Balanceamento de Carga Dinâmico em Ambientes Kubernetes com o Kubernetes Scheduling Extension (KSE)

Pedro Moritz de Carvalho Neto[1], **Márcio Castro**[1], **Frank Siqueira**[1]

[1]Universidade Federal de Santa Catarina (UFSC)
Florianópolis/SC

pedro.moritz@posgrad.ufsc.br, {marcio.castro,frank.siqueira}@ufsc.br

Resumo. *A distribuição ineficiente de pods nos nós de um cluster Kubernetes pode levar a um cenário de desbalanceamento do sistema, afetando sua escalabilidade e disponibilidade. Este artigo propõe o Kubernetes Scheduling Extension (KSE), um arcabouço de software que possibilita a implementação de diferentes algoritmos de balanceamento de carga no Kubernetes. Os resultados obtidos com dois algoritmos de balanceamento de carga mostram que o* KSE *é capaz de melhorar o balanceamento de carga entre os nós sem afetar significativamente a disponibilidade do sistema.*

1. Introdução

As tecnologias de virtualização têm sido intensamente utilizadas em nuvens privadas e públicas visando otimizar o uso de recursos computacionais. Em especial, a virtualização baseada em contêineres permite cenários dinâmicos e escaláveis, mais flexíveis do que a abordagem padrão baseada em máquinas virtuais. Esta tecnologia permitiu uma revolução no paradigma de computação em nuvem ao criar uma camada de abstração que encapsula e isola as aplicações em um *cluster* [Buyya et al. 2018].

Para aproveitar a versatilidade dos contêineres em sua totalidade e facilitar seu gerenciamento é necessário utilizar um orquestrador. Dentre os orquestradores de contêineres existentes, o *Kubernetes*[1] é um dos mais utilizados na atualidade [CNCF 2022]. Um *cluster* Kubernetes é constituído por um conjunto de servidores de processamento, denominados *nós*, que executam aplicações em contêineres, os quais são agrupados logicamente em estruturas denominadas *pods*. A *camada de gerenciamento* (também chamada de *plano de controle*) administra os nós de processamento e os *pods* no *cluster* [Luksa 2017].

O escalonador padrão do Kubernetes, o KUBE-SCHEDULER, é responsável pelo escalonamento de novos *pods* nos nós do *cluster*, buscando, entre outros objetivos, o equilíbrio do uso de recursos [Vohra 2017]. No entanto, como a distribuição do uso das aplicações ao longo do tempo é dinâmica e dependente do perfil dos usuários externos, ambientes que inicialmente estavam em equilíbrio de consumo de recursos podem rapidamente alcançar um estado desbalanceado. Nesse sentido, o KUBE-SCHEDULER apresenta um comportamento extremamente limitado, pois não é capaz de realizar a migração dinâmica de *pods* em função do estado de desbalanceamento do sistema. Isso faz com que nós sobreutilizados possam ter seus recursos exauridos, causando o mau funcionamento das aplicações hospedadas por eles, ou que nós subutilizados possam levar a desperdício de recursos financeiros, em especial no caso de nuvens públicas.

[1]https://kubernetes.io/

Aproveitando-se da flexibilidade do Kubernetes e de sua capacidade de expansão, este artigo propõe o *Kubernetes Scheduling Extension* (KSE), um arcabouço de software que atua no reescalonamento dinâmico (balanceamento de carga) dos *pods*. O KSE permite o uso de diferentes métricas para obtenção de informações sobre a carga dos nós do *cluster*, assim como oferece uma interface padrão para a implementação de diferentes algoritmos de balanceamento de carga, realizando, de forma transparente, as migrações de *pods* sempre que necessário. Os resultados obtidos com dois algoritmos clássicos de balanceamento de carga implementados no KSE, quando submetidos a cargas de trabalho desbalanceadas em termos de uso de memória e de processamento, mostraram que o KSE é capaz de melhorar o balanceamento entre os nós do *cluster* sem afetar significativamente a disponibilidade do sistema.

O restante desse artigo está organizado da seguinte maneira. Na Seção 2 são apresentados os fundamentos para contextualização das tecnologias envolvidas e do problema a ser abordado. A Seção 3 apresenta os trabalhos relacionados. Na Seção 4 são apresentados os detalhes do KSE. A Seção 5 detalha o método de avaliação experimental para verificar a eficácia do KSE. Por fim, os resultados experimentais e a conclusão são apresentados nas Seções 6 e 7, respectivamente.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais sobre a virtualização com contêineres (Seção 2.1) e sobre orquestradores, com foco no Kubernetes (Seção 2.2).

2.1. Virtualização Baseada em Contêineres

A virtualização cria uma camada de abstração sobre um *hardware* real, permitindo que os elementos de um único servidor sejam oferecidos como vários servidores virtuais, que consomem e compartilham os recursos físicos, como processador, memória, armazenamento e outros *hardwares* do servidor físico no qual estão hospedadas. Isso permite um provisionamento mais rápido e fácil de ambientes de aplicações sob demanda, fornecendo alta disponibilidade e escalabilidade com custos reduzidos. Um único servidor físico pode hospedar várias Máquinas Virtuais (VMs), sendo que cada uma destas VMs pode possuir um Sistema Operacional (SO) diferente . Embora a virtualização baseada em VMs tenha sido importante para a computação em nuvem, ela apresenta algumas desvantagens: (i) a construção, disponibilização e alteração de VMs é demasiadamente lenta; (ii) VMs podem ocupar muito espaço de armazenamento, o qual pode crescer rapidamente; e (iii) a execução simultânea de várias instâncias de SOs aumenta significativamente o consumo de recursos computacionais, como memória e processamento.

