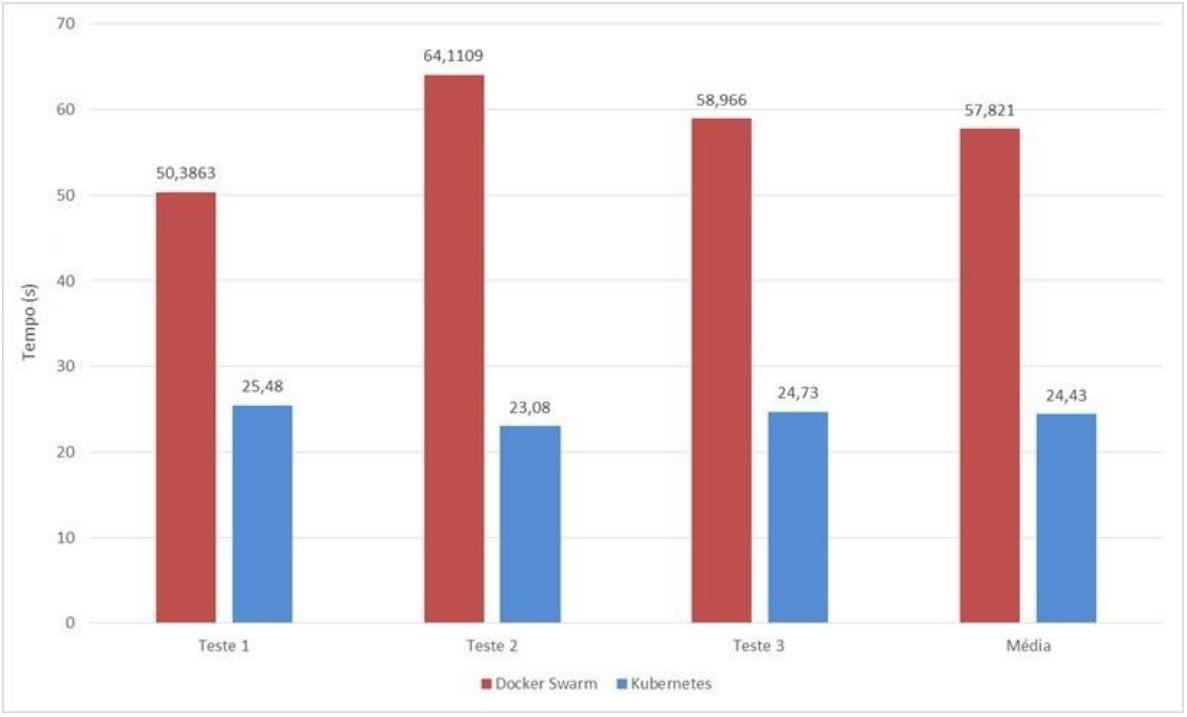

Figura 4.3: Tempo necessário à implementação de um serviço *NGINX* com 50 réplicas com todos os *nodes* disponíveis em cada ferramenta.

Cálculo das diferenças das médias relativamente à primeira experiência com todos os nodes disponíveis.	
Figura 4.1	13.0218 s
Figura 4.2	21.6152 s
Figura 4.3	33.391 s

Tabela 4.1: Tabela de diferença de médias relativamente à primeira experiência com todos os nodes disponíveis.

As figuras 4.1, 4.2 e 4.3 demonstram os resultados dos tempos de implementação com todos os *nodes* disponíveis em cada uma das ferramentas de orquestração. Verifica-se que em todos os histogramas o *Kubernetes* demora, em média, menos tempo que o *Docker Swarm* no processo de implementação. Na tabela 4.1 são apresentadas os valores da diferença média entre as ferramentas.

Após a análise dos dados verificou-se que:

- Quanto maior o número de réplicas maior o tempo do processo de implementação.
- Quanto maior o número de réplicas maior a diferença entre as médias das ferramentas.
- O *Kubernetes* é uma melhor solução no que toca a implementação com todos os *nodes* disponíveis.

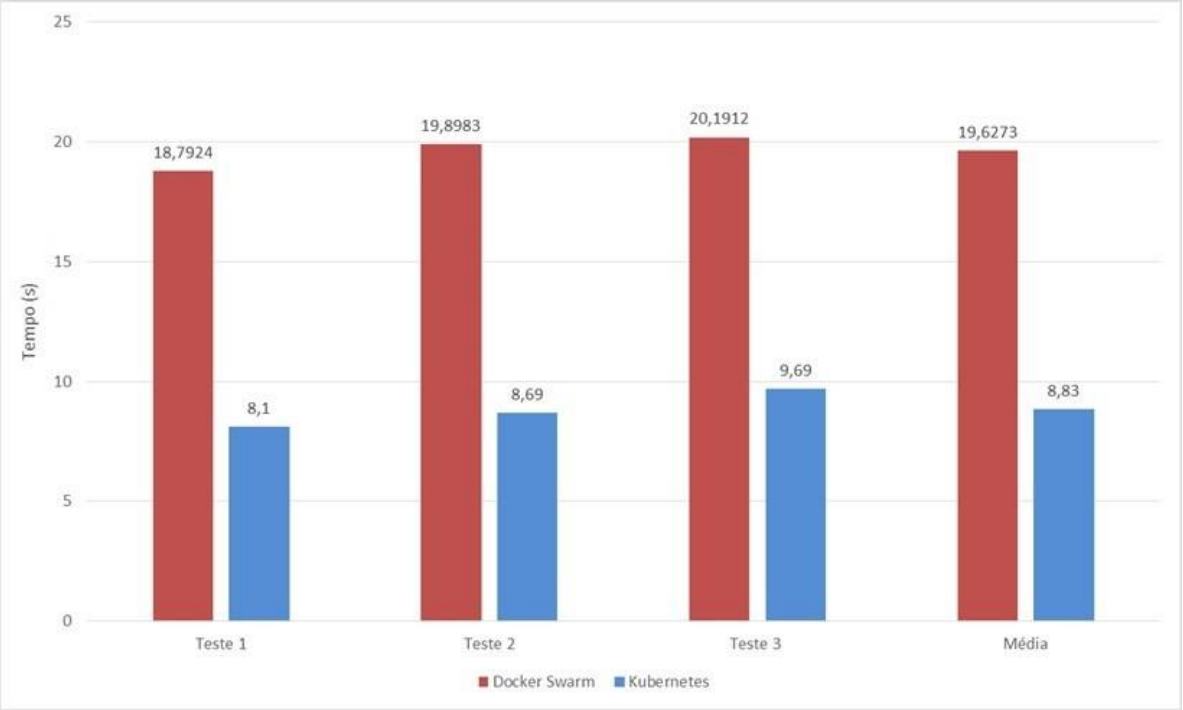

Figura 4.4: Tempo necessário à implementação de um serviço *NGINX* com 10 réplicas com um dos *workers* em baixo em cada ferramenta.

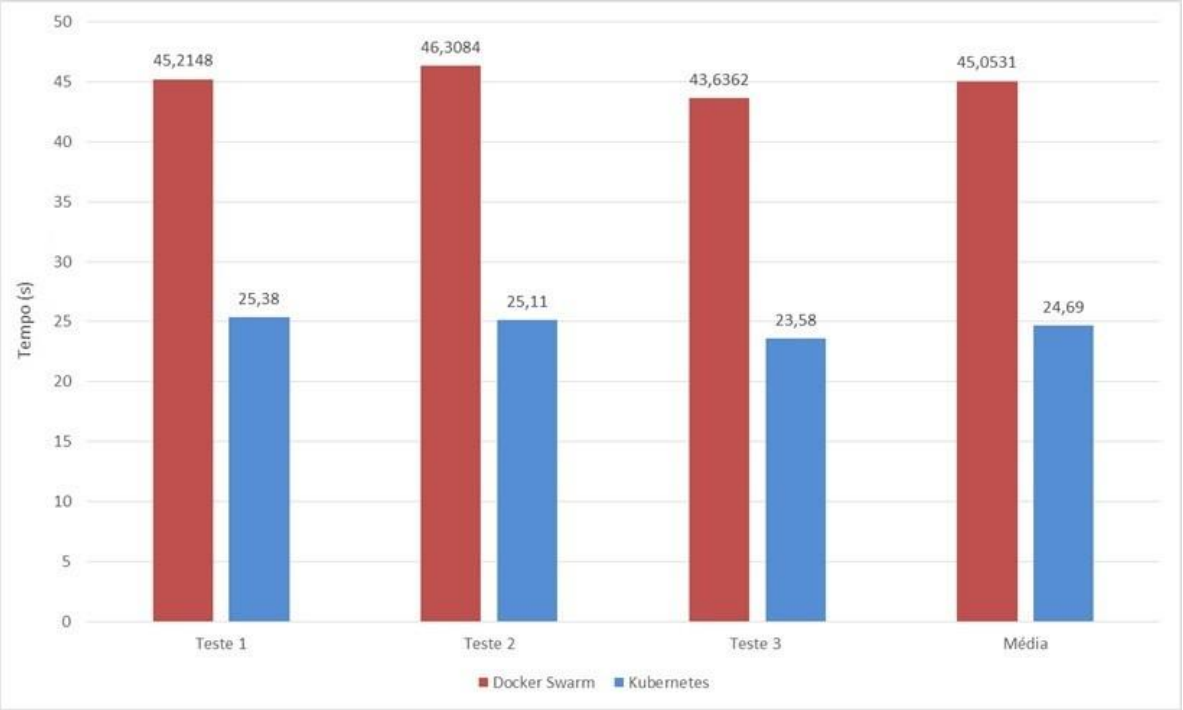

Figura 4.5: Tempo necessário à implementação de um serviço *NGINX*com 30 réplicas com um dos *workers* em baixo em cada ferramenta.

