Leonardo Valério Anastácio

Desenvolvimento de um escalonador para *Kubernetes* distribuído em microsserviços

Joinville

2022

Leonardo Valério Anastácio

Desenvolvimento de um escalonador para *Kubernetes* distribuído em microsserviços

Universidade do Estado de Santa Catarina – Udesc

Centro de Ciências Tecnológicas

Bacharelado em Ciência da Computação

Orientador: Dr. Guilherme Piêgas Koslovski

Joinville

2022

Leonardo Valério Anastácio

Desenvolvimento de um escalonador para *Kubernetes* distribuído em microsserviços

Trabalho aprovado. Joinville, – de ——— de —-

Dr. Guilherme Piêgas Koslovski
Orientador

Dr. Maurício Aronne Pillon
Membro Banca Examinadora

Dr. Rafael Rodrigues Obelheiro
Membro Banca Examinadora

Joinville
2022

Resumo

Na contemporaneidade, nota-se uma tendência no desenvolvimento de sistemas distribuídos, a qual é impulsionada pela conteinerização de microsserviços provisionados em *data centers*. Especificamente, contêiner é uma tecnologia responsável por encapsular o ambiente da aplicação abstraindo particularidades de sistema operacional, bibliotecas e configurações. Em plataformas de computação atuais (*e.g.*, *data centers*, nuvem) essa tecnologia é implantada em larga escala, pois o contêiner empacota aplicações com o objetivo de otimizar o uso de recurso computacional da máquina hospedeira. O *Kubernetes* é um orquestrador de contêineres responsável por criar, gerenciar e escalar microsserviços na forma de contêineres. O escalonador padrão do *Kubernetes* atua apenas nas estratégias de espalhamento ou agrupamento de contêineres. Para reparar essa deficiência, na literatura encontram-se técnicas de escalonamento que refletem em redução de consumo energético, uso total de recursos ou equidade entre usuários. Entretanto, grande parte desses estudos não consideram o desenvolvimento de um algoritmo escalável. O escalonador baseado em arquitetura distribuída fornece otimização do uso dos recursos do *data center*, como também, melhor tolerância a falhas, por consequência, diminuição considerável do tempo de espera de escalonamento em cenário de falhas. Dessa forma, o presente trabalho visou o desenvolvimento de um escalonador distribuído baseado em microsserviços para *Kubernetes*, a partir do resultados observou-se um bom desempenho da proposta quando comparado com a abordagem padrão de escalonamento do *Kubernetes*.

Palavras-chave: Sistemas distribuídos, microsserviços, contêiner, *Kubernetes*, escalonamento.

Abstract

Nowdays there is a tendency in the development of distributed systems, which is driven by the containerization of microservices provisioned in data centers. Specifically, a container is a technology that encapsulate the application environment, abstract particularities of the operating system, libraries and configurations. In actual computing platforms (*e.g.*, data centers, cloud) this technology is deployed on a large scale, because the container packages applications with the aim of optimizing the use of the host machine's computational resources. The *Kubernetes* is a container orchestrator responsible for creating, managing and scaling microservices represented by containers. The default scheduler of *Kubernetes* only works on spreading or container binpacking strategies. In the literature there are scheduling techniques that reflect in energy consumption reduction, total use of resources or equity between users. But, most of these studies do not consider the development of a scalable algorithm. The scheduler based on distributed architecture provides optimization at the use of *data center* resources, better fault tolerance, consequently, a considerable reduction in the scheduling waiting time in a failure scenario. Therefore, this study aims to develop a distributed scheduler based on microservices for *Kubernetes*, thus increasing scalability and reducing scheduling waiting time and *makespan* metrics.

Keywords: Distributed Systems, microservices, container, *Kubernetes*, scheduling.

Lista de ilustrações

Figura 1 – Componentes *cluster Kubernetes* 18
Figura 2 – Comparação entre Monolítico e Microsserviços 22
Figura 3 – Visualização escalonamento por meio do diagrama de Gantt 23
Figura 4 – Comparação das abordagens de escalonamento 30
Figura 5 – Estratégias de distribuição de cargas entre *master* e *workers* 32
Figura 6 – Comparativo Microsserviço x *Worker* 33
Figura 7 – particionamento do *cluster* entre *workers* 34
Figura 8 – Diagrama de Fluxo 35
Figura 9 – Diagrama de Sequência *Kubernetes Micro Scheduler* (*KMS*) 36
Figura 10 – Processos de eleição do *Master* 38
Figura 11 – Exemplo de implantação. 40
Figura 12 – Direcionamento do *pod* para o escalonador *my-scheduler* 40
Figura 13 – Diagrama de Classes *worker* 41
Figura 14 – Diagrama de Classes *Master* 42

Lista de abreviaturas e siglas

DDD	*Design* orientado a domínio
HPC	*High Performance Computing*
HTTP	Protocolo de Transferência de Hipertexto
k8's	*Kubernetes*
PaaS	Plataforma como um Serviço
QoS	*Quality-of-Service*
RPC	Protocolo de Chamada de Procedimento Remoto
SaaS	Software como um Serviço
SO	Sistema Operacional
VM's	Máquinas virtuais
WWW	*World Wide Web*
KMS	*Kubernetes Micro Scheduler*

Sumário

1 INTRODUÇÃO 9
1.1 Objetivos 11
1.1.1 Objetivo geral 11
1.1.2 Objetivos específicos 11
1.1.3 Organização do texto 12

2 REVISÃO DE LITERATURA 13
2.1 Computação em Nuvem 13
2.1.1 Virtualização e Nuvem 13
2.1.2 Contêineres 14
2.1.3 Contêiner de Aplicação e Contêiner de Sistema 15
2.1.4 Gerenciadores de Contêiner de Aplicação 15
2.2 *Kubernetes* 16
2.2.1 Componentes *Kubernetes* 17
2.2.2 Objetos e serviços Kubernetes 18
2.2.3 Detalhes de escalonamento 19
2.2.4 Customização do *kube-scheduler* 19
2.2.5 Alta disponibilidade 20
2.3 Microsserviços 21
2.3.1 Definições e conceitos 21
2.3.2 Do monolítico ao microsserviços 21
2.3.3 Projeto de Microsserviço guiado pelo *DDD* 22
2.4 Escalonamento de Tarefas 23
2.4.1 Métricas de Escalonamento e Função Objetivo 24
2.4.1.1 Tempo de Espera 24
2.4.1.2 *Makespan* 25
2.4.1.3 *Slowdown* 25
2.4.1.4 Função Objetivo 25
2.4.2 Escalonamento Distribuído 26
2.5 Trabalhos Relacionados 26
2.5.1 Redução do custo energético 27
2.5.2 Otimização multi objetivo 27
2.5.3 Considerações parciais 28

3 KMS - *KUBERNETES MICRO SCHEDULER* 29
3.1 Comparativo com abordagem padrão do *Kubernetes* 30

3.2 Levantamento de Requisitos . . . 31
3.3 Identificação dos Componentes . . . 32
3.3.1 *Master* . . . 32
3.3.2 *Worker* . . . 33
3.3.3 Relação *Master-Worker* . . . 34
3.4 Trocas de Mensagens . . . 36
3.5 Eleição do Componente *Master* . . . 37
3.5.1 Algoritmo de eleição apoiado no *Redis* . . . 37
3.6 Implantação em *Kubernetes* . . . 39
3.7 Representação por Diagramas de Classes . . . 41
3.7.1 Worker . . . 41
3.7.2 Master . . . 42
3.8 Considerações parciais . . . 42

4 ANÁLISE EXPERIMENTAL . . . 43
4.1 Métricas e Parâmetros . . . 43
4.2 Cenário experimental . . . 43
4.3 Protocolo Experimental . . . 44
4.4 Resultados . . . 44

5 CONCLUSÃO . . . 45
5.1 Trabalhos futuros . . . 45

REFERÊNCIAS . . . 47

1 Introdução

Na contemporaneidade nota-se uma tendência das organizações em mover aplicações de escala corporativa para a Computação em Nuvem. As razões para essa ocorrência são múltiplas: alta disponibilidade e redundância, escala automática, gerenciamento integrado de *data center* e fluxo de desenvolvimento (FRITZSCH et al., 2019). Impulsionado pelo paradigma imposto pela Computação em Nuvem, a forma de arquitetar, testar, e implantar software mudou fundamentalmente, de forma que aplicações escaláveis migram de monolíticas para distribuída em microsserviços, para se enquadrar na Computação em Nuvem.

