UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE GRADUAÇÃO

Rodolfo Picoreti

Desenvolvimento de plataforma baseada em computação em nuvem para aproveitamento de recursos ociosos

Vitória-ES

Agosto/2017

Rodolfo Picoreti

Desenvolvimento de plataforma baseada em computação em nuvem para aproveitamento de recursos ociosos

Parte manuscrita do Projeto de Graduação do aluno Rodolfo Picoreti, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Vitória-ES

Agosto/2017

Rodolfo Picoreti

Desenvolvimento de plataforma baseada em computação em nuvem para aproveitamento de recursos ociosos

Parte manuscrita do Projeto de Graduação do aluno Rodolfo Picoreti, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Aprovado em 01, de Agosto de 2017.

COMISSÃO EXAMINADORA:

Prof. MSc. Alexandre Pereira do Carmo
Instituto Federal do Espírito Santo
Orientador

Profa. Dra. Raquel Frizera Vassallo
Universidade Federal do Espírito Santo
Coorientadora

Profa. MSc Cristina Klippel Dominicini
Instituto Federal do Espírito Santo
Examinador

Prof. Dr. Rodolfo da Silva Villaça
Universidade Federal do Espírito Santo
Examinador

Vitória-ES

Agosto/2017

AGRADECIMENTOS

Eu gostaria de agradecer a todos os meus colegas que me ajudaram tanto emocionalmente quanto de fato na execução deste trabalho. Eu só consegui completa-lo em tempo hábil devido a ajuda de vocês. Como diria a Raquel, "Seus amigos te bajulam demais". Então, vou usar esse espaço pra agradecer cada um desses insetões.

Gostaria de agradecer ao KBlo e Raquel por terem me adotado, orientado e por terem me aturado com tanta paciência durante todo esse tempo. Ao Fifi pela ajuda em diversos aspectos do projeto, principalmente na execução dos experimentos e na montagem dos gráficos. Ao Messi por ter me ajudado pacientemente a melhorar minha capacidade de escrita miserável. Ao Cabrunco pela ajuda no texto, mas principalmente por sempre me motivar a ser cada vez melhor, "Ainda bem que você sabe programar Rods, se não ninguém ia gostar de você"(>_>"). Ao Goiano, Flavin e Vitão pelos *fire's* e pelos memes.

Valeu insetos, amo vocês.

RESUMO

A evolução das redes de comunicação de computadores tornou possível a utilização de vários computadores trabalhando em paralelo na solução de problemas computacionais. Os sistemas formados por esses computadores foram denominados sistemas distribuídos. O mercado crescente e cada vez mais competitivo impulsionou o desenvolvimento desses sistemas. Atualmente, o paradigma mais utilizado é a computação em nuvem, que é resultado da evolução e da ampla adoção de tecnologias e paradigmas anteriores, tais como a computação em grade. O trabalho aqui proposto tem como objetivo a implementação de uma plataforma distribuída de processamento capaz de utilizar o tempo ocioso de computadores, a fim de acelerar aplicações que possuam uma alta necessidade de processamento. Seu diferencial em relação às plataformas semelhantes é a utilização de tecnologias modernas empregadas na computação em nuvem, como virtualização em nível de sistema operacional e arquitetura orientada a serviços. Além disso, a plataforma não impõe aos seus usuários o emprego de nenhum software/*middleware* específico. Os testes mostraram que a aplicação escolhida conseguiu ser executada com sucesso, mesmo não havendo nenhuma garantia de disponibilidade por parte da infraestrutura. Além disso, mostraram que a plataforma alcançou todos os objetivos propostos neste trabalho, se limitando apenas a exigencia de que os serviços sejam *stateless*.

Palavras-chave: Sistemas Distribuídos, Computação em Nuvem, Arquitetura Orientada a Serviços.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
CLI	*Command-line Interface*
HMAC	*Hash-based Message Authentication Code*
IaaS	*Infrastructure as a Service*
JSON	*Javascript Object Notation*
JWT	*JSON Web Token*
NIST	*National Institute of Standards and Technology*
OASIS	*Organization for the Advancement of Structured Information Standards*
PaaS	*Platform as a Service*
RC5	*Rivest Cipher*
REST	*Representational State Transfer*
SaaS	*Software as a Service*
SETI	*Search for Extraterrestrial Intelligence*
SQL	*Structured Query Language*
UFES	*Universidade Federal do Espírito Santo*

SUMÁRIO

1 INTRODUÇÃO

A evolução das redes de comunicação de computadores tornou possível a transferência de um grande volume de dados rapidamente. Essa capacidade possibilitou a utilização de vários computadores trabalhando em paralelo na solução de problemas computacionais. Os sistemas formados por esses computadores foram denominados sistemas distribuídos.

Um sistema distribuído pode ser compreendido como um conjunto de entidades independentes que cooperam entre si de modo a efetuar uma ação, como solucionar um problema, que não poderia ser realizada individualmente (COULOURIS et al., 2012). O estudo sobre sistemas distribuídos iniciou-se na década de 1970, onde cientistas do centro de pesquisa da Xerox Palo Alto criaram a rede Ethernet (IEEE, 2013), que acabou tornando-se o padrão de redes locais de comunicação.

Na década de 1990, surgem dois projetos que revolucionaram a história da computação distribuída. O primeiro - distributed.net - utilizou milhares de computadores pelo mundo de modo a quebrar por força bruta o algoritmo de criptografia RC5 (*Rivest Cipher*) de 56 bits, tarefa que durou 212 dias (DISTRIBUTED.NET, 2015). O segundo - SETI@HOME (*Search for Extraterrestrial Intelligence*) - foi criado pela Universidade de Berkeley com o objetivo de procurar por flutuações de sinais de rádio que pudessem indicar a existência de vida inteligente extraterrestre (SETI@HOME, 2011). Ambos adotaram um paradigma de computação conhecido como computação em grade.

O mercado crescente e cada vez mais competitivo impulsionou o desenvolvimento de novos paradigmas. Um exemplo dessa evolução pode ser observado no fim da década de 1990, onde a Salesforce.com dá início à ideia de vender software como serviços provisionados em sua própria infraestrutura (ERL; MAHMOOD; PUTTINI, 2013). Abordagem contrária à tradicional de vender programas que eram instalados e executados no computador do usuário.

Já em 2002, devido ao avanço das tecnologias de virtualização e computação em nuvem, a Amazon.com lança a plataforma AWS (*Amazon Web Services*) composta de um conjunto de serviços provisionados remotamente. Essa plataforma permitiu aos seus usuários alugar recursos computacionais de maneira autônoma por meio de uma interface *web*.

Com o passar do tempo, devido ao aumento na confiabilidade dos sistemas em nuvem, cada vez mais empresas adotaram tal paradigma, tornando-o cada vez mais difundido. Por exemplo, em 2008, devido a um grave problema em um de seus bancos de dados, a Netflix, empresa provedora global de filmes e séries de televisão via *streaming*, resolve mudar sua

infraestrutura para a nuvem. Desde então, ela teve um aumento de oito vezes no número de assinantes e um crescimento de três ordens de grandeza no número de visualizações. Acompanhar tal crescimento, só foi possível dado a elasticidade da nuvem, que possibilita provisionar uma enorme quantidade de recursos rapidamente (IZRAILEVSKY, 2016).

A partir da fusão da computação em nuvem e computação voluntária surge um novo paradigma denominado *Desktop Cloud*. Sua principal característica é prover serviços de nuvem em uma infraestrutura não dedicada para tal de modo a reduzir custos de execução e manutenção (ALWABEL; WALTERS; WILLS, 2014).

Alinhado com esses conceitos, o trabalho aqui proposto tem como objetivo a implementação de uma plataforma de processamento paralelo distribuído, permitindo que aplicações paralelizáveis sejam aceleradas. Para tal, a plataforma explorará o poder computacional originário do tempo ocioso de computadores de uso geral sem prejudicar as aplicações do usuário físico [1]. Dessa maneira, quando o computador estiver sendo empregado em sua designação principal, todas as aplicações da plataforma presentes no mesmo devem ser suspendidas.

A priorização do usuário físico gera como efeito adverso uma grande elasticidade na disponibilidade da infraestrutura, ou seja, um computador poderá tornar-se indisponível para a plataforma a qualquer momento. Mesmo nesses casos, a plataforma deve ser capaz de manter o funcionamento da aplicação com os recursos disponíveis.

De modo a minimizar os impactos da baixa confiabilidade da infraestrutura, foi proposto o desenvolvimento das aplicações seguindo uma arquitetura orientada a serviços. Espera-se que ao adotar tal arquitetura, o posicionamento e escalonamento de serviços na infraestrutura montada sejam simplificados e as diversas partes de uma aplicação possam ser escaladas individualmente.

1.1 Justificativa

A maioria das universidades brasileiras possui laboratórios de informática equipados com uma grande quantidade de computadores. Esses computadores são disponibilizados aos alunos para uso geral ou em aulas que necessitam de tal recurso. Por se tratarem de computadores de uso geral, sua aquisição visa atender os requisitos de pico de utilização. Por exemplo, tais computadores podem ser empregados tanto para navegação na internet quanto para design de projetos com softwares de CAD (*Computer Aided Design*). Assim,

[1] No decorrer do texto se utiliza apenas a palavra usuário para se referir a um cliente da plataforma proposta, já a palavra usuário físico se refere a um usuário de uma das máquinas da infraestrutura utilizada

eles deverão ser especificados de modo a atender bem a tarefa que necessite de maior quantidade de recursos, mesmo que essa não seja a tarefa mais frequente na qual o computador é empregado.

Por isso, ao se analisar a taxa de utilização e disponibilidade desses recursos, pode-se notar que estes, na maioria das vezes, são subutilizados. Paralelamente, grupos de pesquisas dessas universidades usualmente necessitam de um grande poder computacional na execução de seus experimentos, como simulações de modelos complexos entre outros. Tal poder computacional poderia ser originário do tempo ocioso dos computadores que se encontram nos laboratórios de informática.

Essa ideia de utilizar o tempo ocioso de computadores para acelerar aplicações cientificas já vem sendo amplamente empregada por projetos de computação em grade como BOINC (ANDERSON, 2004), Distributed.Net (DISTRIBUTED.NET, 2015) e XtremWeb (FEDAK et al., 2001). Esses projetos adotam um modelo de computação voluntária, permitindo que pessoas de todo o mundo possam doar o tempo ocioso de seus computadores, formando uma rede de computadores conhecida como *Desktop Grid*. O desenvolvimento de aplicações para esses projetos é feito através da utilização de *middlewares* específicos.

O BOINC criado em 2002 tornou-se o *middleware* de computação voluntária mais popular, com cerca de 900 mil usuários ativos. Entretanto, apesar de sua popularidade esse projeto ainda apresenta algumas desvantagens (MCGILVARY et al., 2013), requerendo que: as aplicações sejam portadas para todas as arquiteturas das máquinas da infraestrutura; que os desenvolvedores incluam em suas aplicações verificações para garantir que o progresso não seja perdido em caso de falha da infraestrutura; e as aplicações são executadas no espaço do usuário. Assim, os usuários do BOINC devem confiar que os projetos para os quais estão doando não distribuirão aplicações maliciosas.

Ainda em (MCGILVARY et al., 2013) é proposta a utilização de técnicas de virtualização como solução para os problemas citados. Com o emprego de máquinas virtuais, o usuário precisa apenas portar sua aplicação para a arquitetura de máquina virtual. Além disso, as preocupações com a segurança são tratadas naturalmente já que a aplicação roda em um ambiente isolado.

Assim, este trabalho tem como contribuição a criação de uma plataforma distribuída de processamento capaz de utilizar o tempo ocioso de computadores. Seu diferencial em relação às plataformas citadas é a utilização de tecnologias modernas empregadas na computação em nuvem, como virtualização em nível de sistema operacional. Além disso, a plataforma não impõem aos seus usuários a utilização de nenhuma linguagem

de programação ou *middleware* específico, ou seja, usuários podem escolher livremente ferramentas nas quais já estão familiarizados para desenvolver suas aplicações.

1.2 Objetivo Geral

O objetivo deste projeto de graduação é criar uma plataforma de processamento distribuído na qual membros dos diversos grupos de pesquisa da UFES possam acelerar a execução de seus experimentos/simulações através do aproveitamento de recursos computacionais ociosos presentes em laboratórios de informática da universidade. A plataforma deve ser capaz de atender a múltiplos clientes simultaneamente definindo para cada um deles uma quota máxima de recursos. Além disso, a utilização dos computadores para sua designação original deve ser priorizada.

1.3 Objetivos Específicos

De modo a atingir o objetivo geral deste projeto os seguintes objetivos específicos foram definidos:

1. instrumentar o uso de recursos em cada máquina da infraestrutura;
2. definir um modo padrão na qual as aplicações dos usuários serão empacotadas (e.g. uma imagem de uma máquina virtual/contêiner, etc). Nesse contexto, tal pacote será denominado tarefa;
3. criar uma ou mais interfaces na qual os usuários da plataforma sejam capazes de criar e gerenciar suas tarefas de maneira autônoma;
4. implementar um modo de limitar a quantidade de recursos que um usuário poderá utilizar;
5. permitir que vários usuários utilizem a infraestrutura simultaneamente caso esse possua recursos suficientes para tal;
6. implementar um método de detecção do login de um usuário físico nas máquinas da infraestrutura de modo a garantir que as tarefas só sejam executadas em máquinas ociosas.

2 EMBASAMENTO TEÓRICO

Neste capítulo será apresentada a base teórica necessária para o desenvolvimento do projeto proposto.

2.1 Sistemas Distribuídos

Um sistema distribuído pode ser definido como um conjunto de entidades independentes que cooperam entre si de modo a atingir um objetivo que não seria possível ser realizado caso contrário (COULOURIS et al., 2012). Diversos exemplos de sistemas distribuídos podem ser encontrados na natureza, como cardumes de peixes, colônia de formigas entre outros.

Em computação, um sistema distribuído pode ser definido como um conjunto de processadores autônomos separados geograficamente por qualquer distância que comunicam-se por meio de uma rede (COULOURIS et al., 2012). Um sistema distribuído tem como principais características:

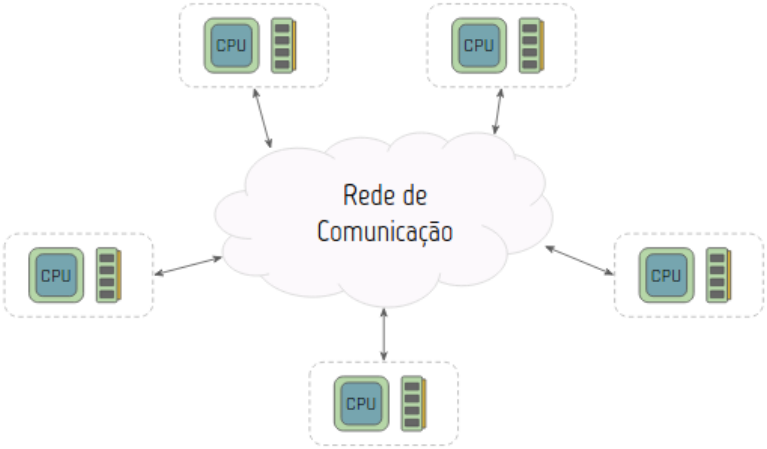
- **ausência de *clock* comum**: ausência de sincronia entre os processadores;
- **ausência de memória compartilhada**: essa característica implica que toda comunicação realizada entre os processadores tem que ser feita por meio de troca de mensagens;
- **heterogeneidade**: cada unidade pode possuir características distintas das outras, como *hardware*, sistema operacional, entre outros.

A Figura 1 ilustra o modelo básico de um sistema distribuído. Nele, cada computador possui uma unidade de processamento e memória. Os computadores são conectados por meio de uma rede de comunicação.

De modo a facilitar o desenvolvimento de software distribuído e abstrair as particularidades de cada sistemas, é comum a utilização de uma arquitetura dividida em camadas como mostrado na Figura 2.