Diferentemente das VMs, a abordagem de virtualização baseada em contêineres virtualiza apenas as camadas de *software* acima do nível do SO. Em ambientes Linux, contêineres utilizam técnicas do próprio *kernel* para isolar caminhos de acesso para diferentes recursos [Jain 2020], aumentando o nível de segurança e isolamento entre aplicações. É comum encontrar sistemas híbridos, com contêineres sendo executados dentro de VMs, que aproveitam as características de ambas as tecnologias: enquanto as VMs podem possuir SOs completos, totalmente diferentes entre si, os contêineres são capazes de prover a flexibilidade e a velocidade que os sistemas de aplicações mais dinâmicos demandam.

2.2. Kubernetes

Um orquestrador de contêineres é um conjunto de ferramentas que permite o provisionamento, implantação, conectividade, dimensionamento, disponibilidade e também o gerenciamento do ciclo de vida dos contêineres [Candel 2022]. O presente artigo se concentra no Kubernetes devido a sua popularidade [CNCF 2022]. O Kubernetes é um *software* de código-fonte aberto, com alto nível de automação e gerenciamento via código, que pode ser estendido através de suas APIs e outros mecanismos. Ele foi inicialmente desenvolvido pela Google, mas teve seu código aberto em 2014 é atualmente desenvolvido pela *Cloud Native Computing Foundation*.

O Kubernetes executa aplicações conteinerizadas em estruturas atômicas chamadas *pods*, executadas nos nós do *cluster*. Ele fornece APIs e outras formas de extensão, o que possibilita a customização de seus componentes por meio de artefatos personalizados e especializados [Burns et al. 2022]. De forma geral, a hierarquia simplificada dos componentes de um ambiente Kubernetes é: aplicações $\rightarrow$ contêineres $\rightarrow$ *pods* $\rightarrow$ nós $\rightarrow$ *cluster*. Cada nó de um *cluster* Kubernetes pode ser uma máquina física ou VM, que é responsável por disponibilizar os recursos necessários para execução dos *pods*. Os nós hospedam os *pods* como componentes da carga de trabalho das aplicações. O plano de controle gerencia os nós e os *pods* no *cluster*. Em ambientes de produção, o plano de controle é geralmente executado em vários computadores e um *cluster* geralmente executa vários nós, fornecendo tolerância a falhas e alta disponibilidade. Os componentes do plano de controle tomam decisões globais sobre o *cluster* (por exemplo, escalonamento), além de detectar e responder a eventos do *cluster*. Os componentes do plano de controle podem ser executados em qualquer máquina do *cluster*.

O escalonador original do Kubernetes faz um trabalho eficiente ao atribuir um novo *pod* ao melhor nó do *cluster* de acordo com um algoritmo interno. No entanto, ele atua **apenas no evento de criação do *pod*** e não durante todo o seu ciclo de vida [Lamouchi 2021]. Portanto, ele não realiza nenhuma ação caso os nós se tornam desbalanceados, ocasionando uma degradação de desempenho e, possivelmente, indisponibilidade do sistema. Atualmente, não há nenhum recurso nativo para prover balanceamento de carga dinâmico dos nós de forma integrada, flexível e inteligente no Kubernetes. Nesse contexto, este artigo propõe um arcabouço de software que permite reescalonar os *pods*, de maneira dinâmica, durante seu ciclo de vida com base em regras definidas por um algoritmo de balanceamento de carga, que pode ser configurado e modificado pelo usuário.

3. Trabalhos Relacionados

Diferentes propostas de escalonadores e algoritmos podem ser encontradas na literatura. De uma forma geral, os escalonadores podem ser classificados em função da estratégia utilizada para a tomada de decisão durante o escalonamento, a qual pode usar modelagem matemática, heurísticas, meta-heurísticas ou aprendizado de máquina [Ahmad et al. 2021]. Além disso, eles podem utilizar diferentes critérios para a tomada de decisão de escalonamento, tais como processamento, memória, armazenamento, uso da rede, entre outros. Em relação à sua flexibilidade, podemos atribuir certas características aos escalonadores: **expansíveis** são aqueles que admitem outras métricas além daquelas coletadas através das APIs do Kubernetes e **customizáveis** são aqueles que permitem o uso de algoritmos definidos pelos desenvolvedores.

O escalonador EDGETIC tem como foco principal a melhoria da eficiência

Tabela 1. Comparação entre trabalhos relacionados.

Proposta	Estratégia	Expansível	Customizável	Dinâmico	Critério
EDGETIC	Meta-heurística	Não	Não	Não	Energia
KCSS	Heurística	Não	Sim	Não	Múltiplo
HELIOTROPIC	Heurística	Sim	Não	Não	Energia
BOREAS	Heurística	Não	Sim	Não	*Makespan*
KSE	Definida pelo usuário	Sim	Sim	Sim	Múltiplo

energética em *data centers* usando um padrão de escalonamento especializado para cargas de trabalho do Kubernetes [Townend et al. 2019]. Esse escalonador personalizado usa métricas relacionadas a recursos de computação e consumo de energia para decidir como será realizada a distribuição dos *pods* nos nós do *cluster*.

