Revista Brasileira de Computação Aplicada, Novembro, 2020

DOI: 10.5335/rbca.v12i3.10993
Vol. 12, Nº 3, pp. 70–84
Homepage: seer.upf.br/index.php/rbca/index

ISSN 2176-6649

ARTIGO ORIGINAL

Análise de desempenho de Rolling Updates do Kubernetes em ambientes de estresse

Performance analysis of Kubernetes Rolling Updates under stress environments

Gabriel Vassoler[1] and Marco Aurélio Spohn [1]

[1]Universidade Federal da Fronteira Sul

gabriel.vassoler@estudante.uffs.edu.br; *marco.spohn@uffs.edu.br;

Recebido: 11/05/2020. **Revisado:** 08/08/2020. **Aceito:** 10/10/2020.

Resumo

Este trabalho apresenta, via estudo de caso, uma análise de desempenho do *Rolling Updates* do *Kubernetes* em ambientes com diferentes tipos de estresse aplicados às máquinas do *cluster*. Foram propostos ambientes de testes, cenários possíveis e parâmetros a serem alterados para a coleta de dados. Após a exposição dos dados coletados, realizou-se uma análise dos resultados obtidos e uma discussão sobre o impacto que os tipos de estresse aplicados podem gerar na experiência do usuário no momento de uma atualização. Os resultados indicam que o mecanismo de atualização em sistemas em produção é factível e parametrizável de modo a minimizar os efeitos colaterais no desempenho do sistema.

Palavras-Chave: Kubernetes; Rolling Updates; Análise de Desempenho; Estresse de Sistemas

Abstract

This work presents, via a case study, a performance analysis of Kubernetes' Rolling Updates features with cluster machines under different types of stress. Test environments, possible scenarios, and proper parameters' settings were proposed for data collection. After the presentation of the collected data, it is shown an analysis of the results and a discussion on the impact that the types of stress can have on the user experience at the time of an update. The results show that the update mechanism in production systems is feasible and configurable to minimize side effects on the system's performance.

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

1 Introdução

O ciclo de desenvolvimento de *software* acelerou desde sua introdução nos anos 1990. 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 agilidade de entrega, sendo o conceito de *DevOps* um deles: introduzido em 2009, apresentou uma nova visão do ciclo de desenvolvimento de *software* com o intuito de deixar as tarefas mais rápidas e simples (Saito et al., 2019). 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

(Saito et al. (2019)). A arquitetura de microsserviços, como apontado por Newman (2015), tem como principais vantagens a fácil replicação e manutenção. Os serviços pertinentes à execução do sistema são apenas um módulo, que podem ser replicados conforme a quantidade de acessos, bem como facilmente isolados em caso de problemas. Cada módulo busca ser independente dos demais, permitindo sua reutilização em outros projetos conforme a necessidade. Desta forma, uma aplicação necessita apenas importar os módulos necessários, sem precisar se preocupar em controlar todos os demais (Saito et al., 2019).

Nesse contexto, a tecnologia de virtualização se tornou atrativa para o desenvolvimento de *software*: cada microsserviço poderia residir em uma máquina virtual e, conforme a necessidade, as máquinas serem instanciadas ou encerradas (Saito et al., 2019). Entretanto, com a introdução dos *Linux Containers*, trabalhos como os de Felter et al. (2015) e Joy (2015) demonstraram que os *containers* são mais eficientes que as máquinas virtuais.

A crescente aderência à conteinerização de aplicações gerou a necessidade de serviços que poderiam organizar e monitorar os contêineres em produção. Tais sistemas, comumente denominados orquestradores, têm como principal função manter o sistema num estado desejado, provido pelos desenvolvedores (Sun, 2015). Logo, aliado ao desenvolvimento e entrega contínua de aplicações, tornou-se necessária a implementação de tecnologias que fizessem a atualização das aplicações sem interromper a execução do sistema. Nesse contexto, tem-se as *Rolling Updates* do orquestrador de contêineres *Kubernetes*: o processo de atualização é realizado de forma segura, sustentando transações em andamento e objetivando manter o sistema nas melhores condições de funcionamento em termos de latência até a finalização da atualização (Hightower et al., 2017).

Soluções como o *Kubernetes* se tornaram conhecidos por suportar a execução dos serviços em situações adversas (Baier, 2015). Hightower et al. (2017) dispõe 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. Essa funcionalidade (*i.e.*, *Rolling Updates*) foi projetada para ser confiável e livre de falhas, para garantir que os novos contêineres assumam o lugar dos antigos, sem prejudicar serviços que estavam sendo prestados a clientes, ou perder requisições, durante a atualização.

Os dois principais parâmetros que regem o processo de atualização no *Kubernetes* são *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. 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 de maior demanda. Assim, em casos de uso não regular ou atualização do sistema, há recursos suficientes para a realização das operações sem prejuízo (Morais et al., 2017).

Neste trabalho, abordagens de provisionamento não serão empregadas, visando-se justamente um uso extremo de recursos, de forma a debilitar o sistema e possibilitar uma análise de seus impactos. Destarte, torna-se relevante avaliar 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 módulos que realizam apenas uma operação e trabalham em conjunto a outros microsserviços ou sistemas para fornecer uma aplicação (Newman, 2015). Normalmente, eles podem ser executados, escalados e testados de forma unitária (Thönes (2015)). Os microsserviços são mais facilmente corrigidos em caso de *bugs* ou implementação de funcionalidades em contraste com os sistemas monolíticos, que possuem elementos dispersos por toda a plataforma e exigem, muitas vezes, que os desenvolvedores avaliem o código por muitas horas para identificar e corrigir os problemas (Newman, 2015).

Segundo Newman (2015), cada microsserviço é uma entidade separada, que pode ser instanciada em um serviço do tipo *Platform-as-a-Service* (PaaS), 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 (2015) 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 empregar 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 demanda por recursos da tecnologia da informação, um grande número de empresas utilizam-se de aplicações que são hospedadas em *data centers* (públicos ou privados). Estes, por sua vez, utilizam-se extensivamente de máquinas virtualizadas, as quais são atribuídas a máquinas físicas sob demanda. A virtualização garante uma alocação flexível das aplicações, além de permitir que mais de um serviço execute isoladamente em uma única máquina física simultaneamente. 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 simultaneamente, podendo ser instanciada em momentos de maior demanda pelo serviço, bem como ser desativada quando a demanda diminui (Sharma et al., 2016).

Sharma et al. (2016) 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 (2015), é um *software* que se propõe a isolar diferentes máquinas virtuais dentro de uma mesma máquina física. É esse *software*, em conjunto com suporte exclusivo do *hardware*, que é responsável em executar núcleos isolados 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 da maior demanda por recursos de *hardware* (*e.g.*, memória e processamento). O segundo tipo de virtualização também realiza o uso de hipervisores; contudo, eles são executados sobre um sistema operacional anfitrião, como apontado por Merkel (2014).

2.3 Contêineres

Contêineres, diferentemente dos tipos já apresentados de virtualização, utilizam apenas um núcleo compartilhado por todos os processos executando em regiões isoladas referenciadas como *namespaces* (Merkel, 2014). Desta maneira, duas aplicações isoladas e utilizando o mesmo núcleo não tem conhecimento que estão utilizando recursos compartilhados. Assim, como apontado por Joy (2015) e Sharma et al. (2016), essa abordagem é muito mais leve se comparada com máquinas virtuais, já que não requer a virtualização do *hardware* para cada uma das aplicações isoladas. Cada aplicação apenas necessita de suas dependências específicas, deixando tudo o que é de encargo do sistema operacional como responsabilidade do gerenciador de contêineres. Estes gerenciadores vêm se tornando muito populares, como é o caso do Docker (*Docker Overview*, 2019). Pode-se ter uma visão geral da diferença entre a arquitetura de contêineres e dos dois tipos de virtualização pela Figura 1.