Dentre essas camadas, a camada de *middleware* reside entre a aplicação e o sistema operacional. Seu objetivo é prover uma plataforma adequada para o desenvolvimento de aplicações distribuídas (SCHANTZ; SCHMIDT, 2007). Esse objetivo é alcançado por meio da criação de abstrações que escondem do usuário particularidades relacionadas às

Figura 1 – Modelo básico de um sistema distribuído



Fonte: Produção do próprio autor.

Figura 2 – Arquitetura dividida em camadas

Fonte: Produção do próprio autor.

camadas de nível mais baixo, como sistema operacional, que podem variar entre os diversos computadores do sistema.

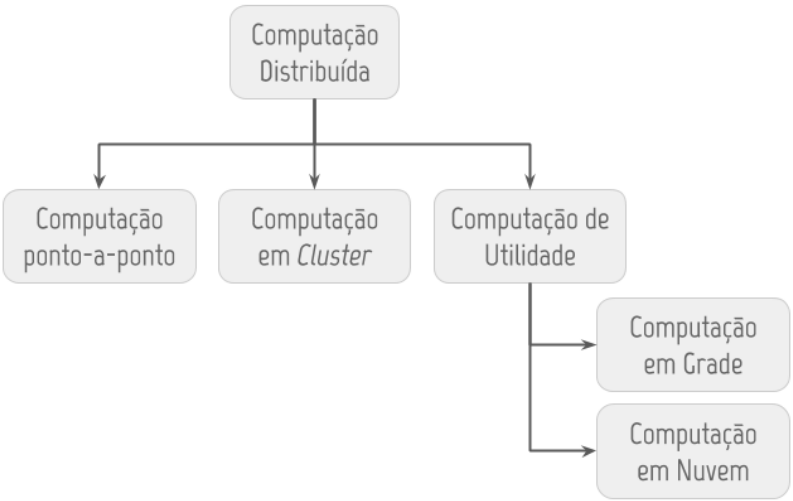
Dentre as motivações e vantagens da utilização de sistemas distribuídos pode-se destacar:

- **compartilhamento de recursos**: o termo recurso se refere a gama de coisas que podem ser proveitosamente compartilhadas, desde componentes de hardware (discos, impressoras, etc.) a entidades definidas por software (arquivos, bancos de dados, entre outros);
- **redução da razão custo/desempenho**: essa redução é uma das consequências do compartilhamento de recursos. Tarefas podem ser divididas entre os vários computadores do sistema de modo a acelerar a execução da mesma;

- **escalabilidade**: mais processadores podem ser adicionados ao sistema sem causar um grande impacto na rede de comunicação;
- **confiabilidade**: um sistema distribuído pode proporcionar uma maior confiabilidade devido a possibilidade de replicar recursos e tarefas em diversas partes do sistema. Adicionalmente, recursos geograficamente distribuídos possuem uma baixa probabilidade de falha simultânea. Confiabilidade envolve vários aspectos, como **disponibilidade** (se o recurso está acessível), **integridade** (se o recurso está em um estado consistente), **tolerância a falhas** (a capacidade do sistema de permanecer em funcionamento mesmo em situações de falha) e **resiliência** (a capacidade do sistema de se recuperar em caso de falha).

De modo a solucionar diferentes problemas, diversos paradigmas de computação distribuída foram surgindo com o tempo. A Figura 3 ilustra alguns paradigmas e suas relações.

Figura 3 – Alguns paradigmas da computação distribuída e suas relações



Fonte: Produção do próprio autor.

Nas seções a seguir serão apresentados os paradigmas e tecnologias mais relevantes para a compreensão da solução proposta.

2.2 Computação em *Cluster*

Um *cluster* é um conjunto de computadores independentes conectados por uma rede local de alta velocidade formando um recurso computacional único. (KAHANWAL; SINGH, 2012) Dentre os principais tipos de *cluster*, destacam-se:

- ***cluster* de alta-disponibilidade**: este tipo de *cluster* tem como objetivo aumentar a confiabilidade de uma aplicação por meio da introdução de redundância. Caso uma falha de hardware/software seja detectada, a aplicação é imediatamente transferida para outra parte do *cluster*, processo conhecido como *failover*.
- ***cluster* de balanceamento de carga**: este tipo de *cluster* tem como objetivo distribuir a carga de trabalho entre os computadores do *cluster* de maneira uniforme. É geralmente utilizado em aplicações que necessitam atender um alto volume de pedidos. O balanceamento de carga pode ser realizado por software ou hardware.
- ***cluster* de alta performance**: este tipo de *cluster* tem como objetivo prover uma alta capacidade computacional por meio da execução paralela de diversas tarefas. Este tipo de *cluster* é amplamente utilizado por pesquisadores e cientistas de modo a acelerar a execução de experimentos.

É comum que os componentes de um *cluster* possuam hardware razoavelmente semelhante, de modo a fornecer níveis de desempenho equivalente quando um componente com falha for substituído por outro.

2.3 Computação em Grade

Computação em grade é um paradigma da computação distribuída em que inúmeros sistemas heterogêneos são conectados de modo a compartilharem seus recursos e assim se comportarem como uma entidade lógica única com alto poder de processamento (FERREIRA et al., 2003). Esse paradigma se difere de clusterização no sentido que seus componentes são altamente desacoplados e distribuídos (ERL; MAHMOOD; PUTTINI, 2013).

O uso desse paradigma apresenta duas grandes vantagens: a possibilidade de aumentar a eficiência de recursos computacionais subutilizados conectando-os ao sistema; e a enorme capacidade de poder de processamento paralelo.

Para que essa última vantagem seja aproveitada, as aplicações devem ser construídas de modo que seus subcomponentes possam ser executados de maneira independente. Entretanto, esse particionamento nem sempre é possível devido a limitações nos algoritmos utilizados ou devido ao alto acoplamento entre duas partes. Por exemplo, se todos subcomponentes necessitam ler ou/e escrever em um banco de dados, a capacidade do banco de dados pode limitar a escalabilidade da aplicação.

2.4 Computação em Nuvem

A computação em nuvem é o resultado da evolução e da ampla adoção de tecnologias e paradigmas existentes. Pesquisas na área de computação em grade influenciaram vários aspectos de plataformas de computação em nuvem. Por exemplo, computação em grade requer a presença de um *middleware* que pode conter uma série de funções como, balanceamento de tarefas, coordenação, controle de falhas. Todas essas funções e outras ainda mais sofisticadas são encontradas atualmente em plataformas de computação em nuvem. É por essa razão que alguns classificam a computação em nuvem como uma evolução da computação em grade (ERL; MAHMOOD; PUTTINI, 2013).

O termo computação em nuvem é utilizado para capturar a visão de computação como uma utilidade (KSHEMKALYANI; SINGHAL, 2008). Como definido pelo Instituto Nacional de Padrões e Tecnologias dos Estados Unidos (NIST, do inglês *National Institute of Standards and Technology*) em (MELL; GRANCE, 2011a):

> Computação em nuvem é um modelo que permite um acesso conveniente e sob demanda por meio de uma rede a um conjunto de recursos computacionais configuráveis que podem ser provisionados rapidamente com um esforço administrativo mínimo ou sem interação com o provedor do serviço.

Segundo (MELL; GRANCE, 2011b), a computação em nuvem apresenta basicamente cinco características essenciais:

- ***on-demand self-service***: um usuário é capaz de provisionar recursos computacionais de maneira autônoma;
- **acesso abrangente**: os recursos são disponibilizados por meio de uma rede e acessados por meio de mecanismos padrões em diversas plataformas;
- **compartilhamento de recursos**: os recursos computacionais provisionados são compartilhados entre diversos consumidores. O consumidor não tem controle ou conhecimento do local exato no qual o recurso foi provisionado. Pode ser possível entretanto especificar o local de maneira mais geral (região, *datacenter*, etc);
- **elasticidade**: recursos podem ser alocados e desalocados facilmente, em alguns casos até automaticamente, de modo a escalar rapidamente o recurso dado uma alteração na demanda;

- **transparência**: a utilização de recursos deve ser monitorada, garantindo transparência para ambos o usuário e o provedor.

Além dessas características, também é comum dividir sistemas em nuvem em três modelos de serviço:

- software como um serviço (**SaaS**, do inglês *Software as a Service*): nesse modelo, o usuário tem acesso a aplicações gerenciadas por um provedor. O serviço é provisionado por meio da *web*, eliminando a necessidade de instalação do software no computador do usuário. Exemplos de SaaS: Google Apps e Salesforce.
- plataforma como um serviço (**PaaS**, do inglês *Platform as a Service*): nesse modelo, plataformas de computação (*frameworks*, banco de dados, etc) são oferecidas ao usuário na forma de um serviço. Exemplos de PaaS: Heroku, Force.com e Google App Engine.
- infraestrutura como um serviço (**IaaS**, do inglês *Infrastructure as a Service*): nesse modelo, o usuário pode alugar, gerenciar e monitorar recursos computacionais (máquinas virtuais, armazenamento, endereços IP, etc) presentes em um *datacenter*. Exemplos de IaaS: Amazon Web Services (AWS), Microsoft Azure e Google Compute Engine (GCE).

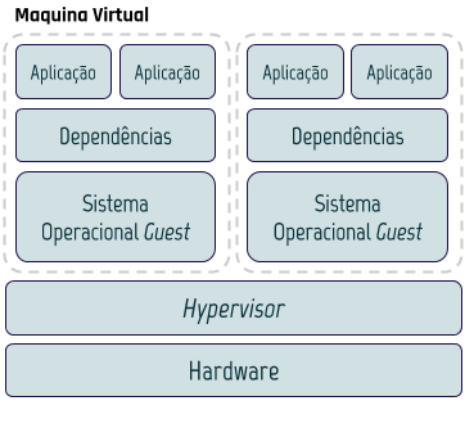
2.5 Virtualização

Virtualização é uma tecnologia que permite a criação de um recurso computacional virtual. Esta técnica pode ser aplicada em diversos níveis, como memória virtual, virtualização de funções de rede, virtualização de servidor, entre outros.

O uso de virtualização de servidor foi essencial para o sucesso do paradigma de computação em nuvem, pois permitiu que aplicações de diversos usuário pudessem ser executadas em um mesmo servidor de maneira isolada e segura, reduzindo custos e aumentando a flexibilidade da infraestrutura. Isso é possível por meio da utilização de um *software* ou *firmware*, denominado *hypervisor*, capaz de virtualizar recursos físicos do sistema. O conjunto formado pela união desses recursos virtuais e um sistema operacional é denominado máquina virtual. A Figura 4 apresenta um diagrama com os conceitos citados.

Outro tipo de virtualização importante no escopo deste trabalho é a virtualização em nível de sistema operacional. Contêiner é o termo genérico dado a implementação desse tipo de

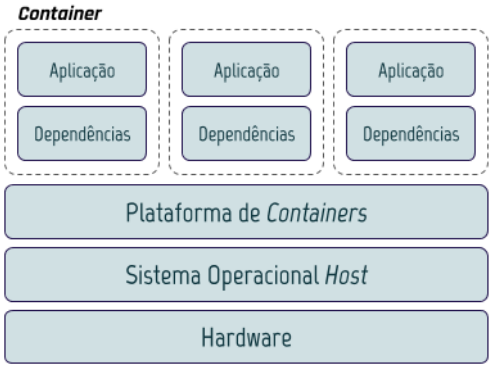
Figura 4 – Virtualização de servidor



Fonte: Produção do próprio autor.

virtualização. Tal implementação utiliza mecanismos de isolamento e gerenciamento de recursos oferecidos pelo próprio sistema operacional. No caso do Linux, tais mecanismos são conhecidos como *namespaces* e cgroups. O primeiro permite limitar o que a aplicação consegue acessar (arquivos, redes, etc). Já o segundo permite limitar o quanto uma aplicação pode usar de um determinado recurso (*CPU*, memória, etc). A Figura 5 apresenta um diagrama com os conceitos citados.

Figura 5 – Virtualização em nível de sistema operacional



Fonte: Produção do próprio autor.

Por compartilharem o *kernel* do sistema operacional com o *host*, contêineres utilizam menos recursos computacionais do que máquinas virtuais, fazendo com que mais aplicações possam ser executadas em uma mesma máquina. Entretanto, o compartilhamento do *kernel* faz com que contêineres possam ser considerados menos seguros do que máquinas virtuais. Além disso, o comprometimento de um contêiner pode afetar todos os contêineres em um mesmo *host*. Portanto, é comum a utilização de uma abordagem mista com máquinas virtuais e contêineres.

2.6 Arquitetura Orientada a Serviços

Arquitetura orientada a serviços apresenta uma abordagem para o desenvolvimento de sistemas distribuídos na qual as funcionalidades de uma aplicação são expostas na forma de serviços (ENDREI et al., 2004). Essa abordagem visa aumentar a eficiência, a agilidade e a produtividade de uma empresa (ERL, 2007).

Em (MACKENZIE et al., 2006), OASIS (*Organization for the Advancement of Structured Information Standards*) define uma arquitetura orientada a serviços como "[...] um paradigma para organizar e utilizar recursos distribuídos que podem estar sob o controle de diferentes domínios [...]".

Um serviço é o bloco fundamental de uma arquitetura orientada a serviços e pode ser definido como uma entidade de software autônoma que implementa uma ou mais funcionalidades. O acesso a suas funcionalidades é realizado por meio de uma API (*Application Programming Interface*) e deve respeitar possíveis restrições especificadas na descrição do serviço. O principal papel de um serviço é expor uma função relevante a um sistema de maneira flexível de modo que essa possa ser facilmente reutilizada e composta com outros serviços.

Os serviços também podem ser caracterizados como *stateful* e *stateless*. Um serviço *stateless* delega o armazenamento de estado para um outro componente do sistema na qual está imerso. Assim, cada invocação desse serviço pode ser feita independentemente de qualquer outra invocação previamente realizada, seja pelo mesmo consumidor ou não. Essa propriedade faz com que esses serviços possam ser escalados facilmente por meio do aumento do número de unidades do mesmo (escala horizontal). Desse modo, um balanceador de carga pode facilmente distribuir a carga de trabalho entre as instâncias do serviço. Já um serviço *stateful* armazena consigo o estado, nesse caso cada invocação do serviço estará acoplada a invocações previamente realizadas e ao consumidor que realizou o pedido. Assim, o balanceamento da carga dos serviços se torna mais complexa.

2.7 Kubernetes

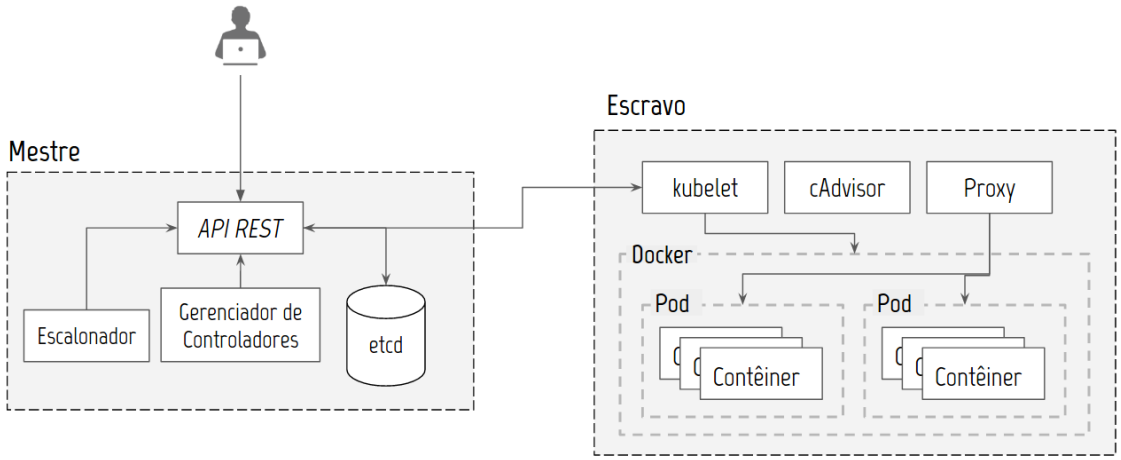
Kubernetes é uma plataforma de código aberto desenvolvida para o sistema operacional Linux que permite automatizar operações em aplicações conteinerizadas. O projeto Kubernetes foi iniciado pelo Google em 2014. Seu desenvolvimento foi amplamente influenciado pela experiência do Google em seu sistema interno de orquestração de contêineres, o Borg (KUBERNETES, 2017c).