O escalonador *Kubernetes Container Scheduling Strategy* (KCSS) otimiza o escalonamento simultâneo de muitos contêineres para melhorar o desempenho em relação às necessidades do usuário e do provedor de nuvem. O KCSS é baseado em um algoritmo de análise de decisão multicritério, que agrega os critérios em uma única classificação, considerando seis critérios principais: (i) taxa de utilização de processador; (ii) taxa de utilização de memória; (iii) taxa de utilização do disco; (iv) consumo de energia; (v) número de contêineres em execução por nó; e (vi) o tempo de transferência da imagem selecionada para um contêiner [Menouer 2020].

O *Heliotropic Scheduler* (HELIOTROPIC) é outra proposta de escalonador, focada no balanceamento de carga (*pods*) entre nós localizados em diferentes regiões geográficas. Seu objetivo é migrar *pods* para nós situados em regiões com menor intensidade de carbono na produção de eletricidade. Sua principal contribuição é um projeto de escalonador que fornece um modelo genérico para otimizar o posicionamento da carga de trabalho em regiões com a menor intensidade de carbono [James and Schien 2019].

Diferentemente das propostas anteriores, o BOREAS é um escalonador projetado para avaliar rajadas de solicitações de criação de contêineres. BOREAS encontra a localização ideal para contêineres utilizando um otimizador de configuração. Os resultados mostram que BOREAS é capaz de associar de maneira adequada um *pod* a um nó em situações onde o escalonador padrão do Kubernetes falha, desperdiçando menos recursos de computação ou provando que nenhuma solução é viável [Lebesbye et al. 2021].

O KSE, proposto neste artigo, não é somente um escalonador, mas um arcabouço de *software* que permite a implementação de algoritmos de reescalonamento dinâmico (i.e., balanceamento de carga) que permite que migrações ocorram de forma dinâmica durante o ciclo de vida dos *pods*. Portanto, a estratégia de migração a ser utilizada não se restringe a heurísticas ou meta-heurísticas, como nos trabalhos anteriores. Ele é facilmente expansível e customizável, trazendo grande flexibilidade para os desenvolvedores. Detalhes sobre a arquitetura e o funcionamento do KSE são apresentados e discutidos na seção seguinte.

4. Kubernetes Scheduling Extension (KSE)

O KSE é uma extensão do Kubernetes que permite aos desenvolvedores implementar estratégias de **reescalonamento dinâmico (balanceamento de carga)** de *pods* nos nós de um *cluster* Kubernetes de forma extremamente flexível e customizada. Ele permite

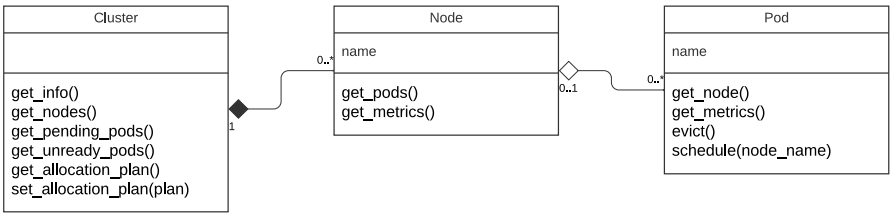


Figura 1. Diagrama de classes simplificado do KSE.

analisar métricas do *cluster* (ou externas a ele) e atuar durante o ciclo de vida dos *pods*, refazendo as associações entre estes *pods* e os nós do *cluster* com o intuito de melhorar o balanceamento da carga. Por padrão, o KSE captura duas métricas básicas dos nós, as quais são coletadas pelo *Kubernetes Metrics Server*[2] e disponibilizadas pela API *metrics*: (i) uso instantâneo de CPU dos nós e dos *pods*; e (ii) memória alocada nos nós e nos *pods*. Para que isto seja possível, o *Kubernetes Metrics Server* precisa ser habilitado no *cluster* Kubernetes onde o KSE será utilizado. O KSE permite, entretanto, ampliar as métricas utilizadas na decisão do rebalanceamento através do consumo de outras fontes, como APIs externas e bancos de dados.

A Figura 1 exibe o diagrama de classes simplificado do KSE. A classe *Cluster* é uma abstração cujos métodos permitem a interação com um *cluster* Kubernetes, definido no arquivo de configuração *kube-config*. A classe *Node*, por sua vez, permite realizar operações e coletar métricas e demais informações de um determinado nó do *cluster*, enquanto a classe *Pod* possibilita a interação com os *pods* disponíveis no ambiente Kubernetes. A Tabela 2 apresenta uma descrição sucinta de cada um dos principais métodos fornecidos pelas classes do KSE.

O algoritmo de balanceamento de carga deve ser implementado no método `get_allocation_plan()`, a qual retornará um **plano de alocação**. Este plano de alocação nada mais é do que uma estrutura de dados que relaciona os *pods* aos nós do *cluster*, indicando, assim, em qual nó cada *pod* deverá estar alocado. Atualmente, o KSE possui dois algoritmos de balanceamento de carga já implementados, os quais foram extraídos de [Zheng et al. 2011]:

KSE-GREEDYLB: este balanceador de carga executa um algoritmo guloso que associa o *pod* com maior carga ao nó com menor carga, sem considerar a distribuição prévia dos *pods*, até que todos os *pods* tenham sido distribuídos. A eventual queda de disponibilidade causada por este algoritmo (devido à grande quantidade de migrações realizadas) pode, eventualmente, ser compensada pela sua eficiência no balanceamento de carga.