Em questão de desempenho, os contêineres demonstraram-se superiores em diversos aspectos, como apresentado nos estudos de Felter et al. (2015) e Joy (2015). Além disso, a capacidade de escalabilidade e a reduzida utilização de recursos contribuíram para a rápida adoção desse tipo de solução.

2.4 Docker

O Docker (*Docker Overview*, 2019), como apontado por Anderson (2015), é uma tecnologia para operacionalização de contêineres, empregando-se vários objetos para a execução das aplicações de forma isolada. Os contêineres são representados em imagens (*i.e.*, *templates*) que possuem informações pertinentes à criação

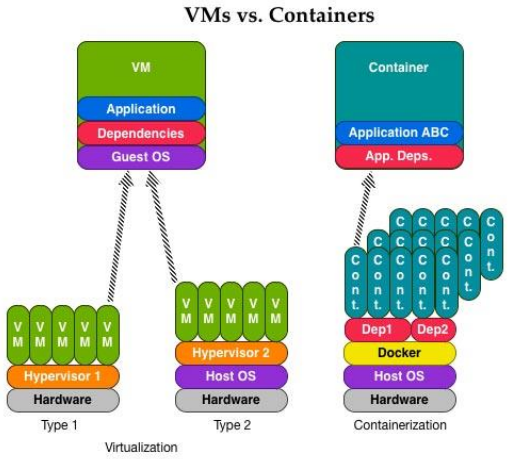

Figura 1: Comparação de arquitetura entre máquinas virtuais e contêineres. Fonte: Merkel (2014).

dos contêineres. Frequentemente, as imagens são derivadas de outras, incorporando apenas algumas modificações e detalhes necessários à aplicação que está sendo executada.

O Docker emprega *namespaces* (*Docker Overview*, 2019) para prover o isolamento dos contêineres. Cada aspecto do contêiner roda em um *namespace* específico: 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 um grau de isolamento entre os processos de diversos contêineres executando em uma mesma máquina. Além disso, o Docker também adota *control groups* (*i.e.*, *cgroups*) que restringem os recursos alocados à aplicação (*e.g.*, pode-se limitar a quantidade de memória principal alocada a um contêiner).

2.5 Orquestração de Contêineres

Quando se trabalha com aplicações que necessitam de múltiplos contêineres, torna-se relevante a existência de um serviço de controle e gerenciamento dos contêineres. 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 (Sun, 2015).

Khan (2017) 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 essencial para que as operações consigam ser executadas de forma correta. O orquestrador deve suportar tarefas de manutenção e ativá-las no *cluster* quando necessário. 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.

- Alta disponibilidade e tolerância a falhas: O orquestrador deve manter um nível de desempenho aceitável, eliminando pontos de falha, criando redundância de recursos e detectando falhas quando elas ocorrerem. Técnicas como balanceamento de carga são apontadas para que os recursos sejam melhor utilizados, bem como para reduzir o tempo de resposta.
- Segurança: É necessário que o orquestrador mantenha o *cluster* seguro contra possíveis ataques, implementando mecanismos de controle de acesso e aceitando apenas imagens assinadas e com registro confiável. Além disso, o orquestrador precisa garantir que os contêineres em execução utilizem apenas os recursos que foram destinados a eles, bem como protegendo de eventuais falhas provenientes do *kernel*.
- Comunicação: A plataforma de orquestração também deve prover uma comunicação eficiente e segura para os contêineres em execução. Deve garantir que os serviços em execução não tenham sua comunicação prejudicada em situação de larga escala, provendo dinamicamente os recursos necessários.
- Descoberta de Serviço: Conforme o serviço executado é escalado, novos contêineres receberão portas e endereços IP próprios. Logo, para se acessar um determinado recurso sendo executado no *cluster*, torna-se necessário um mecanismo que localize as instâncias responsáveis pela execução do serviço requerido e direcione o tráfego de forma apropriada.
- Entrega e integração contínuas: O orquestrador deve fornecer ferramentas que permitam a modificação contínua das aplicações, 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.
- Monitoramento: O orquestrador precisa manter um sistema de monitoramento do *cluster*, mensurando e registrando o desempenho de diversos recursos pertinentes aos contêineres como, por exemplo, a utilização de recursos de rede, CPU e memória. Com base nessas informações, possibilitar ajustes que permitam o funcionamento dos serviços dentro dos padrões requisitados.

2.6 Kubernetes

Originado do Borg (Burns et al., 2016), o Kubernetes é uma plataforma de orquestração de contêineres de código aberto, oferecendo 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 (*Kubernetes*, 2019). Além de prover esses serviços, possui diversas soluções para avaliar o desempenho e monitorar o estado das entidades envolvidas. Inicialmente desenvolvido pela *Google Inc*, atualmente o *Kubernetes* é mantido pela *Cloud Native Computing Foundation*.

Diferenciando-se de outros serviços do tipo PaaS, o *Kubernetes*, apesar de oferecer todas as opções padrão desse tipo de serviço (*i.e.*, balanceamento de carga, escalabilidade e monitoramento), não obriga o usuário a utilizar todas as funcionalidades. Os serviços são ativados e empregados de acordo com as decisões do desenvolvedor (*Kubernetes*, 2019). A seguir, descreve-se os elementos e conceitos intrínsecos ao *Kubernetes*, essenciais para o melhor entendimento dessa avaliação de desempenho.

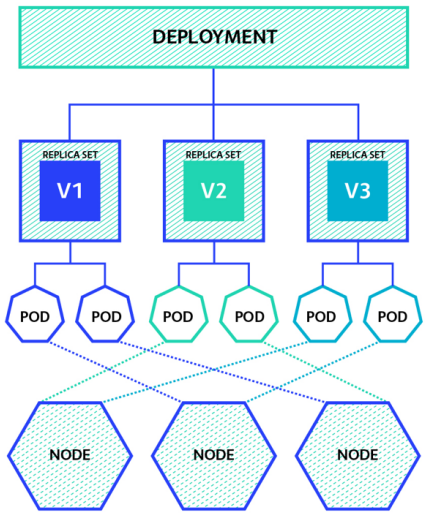

Figura 2: Estrutura dos *pods* do Kubernetes. Fonte: Hoogendoorn (2017).

2.6.1 *Cluster*

O *cluster* de uma, ou mais aplicações, é o conjunto de máquinas que são gerenciadas pelo *Kubernetes*. Cada máquina executa uma instância da aplicação *kubelet*, a qual é responsável pela comunicação com os outros nós, bem como pela realização de ações no próprio nó (*Kubernetes*, 2019).

2.6.2 *Pods*

Um *pod* é a unidade básica do *Kubernetes*, englobando os processos executando no *cluster* (a arquitetura está representada na Figura 2). 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 compartilhados (*Kubernetes*, 2019).

Os *pods* são substituíveis, significando 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. Os *pods* podem abrigar um ou mais contêineres, sendo executados dentro de uma ou mais máquinas (*Kubernetes*, 2019).

2.6.3 Nodes (nós)

Cada nó representa uma máquina física dentro do *cluster*, podendo ser do tipo *Master* ou *Worker (Minion)* (*Kubernetes*, 2019).

Os *Master Nodes* são um ou mais nodos especiais executando componentes que tratam do gerenciamento dos nós. Operações de configuração do *cluster* como, por exemplo, definição do estado desejado da implantação (*deployment*), são realizadas nesses nós. A arquitetura pode ser melhor visualizada na Figura 3.