Nesta seção, serão apresentados os aspectos do Kubernetes mais relevantes para o desenvolvimento deste trabalho. Inicialmente será apresentada uma visão geral da arquitetura de um *cluster* Kubernetes e então serão explicadas algumas de suas abstrações, denominadas recursos de API (e.g. pods, controladores e serviços).

2.7.1 Arquitetura

Um *cluster* Kubernetes é composto basicamente por um plano de controle, um agente (kubelet) que expõe uma API de controle em cada nó escravo (nós onde os contêineres de usuários serão executados) e um banco de dados distribuído responsável por armazenar o estado do *cluster*. A Figura 6 mostra um diagrama de alto nível de cada um desses componentes e como eles estão conectados.

Figura 6 – Visão simplificada da arquitetura de um *cluster* Kubernetes



Fonte: Produção do próprio autor.

O plano de controle é dividido em um conjunto de componentes que podem ser executados em um único nó mestre ou replicados em diversos nós (alta disponibilidade). O servidor de API oferece uma API REST (*Representational State Transfer*) que serve como um *gateway* de acesso as funcionalidade do *cluster*. Este servidor é responsável por validar

e executar os comandos REST e então atualizar o estado do *cluster*. Todo o estado do *cluster* é armazenado em um banco de dados chave-valor distribuído, o etcd. O escalonador, como o nome indica, é o componente que determina onde cada Pod (unidade básica de escalonamento) será escalonado dentro do *cluster* levando em conta requisitos de recursos, exigência de qualidade de serviço entre outros. Por fim, o gerente de controladores é o componente que encapsula os principais algorítimos de controle do *cluster*.

Já um nó escravo tem os serviços necessários para que aplicações conteinerizadas possam ser executadas e gerenciadas pelo plano de controle. O kubelet é o agente que expõe tais funcionalidades em cada nó escravo. Adicionalmente, cada nó executa um processo kube-proxy responsável por implementar a abstração de serviço da API do Kubernetes. Para tal, o kube-proxy modifica as regras do iptables de modo a interceptar o acesso aos IPs de serviço e redirecioná-los para os *backends* corretos.

2.7.2 Recursos de API

Um pod é a unidade básica de escalonamento em um *cluster* Kubernetes. Ele agrega um ou mais contêineres de modo que sejam sempre alocados em um mesmo nó. Cada Pod possui um endereço IP único dentro do *cluster*, evitando problemas de conflitos de porta entre aplicações. Esse recurso pode ser gerenciado diretamente por meio da API do Kubernetes, entretanto, é comum que seu gerenciamento seja delegado a um controlador.

Um controlador é um recurso do Kubernetes que tem como função alterar o estado atual do *cluster* de modo que ele alcance um estado desejado, similar a ideia de um controlador utilizado na robótica e automação. Alguns exemplos de controladores e suas funções são:

1. ***replication controller***: garante que um número de réplicas de um determinado pod esteja em execução;
2. ***daemonset controller***: garante que exatamente uma unidade de um pod esteja rodando em cada nó do *cluster*;
3. ***job controller***: cria um ou mais pods garantindo que um determinado número deles sejam executados com sucesso.

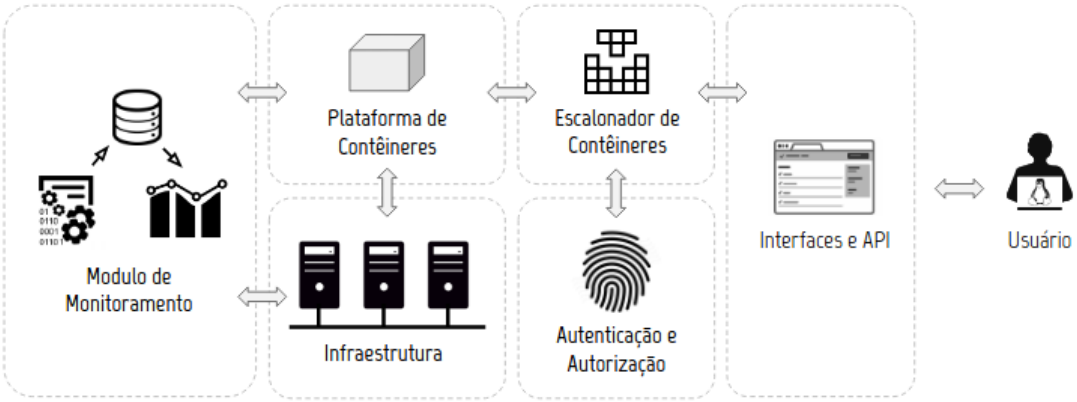
Apesar de cada pod possuir um endereço IP único dentro do *cluster*, o Kubernetes não oferece nenhuma garantia que esses endereços se manterão constantes ao longo do tempo. Adicionalmente, pods gerenciados por controladores podem ser criados e destruídos dinamicamente. Logo, aplicações que necessitem acessar uma funcionalidade oferecida por esses pods teriam que manter uma lista atualizada dos endereços IPs dos mesmos.

A abstração de serviços do Kubernetes tem como objetivo solucionar esse tipo de problema. Um serviço, nesse contexto, agrupa um conjunto de pods que executam a mesma função e define uma política de acesso aos mesmos. Dessa maneira, aplicações que necessitem de uma funcionalidade oferecida por um conjunto de pods podem, por exemplo, sempre acessar o mesmo endereço IP do serviço e então fica a cargo do serviço redirecionar esse pedido ao *backend* apropriado.

3 ARQUITETURA PROPOSTA

Para que fosse possível criar a plataforma de processamento distribuído desejada, a arquitetura apresentada na Figura 7 foi desenvolvida.

Figura 7 – Arquitetura proposta



Fonte: Produção do próprio autor.

A Figura 7 mostra os diversos componentes da arquitetura proposta e a interação entre eles. A plataforma de contêineres encapsula as aplicações de forma que possam ser executadas na infraestrutura. O escalonador determina em qual máquina física dessa infraestrutura cada contêiner será alocado, possibilitando que as aplicações possam ser implantadas elasticamente. O escalonador conhece o limite de recursos disponível em cada máquina e o quanto já foi alocado nela, mas não armazena nenhum histórico desses dados. Essa função é do módulo de monitoramento de recursos. Dessa forma, é possível analisar o impacto do sistema proposto na infraestrutura.

Para que o usuário consiga utilizar o sistema de maneira autônoma, foram criadas duas interfaces com diferentes compromissos: uma web simplificada contendo apenas funcionalidades básicas, e uma interface REST contendo uma gama maior de funcionalidades, destinada a usuários avançados.

Esses componentes são suficientes para que um usuário possa utilizar o sistema. Entretanto, para que vários usuários possam utilizar a plataforma simultaneamente, é preciso haver algum tipo de controle. Ou seja, que a plataforma seja capaz de identificar quem é o usuário e o que ele pode fazer. Essa é a função do módulo de autenticação e autorização.

Todos esses componentes definem a arquitetura do sistema e serão explicados em detalhes

e especificados nas seções desse capítulo. Essas seções estão separadas em plataforma e escalonador de contêineres (Seções 3.1 e 3.2), monitoramento de recursos (Seção 3.3), interface do usuário (Seção 3.4) e autenticação e autorização (Seção 3.5).

3.1 Plataforma de contêineres

Durante a implantação de uma aplicação, alguns problemas podem surgir. Por exemplo, o ambiente na qual a aplicação será executada pode não conter as bibliotecas necessárias ou conter versões incompatíveis com as requeridas pela aplicação. Um modo de resolver esses problemas é a utilização de técnicas de virtualização.

Por meio da utilização de um mesmo ambiente virtual no desenvolvimento e implantação da aplicação, é possível garantir que ela funcionará conforme o esperado independentemente da máquina física que for implantada, já que todas suas dependências estarão presentes no ambiente virtual.

Além disso, como a aplicação e suas dependências estão empacotadas em um único elemento, se torna mais prático distribuí-la dinamicamente dentro de uma infraestrutura[1]. O emprego de técnicas de virtualização permite também que o hardware físico possa ser compartilhado entre aplicações distintas, e que a utilização de recursos físicos por uma aplicação possa ser controlada/limitada.

Em resumo, é necessário que as aplicações sejam empacotadas para que possam ser executadas em qualquer lugar de modo previsível. Escolheu-se a utilização de contêineres como o modo padrão de empacotamento de aplicações. Uma outra opção seria a utilização de máquinas virtuais, entretanto, elas encapsulam em si não só a aplicação como também um sistema operacional completo. Portanto, possuem tamanho superior que os contêineres. Desse modo, numa mesma máquina, é possível alocar mais contêineres do que máquinas virtuais para uma mesma aplicação, o que justifica a escolha de contêineres.

Dentre as distintas plataformas de contêineres existentes (e.g. Docker, rkt (RKT, 2017), etc.), optou-se pela utilização da plataforma Docker. Atualmente, é a plataforma de contêineres mais adotada na indústria (DOCKER, 2017e), sendo utilizada por grandes empresas como PayPal, Ebay entre outras (DOCKER, 2017a).

O Docker possui todas as ferramentas necessárias para a criação de imagens de contêineres tanto de maneira manual como automática (através do uso de um Dockerfile (DOCKER, 2017d)). Além disso, dispõe de um repositório aberto de imagens, o Docker Hub, que,

[1] É mais fácil levar várias bolinhas de gude quando estão todas dentro de um pote de vidro, por exemplo.

atualmente, conta com mais de cem mil imagens públicas (DOCKER, 2017b).

3.2 Escalonador de contêineres

Para que as aplicações possam ser alocadas de maneira elástica, a escolha de onde o contêiner será alocado na infraestrutura deve ser feita de maneira automática. O processo de escolha deve também levar em consideração os requisitos individuais de cada aplicação e os limites de recursos de cada máquina. O elemento responsável por essa tarefa na arquitetura proposta é o escalonador de contêineres.

A infraestrutura utilizada pode se tornar indisponível a qualquer momento, já que ela deve atender a sua designação principal sem prejudicar a experiência do usuário físico. Somente seu tempo ocioso deve ser utilizado pela plataforma proposta. Desse modo, o escalonador deve ser capaz de identificar a presença de um usuário físico e então mover todas as tarefas alocadas na máquina para outra parte da infraestrutura, quando possível.

Dentre as principais opções de escalonadores, pode-se citar Mesos (MESOS, 2017), Docker Swarm Mode (DOCKER, 2017c) e Kubernetes. Todos esses são capazes de escalonar contêineres Docker e possuem uma API REST que dá acesso a suas funcionalidades.

O Docker Swarm Mode é um recurso do Docker que fornece funcionalidades de orquestração de contêiner. Uma das vantagens desse recurso é a extrema facilidade com que um *cluster* pode ser gerenciado. Entretanto, ele foi descartado pois, não possui um mecanismo no qual um contêiner possa reservar uma quantidade de recursos para si.

Já no Kubernetes, é possível que uma certa quantidade de um recurso seja reservada a um contêiner. Adicionalmente, o Kubernetes possui o conceito de *namespaces*, que permite que um *cluster* físico seja subdividido em diversos *cluster* virtuais. Assim, ao atribuir uma quota máxima de recursos para um *namespace* e limitando o acesso de um usuário a apenas seu *namespace*, é possível limitar o quanto um usuário pode utilizar do total de recursos do *cluster*.

Uma das desvantagens do Kubernetes é que sua implantação é extremamente difícil quando comparada com o Docker Swarm Mode, por exemplo. De modo a facilitar a implantação de um *cluster* Kubernetes, a ferramenta Kubeadm (KUBERNETES, 2017c) foi criada pela comunidade. Com ela, é possível implantar o *cluster* executando um único comando em cada um de seus nós, de modo similar ao Docker Swarm. Atualmente, essa ferramenta se encontra em uma versão alfa, apresentando limitações. Por exemplo, o *cluster* criado conterá apenas um nó mestre, com uma única instância do etcd. Logo, caso o nó mestre se

torne indisponível, o *cluster* perderá sua configuração e terá que ser recriado a partir do zero.

Portanto, por atender todos os requisitos especificados e possuir uma facilidade de gerenciamento semelhante ao Docker Swarm Mode, adotou-se o Kubernetes como o escalonador de contêineres da plataforma.

3.3 Monitoramento de Recursos

De modo a medir o impacto da implantação da plataforma na infraestrutura escolhida, a arquitetura proposta conta com um sistema de monitoramento de recursos composto por três elementos: um elemento responsável por coletar as métricas de utilização de cada máquina; um banco de dados que armazenará essas métricas; um painel na qual as métricas possam ser visualizadas e analisadas.

Com o Kubernetes foi possível implantar o sistema de monitoramento de recursos descrito por meio da utilização de um de seus *addons*, o *cluster-monitoring* (KUBERNETES, 2017d). Ele é composto pelos seguintes elementos de código aberto:

- **Heapster**: uma ferramenta que coleta diversas métricas do *cluster*, como utilização de *cpu* e memória, e as expõe por meio de um API REST (KUBERNETES, 2017a). Além disso, o Heapster é capaz de enviar essas métricas para uma série de *backends* de armazenamento;
- **InfluxDB**: um banco de dados especializado para o armazenamento de séries temporais (INFLUXDATA, 2017). Ele possui uma API REST que permite a seus usuários realizar consultas de maneira flexível, utilizando uma linguagem semelhante a SQL (*Structured Query Language*). O InfluxDB serve como o *backend* de armazenamento das diversas métricas do *cluster*.
- **Grafana**: uma plataforma de análise e monitoramento de métricas que permite a criação de painéis e gráficos. Ela é capaz de coletar métricas de diversos *backends*, sendo o InfluxDB um deles (GRAFANA, 2017). A Figura 8 mostra o exemplo de um painel criado com a utilização do Grafana. Vale ressaltar que esse painel foi gerado automaticamente pelo *addon* de monitoramento do Kubernetes.

Figura 8 – Exemplo de painel de visualização do Grafana

Fonte: Produção do próprio autor.

3.4 Interface do usuário

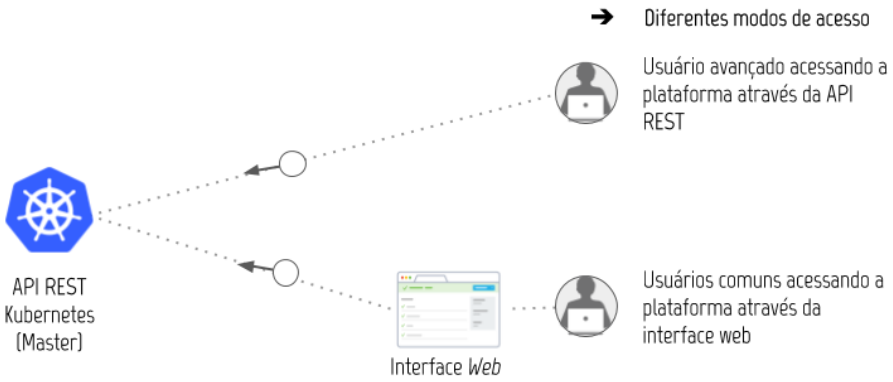
O usuário deve ser capaz de implantar suas aplicações no sistema de maneira autônoma. Para tal, o sistema conta com duas interfaces: uma *web* simplificada contendo apenas funcionalidades básicas, e uma interface REST com uma gama maior de funcionalidades, permitindo também que os usuários interajam com o sistema de uma maneira programática. As duas interfaces estão representadas na Figura 9.

Como o Kubernetes já possui uma API REST, optou-se por adotá-la [2]. Já a aplicação *web* foi criada utilizando o *framework* javascript Meteor (METEOR, 2017), que proporciona um ambiente completo para o desenvolvimento de todas as camadas que compõem a aplicação (*fullstack*).

O *backend* da aplicação *web* criada faz uso da API REST do Kubernetes para executar comandos e ler o estado atual do *cluster*. A Figura 10 mostra o painel de gerenciamento de aplicações. Nele, o usuário pode implantar aplicações, visualizar o estado das mesmas e verificar a quantidade de recursos que suas aplicações estão consumindo em relação a sua quota. Já a Figura 11 mostra o formulário que o usuário deve preencher para implantar uma nova aplicação no *cluster*.