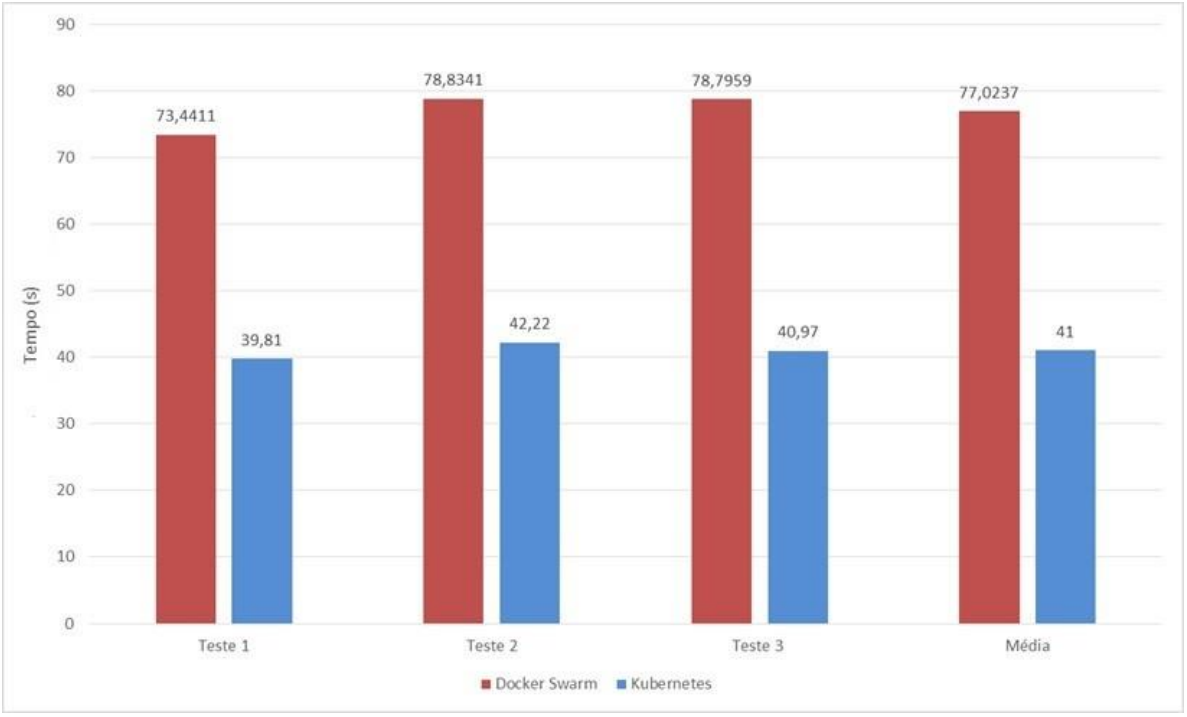

Figura 4.6: Tempo necessário à implementação de um serviço *NGINX* com 50 réplicas com um dos *workers* em baixo em cada ferramenta.

Cálculo das diferenças das médias relativamente à primeira experiência com um dos workers em baixo.	
Figura 4.4	10.7973 s
Figura 4.5	20.3631 s
Figura 4.6	36.0237 s

Tabela 4.2: Tabela de diferença de médias relativamente à primeira experiência com um dos workers em baixo.

As figuras 4.4, 4.5 e 4.6 demonstram os resultados dos tempos de implementação com um *worker* em baixo em cada uma das ferramentas de orquestração. Na tabela 4.2 são apresentadas os valores da diferença média entre as ferramentas.
Após a análise dos dados verificou-se que:

- Na figura 4.4 a diferença das médias dos testes realizados com dez réplicas é inferior à diferença calculada no caso anterior. Isto acontece devido ao facto de que o *Kubernetes*, com apenas um *worker*, fica mais lento na implementação e com base nos resultados demora aproximadamente dois segundos a mais enquanto o tempo no *Docker Swarm* permanece mais ou menos intacto.

- Na figura 4.5 a diferença das médias dos testes realizados com trinta réplicas é também inferior. Com base nos resultados, o *Kubernetes* demora aproximadamente oito segundos a mais enquanto o *Docker Swarm* demora aproximadamente sete segundos em relação aos resultados anteriores.

- No caso dos testes realizados com cinquenta réplicas, o panorama é diferente. De acordo com os resultados da figura 4.6, tanto no *Kubernetes* como no *Docker Swarm* demoraram aproximadamente vinte segundos a mais, o que é uma surpresa visto que ao analisar os resultados anteriores, era espectável o *Kubernetes* demorar mais que o *Docker Swarm* em relação ao teste anterior.
- Mesmo com os aumentos dos tempos médios, o *Kubernetes* é a melhor solução no que toca à implementação com um *worker* em baixo.

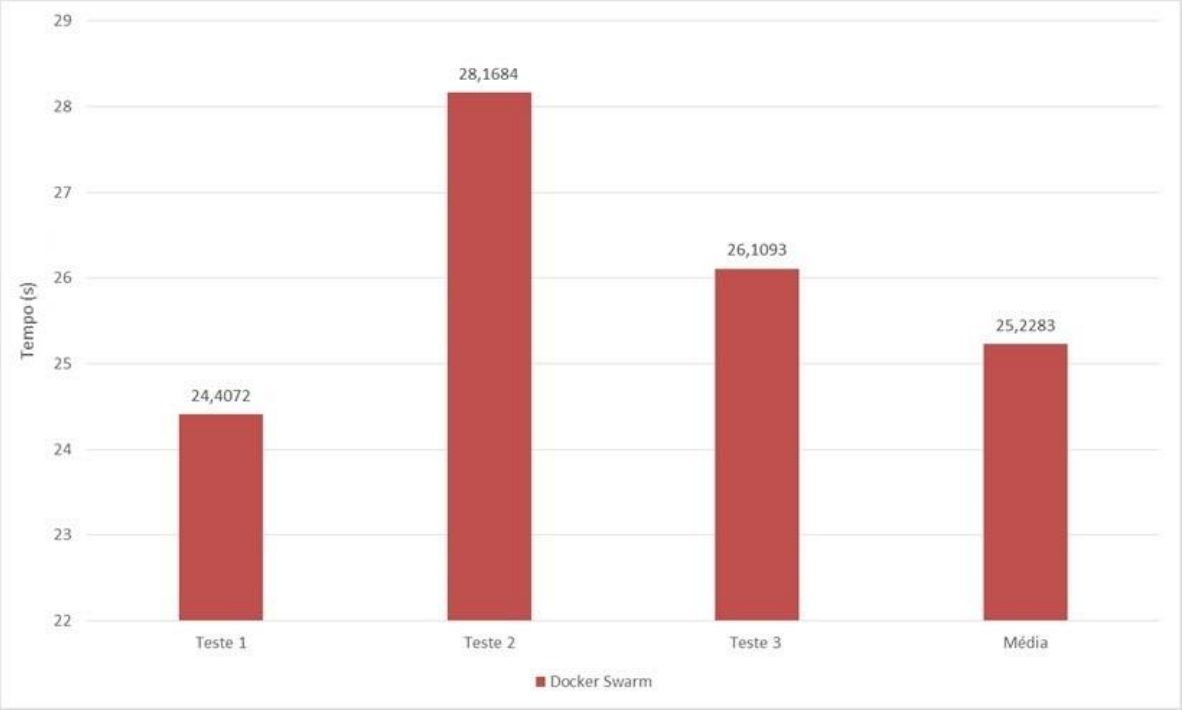

Figura 4.7: Tempo necessário à implementação de um serviço *NGINX* com 10 réplicas com os *workers* em baixo em cada ferramenta.

