UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GABRIEL VASSOLER

ANÁLISE DE DESEMPENHO DE ROLLING UPDATES DO KUBERNETES EM AMBIENTES DE ESTRESSE

CHAPECÓ
2019

GABRIEL VASSOLER

ANÁLISE DE DESEMPENHO DE ROLLING UPDATES DO KUBERNETES EM AMBIENTES DE ESTRESSE

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Marco Aurélio Spohn

CHAPECÓ
2019

Vassoler, Gabriel

ANÁLISE DE DESEMPENHO DE ROLLING UPDATES DO KUBERNETES EM AMBIENTES DE ESTRESSE / Gabriel Vassoler. – 2019.

50 f.: il.

Orientador: Marco Aurélio Spohn.

Trabalho de conclusão de curso (graduação) – Universidade Federal da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2019.

1. Kubernetes. 2. Rolling Updates. 3. Desempenho. 4. Análise. 5. Estresse de Sistema. I. Spohn, Marco Aurélio, orientador. II. Universidade Federal da Fronteira Sul. III. Título.

E-mail: gabiv98@gmail.com

GABRIEL VASSOLER

ANÁLISE DE DESEMPENHO DE ROLLING UPDATES DO KUBERNETES EM AMBIENTES DE ESTRESSE

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Marco Aurélio Spohn

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em: 04/12/2019.

BANCA AVALIADORA

Marco A. Spohn

Marco Aurélio Spohn – UFFS

Emilio Wuerges

Braulio Adriano de Mello

AGRADECIMENTOS

É impossível colocar em palavras todo o apreço e agradecimento que tenho por todas as pessoas que participaram desta jornada comigo, é um grande conjunto de pessoas que espero estar junto durante toda a minha vida. Em específico, gostaria de primeiramente agradecer aos meus pais pelo auxílio durante toda a caminhada que já trilhei, e todo o incentivo nos estudos. Também gostaria de agradecer ao Prof. Dr. Marco Aurélio Spohn pela orientação durante este trabalho, e, em extensão, a todos os professores que tive durante a vida. Jamais teria chegado até aqui sem eles, e agradeço grandemente pelo incentivo e dedicação por todos estes anos. Finalmente, agradeço em especial a todos os meus amigos e família, que sempre estiveram ao meu lado em todos os momentos difíceis oferecendo refúgio, risadas, conselhos e apoio. É por conta de vocês que eu pude chegar até aqui e este trabalho também é conquista sua. Obrigado por tudo.

“A vida sem ternura não é la grande coisa.”

(José M. de Vasconcelos, Meu Pé de Laranja Lima)

RESUMO

Este trabalho visa analisar o desempenho de *Rolling Updates* do Kubernetes em ambientes com diferentes tipos de estresse nas máquinas do *cluster*. Além disso, buscou-se verificar o impacto que tal estresse durante a atualização gerou para os usuários dos serviços instanciados no sistema. Para isto, foram propostos ambientes de testes, cenários possíveis e parâmetros a serem alterados para a coleta de dados. Após a exposição dos dados coletados, foi realizada uma análise dos resultados obtidos, e uma discussão sobre o impacto que os tipos de estresse aplicados num sistema podem gerar no momento de uma atualização.

Palavras-chave: Kubernetes. Rolling Updates. Desempenho. Análise. Estresse de Sistema.

ABSTRACT

This paper aims to analyze the performance of Kubernetes Rolling Updates in environments with different types of stress on cluster machines. In addition, we sought to verify the impact that such stress during the update generated for users of services instantiated in the system. For this, test environments, possible scenarios and parameters for data collection were proposed. After exposing the collected data, an analysis of the results obtained was performed, and a discussion about the impact that the types of stress applied to a system can generate at the time of an update.

Keywords: Kubernetes. Rolling Updates. Analysis. Performance. System Overload.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O ciclo de desenvolvimento de *software* acelerou grandemente desde sua introdução, nos anos 90. Inicialmente, os *softwares* eram estaticamente desenvolvidos, entregues em mídias físicas e raramente atualizados, pois levava-se anos para aplicar novas alterações. Com o passar dos anos, novas tecnologias e técnicas de desenvolvimento auxiliaram na melhoria da velocidade de entrega, sendo o conceito de *DevOps* um deles. Primeiramente introduzido em 2009, esta nova visão do ciclo de desenvolvimento de *software* buscava deixar as tarefas mais rápidas e simples (25).

Outras técnicas, tal como Integração Contínua e Entrega Contínua também contribuíram para o rápido desenvolvimento de *software* com menos esforço. Uma outra maneira de desenvolver aplicações também ganhou destaque rapidamente: os microsserviços (25). A arquitetura de microsserviços, como apontado por Newman (23), tem como vantagens principais a possibilidade de fácil replicação e manutenção.

Os serviços pertinentes a execução do sistema são apenas um módulo, que podem ser replicados conforme a quantidade de acessos e facilmente isolados em caso de problemas. Cada módulo busca ser independente dos outros e reutilizável em outros projetos conforme a necessidade. Desta forma, uma aplicação apenas necessita importar os módulos que precisa, sem ter que se preocupar em controlar todos (25).

Em união a isso, a tecnologia de virtualização se tornou atrativa para o desenvolvimento de *software*. Cada microsserviço poderia estar em uma máquina virtual, e conforme a necessidade, instanciadas ou destruídas (25). Entretanto, com a criação dos *Linux Containers*, trabalhos como o de Felter et al. (12) e Joy (15) demonstraram que estes são mais eficientes que as máquinas virtuais presentes naquele momento.

O crescimento da conteinerização de aplicações criou a necessidade de serviços que poderiam organizar e monitorar os contêineres em produção. Tais *softwares*, chamados de orquestradores, têm como função manter o sistema num estado desejado, provido pelos desenvolvedores (28). Logo, aliados ao desenvolvimento e entrega contínuas de aplicações, se tornou necessária a implementação de tecnologias que fizessem a atualização sem que o sistema precisasse interromper sua execução, como é o caso das *Rolling Updates* do Kubernetes, um orquestrador de contêineres. Devendo esta atualização ser segura, sustentar transações já abertas e manter o sistema nas melhores condições de funcionamento em questão de latência até sua finalização (13).

Para a realização da atualização, existem parâmetros que podem ser alterados conforme necessidade, pelo gerente do sistema. Dentre eles, existem dois que foram o alvo deste trabalho, sendo *MaxSurge* e *MaxUnavailable*. Tais parâmetros indicam a quantidade de nós do serviço em execução que podem ser criados ou removidos de forma extra durante uma atualização. A explicação aprofundada de tais parâmetros se dará durante o desenvolvimento do conteúdo deste trabalho, entretanto, é importante deixar claro que eles têm importante papel numa atualização

em tempo de produção.

Além disso, para contornar possíveis problemas de uso de recursos durante uma atualização, existem técnicas como o provisionamento de recursos de forma reativa, que busca deixar uma quantidade de recursos disponível para ser utilizada em momentos que seja necessário. Assim, em casos de uso não regular, ou como objetivado por este trabalho, atualização do sistema, existam recursos suficientes para a realização das operações sem prejuízo (22). Neste trabalho, tais técnicas não serão utilizadas, visando um uso extremo de recursos, de forma a debilitar o sistema e estudar o impacto gerado.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Avaliar o desempenho em relação ao tempo das *Rolling Updates* do Kubernetes em situações de alto estresse do sistema, e encontrar os possíveis gargalos que impedem que esta funcionalidade se realize mais rapidamente.

1.1.2 Objetivos Específicos

- Avaliar e definir parâmetros que possam vir a ser relevantes e gerem impacto no desempenho do sistema.
- Avaliar ferramentas que auxiliem no processo de obtenção de dados dos serviços em funcionamento.
- Identificar e analisar a variação dos tipos de estresse do sistema.
- Propor e realizar testes de impacto dos diferentes tipos de estresse gerados no sistema, de forma a analisar seus resultados para encontrar os possíveis gargalos que viriam a interferir no funcionamento ideal da atualização.

1.2 JUSTIFICATIVA

Desde 2014, as aplicações conteinerizadas ganharam publicidade significativa, como preceitua Baier (3). Softwares como o Kubernetes, que realizam a orquestração dos contêineres provedores dos serviços dentro de um sistema, se tornaram conhecidos por garantir que os serviços continuem executando livres de falhas, como quedas ou indisponibilidade de serviço.

Hightower, Burns e Beda (13) dispõem que no princípio, a capacidade de realizar a atualização de contêineres dos serviços em tempo real, foi uma das características que mais deu espaço para o Kubernetes. Desta maneira, o serviço não precisaria sair de funcionamento para que novas funcionalidades ou correções fossem implementadas. Esta função, chamada

de *Rolling Update*, deve ser confiável e livre de falhas, para garantir que os novos contêineres assumam o lugar dos antigos, sem danificar serviços que já estavam sendo prestados a clientes ou perder requisições.

Destarte, é necessário saber até que ponto o sistema pode ser estressado, e de que maneiras esse estresse pode ser gerado, de forma possível a avaliar a capacidade de recuperação, bem como, avaliar o desempenho dessa configuração em comparação com o sistema em condições regulares.

2 REFERENCIAL TEÓRICO

2.1 MICROSSERVIÇOS

Microsserviços são serviços que realizam apenas uma operação e trabalham com outros microsserviços ou sistemas para fornecer uma aplicação (23). Normalmente, eles podem ser executados, escalados e testados de forma unitária (30). Os microsserviços são mais facilmente corrigidos em caso de *bugs* ou implementação de funcionalidades em contraste com os sistemas monolíticos, que possuem códigos espalhados por toda a plataforma e exigem, muitas vezes, que programadores precisem estudar o código por muitas horas para encontrar os problemas (23).

Segundo Newman (23), cada microsserviço é uma entidade separada, que pode ser instanciada em um serviço do tipo *Platform-as-a-Service*, o que resulta na simplicidade do sistema como um todo. Todas as comunicações entre os microsserviços são feitas através de chamadas de rede. Os microsserviços têm a autonomia de serem modificados sem precisar fazer mudanças em outros microsserviços, ou até mesmo na forma que o usuário final acessa o sistema.

Outra característica chave apontada por Newman (23) é a capacidade de escalar o serviço conforme a necessidade. Como cada microsserviço executa apenas uma função, e havendo um gargalo em uma funcionalidade, é possível criar mais instâncias dela para suprir as necessidades de conexão que estão sendo apresentadas. Entretanto, ainda é explicitado que nem sempre utilizar microsserviços é a tarefa mais fácil de ser implementada, principalmente quando se sai de um sistema originalmente monolítico.

2.2 VIRTUALIZAÇÃO

Com o crescimento da utilização de recursos provenientes do campo da tecnologia da informação, um grande número de empresas se utilizam de aplicações que são hospedadas em *data centers*. Estes, por sua vez, se utilizam extensivamente de máquinas virtualizadas, que são atribuídas a máquinas físicas sob demanda. A virtualização garante uma alocação flexível das aplicações, além de permitir que mais de um serviço rode em uma única máquina física ao mesmo tempo. A escalabilidade de serviços virtualizados é outro motivo que contribuiu para o crescimento acelerado do uso desta técnica. Uma aplicação precisa apenas ser desenvolvida para sustentar sua utilização em múltiplos computadores ao mesmo tempo, e pode ser instanciada em momentos de maior acesso ao serviço, bem como ser desativada quando a demanda diminui (26).