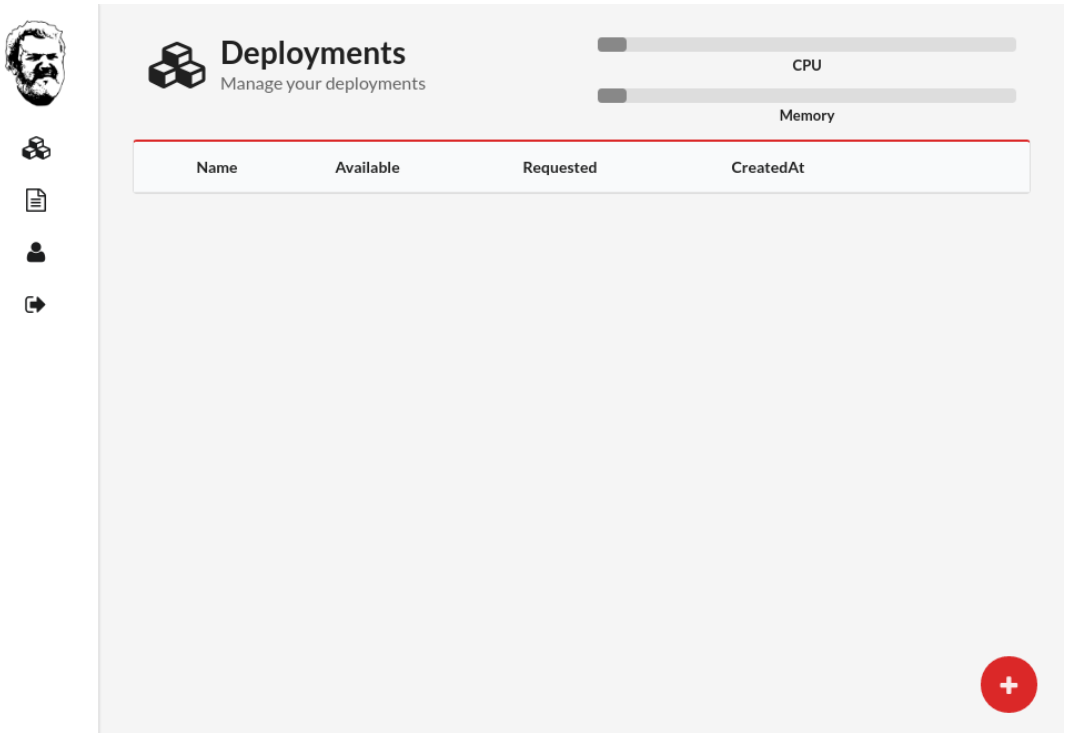
[2] A documentação dessa API pode ser encontrada em (KUBERNETES, 2017b)

Figura 9 – Diferentes interfaces da plataforma



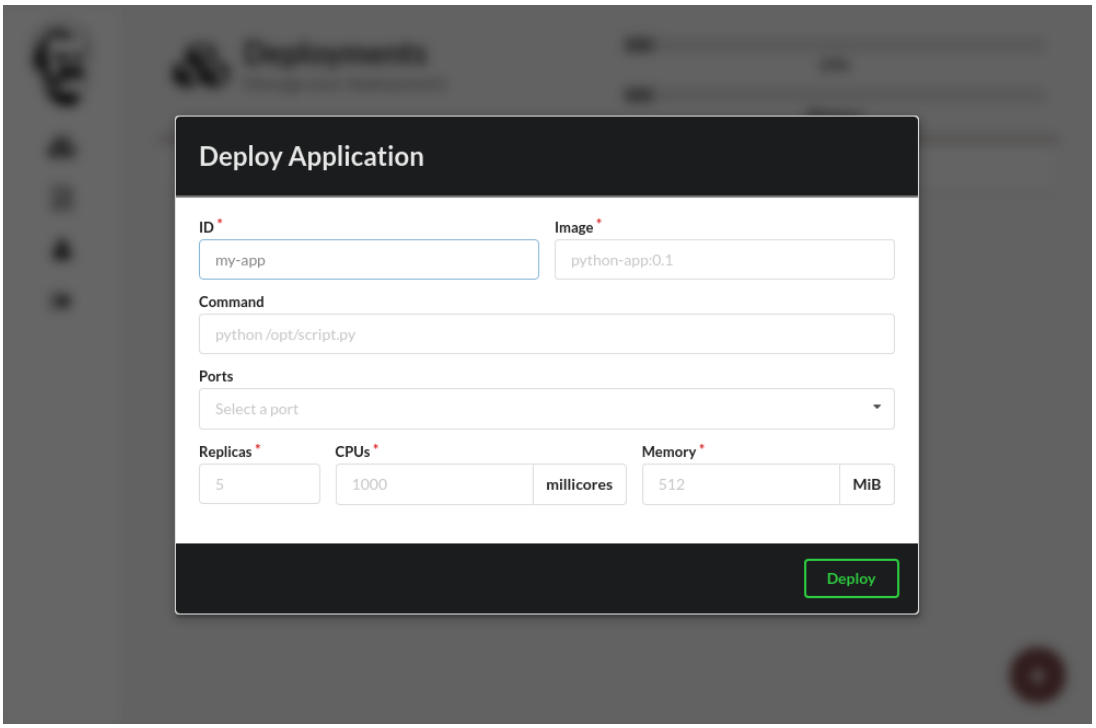
Fonte: Produção do próprio autor.

Figura 10 – Painel de gerenciamento de aplicações



Fonte: Produção do próprio autor.

Figura 11 – Formulário para implantação de uma nova aplicação



Fonte: Produção do próprio autor.

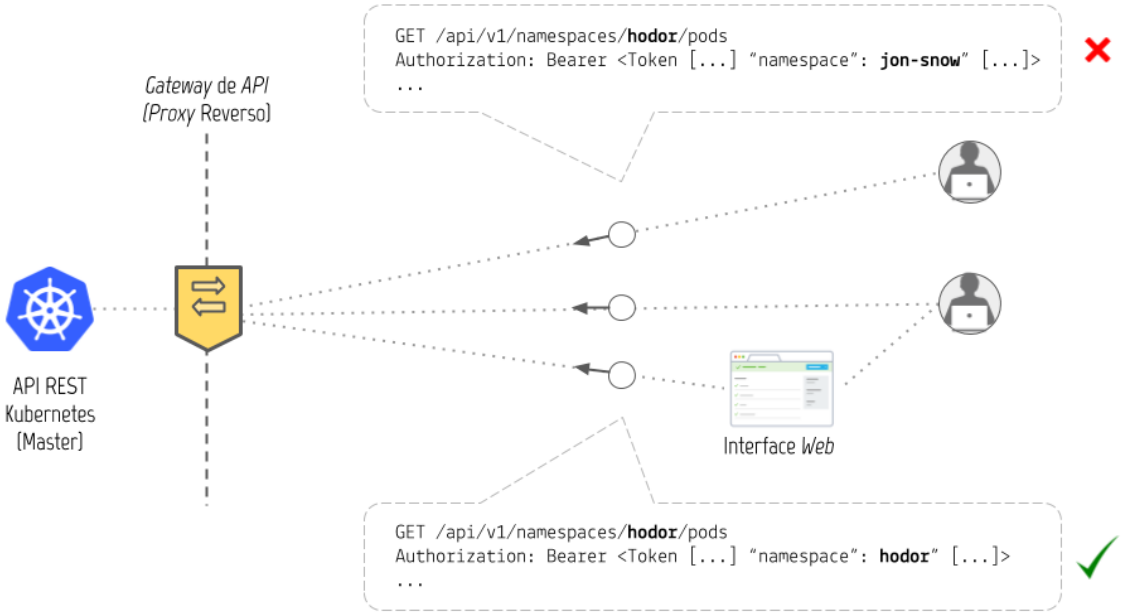
3.5 Autenticação e Autorização

Para permitir que vários usuários possam utilizar a plataforma, é necessário que a plataforma seja capaz de identificar quem é o usuário e determinar o que ele pode fazer. O Kubernetes possui diversos mecanismos para tal, como *tokens*, certificados, entre outros. Entretanto, por uma questão de simplicidade, propôs-se uma solução rápida e suficiente para implementação destes mecanismos, a utilização de um *proxy* reverso como mostrado na Figura 12.

O *proxy* reverso é responsável por autenticar e autorizar as solicitações de clientes e então encaminha-los para o *endpoint* correto da API do Kubernetes. O mecanismo proposto foi implementado por meio da utilização de *tokens* JWT (*JSON Web Tokens*).

JWT é um padrão aberto (RFC 7519) que define um método compacto e auto-contido de transmitir informações entre duas partes de maneira segura (JWT, 2017). Ele é auto-contido no sentido que toda a informação necessária sobre o usuário está no próprio *token*, evitando a necessidade de consulta a um banco de dados mais de uma vez. A integridade da informação transmitida pode ser verificada já que os *tokens* são assinados digitalmente. Um *token* pode ser assinado com um segredo por meio da utilização do algoritmo HMAC (*Hash-based Message Authentication Code*). A Figura 13 mostra um exemplo de um *token*

Figura 12 – Esquema de *proxy* reverso proposto



Fonte: Produção do próprio autor.

utilizado na plataforma.

Figura 13 – Exemplo de *token* JWT

Fonte: Produção do próprio autor.

Como mostrado na Figura 13, o *token* é formado pela concatenação de três partes

delimitadas por pontos: o cabeçalho, o *payload* e a assinatura. Cada parte é codificada com o algoritmo Base64. O cabeçalho indica qual algoritmo deve ser utilizado para gerar a assinatura do *token* e consequentemente verificar se o *token* não foi modificado indevidamente.

O *payload* do *token* contém as informações necessárias para realizar a autenticação (*name*) e autorização (*role, namespace, limits*) do usuário. Os campos *namespace* e *limits* indicam respectivamente o *namespace* no qual o usuário tem acesso e a quota máxima de recursos disponível nesse *namespace*. O *token* também conta com um campo de expiração (exp) e um identificador único (jti).

Desse modo, o *proxy* reverso só redirecionará o pedido do usuário caso o *endpoint* requisitado pertença ao *namespace* indicado no *token* e caso o *token* não tenha expirado. Esse processo é ilustrado na Figura 12, onde o usuário **jon-snow** tenta acessar um *endpoint* referente ao *namespace* **hodor**, mas tem seu acesso negado.

3.6 Detecção de Usuário Físico

Escalonadores de contêineres são desenvolvidos de modo a gerenciarem contêineres em uma infraestrutura dedicada para tal. Assim, foi necessário o desenvolvimento de um módulo a parte, responsável por garantir que as tarefas da plataforma sejam executadas apenas no tempo ocioso das maquinas da infraestrutura.

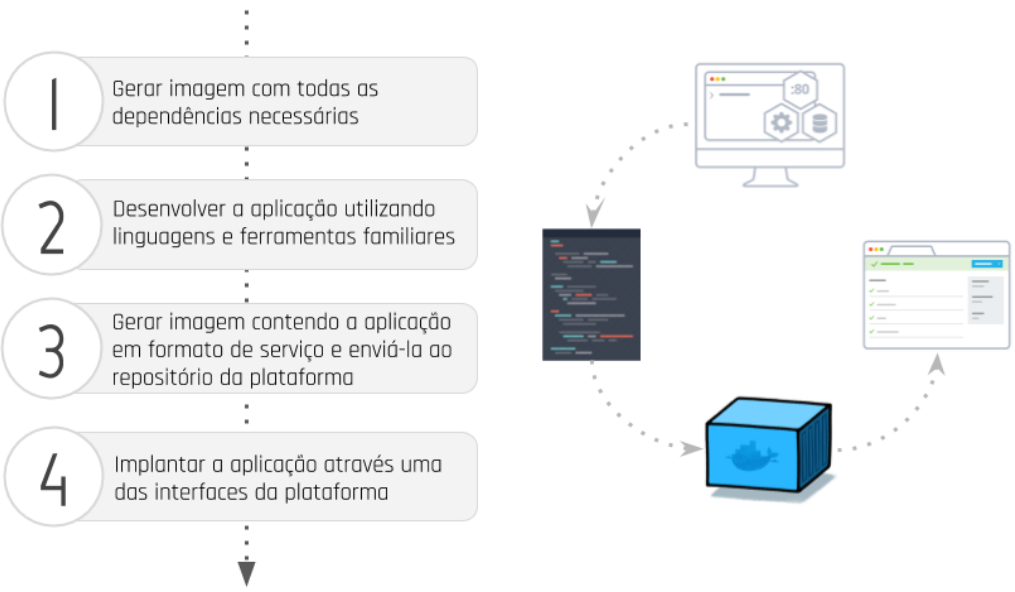
Para tal, desenvolveu-se um *daemon* que monitora periodicamente as sessões do sistema. Ao detectar a criação de uma nova seção, o *daemon* envia um requisição REST para o *proxy* reverso do sistema informando que o nó na qual ele está alocado deve se tornar indisponível. A autenticação é feita através da utilização de um *token* especial que só da acesso a essa funcionalidade.

Ao receber tal requisição, o *proxy* reverso envia um pedido à API REST do Kubernetes marcando o nó em questão como *unschedulable*, fazendo assim, com que nenhum pod seja escalonado nele. Em seguida, todos os pods presentes neste nó são removidos. Por fim, ao detectar o fim de uma seção, o *daemon* envia uma requisição REST ao *proxy* reverso de modo a permitir que pods sejam novamente escalonados no nó em questão.

4 DESENVOLVIMENTO DE APLICAÇÕES

Neste capítulo, serão abordadas todas as etapas do ciclo de desenvolvimento de uma aplicação de modo que ela possa ser implantada na plataforma proposta. A Figura 14 ilustra cada uma dessas etapas de maneira resumida. Paralela a essa explicação, será apresentado um exemplo de uma aplicação desenvolvida em Python e como os conceitos abordados nesse capítulo se aplicariam a ela. Além disso, assume-se que o usuário possui um conhecimento básico de Docker e de algum *middleware* de computação distribuída. No exemplo mostrado neste capítulo, será utilizado o *middleware* Celery, por ser amplamente adotado e de fácil uso, permitindo ao usuário preocupar-se apenas com a lógica da sua aplicação.

Figura 14 – Passos básicos para a utilização da plataforma proposta



Fonte: Produção do próprio autor.

4.1 Preparação do ambiente de desenvolvimento

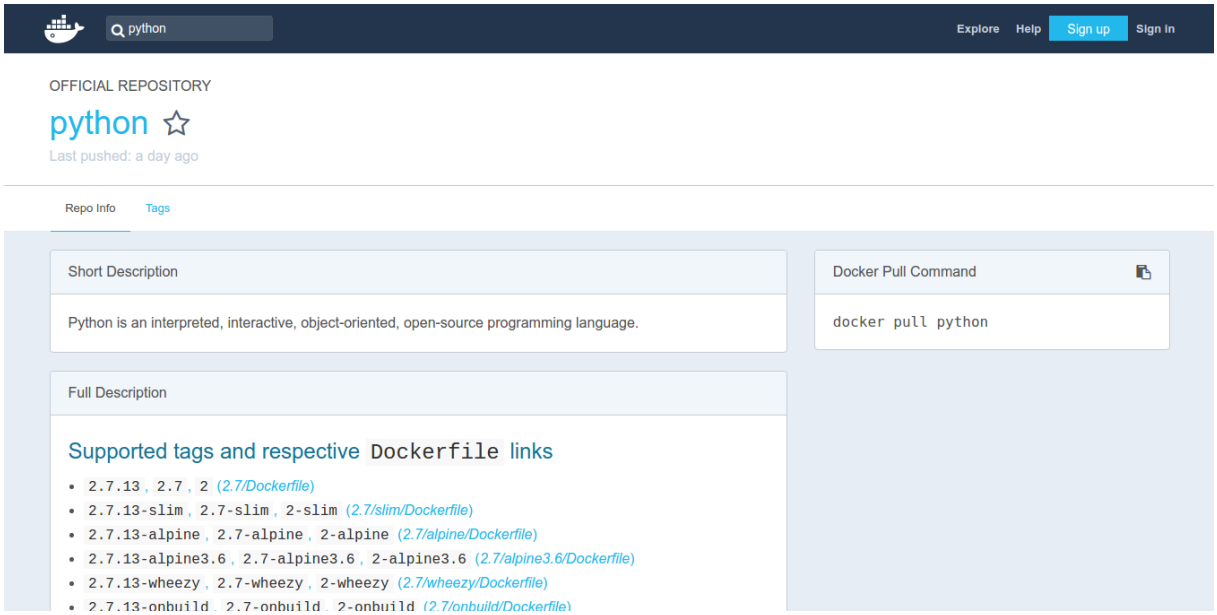
O primeiro passo pra criação de uma aplicação é a preparação do ambiente de desenvolvimento. Para tal, o usuário deve instalar a plataforma de contêineres escolhida, o Docker. Ele pode ser facilmente instalado em todos os três principais sistemas operacionais (Windows, Mac e Linux).