KSE-REFINELB: este balanceador de carga observa a distribuição prévia dos *pods* nos nós e promove um ajuste fino no balanceamento. Ele classifica inicialmente os nós em "pesados" ou "leves" em função de suas cargas em relação à carga média dos nós (avg). Um nó é considerado "pesado" se sua carga é maior que $avg * overload$, onde $overload \geq 1.0$ é um fator de sobrecarga definido pelo usuário. Um nó é considerado "leve" se sua carga é menor que avg. Então, executa um algoritmo iterativo que busca, em cada iteração, encontrar um par (p, l), tal que p seja um *pod* de um nó "pesado" (h) com maior carga que, caso seja migrado, mais aproximará

[2]https://github.com/kubernetes-sigs/metrics-server

Tabela 2. Descrição dos métodos fornecidos pelas classes do KSE.

Classe	Método	Descrição
Cluster	`get_info()`	*Retorna informações sobre o cluster Kubernetes.*
	`get_nodes()`	*Retorna a lista de nós do cluster.*
	`get_pending_pods()`	*Retorna a lista de pods sem nó associado.*
	`get_unready_pods()`	*Retorna a lista de pods em estado inativo.*
	`get_allocation_plan()`	*Retorna um plano de alocação.*
	`set_allocation_plan(plan)`	*Envia um plano de alocação (*`plan`*) ao cluster.*
Node	`get_pods()`	*Retorna a lista de pods do nó.*
	`get_metrics()`	*Retorna as métricas do nó.*
Pod	`get_node()`	*Retorna o nó associado ao pod.*
	`get_metrics()`	*Retorna as métricas do pod.*
	`evict()`	*Remove o pod do seu nó atual.*
	`schedule(node_name)`	*Associa o pod ao nó* `node_name`*.*

a carga de um nó "leve" (l) à carga $avg{*}overload$ (sem ultrapassá-la). Em seguida, atribui p ao nó l, atualiza as cargas dos nós h e l, e reclassifica h e l como "pesado" ou "leve" em função das suas cargas atualizadas. O algoritmo termina quando não há mais nenhum nó "pesado" ou quando não há mais *pods* candidatos à migração nos nós "pesados". Sua eficiência pode ser inferior à do algoritmo anterior em alguns casos, porém permite que a disponibilidade seja mantida em um nível mais elevado devido ao menor número de migrações realizadas.

O plano de alocação é enviado a uma abstração (objeto `cluster` do KSE), que se comunicará com o *cluster* Kubernetes através da API e realizará as ações necessárias para obter a configuração definida no plano de alocação. As métricas utilizadas pelo KSE para determinar o plano de alocação podem ser expandidas para além das métricas disponíveis na *Metrics API*, podendo incluir outras fontes de dados como, por exemplo, dados históricos, informações do SO e contadores de desempenho de *hardware*.

O fluxo geral de execução do KSE é apresentado na Figura 2. Este fluxo é executado periodicamente em um intervalo fixo definido pelo usuário de forma paralela com a execução dos *pods*. As seguintes tarefas são realizadas periodicamente pelo KSE: (i) leitura das métricas do Kubernetes; (ii) execução do algoritmo de balanceamento de carga que definirá rebalanceamento dos *pods* nos nós do *cluster* (método `get_allocation_plan()`); e (iii) execução do plano de alocação, que comandará a migração dos *pods*, quando necessário, em função do novo plano de alocação.

5. Método de Avaliação Experimental

Para avaliar o KSE em diferentes cenários, foi criado um ambiente de testes composto por um servidor Linux com processador Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 128 GB de memória RAM rodando Ubuntu 16.04.7 LTS. Este ambiente de testes possui componentes que hospedam aplicações sintéticas, utilizadas nos cenários de teste, e também

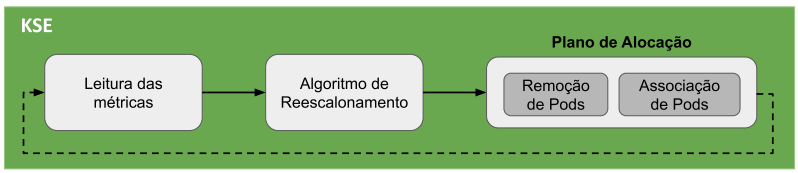


Figura 2. Fluxo de execução periódica do KSE.

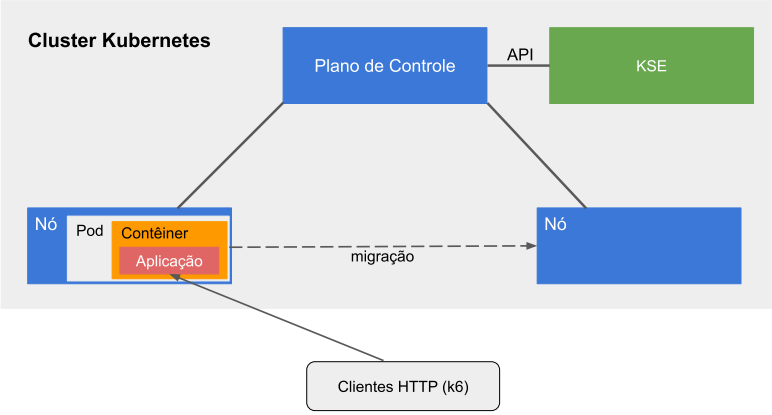


Figura 3. Visão geral do ambiente experimental.