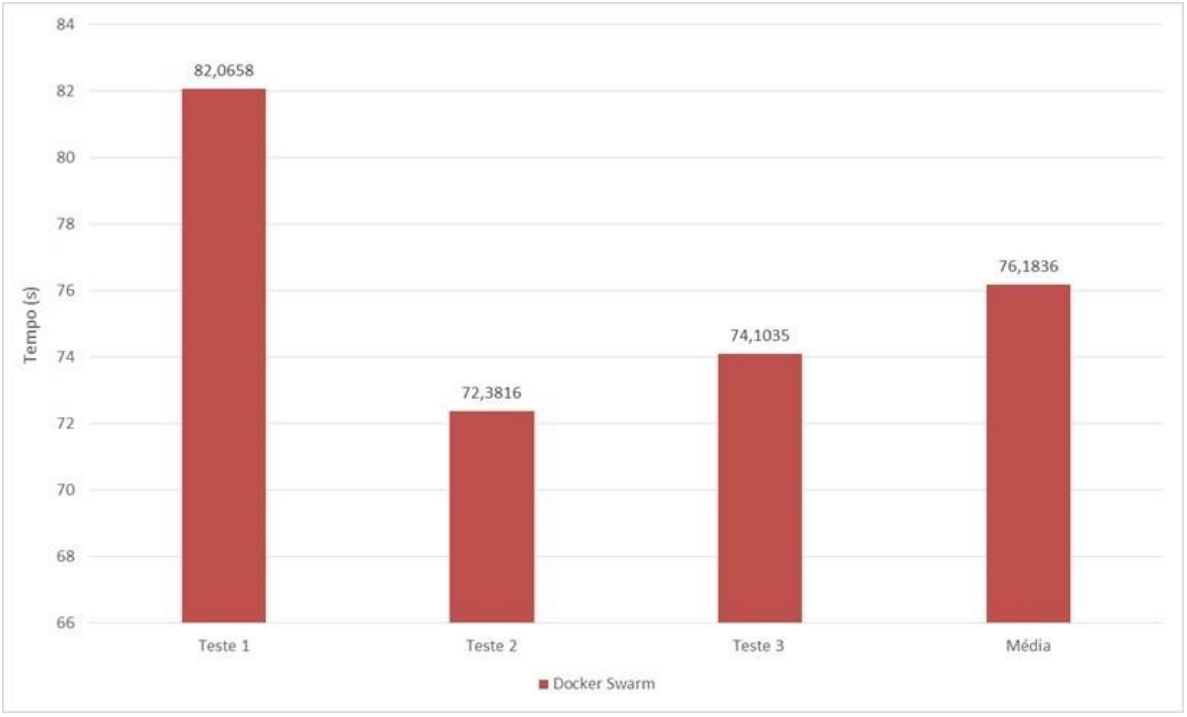

Figura 4.8: Tempo necessário à implementação de um serviço *NGINX* com 30 réplicas com os *workers* em baixo em cada ferramenta.

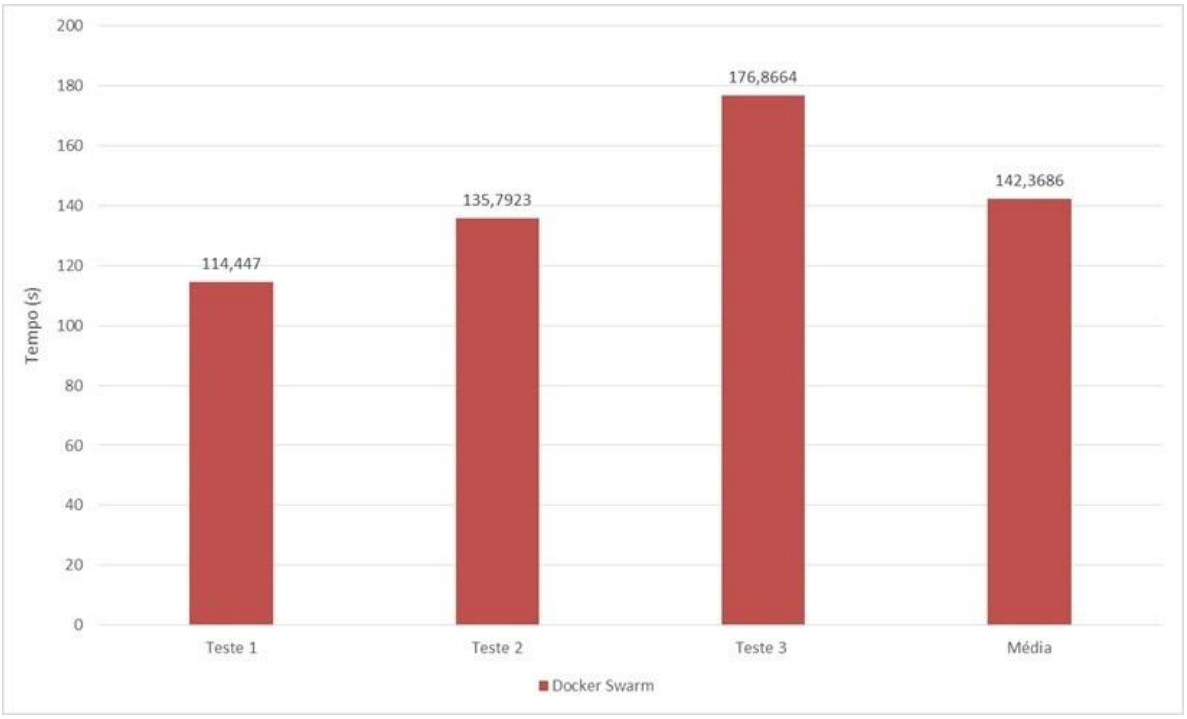

Figura 4.9: Tempo necessário à implementação de um serviço *NGINX* com 50 réplicas com os *workers* em baixo em cada ferramenta.

As figuras 4.7, 4.8 e 4.9 demonstram os resultados dos tempos de implementação com os *workers* em baixo. Como se verifica, apenas são apresentados os resultados no Docker Swarm. Isto deve-se ao facto, anteriormente explicado, que o *Kubernetes* sem *workers* não suporta as aplicações visto que o *master* apenas é responsável pela gestão das regras e

tarefas para os *workers* enquanto no *Docker Swarm o manager* também exerce funções de *worker*. Portanto pode-se concluir que no toca à implementação do *NGINX* sem *workers* em funcionamento, o *Docker Swarm* é melhor visto que o *Kubernetes* não apresenta soluções para resolver este tipo de situação.

4.3.2 Escalonamento das réplicas de um serviço *NGINX* no *Kubernetes* e no *Docker Swarm*

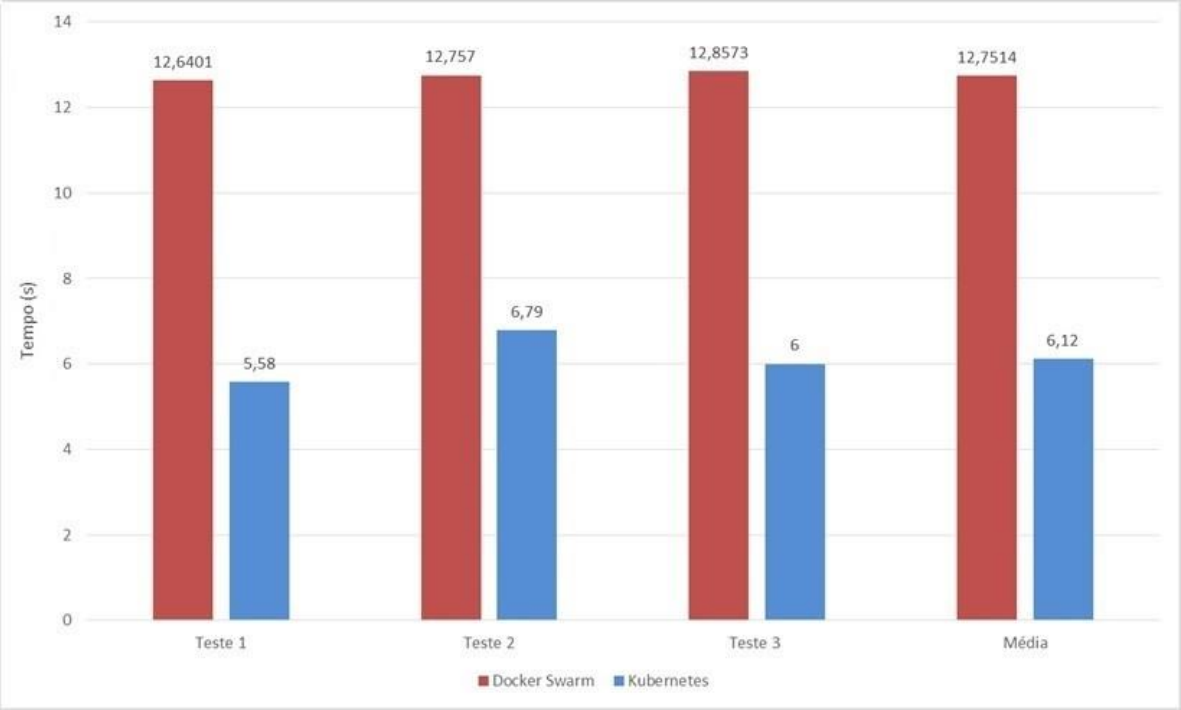

Figura 4.10: Tempo necessário ao escalonamento de 1 para 10 réplicas de um serviço *NGINX* em cada ferramenta.

