UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM REDES DE COMPUTADORES

EVERTON CUNICO

APLICAÇÃO JAVA PARA TRANSFERÊNCIA DE ARQUIVOS UTILIZANDO MICROSSERVIÇOS REST E CONTÊINER

MONOGRAFIA DE ESPECIALIZAÇÃO

PATO BRANCO
2018

EVERTON CUNICO

APLICAÇÃO JAVA PARA TRANSFERÊNCIA DE ARQUIVOS UTILIZANDO MICROSSERVIÇOS REST E CONTÊINER

Monografia de especialização apresentada ao III Curso de Especialização em Redes de Computadores – Configuração e Gerenciamento de Servidores e Equipamentos de Rede, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Especialista.

Orientador: M.Eng. Anderson Luiz Fernandes

PATO BRANCO
2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
III Curso de Especialização em Redes de Computadores

TERMO DE APROVAÇÃO

APLICAÇÃO JAVA PARA TRANSFERÊNCIA DE ARQUIVOS UTILIZANDO MICROSSERVIÇOS REST E CONTÊINER

por

Everton Cunico

Esta monografia foi apresentada às 19h30min do dia 11 de dezembro de 2018, como requisito parcial para obtenção do título de ESPECIALISTA, no III Curso de Especialização em Redes de Computadores – Configuração e Gerenciamento de Servidores e Equipamentos de Redes, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho **aprovado**.

Prof. M. Eng. Anderson Luiz Fernandes
Orientador / Faculdade Mater Dei

Prof. M. Eng. Vinícius Pegorini
UTFPR-PB

Prof. Dr. Fábio Favarim
UTFPR-PB

Prof. Dr. Eden Ricardo Dosciatti
UTFPR-PB

Prof. Dr. Fábio Favarim
Coordenador do III Curso de Especialização
em Redes de Computadores

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

Dedico este trabalho à minha família, a Deus e a todos os que de alguma forma contribuíram para que eu pudesse chegar até aqui. Dedico também este trabalho a todos os que acreditam no meu potencial, e que depositam em mim suas expectativas.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me dar a oportunidade de estar aqui, por me dar a saúde e o discernimento necessários para que eu pudesse desenvolver este trabalho.
Agradeço à minha esposa Aline, por me incentivar, estar do meu lado incondicionalmente, por me dar apoio e me fazer acreditar que conseguiria.
Agradeço aos meus pais e a empresa onde atualmente trabalho, IDS Software, pelo aparato financeiro para conseguir arcar com os custos do curso.
Agradeço ao meu orientador MEng. Anderson Luiz Fernandes, pela sabedoria e paciência com que me guiou nesta trajetória.
Enfim, a todos os que por algum motivo contribuíram para a realização deste trabalho.

As pessoas costumam dizer que a motivação não dura sempre. Bem, nem o efeito do banho, por isso recomenda-se diariamente.

Zig Ziglar

RESUMO

CUNICO, Everton. Aplicação Java para gerenciamento de arquivos utilizando microsserviços REST e contêiner. 2018. 42 f. Monografia (Especialização em Redes de Computadores) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

Este projeto trata de uma abordagem sobre o novo paradigma para servir aplicações. Nele são desenvolvidos dois *webservices*, que são destinados a funcionarem em diferentes ambientes, sendo um deles o que é utilizado em larga escala no meio empresarial, que é o modelo monolítico. O outro *webservice* é destinado a funcionar em um *cluster* de contêineres gerido pela aplicação Kubernetes. Com isto, pretende-se demonstrar como este novo modelo pode ter melhor desempenho e estabilidade.

Palavras-chave: REST. Docker. Kubernetes. Webservice. Contêiner.

ABSTRACT

CUNICO, Everton. Aplicação Java para transferência de arquivos utilizando microsserviços REST e contêiner. 2018. 42 f. Monografia (Especialização em Redes de Computadores) – Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco. Pato Branco, 2018.

This project deals with an approach on the new paradigm to serve applications. In it are developed two webservices, which are designed to operate in different environments, one of which is used in large scale in the business environment, which is the monolithic model. The other webservice is intended to work in a container *cluster* managed by the Kubernetes application. With this, it is tried to demonstrate how this new model can have better performance and stability.

Keywords: REST. Docker. Kubernetes. Webservice. Container.

LISTA DE FIGURAS

Figura 1 – Comparação entre a arquitetura de máquina virtual e contêiner 19

Figura 2 – Comparação entre a arquitetura de microsserviços e a arquitetura monolítica 20

Figura 3 – Fluxo de desenvolvimento de aplicações personalizadas 30

Figura 4 – Configuração dos nós do *cluster* Kubernetes 35

Figura 5 – *Services* 35

Figura 6 – *Pods* 36

Figura 7 – *Deployments* 36

Figura 8 – Teste na aplicação monolítica 38

Figura 9 – Teste na aplicação em contêiner 38

LISTAGENS DE CÓDIGOS

Listagem de Código 1 - Código fonte dos métodos de *download* e *upload*............................33

Listagem **de Código 1 - Código fonte dos métodos de download e upload**istagem de Código 2 - *Dockerfile* 34

LISTA DE SIGLAS

API *Address Resolution Protocol*

DHCP *Infrastructure as a Service*

DNS *Basic Service Set*

FTP *File Transport Protocol*

HTTP *HyperText Transfer Protocol*

IAAS *Infraestructure as a Service*

LDAP *Lightweight Directory Access Protocol*

NIST *Access Point*

REST *Representational State Transfer*

SFTP *Secure File Transport Protocol*

URI *Platform as a Service*

VPN *Basic Service Area*

SUMÁRIO

1 INTRODUÇÃO **10**

1.1 OBJETIVOS 11

1.1.1 Objetivo Geral *11*

1.1.2 Objetivos Específicos *11*

1.2 JUSTIFICATIVA 11

1.3 ESTRUTURA DO TRABALHO 12

2 REFERENCIAL TEÓRICO **14**

2.1 CLOUD COMPUTING 14

2.1.1 Modalidades de Serviço *14*

2.2 VIRTUALIZAÇÃO 15

2.2.1 Hypervisors e Máquinas Virtuais *16*

2.2.2 Contêineres *17*

2.3 MICROSSERVIÇOS 18

2.4 ELASTICIDADE NA COMPUTAÇÃO EM NÚVEM 19

2.4.1 Classificação *20*

3 MATERIAIS E METODOLOGIA **22**

3.1 MATERIAIS 22

3.1.1 Java EE *22*

3.1.2 Rest *23*

3.1.3 Docker *24*

3.1.4 Kubernetes *25*

3.2 METODOLOGIA 28

3.2.1 Etapas do desenvolvimento de aplicações conteinerizadas *28*

4 RESULTADOS **31**

4.1 DESENVOLVIMENTO DA APLICAÇÃO 31

4.2 UTILIZAÇÃO DA APLICAÇÃO NO KUBERNETES 33

4.3 TESTES DE DESEMPENHO 36

5 CONCLUSÕES **39**

REFERÊNCIAS **40**

1 INTRODUÇÃO

Hoje, em determinada empresa, tem-se uma aplicação monolítica, desenvolvida na linguagem Java, e que executa no servidor de aplicação Apache Tomcat.

Nessa aplicação há diversos *servlets*, que provém diversos métodos inerentes ao seu intuito. Dentre esses métodos, um deles tem a função de permitir o download de arquivos de extensão .apk, que são aplicativos para dispositivos da plataforma Android.

Quando este método é requisitado com muita frequência, acaba sobrecarregando este servidor de aplicação, ocasionando falhas de conexão, além de gargalos de processamento, memória e rede no servidor.

A partir desta situação-problema, tem-se buscado soluções para obter maior estabilidade e desempenho nesta aplicação. Com este trabalho foi detalhada uma das soluções possíveis para tal, que é o desenvolvimento de uma nova aplicação, também escrita na linguagem Java, que provenha apenas a transferência de arquivos, isolando esta funcionalidade do restante da aplicação já existente.

Esta nova aplicação, utiliza de tecnologias e ferramentas que proporcionam meios para que se tenha maior simplicidade na escalabilidade, gerindo com maior eficiência os recursos do servidor conforme as demandas.

O projeto foi construído baseado na arquitetura de microsserviços, na qual pequenas partes de uma mesma aplicação podem executar em diferentes plataformas, diferentes servidores, comunicando-se através de requisições de rede para alcançar o objetivo comum.

Uma das ferramentas utilizadas é o Docker. Este *software* permite isolar determinada aplicação em contêineres. Dessa forma, todo o sistema e o aparato necessário para que o aplicativo possa funcionar, fica embutido em um contêiner, sem sofrer interferências externas de outros sistemas.

Para gerir estes contêineres e permitir escalabilidade, foi utilizada uma ferramenta de orquestração, o Kubernetes, que controla a localização de cada contêiner, faz a gerência da estabilidade, recompondo serviços que eventualmente sofram falha, e também a escalabilidade, replicando os contêineres de acordo com a demanda de requisições.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem por objetivo principal prover uma aplicação de transferência de arquivos em rede, no qual será possível realizar listagem, *upload* e *download* de arquivos entre cliente/servidor.