Sharma et al. (26) ainda define dois tipos de virtualização: a nível de *hardware* e a nível de sistema operacional. No primeiro tipo especificado, é necessária a atuação do hipervisor sobre o *hardware*, que segundo Joy (15) é um *software* que se propõe a isolar diferentes máquinas

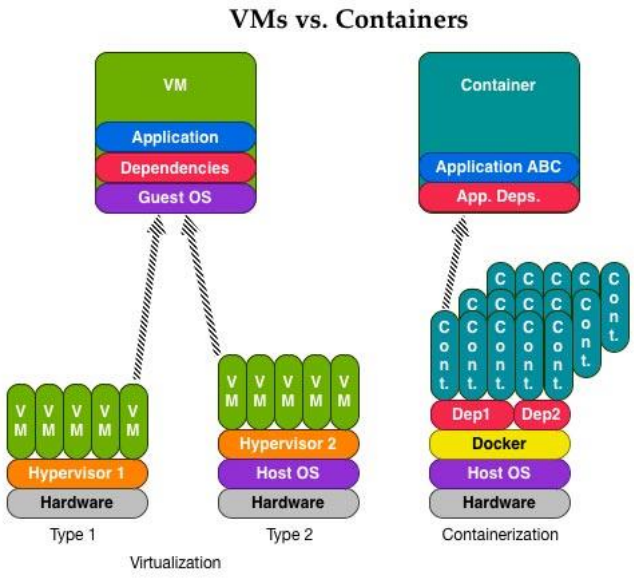
virtuais dentro de uma mesma máquina física. É esse software que se preocupa em executar diferentes núcleos para cada uma das máquinas virtuais ativas. Ressalta-se que esta prática é extremamente custosa e gera um impacto considerável no desempenho final das aplicações, além na utilização dos recursos. O segundo tipo de virtualização também realiza o uso de hipervisores, contudo, eles são executados sobre um sistema operacional anfitrião, como apontado por Merkel (21).

Com os avanços nas tecnologias de virtualização, surgiu um outro tipo a nível de sistemas operacionais: os contêineres (15).

2.3 CONTÊINERES

Contêineres, diferentemente dos tipos já apresentados de virtualização, utilizam apenas um núcleo, que é compartilhado por todos os processos rodando de forma isolada. Desta maneira, duas aplicações utilizando o mesmo núcleo não tem conhecimento que estão utilizando recursos compartilhados (21). Assim, como apontado por Joy (15) e Sharma et al. (26), esta abordagem é muito mais leve se comparada com máquinas virtuais, já que não precisa realizar a virtualização do *hardware* para cada uma das aplicações. O *software* em si apenas necessita de suas dependências específicas, e tudo o que é de encargo do sistema operacional é responsabilidade do programa que hospeda o serviço. Estes programas vêm se tornando muito populares, como é o caso do Docker. Pode-se ter uma visão geral da diferença da arquitetura de contêineres e dos dois tipos de virtualização na Figura 1.

Figura 1 – Comparação de arquitetura entre máquinas virtuais e contêineres



Fonte – Merkel, 2014

Em questão de desempenho, os contêineres se demonstraram superiores em diversos sentidos, como apresentado nos estudos de Felter et al. (12) e Joy (15). Além disso, a alta

capacidade de escalabilidade e a boa utilização de recursos contribuíram grandemente para a rápida adoção deste tipo de técnica.

2.4 DOCKER

O Docker, como apontado por Anderson (2), é uma tecnologia para virtualização de contêineres. Além de prover a capacidade de criar contêineres, a ferramenta se preocupa em auxiliar os desenvolvedores a fazer isso da maneira mais fácil possível.

Segundo sua própria documentação (7), o Docker se utiliza de vários objetos para o funcionamento das aplicações. Sendo um deles as imagens, elas são *templates* que possuem informações pertinentes à criação dos contêineres que serão executados. A documentação aponta, ainda, que normalmente as imagens são embasadas em outras, com apenas algumas modificações e detalhes necessários à aplicação que está sendo executada.

Ainda é apontado em sua documentação (7) que o Docker se utiliza de *namespaces* para fazer a separação dos contêineres. Cada aspecto do contêiner roda em um *namespace* diferente, por exemplo, existe um *namespace* para acesso aos recursos de rede e outro para comunicação entre processos. Processos apenas podem ver e se comunicar diretamente com outros processos dentro do mesmo *namespace*, o que garante uma maior isolamento entre os processos em diversos contêineres executando em uma máquina. Além disso, o Docker também se utiliza de *control groups* (*cgroups*) que limitam a aplicação a um determinado valor em algum recurso. Por exemplo, através dos *cgroups* é possível limitar a quantidade de memória que um contêiner pode utilizar.

2.5 ORQUESTRAÇÃO DE CONTÊINERES

Quando se trabalha com aplicações que necessitam de mais de um contêiner para executar, é interessante que exista um serviço que seja responsável pela interação deles. Desta forma, os orquestradores servem para escalonar os contêineres dentro de máquinas do *cluster* de trabalho, escalar o sistema quando necessário e gerenciar sua comunicação (28).

Khan (16) define sete características chave essenciais para o funcionamento de um orquestrador de contêineres, sendo elas:

- Gerenciamento e escalonamento do *cluster*: Manter o *cluster* estável é importante para que as operações consigam ser executadas de forma correta. O orquestrador deve suportar tarefas de manutenção e ativá-las no *cluster* quando necessário. Também deve manter o estado definido pelos desenvolvedores de forma confiável e gerenciar os recursos disponíveis, além de comunicar serviços que sejam dependentes de mudanças relevantes ao seu funcionamento (16).

- Alta disponibilidade e tolerância a falhas: O orquestrador deve manter um nível de desempenho aceitável, eliminando pontos de falha, criando redundância de recursos e detectar falhas quando elas acontecem. Técnicas como balanceamento de carga são apontadas para que os recursos sejam melhor utilizados e minimizar o tempo de resposta (16).

- Segurança: É necessário que o orquestrador mantenha o *cluster* seguro contra possíveis ataques, implementando mecanismos de controle para acesso de determinados usuários, aceitando apenas imagens assinadas e que possuam registro confiável. Além disso, o orquestrador precisa garantir que os contêineres em execução apenas utilizem os recursos que foram destinados a eles, e serem imunes a problemas que possam ocorrer no *kernel* que está executando abaixo deles (16).

- Comunicação: A plataforma de orquestração também deve prover uma comunicação eficiente e segura para os contêineres que estão executando. Deve garantir que os serviços em execução não sejam prejudicados em sua comunicação em situação de larga escala e dinamicamente associar recursos quando necessário (16).

- Descoberta de Serviço: Conforme o serviço executado é escalado, novos contêineres receberão portas e IP's diferentes. Logo, para se acessar um determinado recurso sendo executado no *cluster*, é necessário que exista um tipo de sistema que encontre as instâncias que estão executando esse serviço e redirecione o tráfego de forma correta (16).

- Entrega e integração contínuas: O orquestrador deve fornecer ferramentas que permitam a modificação contínua das aplicações executadas, de maneira segura e rápida. O código deve ser desenvolvido de forma ininterrupta, e ao final de cada adição, o código deve estar pronto e em produção, sem que o sistema precise sofrer paradas (16).

- Monitoramento: Além de todos os tópicos já abordados, o orquestrador precisa manter um sistema de monitoramento do *cluster*. Os sistemas de monitoramento devem medir e registrar o desempenho de diversos recursos pertinentes aos contêineres, tal como utilização de rede, CPU e memória, e, com base nessas informações, agir da melhor maneira possível para garantir o funcionamento dos serviços sem problemas (16).

2.6 KUBERNETES

Originado do Borg (5), o Kubernetes é uma plataforma de orquestração de contêineres de código aberto. Segundo seu próprio site (17), o Kubernetes oferece um ambiente de gerenciamento dos serviços a serem disponibilizados, controlando onde as aplicações serão executadas, como se comunicarão com os usuários e garantindo a estabilidade do serviço. O Kubernetes, além de prover esses serviços, possui diversas soluções para avaliar o desempenho e monitorar

o estado dos nós. Inicialmente desenvolvido pela Google Inc, esta ferramenta é atualmente mantida pela Cloud Native Computing Foundation.

Diferenciando-se de outros serviços do tipo *Platform-as-a-Service*, o Kubernetes, apesar de oferecer todos as opções padrão desse tipo de sistema (tal como balanceamento de carga, escalabilidade e monitoramento) não obriga o usuário a utilizar todas as funcionalidades. Os serviços são ativados e utilizados de acordo com as decisões do desenvolvedor (17).

2.6.1 Estrutura

Esta seção tem como objetivo explicar a estrutura do Kubernetes necessária para o desenvolvimento desta monografia e o funcionamento de cada um de seus componentes, tendo como base a documentação da ferramenta (17).

2.6.1.1 *Cluster*

O *cluster* de uma ou mais aplicações é o conjunto de máquinas que são gerenciadas pelo Kubernetes. Todas elas possuem uma instância da aplicação 'kubelet', que é o responsável por se comunicar com os outros nós e realizar ações no nó em que se encontra (17).

2.6.1.2 Nodes (nós)

Cada nó representa uma máquina física dentro do *cluster* do Kubernetes. Cada nó pode ser do tipo *Master* ou *Worker (Minion)* (17).

2.6.1.2.1 Master Node (Nó Mestre)

Os *Master Nodes* são um ou mais nodos especiais executando componentes do Kubernetes que tratam do gerenciamento dos nós. São esses nós que normalmente se acessa para realizar algum tipo de configuração no *cluster*, como por exemplo, alterar seu estado desejado (*deployment*). Os componentes executados nos nós *Master* são explicitados abaixo como descrito em Kubernetes (17), exceto quando apontado. A arquitetura pode ser melhor visualizada na Figura 2.

- kube-apiserver: Esse é o componente que expõe a API do Kubernetes. É através dele que se entra em contato com o *cluster* e se realizam as alterações desejadas.
- etcd: Esse é o componente que armazena as informações cruciais do *cluster*. Nele encontram-se armazenadas as situações dos contêineres, o estado desejado e falhas que vieram a ocorrer durante a execução. De fato, o etcd é um projeto à parte do Kubernetes, desenvolvido para ser um local de armazenamento de dados de sistemas distribuídos, como apontado em seu repositório no Github (11).

- kube-scheduler: Esse componente é o responsável por atribuir os *pods* a um nó. Um desenvolvedor pode optar por reescrever esse componente para uma aplicação conforme a necessidade. Seu funcionamento padrão leva em consideração diversos fatores, como recursos necessários para determinados contêineres, localidade de dados ou uso de *hardware* atual dos nós para realizar sua escolha.

- kube-controller-manager: É o componente que executa os controladores do *cluster*. De fato, existe mais de um controlador, entretanto são todos compilados em apenas um binário. Exemplos de controladores existentes nesse componente são o controlador de nós, que verifica e notifica quando um nó deixa de funcionar, e o controlador de replicação, que é o responsável por sempre manter o número de *pods* correspondente ao estado ideal (*deployment*) provido para a aplicação.

2.6.1.2.2 Minion/Worker Node (Nós Trabalhadores)

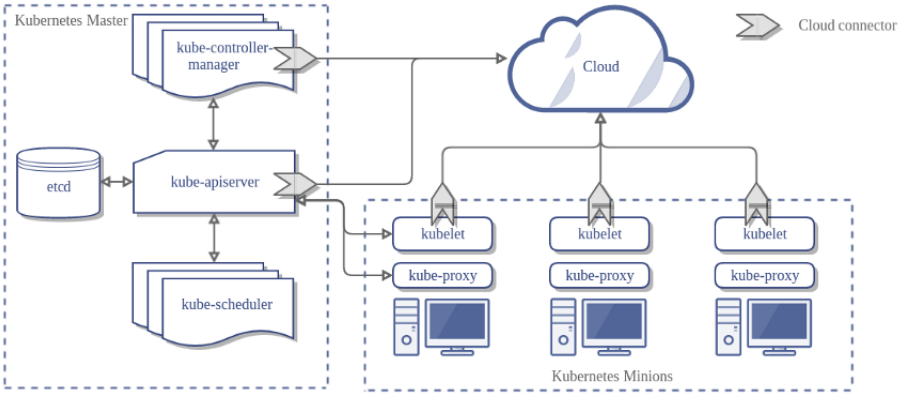
Os *Minion/Worker Nodes* são onde ficam hospedados os contêineres que possuem a aplicação que o desenvolvedor deseja executar. Todos os nós trabalhadores precisam dos processos de gerenciamento do Kubernetes e um software que possa rodar os contêineres. Os agentes presentes nos nós trabalhadores, como dispostos em Kubernetes (17), são:

- kubelet: Esse é o agente que assegura que os contêineres instanciados pelo Kubernetes estão dentro de um *pod*. Ele garante, através de diversos recursos, que os contêineres estão executando sem problemas. É importante ressaltar que ele não gerencia contêineres não instanciados pelo Kubernetes.