Os seguintes componentes são executados nos nós *Master* (*Kubernetes*, 2019):

- *kube-apiserver*: Esse é o componente que expõe a API do *Kubernetes*. É através dele que ocorre a comunicação com o *cluster* para a realização de operações de configuração.
- *etcd*: Esse é o componente que armazena as informações cruciais do *cluster*. Nele, encontram-se armazenadas as situações dos contêineres, o estado desejado e eventuais falhas ocorridas durante sua 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 bem apontado em seu repositório (*ETCD*, 2019).
- *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, por exemplo, recursos necessários para determinados contêineres, localidade de dados e perfil de *hardware* desejado.
- *kube-controller-manager*: É o componente que executa os controladores do *cluster*. De fato, existe mais de um controlador; entretanto, todos são compilados em um único binário. Como exemplos de controladores tem-se: o controlador de nós, que verifica e notifica quando um nó deixa de funcionar, e o controlador de replicação, que é o responsável por manter o número de *pods* correspondente ao estado ideal de *deployment* provido para a aplicação.

Os *Minion/Worker Nodes* (nós trabalhadores) hospedam os contêineres responsáveis pela execução das aplicações do usuário. Todos os nós trabalhadores precisam dos processos de gerenciamento do *Kubernetes* e um sistema que possa executar os contêineres. Os agentes presentes nos nós trabalhadores são (*Kubernetes*, 2019):

- *kubelet*: É responsável por manter os contêineres dentro de um *pod*. Ele garante, através de diversos recursos, o bom funcionamento dos contêineres. É importante ressaltar que ele não gerencia contêineres não instanciados pelo *Kubernetes*.
- *kube-proxy*: É o *proxy* de rede, responsável pelo tratamento e encaminhamento de mensagens para todo o *cluster*.
- Gerenciador de Contêineres: Como explicitado anteriormente, os nós precisam de um gerenciador para executar os contêineres. O *Kubernetes* suporta diversos gerenciadores como, por exemplo, o *Docker*.

2.6.4 Deployments

Tendo em mãos o *cluster* e as imagens referentes às aplicações que se deseja instanciar, o passo seguinte consiste em informar ao *Kubernetes* como ele deve proceder. Os *deployments* são a maneira de estabelecer o estado desejado do *cluster*; ou seja, quais e quantos contêineres devem ser instanciados e em quais máquinas (*Kubernetes*, 2019).

Após definido o estado desejado, o *deployment* é aplicado com o *Kubernetes* reservando os recursos necessários, criando os *pods* e designando as imagens às máquinas especificadas. Os *deployments* também podem determinar a remoção de *pods* já existentes, destruindo contêineres instanciados conforme necessário (*Kubernetes*, 2019).

Ao criar o *deployment*, um outro objeto é instanciado: o *ReplicaSet*. Ele é o responsável por manter os *pods* especificados ativos. Esse componente é o agente que, de fato, executa os comandos de criação, eliminação ou alteração dos *pods* (*Kubernetes*, 2019).

2.6.5 Services (Serviços)

Considerando-se que os *pods* são facilmente criados e destruídos com novos endereços IP e portas, faz-se necessária uma abstração com endereço IP e porta imutáveis que possibilite 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 seja lançada. Os *services* encapsulam um ou mais *pods* com IP único, pelo qual agentes externos podem acessar os recursos do componente (*Kubernetes*, 2019). Essa abstração pode ser visualizada na Figura 4.

2.6.6 Rolling Updates

Havendo uma implantação (*deployment*) ativa de um serviço, é possível especificar outro estado desejável do *cluster* para que o *Kubernetes* atualize o serviço sem que ele fique indisponível. Esse recurso, chamado de *Rolling Updates*, substitui gradativamente os *pods* ativos por novas instâncias, garantindo que as conexões ativas não sejam abruptamente encerradas, desativando o respectivo *pod* somente após a conclusão do serviço (*Kubernetes*, 2019).

Quando um *pod* entra em estado de finalização, o *Kubernetes* para de rotear novas conexões para o *pod* e estabelece um tempo limite para que suas transações sejam encerradas. Após o tempo limite, o *pod* é excluído e outro assume sua posição (*Kubernetes*, 2019).

Existem duas variáveis que podem ser alteradas para melhor controle dos recursos durante uma *Rolling Update*: 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* determinados pelo *deployment*. Também é possível definir 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 (*Kubernetes*, 2019).

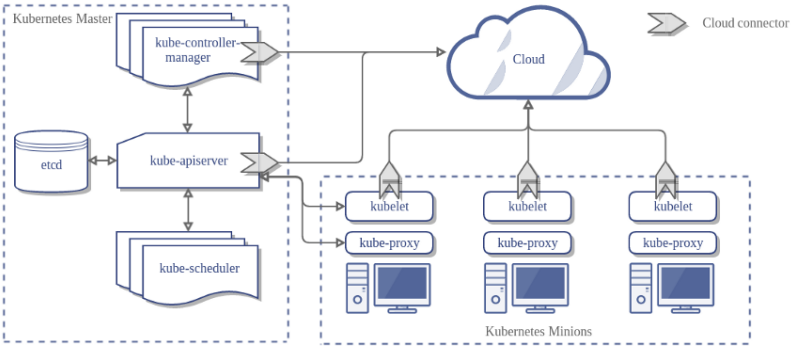

Figura 3: Estrutura dos componentes dos nós do Kubernetes. Fonte: *Kubernetes* (2019).

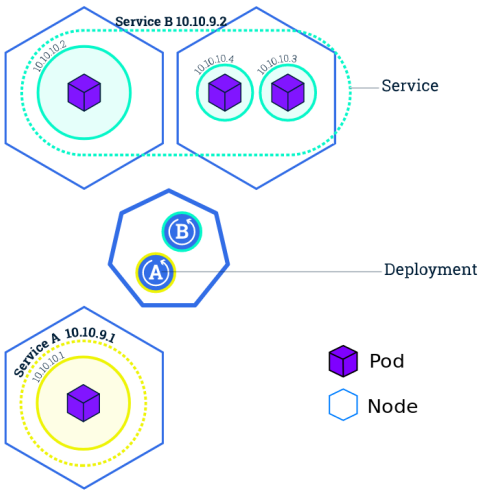

Figura 4: Estrutura dos Services do Kubernetes Fonte: Adaptado de *Kubernetes* (2019)

3 Trabalhos relacionados

Até onde foi possível constatar, este é o primeiro trabalho a realizar uma avaliação de desempenho do *Kubernetes* durante uma atualização em tempo de produção sob um ambiente de estresse. Entretanto, publicações relacionadas foram identificadas e estão dispostas a seguir:

- Os Trabalhos de Medel et al. (2016a,b) apresentam 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 and van Hoorn (2017) 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 (2016) 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 (2017) 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 tráfego de dados, mas gera erro no lado oposto 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.

4 Materiais e Métodos

A metodologia adotada neste trabalho está baseada na criação de uma série de ambientes similares ao de uma aplicação real, objetivando-se realizar testes de desempenho do sistema. Tais testes envolvem a coleta de dados referentes à latência de resposta, utilização de memória, processador e tempo de atualização nas máquinas anfitriãs do *cluster*. Para a execução dos testes, desenvolveu-se uma simples aplicação que retorna uma determinada página *web* ao ser acessada.

Dado o cenário e a aplicação propostos, definiu-se as ferramentas (descritas nas seções a seguir) para atuar no processo de geração do estresse do sistema. O estresse, então, pôde ser gerado através da criação artificial de usuários contemplando requisições ao sistema, estabelecendo-se também a limitação da utilização de recursos pelo *Kubernetes* nas máquinas anfitriãs via *pods* responsáveis pela execução das aplicações.