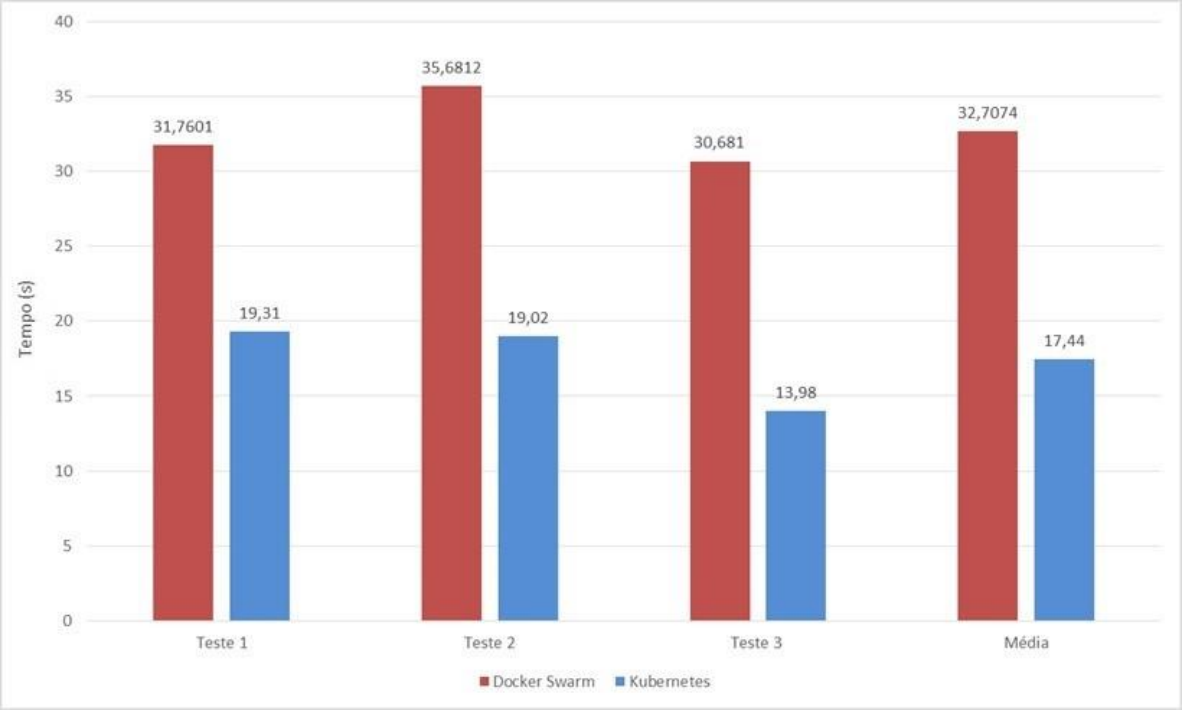

Figura 4.11: Tempo necessário ao escalonamento de 1 para 30 réplicas de um serviço *NGINX* em cada ferramenta.

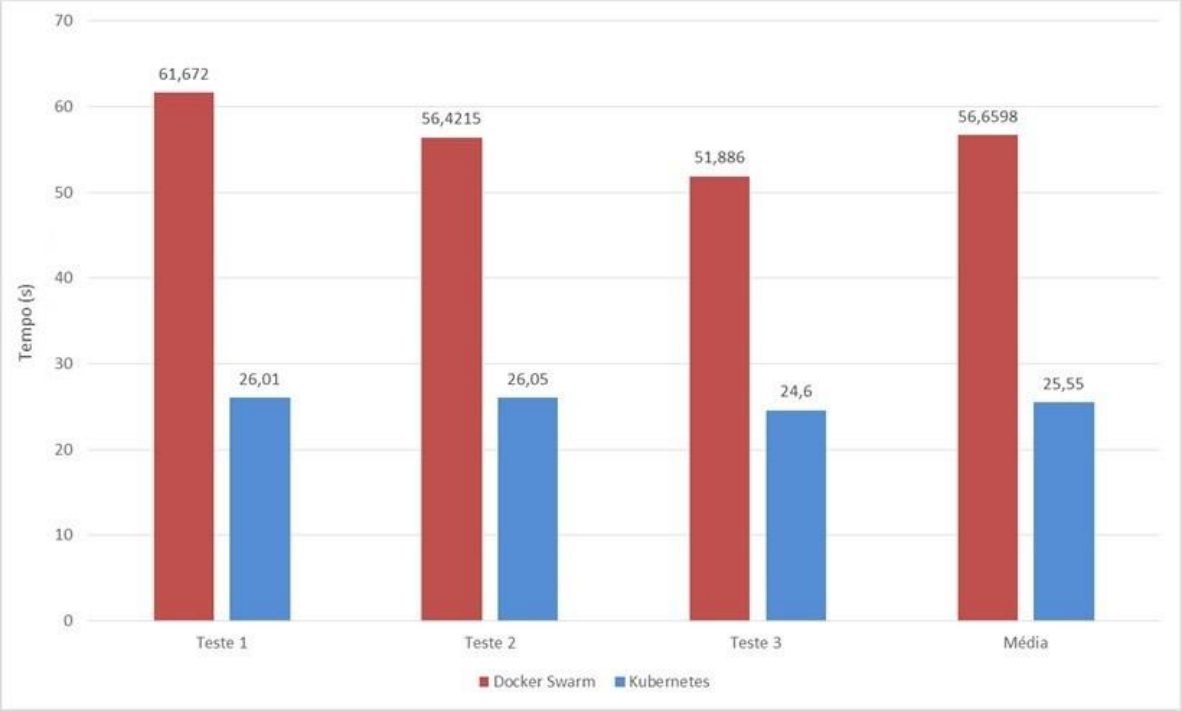

Figura 4.12: Tempo necessário ao escalonamento de 1 para 50 réplicas de um serviço *NGINX* em cada ferramenta.

As figuras 4.10, 4.11 e 4.12 demonstram os resultados dos tempos de escalonamento para cima em cada uma das ferramentas de orquestração. Na tabela 4.3 são apresentadas os valores da diferença média entre as ferramentas.
Após a análise dos dados verificou-se que:

Cálculo das diferenças das médias relativamente à segunda experiência com escalonamento para cima.	
Figura 4.10	6.6314 s
Figura 4.11	15.2674 s
Figura 4.12	31.1098 s

Tabela 4.3: Tabela de diferença de médias relativamente à segunda experiência com escalonamento para cima.

- Quanto maior o número de réplicas maior o tempo do processo de escalonamento, tal como aconteceu no primeiro teste.

- Quanto maior o número de réplicas maior a diferença entre as médias das ferramentas, mais uma vez, tal como aconteceu no primeiro teste.

- Em termos de escalonamento, o *Kubernetes* demorou menos tempo a escalonar o *NGINX* para cima, o que o torna a melhor ferramenta nesta situação.

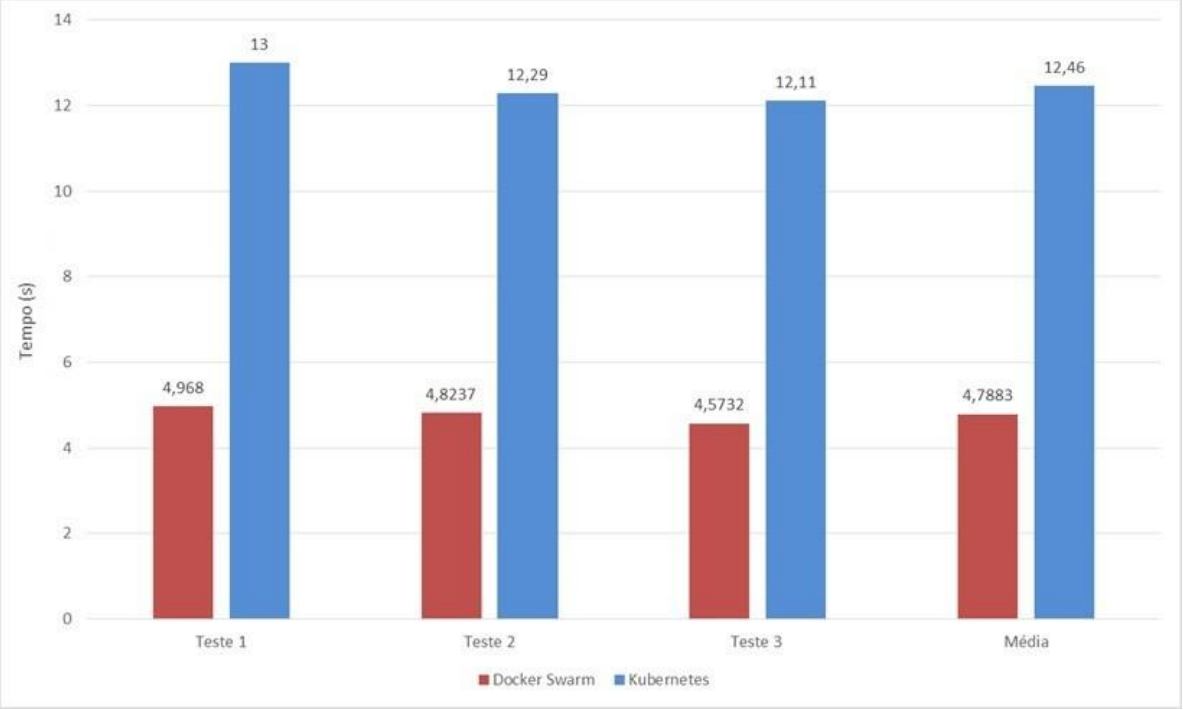

Figura 4.13: Tempo necessário ao escalonamento de 10 para 1 réplicas de um serviço *NGINX* em cada