1.1.2 Objetivos Específicos

Com o desenvolvimento desta aplicação pretende-se alcançar alguns objetivos específicos, que são:

- Criar um *cluster* Kubernetes em alta disponibilidade e elástico que permita a gestão e *deployment* de contêineres;
- Identificar as vantagens e desvantagens que a conteinerização de aplicações pode trazer a uma organização que desenvolve *software*;
- Identificar os desafios e os problemas existentes no momento da adoção deste novo paradigma, mudando do modelo tradicional monolítico, assim como apresentar as soluções para esses problemas;
- Obter um servidor de arquivos baseado no protocolo HTTP[1].

1.2 JUSTIFICATIVA

Este trabalho é de suma importância para a resolução de uma situação com problemática real em uma empresa da cidade de Pato Branco – PR. Determinada aplicação dessa empresa possui como *front end* um aplicativo para a plataforma Android e como *back end* um *web service* na linguagem Java.

Este aplicativo, não está hospedado na Google Play Store, devido a questões de compatibilidade entre versões utilizadas por diferentes clientes. Dessa forma, novas compilações do aplicativo são disponibilizadas para cada cliente de forma individual, alocando o arquivo de instalação em uma pasta no servidor, a qual é acessada pelo *back end*

---

[1] HTTP: *Hyper Text Transfer Protocol*. Protocolo de comunicação entre sistemas.

da aplicação, que disponibiliza este arquivo aos dispositivos móveis através da arquitetura REST[2].

Porém, em clientes de maior porte, a quantidade de *downloads* simultâneos acaba sobrecarregando o servidor de aplicação, ocasionando travamentos, gargalos, e em casos mais graves, até quedas de todo o sistema.

Uma solução paliativa para esta questão é aumentar consideravelmente o poder de processamento e memória do servidor, permitindo alocar uma quantidade consideravelmente maior de recursos para o servidor de aplicação. Porém esta solução torna-se inviável, pois em boa parte do tempo, em um cenário "normal" de uso, estes recursos tornam-se ociosos.

Sendo assim, deve-se encontrar uma solução viável para este caso. Este trabalho visa disponibilizar uma aplicação isolada do *web service* principal, que permita aos aplicativos realizarem o *download* dos arquivos .apk.

Para isto, neste projeto Java, foi utilizada a arquitetura de microsserviços. A partir de então, sua implantação feita em contêineres no *software* Docker, gerido por um orquestrador, o Kubernetes, que realiza a gerência de recursos utilizados no servidor *cloud*, replicando os contêineres conforme a demanda de carga de CPU[3].

A contribuição deste trabalho é de suma importância na melhora da estrutura computacional dos aplicativos comerciais disponibilizados pela empresa, centralizando a disponibilização dos aplicativos para dispositivos móveis em um único ambiente que poderá ser acessado por todos os clientes, permitindo efetivar a entrega contínua de *software*.

1.3 ESTRUTURA DO TRABALHO

Este texto está organizado em capítulos, dos quais este é o primeiro e apresentou a ideia e o contexto do trabalho, incluindo os objetivos e a justificativa.

O Capítulo 2 contém o referencial teórico. O referencial teórico apresenta uma breve introdução aos assuntos abordados neste trabalho, os quais são necessários para compreender os conceitos inerentes ao objetivo do projeto.

No Capítulo 3 estão os materiais e a metodologia empregados no desenvolvimento deste trabalho. Neste capítulo estão explicadas as ferramentas utilizadas para que o objetivo

---

[2] REST: *Representational State Transfer*. Arquitetura que define conjunto de restrições e propriedades baseados em HTTP.

possa ser alcançado, bem como as técnicas empregadas e as dificuldades encontradas no desenvolvimento do trabalho.

O Capítulo 4 contém o resultado do desenvolvimento deste trabalho. Nele é apresentado um comparativo de desempenho de um *webservice* executando no modelo atual monolítico contra um mesmo *webservice* configurado para utilização em um *cluster* Kubernetes.

No Capítulo 5 é apresentada a conclusão com as considerações finais deste trabalho, bem como a opinião pessoal do autor a respeito da adoção deste novo paradigma para servir aplicações *web*.

[3] CPU: *Central Process Unit.* Unidade de Processamento central.

2 REFERENCIAL TEÓRICO

Neste capítulo são explicados os conceitos que serão abordados ao longo deste projeto. Inicialmente, será apresentado o conceito de *cloud computing*, bem como suas principais características. Após isto, se dará a abordagem da tecnologia de virtualização, além do aprofundamento para o conceito de conteinerização de aplicações. Em complemento, será explicado o conceito da arquitetura de microsserviços, bem como a elasticidade das aplicações desenvolvidas na mesma.

Ao fim deste capítulo, espera-se obter o conhecimento necessário para que o entendimento dos itens apresentados no escopo deste trabalho seja facilitado.

2.1 CLOUD COMPUTING

Cloud Computing, ou computação em nuvem, é um termo utilizado para descrever uma rede global de servidores, cada um deles exercendo uma função distinta. Costumeiramente denominada de *cloud*, ela não é uma entidade física, mas sim uma ampla rede de servidores remotos interligados em todo o mundo que devem funcionar como um ecossistema único (MICROSOFT, 2018).

Esses servidores foram concebidos com diferentes finalidades, como armazenar e gerir dados, executar aplicações ou até mesmo fornecer conteúdo ou serviços, como vídeos em *streaming*, *webmail*, *softwares* aplicativos ou comunicação social. Para o usuário final, ao invés deste acessar seus arquivos e aplicações armazenados em um computador local, ele os acessará a partir de uma conexão com a internet (MICROSOFT, 2018). Dessa forma, dispõe da liberdade de acesso, não dependendo mais de plataforma ou de *hardware* compatível.

2.1.1 Modalidades de Serviço

Segundo a NIST[4], o segmento de *cloud computing* divide-se basicamente em três modalidades de serviço, os quais agrupam as diferentes utilizações da infraestrutura da nuvem. São eles:

- ***Infrastructure as a Service*** **(IaaS):** Nessa modalidade, o cliente utiliza recursos como armazenamento, processamento e rede. Normalmente, nesta modalidade a

---

[4] NIST: *National Institute of Standards and Technology*.

utilização baseia-se na instalação de máquinas virtuais que operam aplicações (SANTOS REIS, 2017).

- ***Platform as a Service* (PaaS):** É um serviço de computação que consiste numa plataforma de alojamento e implementação de *software* que disponibiliza ferramentas para que os utilizadores possam desenvolver e customizar as suas aplicações de uma forma coordenada, rápida e automatizada. Dessa maneira, as aplicações de desenvolvimento, ferramentas de compilação, bibliotecas, bases de dados, servidores aplicacionais, e entre outros, não têm de ser instalados nas máquinas desses utilizadores (SANTOS REIS, 2017).
- ***Software as a Service* (SaaS):** Já esta é uma forma de distribuição de *software*, no qual o provedor é responsável por toda a estrutura necessária para a disponibilização da aplicação aos clientes. O cliente apenas é responsável por adquirir, utilizar e customizar o *software*. Os serviços são acessados por uma interface disponibilizada através da Internet, como um navegador ou uma API[5] (SANTOS REIS, 2017).

2.2 VIRTUALIZAÇÃO

O termo virtualização refere-se a um conceito já plenamente difundido nos dias de hoje. Esta tecnologia vem sido desenvolvida há muito tempo, e, portanto, já está presente em grande parte dos serviços computacionais que se utiliza hoje.

Segundo Alexandre Carissimi (2008), a virtualização é a técnica que permite com que se particione um sistema computacional individual em vários outros, que são denominados de máquinas virtuais. Cada uma dessas máquinas virtuais oferece uma estrutura completa, muito similar a uma máquina física. Assim, em cada uma delas pode-se instalar seu próprio sistema operacional, aplicativos e serviços de rede. É possível ainda interconectar (virtualmente) cada uma dessas máquinas através de interfaces de redes, *switches*, roteadores e *firewalls* virtuais, além do uso já bastante difundido de VPN [6] (CARISSIMI, 2008, p. 173).

Hoje, é muito difícil imaginar um ambiente de serviços em rede que não utilize a virtualização. Esta tecnologia permite a utilização de uma única estrutura de *hardware*

[5] API: *Application Programming Interface*. É um conjunto de rotinas e padrões de programação para acesso a um aplicativo de software ou plataforma baseado na Web.

[6] VPN: *Virtual Private Networks.* Rede de comunicações privada construída sobre uma rede de comunicações pública.

compartilhada entre diversas máquinas virtuais, ampliando significativamente as possibilidades de aproveitamento de recursos entre diversos serviços, bem como servidores *web*, DNS[7], de e-mail ou até mesmo de firewall.

Com a virtualização, abriu-se espaço para o crescimento da computação em nuvem. Empresas como a Amazon, na divisão Amazon Web Services (AWS), utilizam largamente desse conceito para disponibilizar diversos serviços, como hospedagem de serviços *web*, servidores de arquivos, entre outros.

Hoje, tornou-se comum no nosso dia-a-dia a utilização dos serviços em nuvem. Após a ascensão dos *smartphones*, grande parte dos serviços que antes dependiam do poder computacional do próprio dispositivo final estão sendo convertidos nesse tipo de aplicação virtual. Com isso, toda a responsabilidade pelo processamento, disponibilidade, escalabilidade, confiabilidade e confidencialidade passa para os *data centers*, que possuem estrutura de *hardware* e *software* preparada exclusivamente para atender a estes requisitos.