- kube-proxy: É o *proxy* de rede do Kubernetes. Ele aceita e replica informações passadas para o *cluster* todo. Este componente será melhor apresentado na sequência, por possuir outras características relevantes para este trabalho.

- Gerenciador de Contêineres: Como explicitado anteriormente, os nós precisam de um gerenciador para executar os contêineres. O Kubernetes suporta diversos desses *softwares*, como por exemplo, o Docker.

Figura 2 – Estrutura dos componentes dos nós do Kubernetes



Fonte – Kubernetes, 2019

2.6.1.3 *Deployments*

Tendo em mãos o *cluster* e as imagens referentes às aplicações que se deseja instanciar, o passo seguinte é informar ao Kubernetes como ele deve se comportar em relação a eles. Os *deployments* são a maneira de estabelecer o estado desejado do *cluster*, ou seja, quais e quantos contêineres devem ser instanciados em quais máquinas (17).

Após definido o estado desejado, o *deployment* é aplicado e o Kubernetes trata de reservar os recursos, criar os *pods* necessários e designar as imagens para as máquinas escolhidas. Os *deployments* também podem especificar a remoção de *pods* já existentes, destruindo contêineres instanciados conforme necessário (17).

Ao criar o *deployment*, um outro objeto é instanciado: o *ReplicaSet*. Ele é o responsável por manter os *pods* definidos no *deployment* ativos. Esse componente é o agente que de fato enviará os comandos de criar, destruir ou alterar os *pods* definidos (17).

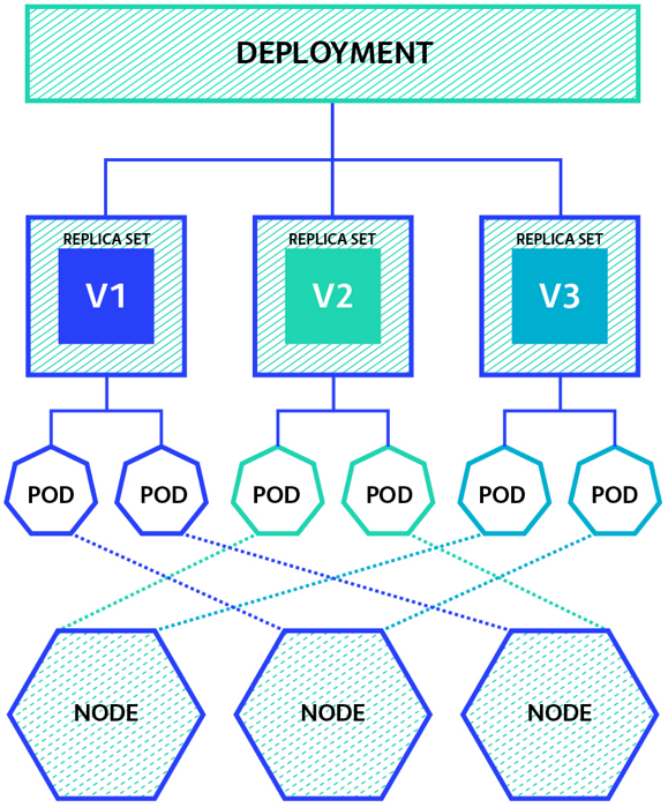
2.6.1.4 *Pods*

Um *pod* é a unidade básica do Kubernetes, que representa os processos que estão executando no *cluster*. Eles são um conjunto de um ou mais contêineres instanciados que compartilham rede e armazenamento. Contêineres dentro de um *pod* possuem apenas um endereço IP e porta (17).

Os *pods* são substituíveis, o que significa que ao menor sinal de problemas eles podem ser destruídos e instanciados novamente. Além disso, os *pods*, apesar de possuírem um IP e porta únicos e estarem prontos para serem acessados fora da rede interna do Kubernetes, normalmente não são expostos por serem demasiadamente voláteis (17).

Os *pods* podem abrigar um ou mais contêineres, sendo executados dentro de uma ou mais máquinas (17). Tal arquitetura pode ser visualizada na Figura 3.

Figura 3 – Estrutura dos *pods* do Kubernetes



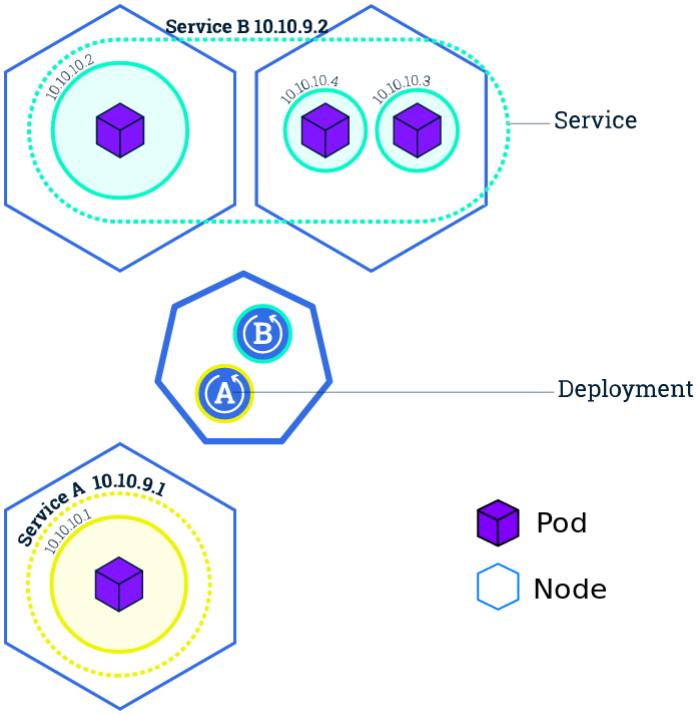
Fonte – Hoogendoorn, 2017

2.6.1.5 *Services* (Serviços)

Tendo em mente que os *pods* são facilmente criados e destruídos com IP's e portas diferentes, faz-se necessária a inserção de uma abstração que possua um IP e porta que são imutáveis e garantam acesso a um conjunto de *pods*. Esta abstração, chamada de *service*, serve para que o desenvolvedor final não precise alterar o IP/porta do seu serviço toda vez que algum problema ou atualização é lançada (17).

Os *services* encapsulam um ou mais *pods* e os garantem um IP único, pelo qual agentes externos poderão acessar os recursos do componente (17). Esta abstração pode ser visualizada na Figura 4.

Figura 4 – Estrutura dos Services do Kubernetes



Fonte – Adaptado de Kubernetes, 2019

2.6.1.6 *Rolling Updates* (Atualizações em Tempo de Produção)

Possuindo um *deployment* ativo em um serviço, é possível especificar outro estado desejável para o *cluster* por meio do Kubernetes, para que ele faça a atualização sem que o serviço saia do ar. Esse recurso, chamado de *Rolling Update*, substitui gradativamente os *pods* ativos por novos *pods*, e é importante que as conexões que já estavam abertas não sejam instantaneamente encerradas, mas que terminem suas requisições em tempo para depois o *pod* ser encerrado (17).

Quando um *pod* entra em estado de finalização, o Kubernetes para de rotear novas conexões para ele e dá um tempo limite para que suas transações sejam encerradas. Ao final do tempo, o *pod* é excluído e outro toma seu lugar (17).

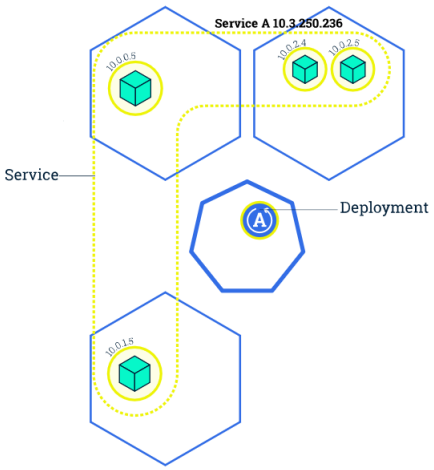
Existem duas variáveis que podem ser alteradas para melhor controle dos recursos durante uma *Rolling Update*, sendo eles a quantidade máxima de *pods* que podem estar indisponíveis durante a atualização, e o máximo de *pods* que podem ser criados acima do limite superior de *pods* especificados pelo *deployment*. Também é possível escolher uma porcentagem da quantidade total de *pods* para qualquer uma das variáveis acima. Desta maneira, é possível escolher o quanto o Kubernetes pode ocupar de recursos das máquinas anfitriãs enquanto passa por uma atualização (17).

2.6.1.6.1 Fases da atualização

Quando se inicia uma *rolling update*, o sistema passa pelas seguintes fases:

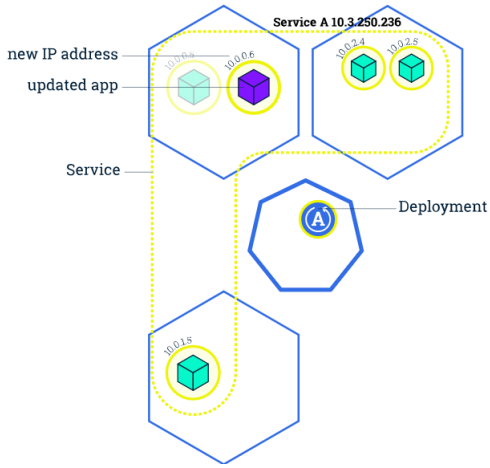
- Fase inicial: Quando nenhum *pod* foi alterado. Essa é a situação que o serviço se encontra, como demonstrado na Figura 5.
- Fase de troca: Quando os *pods* estão sendo trocados. Como ilustrado na Figura 6, Cada *pod* é instanciado com um IP e porta novos.
- Fase final: Quando todos os *pods* foram atualizados e a atualização chegou ao seu final, como se pode ver na Figura 7.

Figura 5 – Estado do serviço no início da *rolling update*



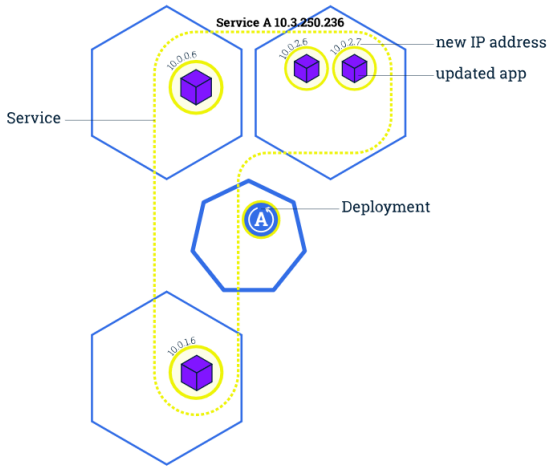
Fonte – Adaptado de Kubernetes, 2019

Figura 6 – Estado do serviço durante a *rolling update*



Fonte – Adaptado de Kubernetes, 2019

Figura 7 – Estado do serviço ao final da *rolling update*



Fonte – Adaptado de Kubernetes, 2019

2.6.1.7 Kube-Proxy

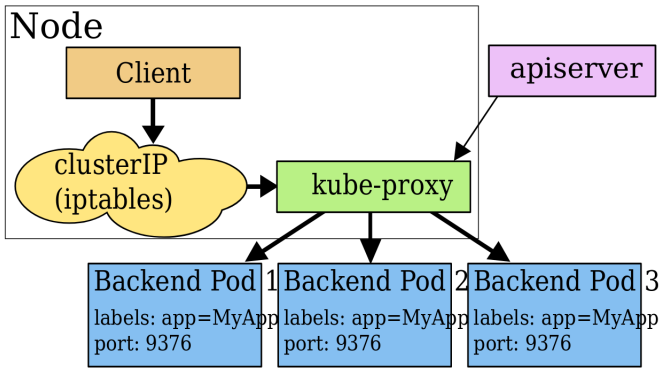
Uma vez criado o serviço, todo o tráfego passará por ele. Logo, faz sentido que o serviço possua alguma tecnologia que realize o balanceamento de pedidos entre os *pods* ativos. Um dos recursos implementados para isto é o *kube-proxy*, que tem como função garantir que as requisições enviadas para um serviço sejam corretamente destinadas para os *pods* necessários (24).