O termo microsserviço surgiu nos últimos anos para descrever um método específico de arquitetar *software* como um conjunto de serviços que são escalados e implementados independentemente (FOWLER MARTIN E LEWIS, 2014). No contexto de Nuvem Computacional, os microsserviços podem ser utilizados na forma de contêineres, que são unidades de software autônomas que encapsulam aplicações e todas suas dependências. No contêiner, os recursos de um nó computacional são compartilhados sem a necessidade da instalação e configuração de dependências. Dessa forma, permitindo que os provedores de estrutura virtual instanciem, realoquem e otimizem suas aplicações de forma mais flexível com desempenho virtual próximo ao *bare metal*. Além de todos esses benefícios, os contêineres oferecem isolamento a nível de sistema operacional, assim, no ponto de vista do usuário, o contêiner executa um sistema operacional independente (FAZIO et al., 2016; ASSUNÇÃO; VEITH; BUYYA, 2018).

A virtualização de contêineres baseada no sistema operacional GNU/Linux é a mais popular, sendo denominada de *Linux container virtualization* (LCV). Gerenciadores LCV encontrados no presente são *Docker*, *Linux Containers* (LXC) e *OpenVZ* (FAZIO et al., 2016). Entre os softwares de virtualização de microsserviços baseados em contêineres, destaca-se *Docker*. Essa tecnologia proporciona funcionalidades além da capacidade de virtualização, pois facilita o processo de criação, construção e controle de versão de imagens de contêineres (RED HAT, 2019b).

Existem ferramentas que gerenciam e coordenam o escalonamento de contêineres em *clusters*. Para a administração de contêineres *docker* é comum o uso de tecnologias como *Docker Swarm* e *Kubernetes* (k8's). Ao utilizar essas ferramentas, o usuário possuirá controle da infraestrutura de maneira simplificada e será capaz de criar, alterar e remover contêineres de forma eficiente. De acordo com Rodriguez (2018), tecnologias como *Docker Swarm* e *Kubernetes* são classificados como orquestradores de contêineres. Os orquestradores atuais, além de oferecer estrutura de controle do ambiente de virtualização, também são

responsáveis pelo escalonamento, replicação e alta disponibilidade das aplicações.

Para Arundel (2019), o *Kubernetes* é considerado o Sistema Operacional das Nuvens Computacionais, sendo o sistema de orquestração de contêineres padrão no mercado. Para isso, o *Kubernetes* oferece um ambiente robusto para implantação de sistemas voltados para Nuvem. As principais características do orquestrador são: escala automática de cargas de trabalho, balanceamento de carga, escalonamento por agrupamento ou espalhamento, e monitoramento do *cluster*.

O *Kubernetes* é um sistema de orquestração que agrupa contêineres em *pods*. *Pod* é um grupo com um ou mais contêineres que compartilham recursos de armazenamento e rede. Dessa forma, contêineres dispostos em um mesmo *pod* são escalonados e executados simultaneamente (GOOGLE KUBERNETES, 2019b). Ou seja, todas as dependências que a aplicação conteinerizada necessita serão lançadas e agrupadas ao mesmo tempo e *pod*, respectivamente. Além disso, a *Google* disponibiliza, de forma oficial, bibliotecas que utilizam API *(Application Programming Interface)* do *cluster Kubernetes* para as principais linguagens de programação (*e.g.*, *.NET*, *Python*, *Java*, *Go*, *JavaScript*, *Haskell*). Por meio da API é possível obter o estado do *cluster*, configurar, lançar e escalonar *pods Kubernetes* (GOOGLE KUBERNETES, 2019a).

Para realizar o escalonamento dos *pods*, o *kube-scheduler* executa um fluxo de operações que são separadas em duas categorias, sendo elas, (1) filtragem e (2) ranqueamento. Em resumo, a filtragem consiste em investigar *nodes* que são capazes de executar o *pod* a ser escalonado, ou seja, nessa etapa há a seleção dos *nodes* do *cluster* que satisfazem a solicitação de recursos do *pod*. O ranqueamento, por sua vez, classifica os *nodes* eleitos pela filtragem e seleciona o *node* que obter a maior pontuação de acordo a solicitação de recursos do *Pod* (GOOGLE KUBERNETES, 2020a).

A principal limitação do *kube-scheduler* é na etapa anterior à Filtragem, o *Kubernetes* utiliza método ingênuo na escolha do próximo contêiner a ser escalonado, que consiste em escalonar de acordo com a ordem de chegada. Essa técnica também é conhecida como *FCFS First-Come-First-Serverd* ou *FIFO First-In-First-Out* (YE et al., 2007). A limitação do *kube-scheduler*, na utilização do *FCFS*, reflete diretamente na escalabilidade do problema, que por consequência, degrada o *makespan* e tempo de espera de escalonamento. Alguns motivos são elencados para defender a escolha do *FCFS*, por exemplo, garantia da ausência de inanição e simples implementação algorítmica. Embora exista o consenso que há espaço de melhoria algorítmica, substituir essa técnica de escalonamento por um algoritmo refinado é uma tarefa complexa (CARASTAN-SANTOS et al., 2019).

Para o desenvolvimento de uma arquitetura distribuída de escalonamento há a necessidade de elencar algumas métricas para validação da abordagem. O presente trabalho considerou de forma primordial duas métricas relacionadas com o desempenho de escalonamento: tempo de espera de escalonamento e *makespan*. O tempo de espera, como a

denominação sugere, é o tempo decorrido do contêiner na fila de espera até ser escalonado, considerado um *delay* do momento em que o contêiner foi submetido à plataforma até sua execução. O *makespan*, uma métrica diretamente proporcional ao tempo de espera, reflete no tempo total que o contêiner permaneceu na plataforma, ou seja, do momento em que foi submetido até a sua finalização. Além dessas duas métricas será analisado o comportamento do *slowdown* na arquitetura distribuída proposta. O propósito do *slowdown* é estabelecer proporção entre tempo de espera de um trabalho em relação ao seu tempo de processamento. As métricas aqui apresentadas, de forma resumida, são melhor exploradas na Seção 2.4.1.

1.1 Objetivos

1.1.1 Objetivo geral

Os objetivos gerais deste trabalho refletiram no desenvolvimento de um escalonador distribuído para *Kubernetes*, utilização da abordagem de microsserviços na implementação e validação da abordagem diante as métricas de otimização: tempo de espera de escalonamento e *makespan*.

1.1.2 Objetivos específicos

- Compreender *Kubernetes*;
- Revisar escalonamento descentralizado (utilizando particionamento e duplicação de trabalho);
- Revisar orquestração de contêineres;
- Revisar arquitetura de microsserviços;
- Desenvolver um escalonador distribuído baseado em microsserviços;
- Analisar o desempenho do escalonador proposto em relação ao tempo de espera e *makespan*; e
- Analisar o comportamento do *slowdown* em arquitetura distribuída.

1.1.3 Organização do texto

O presente trabalho está organizado a partir da seguinte estrutura: o Capítulo 2 aborda uma revisão de literatura acerca do tema proposto, desde revisão de Computação em Nuvem até alcançar as especificidades do projeto revisando *Kubernetes*, microsserviços e escalonamento de tarefas. Ao fim do Capítulo 2 são apresentados os trabalhos relacionados sobre escalonamento de contêineres com diferentes objetivos. No Capítulo 4 é detalhado a proposta distribuída como também explora os seus resultados, ao passo que o Capítulo 5 conclui o trabalho.