componentes que simulam requisições HTTP, enviadas para simular o tráfego originado por clientes destas aplicações. Os componentes do ambiente de testes são os seguintes:

- Cluster Kubernetes (v1.27.3) implementado com a ferramenta de *cluster* local *Minikube*[3] (v1.31.1);
- Geradores de cargas de trabalho sintéticas para exploração dos recursos de memória[4] e CPU[5] dos nós, a serem executados nos *pods* submetidos ao *cluster*;
- Clientes HTTP, implementados com a ferramenta de teste de carga *K6*[6], responsável por enviar requisições aos geradores de cargas de trabalho sintéticas, criando diferentes padrões de consumo de recursos computacionais;
- Dois balanceadores de carga implementados no KSE (KSE-GREEDYLB e KSE-REFINELB), além da solução padrão do Kubernetes (KUBE-SCHEDULER).

Os componentes do ambiente experimental são apresentados na Figura 3. Os balanceadores de carga KSE-GREEDYLB e KSE-REFINELB, implementados com o KSE, se comunicam com o *cluster* Kubernetes do ambiente de testes através de sua API. Os geradores de cargas de trabalho (encapsulados em *pods*) são distribuídos aleatoriamente nos nós do *cluster* no início da execução de cada cenário de testes. Os clientes HTTP se comunicam com os geradores de carga, simulando o tráfego entre clientes externos e aplicações executadas no *cluster* Kubernetes.

Para a realização dos testes em diferentes cenários, foi elaborado um projeto experimental que incluiu 6 fatores, sendo 5 deles com 2 níveis e 1 com 3 níveis (Tabela 3). Adotou-se um projeto experimental do tipo fatorial completo, onde os cenários de teste considerados são resultado da combinação de todos os fatores e níveis, totalizando 96 cenários de teste diferentes (ou seja, 32 cenários para cada balanceador de carga). Em todos os cenários, utilizou-se um *cluster* Kubernetes composto por 4 nós trabalhadores e um nó mestre. Cada nó trabalhador foi configurado com 2 CPUs, 2 GB de RAM e 5 GB de armazenamento em disco.

Cada experimento foi executado por um tempo fixo de 10 minutos, totalizando 16 horas para uma execução dos 96 cenários. O KSE foi configurado de forma que a coleta

[3]https://minikube.sigs.k8s.io
[4]https://hub.docker.com/r/pmoritz/workloadapp-memory
[5]https://hub.docker.com/r/pmoritz/workloadapp-cpu
[6]https://k6.io/

Tabela 3. Projeto experimental.

Fatores	Níveis
Quantidade de *pods*	20, 40
Requisições por segundo	20, 40
Taxa das requisições	Constante, Crescimento linear
Distribuição das requisições	Exponencial (λ=5/pods), Normal (μ=pods/2, σ=μ/3)
Carga de trabalho sintética	Memória, CPU
Balanceador de carga	KUBE-SCHEDULER, KSE-GREEDYLB, KSE-REFINELB

das cargas dos *pods* e a execução do balanceador de carga ocorra em intervalos de 1 minuto. Portanto, o balanceador de carga atuará, no máximo, 9 vezes durante a execução de cada experimento. As seguintes métricas foram coletadas para comparar os resultados obtidos com os balanceadores de carga implementados no KSE com o KUBE-SCHEDULER: (i) uso de memória (em MBs); (ii) uso do processador (em millicpus[7]); (iii) número de requisições realizadas durante o experimento; e (iv) taxa de sucesso das requisições. O **Erro Médio Absoluto (EMA)** foi utilizado para calcular o grau de desbalanceamento entre os nós (Equação 1). Desta maneira, quanto mais próximo de zero o valor do EMA, melhor será o balanceamento de carga entre os nós do *cluster*.

$$EMA = \sum_{i=1}^{D} |x_i - y_i| \tag{1}$$

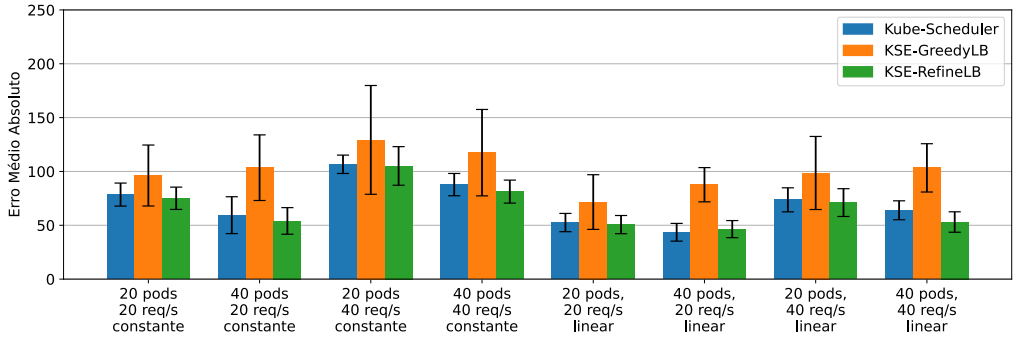
6. Resultados Experimentais

Os resultados apresentados nesta seção foram obtidos a partir da execução de 96 diferentes casos de teste, produzidos pela combinação dos parâmetros descritos na Tabela 3. Os valores mostrados nas Figuras 4 e 5 referem-se aos EMAs calculados a partir de uma média de 10 execuções de cada experimento e as barras de erro referem-se ao desvio padrão obtido. A variabilidade dos experimentos ocorre pela forma de geração das cargas, realizada através de requisições de clientes (K6) com distribuição de probabilidade, produzindo uma carga variável nos *pods*. É possível observar que o gerador de carga de memória é mais afetado pela variabilidade dos experimentos do que o gerador de carga de CPU, pois envolve a alocação/desalocação de dados pelo sistema operacional.