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

Como foco principal da análise, avaliaram-se dados gerados pelo *cluster* durante uma atualização em um ambiente com pouca carga, em que os recursos não estão sendo severamente utilizados, e um ambiente

Tabela 1: Configurações das máquinas utilizadas

Identificação	Processador	Clock (GHz) – Núcleos	Memória RAM (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

totalmente estressado. Os dados foram comparados para identificar alterações no desempenho do sistema e eventuais reflexos na experiência dos usuários.

Para a realização dos testes, com objetivo de coletar os dados para análise, instanciou-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 na Tabela 1. Cabe ressaltar que todas as máquinas utilizaram a distribuição Ubuntu do sistema operacional Linux, na sua versão 18.04.

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 (*Kubernetes Issues*, 2017).

4.1 Parâmetros de Configuração

Os parâmetros de configuração dispostos e definidos previamente à atualização, em tempo de produção, estão dispostos a seguir:

- **MaxSurge (MS)**: É um parâmetro opcional que define quantos *pods* extras, além do número desejado, estabelecido pelo *deployment*, podem ser criados. Por exemplo, suponha que o *deployment* define como sendo 10 o número alvo de *pods* a ser alcançado. Definindo o *MaxSurge* como 5, o *Kubernetes* pode criar até 5 *pods* além desses 10, resultando em um total máximo de 15 *pods* ativos durante a transição. Desta forma, quando uma determinada quantidade $X > 10$ de *pods* for atingida, os $X - 10$ *pods* antigos serão eliminados para voltar à quantidade desejada, contemplando a versão atualizada. Essa ação será repetida até que todos os *pods* antigos sejam substituídos (*Kubernetes*, 2019).
- **MaxUnavailable (MU)**: De maneira similar ao parâmetro anterior, este é um campo opcional que define quantos *pods* aquém do número desejado, estabelecido pelo *deployment*, podem ser destruídos. Por exemplo, suponha que o *deployment* define como 10 o número desejado de *pods* a ser alcançado. Definindo o *MaxUnavailable* como 5, o *Kubernetes* pode destruír até 5 *pods* abaixo destes 10, resultando em um total mínimo de 5 *pods* ativos durante a transição. Destarte, quando uma quantidade $X < 10$ de *pods* for alcançada, os $10 - X$ *pods* antigos serão eliminados para voltar à quantidade desejada, conforme a versão atualizada. Essa ação será repetida até que todos os *pods* antigos sejam substituídos (*Kubernetes*, 2019).

Tabela 2: Configurações para avaliação

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

Esses dois parâmetros podem assumir valores absolutos ou percentuais. O valor padrão de ambos é 25%. A partir destes parâmetros foram estabelecidas cinco configurações possíveis para as atualizações. Tais configurações podem ser verificadas na Tabela 2.

4.2 Ferramentas Auxiliares

4.2.1 *Stress-ng*

O *Stress-ng* (2019), disponível nos repositórios padrões da distribuição Linux Ubuntu (*Ubuntu*, 2019), é utilizado para gerar demandas com vários aspectos de estresse em um sistema operacional. Os perfis de testes vão desde sobrecarga de memória, CPU e I/O, até monitoramento de temperatura e uso de outros recursos. Os testes são realizados utilizando trabalhadores, que são processos monitorados pelo processo principal, exercendo o tipo de função programada. Por exemplo, num teste de sobrecarga de CPU, o Stress-ng cria processos que irão utilizar uma determinada porcentagem de capacidade de processamento.

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. Empregaram-se os testes 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.

4.2.2 *SAR*

Sendo parte do pacote *SysStat* (2019), o SAR é um aplicativo que pode ser usado para medir a utilização de recursos em qualquer distribuição do sistema operacional Linux. As métricas coletadas vão desde utilização de CPU, memória, rede e uso de disco, até de consumo de energia e atividades em terminais do sistema.

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 aplicativo foi instanciado dentro do próprio sistema operacional anfitrião, de forma que fosse possível ter a utilização total dos recursos da máquina.

4.2.3 *cURL*

O cURL (*cURL*, 2019), ou *Client for URLs*, 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. Essa 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 no cURL 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. Essa ferramenta foi utilizada com este intuito neste trabalho, com a finalidade de obter a latência do serviço para os usuários do *Kubernetes*.

4.3 Métricas

A seguir, estão explicitadas as métricas utilizadas para a análise de desempenho das *Rolling Updates* do *Kubernetes*. As métricas possibilitam mapear as divergências de desempenho entre o sistema executando em uma situação regular e outra sobrecarregada.

4.3.1 *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 pode auxiliar na identificação de congestionamento da rede e, no contexto dessa análise, pode indicar o tempo extra na resposta do sistema durante as atualizações, o que remete diretamente à Qualidade de Serviço (QoS) (Almes et al., 1999).

4.3.2 *Latência de Atualização*

A latência ou tempo total da atualização considera o tempo desde o início da atualização (*i.e.*, o momento em que o pedido de alteração foi enviado ao *cluster*) até o momento em que o último *pod* atualizado for iniciado. Esta métrica é relevante para avaliar a celeridade com que o *cluster* consegue realizar as alterações requisitadas a fim de atingir o novo estado desejado.

4.3.3 *Utilização de Recursos nas Máquinas Anfitriãs*

A medição da utilização de recursos nas máquinas do *cluster* é relevante para definir a quantidade necessária dos recursos em uma situação regular de atualização, identificando-se quais recursos são sobrecarregados em função da utilização exacerbada de alguns recursos específicos. Além disso, oferece subsídios para avaliar o impacto dos respectivos recursos na execução em produção das operações no *cluster*.

5 Resultados

Todas as métricas analisadas foram submetidas a um cálculo de grau de confiança, estabelecido em 95%, de forma a avaliar a confiabilidade dos resultados. Um conjunto de usuários sintéticos é instanciado, gerando-se uma demanda agregada de 3 requisições por segundo ao serviço prestado. Para cada configuração, foram realizadas 32 execuções, de forma que o maior e menor valor de cada métrica foram descartados. Assim, para cada configuração de serviço (*i.e.*, quantidade de *pods*), a média e respectivo intervalo de confiança correspondem a 30 execuções/amostras.

Cada execução é realizada até a finalização das atualizações dos *pods*, focando-se nas métricas dentro do período total da atualização. Um limiar superior (*i.e.*, *threshold*) de 5 minutos é estabelecido para interrupção de cada execução, assumindo-se a não finalização do teste devido a fatores externos (*e.g.*, falha no *cluster*). Neste caso, quaisquer dados obtidos durante a execução parcial são removidos e o teste descartado.

5.1 Configuração 1

Conforme supra exposto, na primeira configuração avaliada, define-se o parâmetro MaxSurge (MS) com o valor de 100% e o parâmetro MaxUnavailable (MU) com o valor de 0%.

5.1.1 *Utilização de Recursos dos Nós do* Cluster

Conforme é possível se observar na Figura 5, a utilização do processador nas 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, mantendo-se em torno de 70% em um ambiente com processador estressado.

5.1.2 *Latência de Atualização*

Na Figura 6 é possível observar que, utilizando apenas um *pod*, o tempo da atualização nesta configuração em ambiente regular é de aproximadamente 3 s, enquanto que sob estresse de CPU o valor aumenta para aproximadamente 5 s. O tempo permanece similar utilizando-se 5 *pods* em ambas configurações, aumentando para 13 s e aproximadamente 15 s com 10 *pods*, e de 39 s e 53 s com 50 *pods*. Observa-se que, conforme o intervalo de confiança estabelecido, a variação dos resultados altera consideravelmente conforme se aumenta a quantidade de *pods*, sendo mais acentuada nos cenários com processador sob estresse.