2 Revisão de Literatura

Este capítulo aborda os conceitos fundamentais relacionados ao presente trabalho. Na Seção 2.1 é apresentada os principais conceitos referentes a virtualização baseada em contêineres, em seguida, na Seção 2.2 é realizada uma revisão sobre os componentes principais do *Kubernetes*. Logo, na Seção é 2.3, é introduzido o tema microsserviços, por fim, na Seção 2.4 é apresentada uma revisão acerca do tema escalonamento de tarefas.

2.1 Computação em Nuvem

Há muitas definições formais para o termo Computação em Nuvem. A definição internacionalmente aceita pelo *NIST* (*National Institute of Standards and Technology*)(MELL; GRANCE et al., 2011) visa o acesso compartilhado, isolado e sob-demanda para um ambiente de recursos computacionais (*e.g.*, redes, servidores, discos, aplicações e serviços). Nesse contexto, os recursos de computação são agrupados e isolados para atender multiusuários. Ou seja, usuários isolados que compartilham recursos computacionais, os quais são dinamicamente provisionados conforme a demanda. Nesse formato há a exigência de rápido ajuste de acordo com a necessidade de recursos (Bernstein, 2014).

2.1.1 Virtualização e Nuvem

A virtualização é considerada uma técnica que permite criar ambientes simulados a partir de um único sistema, seja ele físico ou não. Dessa forma, classifica-se virtualização como tecnologia, enquanto que Nuvem como um ambiente, conjunto de recursos computacionais gerenciável, que, pode ou não, manipular virtualização (RED HAT, 2020). Nos dias atuais, a virtualização de recursos é um dos pontos chaves das Nuvens modernas, a partir da virtualização as Nuvens são capazes de gerenciar ambientes elásticos, que compartilham recursos computacionais isolados logicamente (ZHANG; CHENG; BOUTABA, 2010). Portanto, Virtualização e Nuvem não são sinônimos.

A virtualização convencional é a baseada na execução de um *hipervisor* no topo de uma máquina física. Um *hipervisor* é um software que cria e executa Máquinas virtuais (*VM's*). Ao utilizar *hipervisor* o administrador é capaz de otimizar o uso de recursos físicos e modular partes individuais da infraestrutura no formato de *VM's*. Assim, os recursos são consumidos com mais eficiência na arquitetura *hipervisor* quando comparado com implantações *bare-metal* (direto em *hardware*). Em contrapartida, apesar da otimização em relação ao *bare-metal*, o *hipervisor* executa múltiplos *kernels* em uma única máquina física, tornando custoso o isolamento das aplicações e processos (SCHEEPERS, 2014).

O presente trabalho não realizará revisão completa sobre virtualização em nuvem, uma vez que na literatura são encontrado diversos trabalhos exploratórios acerca do assunto (ASSUNÇÃO; VEITH; BUYYA, 2018), (LU; ZENG, 2014), (ACETO et al., 2013)

2.1.2 Contêineres

Na contemporaneidade, os contêineres são considerados a abordagem padrão para executar cargas de trabalho. No contexto do presente trabalho, um contêiner reflete em uma unidade computacional, que é responsável por executar as aplicações. Para gerenciamento de contêineres existem diferentes orquestradores, como por exemplo, *Kubernetes* e *docker swarm*.

Um contêiner compreende em um conjunto de um ou mais processos dispostos de forma isolada do Sistema Operacional (*SO*) hospedeiro. Todas dependências que são necessárias para execução do contêiner são fornecidos por uma imagem distinta. Para a maioria dos motores de contêineres, a imagem representa os passos necessários para construção de um contêiner. A virtualização baseada em contêineres (conteinerização) é considerada uma solução portável, pois permite executar o mesmo contêiner em implantações distintas. Portanto, os contêineres são ideais para encapsular aplicações, migrá-las para hospedeiros distintos sem maiores impasses, e implantar em Nuvem RED HAT (2019a).

Na literatura há diversas comparações entre contêineres e *VM's*, como desempenho, uso de recursos e segurança. O contêiner é um processo que utiliza o mesmo *kernel* que o *SO* hospedeiro. Já a *VM's* simula um *SO* pleno. Outro ponto a destacar, por convenção, o tamanho de um contêiner é mensurado em *megabytes*, já a *VM's* em *gigabyte*. Além disso, nota-se uma tendência de utilizar contêineres para construção de sistemas baseado em microsserviços. Os principais fatores para essa associação é que os contêineres se enquadram no fluxo de desenvolvimento e implantação em Nuvem, são mais leves, e possuem tempo de construção e inicialização inferiores as tradicionais *VM's*.

Tanto a conteinerização quanto a virtualização por *hipervisor* são soluções utilizadas para a implantação de Software como um Serviço (*SaaS*) em Nuvem. Entretanto, ao passar do tempo, a conteinerização se tornará um padrão para construção de *SaaS* com características escaláveis, já as *VM's* irão continuar com papel importante no provisionamento de *SO* sob-demanda e no fornecimento de Plataforma como um Serviço (*PaaS*).

As tecnologias de conteinerização são consideradas um ecossistema que fornecem ferramentas para a conteinerização. Esse ecossistema é composto por Contêiner de Aplicação, Contêiner de Sistema, Gerenciador de Contêiner e Orquestrador de Contêineres. Nessa seção serão explanados os dois primeiros sistemas, enquanto que Orquestrador de contêineres será explicado na seção *Kubernetes*.

2.1.3 Contêiner de Aplicação e Contêiner de Sistema

Apesar de se basearem nas mesmas tecnologias e conceitos, há uma diferença clara entre Contêiner de Aplicação e Contêiner de Sistema, e está relacionada ao seu uso. Contêiner de Sistema visa a execução de um *SO* completo, e o Contêiner de Aplicação é designado à implantação de uma aplicação ou de um componente de uma aplicação Casalicchio e Iannucci (2020). Contêiner de Aplicação refere-se a um contêiner cujo único encargo é executar uma única aplicação dentro de um ambiente isolado. Entretanto, há a possibilidade de utilizar Contêiner de Aplicação como Contêiner de Sistema, mas esse comportamento não está de acordo com *Docker development guidelines and best practices* (BERG; CRAMP; SIEGEL, 2016). Pois Contêiner de Aplicação são designados para executar aplicações no formato de microsserviços. Diferente dos Contêiner de Sistema que são projetados para, em nível de comparação, executar um *SO* de forma similar as *VM's*. Um exemplo de Contêiner de Aplicação é o *docker*, um exemplo de Contêiner de Sistema é o *Linux Containers (LXD)*.

2.1.4 Gerenciadores de Contêiner de Aplicação

O Gerenciador de Contêiner de Aplicação é considerado um *framework* que fornece ferramentas que auxiliam no gerenciamento de todo o ciclo de vida do contêiner. O ciclo de vida de um contêiner, desde o desenvolvimento até execução em ambiente de produção, de acordo com *IBM* (Jason McGee, 2016), compreende em:

Obtenção: Aquisição da imagem do contêiner. Essa etapa é considerada o ponto inicial para a criação de um novo contêiner. Em contêineres *docker* há a concepção de camadas de imagens, significa que uma imagem pode ser derivada de outras imagens. Além disso, a imagem pode ser limpa, reflexo direto do *kernel*.

Construção: Em *docker* há um arquivo de definição chamado *Dockerfile*, nele é definido todas as dependências e a maneira que a aplicação será executada. Normalmente em um arquivo *Dockerfile* há a escolha da imagem base, passos para montagem do contêiner, e um ponto de entrada para execução de um processo, comumente são utilizados processos em segundo plano. Para que ocorra a construção do contêiner com sucesso, todos os passos de montagem devem ser executados com êxito.

Entrega: Esse passo é referente a forma que será concebida a entrega da imagem do contêiner, já construído, para o ambiente de produção.

Implantação: Consiste na implantação da imagem do contêiner em ambiente de produção. Ou seja, essa etapa envolve todos os passos necessários para que uma nova imagem seja implantada no ambiente de produção.

Execução: Nessa etapa é definido o ambiente de execução do contêiner, que pode envolver: dimensionamento da aplicação por meio de réplicas, verificação de falhas e conexão com outros serviços e réplicas.