O *kube-proxy* pode ser executado de três maneiras diferentes, como descrito em Kubernetes (17):

- *User Space mode*: Nesse modo, o *kube-proxy* fica verificando a criação de novos *services* e cria uma porta única para cada um deles no nó onde está sendo executado. Desta

forma, cada conexão para a porta do serviço será roteado para um *pod* do serviço, por *round-robin* no modo padrão. Existe ainda, antes do nó ser conectado com a requisição do usuário, uma tabela de IP's, que decidirá para qual nodo a requisição será enviada, capturando as requisições que chegam no IP e porta do serviço dentro do *cluster*. Pode-se ver a arquitetura desse modo na Figura 8.

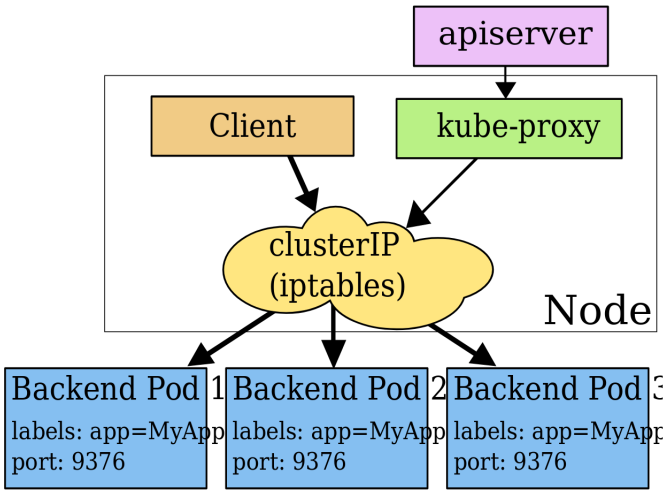
Figura 8 – Arquitetura do *kube-proxy* no modo *User Space*



Fonte – Kubernetes, 2019

- *Iptables mode*: Diferentemente do modo anterior, uma *iptable* é implementada com os IP's diretos de cada um dos *pods* dos serviços e todas as requisições enviadas para o serviço são capturadas pela tabela e roteadas (randomicamente, no modo padrão) para um dos *pods* ativos. Nesse modo, caso a requisição não chegue no *pod* por conta dele ter falhado, ela não é restaurada. Logo, apenas os *pods* listados como saudáveis são escolhidos no roteamento, o que não acontecia no modo *user space*, onde as solicitações que falhavam eram redirecionadas quando o sistema percebia a falha. Sua arquitetura está ilustrada na Figura 9.

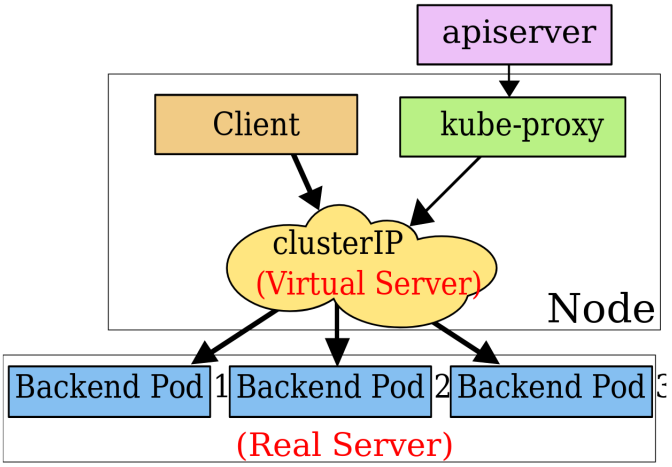
Figura 9 – Arquitetura do *kube-proxy* no modo *Iptables*



Fonte – Kubernetes, 2019

- IPVS: Funcionando de maneira similar ao modo *iptable*, o IPVS controla todo o tráfego para os nós, entretanto, por utilizar-se do *kernel* do linux (e não uma solução alto nível como as apresentadas anteriormente) e de uma estrutura mais rápida (tabelas *hash*), o IPVS é apontado, no momento desta publicação, como a solução mais viável atualmente para o balanceamento de requisições para os *pods*. Esse modo ainda suporta vários tipos de funcionamento, como escolha do *pod* com menos requisições abertas e menor tempo de *delay* como diferencial. Na Figura 10 pode-se ver a estrutura desse modo, sendo muito similar ao anterior.

Figura 10 – Arquitetura do *kube-proxy* no modo IPVS



Fonte – Kubernetes, 2019

2.6.2 Trabalhos relacionados

Ao realizar a revisão bibliográfica necessária para a realização deste trabalho, utilizou-se os mecânismos de busca *Google Scholar, IEEE e ACM Digital Library*. Infelizmente, nenhum dos trabalhos encontrados se propôs a realizar um teste de desempenho do Kubernetes e do serviço prestado durante uma atualização em tempo de produção sob um ambiente de estresse. Entretanto, publicações relacionadas foram encontradas e estão dispostas a seguir:

- Os Trabalhos de Medel et al. (20, 19) criam um modelo de referência para o desenvolvimento de aplicações utilizando o Kubernetes de forma a utilizar os recursos disponíveis da maneira mais eficiente o possível.
- O trabalho de Düllmann e van Hoorn (9) teve como resultado final uma plataforma que aceita uma arquitetura de microsserviços e gera microsserviços sintéticos que podem ser monitorados enquanto problemas são inseridos no ambiente de execução.
- Outro trabalho de Düllmann (8) cria um modelo para detecção de anomalias num ambiente de microsserviços, de forma a utilizar *logs* de eventos para melhorar os resultados, não utilizando falsos positivos, tais como instanciação de novos microsserviços, que geram leves instabilidades na utilização total de recursos.
- O trabalho de Dupenois (10) trata de um problema nas atualizações em tempo de produção do Kubernetes quando se executam conexões HTTP persistentes. O Kubernetes apenas as desativa quando não há trafego de dados, mas gera erro para o outro lado da conexão. O problema foi resolvido fazendo com que existam pelo menos mais duas réplicas do *pod* que vai ser desativado e tratando o sinal de desligamento nos contêineres, não apenas os encerrando.

3 METODOLOGIA

A metodologia proposta para este estudo foi a criação de uma série de ambientes similares ao de uma aplicação real para a realização de testes de desempenho da ferramenta. Tais testes tiveram como objetivo a coleta de dados de latência de resposta, utilização de memória, processador e tempo de realização da atualização nas máquinas anfitriãs do *cluster*. Para a execução dos testes também foi realizado o desenvolvimento de uma aplicação simples que retorna uma página web ao ser acessada.

Dado o cenário e a aplicação propostos, foram encontradas ferramentas (descritas nas seções a seguir) que auxiliassem no processo de geração do estresse do sistema. O estresse, então, pôde ser gerado através da criação artificial de usuários que fizeram requisições ao sistema e a limitação da utilização de recursos pelo Kubernetes nas máquinas anfitriãs, por meio de *pods* executando aplicações com esta finalidade.

Para a coleta de dados, além das informações geradas pelos usuários artificiais (latência de resposta), foram utilizadas as informações de utilização de recursos dos nós do Kubernetes, obtidos através da própria ferramenta e por meio de medições de programas externos, dispostos a seguir.

No âmbito da análise de dados, foram avaliados dados gerados pelo *cluster* durante uma atualização num ambiente com pouca carga, em que os recursos não estão sendo severamente utilizados, e um ambiente totalmente estressado de determinadas maneiras a serem expostas. Os dados foram comparados para verificar a mudança de desempenho da ferramenta e o quanto os usuários foram afetados.

3.1 CENÁRIOS DE AVALIAÇÃO

Nesta seção são apresentadas as máquinas e parâmetros utilizados para a realização da coleta de dados. Além disso, são expostas as ferramentas necessarias para auxiliar esta função, e suas respectivas peculiaridades.

3.1.1 Equipamentos Físicos e Versões do Kubernetes

Para a realização dos testes, com objetivo de coletar os dados para análise, criou-se um *cluster* com quatro máquinas anfitriãs, sendo um nó-mestre e três nós-trabalhadores. As especificações das máquinas, bem como as versões do Kubernetes instaladas em cada uma delas, estão dispostas a seguir na Tabela 1. Cabe ressaltar que todas as máquinas utilizaram o sistema operacional Ubuntu, na versão 18.04.

Tabela 1 – Configurações das máquinas utilizadas

Identificação	Processador	Clock (GHz)	Memória (GB)	Versão do Kubernetes
Master-Node	Intel i5-3470	3.2*4	8	1.16.0
Slave-Node-1	Intel i5-3470	3.2*4	8	1.15.3
Slave-Node-2	Intel i7-3770	3.4*8	8	1.16.2
Slave-Node-3	Intel i7-3770	3.4*8	8	1.16.2

Fonte – Do Autor

Também é pertinente expor que o Kubernetes, por padrão, não permite que nenhuma das máquinas do *cluster* possua a função de *swap* ativa. Isto se dá pelas premissas da ferramenta, que propõe que as operações realizadas sejam o mais eficientes possíveis, reduzindo o tempo de busca e armazenamento de dados voláteis. Além disso, com o *swap* ativo não é possível predizer desempenho, latência ou quantidade de acessos necessários para recuperar informações (18).

3.1.2 Parâmetros utilizadas

Os parâmetros alterados para a coleta de dados são dois, dispostos e definidos previamente à atualização em tempo de produção. Tais parâmetros estão dispostos a seguir, com suas respectivas funções.

- **MaxSurge (MS)**: Este é um campo opcional que pode ser adicionado no arquivo que dará origem a uma atualização, e que define quantos *pods* além do número desejado estabelecido pelo *deployment* podem ser criados. Por exemplo, suponha que o *deployment* define que 10 é o número desejado de *pods* a ser alcançado. Definindo o *MaxSurge* como 5, o Kubernetes pode criar até 5 *pods* além destes 10, somando um total máximo de 15 *pods* ativos durante a transição. Desta forma, quando uma definida quantidade X de *pods* for alcançada (maior que 10), os X-10 *pods* antigos serão destruídos para voltar a quantidade desejada, com a nova versão. Esta ação será repetida até que todos os *pods* antigos sejam substituídos (17).

- **MaxUnavailable (MU)**: De maneira similar à variável anterior, este é um campo opcional que pode ser adicionado no arquivo que dará origem a uma atualização, e que define quantos *pods* aquém do número desejado estabelecido pelo *deployment* podem ser destruídos. Por exemplo, suponha que o *deployment* define que 10 é o número desejado de *pods* a ser alcançado. Definindo o *MaxUnavailable* como 5, o Kubernetes pode destruír até 5 *pods* abaixo destes 10, somando um total máximo de 5 *pods* ativos durante a transição. Destarte, quando uma definida quantidade X de *pods* for alcançada (menor que 10), os 10-X *pods* antigos serão destruídos para voltar a quantidade desejada, com a nova versão. Esta ação será repetida até que todos os *pods* antigos sejam substituídos (17).

Estes dois parâmetros podem assumir valores absolutos ou porcentagens. O valor padrão de ambos é 25%. A partir destes parâmetros foram montados cinco configurações possíveis para as atualizações. Tais configurações podem ser verificados na Tabela 2.

Tabela 2 – Configurações para avaliação

Identificação	MaxSurge(MS)(%)	MaxUnavailable(MU)(%)
Configuração 1	100%	0%
Configuração 2	50%	0%
Configuração 3	50%	50%
Configuração 4	0%	50%
Configuração 5	0%	100%

Fonte – Do Autor

3.1.3 Ferramentas Auxiliares Utilizadas

3.1.3.1 Stress-ng

O Stress-ng (27) é um programa disponível nos repositórios padrões do sistema operacional Ubuntu (31), que é utilizado para criar vários tipos de estresse em um sistema. Os testes, como apontados por sua página oficial (27) vão desde sobrecarga de memória, CPU e I/O, até monitoramento de temperatura e uso de recursos. Os testes são realizados utilizando trabalhadores, que são processos monitorados pelo processo principal, e exercem o tipo de função necessária. Por exemplo, num teste de sobrecarga de CPU, o Stress-ng cria processos que irão utilizar uma definida porcentagem de recursos até o estabelecido. Tais quantidades de sub-processos podem ser definidas no momento de execução do programa.

Para este trabalho, o Stress-ng foi executado dentro de pods em cada um dos nós do *cluster* instanciado. O acesso ao programa foi realizado por meio de um terminal de acesso remoto. Os testes utilizados da ferramenta foram o de sobrecarga de CPU e memória. No teste de memória, em específico, é possível definir a quantidade de bytes a ser utilizada pelo programa. No teste de CPU, é possível especificar quantos núcleos do processador serão estressados na máquina anfitriã.