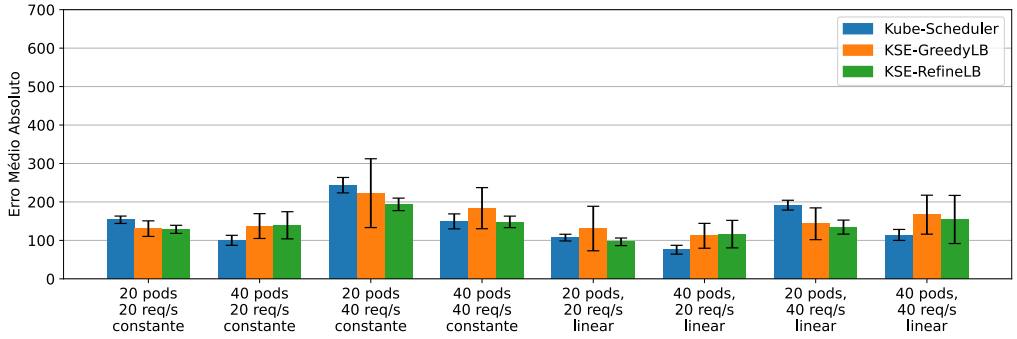
A Figura 4 apresenta os resultados de desbalanceamento (EMA) usando as duas cargas de trabalho (memória e CPU) obtidos quando as requisições enviadas aos *pods* obedecem uma distribuição normal. Cada cenário foi executado com os 3 balanceadores de carga: KUBE-SCHEDULER, KSE-GREEDYLB e KSE-REFINELB. De forma geral, KSE-REFINELB atingiu um grau de desbalanceamento médio muito próximo ao obtido com o escalonador padrão do Kubernetes (KUBE-SCHEDULER). Como a distribuição normal de requisições gera um grau de desbalanceamento pequeno entre os *pods*, apenas 3 dos 16 casos apresentaram um melhor balanceamento de carga com uso do KSE.

A Figura 5 apresenta a mesma análise, porém com requisições enviadas aos *pods* obedecendo uma distribuição exponencial. Diferentemente do experimento anterior, a distribuição exponencial gera um desbalanceamento de carga bem mais elevado entre os

[7]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu



(a) Carga de trabalho: memória

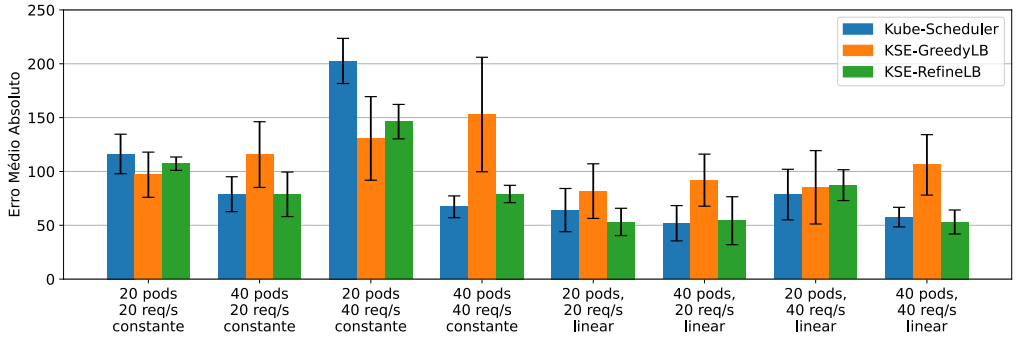


(b) Carga de trabalho: CPU

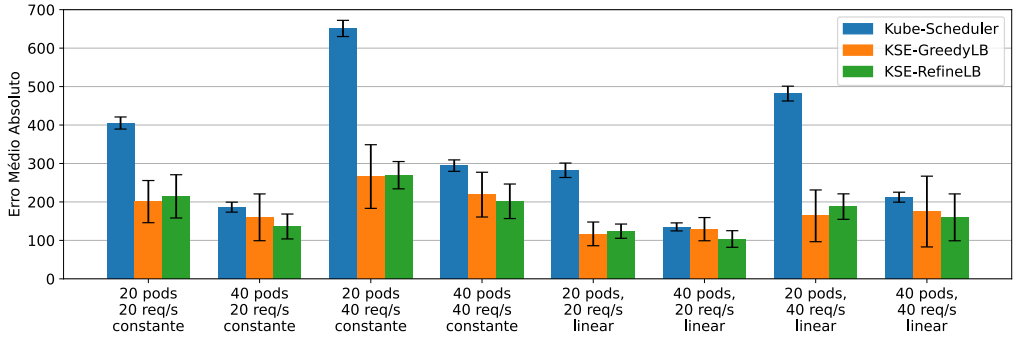
Figura 4. Grau de desbalanceamento: requisições com distribuição normal.

pods, favorecendo, portanto, o uso dos balanceadores de carga implementados no KSE. Neste caso, os resultados demonstraram que ao menos um dos balanceadores do KSE proporcionou um melhor balanceamento de carga em 7 dos 16 cenários de teste. É importante notar, também, que o KSE-REFINELB apresentou uma variabilidade muito menor nos resultados que o KSE-GREEDYLB. Isso se deve ao fato deste balanceador de carga realizar menos migrações, refinando o balanceamento de carga somente quando necessário.