5.1.3 *Latência do Serviço Durante a Atualização*

Em questão de latência (Figura 7), o serviço apresentou um valor máximo em torno de 13 ms no ambiente regular com um *pod*, reduzindo para aproximadamente 7 ms em média com um total de 50 *pods*. Esta redução resulta da existência de um número maior de réplicas do serviço, possibilitando um balanceamento de carga. No ambiente com estresse de CPU, pode-se perceber

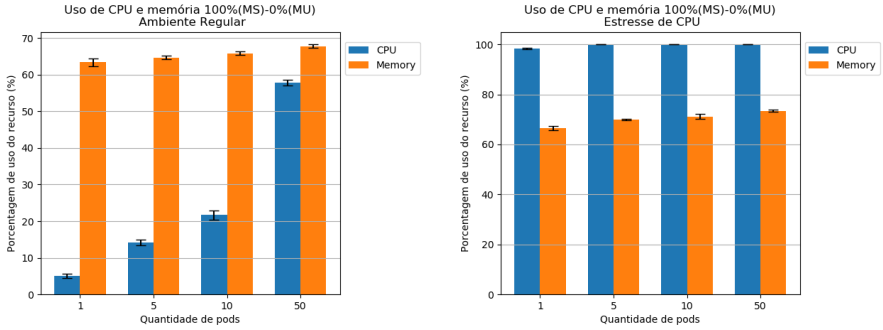

Figura 5: Utilização de recursos na configuração MS 100% e MU 0%

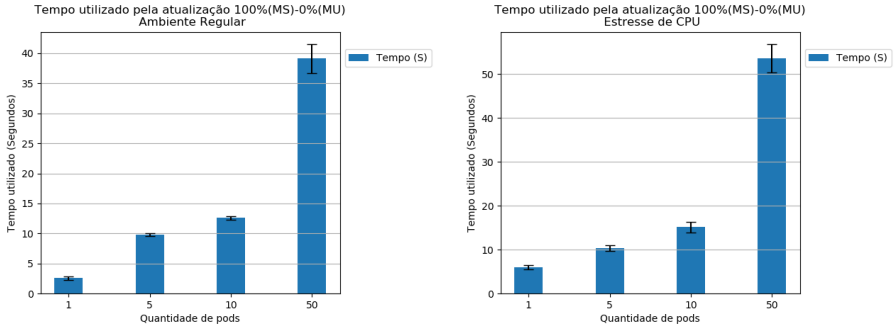

Figura 6: Latência de atualização na configuração MS 100% e MU 0%

que a latência varia em média entre, aproximadamente, 26 ms e 15 ms com, respectivamente, 1 e 50 *pods*.

5.2 Configuração 2

Abaixo estão relatados os resultados da segunda configuração avaliada, tendo o parâmetro MaxSurge (MS) com valor de 50% e o parâmetro MaxUnavailable (MU) com valor de 0%.

5.2.1 *Utilização de Recursos dos Nós do* Cluster

Conforme é possível ser observado na Figura 8, a utilização da CPU e da memória é similar ao cenário anterior (vide Figura 5).

5.2.2 *Latência de Atualização*

Na Figura 9 observa-se que o tempo médio da atualização em um ambiente regular se mantém similar ao observado na configuração anterior (vide Figura 6); entretanto, quando existem 50 *pods* instanciados, o tempo aumenta para aproximadamente 53 segundos em média, o que pode ser um indicativo de que o *Kubernetes* não conseguiu instanciar simultaneamente todos os *pods* com a nova versão. Adicionalmente, percebe-se um considerável aumento na latência em um ambiente com estresse de CPU, alcançando-se 70 segundos para a realização da atualização com 50 *pods*.

5.2.3 *Latência do Serviço Durante a Atualização*

Como ilustrado na Figura 10, o tempo médio de requisições para o serviço num ambiente regular é de 3 *ms*, enquanto no cenário sob estresse a latência cresce para aproximadamente 30 *ms* quando há 1 *pod*, reduzindo para aproximadamente 15 *ms* com 50 *pods*. Observa-se que a latência tem uma variação mais alargada no ambiente sob estresse, podendo alcançar os 40 *ms* com apenas 1 *pod* instanciado.

5.3 Configuração 3

Na terceira configuração avaliada, tem-se o parâmetro MaxSurge (MS) definido com valor igual a 50% e o parâmetro MaxUnavailable (MU) com valor de 50%.

5.3.1 *Utilização de Recursos dos Nós do* Cluster

Conforme os resultados apresentados na Figura 11, a utilização do processador e a quantidade de memória é equivalente aos resultados anteriores, com exceção do cenário com 50 *pods* no ambiente regular, quando a utilização da CPU atinge um pico de 70%.

5.3.2 *Latência de Atualização*

Pelos resultados apresentados na Figura 12, percebe-se que o tempo total da atualização se mantém similar às outras configurações, com exceção do cenário com 50 *pods*. Neste caso, observa-se uma latência média de 40 segundos num ambiente regular e quase 1 minuto em um ambiente com estresse de CPU.

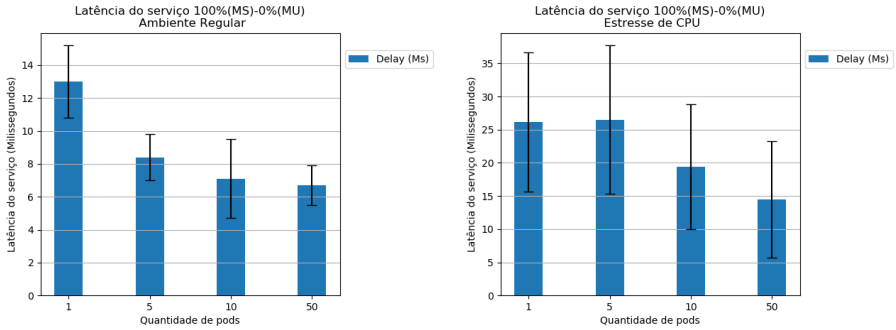

Figura 7: Latência do serviço durante a atualização na configuração MS 100% e MU 0%