3.1.3.2 SAR

Sendo parte do pacote SysStat (29), o SAR é um comando que pode ser usado para medir a utilização de recursos em qualquer sistema operacional Linux. As métricas coletadas por este comando vão desde utilização de CPU, memória, rede e uso de disco, até de energia e atividade em terminais do sistema. Atualmente, o pacote SysStat (29) faz parte dos repositórios oficiais do Ubuntu (31) e de diversos outros sistemas operacionais.

No presente trabalho, o SAR foi utilizado para coletar os dados de uso de memória e CPU do sistema durante as atualizações em tempo de produção. O comando foi instanciado dentro do próprio sistema operacional anfitrião, de forma que fosse possível ter a utilização total dos recursos da máquina.

3.1.3.3 cURL

O cURL, ou *Client for URLs*, como apontado por seus manuais (6), é uma ferramenta para transferir dados de ou para um servidor específico. Ele pode trabalhar com os mais diversos protocolos, tais como, mas não limitado a: HTTP, HTTPS, FTP, POP3 e IMAP. Esta ferramenta foi criada com o intuito de trabalhar com URLs e requisitar ou enviar dados para serviços ou sistemas. Um de seus usos mais difundidos é a obtenção de dados, na íntegra, de páginas WEB, apenas digitando a URL desejada.

Uma função disponível na ferramenta é a possibilidade de consultar o tempo total de uma requisição feita para determinado serviço, de forma a descobrir a latência de acesso ao sistema. Este programa foi utilizado desta forma neste trabalho, com a finalidade de obter a latência do serviço para os usuários do sistema.

3.1.4 Avaliação de Desempenho

3.1.4.1 Métricas

Abaixo estão explicitadas as métricas utilizadas para a análise de desempenho das *Rolling Updates* do Kubernetes. Tais métricas objetivam encontrar as divergências do sistema entre uma configuração regular e uma configuração sobrecarregada. Todas as métricas analisadas foram submetidas a um cálculo de grau de confiança, estabelecido em 95%, de forma a avaliar a possibilidade de alteração da média destes valores.

3.1.4.1.1 Latência de Resposta

A latência de um serviço diz respeito ao tempo total da troca de bits entre uma origem e um destino. Tal medida leva em conta o tempo do envio do primeiro bit trocado pela origem, até o tempo do recebimento do último bit enviado pelo destino. A latência é um ótimo indicativo para congestionamento e uso de determinada rede, além de poder medir, no caso deste trabalho, o tempo extra levado por um sistema para responder os usuários ao sofrer alterações, o que remete diretamente à Qualidade de Serviço *QoS* (1).

3.1.4.1.2 Latência de Atualização

A latência ou tempo total de realização da atualização leva em conta o tempo do início da atualização, ou seja, o momento que o pedido de alteração foi enviado ao *cluster*, até o momento que o último novo *pod* for iniciado. A latência de atualização é uma métrica importante para medir a velocidade que o *cluster* consegue realizar as alterações requisitadas e o tempo necessário para se alcançar o novo estado desejado.

3.1.4.1.3 Utilização de Recursos nas Máquinas Anfitriãs

A medição da utilização de recursos nas máquinas do *cluster* é útil para definir a quantidade necessária dos recursos em uma situação regular de atualização e quanto os outros recursos são sobrecarregados em função da utilização exacerbada de um ou mais deles. Além disso, é interessante a adição desta métrica para se obter uma noção de quanto o impacto do recurso em questão realiza no andamento normal das operações do *cluster*.

4 RESULTADOS E ANÁLISE

Nesta seção, serão apresentados os resultados obtidos a partir dos testes realizados no *cluster*, com as configurações definidas na Tabela 2. Cada resultado será apresentado com sua variante do tipo de estresse aplicado, em comparação com o ambiente em situação regular. A configuração de estresse de memória obteve resultados concisos, que serão apresentados ao final desta seção.

4.1 CONFIGURAÇÃO 1

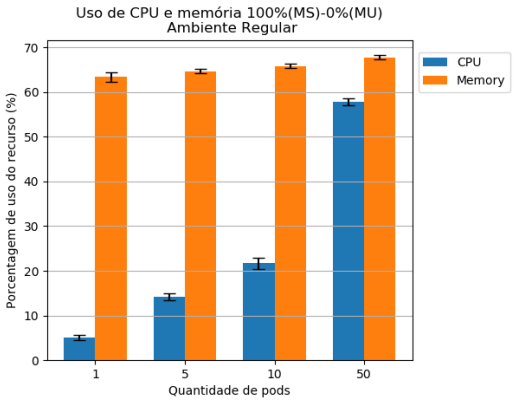
Conforme supraexposto, estão os resultados da primeira configuração testada, utilizando-se a variação que possui a variável MaxSurge (MS) no valor de 100%, e a variável MaxUnavailable (MU) no valor de 0%.

4.1.1 Utilização de Recursos dos Nós do *Cluster*

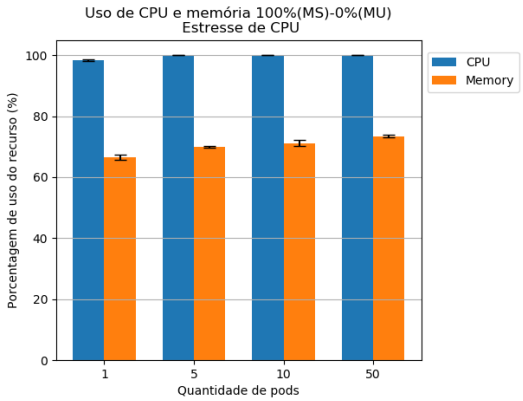
Conforme é possível se observar na Figura 11, o uso do tempo do processador pelas máquinas do *cluster* em um ambiente regular aumenta gradativamente, conforme a quantidade de *pods* instanciados pelo serviço. Em contrapartida, o uso de memória aumenta de pouco mais de 60% para quase 70% em uso regular, e fica em torno de 70% num ambiente com processador estressado.

Figura 11 – Comparação de uso de recursos na configuração MS 100% e MU 0%

(a) Uso de recursos em estado regular



(b) Uso de recursos com estresse de CPU



Fonte – Do Autor

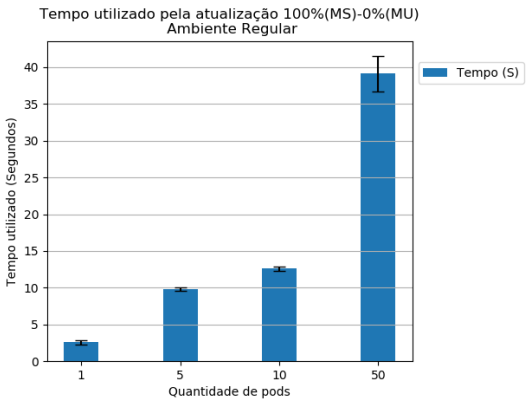
4.1.2 Latência de Atualização

Na Figura 12 é possível observar que utilizando apenas um *pod*, o tempo da atualização nesta configuração em ambiente regular é de aproximadamente 3 segundos, enquanto sob

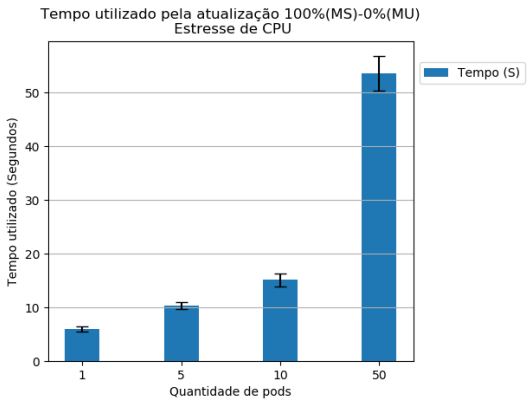
estresse de CPU o valor sobe para aproximadamente 5 segundos. O tempo permanece similar utilizando-se 5 *pods* em ambas configurações, cresce de 13 para aproximadamente 15 segundos com 10 *pods* e sobe de 39 para 53 segundos com 50 *pods*. É importante ressaltar que conforme o intervalo de confiança estabelecido, a variação de tempo cresce consideravelmente conforme se aumenta a quantidade de *pods*, e é mais alta num ambiente com processador sob estresse.

Figura 12 – Comparação de uso de tempo na configuração MS 100% e MU 0%

(a) Tempo médio da atualização em estado regular



(b) Tempo médio da atualização com estresse de CPU



Fonte – Do Autor

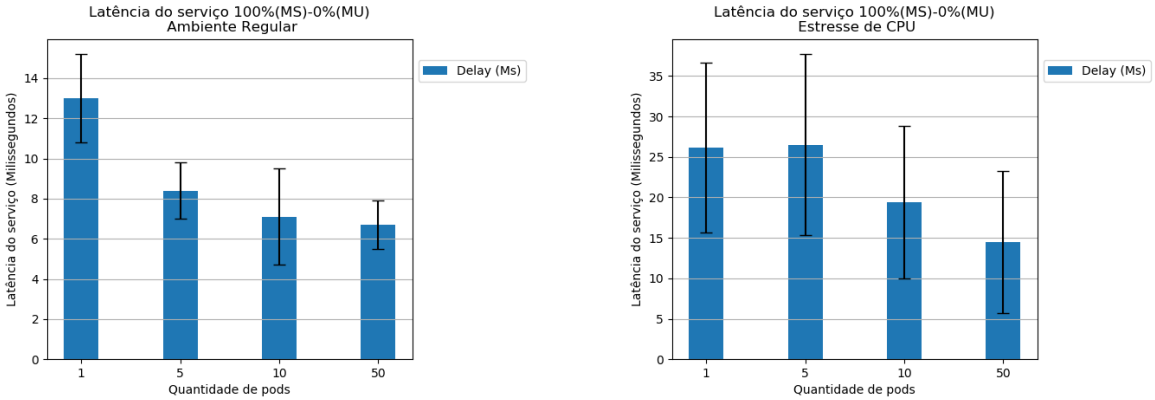
4.1.3 Latência do Serviço Durante a Atualização

Em questão de latência, o serviço possui um máximo de 13 milissegundos (ms) para resposta num ambiente regular, diminuindo para 7 *ms* em média com o total de 50 *pods*. Tal comportamento faz sentido, tendo em vista que existem mais réplicas do serviço para assumir a carga do sistema. No ambiente com estresse de CPU, pode-se perceber que a latência alcança de 26 *ms* a aproximadamente 15 *ms* com respectivamente 1 e 50 *pods*. Isto pode ser observado na Figura 13.

Figura 13 – Comparação de tempo médio de requisição do serviço na configuração MS 100% e MU 0%

(a) Tempo médio das requisições em estado regular

(b) Tempo médio das requisições com estresse de CPU



Fonte – Do Autor

4.2 CONFIGURAÇÃO 2

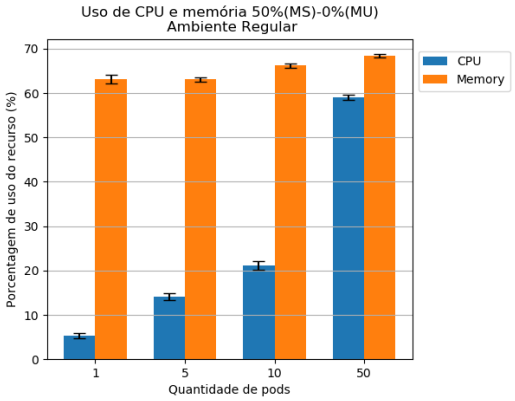
Abaixo estão relatados os resultados da segunda configuração testada, sendo esta a variação que possui a variável MaxSurge (MS) no valor de 50%, e a variável MaxUnavailable (MU) no valor de 0%.