No ambiente empresarial, a virtualização pode trazer também diversos benefícios. Atualmente, as empresas têm transferido grande parte de seus serviços para *data centers*, devido a diversas questões, que vão desde a segurança até a economia em manutenção de servidores. Porém, servidores locais ainda são utilizados para alguns destes, como serviços de impressão, DHCP[8], *firewall*, autenticação, e aplicações relacionadas com o negócio da empresa (CARISSIMI, 2008, p. 173).

De fato, quando se fala em ambiente computacional, sobretudo em serviços interconectados em rede, torna-se quase que essencial mencionar a tecnologia de virtualização.

2.2.1 Hypervisors e Máquinas Virtuais

Para que a virtualização seja possível, de fato, há um sistema que faz o "meio de campo" entre o *hardware* e as máquinas virtuais, o *hypervisor*. Este sistema fornece uma camada que gerencia e fornece recursos de *hardware* às máquinas virtuais de forma organizada (ANDRADE, 2018).

---

[7] DNS: *Domain Name System.* Sistema hierárquico e distribuído de gerenciamento de nomes para dispositivos conectados na internet ou em redes privadas.

[8] DHCP: *Dynamic Host Configuration Protocol*. Protocolo para configuração dinâmica de endereçamento de terminais.

Há dois tipos básicos de *hypervisors,* que são o *bare-metal* e o *hosted hypervisor.* Este primeiro é instalado diretamente sobre o *hardware.* A partir de então as máquinas virtuais são configuradas. Já o segundo tipo é o que normalmente conhecemos em ambiente doméstico ou comercial. Nele o *hypervisor* é instalado em um sistema operacional, e a partir dele faz acesso ao *hardware* (SANTOS REIS, 2017).

De fato, o *hypervisor* do tipo *bare-metal* é o mais apropriado para a *cloud,* pois como não sofre interferência do gerenciamento de processos de um sistema operacional e faz acesso direto ao *hardware*, permite maior desempenho e eficiência (SANTOS REIS, 2017).

A virtualização com *hypervisors* e máquinas virtuais é normalmente indicada para cenários em que há a necessidade de se utilizar aplicações que executam em diferentes sistemas operacionais. Sendo assim, os *hypervisors* conseguem emular e gerenciar máquinas inteiras, criando nas aplicações hospedeiras a sensação de que estão executando em máquinas físicas isoladas (SANTOS REIS, 2017).

2.2.2 Contêineres

Com a tecnologia de virtualização com *hypervisors*, criam-se máquinas virtuais inteiras, incluindo além da aplicação que se deseja hospedar, toda estrutura do sistema operacional, *drivers*, entre outros.

Com a necessidade de se ter instâncias de sistemas operacionais tal qual a quantidade de máquinas virtuais instaladas, surgem alguns problemas. Há uma necessidade maior de tempo e custo para manter licenças, atualizações de segurança, entre outros. Além disto, parcela importante dos recursos de *hardware* que poderia ser utilizada pelas aplicações hospedadas acaba sendo reservada para estes sistemas operacionais (SOUZA, 2018).

Para sanar estes problemas, surge o conceito de contêiner, em que ao invés de virtualizar a camada de *hardware*, como fazem os *hypervisors,* agora é virtualizada a camada de aplicação, ou seja, o sistema operacional (SOUZA, 2018).

Na Figura 1, pode-se verificar a comparação entre os dois tipos de arquitetura. E notável que agora há a necessidade apenas de um sistema operacional, e sobre ele o *software* para contêineres. Isto aperfeiçoa a utilização dos recursos de *hardware*, e o *host* acaba suportando um número maior de aplicações.

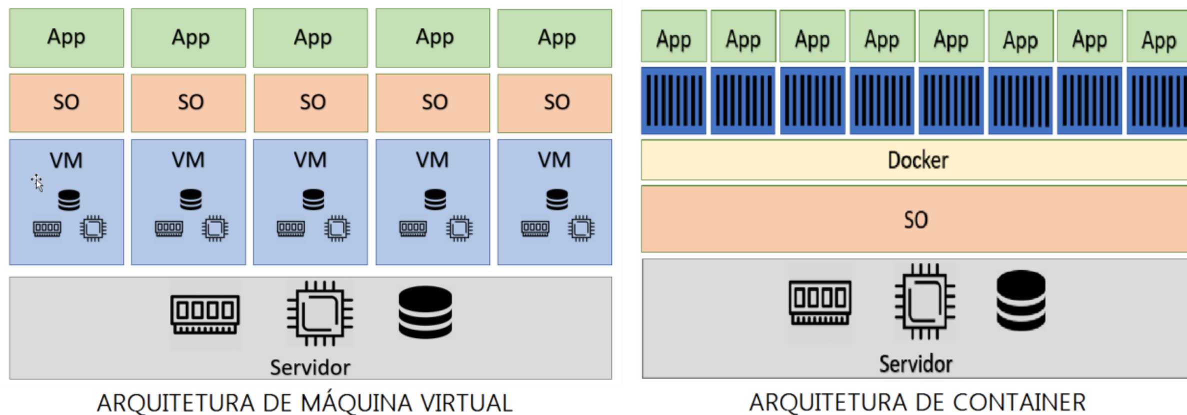

Figura 1 – Comparação entre a arquitetura de máquina virtual e contêiner.
Fonte: (SOUZA, 2018)

2.3 MICROSSERVIÇOS

Em resumo, a expressão microsserviços é uma abordagem para o desenvolvimento de aplicações. Neste conceito, o *software* deve ser construído para que seja dividido em pequenos "pedaços", que são fracamente acoplados, isolados, e comunicam-se através de algum mecanismo leve e rápido, como APIs HTTP (MENDES, 2016).

Este modelo de arquitetura vem em oposição ao que se vem praticando no mercado de *software*, que são as aplicações monolíticas. Neste modelo de aplicação, estas aplicações podem ser compostas também por vários serviços, porém a sua implantação é realizada apenas por um elemento, como, por exemplo um arquivo *war*. Esta abordagem traz dificuldades na realização de alterações, pois cada alteração na aplicação requer a criação de uma nova versão de todo o sistema, e consequentemente que toda a implantação seja feita novamente.

Aplicações monolíticas são uma maneira natural de desenvolver as aplicações, e são em suma bem-sucedidas. Porém, com a ascensão dos serviços em nuvem, começaram a surgir dificuldades. Os ciclos de alterações começaram a ficar amarrados e custosos, nos quais qualquer mínima alteração em apenas uma parte do *software* faz com que toda a aplicação monolítica necessite ser republicada. Com o tempo fica cada vez mais difícil manter esta estrutura modular, sendo difícil separar as mudanças que deveriam afetar somente um módulo.

Outra dificuldade encontrada na utilização de arquiteturas monolíticas é relativa a escalabilidade. Sempre que é necessário realizar ações de escalabilidade em algum

componente, precisa-se escalar a aplicação por completo, levando a um desperdício de recursos. Estas aplicações normalmente são compostas por uma grande quantidade de código com um grau de complexidade alto, elevando a dificuldade para que a equipe de desenvolvimento possa gerenciar e realizar as implementações durante o ciclo de vida da aplicação.

Na Figura 2 é possível visualizar uma breve comparação entre a presente arquitetura monolítica e a arquitetura de microsserviços, sobretudo na visão da escalabilidade.

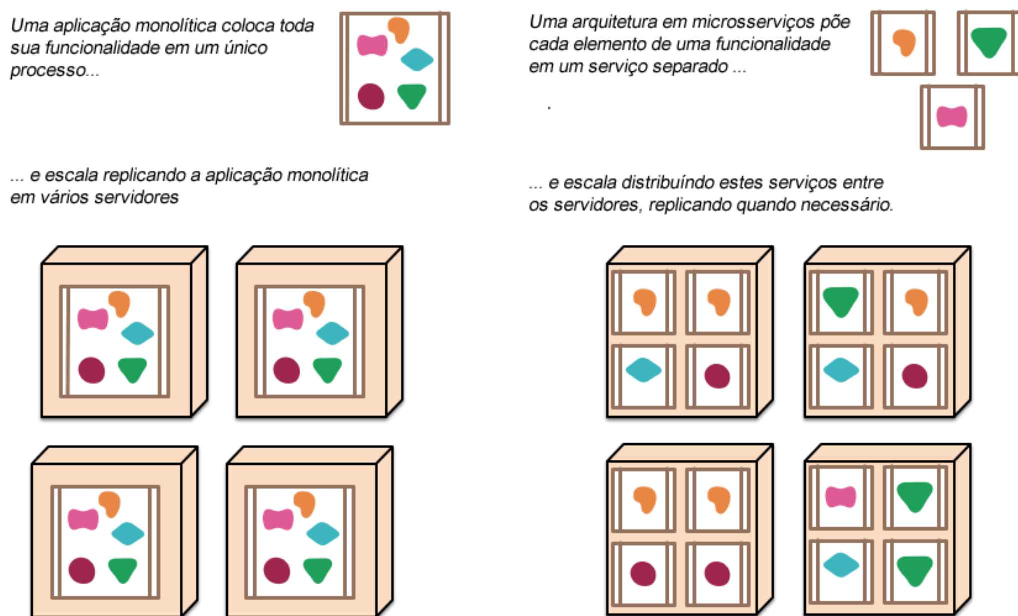

Figura 2 – Comparação entre a arquitetura de microsserviços e a arquitetura monolítica.
Fonte: (MENDES, 2016)

2.4 ELASTICIDADE NA COMPUTAÇÃO EM NÚVEM

Como a *cloud* é uma estrutura capaz de hospedar diferentes serviços a baixos custos, muitas empresas estão sendo seduzidas por estas vantagens e disponibilizando suas aplicações neste ambiente. Dessa maneira, as empresas fornecedoras deste serviço devem implementar mecanismos que permitam uma maior otimização dos recursos computacionais, aumentando e diminuindo recursos alocados para cada cliente conforme a demanda.