Gerenciamento: O passo final refere-se ao gerenciamento da aplicação conteinerizada em ambiente de produção. Nessa fase, a aplicação pode estar em execução ou em suspensão. Em gerenciamento, comportamento da aplicação é monitorada, o sistema deve ser capaz de gerenciar as falhas nos contêineres que estão em execução, por exemplo, forçando a reinicialização. Enquanto que a suspensão é um estado para depuração da aplicação, com o objetivo de encontrar a origem da falha. Posteriormente, o processo passa por todas as fases novamente, iniciando em Obtenção.

Gerenciadores de Contêineres são classificados em auto-hospedados ou baseados em soluções gerenciadas. Soluções auto-hospedadas percorrem todo o ciclo de implementação de um Gerenciador de Contêiner de Aplicação: instalação, configuração e gerenciamento em *datacenters* privados, máquinas virtuais ou até mesmo em *clouds* geograficamente distribuídas. Por outro lado, soluções gerenciadas são ofertados por provedores de nuvem e necessitam apenas de configuração parcial (CASALICCHIO; IANNUCCI, 2020). Assim, as plataformas de conteinerização *docker, LXD, OpenVZ* podem ser configuradas da mesma maneira que Gerenciadores de Contêiner de Aplicação, portanto, são exemplos de soluções auto-hospedas. Os exemplos mais populares de soluções gerenciadas são das empresas *Google* e *Amazon* que possuem serviços como, respectivamente, *GKE* (*Google Kubernetes Engine*) e *EKS* (*Elastic Kubernetes Service*).

2.2 *Kubernetes*

Para Arundel (2019), o *Kubernetes* é considerado o Sistema Operacional das Nuvens Computacionais, sendo o sistema de orquestração de contêineres padrão no mercado. Para isso, o *Kubernetes* oferece um ambiente robusto para implantação de sistemas voltados para nuvem. As principais características do orquestrador são: escala automática de cargas de trabalho por meio de replicação serviço, balanceamento de carga, escalonamento por agrupamento ou espalhamento, e monitoramento do *cluster*.

O *Kubernetes* é um sistema de orquestração que agrupa contêineres em *pods*. *Pod* é um grupo com um ou mais contêineres que compartilham recursos de armazenamento e rede. Dessa forma, contêineres dispostos em um mesmo *pod* são escalonados e executados juntos (GOOGLE KUBERNETES, 2019b). Ou seja, todas as dependências que a aplicação conteinerizada necessita serão lançadas e agrupadas ao mesmo tempo e *pod*, respectivamente. Além disso, a *Google* disponibiliza, de forma oficial, bibliotecas que utilizam API *(Application Programming Interface)* do *cluster Kubernetes* para as principais linguagens de programação (*e.g.*, *.NET*, *Python*, *Java*, *Go*, *JavaScript*, *Haskell*). Por meio da API

é possível obter o estado do *cluster*, configurar, lançar e escalonar *pods* no *Kubernetes* (GOOGLE KUBERNETES, 2019a).

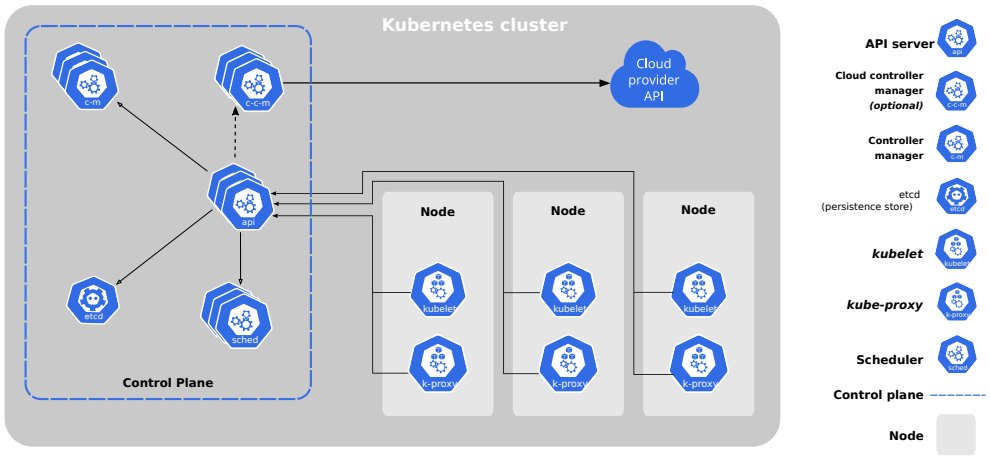
2.2.1 Componentes *Kubernetes*

Um *cluster Kubernetes* consiste em um conjunto de servidores, chamados de *nodes*, que executam serviços conteinerizados. O sistema principal do *Kubernetes* é o *Control Plane*, que é responsável por tomada de decisões, escalonamento e controle do estado do *cluster*. O *Control Plane* reside em um *node* específico denominado *Master* e consiste em cinco componentes:

- *Kube-apiserver*: Servidor *frontend* para o *Control Plane*, trata requisições que são direcionadas à *API* do *Kubernetes*;
- *Etcd*: Banco de dados, do tipo *chave-valor*, que armazena informações sobre o estado do *cluster*;
- *Kube-scheduler*: Componente responsável por observar novos *pods* e escaloná-los em um *node*;
- *Kube-controller-manager*: Executa os controladores de recurso, que incluem: Controlador de *nodes*, Controlador de Réplicas, Controlador de Autenticação. Este componente é responsável verificar o estado do *cluster*, das cargas de trabalho e serviços, com objetivo de mover o estado atual para o estado desejado.
- *Cloud-controller-manager*: Responsável por gerenciar a interação com o provedor de Nuvem.

O diagrama de um *cluster kubernetes* com todos os componentes interligados é visualizado na Figura 1.

Figura 1 – Componentes *cluster Kubernetes*



Fonte: Kubernetes Documentation (2019)

2.2.2 Objetos e serviços Kubernetes

Além dos componentes do *Control Plane* existem outros objetos e serviços que são fundamentais para a conteinerização, funcionamento do *Kubernetes*, ou são pertinentes a este trabalho.

***Pods*:** representa a menor unidade de objeto no *Kubernetes*. Em resumo um *pod* é uma carga de trabalho que é executado em *nodes* do *cluster* no formato de um processo. *Pod* é considerado um grupo com um ou mais contêineres, os quais compartilham recurso de disco e rede, cada *pod* contém especificação declarativa da execução dos contêineres que possui. A comunicação entre os contêineres de um mesmo *pod* é feita por meio de *inter-process comunication (IPC)*, como semáforos *SystemV*, ou, utilizando a memória compartilhada *POSIX*. Por padrão, contêineres em *pods* distintos não se comunicam por meio de *IPC* sem configuração personalizada. Desse modo, a comunicação entre contêineres em *pods* distintos é concebida via troca de mensagem utilizando protocolo *IP*.

***ReplicaSet*:** Considerado um serviço responsável pelo controle de réplicas de *pods* solicitado. Assim, o *ReplicaSet* garante a disponibilidade de um número de *pods*.

***Deployments*:** fornece atualizações declarativas para *pods* e *replicaSets*. Ou seja, o *deployment* é usado para atualização de estado de *pods* e *ReplicaSets*. O uso comum de *deployments* em relação aos *pod* é na atualização de imagem do contêiner, já para *ReplicaSet* seria na atualização no número de réplicas disponíveis desejado.

2.2.3 Detalhes de escalonamento

Para realizar o escalonamento dos *pods*, o *kube-scheduler* executa um fluxo de operações que são separadas em duas categorias: Filtragem e Ranqueamento. Em resumo, a filtragem consiste em investigar *nodes* que são capazes de executar o *pod* a ser escalonado, ou seja, nessa etapa há a seleção dos *nodes* do *cluste* que satisfazem a solicitação de recursos do *pod*. O ranqueamento, por sua vez, classifica os *nodes* eleitos pela filtragem e seleciona o *node* que obter a maior pontuação de acordo a solicitação de recursos do *pod* (GOOGLE KUBERNETES, 2020a).