4.2.1 Utilização de Recursos dos Nós do *Cluster*

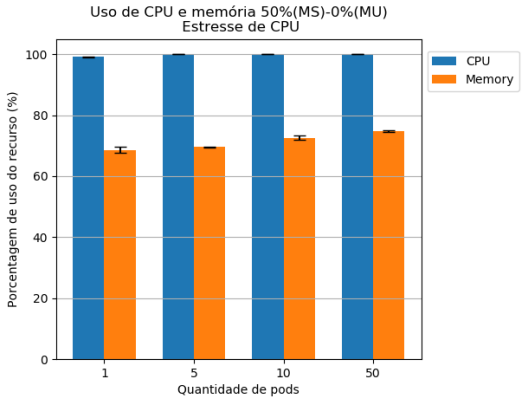
Conforme é possível se observar na Figura 14, o uso do tempo do processador e a quantidade de memória é altamente similar ao observado na Figura 11.

Figura 14 – Comparação de uso de recursos na configuração MS 50% e MU 0%

(a) Uso de recursos em estado regular



(b) Uso de recursos com estresse de CPU



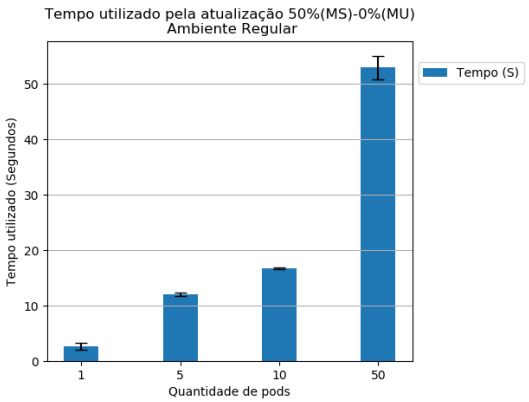
Fonte – Do Autor

4.2.2 Latência de Atualização

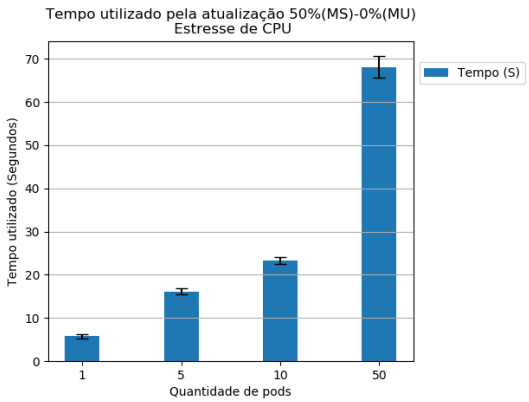
Na Figura 15 é possível observar que o tempo médio da atualização num ambiente regular se mantém similar ao observado na Figura 12, entretanto, quando existem 50 *pods* instanciados, o tempo cresce para aproximadamente 53 segundos em média, que podem ser explicados pela não possibilidade do Kubernetes instanciar todos os *pods* com a nova versão de uma única vez. Pode-se também perceber um considerável aumento de tempo num ambiente com estresse de CPU alcançando 70 segundos para a realização da atualização com 50 *pods*.

Figura 15 – Comparação de uso de tempo na configuração MS 50% e MU 0%

(a) Tempo médio da atualização em estado regular



(b) Tempo médio da atualização com estresse de CPU



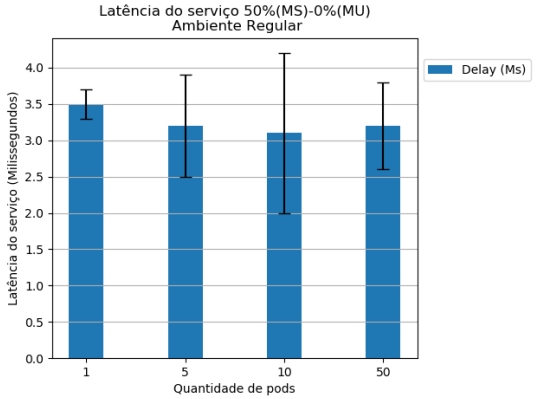
Fonte – Do Autor

4.2.3 Latência do Serviço Durante a Atualização

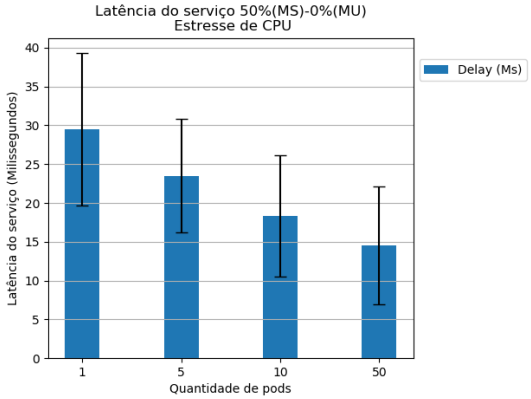
Como demonstrado na Figura 16, o tempo médio de requisições para o serviço num ambiente regular é de 3 *ms*, enquanto este valor cresce para aproximadamente 30 *ms* com apenas 1 *pod*, e decai para aproximadamente 15 *ms* com 50 *pods*. É interessante observar que a variação possível para este valor é consideravelmente mais alta num ambiente estressado, podendo chegar aos 40 *ms* com apenas 1 *pod* instanciado.

Figura 16 – Comparação de tempo médio de requisição do serviço na configuração MS 50% e MU 0%

(a) Tempo médio das requisições em estado regular



(b) Tempo médio das requisições com estresse de CPU



Fonte – Do Autor

4.3 CONFIGURAÇÃO 3

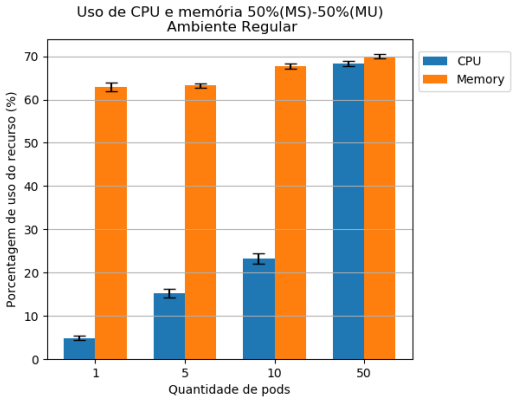
A seguir, expõe-se os resultados da terceira configuração testada, a variável MaxSurge (MS) no valor de 50%, e a variável MaxUnavailable (MU) no valor de 50%, foram aplicadas como variação.

4.3.1 Utilização de Recursos dos Nós do *Cluster*

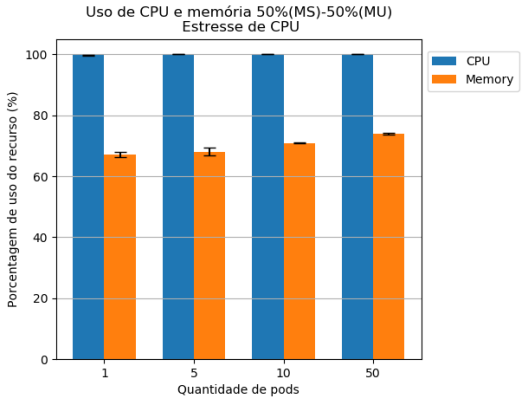
Conforme é possível se observar na Figura 17, o uso do tempo do processador e a quantidade de memória é equivalente aos resultados anteriores, com exceção do tempo de uso do processador quando se tem 50 *pods* no ambiente regular, que apresenta um pico de 70% de uso.

Figura 17 – Comparação de uso de recursos na configuração MS 50% e MU 50%

(a) Uso de recursos em estado regular



(b) Uso de recursos com estresse de CPU



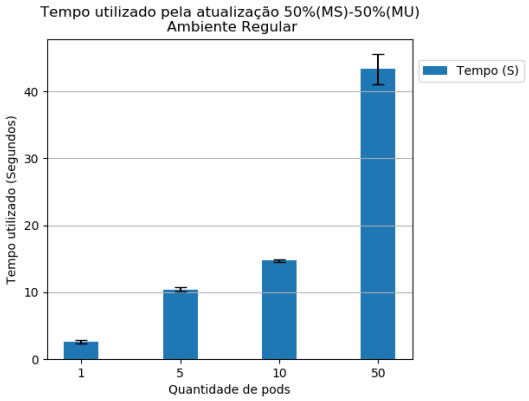
Fonte – Do Autor

4.3.2 Latência de Atualização

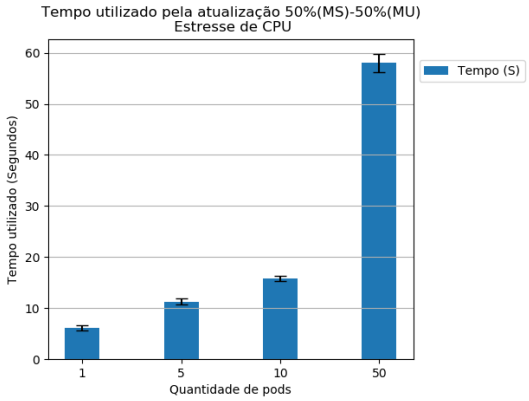
Na Figura 18 percebe-se que o tempo total da atualização se mantém similar às outras configurações, tendo sua maior divergência quando se alcança 50 *pods*. Nesta situação, alcança-se por volta de 40 segundos num ambiente regular e quase 1 minuto num ambiente com estresse de CPU.

Figura 18 – Comparação de uso de tempo na configuração MS 50% e MU 50%

(a) Tempo médio da atualização em estado regular



(b) Tempo médio da atualização com estresse de CPU



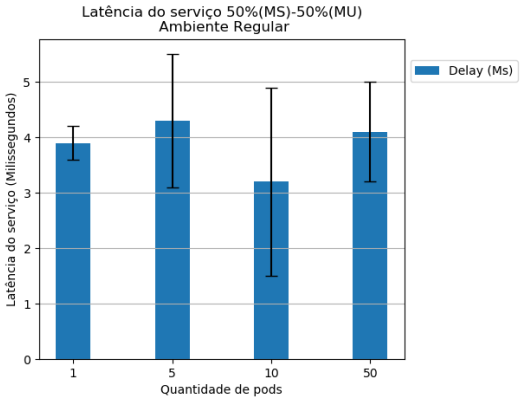
Fonte – Do Autor

4.3.3 Latência do Serviço Durante a Atualização

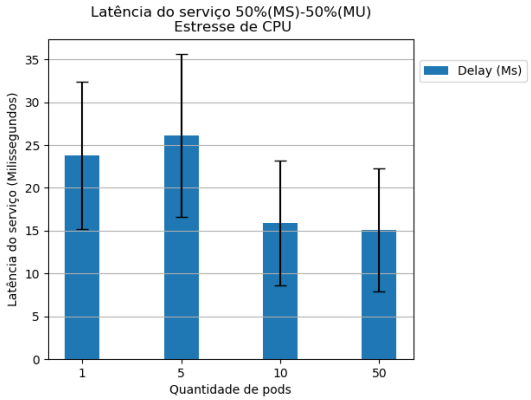
Como explicitado na Figura 19, o tempo total das requisições num ambiente regular não passa de 5 *ms*, enquanto se possui 26 *ms* num ambiente estressado, decaindo para 15 *ms* na marca de 50 *pods*, comportamento observado nas configurações anteriores.

Figura 19 – Comparação de tempo médio de requisição do serviço na configuração MS 50% e MU 50%

(a) Tempo médio das requisições em estado regular



(b) Tempo médio das requisições com estresse de CPU



Fonte – Do Autor

4.4 CONFIGURAÇÃO 4

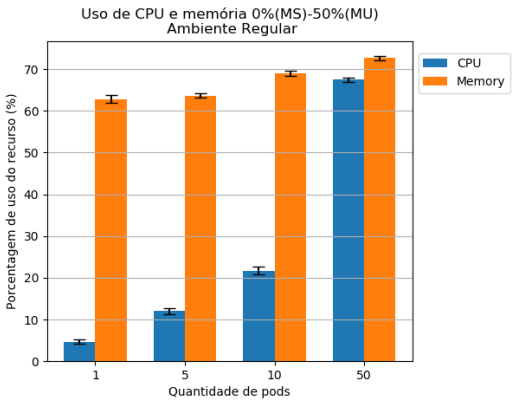
Dispõe-se a seguir, os resultados da quarta configuração testada, sendo esta a variação que possui a variável MaxSurge (MS) no valor de 0%, e a variável MaxUnavailable (MU) no valor de 50%.