A criação do ambiente de desenvolvimento começa pela escolha da imagem base que será utilizada. O usuário pode procurar imagens utilizando o Docker Hub (DOCKER, 2017b). A escolha da imagem dependerá da linguagem escolhida pelo usuário no desenvolvimento

da aplicação. Por exemplo, caso o usuário queira desenvolver sua aplicação utilização a linguagem Python, ele poderia escolher uma das imagens oficiais de Python disponíveis no Docker Hub como mostrado na Figura 15.

No exemplo apresentado nesse capítulo, será utilizada a imagem **python:2.7-alpine**, que é uma imagem baseada na distribuição linux Alpine (ALPINE, 2017). Uma das vantagens do uso dessa distribuição é seu tamanho reduzido [1].

Figura 15 – Resultado da busca por imagens Python no Docker Hub



Fonte: Produção do próprio autor.

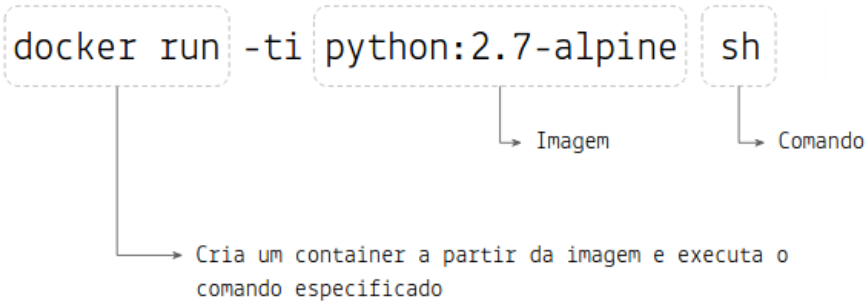
Escolhida a imagem base adequada, deve-se agora instalar todas as dependências necessárias para o desenvolvimento da aplicação, gerando-se uma nova imagem. Esse processo pode ser realizado manualmente ou automaticamente. Por ser a abordagem mais natural, o processo manual de criação de imagens será discutido primeiro e em sequência será mostrado como ele poderia ser automatizado com a utilização de um Dockerfile.

Um novo contêiner pode ser criado executando-se o comando mostrado na Figura 16 em um terminal. Após a execução do comando, o terminal estará vinculado com uma sessão dentro do contêiner executando um processo **sh** (*bourne shell*). Pode-se agora instalar as dependências do mesmo modo que em uma máquina física.

Entretanto, a escolha de uma imagem base de tamanho reduzido faz com que ocasionalmente seja necessário a instalação de alguns pacotes que já estariam presentes por padrão. Por

[1] Para efeitos de comparação, uma imagem Python 2.7 baseada na distribuição Debian Weezy tem um tamanho comprimido de 203 MB já sua versão baseada em Alpine possui um tamanho de apenas 25 MB

Figura 16 – Comando utilizado para criar um contêiner Docker



Fonte: Produção do próprio autor.

exemplo, a instalação do pacote numpy requer a presença de pacotes básicos de compilação, como o gcc, make, entre outros. Entretanto, tais pacotes podem ser removidos após a instalação visto que só são necessários nesse momento. Com isso, o tamanho da imagem será reduzido.

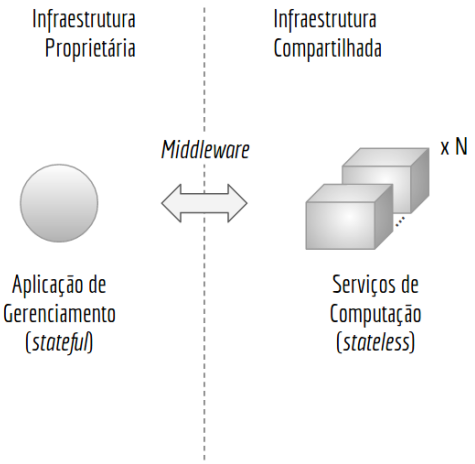
Além disso, é recomendado que a aplicação seja implementada em forma de um serviço *stateless* e todo o estado seja delegado para uma aplicação sendo executada na infraestrutura do próprio usuário, como mostrado na Figura 17. Tal recomendação se deve a baixa confiabilidade da infraestrutura na qual ela será implantada, ou seja, o computador na qual as aplicações estão sendo executadas podem ser tornar indisponíveis a qualquer momento, assim, caso uma aplicação contenha algum estado e ele não tenha sido persistindo de alguma maneira, esse estado será perdido pra sempre. Uma vantagem da criação de serviços *stateless* é que esses podem ser facilmente escalados.

No desenvolvimento da aplicação é interessante que um *middleware* seja utilizado, de modo que o desenvolvedor possa focar apenas na lógica da aplicação. Recomenda-se o uso do pacote celery (CELERY, 2017a) para o desenvolvimento de aplicações em Python, uma vez que esse *middleware* permite facilmente transformar uma função em um serviço. A Figura 18 mostra a sequência de comandos necessários para instalação das dependências citadas (numpy e celery) dentro do contêiner **python:2.7-alpine** escolhido.

Assim, o ambiente de desenvolvimento está pronto, restando apenas atribuir-se um nome para a imagem criada. Isso pode ser feito com o comando mostrado na Figura 19.

Todo o processo apresentado nessa seção pode ser automatizado por meio da utilização de um Dockerfile, como o mostrado na Figura 20. A imagem pode então ser gerada executado-se o comando apresentado na Figura 21.

Figura 17 – Arquitetura recomendada para a construção de aplicações



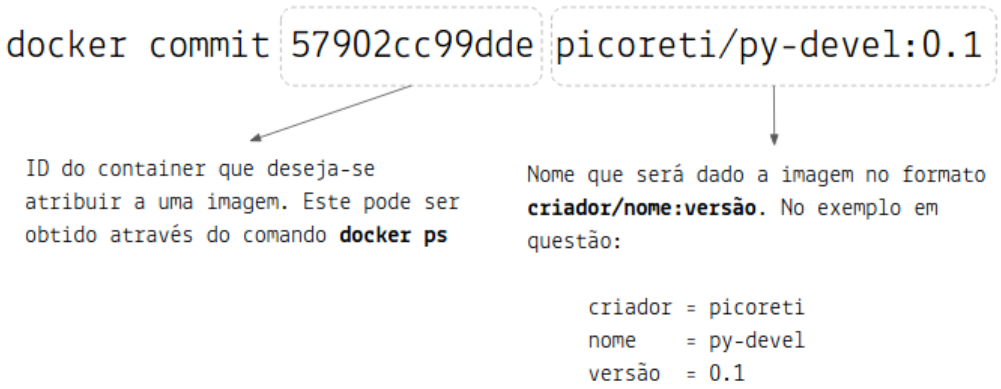
Fonte: Produção do próprio autor.

Figura 18 – Sequência de comandos para instalação das dependências

```
/ # apk add --update --no-cache build-base
/ # pip install numpy
/ # pip install celery
/ # apk del build-base
```

Fonte: Produção do próprio autor.

Figura 19 – Comando Docker para guardar uma imagem em execução



Fonte: Produção do próprio autor.

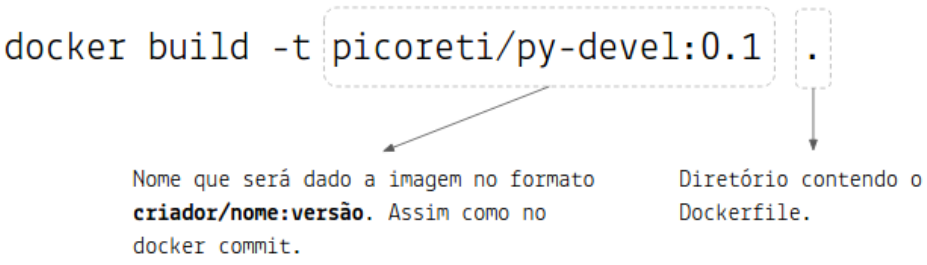
Figura 20 – Conteúdo do Dockerfile

```
FROM python:2.7-alpine

RUN apk add --update --no-cache build-base &&\
   pip install numpy &&                      \
   pip install celery &&                     \
   apk del build-base
```

Fonte: Produção do próprio autor.

Figura 21 – Comando para a construção da imagem a partir de um Dockerfile

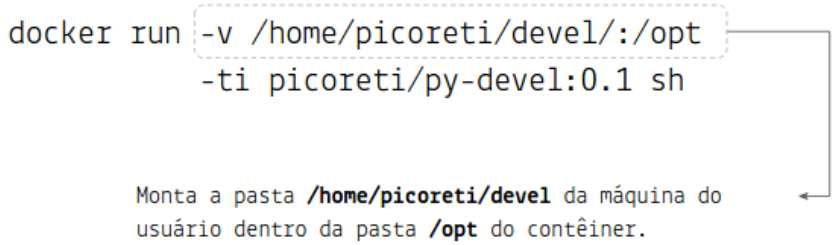


Fonte: Produção do próprio autor.

4.2 Desenvolvimento da Aplicação

Com o ambiente de desenvolvimento criado, o usuário pode agora desenvolver sua aplicação utilizando as ferramentas na qual está acostumado. Para que o desenvolvedor consiga executar a aplicação, é necessário que ele monte a pasta contendo a aplicação dentro do contêiner. Isso pode ser feito utilizando-se os comandos mostrados na Figura 22.

Figura 22 – Montando a pasta de desenvolvimento dentro do contêiner



Fonte: Produção do próprio autor.

Finalizado o desenvolvimento da aplicação, ela pode ser convertida para um serviço com a biblioteca celery utilizando-se um *decorator*, como mostrado na Figura 23. Nela uma aplicação que soma dois vetores elemento a elemento é transformada em um serviço por meio da adição das três linhas indicadas com setas na figura.

Figura 23 – Transformando uma aplicação em um serviço com o Celery

```
    import numpy as np

⇨   from celery import Celery
⇨   app = Celery('tasks', backend='rpc://', broker='pyamqp://guest@192.168.1.113//')

⇨   @app.task
    def add(x, y):
      return list(np.array(x) + np.array(y))
```

Fonte: Produção do próprio autor.

No celery, os serviços (também denominados *workers*) e seus usuários se comunicam por meio de troca de mensagens, mediadas por um *broker*. Para se comunicar com um serviço, um cliente adiciona uma mensagem na fila do *broker*, que por sua vez, a redireciona para o serviço em questão (CELERY, 2017b). Em suma, o celery exige a presença de um *broker*

para funcionar. O recomendado em sua documentação é o RabbitMQ (RABBITMQ, 2017), que possui uma imagem Docker. Assim, é possível utilizá-lo simplesmente, por exemplo, executando-se o comando mostrado na Figura 24.

Figura 24 – Implantando um broker Rabbitmq com o Docker

```
docker run -d --name rabbitmq
              --network host rabbitmq:3.6-alpine
```

Fonte: Produção do próprio autor.

Ao configurar a rede do contêiner como *host*, será atribuído a ele o mesmo endereço IP da máquina física na qual o Docker está sendo executado. Assim, é necessário garantir que o IP no código do serviço mostrado na Figura 23 (192.168.1.113) seja o do *broker*.

Por fim, é necessário desenvolver o código cliente, ou seja, que requisitará o serviço. A Figura 25 mostra um exemplo de código cliente, que faz um pedido para o serviço *add*, passando dois vetores, e então imprime o resultado.

Figura 25 – Código que faz uma requisição para o serviço criado

```
from tasks import *
print add.delay([1,2,3],[2,3,4]).get()
```

Fonte: Produção do próprio autor.

4.3 Preparando a imagem do serviço

Desenvolvida a aplicação, basta agora que um novo contêiner contendo as dependências e a aplicação seja criado, de modo que ela seja implantada na plataforma. Tal contêiner pode ser facilmente criado utilizando-se um Dockerfile na qual a imagem base é a que já foi criada anteriormente, como mostrado na Figura 26.

Figura 26 – Dockerfile para criação do contêiner contendo o serviço criado

```
FROM picoreti/py-devel:0.1

COPY tasks.py /opt/
WORKDIR /opt/
CMD celery -A tasks worker --loglevel=info
```

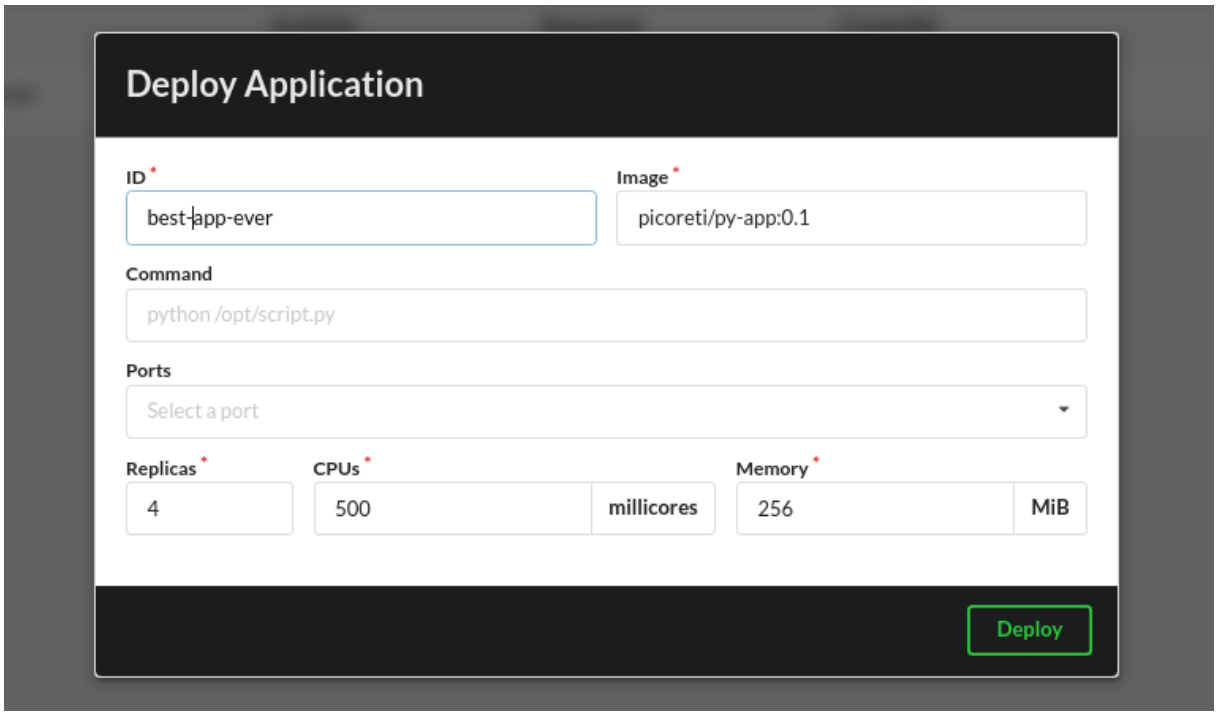
Fonte: Produção do próprio autor.

Por fim, a imagem criada deve ser enviada a um repositório de modo que a plataforma tenha acesso a mesma. Nesse caso, o repositório de imagens utilizado é o Docker Hub. Para conseguir enviar imagens para o Docker Hub o desenvolvedor deve possuir uma conta.