A técnica de escalonamento utilizada pelo *kube-scheduler* é denominada *First-Come-First-Served* (FCFS), conhecido também como *First-In-First-Out* (FIFO), que consiste em escalonar os serviços pela ordem de chegada. Alguns motivos são elencados para defender a escolha dessa técnica, por exemplo, garantia da ausência de inanição e simples implementação algorítmica. Embora exista o consenso que há espaço de melhoria, substituir essa técnica de escalonamento por um algoritmo aprimorado requer um estudo de caso específico bem definido (CARASTAN-SANTOS et al., 2019).

2.2.4 Customização do *kube-scheduler*

De acordo com Wei Huang (2019) há 4 meios de customizar o escalonador do *Kubernetes*, que serão abordados nessa seção.

Alteração do código fonte: A primeira técnica consiste em clonar o repositório do código fonte do *Kubernetes*, modificar o comportamento do escalonador padrão, recompilar o projeto e executar o escalonador. Essa prática não é recomendada pois há a necessidade de alinhar o código alterado com o fluxo de execução do *Kubernetes*, o segundo fator que desmotiva esse método, é que o próprio *Kubernetes* oferece meios para alterar o comportamento do escalonador sem a necessidade de alterar o código fonte.

Múltiplos escalonadores: A segunda técnica é executar um escalonador customizado ao lado do escalonador padrão. O escalonador customizado é conteinerizado no formato de *pod* e orquestrado pelo próprio *Kubernetes*. Para que o escalonador padrão e o customizado não disputem os mesmos *pods*, na criação do *pod* é preciso identificar o método de escalonamento por meio do nome do escalonador. Caso contrário, a co-existência de múltiplos escalonadores causa o bloqueio de sincronização. A comunicação do *pod*, abstraído em escalonador de contêineres, com a *API* do *Kubernetes* é custosa. Pois o *pod* irá consumir a *API* do *Kubernetes* por meio do protocolo Protocolo de Transferência de Hipertexto (*HTTP*).

Extensão do escalonador padrão: A terceira solução é denominada Extensão, considerada a solução mais simples, que objetiva adicionar funcionalidades extras ao *kube-scheduler*. Extensão é entendido como a configuração de ganchos (*webhooks*) que o *Kubernetes* oferece para executar filtragem e ranqueamento dos *nodes* de forma customizada. Ganchos devem ser interpretados como gatilhos, que são funções personalizadas que o *Kubernetes* engatilha para alterar algum comportamento padrão ou adicionar novas funcionalidades. Devido a simplicidade, esse método apresenta algumas limitações. A transferência de dados, entre o escalonador customizado e o padrão, é realizada via protocolo *HTTP*, ocasionando custo alto de comunicação. O segundo problema está relacionado com a própria limitação da abordagem, pois altera apenas os procedimentos das fases de filtragem e ranqueamento, não atuando no início nem no fim de nenhuma outra etapa de escalonamento.

***Framework* de escalonamento:** O quarto método de customizar o escalonador padrão do *Kubernetes* é denominado *framework* de escalonamento. Essa técnica consiste em adicionar pontos de extensão no escalonador padrão do *Kubernetes*, chamados *plugins*, que são incluídos em tempo de compilação. Os *plugins* podem ser habilitados, desabilitados e reordenados. *Plugins* são adicionados nos ciclos padrões de escalonamento – *scheduling* e *binding*. No ciclo *scheduling* está habilitado a extensão por meio de 8 *plugins*: *sort, PreFilter, Filter, PreScore, Score, NormalizeScore, Reserve, Permit*. Já, na fase de *binding* há 3 pontos de extensão: *PreBind, Bind, PostBind*. O presente trabalho não realizará revisão completa sobre os pontos de extensão, uma vez que a documentação explora este assunto de forma ampla (Kubernetes Documentation, 2020).

2.2.5 Alta disponibilidade

A configuração de um *cluster Kubernetes* visando alta disponibilidade é essencial para o uso de aplicações conteinerizadas em produção. A alta disponibilidade é alcançada a partir da replicação do *node master*, com isso, eliminando um único ponto de falha para os componentes do *Control Plane*, que são fundamentais para o funcionamento do *Kubernetes* (GOOGLE KUBERNETES, 2020b).

Em uma implantação do *Kubernetes* com configuração de alta disponibilidade, a instância do banco de dados *etcd* será replicadas utilizando um algoritmo de consenso, e todos os servidores *kube-apiserver* estarão disponíveis por meio de um balanceador de carga. Enquanto que os demais componentes (*Kube-controller-manager*, *Cloud-controller-manager*, *kube-scheduler*) estarão apenas com uma instância ativa no *cluster*. Portanto, a replicação do *node master* torna-se uma solução eficaz para tolerância a falhas, contudo, não resolve a escalabilidade do problema de escalonamento. Isso ocorre porque as réplicas do *kube-scheduler* não atuam em conjunto no processo de escalonamento.

2.3 Microsserviços

De acordo com Fowler Martin e Lewis (2014) não há definição precisa de microsserviços, contudo, existem algumas características em comum acerca das implantações dessa arquitetura. Como por exemplo, um conjunto de serviços que constitui um sistema, controle descentralizado da informação, automação da implantação de software e uso heterogêneo de linguagens de programação e banco de dados.

2.3.1 Definições e conceitos

A arquitetura de microsserviços visa o desenvolvimento de um sistema como um conjunto de serviços sucintos seccionados em processos independentes, podendo ou não compartilhar o mesmo hospedeiro (LEWIS, 2012). Isto é, microsserviços são pequenas aplicações que podem ser implantadas, escaladas e testadas independentemente, que possuem um conjunto limitado de funcionalidades para resolver um só objetivo. Uma única responsabilidade, por um lado, deve ser interpretado como uma única razão para mudar ou uma única razão para ser substituído. Por outro lado, deve ser interpretada como um sistema que possuí uma funcionalidade apenas, o qual tem de ser facilmente compreendido fora de seu contexto. Considera-se que o conceito de microsserviço está relacionado com a filosofia *unix*: "Escreva programas que façam apenas uma coisa, mas que façam bem feita"e "Escreva programas que trabalhem juntos"(THÖNES, 2015).

Para a interação entre os serviços que compõe um sistema, os microsserviços se beneficiam de protocolos leves para troca de mensagem, como Protocolo de Chamada de Procedimento Remoto (*RPC*), protocolo de mensagens e, se for necessário, é possível utilizar *HTTP*. O *HTTP* é a forma habitual que o navegador carrega páginas da *World Wide Web* (*WWW*), enquanto que, *RPC* e fila de mensagens são protocolos comumente empregado para comunicação de sistemas distribuídos (RAYMOND, 2003).