4.4.1 Utilização de Recursos dos Nós do *Cluster*

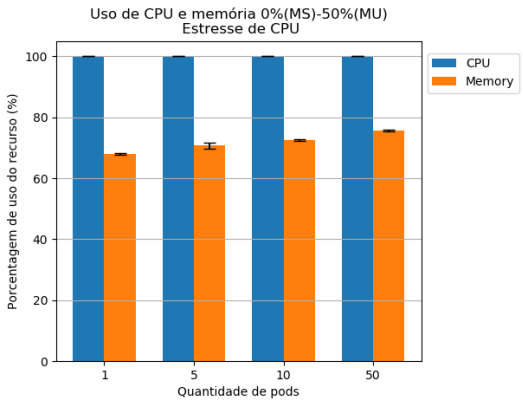
Conforme se vê na Figura 20, o uso dos recursos das máquinas anfitriãs são similares aos resultados previamente apresentados.

Figura 20 – Comparação de uso de recursos na configuração MS 0% e MU 50%

(a) Uso de recursos em estado regular



(b) Uso de recursos com estresse de CPU



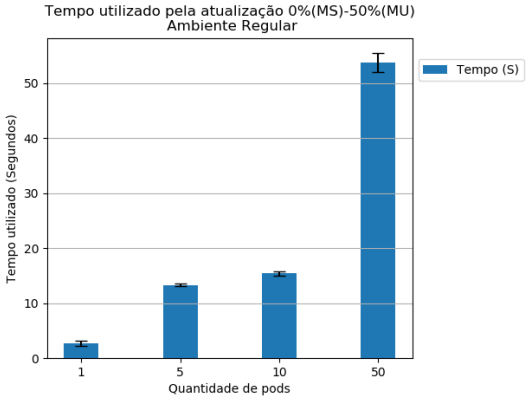
Fonte – Do Autor

4.4.2 Latência de Atualização

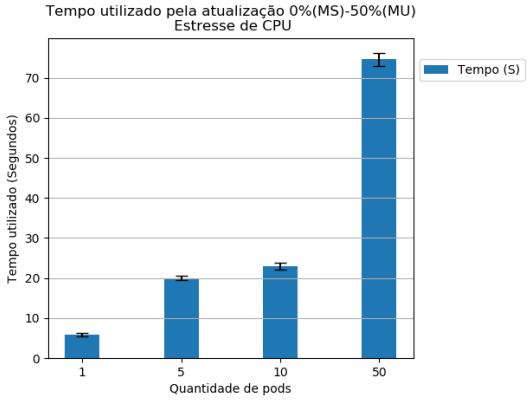
Como apresentado na Figura 21 pode-se perceber que sua principal divergência dos resultados anteriores é no ambiente estressado, em que se tem um tempo de quase 75 segundos. O ambiente regular se mantém similar aos resultados já apresentados.

Figura 21 – Comparação de uso de tempo na configuração MS 0% e MU 50%

(a) Tempo médio da atualização em estado regular



(b) Tempo médio da atualização com estresse de CPU



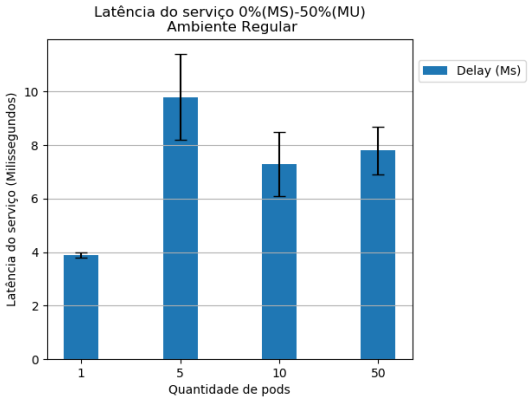
Fonte – Do Autor

4.4.3 Latência do Serviço Durante a Atualização

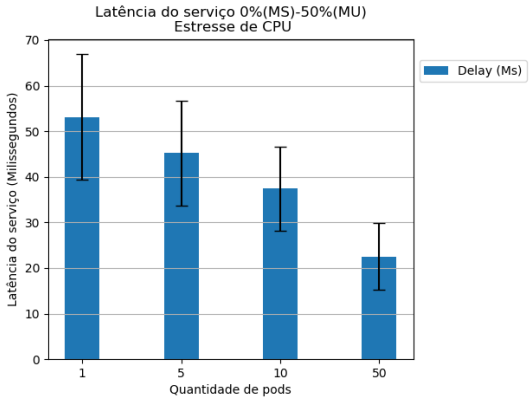
Como visto na Figura 22, o tempo total das requisições num ambiente estressado chega a um máximo de 53 *ms*, chegando até por volta de 22 *ms* na marca de 50 *pods*. É interessante verificar que nesta situação, o ambiente regular possui uma média de tempo de resposta maior em comparação às configurações anteriores, de até 10 *ms* com 5 *pods* instanciados. Isto se dá por conta da quantidade menor de *pods* em execução que o estado desejado, podendo possuir até um mínimo de 3 *pods* executando.

Figura 22 – Comparação de tempo médio de requisição do serviço na configuração MS 0% e MU 50%

(a) Tempo médio das requisições em estado regular



(b) Tempo médio das requisições com estresse de CPU



Fonte – Do Autor

4.5 CONFIGURAÇÃO 5

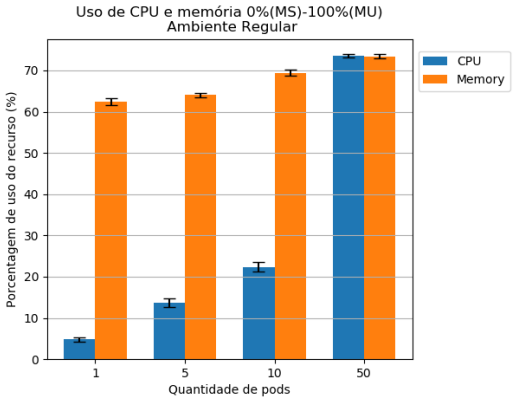
Conforme exposto subsequentemente, têm-se os resultados da quinta configuração testada, variação que possui a variável MaxSurge (MS) no valor de 0%, e a variável MaxUnavailable (MU) no valor de 100%.

4.5.1 Utilização de Recursos dos Nós do *Cluster*

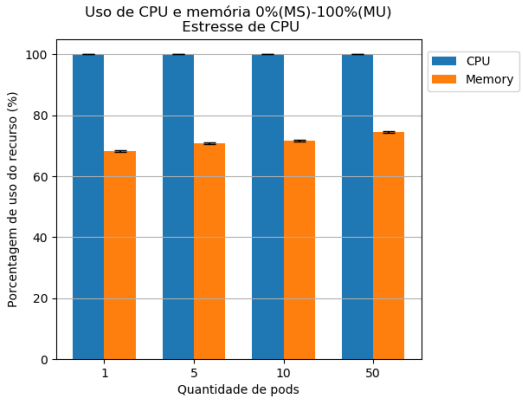
Conforme se vê na Figura 23, o uso dos recursos das máquinas anfitriãs é similar ao que já foi apresentado, com um pico de uso de 70% do processador num ambiente regular, com 50 *pods*.

Figura 23 – Comparação de uso de recursos na configuração MS 0% e MU 100%

(a) Uso de recursos em estado regular



(b) Uso de recursos com estresse de CPU



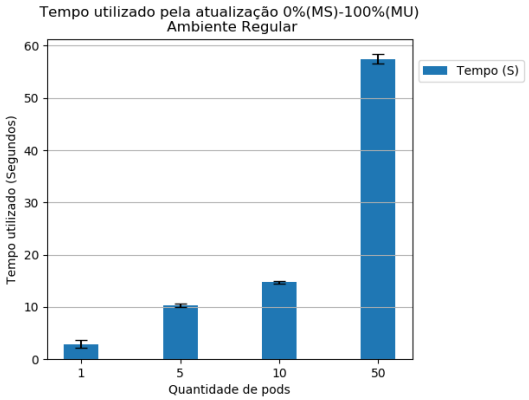
Fonte – Do Autor

4.5.2 Latência de Atualização

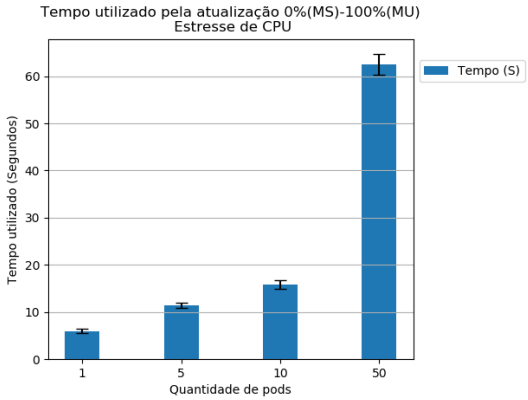
Neste caso em específico, pode-se perceber que não existe muita divergência do tempo utilizado pela atualização em ambas configurações. Ainda assim, como seria de se esperar, a atualização que está com sobrecarga de uso do processador leva mais tempo que a atualização em um ambiente regular. Isto pode ser verificado na Figura 24.

Figura 24 – Comparação de uso de tempo na configuração MS 0% e MU 100%

(a) Tempo médio da atualização em estado regular



(b) Tempo médio da atualização com estresse de CPU



Fonte – Do Autor

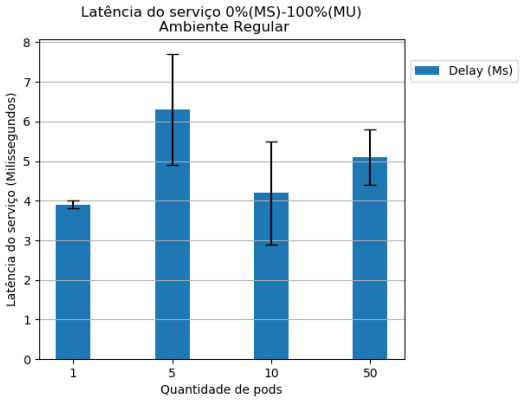
4.5.3 Latência do Serviço Durante a Atualização

Como visto na Figura 25, o tempo total de requisição num ambiente regular é de no máximo 6 *ms*, enquanto se alcança 51 *ms* num ambiente estressado. Pode-se perceber que num

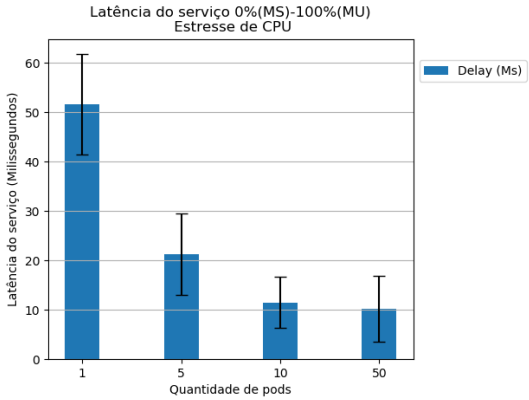
ambiente com apenas 1 *pod* é onde se obtêm este pico. Isto se dá porque o único *pod* que estava provendo o serviço foi terminado e se está esperando a instanciação de um novo para assumir seu lugar. Este pico em tempo de requisição se reduz drasticamente nos ambientes seguintes, descendo para 10 *ms* num ambiente com 50 *pods*.

Figura 25 – Comparação de tempo médio de requisição do serviço na configuração MS 0% e MU 100%

(a) Tempo médio das requisições em estado regular



(b) Tempo médio das requisições com estresse de CPU



Fonte – Do Autor

4.6 ESTRESSE DE MEMÓRIA

Durante a realização deste trabalho, outro ambiente para testes foi utilizado, em que as máquinas anfitriãs estavam sob uso extremo da memória disponível. Entretanto, foi observado que enquanto não era ocupada determinada quantidade de memória, o sistema e os serviços instanciados se comportavam de maneira idêntica ao ambiente em estado regular. Ao se adicionar mais quantidade de memória a ser consumida pelos processos que geravam o estresse do sistema, o Kubernetes não instanciava os *pods* do serviço a ser testado. Isto se dá porque o Kubernetes só instancia um serviço quando possui memória suficiente para executá-lo sem nenhum problema, e enquanto as máquinas anfitriãs não liberarem o recurso, o Kubernetes não tentará fazer o serviço alcançar o estado desejado.