4.4 Implantação do serviço na plataforma

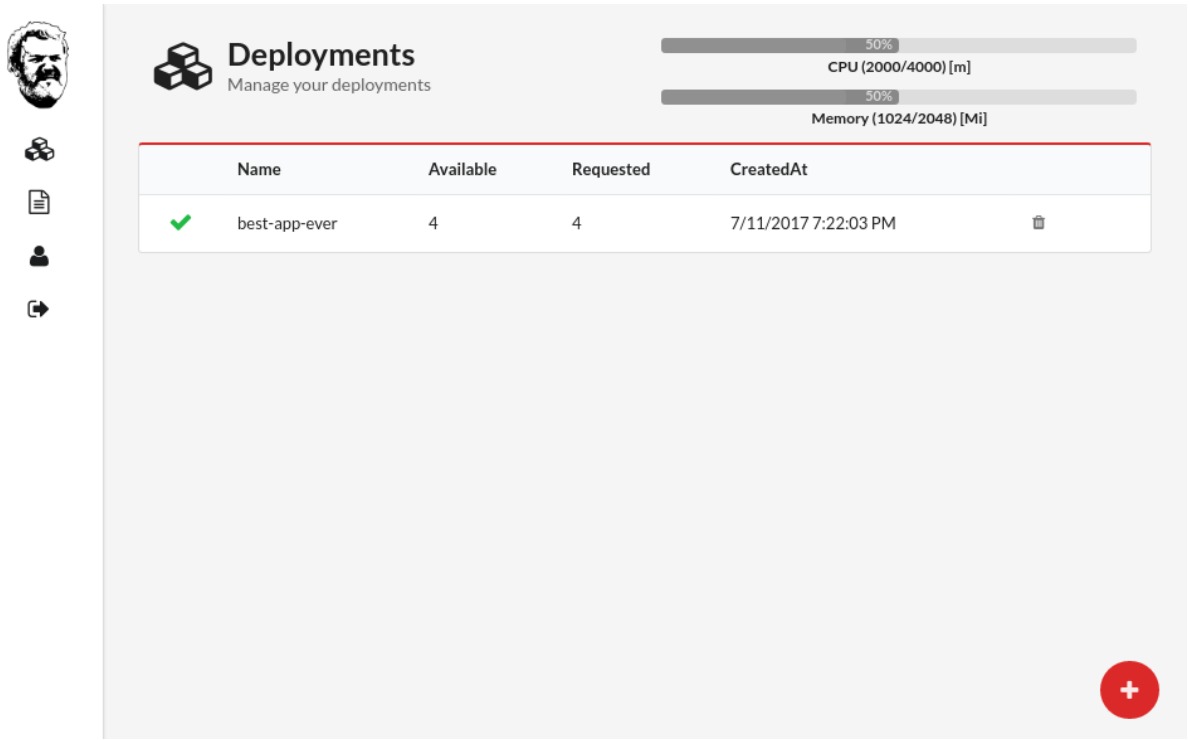
Com a imagem do serviço criada e presente no repositório da plataforma, é possível agora implantá-la na plataforma. Para tal, é necessário que o usuário possua um *token* de acesso (dado a ele por um administrador da plataforma). Então, basta que o usuário preencha o formulário de implantação como mostrado na Figura 27. Nela o usuário está implantando no sistema 4 instancias de sua aplicação sendo que cada uma delas utilizará uma quota de 500 millicores e 256 MB de RAM. A Figura 28 mostra o painel de gerenciamento de aplicações de um usuário com uma quota de 4000 millicores e 2048 MB após a implantação do formulário da Figura 27.

Figura 27 – Formulário para implantação da aplicação criada



Fonte: Produção do próprio autor.

Figura 28 – Visão do paínel de gerenciamento após a implantação da aplicação



Fonte: Produção do próprio autor.

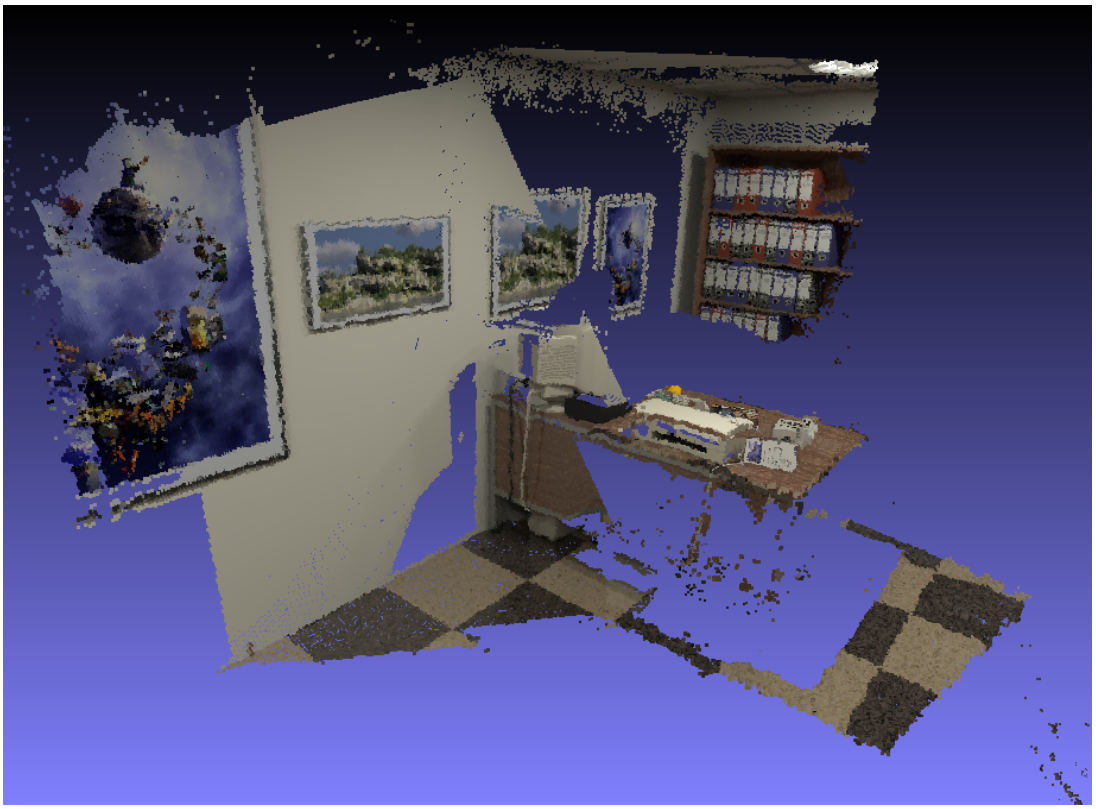
5 EXPERIMENTOS E RESULTADOS

Para validar a plataforma desenvolvida, foram realizados experimentos utilizando uma aplicação paralelizável com alto custo computacional desenvolvida por (SILVA; LUBE; VASSALLO, 2016). A aplicação implementa um algoritmo de visão computacional conhecido como reconstrução 3D, que processa um conjunto de imagens e estima a informação tridimensional dos *pixels*. O processo de reconstrução é executado em pares de imagens de modo independente, tornando possível a execução de cada par em paralelo.

Por meio da utilização de um *middleware* desenvolvido em (QUEIROZ, 2016), a aplicação foi reescrita seguindo uma arquitetura orientada a serviços. Dessa forma, o algoritmo de reconstrução passou a ser um serviço *stateless* que aceita um par de imagens e retorna uma estimativa da informação 3D dos *pixels*. Um contêiner Docker com esse serviço foi criada de modo a implantá-lo na plataforma desenvolvida.

Foi desenvolvida também uma aplicação responsável por ler as imagens do disco, requisitar a execução do serviço criado para cada par de imagens e então combinar seus resultados produzindo uma nuvem de pontos tridimensionais, como apresentado na Figura 29.

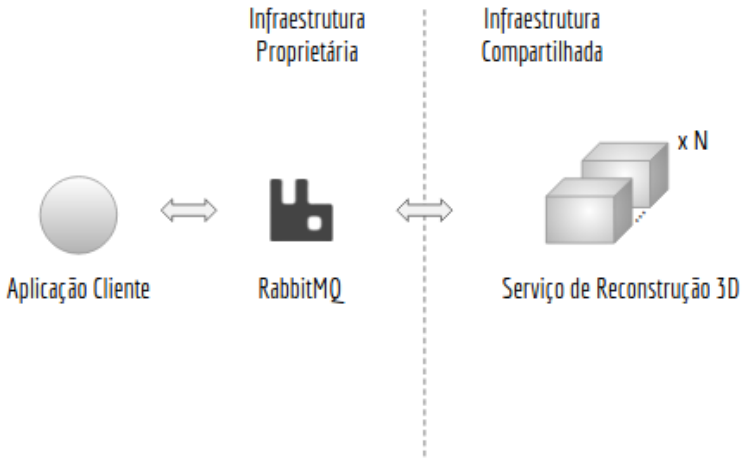
Figura 29 – Resultado do processo de reconstrução 3D



Fonte: Silva, Lube e Vassallo (2016).

O *middleware* utilizado é capaz de balancear os pedidos realizados aos serviços de reconstrução caso mais de uma instância desse esteja disponível. Além disso, ele requer a presença de um *broker* AMQP (*Advanced Message Queuing Protocol*) para funcionar adequadamente, portanto utilizou-se o RabbitMQ. A Figura 30 apresenta o diagrama da arquitetura utilizada nos testes, note que essa segue a arquitetura recomendada no Capítulo 4.

Figura 30 – Arquitetura utilizada para os testes de validação da plataforma



Fonte: Produção do próprio autor.

Na Figura 30, são mostrados dois tipos de infraestrutura: uma proprietária (sob o controle do usuário) e outra compartilhada (infraestrutura referente à plataforma proposta). Conforme já citado, na infraestrutura compartilhada são executados apenas os serviços *stateless* criados pelo usuário, nesse caso, o serviço de reconstrução 3D. Isso, pois, a infraestrutura em questão tem baixa confiabilidade, portanto, o estado (e.g. a lista de tarefas já concluídas) poderia ser perdido caso algum nó se tornasse indisponível. Assim, todo o estado é delegado para aplicações executadas na infraestrutura proprietária, na qual o usuário tem controle.

Com a aplicação base definida, foram realizados quatro experimentos com o objetivo de validar a arquitetura desenvolvida. Para tal, os resultados dos experimentos devem mostrar que:

- quando um usuário físico acessa uma máquina, ela se torna indisponível para a plataforma[1];
- a tarefa consegue ser concluída mesmo com nós se tornando indisponíveis no decorrer da mesma;

[1] Um nó pode tornar-se indisponível devido a alguma falha ou quando um usuário físico acessa a máquina

- o limite de recursos por usuário é respeitado;
- a plataforma suporta mais de um usuário.

Os experimentos foram divididos em: teste de execução para um único usuário (Seção 5.1), teste de execução para um único usuário com falhas (Seção 5.2), teste de execução para dois usuários com falhas (Seção 5.3) e teste de detecção de usuário físico (Seção 5.4)

Os resultados foram avaliados em termos de uso de *CPU* (média em 1 minuto) e número de instâncias do serviço. Essas métricas foram obtidas utilizando o próprio sistema de monitoramento da plataforma, com uma taxa de amostragem de 1 minuto (menor taxa de amostragem suportada pelo Heapster).

Durante a execução dos experimentos cinco máquinas foram utilizadas. O Quadro 1 apresenta a especificação de cada uma. Além disso, as diversas versões dos softwares e imagens de contêineres utilizados são mostradas no Quadro 2.

Quadro 1 – Especificação das maquinas utilizadas

Hostname	CPU					RAM (GB)
	Qtd.	Tipo	Freq. (GHz)	Núcleos/Threads	L3 (MB)	
Edge	2	Xeon E5504	2.00	8/8	4	4
Ninja	1	i5-3570	3.40	4/4	6	16
Jonathan	1	i5-4460S	2.90	4/4	6	8
Penguin	1	i5-4460S	2.90	4/4	6	8

Fonte: Produção do próprio autor.

Quadro 2 – Versões dos softwares e contêineires utilizados

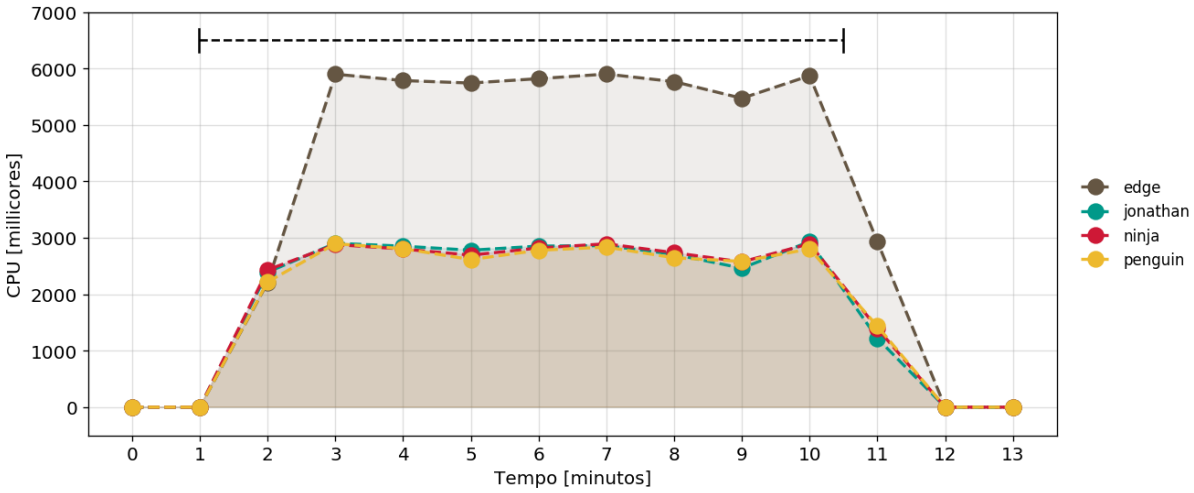
Software	Versão
Ubuntu	16.04.2
Docker	17.05.0-ce
Kubernetes	1.6.4
Contêineres	Imagem
Grafana	gcr.io/google_containers/heapster-grafana-amd64:v4.0.2
InfluxDB	gcr.io/google_containers/heapster-influxdb-amd64:v1.1.1
Heapster	gcr.io/google_containers/heapster-amd64:v1.4.0
RabbitMQ	picoreti/rabbitmq:latest

Fonte: Produção do próprio autor.

5.1 Teste de execução para um único usuário

Inicialmente, foi realizado um teste com apenas um usuário, no qual 15 instâncias do serviço foram alocadas. À cada instância, foi reservado um limite de 1 *CPU*. Os nós do *cluster* estavam disponíveis durante todo a duração do experimento, que durou 9 minutos e 31 segundos. A Figura 31 mostra o uso de *CPU* em cada nó do *cluster* referente aos serviços de reconstrução 3D. A linha tracejada horizontal na parte superior do gráfico indica a duração do experimento.

Figura 31 – Uso de *CPU* dos serviços de reconstrução 3D por nó



Fonte: Produção do próprio autor.

Como pode-se observar na Figura 31, todos os recursos das máquinas da infraestrutura alocados para as instâncias desse serviço foram utilizados sem interrupção. Nesse teste, foi possível constatar o adequado funcionamento da plataforma e mensurar o tempo necessário para execução da tarefa quando o sistema está disponível todo tempo.