A Figura 6 apresenta a evolução da carga de cada nó do *cluster* ao longo da execução dos experimentos em um cenário com 20 *pods*, 40 requisições por segundo, distribuição exponencial das requisições e carga de trabalho CPU. A Figura 6(a) apresenta o resultado obtido quando a taxa de requisições gerada é constante. Como esperado, a atuação do KUBE-SCHEDULER acontece somente no momento da criação dos *pods*, fazendo com que os *pods* permaneçam nos nós ao longo de toda a execução do experimento. Como os nós do *cluster* não possuem nenhum *pod* no início do experimento e os *pods* são criados praticamente ao mesmo tempo, o KUBE-SCHEDULER acaba por não ter informações suficientes para realizar um bom balanceamento de carga. Apesar de os balanceadores de carga implementados com o KSE partirem do mesmo cenário desbalanceado, eles são capazes de realizar o balanceamento de carga de forma dinâmica ao longo da execução do experimento. Os momentos em que os balanceadores de carga do KSE atuam realizando migrações de *pods* são representados pelas linhas verticais vermelhas nos gráficos. É possível observar que a carga de cada um dos nós do *cluster* converge para um valor médio à medida em que os balanceadores de carga do KSE atuam. Nota-se, também, que o KSE-REFINELB atua menos vezes, pois emprega um algoritmo de balanceamento de carga menos agressivo que o KSE-GREEDYLB.



(a) Carga de trabalho: memória

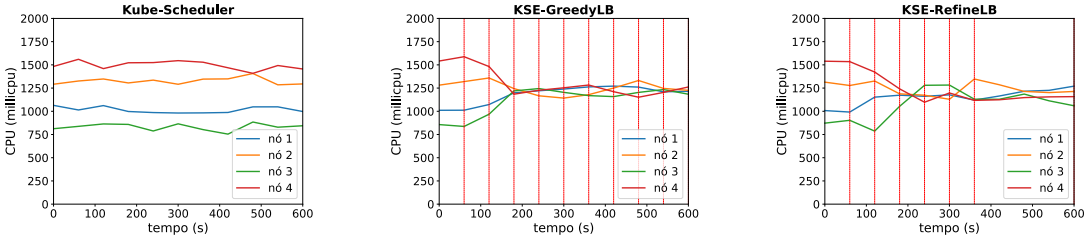


(b) Carga de trabalho: CPU

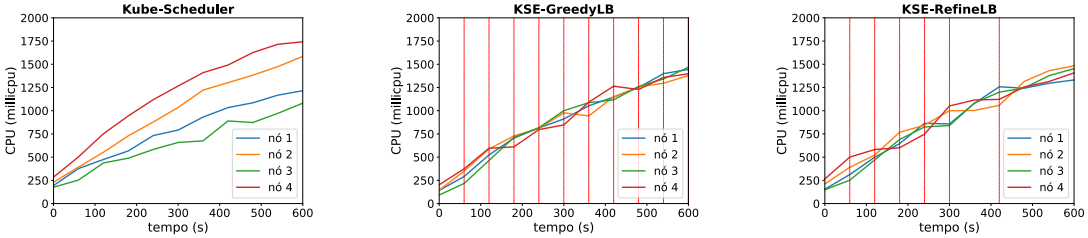
Figura 5. Grau de desbalanceamento: requisições com distribuição exponencial.

A Figura 6(b) apresenta a mesma análise, porém para um cenário com taxa de requisições com crescimento linear. Diferentemente do caso anterior, as cargas dos *pods* iniciam praticamente iguais, partindo de um cenário inicialmente balanceado, mas crescem ao longo da execução em diferentes proporções seguindo a distribuição exponencial. Este experimento mostra que os balanceadores de carga do KSE conseguem manter um bom balanceamento de carga entre os nós do *cluster* mesmo nos casos em que a carga dos *pods* varia ao longo da execução. Assim como no caso anterior, nota-se que o KSE-REFINELB consegue manter a carga dos nós balanceada, porém atuando menos vezes.

Sempre que o KSE realiza migrações de *pods* poderá haver uma certa redução na disponibilidade das aplicações, pois os clientes permanecem gerando requisições aos *pods* durante o processo de migração. Portanto, a disponibilidade é também um fator importante a ser avaliado. A Tabela 4 compara o número médio de migrações realizadas pelos balanceadores de carga do KSE e a disponibilidade média, considerando-se todos os 96 cenários avaliados neste artigo. Na grande maioria dos cenários de teste analisados, observou-se que o balanceador de carga padrão do Kubernetes (KUBE-SCHEDULER) atingiu 100% de disponibilidade. Isso se deve a dois fatores principais: (i) apenas dois cenários de teste executados geraram carga de trabalho de forma a exaurir a capacidade de memória dos nós; e (ii) o KUBE-SCHEDULER não realiza nenhuma migração de *pods*. Com relação ao KSE, os resultados mostram que o balanceador de carga mais agressivo (KSE-GREEDYLB) realiza um número muito elevado de migrações de *pods* (em cada atuação do balanceador diversas migrações são feitas), impactando de forma mais significativa a disponibilidade. Por outro lado, o KSE-REFINELB realizou um número muito pequeno de migrações, afetando muito pouco a disponibilidade.