Tal comportamento foi observado em situações com um, cinco e dez *pods*. O sistema, durante a atualização, se comportava regularmente com um e cinco *pods*, mas não foi possível instanciar um serviço com dez *pods*. Ao aumentar mais um pouco o uso de memória, apenas um *pod* pôde ser instanciado, o que não permitia que a coleta de dados pudesse ser replicada nesse caso como foi no caso do estresse de tempo de uso do processador. Além disso, ao chegar a determinada quantidade de uso de memória, as máquinas anfitriãs entravam em suspensão total, e só retornavam ao se encerrar os *pods* que estavam ocupando memória de forma exacerbada.

4.7 DISCUSSÃO

Após a coleta e apresentação dos dados, é necessário realizar uma análise aprofundada do que foi apresentado, de forma a retirar conclusões mais pontuais sobre o assunto. Esta seção está dividida entre as métricas coletadas, sintetizando os resultados de forma a possuir uma visão completa dos dados.

4.7.1 Utilização de Recursos

Como apresentado nos dados, o uso de recursos se mantém estável em todas as configurações testadas, com leves picos de utilização de CPU quando se possui 50 *pods* ativos. O uso de memória se manteve o mesmo em todos os cenários, e mesmo em um ambiente estressado oscilou entre 60% e 70% de utilização da capacidade total do *cluster*.

É interessante ressaltar, no entanto, que o uso de processamento cresce conforme cresce a quantidade de *pods* no sistema. Este comportamento faz sentido, dado que existem mais processos a serem executados no *cluster*.

4.7.2 Latência de Atualização

Esta métrica possuiu grande alteração em todos os cenários testados. Pode-se perceber que apenas na última configuração apresentada a diferença de tempo foi menor que os outros cenários. Da mesma forma, ainda assim é possível verificar que existe uma diferença de aproximadamente 5 segundos entre um ambiente em estado regular e um ambiente estressado.

Os resultados mais divergentes são observados nas configurações 2, 3 e 4, que em um ambiente com 50 *pods* instanciados se possui uma diferença de até 20 segundos para a finalização da atualização. Destaca-se, entretanto, que o tempo final coletado nos testes é o tempo da instanciação do último *pod* com a nova versão. Isto pode significar que *pods* antigos ainda podem estar instanciados, e em fase de finalização, logo, a diferença dos tempos pode oscilar para ambos os sentidos.

De qualquer forma, observa-se que existe, sim, uma divergência considerável dos tempos observados, mesmo em ambientes com menor quantidade de *pods*, variando entre 3 e 6 segundos nos ambientes com 1, 5 e 10 *pods*. Assim, é interessante considerar que o sistema esteja passando por um momento de estabilidade do uso de recursos, no caso, CPU, enquanto passa por uma atualização.

Outro fato importante de ser observado é o aumento de tempo em relação às configurações que possuem valores não nulos no parâmetro *MaxUnavailable* e os que não possuem, principalmente quando se fala na comparação entre as configurações 1 e 5. Pode-se perceber que ao aumentar o valor deste parâmetro, o tempo de atualização cresce, mesmo em ambientes regulares.

4.7.3 Latência do Serviço

Em questão de latência, os resultados são mais concisos e preocupantes em relação à experiência de um possível usuário, ou sistemas que são sensíveis ao tempo de resposta. Tem-se variações de até 52 *ms* entre o resultado em um ambiente regular e um estressado, indicando que possíveis transações e requisições para o serviço demorarão mais tempo para o seu atendimento.

Faz-se necessário ressaltar que o ambiente em que estes testes foram realizados não são idênticos ao que se é experienciado no mundo real, pois os usuários sintéticos precisaram estar dentro da mesma rede que os computadores que o *cluster* por conta de limitações da rede utilizada. Assim, é possível que os valores apresentados neste trabalho sejam muito maiores no mundo real, criando um problema em relação à Qualidade de Serviço (Quality-Of-Service, QoS) da aplicação.

Aos casos menos expressivos, ainda se possui uma distinção considerável em questão de tempo de resposta, possuindo-se de 12 *ms* até 40 *ms* de diferença nos resultados. Sendo possível observar que quanto maior a quantidade de *pods* do serviço, menor a latência apresentada, o que é explicado por existirem mais processos capazes de responder às solicitações dos usuários do serviço.

Assim como a métrica anterior, o aumento do valor do parâmetro *MaxUnavailable* impacta significativamente na qualidade do serviço durante a atualização. A diminuição de *pods* em execução enquanto se executa a *Rolling Update* tende a reduzir a qualidade do serviço significativamente, em relação às configurações que permitem a instanciação de *pods* substitutos antes da remoção dos antigos.

5 CONCLUSÃO

A rápida evolução do ciclo de desenvolvimento de software possibilitou o acelerado crescimento de aplicações responsivas, dinâmicas e de fácil alteração. Com o advento dos microsserviços, aplicações que pudessem controlar e gerenciar os módulos dos *softwares* ganharam grande destaque. A partir do progresso da virtualização para a conteinerização de sistemas, os orquestradores de contêineres se tornaram muito populares, sendo o Kubernetes um dos mais utilizados no presente momento.

A necessidade de realizar atualizações em tempo e produção, levando o nome de *Rolling Updates* na ferramenta analisada, se tornou de grande impacto com a quantidade de acesso aos mais diversos sistemas disponíveis atualmente, enquanto a possibilidade ou o desejo de suprimir a necessidade de parar o serviço para realizar atualizações se tornou um grande atrativo para softwares de orquestração de contêineres.

O presente trabalho possibilitou a verificação do impacto que tipos de estresse aplicados no sistema podem gerar para as aplicações instanciadas no *cluster* de trabalho. Os resultados apresentados demonstram que existe uma considerável dessemelhança entre aplicações instanciadas em um ambiente regular e sobrecarregados. Logo, é interessante se considerar a utilização dos recursos do *cluster* visando a necessidade de atualizar determinada aplicação, e o quanto isso pode impactar em outros serviços sendo executados pelo mesmo grupo de máquinas. Destaca-se também a necessidade de utilização de técnicas como o provisionamento reativo de recursos, para que os possíveis problemas apresentados por este trabalho sejam contornados de forma eficiente. Ainda é necessário lembrar que, por não permitir a utilização de *swap*, o sistema fica limitado à memória RAM que possui, não sendo possível a criação de *pods* de forma exaustiva. Assim, tal recurso precisa ser utilizado de forma consciente para que sejam evitadas as complicações já dispostas.

Os trabalhos futuros sugeridos são a adição de mais tipos de estresse para a realização da coleta de dados, aumentar a quantidade de usuários sintéticos para a análise e trazê-los mais próximos à uma situação real, em que existe mais tráfego na rede competindo com o acesso ao serviço. Além disso, sugere-se aumentar a quantidade de máquinas disponíveis no *cluster*, para uma conferência mais próxima da realidade observada em *Data Centers* e a adição de mais métricas relacionadas ao uso e qualidade do serviço instanciado.

REFERÊNCIAS

1 ALMES, Guy; KALIDINDI, Sunil; ZEKAUSKAS, Matthew. **A round-trip delay metric for IPPM**. [S.l.], 1999.

2 ANDERSON, C. Docker [Software engineering]. **IEEE Software**, maio 2015.

3 BAIER, Jonathan. **Getting Started with Kubernetes**. UK: Packt Publishing, 2015.

4 BERNSTEIN, D. Containers and Cloud: From LXC to Docker to Kubernetes. **IEEE Cloud Computing**, set. 2014.

5 BURNS, Brendan et al. Borg, Omega, and Kubernetes. **Commun. ACM**, ACM, New York.

6 CURL. [S.l.]: cURL, 2019. Disponível em: <https://curl.haxx.se/docs/manpage.html>. Acesso em 15 Nov. 2019.

7 DOCKER Overview. San Francisco, CA: DOCKER, 2019. Disponível em: <https://docs.docker.com/engine/docker-overview/>. Acesso em: 12 Jun. 2019.

8 DÜLLMANN, Thomas F. **Performance anomaly detection in microservice architectures under continuous change**. 2017. Diss. (Mestrado).

9 DÜLLMANN, Thomas F.; HOORN, Andre van. Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches. In: PROCEEDINGS of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. L'Aquila, Italy: ACM.

10 DUPENOIS, Max. **Kubernetes: zero-downtime rolling updates**. [S.l.]: Driftrock, out. 2017. Disponível em: <https://www.driftrock.com/engineering-blog/2017/10/6/kubernetes-zero-downtime-rolling-updates>. Acesso em: 17 Jun. 2019.

11 ETCD. NY: ETCD, 2019. Disponível em: <https://github.com/etcd-io/etcd>. Acesso em 12 Jun. 2019.

12 FELTER, W. et al. An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Philadelphia, PA, USA: IEEE Software, mar. 2015.

13 HIGHTOWER, K.; BURNS, B.; BEDA, J. **Kubernetes: Up and Running: Dive Into the Future of Infrastructure**. CA, USA: O'Reilly Media, 2017. Disponível em: <https://books.google.com.br/books?id=K5Q0DwAAQBAJ>.

14 HOOGENDOORN. **How Kubernetes Deployments Work**. Amsterdã: The New Stack, 2017. Disponível em: <https://thenewstack.io/kubernetes-deployments-work/>. Acesso em: 12 Jun. 2019.

15 JOY, A. M. Performance comparison between Linux containers and virtual machines. In: 2015 International Conference on Advances in Computer Engineering and Applications. Ghaziabad, India: IEEE Software, mar. 2015.

16 KHAN, A. Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. **IEEE Cloud Computing**, set. 2017.

17 KUBERNETES. [S.l.]: Linux Foundation, 2019. Disponível em: <kubernetes.io>. Acesso em: 02 Mai. 2019.

18 KUBERNETES Issues. [S.l.]: Github, 2017. Disponível em: <https://github.com/kubernetes/kubernetes/issues/53533>. Acesso em 14 Nov. 2019.

19 MEDEL, Víctor et al. Adaptive application scheduling under interference in kubernetes. In: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC). Shanghai, China: IEEE, 2016.

20 MEDEL, V. et al. Modelling Performance Resource Management in Kubernetes. In: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC). Shanghai, China: IEEE, dez. 2016.

21 MERKEL, Dirk. Docker: lightweight Linux containers for consistent development and deployment. Linux Journal, v. 2014, mar. 2014.

22 MORAIS, Fábio; LOPES, Raquel; BRASILEIRO, Francisco. Provisionamento Automático de Recursos em Nuvem IaaS: eficiência e limitações de abordagens reativas. In: SBC. ANAIS do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos. [S.l.: s.n.], 2017.

23 NEWMAN, Sam. **Building Microservices**. 1st. CA, USA: O'Reilly Media, Inc., 2015.

24 POLLITT. **Comparing kube-proxy modes: iptables or IPVS?** [S.l.]: Project Calico, 2019. Disponível em: <https://www.projectcalico.org/comparing-kube-proxy-modes-iptables-or-ipvs/>.

25 SAITO, Hideto; LEE, Hui-Chuan Chloe; WU, Cheng-Yang. **DevOps with Kubernetes: Accelerating software delivery with container orchestrators**. Birmingham: Packt Publishing Ltd, 2019.

26 SHARMA, Prateek et al. Containers and Virtual Machines at Scale: A Comparative Study. In: PROCEEDINGS of the 17th International Middleware Conference. Trento, Italy: ACM, 2016.

27 STRESS-NG. [S.l.]: Ubuntu, 2019. Disponível em: <https://kernel.ubuntu.com/~cking/stress-ng/>. Acesso em 14 Nov. 2019.

28 SUN, Lin. **Container Orchestration = Harmony for Born in the Cloud Applications**. [S.l.]: Archive of the IBM Cloud Blog, out. 2015. Disponível em: <https://www.ibm.com/blogs/cloud-archive/2015/11/container-orchestration-harmony-for-born-in-the-cloud-applications/>.

29 SYSSTAT. [S.l.]: Github, 2019. Disponível em: <https://github.com/sysstat/sysstat>. Acesso em 14 Nov. 2019.

30 THÖNES, J. Microservices. **IEEE Software**, jan. 2015.

31 UBUNTU. [S.l.]: Ubuntu, 2019. Disponível em: <https://ubuntu.com>. Acesso em 14 Nov. 2019.