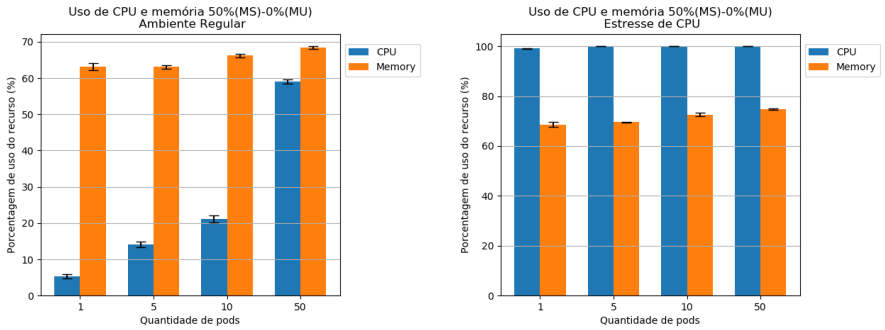

Figura 8: Utilização de recursos na configuração MS 50% e MU 0%

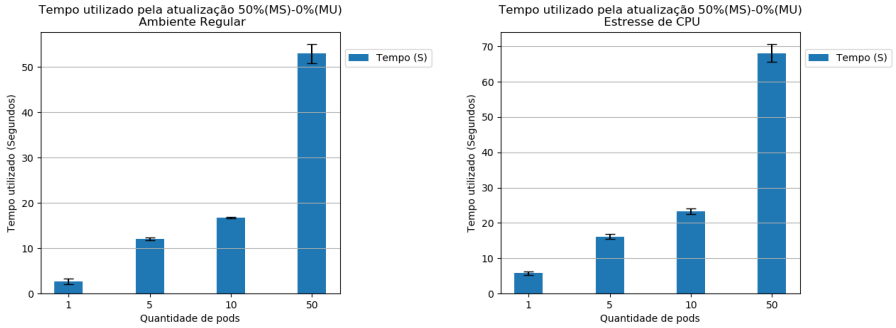

Figura 9: Latência de atualização na configuração MS 50% e MU 0%

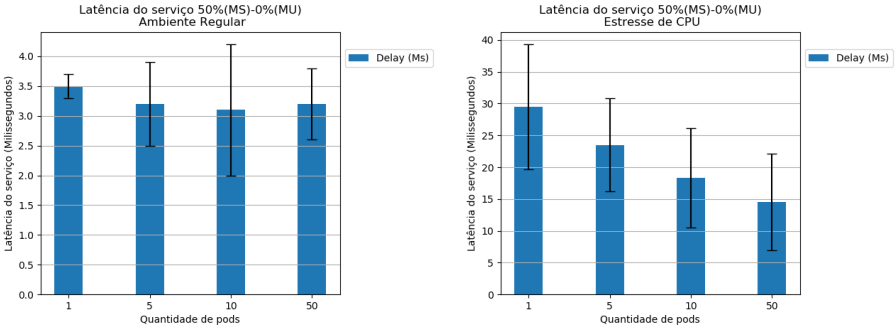

Figura 10: Latência do serviço durante a atualização na configuração MS 50% e MU 0%

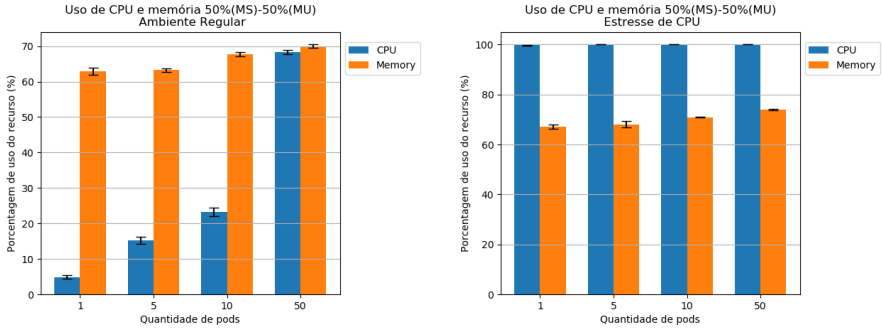

Figura 11: Utilização de recursos na configuração MS 50% e MU 50%

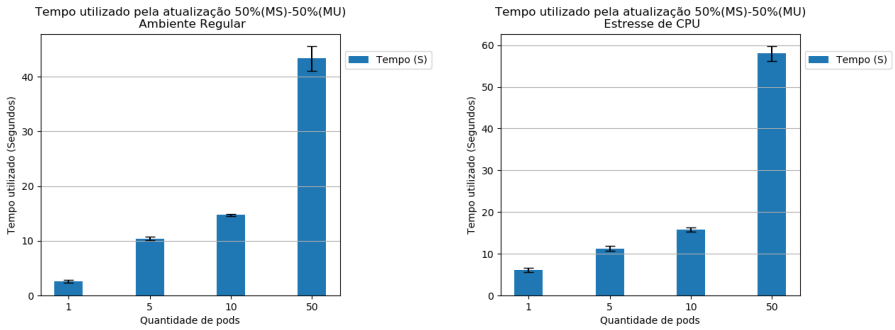

Figura 12: Latência de atualização na configuração MS 50% e MU 50%

5.3.3 *Latência do Serviço Durante a Atualização*

Como explicitado na Figura 13, o tempo total das requisições em um ambiente regular fica, em média, abaixo dos 5 *ms*, enquanto se observa uma latência média de 26 *ms* em um ambiente estressado com apenas um *pod*, reduzindo para um valor médio de 15 *ms* na marca de 50 *pods*, comportamento também observado nas configurações anteriores.

5.4 Configuração 4

Na quarta configuração avaliada, tem-se o parâmetro MaxSurge (MS) com valor de 0%, e o parâmetro MaxUnavailable (MU) com valor de 50%.

5.4.1 *Utilização de Recursos dos Nós do* Cluster

Conforme resultados apresentados na Figura 14, o uso dos recursos das máquinas anfitriãs são similares aos resultados previamente apresentados.

5.4.2 *Latência de Atualização*

Como apresentado na Figura 15, pode-se perceber que a principal divergência dos resultados anteriores está presente no ambiente estressado, observando-se um tempo de quase 75 s. Os resultados referentes ao ambiente regular são similares aos resultados anteriores.

5.4.3 *Latência do Serviço Durante a Atualização*

Conforme apresentado na Figura 16, o tempo total das requisições em um ambiente estressado alcança uma média de 53 *ms*, reduzindo até um valor médio de 22 *ms* na marca de 50 *pods*. Neste caso, verifica-se que o ambiente regular demonstra uma latência de serviço de até 10 ms maior em comparação às configurações anteriores, quando há 5 *pods* instanciados. Esse comportamento resulta da quantidade reduzida de *pods* em execução, podendo-se alcançar um mínimo de 3 *pods* ativos.

5.5 Configuração 5

Conforme exposto subsequentemente, têm-se os resultados da quinta configuração avaliada, tendo-se o parâmetro MaxSurge (MS) com valor de 0% e o parâmetro MaxUnavailable (MU) com valor de 100%.

5.5.1 *Utilização de Recursos dos Nós do* Cluster

Conforme se vê na Figura 17, o uso dos recursos das máquinas anfitriãs é similar ao que já foi apresentado, com um pico de uso de 70% do processador em um ambiente regular com 50 *pods*.

5.5.2 *Latência de Atualização*

Neste caso em específico, pode-se perceber que não existe muita divergência do tempo utilizado pela atualização em ambas configurações. 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 (vide Figura 18).

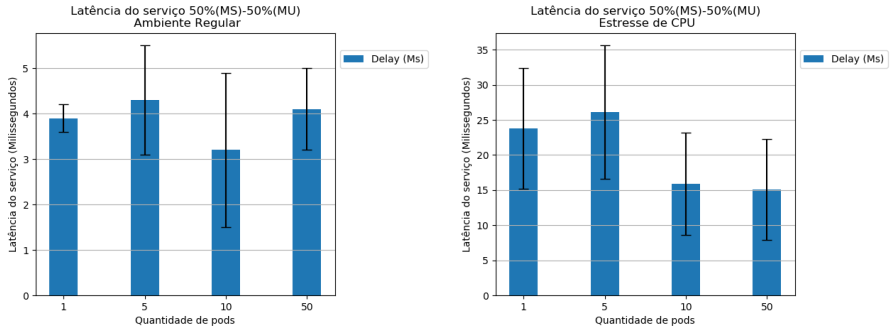

Figura 13: Latência do serviço durante a atualização na configuração MS 50% e MU 50%