(a) Taxa de requisições constante



(b) Taxa de requisições com crescimento linear

Figura 6. Carga dos nós ao longo da execução de experimentos com 20 *pods*, 40 requisições/s, distribuição exponencial e carga CPU.

Tabela 4. Número médio de migrações e disponibilidade média (96 cenários).

Balanceador	Distribuição Normal		Distribuição Exponencial	
	Memória	CPU	Memória	CPU
KUBE-SCHEDULER	0.00 / 100.00%	0.00 / 100.00%	0.00 / 99.92%	0.00 / 100.00%
KSE-GREEDYLB	195.80 / 98.37%	191.75 / 98.45%	186.10 / 98.74%	171.95 / 99.01%
KSE-REFINELB	0.48 / 100.00%	8.09 / 99.96%	0.90 / 99.87%	11.60 / 99.94%

7. Conclusão

Este artigo apresentou o KSE, um arcabouço de *software* que permite balancear a carga dos nós de forma dinâmica com auxilio de algoritmos de balanceamento de carga que podem utilizar diferentes métricas. A análise dos resultados permitiu concluir que o KSE possui um potencial para promover um bom balanceamento de carga, obtendo melhores resultados em 10 das 32 comparações realizadas com diferentes parâmetros. Foi possível constatar que a eficácia dos balanceadores de carga do KSE é maior nos cenários onde o desbalanceamento de carga é mais intenso. O KSE apresentou um custo/benefício aceitável em termos de disponibilidade, atingindo 98,37% de disponibilidade mínima nos experimentos com o KSE-GREEDYLB e 99,87% com o KSE-REFINELB.

Apesar da grande variedade de cenários de teste considerados neste trabalho, é importante salientar que ainda é possível ampliar o escopo dos testes para situações com outras quantidades de *pods* e de nós, e com diferentes cargas de trabalho e distribuições de probabilidade de requisições. Apesar de o KSE ter sido executado como um módulo externo ao Kubernetes nos experimentos realizados neste trabalho, é possível que ele seja também executado em um contêiner, hospedado no próprio *cluster* Kubernetes. Devido à sua flexibilidade, o KSE pode ser também integrado a outros ambientes de execução.

Como trabalhos futuros, pretende-se realizar experimentos com mais nós para avaliar o desempenho e escalabilidade dos balanceadores de carga. Adicionalmente, planeja-se avaliar cenários em que a utilização dos recursos dos nós seja mais intensa,

fazendo com que os recursos de nós sejam exauridos devido a altas demandas de carga das aplicações. Pretende-se, ainda, avaliar o desempenho do KSE em cenários com aplicações reais sendo executadas nos *pods*. Por fim, deseja-se estender a implementação do KSE, adicionando suporte a outros algoritmos de balanceamento de carga existentes na literatura, e avaliar o funcionamento destes algoritmos em diversos cenários de uso.

Referências

Ahmad, I., AlFailakawi, M. G., AlMutawa, A., and Alsalman, L. (2021). Container scheduling techniques: A survey and assessment. *Journal of King Saud University - Computer and Information Sciences*.

Burns, B., Beda, J., Hightower, K., and Evenson, L. (2022). *Kubernetes: up and running*. "O'Reilly Media, Inc.".

Buyya, R., Srirama, S. N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., Gelenbe, E., Javadi, B., Vaquero, L. M., Netto, M. A., et al. (2018). A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, 51(5):1–38.

Candel, J. (2022). *Implementing DevSecOps with Docker and Kubernetes: An Experiential Guide to Operate in the DevOps Environment for Securing and Monitoring Container Applications*. BPB Publications.

CNCF (2022). Cloud native computing foundation annual survey 2022 - the year cloud native became the new normal. *CNCF annual survey 2022*.

Jain, S. M., editor (2020). *Linux Containers and Virtualization: A Kernel Perspective*. Apress, California, USA, 1 edition.

James, A. and Schien, D. (2019). A low carbon kubernetes scheduler. In *ICT4S*.

Lamouchi, N. (2021). *Getting Started with Kubernetes*, pages 291–307. Apress, Berkeley, CA.

Lebesbye, T., Mauro, J., Turin, G., and Yu, I. C. (2021). Boreas – a service scheduler for optimal kubernetes deployment. In *Service-Oriented Computing*, pages 221–237. Springer International Publishing.

Luksa, M. (2017). *Kubernetes in action*. Simon and Schuster.

Menouer, T. (2020). KCSS: Kubernetes container scheduling strategy. *The Journal of Supercomputing*, 77(5):4267–4293.

Townend, P., Clement, S., Burdett, D., Yang, R., Shaw, J., Slater, B., and Xu, J. (2019). Invited paper: Improving data center efficiency through holistic scheduling in kubernetes. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE.

Vohra, D. (2017). *Kubernetes Management Design Patterns*. Apress, Berkeley, CA.

Zheng, G., Bhatelé, A., Meneses, E., and Kalé, L. V. (2011). Periodic hierarchical load balancing for large supercomputers. *The International Journal of High Performance Computing Applications*, 25(4):371–385.