Com esta necessidade tem-se o conceito de elasticidade, que é a capacidade de uma estrutura de hospedagem de serviço conseguir aumentar ou diminuir a alocação de recursos de *hardware* dinamicamente conforme a necessidade da aplicação, melhorando a estabilidade, desempenho e disponibilidade dos serviços.

Segundo Santos Reis (2017), elasticidade é o ponto em que um sistema é capaz de se adaptar às mudanças da carga de trabalho por prover ou desprover recursos automaticamente, de tal forma, que em cada momento os recursos disponibilizados correspondem à demanda atual o mais próximo possível das necessidades.

2.4.1 Classificação

Para que se alcance a elasticidade, diferentes soluções podem ser tomadas, e estas podem ser classificadas em classes, no que diz respeito ao local, política, propósito e método. Suas características serão descritas a seguir: (SANTOS REIS, 2017)

A. **Local:** Nesta classe se define o local no qual as práticas de elasticidade serão tomadas. Podem-se realizar procedimentos em todas as camadas da *cloud*, porém o que mais é praticado são implementações em nível de IaaS[9], no qual o controlador da cloud fornece os recursos necessários às máquinas virtuais de acordo com suas métricas.

B. **Política:** Há duas opções, que são as políticas manuais e automáticas. Nas políticas manuais, o utilizador é responsável por monitorar e tomar ações quanto a liberação de recursos. Já nas políticas automáticas, o próprio sistema que toma as decisões de elasticidade de acordo com algumas regras definidas pelo utilizador, que normalmente constam no *Service Level Agrement* (SLA). Para tanto, informações e dados sobre a utilização dos recursos computacionais devem estar disponíveis em tempo real. Também existe a possibilidade de se utilizar heurísticas, técnicas matemáticas e analíticas que antecipam o comportamento da carga do sistema para que as ações sejam desencadeadas de forma preditiva, baseando-se em histórico ou padrões de comportamento. Além destas, outra forma de desencadear as ações de elasticidade é chamada de **reativa**. Estas soluções são baseadas em mecanismos de Regra-Condição-Ação, em que uma regra é composta por um conjunto de condições que quando satisfeitas desencadeiam um conjunto de ações que fornecem a elasticidade.

C. **Propósito:** Sendo a elasticidade uma das características chaves dos sistemas da *cloud*, esta pode possuir diversos propósitos. Ela fornece eficiência, pois

[9] IaaS: *Infraestructure as a Service.*

apenas se utiliza a quantidade necessária de recursos para o funcionamento do sistema. Outro propósito ao nível do utilizador é a estabilidade no desempenho das aplicações hospedadas na *cloud*, pois os recursos serão aumentados de forma automática de maneira a lidar com possíveis aumentos de tráfego. Além destes, pode-se citar a redução no uso de energia, devido a melhor gerência de recursos.

D. **Método:** Existem três maneiras fundamentais de fornecer e programar elasticidade nas aplicações que executam em grupos de instâncias. Esta pode ser implementada por escalabilidade horizontal, escalabilidade vertical e escalabilidade por migração. A primeira foi a aplicada na realização deste trabalho acadêmico, e consiste em aumentar ou diminuir o número de instâncias virtuais alocadas a um serviço conforme a demanda. Estas podem ser máquinas virtuais, contêineres ou módulos. Neste tipo de escalabilidade por replicação é necessário um componente extra capaz de rotear e dividir os pedidos pelas várias réplicas das instâncias, habitualmente utiliza-se um balanceador de carga, como o Kubernetes, um dos objetos de estudo deste trabalho. Uma vantagem de utilizar este tipo de escalabilidade é que se torna a aplicação mais robusta a falhas, porque introduz redundância no sistema e caso uma das instâncias falhe existem outras executando o mesmo serviço. Na escalabilidade vertical normalmente apenas existe uma única máquina virtual e a elasticidade é conseguida ao redimensionar a cota de capacidades dessa única máquina, tais como CPU, memória ou rede. A principal vantagem deste método, é que se consegue adaptar o sistema mais facilmente e rapidamente à demanda de pedidos, pois ao contrário da escalabilidade horizontal não é necessário criar uma máquina com todos os componentes instalados para correr o serviço, basta aumentar ou diminuir a cota de recursos. Esta forma de elasticidade apenas é exequível se for possível interagir com o *hypervisor* da *cloud* com bastante autoridade, o que faz com que se torne uma opção mais complexa e insegura. Por fim, é introduzido o último método de elasticidade por migração, ao qual este método tem implicações ao nível físico. Neste método a elasticidade é implementada com a migração da máquina virtual para outra máquina física que melhor enquadra a carga a que a máquina virtual está sujeita no momento.

3 MATERIAIS E METODOLOGIA

A ênfase deste capítulo está em reportar o que será utilizado, as configurações e as tecnologias que são necessárias para implementar o aplicativo e/ou executá-lo.

3.1 MATERIAIS

Nesta seção serão apresentadas todas as ferramentas utilizadas para a realização deste trabalho acadêmico. Estas ferramentas são: Java EE, Rest, Docker e Kubernetes.

3.1.1 Java EE

No mundo das aplicações *web*, a linguagem Java é amplamente utilizada para desenvolvimento de *softwares*, sobretudo em aplicações *back-end*. Estas aplicações possuem normalmente muitas regras de negócio em seu código, sendo sua manutenção de grande complexidade para as equipes de desenvolvimento.

Para auxiliar no desenvolvimento dessas aplicações, criou-se uma série de especificações, que quando implementadas podem auxiliar na implementação dos requisitos não-funcionais, que são geralmente a persistência em banco de dados, controle de transação, acesso remoto, *web services*, gerenciamento de *threads*, gerenciamento de conexões HTTP, cache de objetos, gerenciamento da sessão *web*, balanceamento de carga, entre outros.

Estas especificações receberam a denominação de Java EE (Java Enterprise Edition), em que consiste em uma série de especificações bem detalhadas, dando uma receita de como deve ser implementado um *software* que faz cada um desses serviços de infraestrutura (CAELUM, 2018).

Neste trabalho acadêmico, será utilizada sobretudo o conceito de *servlet*, na implementação de um pequeno “servidor”, que receberá requisições HTTP e retornará uma resposta ao cliente.

3.1.2 Rest

Nas aplicações *web* atuais, é comum a utilização de APIs, que são um conjunto de rotinas e padrões estabelecidos e documentados por uma aplicação “A”, para que outras aplicações consigam comunicar-se com esta de forma a trocar informações e dados entre si.

Para que esta comunicação entre as aplicações seja possível, é necessário que haja um protocolo para troca de dados. O principal protocolo utilizado nestes sistemas *web* é o protocolo HTTP, que existe a mais de 20 anos. Há diversas formas de utilizar este protocolo para trocar informações entre sistemas. Dentre as mais utilizadas, está a arquitetura REST.

REST significa *Representational State Transfer*. Em português, Transferência de Estado Representacional. Trata-se de uma abstração da arquitetura da *web*. Resumidamente, o REST trata de um conjunto de princípios/regras/*constraints* que definem como *Web Standards* como HTTP e URIs[10] devem ser usados (TILKOV, 2008).

Na *web*, cada página, vídeo ou documento acessível é denominado de recurso. Dessa maneira, pode-se entender os recursos como a fundação da arquitetura de sistemas baseados na *web*. Cada um desses recursos precisa de um mecanismo de identificação, para que possa ser encontrado e manipulado. Nesse sentido a *web* provê o padrão URI, por meio do qual identifica-se unicamente um recurso, ao mesmo tempo em que o torna endereçável e manipulável em protocolos de aplicação, como o HTTP.

Manipulam-se recursos a partir de suas representações. Uma representação é uma visão de um estado do recurso em determinado instante do tempo; esta visão é codificada em formatos padronizados e passíveis de transferência via *web*, como, XHTML, JSON, XML, TEXT PLAIN, CSV, etc..

Toda interação com um recurso é, portanto, mediada por uma representação do mesmo, em que tanto o cliente quanto o servidor, convergem no entendimento da codificação daquela representação. Esse “acordo” entre cliente e servidor é obtido em tempo de execução, por meio de um mecanismo chamado *Content Negotiation* (LUSA, 2013).

[10] URI: *Uniform Resource Identifier*. É uma cadeia de caracteres compacta usada para identificar ou denominar um recurso na Internet.

3.1.3 Docker

Docker é uma plataforma que possibilita o empacotamento de uma aplicação ou ambiente completamente dentro de um contêiner, e, a partir disto, este se torna portável para qualquer outro *Host* que contenha o Docker instalado.

O Docker foi escrito na linguagem Go, uma linguagem de alto desempenho desenvolvida dentro do Google, que facilita a criação e manutenção deste tipo de ambiente.