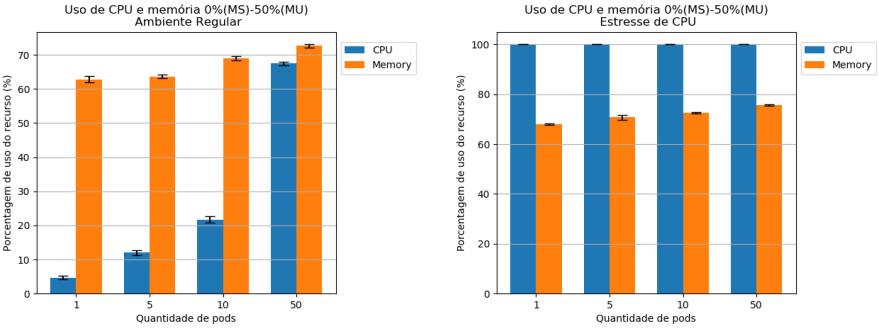

Figura 14: Utilização de recursos na configuração MS 0% e MU 50%

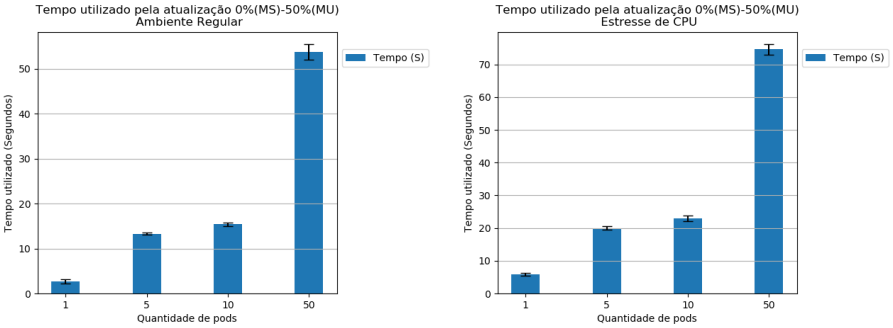

Figura 15: Latência de atualização na configuração MS 0% e MU 50%

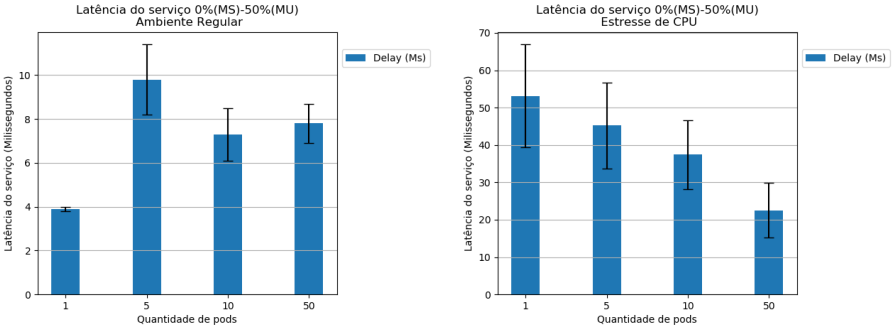

Figura 16: Latência do serviço durante a atualização na configuração MS 0% e MU 50%

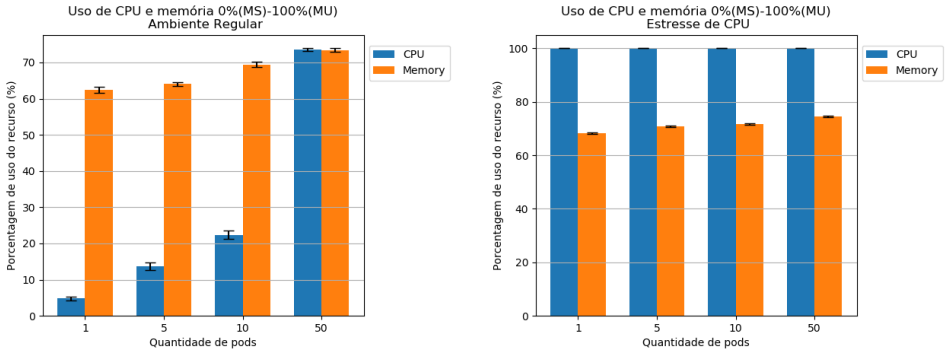

Figura 17: Utilização de recursos na configuração MS 0% e MU 100%

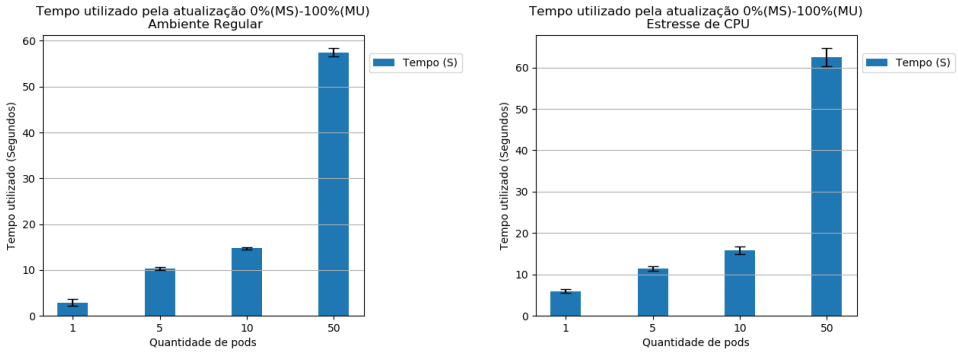

Figura 18: Latência de atualização na configuração MS 0% e MU 100%

5.5.3 *Latência do Serviço Durante a Atualização*

Conforme os resultados apresentados na Figura 19, o tempo total de requisição num ambiente regular é de até 6 *ms* em média, enquanto se observa até 51 *ms* em média em um ambiente estressado com 1 *pod*. Isto ocorre porque o único *pod* que estava provendo o serviço foi finalizado, ficando-se à espera da instanciação de um novo *pod*. A latência de serviço reduz drasticamente nos ambientes seguintes, reduzindo para um valor médio de 10 *ms* em um ambiente com 50 *pods*.

6 Análise

6.1 Utilização de Recursos dos Nós do *Cluster*

A utilização de recursos se mantém estável em todas as configurações avaliadas, com leves picos de utilização de CPU quando empregados 50 *pods*. O uso de memória manteve-se semelhante em todos os cenários e, mesmo em um ambiente estressado, oscilou entre 60% e 70% de utilização da capacidade total do *cluster*.

Ressalta-se, no entanto, que o uso de processamento aumenta conforme cresce o número de *pods* no sistema. Este comportamento é esperado, considerando que existem mais processos do serviço sendo executados no *cluster*.

6.2 Latência de Atualização

Esta métrica apresentou acentuada variação em todos os cenários avaliados. 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, onde em um ambiente com 50 *pods* se observa uma diferença de até 20 segundos para a finalização da atualização. Destaca-se, entretanto, que o tempo final coletado nos testes considera o instante da instanciação do último *pod* com a nova versão (*i.e.*, *pods* antigos ainda podem estar instanciados e em fase de finalização).

Em linhas gerais, observa-se que há uma divergência considerável nos tempos observados, mesmo em ambientes com uma menor quantidade de *pods*, com uma variação entre 3 segundos e 6 segundos nas situações com 1, 5 e 10 *pods*. Assim, é possível considerar que o sistema esteja passando por momentos de instabilidade no uso de CPU enquanto ocorre uma atualização.

Outro fato relevante a ser destacado é o aumento na latência observado nas configurações que possuem valores não nulos no parâmetro *MaxUnavailable*, sobretudo 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

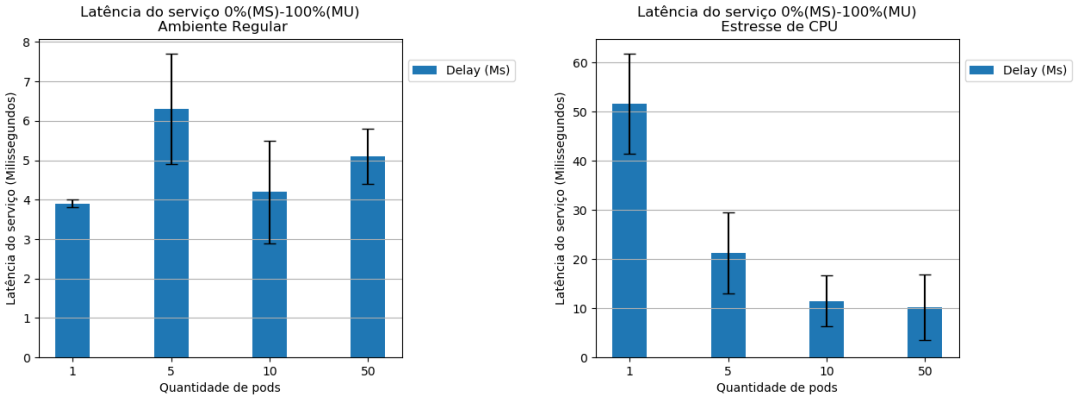

Figura 19: Latência do serviço durante a atualização na configuração MS 0% e MU 100%

ambientes regulares.