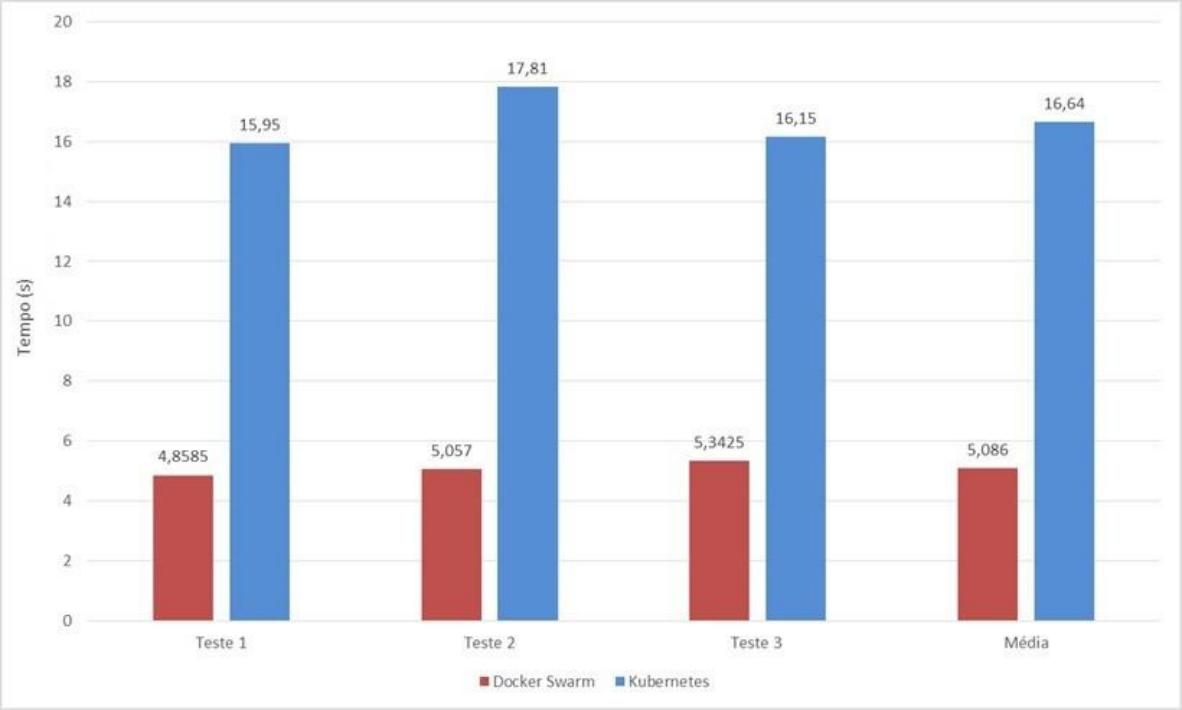

Figura 4.14: Tempo necessário ao escalonamento de 30 para 1 réplicas de um serviço *NGINX* em cada ferramenta.

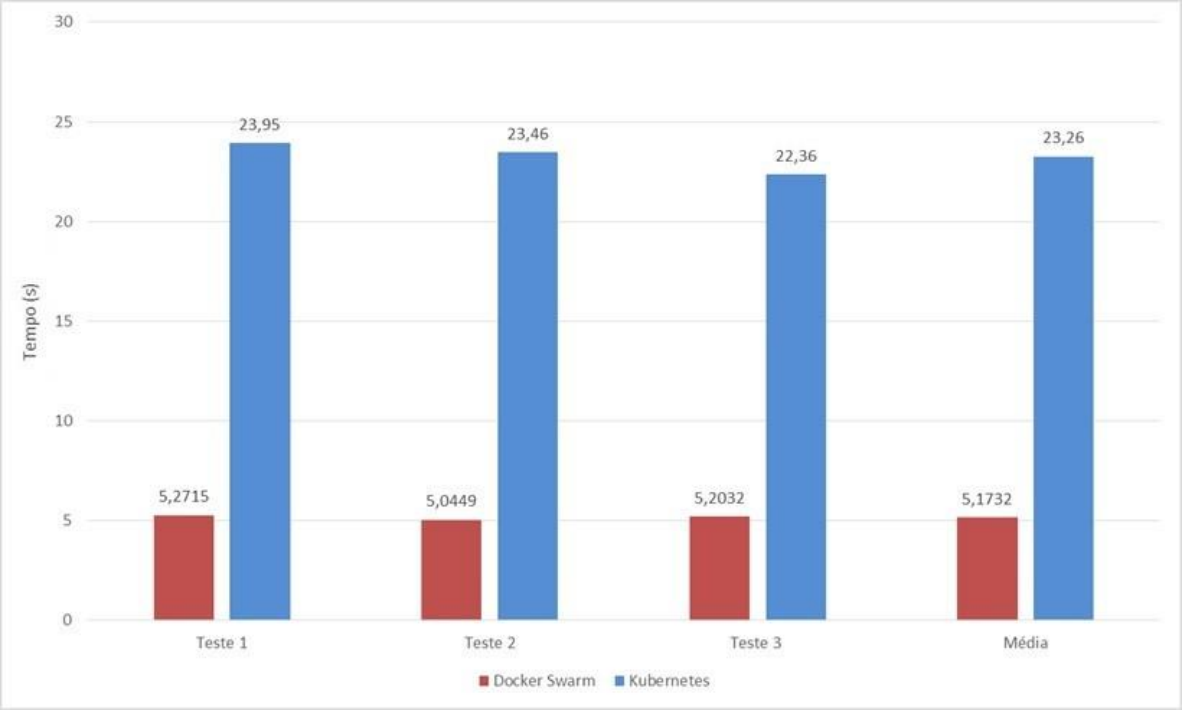

Figura 4.15: Tempo necessário ao escalonamento de 50 para 1 réplicas de um serviço *NGINX* em cada ferramenta.

As figuras 4.13, 4.14 e 4.15 demonstram os resultados dos tempos de escalonamento para baixo em cada uma das ferramentas de orquestração. Na tabela 4.4 são apresentadas os valores da diferença média entre as ferramentas. Após a análise dos dados verificou-se que:

Cálculo das diferenças das médias relativamente à segunda experiência com escalonamento para baixo.	
Figura 4.13	7.6717 s
Figura 4.14	11.5540 s
Figura 4.15	18.0868 s

Tabela 4.4: Tabela de diferença de médias relativamente à segunda experiência com escalonamento para baixo.

- Na figura 4.13 a diferença das médias dos testes realizados com o escalonamento para baixo é bastante semelhante à diferença calculada no caso anterior. Contudo, neste caso o *Docker Swarm* demora menos tempo a escalonar para baixa do o *Kubernetes*.

- A diferença das médias dos testes realizados com o escalonamento para baixo com trinta e cinquenta réplicas é inferior ao caso anterior do escalonamento para cima. Com base nos resultados das figuras 4.14 e 4.15, conclui-se que o tempo do *Docker Swarm* permanece o mesmo em todos os testes realizados, ou seja aproximadamente cinco segundos. No caso do *Kubernetes*, verifica-se quanto maior as réplicas mais lento é o escalonamento.

- No que toca ao escalonamento para baixo, conclui-se que o *Docker Swarm* é bastante mais rápido que o *Kubernetes*.