Esta forma de funcionamento reduz drasticamente o tempo de implantação de alguma infraestrutura ou até mesmo aplicação, pois não há necessidade de ajustes de ambiente para o correto funcionamento do serviço, o ambiente é sempre o mesmo, que é configurado apenas uma vez, podendo ser replicado *n* vezes, conforme a necessidade (DIEDRICH, 2015).

A tecnologia Docker utiliza o *kernel* do Linux e alguns de seus recursos, como *Cgroups* e *namespaces*, os quais permitem segregar processos de modo que eles possam ser executados independentes. Este é o objetivo dos contêineres: criar a habilidade de executar processos e aplicativos separadamente, utilizando melhor a infraestrutura e mantendo a segurança de se manter sistemas separados (RED HAT, 2018).

Uma das características marcantes do Docker, que se torna um ponto positivo, é a possibilidade de criar suas imagens (contêiners prontos para serem implantados) a partir de arquivos de definição chamados *Dockerfiles*. Dessa forma agiliza-se o processo de desenvolvimento e implantação das aplicações (DIEDRICH, 2015).

A arquitetura do Docker inclui basicamente três componentes principais:

A. **Cliente e Servidor:** Também chamado de *daemon* do Docker, o servidor atende à solicitação do cliente por meio da API ou de uma interface de linha de comando. Tanto o cliente e servidor podem ser executados na mesma máquina quanto o cliente pode ser remoto. O servidor é responsável por enviar, obter e gerenciar recursos para o cliente.

B. **Registro:** O *Docker Trusted Registry* (DTR) é um produto comercial que permite o fluxo de trabalho completo de gerenciamento de imagem, com integração LDAP[11], assinatura de imagem, verificação de segurança e integração com o *Universal Control Plane*. O DTR é oferecido como um complemento das assinaturas do

[11] LDAP: *Lightweight Directory Access Protocol.* Protocolo de aplicação aberto, livre de fornecedor e padrão de indústria para acessar e manter serviços de informação de diretório distribuído sobre uma rede de Protocolo da Internet.

Docker Enterprise padrão ou superior. Basicamente, é um ambiente no qual é possível armazenar e distribuir imagens do Docker (DOCKER, 2018)

C. ***Host*****:** É basicamente a imagem executada no Docker. Esta é encapsulada com um contêiner que contém o todo o aparato necessário para executar o aplicativo em um ambiente.

3.1.4 Kubernetes

O Kubernetes é uma ferramenta de código aberto utilizada para orquestrar e gerenciar *clusters* de contêineres. Neste projeto acadêmico o *cluster* está representado por um conjunto de máquinas virtuais executando uma *engine* de contêiner, neste caso o Docker, sendo a principal função do Kubernetes gerenciar, controlar e monitorar o estado desses contêineres ao longo do *cluster* (TRUCCO, 2018).

3.1.4.1 Definições

No Kubernetes todos os elementos são tratados como objetos. Estes serão descritos a seguir (SANTOS REIS, 2017):

1. ***Pod*****:** É a abstração de um conjunto de contêineres executando em conjunto. Neste caso, é necessário que exista um contêiner principal no *Pod*, com apenas um processo, e os demais auxiliando-o em sua execução. Em seu ciclo de vida, eles podem passar pelos estados pendente, em execução, sucedido, falha e desconhecido. Quando um nó é lançado em um *cluster*, este permanece ativo até ser destruído. Se este entrar em estado de falha, outro *Pod* é iniciado em seu lugar, no mesmo *cluster* ou não (KUBERNETES, 2018).
2. ***Volume*****:** No Kubernetes um *Volume* é um diretório do *host* que é montado no *Pod*. O ciclo de vida deste é idêntico ao do *Pod*. Isto faz com que quando um contêiner é reiniciado após uma falha os dados sejam preservados, mas caso um *Pod* seja eliminado o *Volume* também seja eliminado. Além disso o *Volume* permite o compartilhamento de arquivos entre contêineres que pertencem ao mesmo *Pod* (KUBERNETES, 2018).
3. ***Service*****:** *Services* é a denominação de outra camada de abstração que define um conjunto de *Pods* semelhantes que fornecem um serviço. Como os *Pods* são

unidades temporárias, que podem ter um ciclo de vida curto, os pedidos de comunicação não podem ser enviados diretamente do remetente para o *Pod*. Para isso é utilizado o conceito de *Services* como um competente consistente que lida com a distribuição de pedidos pelos vários *Pods*. Então através dos *Services* é possível expor os *Pods* ao tráfego interno e externo ao *cluster* com balanceamento de carga pelas várias réplicas. Para cada *Service* são designados um endereço IP e um nome, permitindo que os outros *Pods* possam descobrir o serviço dentro do *cluster* Kubernetes (KUBERNETES, 2018).

4. ***Label:*** É um mecanismo de seleção e agrupamento constituído de pares chave/valor anexados de objetos que servem para identificar e organizar objetos em grupos relacionados de forma significativa para o utilizador. A partir destas *labels* são selecionadas as réplicas de *Pods* que poderão fazer parte de um *service* (KUBERNETES, 2018).
5. ***ReplicaSet:*** É uma ferramenta que evoluiu do *Repplication Controller,* a qual é responsável por controlar o número de réplicas de um *Pod* a ser executado paralelamente em um *cluster*, bem como manter este número eliminando Pods terminados e repondo *Pods* que entraram em estado de falha (KUBERNETES, 2018).
6. ***Deployment:*** O controlador *deployment* provê maneiras de se programar alterações de estado para os *Pods*, em uma taxa de frequência controlada (KUBERNETES, 2018).
7. ***StatefulSet*:** StatefulSet é um objeto que permite controlar a ordem de *deployment* de um grupo de Pods e atribui identificadores estáveis a esses Pods. Neste objeto o *deployment* dos *Pods* é controlado, ou seja, só passa para o *deployment* do *Pod* seguinte quando o *deployment* do *Pod* anterior tiver sido realizado com sucesso. Com a atribuição de identificadores estáveis permite que quando um *Pod* é relançado noutro nó receba invariavelmente o mesmo identificador. Esta funcionalidade tem vantagens para o *deployment* de aplicações com estado criadas em *cluster* em que cada um dos elementos do *cluster* tem de conhecer os seus pares de forma a interagir diretamente com estes (KUBERNETES, 2018).

Para entender o funcionamento do Kubernetes, se faz necessário compreender a estrutura do *cluster*. Nela são adotados alguns termos, como Master e nós *Workers/Minions*.

O nó Master é a máquina centralizadora, que controla todos os nós do Kubernetes, bem como as atribuições de tarefas do *cluster*. Nesta máquina estão instalados e executando todos os componentes, que são quatro: (TRUCCO, 2018)

1. **ETCD:** É uma base de dados de chave valor. Ele armazena os dados de configuração e estado do *cluster*;
2. **API Server:** Fornece a API através de Json. Os estados de objetos da API são armazenados no ETCD, e o *kubectl* usa o API Serve para se comunicar com o *cluster*;
3. **Controller Manager:** Monitora os controladores de replicação e cria os *Pods* para manter o estado desejado;
4. **Scheduler:** é responsável por executar as tarefas de agendamento, como execução de contêineres nos nós *workers/minions* com base na disponibilidade de recursos. Já nos nós Minion são executados basicamente outros três componentes:
   1. **Kubelet:** Agente que é executado em cada nó worker, se conecta ao Docker e cuida da criação, execução e exclusão de contêineres;
   2. **Docker:** É responsável por executar e executar os contêineres, seguindo as solicitações do Kubernetes;
   3. **Kube-Proxy:** Encaminha o tráfego para os contêineres apropriados com base no endereço IP e no número da porta da solicitação recebida.

3.1.4.2 Rede

O Kubernetes deve garantir que *Pods* possam comunicar-se independentemente do nó do *cluster* em que se encontrem. Para que isto aconteça é fornecido a cada *Pod* um endereço IP interno ao *cluster* que garante comunicação entre os *Pods* sem necessidade de criar *links* explícitos entre estes ou mapeamento entre as portas dos contêineres e as portas do *host*. Do ponto de vista da alocação de portas, dos nomes, da descoberta de serviços, do balanceamento de carga, da configuração de aplicações e migração os *Pods* podem ser tratados como máquinas virtuais ou *hosts* físicos. Este modelo de rede satisfaz os seguintes requisitos de rede:

- Todos os contêineres podem comunicar-se entre si;
- Todos os nós do *cluster* podem comunicar com todos os contentores (e vice-versa);

- O endereço IP que um contêiner vê (por exemplo pelo comando ifconfig) é o mesmo endereço IP que os outros contentores veem (SANTOS REIS, 2017);

Para implementar o modelo de rede que satisfaça os requisitos de comunicação do Kubernetes pode ser utilizado o Flannel. Ele é um sistema simples capaz de implementar uma rede de *overlay* na camada de rede que aloca uma subrede de um espaço de endereçamento pré configurado a cada nó do *cluster* transportando o tráfego entre os nós (SANTOS REIS, 2017).

3.2 METODOLOGIA

Nesta sessão são explanadas as estratégias para lidar com aplicações conteinerizadas desde o desenvolvimento até a fase de produção. Será traçado um caminho que mostra o processo de desenvolvimento de aplicações conteinerizadas com os seus vários componentes que são capazes de lidar com a aplicação nas várias fases do seu ciclo de vida. Posteriormente serão levantados os fatores chaves para que possam ser abordados e analisados com mais detalhe nas secções subsequentes. Esses fatores chaves focam-se essencialmente no desenvolvimento, validação e distribuição de imagens, no *deployment* de aplicações conteinerizadas sobre um *cluster* Kubernetes.