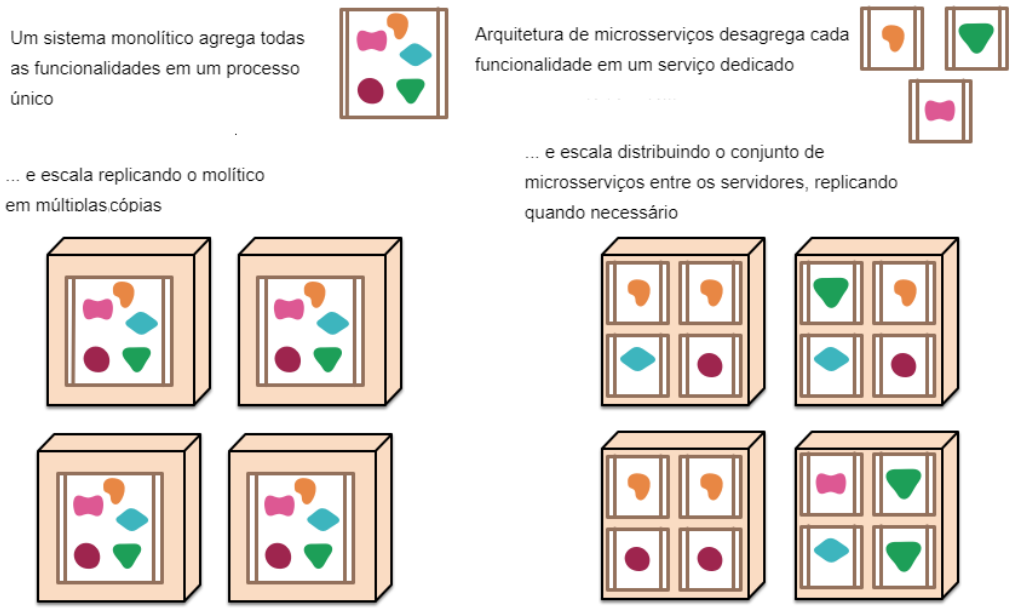
2.3.2 Do monolítico ao microsserviços

Em engenharia de software, um sistema monolítico constituí-se de uma única unidade, todo código é vinculado a um único processo. Aplicações comerciais geralmente são dividas em três partes, lado cliente, habitualmente uma página *web*, banco de dados e uma aplicação do lado servidor. Em uma aplicação *web*, esse servidor lida com o *HTTP*, executa a regra de domínio e persiste informações no banco de dados. O sistema do lado servidor é considerado um monolítico. Por consequência, qualquer alteração no sistema influenciará em construir e implantar uma nova versão da aplicação.

Desenvolver de forma monolítica é a maneira natural de construir sistemas, toda a lógica é gerenciada por um único processo. O projeto para esse tipo de sistema equivale a segregar as partes/módulos do sistema por meio das técnicas de programação da linguagem

utilizada, por exemplo classes, interfaces, hierarquia, etc. Entretanto, sistemas monolíticos não performam na questão de escalabilidade. Pois escalar monolítico consiste em replicar o sistema de forma integral. Já em uma aplicação baseada em microsserviços proporciona um escalabilidade inteligente, pois replica as partes de acordo com a necessidade por meio de um balanceador de carga dedicado para cada serviço, como ilustra a figura 2.

Figura 2 – Comparação entre Monolítico e Microsserviços



Fonte: Fowler Martin e Lewis (2014)

2.3.3 Projeto de Microsserviço guiado pelo *DDD*

O *Design* orientado a domínio (*DDD*), no contexto de engenharia de software, é uma metodologia que conecta conceitos de linguagem de programação, por exemplo nome de classes, métodos, e atributos com o domínio do negócio (DDD Community, 2007). Define-se domínio como a área de conhecimento, ou seja, as funcionalidades do sistema a nível de negócio (EVANS, 2014).

De acordo com Fowler Martin e Lewis (2014), o *DDD* decompõe um domínio complexo em múltiplas partes, na literatura é definido como contextos delimitados (*bounded context*), como também visa o projeto do relacionamento entre esses. Este padrão é utilizada tanto para projetar sistemas monolíticos quanto distribuída em microsserviços. Nota-se uma correlação semântica entre um serviço e um contexto delimitado, que reforça a separação lógica do domínio em serviços isolados, consequentemente, em microsserviços. Isto é, o contexto delimitado do conceito do *DDD* se torna um excelente candidato à um microsserviço (NEWMAN, 2015). Contudo, nem todo contexto delimitado deve ser relacionado à um microsserviço independente. Segundo o criador do *DDD*, Eric Evans,

há diferentes tipos de contexto delimitado que não mapeiam diretamente a um serviço isolado.

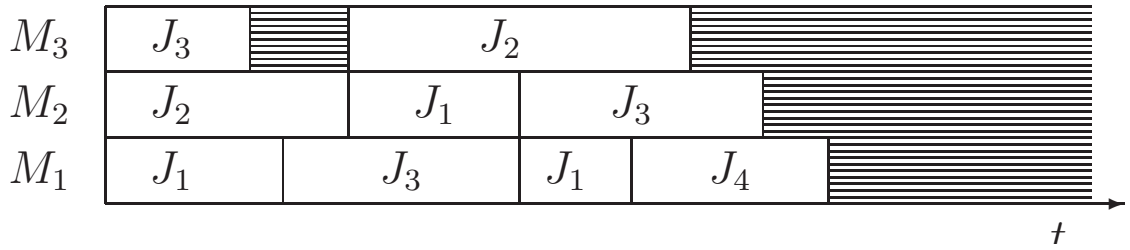
Um dos desafios de projetar sistemas baseados em serviços distribuídos é determinar a granularidade em termos de escopo e complexidade do microsserviço. O conjunto de funções que definem o microsserviço não deve ser extremamente simplista, muito menos agregar muita complexidade (MERSON; YODER, 2020). Portanto, para permitir o gerenciamento descentralizado de acordo com o contexto do domínio, o ideal é unir os princípios do *DDD* com a arquitetura de microsserviços.

2.4 Escalonamento de Tarefas

Escalonamento consiste um processo de tomada de decisão, que é usado regularmente no ramo da manufatura, serviços industriais e sistemas operacionais. O escalonamento lida com alocação de recursos para tarefas em determinados períodos de tempo, visando otimizar um ou mais objetivos (PINEDO, 2012). Sendo alguns desses objetivos, por exemplo, redução do tempo de espera das tarefas, maximização do uso de recursos (espalhamento), ou minimização do uso total de recursos (agrupamento).

Neste trabalho, escalonamento é um procedimento o qual soluciona alocações de recursos computacionais. Recursos refletem em quantidade mensurável de memória, processamento e dispositivos de rede de um, ou conjunto, de nós computacionais (também denominado por máquinas). Segundo Brucker et al. (1999), considere que m máquinas $M_j (j = 1, \ldots, m)$ deverão processar n trabalhos $J_i (i = 1, \ldots, n)$. Vinculado a cada trabalho J_i há um número n_i de tarefas $(O_{i1}, O_{i2}, \ldots, O_{in_i})$, para cada tarefa há uma solicitação de recursos p_{ij}. Dessa forma, o escalonamento é um processo de tomada de decisão que, a partir da requisição de recursos da tarefa, investigará os nós computacionais factíveis e indicará o escalonamento ideal de J_i para M_j de acordo com a otimização de alguma métrica de interesse. Um escalonamento válido para alocação de trabalhos pode ser representado por meio do diagrama de Gantt, como exemplifica a figura 3.

Figura 3 – Visualização escalonamento por meio do diagrama de Gantt



Fonte: Brucker et al. (1999)

De acordo com (PINEDO, 2012), o problema de escalonamento é representado por uma tripla $\alpha|\beta|\gamma$. O campo α representa o ambiente, interpreta-se como o perfil da arquitetura que detém os recursos em que os trabalhos serão escalonados. O campo β define detalhes de processamento e restrições que estão vinculados ao trabalho, pode conter nenhuma, uma única ou várias entradas. Já o campo γ descreve a função objetivo a qual visa otimização de alguma métrica de interesse, normalmente contém apenas uma entrada.

Para este trabalho, a entrada γ será definida pelo perfil **Máquinas em Paralelo com Diferentes Velocidades** e é representado pelo rótulo Qm. Este cenário consiste em m máquinas heterogêneas, que não necessariamente possuem a mesma quantidade de recursos. Considera-se a principal abordagem das nuvens computacionais atualmente (KRAUTER; BUYYA; MAHESWARAN, 2002).

A entrada β é definida pelo conjunto **Prazo ou *deadline*** (P), **Restrição de Elegibilidade** (M) e **Processamento em Lote** ($batch(b)$). Se o rótulo P está presente em β, então para cada trabalho J_k será atribuído um prazo de entrega P_k, dessa forma, J_k não deve ser escalonado após o prazo P_k. Se P não está incluído em β, logo não há restrição de *deadline* de escalonamento. O rótulo M denota que nem todas as máquinas são capazes de processar o trabalho J_k, assim, o conjunto M_k representa as máquinas que satisfazem as restrições de J_k, que por consequência, são elegíveis ao escalonamento. Por último, se há rótulo $batch(b)$ em β, então, uma máquina é apta a processar quantidade b de trabalhos de forma simultânea.