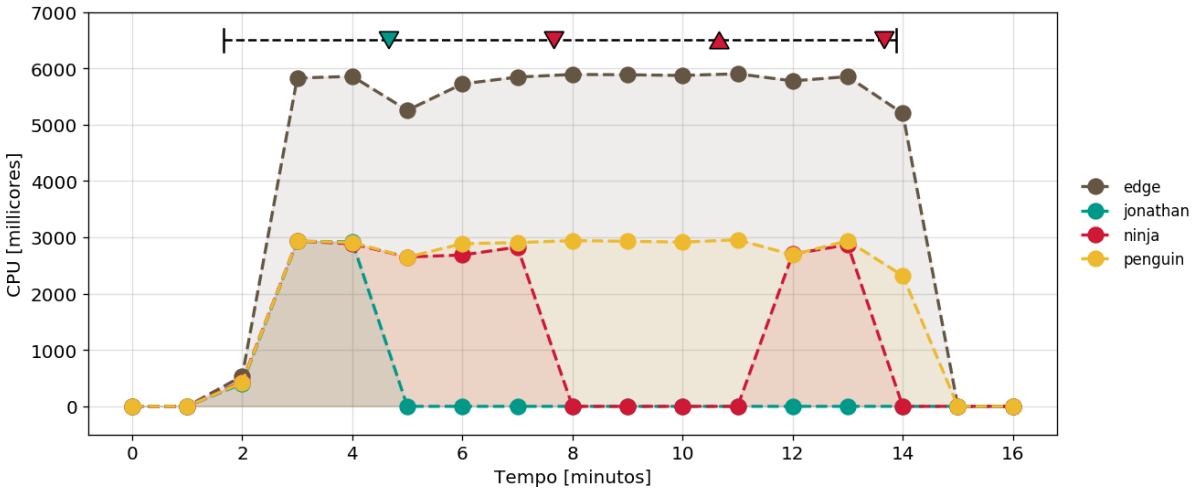
5.2 Teste de execução para um único usuário com falhas

Este teste é similar ao apresentado na Seção 5.1, porém agora deseja-se testar o efeito da indisponibilidade da infraestrutura na execução do experimento. Para tal, um nó aleatório tem seu estado modificado a cada 3 minutos, ou seja, caso esteja disponível, se tornará indisponível e vice-versa. A linha horizontal tracejada, presente na parte superior das figuras dessa seção, indica quando um nó tornou-se disponível (seta para cima) ou indisponível (seta para baixo). Além disso, indica a duração do experimento, que foi de 12 minutos e 13 segundos.

A Figura 32 mostra o uso de *CPU* em cada nó do *cluster* referente aos serviços de reconstrução 3D. Já a Figura 33 apresenta o número de instâncias desse serviço alocadas

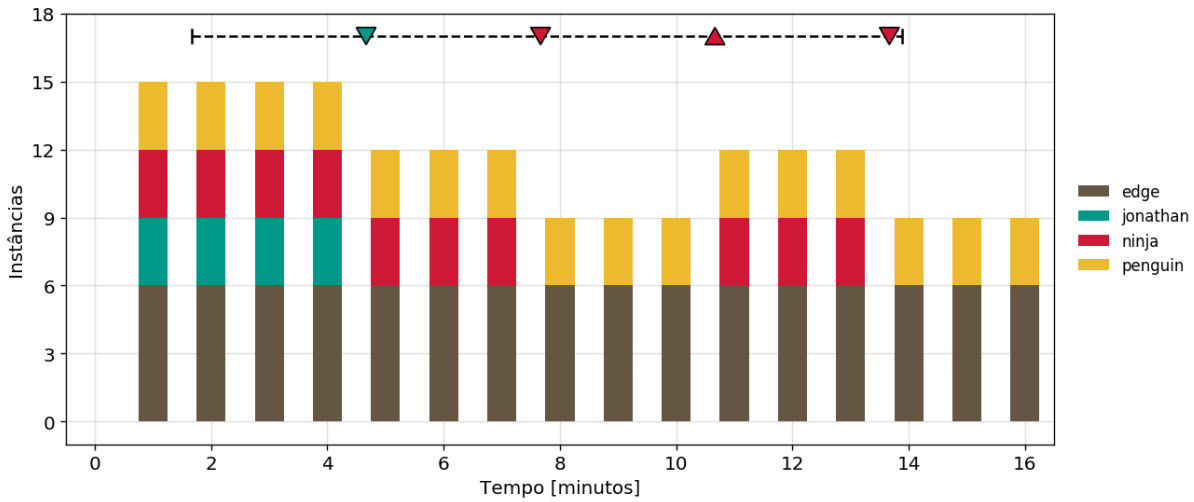
em cada nó. Por fim, a Figura 34 mostra a razão entre a *CPU* requisitada por todos os contêineres alocados nos nós do *cluster* e sua capacidade.

Figura 32 – Uso de *CPU* dos serviços de reconstrução 3D por nó



Fonte: Produção do próprio autor.

Figura 33 – Número de instâncias do serviço de reconstrução 3D por nó

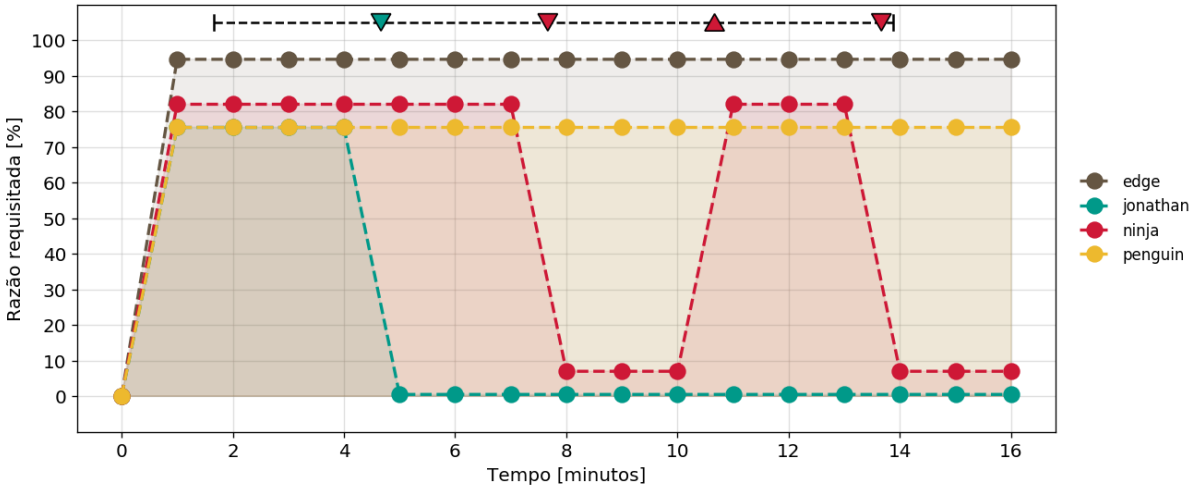


Fonte: Produção do próprio autor.

Na Figura 33, pode-se notar que, próximo ao minuto 4, o nó Jonathan torna-se indisponível. Nesse caso, suas instâncias poderiam ser realocadas para outro nó, entretanto, como pode ser visto na Figura 34, nenhum nó apresentava recursos suficientes. Assim, o processamento continuou, porém agora com um número de instâncias inferior ao requisitado pelo usuário, afetando a duração total do experimento.

Pode-se notar ainda na Figura 33, que o nó ninja torna-se disponível próximo ao minuto 11. Assim, o escalonador percebe essa mudança e aloca novamente três instâncias do serviço nesse nó.

Figura 34 – Razão entre *CPU* total reservada e a capacidade do nó



Fonte: Produção do próprio autor.

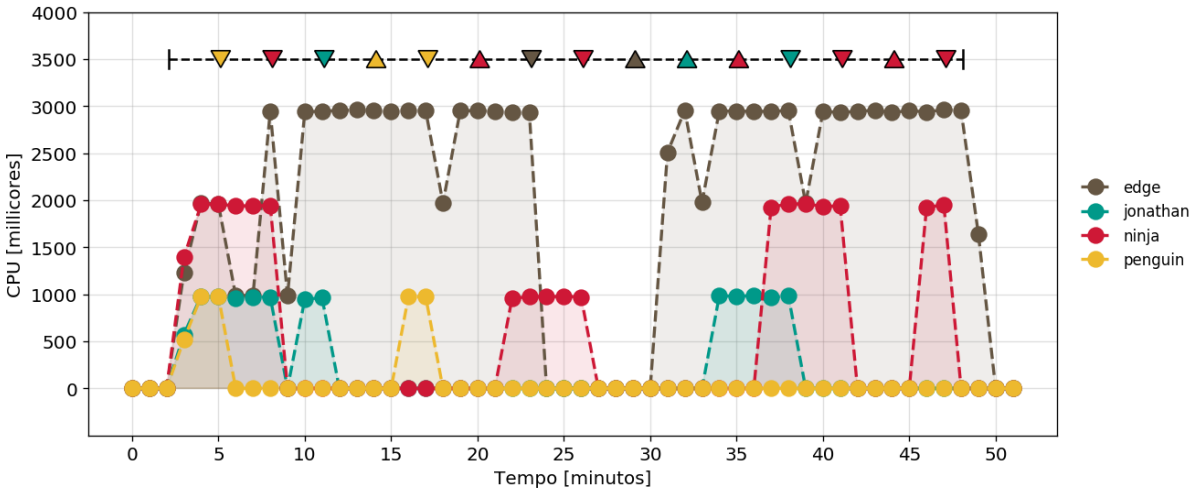
Vale ressaltar que os elementos da plataforma, como o sistema de monitoramento, também são executados dentro de contêineres. Por isso, esses elementos também consomem parte dos recursos do *cluster*. Esse efeito pode ser observado na Figura 34, em que o nó Edge apresenta uma razão de 95%, apesar de possuir apenas 6 instâncias do serviço (o que corresponderia a uma razão de 75%).

5.3 Teste de execução para dois usuários com falhas

De modo a validar a capacidade da plataforma de suportar vários usuários, um teste com dois usuários foi realizado. Nele, cada usuário possui uma quota máxima de 6 *CPUs*, sendo que a cada instância do serviço de reconstrução 3D foi atribuída uma quota de 1 *CPU*. Além disso, simulou-se a existência de falhas periódicas assim como no teste apresentado na Seção 5.2. A duração deste experimento para o usuário 1 foi de 45 minutos e 57 segundos e 42 minutos e 46 segundos para o usuário 2.

A Figuras 35 e 36 mostram o uso de *CPU* em cada nó do *cluster* referente aos serviços de reconstrução 3D para os distintos usuários. Já as Figuras 37 e 38 apresentam o número de instâncias desse serviço alocadas em cada nó para os distintos usuários. Por fim, a Figura 39 mostra a razão entre a *CPU* requisitada por todos os contêineres alocados nos nós do *cluster* e sua capacidade.

Figura 35 – Uso de *CPU* dos serviços de reconstrução 3D por nó - Usuário 1

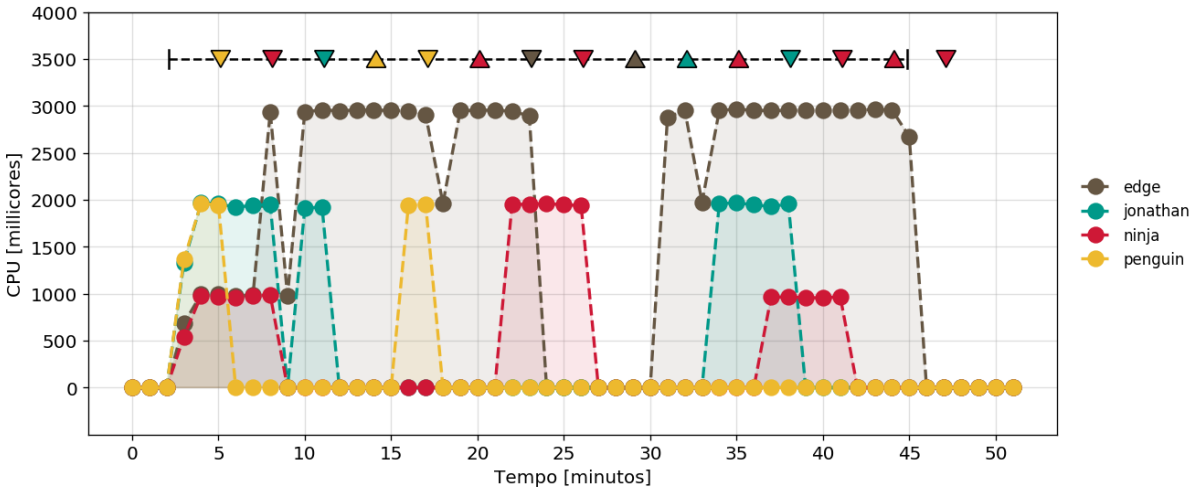


Fonte: Produção do próprio autor.

Pode-se notar na Figura 39 que inicialmente o *cluster* possui uma capacidade superior a utilizada pelos dois usuários em conjunto. Portanto, quando o nó Penguin torna-se indisponível próximo ao minuto 5, as instâncias dos serviços são rapidamente reescalonadas para o Edge, mantendo assim a quantidade de instâncias desejadas pelos usuários.

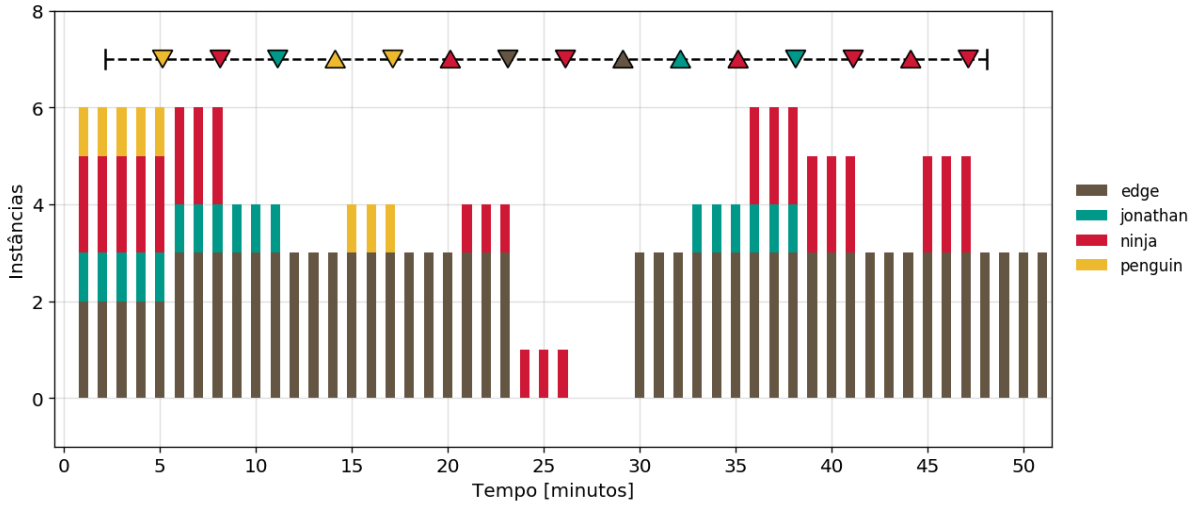
Um dos diferenciais desse teste, é que houveram muitos casos em que os nós tornaram-se indisponíveis, chegando até o ponto de toda a infraestrutura tornar-se indisponível (próximo ao minuto 27). Mesmo assim, todas as tarefas foram concluídas eventualmente, pois conforme os nós foram se tornando disponíveis novamente, as instâncias foram sendo reescalonadas, como esperado.

Figura 36 – Uso de *CPU* dos serviços de reconstrução 3D por nó - Usuário 2



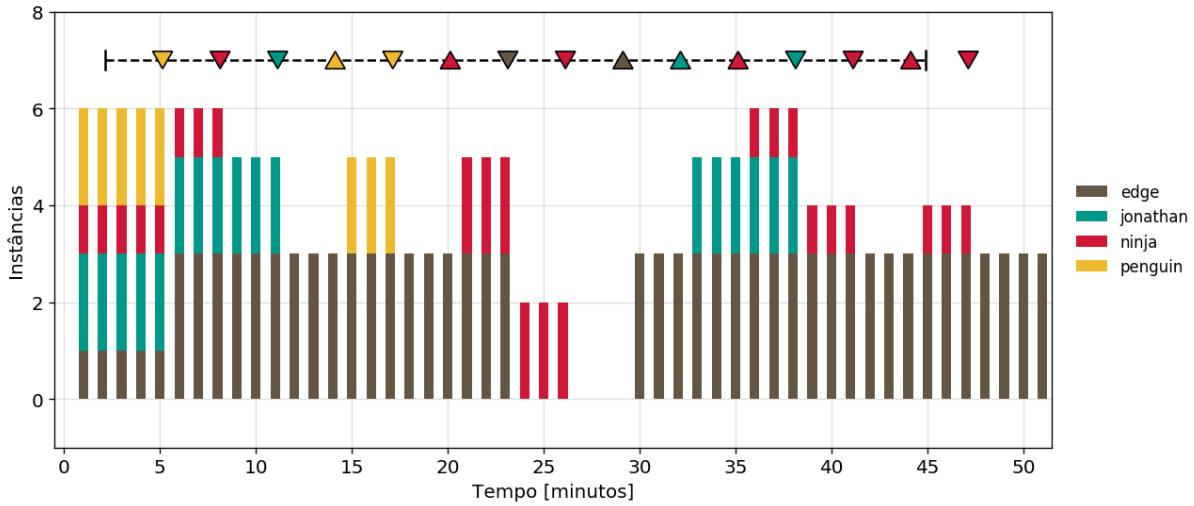
Fonte: Produção do próprio autor.