4.3.3 Tolerância a faltas de um serviço NGINX no Kubernetes e no Docker Swarm

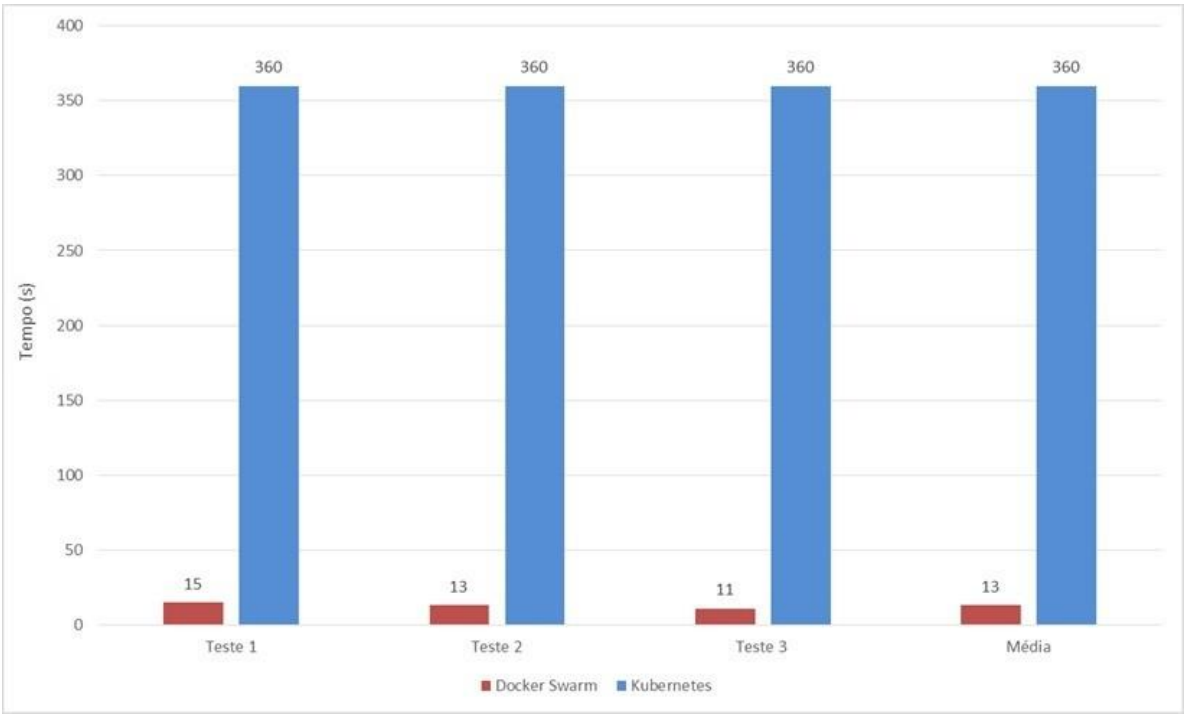

Figura 4.16: Tempo necessário em relação à tolerância a faltas com 10 réplicas de um serviço *NGINX* em cada ferramenta.

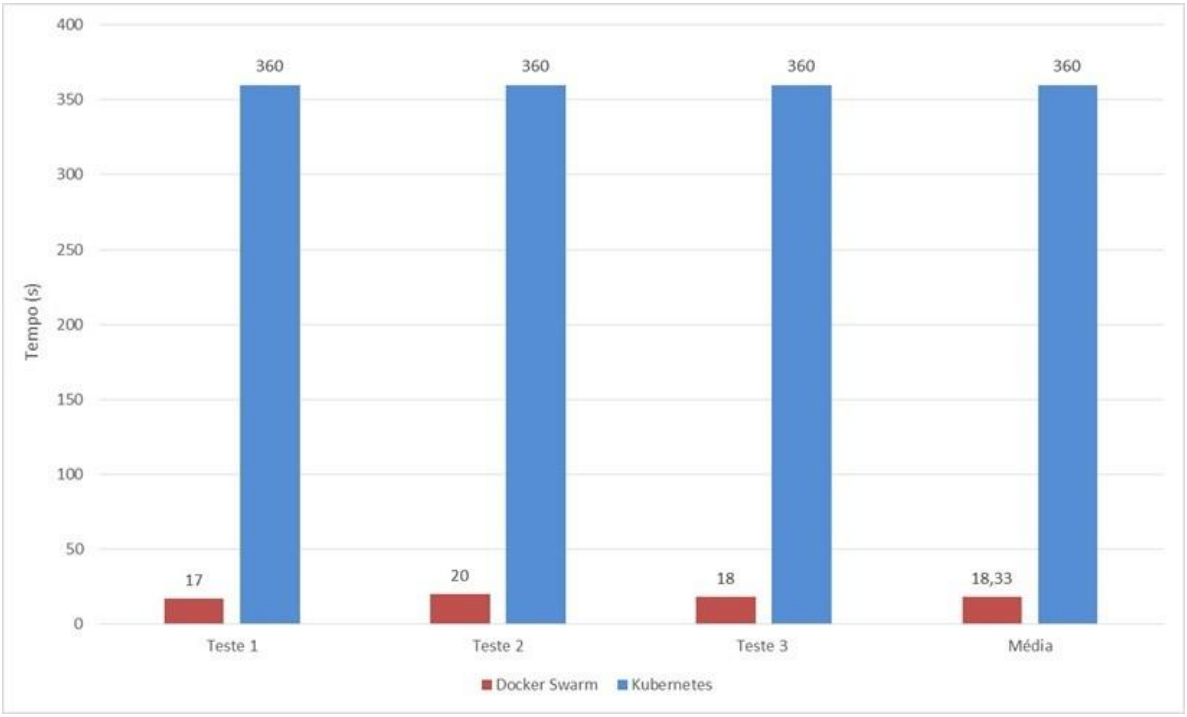

Figura 4.17: Tempo necessário em relação à tolerância a faltas com 30 réplicas de um serviço *NGINX* em cada ferramenta.

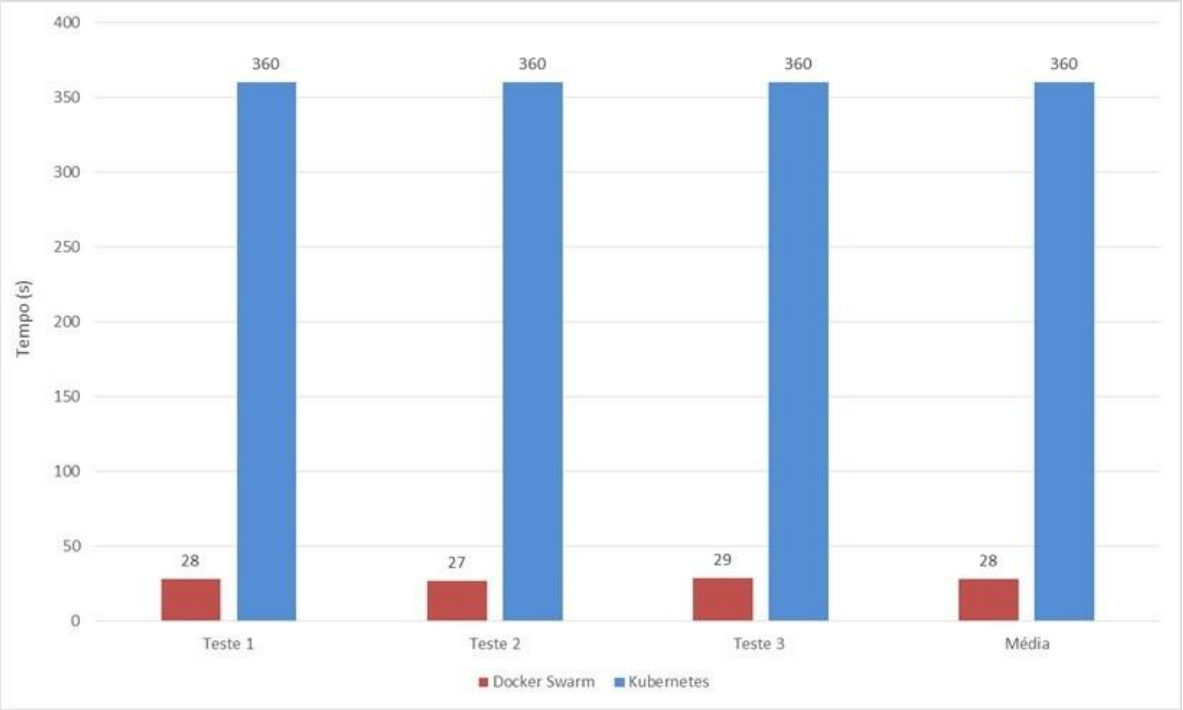

Figura 4.18: Tempo necessário em relação à tolerância a faltas com 50 réplicas de um serviço *NGINX* em cada ferramenta.

Cálculo das diferenças das médias relativamente à terceira experiência.	
Figura 4.16	$347\text{ s} \approx 5.7833\text{min}$
Figura 4.17	$341.671\text{ s} \approx 5.6945\text{ min}$
Figura 4.18	$332\text{ s} \approx 5.533\text{ min}$

Tabela 4.5: Tabela de diferença de médias relativamente à terceira experiência.

As figuras 4.16, 4.17 e 4.18 demonstram os resultados dos tempos relativos à tolerância a faltas em cada uma das ferramentas de orquestração. Na tabela 4.5 são apresentadas os valores da diferença média entre as ferramentas. Após a análise dos dados verificou-se que:

- A diferença entre os resultados presentes na tabela 4.5 é muito acentuada em relação a todas as experiências anteriores. Isto acontece devido ao facto de que o *Kubernetes*, por predefinição, no toca a tolerância a faltas, demora sempre aproximadamente seis minutos, enquanto o tempo no *Docker Swarm* varia de acordo com o número de réplicas, ou seja quanto maior o número de réplicas, maior o tempo de espera dos *containers* serem transportados para outro worker ou manager.