3.2.1 Etapas do desenvolvimento de aplicações conteinerizadas

Para que uma aplicação conteinerizada vá do desenvolvimento ao ambiente de produção, uma série de processos devem ser realizados e vários fatores devem ser levados em consideração. Para facilitar o entendimento, pode ser analisado o fluxo apresentado na Figura 3.

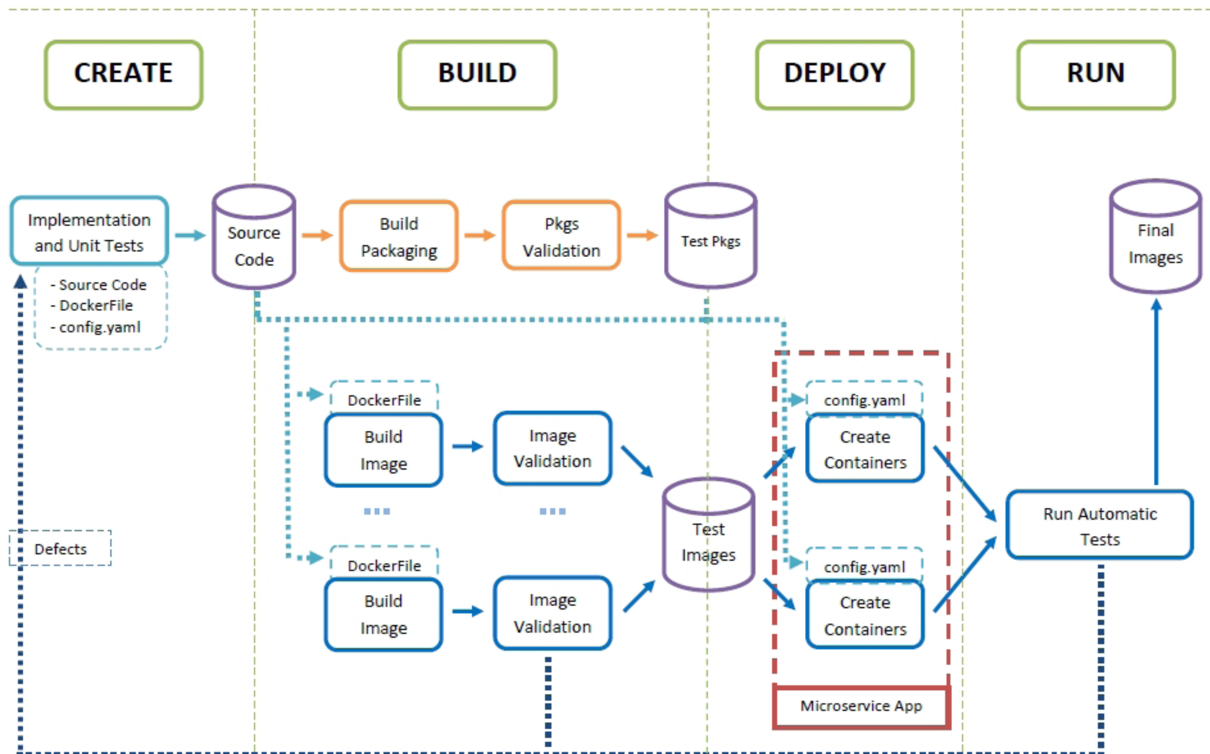

Figura 3 - Fluxo de desenvolvimento de aplicações personalizadas.
Fonte: (SANTOS REIS, 2017)

Como pode ser verificado, o processo se distribui em basicamente quatro etapas. São elas: *create, build, deploy* e *run*. A seguir consta uma breve descrição de cada uma das etapas (SANTOS REIS, 2017):

3.2.1.1 Fase Create

Nesta fase está, além da análise da aplicação, a criação dos *Dockerfiles,* que servirão para automatizar o processo de construção da imagem. Para que estas imagens possam ser instanciadas em contêineres de forma a executar os serviços da aplicação deve ser criado um conjunto de arquivos de configuração YAML, que vão permitir a realização do *deployment* da aplicação no *cluster*.

3.2.1.2 Fase Build

Nesta fase concentra-se a criação da imagem propriamente dita, bem como o empacotamento desta em pacotes denominados RPMs. Aqui são validadas as regras para que esta imagem possa efetivamente ser realizado o *deployment* no *cluster.*

3.2.1.3 Fase Deploy

Nesta fase as imagens serão instanciadas em contêineres e implantadas em um *cluster* semelhante ao ambiente de produção, para que na próxima fase se possa proceder à realização dos testes.

3.2.1.4 Fase Run

Na fase *Run* que vai ser o ambiente em que a aplicação está em execução e totalmente funcional sobre o *cluster* de testes para que possam ser iniciados todos os testes da aplicação.

4 RESULTADOS

4.1 DESENVOLVIMENTO DA APLICAÇÃO

Em um ambiente empresarial, há uma grande utilização de protocolos de controle de arquivos via rede como FTP[12] e SFTP[13]. No mercado existem diversos clientes que permitem a utilização deste protocolo, como Filezilla, por exemplo.

Para este estudo de caso, foi escolhido o protocolo HTTP, através de métodos GET e POST, para que se possa realizar a listagem, *upload*, e *download* de arquivos pela rede.

Através disso, é possível utilizar desta API em diversas ferramentas, como SoapUI, Postman, aplicações *front end* em navegador ou até mesmo em aplicativos móveis.

Atualmente, a maior dificuldade enfrentada no cenário deste trabalho é o desempenho da aplicação HTTP no servidor de aplicação Apache Tomcat, em que o método que disponibiliza *download* de arquivos consome recursos de rede e de processamento que acabam por travar a aplicação como um todo.

Neste cenário, faz-se necessária uma aplicação isolada em contêineres, com possibilidade de elasticidade, que suporte uma maior carga de *downloads* simultâneos mantendo a estabilidade.

Para efeitos de comparação, neste trabalho acadêmico, foi desenvolvido um projeto na linguagem Java, preparado para executar no servidor de aplicação Apache Tomcat. Este foi disponibilizado em dois ambientes: de forma monolítica no Apache Tomcat, e também no mesmo servidor de aplicação alocado em contêineres no Docker, orquestrados pelo Kubernetes.

O projeto possui funcionalidades simples, pois o foco do estudo é na avaliação de performance das duas formas de disponibilizar a aplicação: Modelo monolítico e modelo de microsserviços segregados em contêineres. Os métodos de *download/upload* são estruturados da forma exposta na Listagem de Código 1:

```
@GET
@Path("download")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response download(@QueryParam("nomearquivo") String nomearquivo) throws Exception {
   File file;
   try {
```

[12] FTP: *File Transport Protocol.* É um protocolo de transferência de arquivos em rede.

[13] SFTP: *Secure File Transport Protocol.* É um protocolo de transferência de arquivos e de manipulação funcional. É tipicamente utilizado com o protocolo de segurança SSH-2.

```
        file = new File(this.getDiretorio() + nomearquivo);
    } catch (Exception e) {
        throw new Exception("Falha ao buscar diretório!");
    }
    if (file.exists()) {
        return Response.ok(file, MediaType.APPLICATION_OCTET_STREAM)
                .header("content-disposition","attachment; filename = " + nomearquivo)
                .build();
    } else {
        return Response.noContent().build();
    }
}

@POST
@Path("upload")
@Consumes(MediaType.MULTIPART_FORM_DATA)
public Response uploadFile(
        @FormDataParam("file") InputStream uploadedInputStream,
        @FormDataParam("file") FormDataContentDisposition fileDetail) {

    String uploadedFileLocation = this.getDiretorio() + fileDetail.getFileName();

    try {
        OutputStream out = new FileOutputStream(new File(
                uploadedFileLocation));
        int read = 0;
        byte[] bytes = new byte[1024];

        out = new FileOutputStream(new File(uploadedFileLocation));
        while ((read = uploadedInputStream.read(bytes)) != -1) {
            out.write(bytes, 0, read);
        }
        out.flush();
        out.close();
    } catch (IOException e) {

        e.printStackTrace();
    }

    return Response.status(200).entity("Arquivo salvo com sucesso!").build();

}
```

Listagem de Código 1 - Código fonte dos métodos de download e upload

Para o projeto que foi disponibilizado em contêineres, um arquivo adicional foi necessário, o *Dockerfile*. Neste arquivo constam as configurações necessárias para que seja possível gerar uma imagem do projeto que possa ser disponibilizada em contêiner. O arquivo foi gerado na forma exposta na Listagem de Código 2:

```
FROM tomcat:8.5-jre8
ADD filemgr-dk.war /usr/local/tomcat/webapps/
CMD ["catalina.sh", "run"]
```

Listagem de Código 2 - Dockerfile

Com os comandos descritos no *Dockerfile*, através da cláusula FROM, o Docker detecta que a imagem a ser gerada utiliza como base a imagem do Apache Tomcat. O comando ADD descreve que será copiado o arquivo de extensão .war, que é o projeto compilado, pronto para uso, para dentro do servidor de aplicação, na pasta *webapps*, a fim de que este seja identificado como aplicação. Por fim, é definido o comando para iniciar o *software*.