Figura 37 – Número de instâncias do serviço de reconstrução 3D no tempo do usuário 1



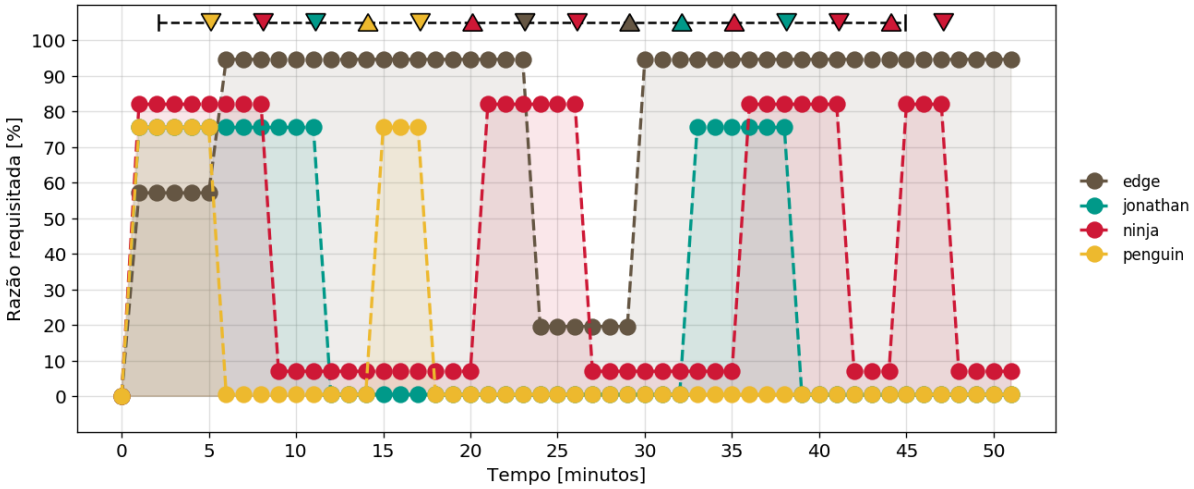
Fonte: Produção do próprio autor.

Figura 38 – Número de instâncias do serviço de reconstrução 3D no tempo do usuário 2



Fonte: Produção do próprio autor.

Figura 39 – Razão entre *CPU* reservada e a capacidade do nó no tempo



Fonte: Produção do próprio autor.

5.4 Teste de detecção de usuário físico

Um dos diferenciais da plataforma proposta é sua capacidade de utilizar o tempo ocioso de computadores de uso geral no processamento das diversas aplicações nela implantadas. Além disso, a plataforma deve garantir que a utilização desses computadores para sua designação principal seja priorizada. Assim, ela conta com um elemento capaz de detectar a presença de um usuário físico.

Este teste tem como objetivo mostrar o funcionamento desse elemento. Para tal, implantou-se em um nó específico da infraestrutura uma aplicação que consome constantemente 100 millicores de *CPU*. Inicialmente, um usuário físico estava utilizando o nó em questão, assim, nenhuma aplicação foi alocada nele.

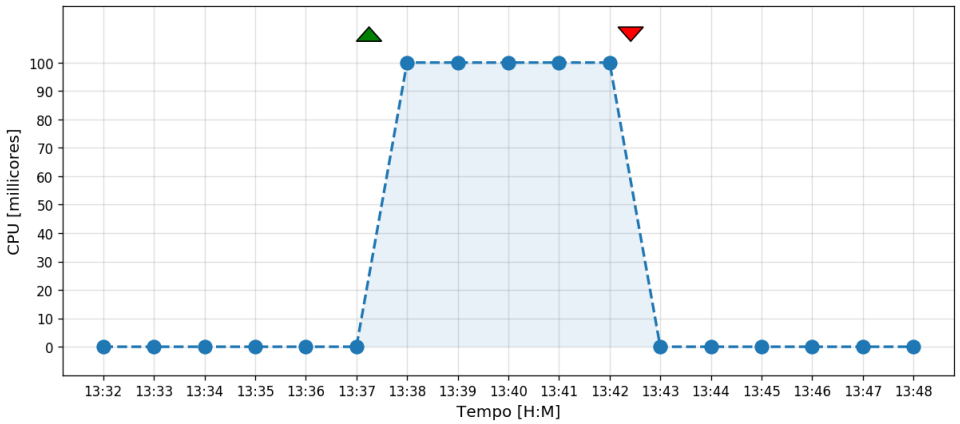
Assim que o usuário efetua o *logout*, o sistema percebe que o nó se tornou disponível e então aloca a aplicação nele. A aplicação roda livremente até que algum usuário efetue novamente um *login*. O comportamento descrito acima pode ser observado nas Figuras 41 e 40 que mostram respectivamente o uso de *CPU* de contêineres no nó em questão e os registros do elemento do sistema que efetua a detecção de usuários físicos.

Figura 40 – Registros do elemento de detecção de usuários físicos

```
0|daemon    | Running...
0|daemon    | [2017-07-12T12:47:24.154Z] Sending cordon action...
0|daemon    | [2017-07-12T12:48:14.166Z] Sending uncordon action...
0|daemon    | [2017-07-12T12:51:14.219Z] Sending cordon action...
0|daemon    | [2017-07-12T13:01:44.370Z] Sending uncordon action...
0|daemon    | [2017-07-12T13:08:44.484Z] Sending cordon action...
0|daemon    | [2017-07-12T13:37:14.976Z] Sending uncordon action...
0|daemon    | [2017-07-12T13:42:25.069Z] Sending cordon action...
```

Fonte: Produção do próprio autor.

Figura 41 – Uso de *CPU* de um nó durante o teste



Fonte: Produção do próprio autor.

6 CONCLUSÃO

Neste trabalho foi desenvolvida uma plataforma de processamento paralelo distribuído capaz de aproveitar o tempo ocioso de computadores de uso geral para acelerar aplicações com alta necessidade de processamento. Seu diferencial em relação a plataformas com o mesmo objetivo, é o emprego de tecnologias modernas utilizadas na computação em nuvem como virtualização por contêineres e uma arquitetura orientada a serviços.

A plataforma proposta foi testada utilizando-se uma aplicação paralelizável com alto custo computacional. Essa aplicação foi transformada em um serviço *stateless* e foi avaliada nos testes realizados em termos de *CPU* utilizada e o número de instâncias em cada nó do *cluster*.

Os testes mostraram que a aplicação escolhida conseguiu ser executada com sucesso, mesmo não havendo nenhuma garantia de disponibilidade, característica de uma infraestrutura compartilhada. Notou-se que a indisponibilidade dos nós afetou apenas a duração total do experimento, sendo verdade até no caso em que toda a infraestrutura esteve indisponível para a aplicação.

Além disso, foi possível ver que o escalonador tenta sempre garantir que a quantidade de instâncias requisitadas pelos usuários esteja sendo executada na infraestrutura. Assim, quando um nó torna-se indisponível, as instâncias dos serviços são reescalonadas para outro nó, quando o *cluster* apresenta capacidade suficiente.

Foi mostrado também que a plataforma é capaz de atender a múltiplos usuários simultaneamente, definindo para cada um deles uma quota máxima de recursos. Por último, mostrou-se que quando um usuário físico acessa uma máquina, ela torna-se indisponível para a plataforma, já que a utilização dos computadores para sua designação original deve ser priorizada.

A plataforma alcançou todos os objetivos aqui propostos, mas, como comentado, ela exige que parte da aplicação (a parte que possui estado) seja executada em uma infraestrutura própria do usuário, uma vez que a mesma requer que os serviços sejam *stateless*.

Assim, como trabalhos futuros, propõe-se permitir que o usuário seja capaz de executar toda sua aplicação na infraestrutura compartilhada. Para tal, é necessário que a plataforma forneça um mecanismo confiável no qual o usuário possa persistir dados. Outra alternativa, é adicionar à plataforma a capacidade de fazer *checkpointing* e migração de contêineres, fazendo assim com que ela seja capaz de migrar contêineres de uma máquina indisponível

para outra parte da infraestrutura e, consequentemente, permitir a seus usuários a criação de serviços com estado. Propõe-se também, adicionar novos tipos de recursos à plataforma, como armazenamento, *GPU*, entre outros. Por fim, poderia-se alterar o algoritmo de escalonamento para permitir que aplicações de usuários remotos e físicos possam coexistir sem comprometer a função principal da infraestrutura.

REFERÊNCIAS BIBLIOGRÁFICAS

ALPINE. *Alpine Linux*. 2017. Disponível em: <https://alpinelinux.org/about/>. Acesso em: 2017-6-30. Citado na página 37.

ALWABEL, A.; WALTERS, R. J.; WILLS, G. B. A view at desktop clouds. 2014. Citado na página 12.

ANDERSON, D. P. Boinc: A system for public-resource computing and storage. In: IEEE. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. [S.l.], 2004. p. 4–10. Citado na página 13.

CELERY. *Celery: Distributed Task Queue*. 2017. Disponível em: <http://www.celeryproject.org/>. Acesso em: 2017-6-30. Citado na página 38.

CELERY. *Introduction to Celery*. 2017. Disponível em: <http://docs.celeryproject.org/en/latest/getting-started/introduction.html>. Acesso em: 2017-6-30. Citado na página 41.

COULOURIS, G. et al. *Distributed Systems - Concepts and Design*. Quinta edição. [S.l.]: Addison-Wesley, 2012. Citado 2 vezes nas páginas 11 e 15.

DISTRIBUTED.NET. *distributed.net History and Timeline*. 2015. Disponível em: <http://www.distributed.net/History>. Acesso em: 2016-11-04. Citado 2 vezes nas páginas 11 e 13.

DOCKER. *Customers*. 2017. Disponível em: <https://www.docker.com/customers>. Acesso em: 2017-6-30. Citado na página 28.

DOCKER. *Docker Hub*. 2017. Disponível em: <https://hub.docker.com/>. Acesso em: 2017-6-30. Citado 2 vezes nas páginas 29 e 36.

DOCKER. *Docker Swarm Mode Overview*. 2017. Disponível em: <https://docs.docker.com/engine/swarm/>. Acesso em: 2017-6-30. Citado na página 29.

DOCKER. *Dockerfile Reference*. 2017. Disponível em: <https://docs.docker.com/engine/reference/builder/>. Acesso em: 2017-6-30. Citado na página 28.

DOCKER. *What is a container*. 2017. Disponível em: <https://www.docker.com/what-container>. Acesso em: 2017-6-30. Citado na página 28.

ENDREI, M. et al. Patterns: Service-Oriented Architecture and Web Services. *Contract*, v. 1, p. 17–42, 2004. Disponível em: <http://www.chinagrid.net/grid/paperppt/Patterns-Services.pdf>. Citado na página 23.

ERL, T. *SOA Principles of Service Design*. Primeira edição. [S.l.]: Prentice Hall, 2007. ISBN 0132344823. Citado na página 23.

ERL, T.; MAHMOOD, Z.; PUTTINI, R. *Cloud Computing - Concepts, Technology and Architecture*. Primeira edição. [S.l.]: Prentice Hall, 2013. ISBN 0133387526. Citado 3 vezes nas páginas 11, 19 e 20.

FEDAK, G. et al. Xtremweb: A generic global computing system. In: IEEE. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*. [S.l.], 2001. p. 582–587. Citado na página 13.

FERREIRA, L. et al. Introduction to Grid Computing with Globus. *Contract*, p. 268, 2003. ISSN 1524-4539. Citado na página 19.

GRAFANA. *Grafana - The open platform for beautiful analytics and monitoring*. 2017. Disponível em: <https://grafana.com/>. Acesso em: 2017-6-30. Citado na página 30.

IEEE. *History of Ethernet*. 2013. Disponível em: <http://standards.ieee.org/events/ethernet/history.html>. Acesso em: 2016-11-04. Citado na página 11.

INFLUXDATA. *InfluxDB is the Time Series Database in the TICK stack*. 2017. Disponível em: <https://www.influxdata.com/time-series-platform/influxdb/>. Acesso em: 2017-6-30. Citado na página 30.

IZRAILEVSKY, Y. *Completing the Netflix Cloud Migration*. 2016. Disponível em: <https://media.netflix.com/pt_br/company-blog/completing-the-netflix-cloud-migration>. Acesso em: 2017-6-30. Citado na página 12.

JWT. *JSON Web Tokens*. 2017. Disponível em: <https://jwt.io/>. Acesso em: 2017-6-30. Citado na página 33.

KAHANWAL, B.; SINGH, T. P. The Distributed Computing Paradigms: P2P, Grid, Cluster, Cloud, and Jungle. v. 1, n. 2, p. 183–187, 2012. Citado na página 18.

KSHEMKALYANI, A. D.; SINGHAL, M. *Distributed Computing*. [S.l.]: Cambridge University Press, 2008. 736 p. ISBN 9780521876346. Citado na página 20.

KUBERNETES. *Heapster: Compute Resource Usage Analysis and Monitoring of Container Clusters*. 2017. Disponível em: <https://github.com/kubernetes/heapster>. Acesso em: 2017-6-30. Citado na página 30.

KUBERNETES. *Kubernetes API Reference Docs*. 2017. Disponível em: <https://kubernetes.io/docs/api-reference/v1.6/>. Acesso em: 2017-6-30. Citado na página 31.

KUBERNETES. *Kubernetes Architecture*. 2017. Disponível em: <https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture.md>. Acesso em: 2017-6-30. Citado 2 vezes nas páginas 24 e 29.

KUBERNETES. *Kubernetes cluster monitoring addon on GitHub*. 2017. Disponível em: <https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/cluster-monitoring/influxdb>. Acesso em: 2017-6-30. Citado na página 30.

MACKENZIE, C. M. et al. Reference Model for Service Oriented Architecture 1.0. p. 8, 2006. Citado na página 23.

MCGILVARY, G. A. et al. V-boinc: The virtualization of boinc. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. [S.l.], 2013. p. 285–293. Citado na página 13.

MELL, P.; GRANCE, T. The NIST definition of cloud computing. p. 2, 2011. Citado na página 20.

MELL, P.; GRANCE, T. The NIST definition of cloud computing. p. 2–3, 2011. Citado na página 20.

MESOS. *Apache Mesos. A distributed systems kernel*. 2017. Disponível em: <http://mesos.apache.org/>. Acesso em: 2017-6-30. Citado na página 29.

METEOR. *Meteor: the fastest way to build javascript apps*. 2017. Disponível em: <https://www.meteor.com/>. Acesso em: 2017-6-30. Citado na página 31.

QUEIROZ, F. M. de. *Desenvolvimento da Infraestrutura de um Espaço Inteligente baseado em Visão Computacional e IoT*. 2016. Citado na página 45.

RABBITMQ. *Messaging that just works*. 2017. Disponível em: <https://www.rabbitmq.com/>. Acesso em: 2017-6-30. Citado na página 42.

RKT. *Rkt, A security-minded, standards-based container engine*. 2017. Disponível em: <https://coreos.com/rkt>. Acesso em: 2017-6-30. Citado na página 28.

SCHANTZ, R. E.; SCHMIDT. Middleware for Distributed Systems. 2007. Citado na página 15.

SETI@HOME. *The science of SETI@home*. 2011. Disponível em: <http://setiathome.berkeley.edu/sah_about.php>. Acesso em: 2016-11-04. Citado na página 11.

SILVA, L. d. A.; LUBE, J. G. P.; VASSALLO, R. F. Aproximação Planar por Partes para Reconstrução 3D Densa. In: *Anais do XXI Congresso Brasileiro de Automática (CBA'2016)*. [S.l.: s.n.], 2016. Citado na página 45.