Portanto, $\beta = \{P, M, batch(b)\}$. O campo γ, que está relacionado com métricas de escalonamento e funções objetivos, será desenvolvido na seção subsequente.

2.4.1 Métricas de Escalonamento e Função Objetivo

De acordo com (FEITELSON, 1998), há um conjunto de métricas relevantes a respeito do desempenho de algoritmos de escalonamentos. O presente trabalho visa a otimização das seguintes métricas: tempo de espera de escalonamento, *makespan*. Além disso, análise do comportamento do *slowdown*.

2.4.1.1 Tempo de Espera

Considere $Submetido_k$ o momento em que o trabalho J_k foi submetido à plataforma, $Inicio_k$ o momento em que o trabalho J_k iniciou sua execução. A Equação 2.1 mensura o tempo de espera, ou seja, o tempo que o trabalho permanece na fila até ser escalonado. Para um trabalho J_k o tempo de espera T_k é determinado pela equação:

$$T_k = Inicio_k - Submetido_k \tag{2.1}$$

Minimização do tempo de espera é um problema de escalonamento conhecido importante no fornecimento de *Quality of Service (QoS)* em muitas indústrias (YE et al., 2007). Por ser classificado como um problema *NP-Hard* (KUBIAK, 1993), na literatura é encontrada diferentes heurísticas que aproximam-se da solução ideal, pois não há algoritmo de busca eficiente que encontre uma sequência ótima (YE et al., 2007).

2.4.1.2 *Makespan*

A próxima métrica a ser considerada é o *makespan*, que é uma métrica diretamente vinculada ao tempo de completude de um trabalho. O cálculo é definido pelo tempo de término da última tarefa de um trabalho a deixar o sistema. Considere que um trabalho J_k possui n tarefas vinculadas $O_{k,1}, O_{k,2}, ..., O_{k,n}$, e para cada tarefa $O_{k,i}$ há vinculado tempo de finalização C_i, assim o *makespan* $C_{max,k}$ de J_k é calculado de acordo com a equação:

$$C_{max,k} = max(C_1, C_2, ..., C_n) \tag{2.2}$$

A minimização do *makespan* resulta na otimização da utilização dos recursos computacionais (PINEDO, 2012). Pois o *makespan* está relacionado ao tempo que o trabalho permanece na plataforma, a sua minimização acarreta em melhor agrupamento do escalonamento, que por consequência, ocorrendo a liberação do uso de recursos em um menor período de tempo.

2.4.1.3 *Slowdown*

Por último, outra métrica abordada neste trabalho é o *slowdown*, que é definido pela relação entre o tempo total que o trabalho permaneceu na plataforma com o tempo atual de processamento gasto com o mesmo. Assim o *slowdown* Sd_k de J_k é definido pela Equação 2.3:

$$Sd_k = \frac{T_k + P_k}{P_k} \tag{2.3}$$

P_k é o tempo de processamento de J_k. O propósito do *slowdown* é estabelecer proporção entre tempo de espera de um trabalho em relação ao seu tempo de processamento (MACCIO; HOGG; DOWN, 2018), com o objetivo de atribuir uma distribuição equilibrada do tempo de espera entre os trabalhos com diferentes cargas de trabalho (CARASTAN-SANTOS et al., 2019).

2.4.1.4 Função Objetivo

Portanto o campo γ (função objetivo) da definição de escalonamento refere-se à minimização do tempo de espera, *makespan*, logo, $\gamma = T, C_{max}$. Em cenários como

High Performance Computing (HPC), os resultados visados por meio da otimização desse conjunto de métricas são: Melhor *QoS* por meio da minimização do tempo de espera; Otimizar a utilização dos recursos computacionais mediante a minimização do *makespan*; Analisar o comportamento do *slowdown*.

2.4.2 Escalonamento Distribuído

Aplicações atuais fundamentam-se na análise e processamento de um grande volume de dados, como por exemplo, *Data Mining*, *Machine Learning*, *Deep Learning* e banco de dados. Dessa forma, conforme a demanda de poder computacional cresce, as estruturas responsáveis pela execução dessas aplicações os acompanham, refletindo no aumento de complexidade tanto em tamanho como em algoritmos refinados de gerenciamento (WANG et al., 2016). A nível de comparação desse reflexo, a empresa *Google* executa centenas de milhares de cargas de trabalho, de muitos milhares de aplicativos em uma conjunto de *clusters*, cada qual com até dezenas de milhares de máquinas (VERMA et al., 2015). Por consequência, os componentes internos de gerenciamento de um *data center* de larga escala deve resolver os problemas internos com algoritmos complexos e sofisticados para cada tipo de cenário.

O escalonamento é um tema amplamente pesquisado, desde de otimização por meio de processamento em *GPU* (NESI et al., 2018) como também baseado em arquitetura descentralizada empregando conceitos de *blockchain* (LOCH et al., 2021). De acordo com a literatura, escalonamento distribuído, hoje, é requisito fundamental para *Data Centers* de larga escala (VERMA et al., 2015; WANG et al., 2019). Uma vez que, a principal dificuldade da abordagem centralizada está na escalabilidade e nos métodos de tolerâncias a falhas, que degradam por completo as métricas de *Quality-of-Service* (*QoS*). O escalonamento, no formato distribuído, resolve este problema de forma elaborada removendo o único ponto de falha ao particionar as requisições de alocação de recursos em um sistema distribuído.

2.5 Trabalhos Relacionados

A área de escalonamento de tarefas é estudada a décadas, considera-se um ramo com intersecção em diferentes campos da ciência da computação. Um dos principais fatores que impulsionam a intersecção entre as áreas está na natureza do problema de escalonamento, pois na maioria dos casos resolver um problema de escalonamento reflete em otimizar um problema *NP-difícil*. Dessa forma, na literatura, encontra-se diferentes pesquisas que refletem em soluções heurísticas e refinadas, muitas vezes relacionadas com a área de inteligência artificial. Por se tratar da solução de um problema *np-difícil*, a solução computada basta ser boa o suficiente para um cenário específico, sendo assim, a solução encontrada nem sempre é refletida na ótima global.

Esta seção analisará trabalhos recentes da literatura que possuem relação com escalonamento de contêineres. Por ainda se tratar de um recorte amplo na área de escalonamento, aqui analisou-se diferentes trabalhos que atacam diferentes características, seja relacionado com otimização energética em *data centers* como também trabalhos que buscaram otimizar o desempenho de escalonamento.

2.5.1 Redução do custo energético

No trabalho de Sureshkumar e Rajesh (2017) os autores propuseram um algoritmo de escalonamento de contêineres, para a tecnologia *docker*, baseado em balanceamento de carga. O algoritmo consiste em um limiar calculado a partir da sobrecarga do *cluster*, o objetivo do limiar é que as cargas dos contêineres não sejam muito altas e baixas. Quando a carga ultrapassa o limiar, um novo contêiner é criado no balanceamento de carga. Por outro lado, quando a carga é muito baixa, o contêiner é destruído com o objetivo de economizar custo energético.

2.5.2 Otimização multi objetivo

Em Liu et al. (2018) os autores desenvolveram um novo algoritmo de escalonamento de contêineres denominado *multipot*. Neste projeto foi estudado múltiplos critérios para a seleção de um *node* para provisionar o contêiner. O algoritmo considera cinco métricas chaves:

1. Uso de CPU de cada *node*;
2. Uso de memória de cada *node*;
3. Tempo da transmissão da imagem do contêiner na internet;
4. Associação entre os contêineres e os *nodes*;
5. Agrupamento de contêineres.

Todas essas métricas foram consideradas, pois, de acordo com os autores, afetam no desempenho das aplicações que estão sendo executadas pelos contêineres. A função objetivo de escalonamento é a otimização da composição de todas as métricas chaves, ou seja, para cada métrica chave é relacionado um escore, esses escores são agrupados em uma função de composição que representa a função objetivo.