Para que a imagem seja efetivamente gerada, é necessário que na máquina o Docker já esteja instalado e devidamente configurado. A partir de então, com o comando *docker build*, a imagem foi gerada e vinculada ao repositório local do docker.

Um passo adicional executado neste projeto, foi o envio ao Docker Hub, que é o repositório de imagens em *cloud* do Docker. Ele funciona de forma similar ao Git, que é um sistema de controle de versão de arquivos distribuído, utilizado principalmente no meio de desenvolvimento de *software*. No Docker, é necessário inicialmente criar um repositório. Então com alguns comandos como *docker login*, *docker commit* e *docker push* a imagem é enviada ao repositório, e pode ser baixada e utilizada em outras máquinas que possuam o Docker, através do comando *docker pull*.

4.2 UTILIZAÇÃO DA APLICAÇÃO NO KUBERNETES

Para que todos os conceitos tratados neste trabalho sejam plenamente satisfeitos, foi implementada a gerência do contêiner em um *cluster* no Kubernetes. Isto se tornou possível após uma série de configurações, que serão descritas a seguir.

Este *cluster* foi montado com duas máquinas executando o sistema operacional Linux Ubuntu 18.04, aqui denominadas master-vm e node-vm. Estas máquinas são os nós do *cluster*, que podem ser visualizados na Figura 4:

```
root@master-vm:/home/master# kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
filemgr-dk-54c64f68d4-h45w9  1/1     Running   1          97m
root@master-vm:/home/master# kubectl get services
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
filemgr-http   ClusterIP   10.101.149.47   <none>        8080/TCP   96m
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP    109m
root@master-vm:/home/master# kubectl get deployments
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
filemgr-dk   1         1         1            1           97m
root@master-vm:/home/master# kubectl get nodes
NAME        STATUS   ROLES    AGE    VERSION
master-vm   Ready    master   109m   v1.12.2
node-vm     Ready    <none>   104m   v1.12.2
root@master-vm:/home/master#
```

Figura 4 - Configuração dos nós do cluster Kubernetes

Para que a imagem da aplicação esteja disponível a ser acessada, mais três configurações foram necessárias, que são a criação de um *Pod*, um *deployment* e um *service*. Com o *Pod*, a imagem é baixada do Docker HUB, e disponibilizada no Kubernetes. No *deployment*, é configurada a elasticidade, para que um novo contêiner seja disponibilizado quando a carga de CPU atingir 50%. Já no *service*, é definida a porta 8080 para que o serviço esteja acessível.

Na Figura 5 abaixo, pode ser visualizada a configuração dos *services*:

```
root@master-vm:/home/master# kubectl describe services
Name:              filemgr-http
Namespace:         default
Labels:            run=filemgr-dk
Annotations:       <none>
Selector:          run=filemgr-dk
Type:              ClusterIP
IP:                10.101.149.47
Port:              <unset>  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.40.0.5:8080
Session Affinity:  None
Events:            <none>


Name:              kubernetes
Namespace:         default
Labels:            component=apiserver
                   provider=kubernetes
Annotations:       <none>
Selector:          <none>
Type:              ClusterIP
IP:                10.96.0.1
Port:              https  443/TCP
TargetPort:        6443/TCP
Endpoints:         192.168.0.120:6443
Session Affinity:  None
Events:            <none>
```

Figura 5 – Services

Na Figura 6 abaixo, pode ser visualizada a configuração dos *Pods*:

```
root@master-vm:/home/master# kubectl describe pods
Name:               filemgr-dk-54c64f68d4-h45w9
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node-vm/192.168.0.121
Start Time:         Sat, 17 Nov 2018 17:19:19 -0200
Labels:             pod-template-hash=54c64f68d4
                    run=filemgr-dk
Annotations:        <none>
Status:             Running
IP:                 10.40.0.5
Controlled By:      ReplicaSet/filemgr-dk-54c64f68d4
Containers:
  filemgr-dk:
    Container ID:   docker://fabc706eac6f79a82749831a7ca26fbc4e66318529364365a2476ded91e50717
    Image:          evertoncunico/filemgr-dk
    Image ID:       docker-pullable://evertoncunico/filemgr-dk@sha256:258b4cde425515e6f45d17a2b203d724ef4be2283b521fb6478276875888da5e
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Sat, 17 Nov 2018 18:22:21 -0200
    Last State:     Terminated
      Reason:       Error
      Exit Code:    255
      Started:      Sat, 17 Nov 2018 17:20:28 -0200
      Finished:     Sat, 17 Nov 2018 18:12:21 -0200
    Ready:          True
    Restart Count:  1
    Environment:
      DOMAIN:  cluster
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-zdc95 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-zdc95:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-zdc95
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
```

Figura 6 – Pods

Na Figura 7 abaixo, pode ser visualizada a configuração dos *deployments*:

```
root@master-vm:/home/master# kubectl describe deploy
Name:                   filemgr-dk
Namespace:              default
CreationTimestamp:      Sat, 17 Nov 2018 17:19:16 -0200
Labels:                 run=filemgr-dk
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               run=filemgr-dk
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=filemgr-dk
  Containers:
   filemgr-dk:
    Image:      evertoncunico/filemgr-dk
    Port:       8080/TCP
    Host Port:  0/TCP
    Environment:
      DOMAIN:  cluster
    Mounts:    <none>
  Volumes:     <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   filemgr-dk-54c64f68d4 (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  105m  deployment-controller  Scaled up replica set filemgr-dk-54c64f68d4 to 1
root@master-vm:/home/master#
```

Figura 7 – Deployments

Com isto, a configuração do Kubernetes está completa para a necessidade do projeto. O sistema está acessível através de um IP interno, na porta 8080. A partir disto foi executado o teste de desempenho.

A maior dificuldade na transição do modelo monolítico está presente na maior "burocratização" do processo, pois é necessário compilar imagens do projeto, disponibilizá-las em um servidor *cloud*, para então utilizar no Kubernetes.

Ainda há pouca informação sobre configuração desta ferramenta, então a adoção deste modelo torna-se complexa para profissionais com pouco conhecimento sobre este novo paradigma.

4.3 TESTES DE DESEMPENHO

No entanto, após compreender os princípios básicos como Pod, *service* e *deployment*, a utilização do Kubernetes torna-se relativamente simples. A possibilidade de utilizar arquivos de autoconfiguração (arquivos de extensão yaml), possibilita automatizar o processo de controle do cluster, bem como desfrutar de suas diversas funcionalidades.

Os testes de desempenho foram realizados para a aplicação monolítica, no servidor e aplicação Apache Tomcat, e também na aplicação hospedada no Kubernetes. Ambos os testes foram executados na mesma máquina virtual, executando o sistema operacional Linux Ubuntu 18.04.

Para realização dos testes foi utilizada a ferramenta JMeter, que realiza *n* requisições HTTP simultâneas para a API REST, de acordo com a configuração realizada, simulando um ambiente de produção.

No escopo deste projeto, o teste foi realizado simulando a execução de requisições de 900 usuários simultâneos, efetivando uma chamada cada para a API. O resultado das requisições foi obtido em uma grade de registros.

Um exemplo de teste realizado, para o método de *download*, foi a execução de 900 chamadas simultâneas, baixando um arquivo de imagem de aproximadamente 750 KB. Na Figura 8, é possível visualizar o resultado do teste realizado na aplicação de modelo monolítico:

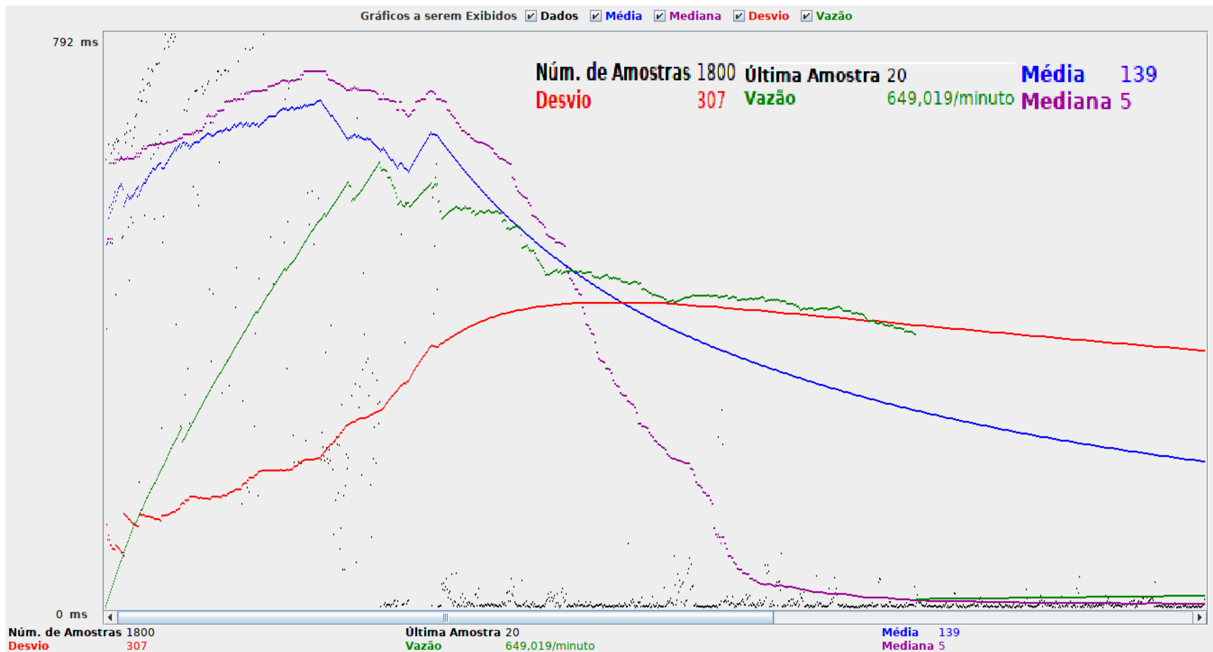

Figura 8 - Teste na aplicação monolítica

No Quadro 1, é possível visualizar um resumo da amostragem de requisições na aplicação executada no Apache Tomcat sem Docker, com os dados mais relevantes.