6.3 Latência do Serviço Durante a Atualização

Em relação à latência do serviço, os resultados são mais concisos e impactantes em relação à experiência do usuário final, ou sistemas sensíveis ao tempo de resposta. Tem-se variações de até 52 ms entre os resultado de um ambiente regular e um estressado, indicando possíveis atrasos adicionais em transações e requisições.

Faz-se necessário ressaltar que o ambiente em que esses testes foram realizados não é idêntico ao que é usualmente presenciado no mundo real, pois os usuários sintéticos foram instanciados dentro da mesma rede onde estava implantado o *cluster*. Assim, é possível que os atrasos observados nesses testes sejam muito inferiores em uma aplicação real executando na nuvem, impactando ainda mais a Qualidade de Serviço da aplicação.

Aos casos menos expressivos, ainda permanece uma distinção considerável na latência do serviço, variando-se entre 12 ms e 40 ms a diferença nos resultados. Observa-se que, quanto maior a quantidade de *pods* do serviço, menor a latência apresentada, o que é esperado pois há mais processos para atender as solicitações dos usuários do serviço.

Assim como observado para 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* impacta na qualidade do serviço significativamente, resultado observado nas configurações que permitem a instanciação de *pods* substitutos antes da remoção dos antigos.

Conclusão

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

A necessidade de realizar atualizações em tempo de produção, denominado de *Rolling Updates* na ferramenta analisada, tornou-se de grande impacto considerando a crescente demanda aos mais diversos sistemas disponíveis atualmente. O presente trabalho possibilitou uma análise do impacto de situações de estresse aplicados ao sistema sobre as aplicações instanciadas no *cluster* de trabalho. Os resultados obtidos demonstram que existe uma considerável diferença entre o desempenho de aplicações instanciadas em um ambiente regular ou sobrecarregado. Logo, torna-se relevante considerar a utilização dos recursos do *cluster* visando a necessidade de atualização de determinada aplicação em produção, bem como identificar efeitos colaterais em outros serviços 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 resultantes da atualização sejam contornados de forma prática e eficiente. Torna-se necessário destacar que, por não permitir a utilização de *swap*, o sistema fica limitado à memória RAM disponível, não sendo possível a criação de *pods* de forma exaustiva. Nesse contexto, tal recurso precisa ser empregado criteriosamente para que sejam evitadas as complicações já dispostas.

Sugere-se como trabalhos futuros, variar os tipos de estresse e a quantidade de usuários sintéticos, bem como um cenário mais realista de tráfego de rede. Adicionalmente, pode-se variar mais amplamente a quantidade de máquinas disponíveis no *cluster*, contemplando uma infraestrutura mais próxima de um *Data Center*. Considerando o amplo espectro de aplicações candidatas a conteinerização, pode-se ampliar o conjunto de métricas relacionadas a qualidade de serviço.

Referências

Almes, G., Kalidindi, S. and Zekauskas, M. (1999). A round-trip delay metric for ippm, *Technical report*, RFC 2681, september. Disponível em https://tools.ietf.org/html/rfc2681.

Anderson, C. (2015). Docker [software engineering], *IEEE Software* **32**(03): 102–c3. https://doi.ieeecomputersociety.org/10.1109/MS.2015.62.

Baier, J. (2015). *Getting Started with Kubernetes*, Packt Publishing, UK.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J. (2016). Borg, omega, and kubernetes, *Commun. ACM* **59**(5): 50–57. https://doi.org/10.1145/2890784.

cURL (2019). Disponível em https://curl.haxx.se/docs/manpage.html.

Docker Overview (2019). Disponível em https://docs.docker.com/engine/docker-overview/.

Düllmann, T. F. (2016). *Performance anomaly detection in microservice architectures under continuous change*, Master's thesis. https://dx.doi.org/10.18419/opus-9066.

Düllmann, T. F. and van Hoorn, A. (2017). Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, Association for Computing Machinery, New York, NY, USA, p. 171–172. https://doi.org/10.1145/3053600.3053627.

Dupenois, M. (2017). Kubernetes: zero-downtime rolling updates. Disponível em https://www.driftrock.com/engineering-blog/2017/10/6/kubernetes-zero-downtime-rolling-updates.

ETCD (2019). Disponível em https://github.com/etcd-io/etcd.

Felter, W., Ferreira, A., Rajamony, R. and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers, *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802.

Hightower, K., Burns, B. and Beda, J. (2017). *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*, O'Reilly Media, CA, USA. Disponpivel em https://books.google.com.br/books?id=K5QODwAAQBAJ.

Hoogendoorn (2017). How kubernetes deployments work. Disponpivel em https://thenewstack.io/kubernetes-deployments-work/.

Joy, A. M. (2015). Performance comparison between linux containers and virtual machines, *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346. https://doi.org/10.1109/ICACEA.2015.7164727.

Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application, *IEEE Cloud Computing* **4**(5): 42–48. https://doi.org/10.1109/MCC.2017.4250933.

Kubernetes (2019). Disponível em kubernetes.io.

Kubernetes Issues (2017). Disponível em https://github.com/kubernetes/kubernetes/issues/53533.

Medel, V., Rana, O., Bañares, J. A. and Arronategui, U. (2016a). Adaptive application scheduling under interference in kubernetes, *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, IEEE, Shanghai, China, pp. 426–427.

Medel, V., Rana, O., Bañares, J. A. and Arronategui, U. (2016b). Modelling performance resource management in kubernetes, *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, IEEE, Shanghai, China, pp. 257–262.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment, *Linux J.* **2014**(239).

Morais, F., Lopes, R. and Brasileiro, F. (2017). Provisionamento automático de recursos em nuvem iaas: eficiência e limitações de abordagens reativas, *Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBC. Disponível em https://sbrc2017.ufpa.br/downloads/trilha-principal/ST11_01.pdf.

Newman, S. (2015). *Building Microservices*, 1st edn, O'Reilly Media, Inc., CA, USA.

Saito, H., Lee, H.-C. C. and Wu, C.-Y. (2019). *DevOps with Kubernetes: Accelerating software delivery with container orchestrators*, Packt Publishing Ltd, Birmingham.

Sharma, P., Chaufournier, L., Shenoy, P. and Tay, Y. C. (2016). Containers and virtual machines at scale: A comparative study, *Proceedings of the 17th International Middleware Conference*, Middleware '16, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2988336.2988337.

Stress-ng (2019). Disponivel em https://kernel.ubuntu.com/~cking/stress-ng/.

Sun, L. (2015). Container orchestration = harmony for born in the cloud applications. Disponível em https://www.ibm.com/blogs/cloud-archive/2015/11/.

SysStat (2019). Available at https://github.com/sysstat/sysstat.

Thönes, J. (2015). Microservices, *IEEE Software* **32**(1): 116–116. https://doi.org/10.1109/MS.2015.11.

Ubuntu (2019). Disponpivel em https://ubuntu.com.