- No que toca à tolerância a faltas, o *Docker Swarm* é incrivelmente superior ao *Kubernetes* com uma diferença média de 5.67 minutos.

4.4 Discussão

O resultado das experiências realizadas mostrou claramente como as ferramentas de orquestração se comportam e qual foi o seu desempenho. Os tempos foram bastante limitados devido ao facto dos *cluster* serem instalados em máquinas virtuais por razões anteriormente referidas. De acordo a análise teórica realizada, era espectável que o *Kubernetes* fosse significativamente mais rápido em todos os testes, mas o comportamento que foi observado mostrou que o *Kubernetes*, pelo menos em pequena escala, não é tão estável quanto o *Docker Swarm*. É importante referir que o comportamento do *Kubernetes* durante os testes foi invulgar, no sentido em que um *container*, quando era suposto ser eliminado através do comando correto, geralmente levava alguns segundos para ser eliminado completamente após a execução desse mesmo comando. Esse comportamento difere significativamente no *Docker Swarm*, que em 100% dos casos eliminou um ou mais *containers* assim que o comando é executado corretamente. De seguida, é apresentado uma tabela acerca da visão geral dos resultados dos pontos de medição anteriormente definidos.

Visão Geral dos Resultados dos Pontos de Medição		
Pontos de Medição	Docker Swarm	Kubernetes
Implementação do NGINX com todos os nodes disponíveis	-	Melhor
Implementação do NGINX com um dos workers em baixo	-	Melhor
Implementação do NGINX sem workers disponíveis	Melhor	-
Escalonamento para cima dos containers do NGINX	-	Melhor
Escalonamento para baixo dos containers do NGINX	Melhor	-
Tolerância a faltas dos containers do NGINX	Melhor	-
Dificuldade na instalação e configuração das ferramentas	Mais Fácil	-
Dificuldade de utilização das ferramentas	Mais Fácil	-

Tabela 4.6: Tabela da visão geral dos resultados dos pontos de medição definidos.

A partir da análise da tabela, é possível verificar que o *Docker Swarm* apresenta mais pontos positivos do que o *Kubernetes* a partir dos pontos de medição definidos. Contudo, existe uma propriedade no Kubernetes, anteriormente referida, o processo de escalonamento automático que na experiências da tolerância a faltas, os serviços do *NGINX* implementados no *Kubernetes* não precisavam ser reiniciados antes de cada caso (dez, trinta e cinquenta réplicas), pois a propriedade de escalonamento foi incorporada desde o início da experiência no *cluster do Kubernetes*. Apesar disso, verifica –se que, em termos de tolerância a faltas, o*Docker Swarm* é bastante superior ao *Kubernetes*. Por fim, a partir de todas as experiências realizadas anteriormente, no que toca à complexidade ou dificuldade de instalação, configuração e utilização dos serviços de cada ferramenta, o *Docker Swarm* é claramente a melhor solução.

4.5 Conclusão

Neste capítulo foi descrito todo o processo dos testes realizados pelas ferramentas de orquestração bem como resultado de cada experiência. Foi bastante importante realizar todas as experiências com o objetivo de analisar cada ferramenta de orquestração ao pormenor para estabelecer uma comparação geral entre as mesmas. Os pontos de medição foram definidos a partir de uma análise prévia pessoal acerca dos componentes mais importantes a avaliar tanto no *Kubernetes como no Docker Swarm*.

Capítulo 5

Conclusões

5.1 Principais Conclusões

O estudo da tecnologia de virtualização do *Docker* permitiu concluir que a virtualização ao nível do sistema operativo está a crescer cada vez mais no mercado atual. As máquinas virtuais e os containers demonstram um desempenho inferior às máquinas nativas devido à presença dos *overheads* de virtualização. Com base nas experiências realizadas, foi possível examinar qual das ferramentas de orquestração mais populares do *Docker* apresenta melhor desempenho de acordo com os pontos de medição definidos e também a dificuldade de instalação e configuração bem como a dificuldade de de utilização ao longo do trabalho conducente a esta dissertação. Os resultados das experiências mostraram todos os pontos positivos tanto do *Kubernetes como do Docker Swarm*. É importante referir que existem outros pontos de comparação de comparação que não foram explorados nesta dissertação devido à grande complexidade destas ferramentas. Em suma, a resposta à questão de qual das ferramentas de orquestração de containers é a melhor, é que nenhuma é melhor que a outra visto que o *Docker Swarm* lida com certos aspetos de uma forma melhor, enquanto o *Kubernetes* é bastante vantajoso noutros contextos. Com base na análise do desempenhos das ferramentas, conclui–se que o *Docker Swarm* é mais adequado para ambientes de *clusters* menores, com uma administração e gestão dos recursos mais facilitada. Por outro lado, o *Kubernetes* é a escolha ideal para ambientes de produção com mais recursos onde é mais rápido implementar vários serviços presentes no mercado atual. Por fim, é importante referir que, em termos de aprendizagem, a melhor ferramenta é o *Docker Swarm* porque apresenta bastante mais documentação oficial que o *Kubernetes*.

5.2 Sugestões para Trabalho Futuro

Os testes realizados nesta dissertação com o objetivo de comparar as duas ferramentas de orquestração mais populares do *Docker* foram realizados em pequena escala. Como sugestão de trabalho futuro, é essencial expandir o número de *nodes* em cada *cluster*, de modo a permitir avaliar o impacto de uma maior carga de trabalho nas ferramentas de orquestração. Dando o exemplo da experiência da tolerância a faltas, foi utilizado um método específico usado para desligar os *nodes*. Em cada caso, a máquina virtual foi desligada manualmente. Contudo o resultado dos tempos podem eventualmente mudar se a falha for simulada de outra maneira, por exemplo, por falta de energia ou a utilização do comando "*sudo init 0*" para desligar o computador pela CLI. Outro grande fator, é realizar este tipo de projetos em máquinas físicas, o que não foi possível devido ao aparecimento da

doença COVID-19. Por fim, é importante enfatizar que essas ferramentas possuem vários recursos que não foram incluídos ou testados neste estudo e que podem afetar a recomendação para condições e áreas específicas por parte das empresas.

Bibliografia

[app] Getting started with app-v for windows 10 [online]. Available from: https://docs.microsoft.com/en-us/windows/application-management/ app-v/appv-getting-started[Acedido pela última vez a 5 Junho 2020]. 12

[ARQ19] Containers vs. virtual machines [online]. 2019. Available from: https: //docs.microsoft.com/en-us/virtualization/windowscontainers/ about/containers-vs-vm[Acedido pela última vez a 5 Junho 2020]. xiii, 12, 13, 14

[BHO18] Sandip Bhowmik. How to configure docker swarm with multiple docker nodes on ubuntu 18.04 [online]. 2018. Available from: https://linuxconfig.org/ how-to-configure-docker-swarm-with-multiple-docker-nodes-on-ubuntu-18-04 [Acedido pela última vez a 19 Junho 2020]. 29

[BOS19] Michael Bose. How to install kubernetes on ubuntu [online]. 2019. Available from: https://www.nakivo.com/blog/install-kubernetes-ubuntu/ [Acedido pela última vez a 19 Junho 2020]. 29

[BRI12] How-to: Virtualbox networking part two - nat and bridged [online]. 2012. Available from: https://catlingmindswipe.blogspot.com/2012/06/ how-to-virtualbox-networking-part-two.html[Acedido pela última vez a 10 Junho 2020]. 27

[Car] Alexandre Carissimi. Virtualização: da teoria a soluções. pages 174–177. 6

[CLO18] Alibaba Cloud. How to install and deploy kubernetes on ubuntu 16.04 [online]. 2018. Available from: https://medium.com/@Alibaba_Cloud/ how-to-install-and-deploy-kubernetes-on-ubuntu-16-04-6769fd1646db [Acedido pela última vez a 19 Junho 2020]. 29

[COH18] Idan Cohen. How to setup a docker swarm cluster on ubuntu 16.04 vps or dedicated server [online]. 2018. Available from: https://hostadvice.com/ how-to/how-to-setup-a-docker-swarm-cluster-on-ubuntu-16-04/[Acedido pela última vez a 19 Junho 2020]. 29

[COM19] Computer Hope. Gui [online]. 2019. Available from: https://www. computerhope.com/jargon/g/gui.htm[Acedido pela última vez a 8 Junho 2020]. 25

[con] What is a container [online]. Available from: https://www.docker.com/ resources/what-container[Acedido pela última vez a 5 Junho 2020]. 9

[COR20] Prometheus operator [online]. 2020. Available from: https://coreos.com/ operators/prometheus/docs/latest/user-guides/getting-started.html [Acedido pela última vez a 27 Junho 2020]. 35