Quadro 1 – Testes na aplicação monolítica

Número de Amostras	Tempo de resposta médio	Vazão de requisições	Dados enviados
1800	139ms	10,81/segundo	10,84KB/s

Na Figura 9, está o resultado do teste na nova aplicação, executando no Kubernetes:

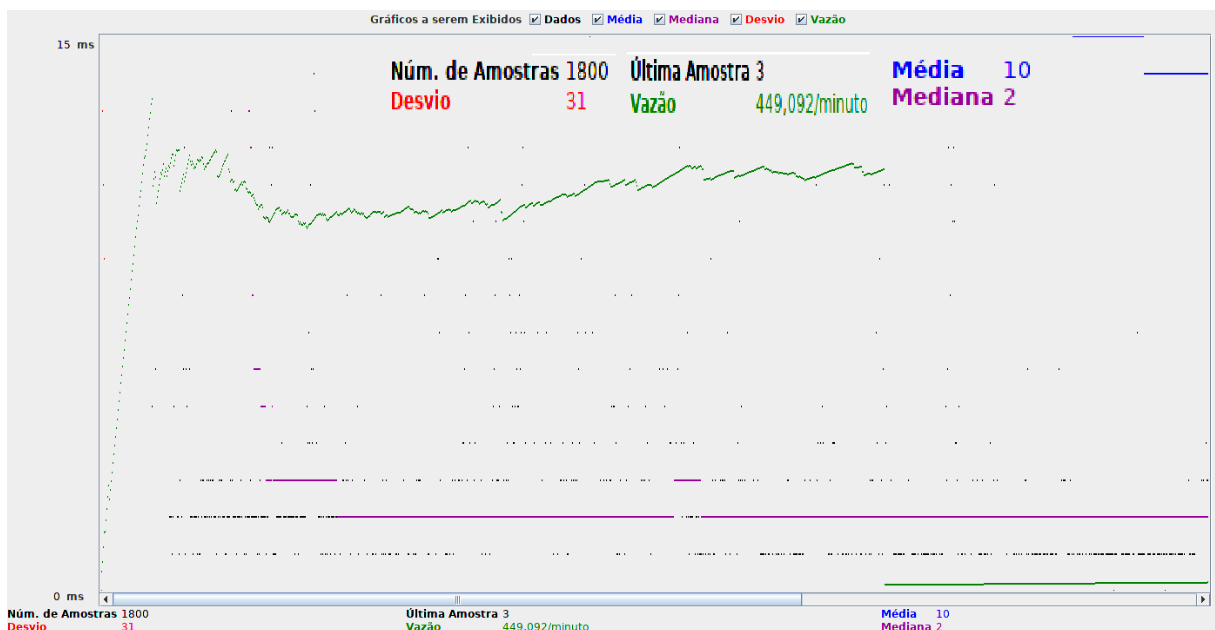

Figura 9 - Teste na aplicação em contêiner

No Quadro 2, é possível visualizar um resumo da amostragem de requisições com a aplicação conteinerizada utilizando Kubernetes, com os dados mais relevantes:

Quadro 2 – Testes na aplicação conteinerizada

Número de Amostras	Tempo de resposta médio	Vazão de requisições	Dados enviados
1800	10ms	84,8/segundo	12,43KB/s

Na execução dos testes, foram realizadas 1800 requisições para o método de *download.* Ao final dos testes em ambos os ambientes, notou-se que todas as requisições possuíram êxito no retorno.

Porém, pode-se verificar maior estabilidade do modelo de infraestrutura com o *cluster* Kubernetes. No modelo monolítico, obteve-se uma média de 143ms (vide Figura 8) para o retorno do arquivo, contra 10ms do *cluster.*

No entanto, a maior vantagem do modelo com Kubernetes é a estabilidade. Como é visível nos gráficos, o desvio padrão do modelo monolítico ficou em 252ms (vide Figura 9), contra apenas 2,28ms no *cluster*.

Por fim, mais um dado em que o novo modelo se mostrou mais eficiente foi na vazão de download, de 12,43KB/s (vide Figura 9), contra 9,68KB/s (vide Figura 8) do modelo antigo.

5 CONCLUSÕES

Ao iniciar as pesquisas para realização deste trabalho, ficou evidente a grande gama de ferramentas e metodologias que estão sendo empregadas para melhorar o desempenho das aplicações na *cloud*.

Novas linguagens de programação, evolução nos servidores de aplicação, ferramentas para melhorar a gerência de aplicações distribuídas, além de diversos padrões de projeto são desenvolvidos com o intuito de que cada vez mais dados possam trafegar em rede, e as aplicações possam comunicar-se de forma mais eficiente.

A arquitetura de contêineres, utilizada neste trabalho acadêmico, mostrou-se consistente e de aplicação pouco complexa. Muitas empresas estão migrando seus grandes sistemas monolíticos para este novo formato, aplicando o velho conceito “dividir para conquistar” na estrutura de microsserviços, e após isto, contêineres.

O Kubernetes está solidificando-se no papel de gerente de *clusters* de contêineres, pois já é a ferramenta oficial utilizada pelo Docker, desbancando o Docker Swarm. Este possui diversas funcionalidades que tem por intenção trazer melhor desempenho e estabilidade, automatizando tarefas que seriam complexas e trabalhosas quando se trata da gerência de microsserviços, como mapeamento de contêineres, entrega contínua, reparação de serviços falhos, entre outros.

Como perspectiva futura para este projeto, há o objetivo de implementar novas funcionalidades à aplicação, complementando com todo o fluxo necessário para um eficiente gerenciamento de arquivos, entregando o *software* nesta junção de Docker e Kubernetes, em que a intenção é melhorar aspectos falhos da entrega de serviços, como manutenção e estabilidade.

REFERÊNCIAS

ANDRADE, Joelson. **O que é um hypervisor?** 2018. Diponível em: < http://www.getcard.com.br/novo/o-que-e-um-hypervisor/> Acesso em: 14 set. 2018.

CAELUM. **O que é Java EE?** 2018. Disponível em: < https://www.caelum.com.br/apostila-java-web/o-que-e-java-ee/> Acesso em: 16 set. 2018.

CARISSIMI, Alexandre. **Virtualização: da teoria a soluções.** Em: 26º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2008, p. 174 – 206.

DIEDRICH, Cristiano. **Docker.** 2015. Disponível em: < https://www.mundodocker.com.br/o-que-e-docker/> Acesso em 18 set. 2018

Docker. Disponível em: < https://docs.docker.com/registry/> Acesso em 19 set. 2018

KUBERNETES. Conecpts. 2018. Disponível em <https://kubernetes.io/docs/concepts/>. Acesso em 27/11/2018

LUSA, Diego Antonio, et. Al. **Conhecendo o modelo arquitetural REST.** 2013. Disponível em <https://www.devmedia.com.br/conhecendo-o-modelo-arquitetural-rest/28052>. Acesso em 27/11/2018

MENDES, Pedro. **Microsserviços, por Martin Fowler e James Lewis,** 2016. Disponível em: < http://www.pedromendes.com.br/2016/01/02/microservicos/> Acesso em: 15 set. 2018

MICROSOFT. **O que é a cloud?**, 2018. Disponível em: < https://azure.microsoft.com/pt-pt/overview/what-is-the-cloud/>. Acesso em: 12 set. 2018.

SANTOS REIS, Diogo Cristiano dos. **Execução e Gestão de Aplicações Conteinerizadas.** Dissertação (Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos) - Faculdade de Ciências da Universidade do Porto. Porto, et. Al. 2017.

RED HAT. **O que é Docker?**, 2018. Disponível em: <https://www.redhat.com/pt-br/topics/containers/what-is-docker> Acesso em 19 set. 2018.

SOUZA, Leandro. **Containers—As primeiras pedras,** 2018. Disponível em: <https://medium.com/trainingcenter/containers-as-primeiras-pedras-b4b5efa49d88> Acesso em: 14 set. 2018.

TILKOV, Stefan. **Uma rápida introdução ao REST.** 2008. Traduzido por André Faria Gomes em 29 out 2008. Disponível em: < https://www.infoq.com/br/articles/rest-introduction/> Acesso em 21 nov. 2018.

TRUCCO, Cristian. **Tudo o que você precisa saber sobre Kubernetes,** 2018. Disponível em: <https://www.concrete.com.br/2018/02/19/tudo-o-que-voce-precisa-saber-sobre-kubernetes/> Acesso em 19 set. 2018