[DER20] docker service create [online]. 2020. Available from: https:// docs.docker.com/engine/reference/commandline/service_create/ [Acedido pela última vez a 2 Agosto 2020]. 40

[DES20] Ankit Deshpande. [diy] how to set up prometheus and ingress on kubernetes [online]. 2020. Available from: https://blog.gojekengineering.com/ diy-how-to-set-up-prometheus-and-ingress-on-kubernetes-d395248e2ba [Acedido pela última vez a 20 Junho 2020]. 33

[DM06] D McCrory D Marshall, WA Reynolds. *Advanced Server Virtualization VMware and Microsoft Platforms in the Virtual Data Center*. ISBN: 978-042-91-3253-7. Auerbach Publications, 2006. Available from: https://www.taylorfrancis.com/books/9780429132537. 5

[DOC20A] Deploy services to a swarm [online]. 2020. Available from: https://docs.docker.com/engine/swarm/services/[Acedido pela última vez a 2 Agosto 2020]. 40

[DOC20B] Docker. Docker overview [online]. 2020. Available from: https://docs.docker.com/get-started/overview/#docker-architecture [Acedido pela última vez a 6 Junho 2020]. xiii, 1, 9, 10

[DSA20] How nodes work [online]. 2020. Available from: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/ [Acedido pela última vez a 5 Junho 2020]. xiii, 18

[EDD02] Jorge J. Moré Elizabeth D. Dolan. Benchmarking optimization software with performance profiles. 2002. Available from: https://link.springer.com/article/10.1007/s101070100263[Acedido pela última vez a 8 Junho 2020]. 25

[FLE18] Flexera. Rightscale 2018 state of the cloud report. pages 26–27, 2018. 22

[FLE19] Flexera. Rightscale 2019 state of the cloud report. pages 27–28, 2019. 1, 22

[GIO19] Giorgio Bonuccelli. Application virtualization | what is it and why does your organization need it? [online]. 2019. Available from: https://www.parallels.com/blogs/ras/application-virtualization/[Acedido pela última vez a 5 Junho 2020]. 12

[GRA17] Prometheus monitoring with grafana [online]. 2017. Available from: https://logz.io/blog/prometheus-monitoring/[Acedido pela última vez a 8 Junho 2020]. 26

[GRA20] What is grafana? [online]. 2020. Available from: https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/[Acedido pela última vez a 8 Junho 2020]. 26

[HAR18] Harsh Binani. Kubernetes vs docker swarm—a comprehensive comparison [online]. 2018. Available from: https://hackernoon.com/kubernetes-vs-docker-swarm-a-comprehensive-comparison-73058543771e [Acedido pela última vez a 5 Junho 2020]. xiii, 16

[HSI20] The package manager for kubernetes [online]. 2020. Available from: https://helm.sh/[Acedido pela última vez a 27 Junho 2020]. 36

[HVE20] Helm releases [online]. 2020. Available from: https://github.com/helm/helm/releases[Acedido pela última vez a 27 Junho 2020]. 36

[JAS20] Jason Baker (Red Hat). What is docker? [online]. 2020. Available from: https://opensource.com/resources/what-docker[Acedido pela última vez a 6 Junho 2020]. 9

[KDA20] Web ui (dashboard) [online]. 2020. Available from: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ [Acedido pela última vez a 8 Junho 2020]. 25

[KES20] Selvaraj Kesavan. Cloud computing-virtualization and containers contents - virtualization. page 11, 2020. 6

[KH11] Geoffrey C. Fox Kai Hwang, Jack Dongarra. Distributed and cloud computing: From parallel processing to the internet of things, 1ST edition, 2011. 6, 7

[KPR17] Kubernetes monitoring with prometheus in 15 minutes [online]. 2017. Available from: https://itnext.io/kubernetes-monitoring-with-prometheus-in-15-minutes-8e54d1de2e13 [Acedido pela última vez a 8 Junho 2020]. 33

[KRS20] Run a stateless application using a deployment [online]. 2020. Available from: https://kubernetes.io/docs/tasks/run-application/ run-stateless-application-deployment/ [Acedido pela última vez a 2 Agosto 2020]. 40

[kub] Kubernetes [online]. Available from: https://kubernetes.io/pt/[Acedido pela última vez a19 Janeiro 2021]. 1

[KUB20A] Web ui (dashboard) [online]. 2020. Available from: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ [Acedido pela última vez a 31 Junho 2020]. 38

[KUB20B] Kubernetes. Using coredns for service discovery [online]. 2020. Available from: https://kubernetes.io/docs/tasks/administer-cluster/ coredns/ [Acedido pela última vez a 20 Junho 2020]. 32

[MEN19] Nicolas Menant. Cis and kubernetes - part 1: Install kubernetes and calico [online]. 2019. Available from: https://devcentral.f5.com/s/articles/ CIS-and-Kubernetes-Part-1-Install-Kubernetes-and-Calico [Acedido pela última vez a 19 Junho 2020]. 29

[MIK16] Mike Coleman. Containers and vms together [online]. 2016. Available from: https://www.docker.com/blog/containers-and-vms-together/ [Acedido pela última vez a 5 Junho 2020]. 15

[MON19] Mona Mangat. Kubernetes vs mesos: Detailed comparison [online]. 2019. Available from: https://phoenixnap.com/blog/kubernetes-vs-mesos [Acedido pela última vez a 5 Junho 2020]. 15, 18

[ngi] What is nginx? [online]. Available from: https://www.nginx.com/resources/glossary/nginx/[Acedido pela última vez a 8 Junho 2020]. 25

[PLA17] Platform9. Kubernetes vs docker swarm [online]. 2017. Available from: https://platform9.com/blog/kubernetes-docker-swarm-compared/ [Acedido pela última vez a 5 Junho 2020]. 20

[POR20A] How simple is it to deploy portainer? [online]. 2020. Available from: https://www.portainer.io/installation/[Acedido pela última vez a 31 Junho 2020]. 38

[POR20B] Quick start [online]. 2020. Available from: https://www.portainer.io/documentation/quick-start/[Acedido pela última vez a 8 Junho 2020]. 25

[PRO20] What is prometheus? [online]. 2020. Available from: https://prometheus.io/docs/introduction/overview/#architecture[Acedido pela última vez a 8 Junho 2020]. 26

[RAN18] Ranvir Singh. Vmware vs virtualbox [online]. 2018. Available from: https://linuxhint.com/vmware_vs_virtualbox/[Acedido pela última vez a 6 Junho 2020]. 24

[REF20] Kubernetes cluster - prometheus [online]. 2020. Available from: https://grafana.com/grafana/dashboards/9797[Acedido pela última vez a 28 Junho 2020]. 36

[SC06] Michael Jeronimo Sean Campbell. An introduction to virtualization. pages 5–7, 2006. 5

[SN05a] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. page 6, 2005. 8

[SN05b] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. page 8, 2005. 8

[SN05c] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. page 17, 2005. 9

[SN05d] Tzi-cker Chiueh Susanta Nanda. A survey on virtualization technologies. page 23, 2005. 11

[SPE18] SPEC INDIA. Kubernetes vs. docker swarm: A complete comparison guide [online]. 2018. Available from: https://hackernoon.com/kubernetes-vs-docker-swarm-a-complete-comparison-guide-15ba3ac6f750 [Acedido pela última vez a 5 Junho 2020]. 16, 17, 19, 20

[Swm] Swarm mode overview [online]. Available from: https://docs.docker.com/engine/swarm/[Acedido pela última vez a 19 Janeiro 2021]. 1

[SWP17] Swarmprom - prometheus monitoring for docker swarm [online]. 2017. Available from: https://www.weave.works/blog/swarmprom-prometheus-monitoring-for-docker-swarm [Acedido pela última vez a 20 Junho 2020]. 33

[VAS12] Deepak.K.Damodaran Vasudevan.M.S, Biju.R.Mohan. Performance measur-

ing and comparison of virtualbox and vmware. page 43, 2012. 23

[VMw] Vmware [online]. Available from: https://www.vmware.com/[Acedido pela última vez a 5 Junho 2020]. 6

[WCO19] Linux watch command [online]. 2019. Available from: https://linuxize.com/post/linux-watch-command/[Acedido pela última vez a 3 Agosto 2020]. 43

[WVS18] Docker windows vs. linux [online]. 2018. Available from: https://medium.com/@javier.ramos1/docker-windows-vs-linux-1bb26d8090b3 [Acedido pela última vez a 10 Junho 2020]. 27

[Xen] Xenapp [online]. Available from: https://www.citrix.com/pt-br/downloads/xenapp/[Acedido pela última vez a 5 Junho 2020]. 12

[XSY18] Nilanjan Dey Amit Joshi Xin-She Yang, Simon Sherratt. *Third International Congress on Information and Communication Technology*. ISBN: 978-981-13-1164-2. Springer, 2018. Available from: https://link.springer.com/book/10.1007 2F978-981-13-1165-9. 1