Em Menouer e Darmon (2019) foi desenvolvido uma nova estratégia de escalonamento baseado em um algoritmo de decisão multi-critério. A abordagem consiste em escalonar contêineres baseado em três critérios que estão relacionados a todos os *nodes*

que constitui a infraestrutura de nuvem: (1) O número de contêineres em execução; (2) A disponibilidade de CPU; (3) A disponibilidade do espaço de memória.

Em Menouer et al. (2019) os autores utilizaram técnicas refinadas de *machine learning* em um ambiente de nuvem para construção de um escalonador de contêineres. O objetivo dessa abordagem é reduzir o consumo energético de infraestruturas de nuvens heterogêneas. O principio corresponde a dois passos denominados (1) aprender e (2) escalonar, que são aplicados a cada novo contêiner que é submetido à plataforma. No passo (1) é estimado o consumo energético de cada *node*, logo, é elencado grupos de *nodes* que formam uma estrutura de nuvem heterogênea em um *cluster* de acordo com o seu consumo energético. Já no passo (2), é selecionado o *node* que corresponde ao menor consumo energético. O algoritmo foi implementado para a plataforma *docker swarm*.

2.5.3 Considerações parciais

Está seção elencou alguns do trabalhos encontrados na literatura acerca do tema escalonamento de contêineres. Nota-se que os trabalhos envolvidos possuem definição de parâmetros e métricas de otimização, como por exemplo: consumo energético, desempenho. Por se tratar de um recorte grande na literatura, o tema escalonamento abre espaço para implementação de diferentes soluções para o problema. Todavia, os trabalhos relacionados aqui elencados não consideram o fator escalabilidade, não há nenhuma solução distribuída. A escassez de projetos de escalonamento de contêineres que envolvam o desenvolvimento de uma arquitetura distribuída, é uma das principais motivações para o presente trabalho.

3 KMS - *Kubernetes Micro Scheduler*

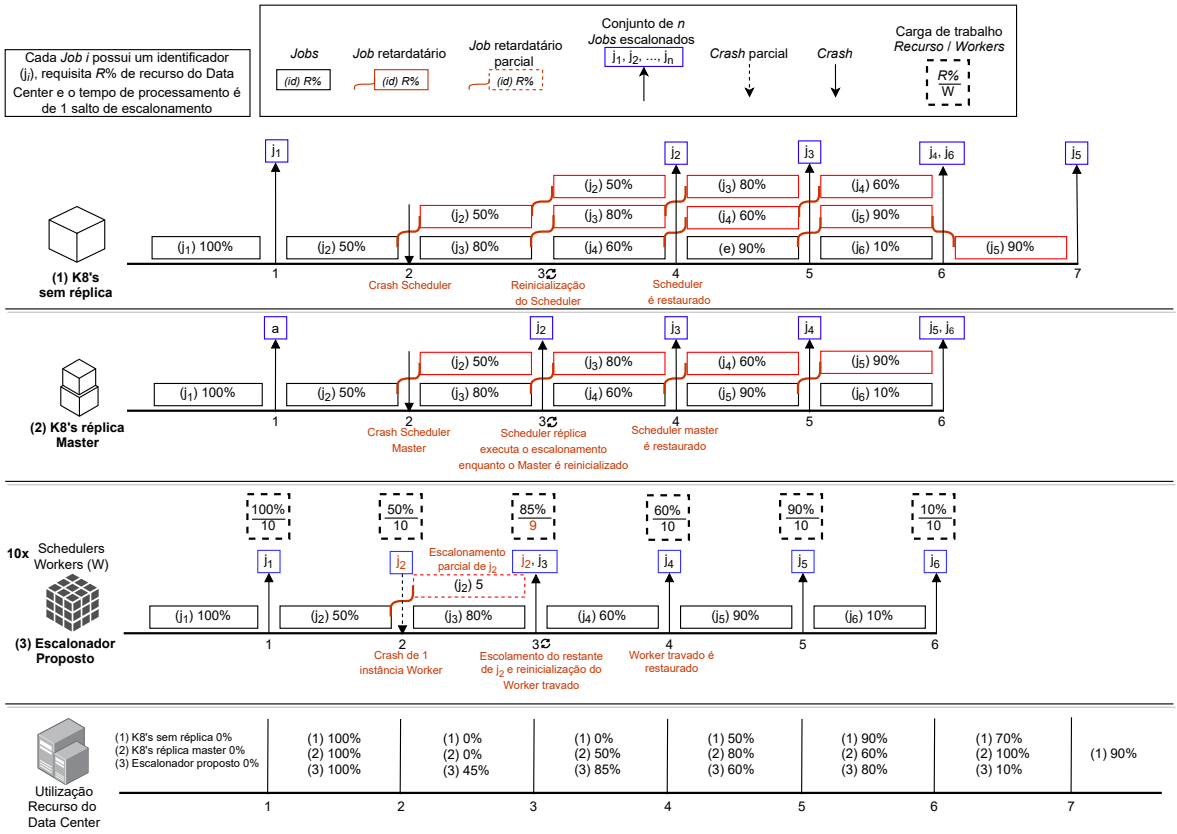
Este capítulo tem como objetivo apresentar os processos de desenvolvimento do escalonador distribuído proposto, denominado *KMS*. A escolha do nome está concentrada na etimologia da palavra **micro**, a qual significa **pequeno**, visto que um dos principais objetivos do presente trabalho é distribuir o escalonador utilizando conceitos de **micro**sserviços. Ou seja, fragmentar um escalonador monolítico em pequenos sistemas que operam de forma independentes, mas em conjunto refletem um sistema distribuído robusto.

No início do capítulo é apresentado um comparativo entre o *KMS* com as atuais abordagens de escalonamento do *Kubernetes*, o principal objetivo nesta seção é confrontar as arquiteturas em cenários de falhas. Ao passo que, as seções subsequentes são dedicadas ao funcionamento dos componentes do *KMS*, como também as trocas de mensagens. Exceto a Seção 3.2, que é destinada à apresentar a documentação baseada na engenharia de requisitos para delimitar o escopo do projeto.

3.1 Comparativo com abordagem padrão do *Kubernetes*

A visualização do desempenho do escalonador proposto distribuído é notável ao ser comparado com as demais abordagens de escalonamento que o *Kubernetes* oferece (com e sem réplica do *node master)*, como esboça a Figura 4.

Figura 4 – Comparação das abordagens de escalonamento



Fonte: O autor

A Figura 4 representa o comportamento de três métodos de escalonamento de contêineres: Escalonador padrão *Kubernetes* com *(1)* e sem *(2)* réplica e o escalonador distribuído proposto *(3)* em um período de tempo dividido em eventos de escalonamento. No primeiro evento de escalonamento há uma solicitação de 100% de uso dos recursos, e em todas as abordagens o escalonamento é executado com sucesso, pois os recursos estavam disponíveis e não ocorreram falhas. No segundo evento há uma nova solicitação que consome 50% dos recursos, entretanto, houve a simulação de uma falha do tipo *crash*. Por conta disso, a abordagem *(1)* utilizará um evento de escalonamento para reinicialização do sistema e a abordagem *(2)* redirecionará as requisições para a réplica de escalonamento. Diferente disso, o escalonador distribuído *(3)* é capaz de executar o escalonamento parcial, dado que a falha ocorreu em apenas uma unidade do sistema distribuído. A principal consequência no comportamento do escalonador *(1)* e *(2)* é no atraso das requisições em efeito cascata. Por consequência, todas as requisições que excedem a capacidade de

recursos são atrasadas para o próximo passo de escalonamento, causando degradamento do tempo de espera e *slowdown*.

3.2 Levantamento de Requisitos

Nessa seção objetiva-se desenvolver uma documentação compreensível do sistema distribuído proposto guiado pela engenharia de requisitos. O levantamento de requisitos é o passo inicial para entendimento do escopo e objetivos do *KMS*, foi desenvolvido utilizando a técnica de análise de cenário e objetivos desejados. No contexto do presente trabalho, buscou-se elencar os requisitos funcionais a partir do comportamento do sistema distribuído e suas funcionalidades, já os requisitos não funcionais aqueles que representam as propriedades e restrições desejadas.

Requisitos funcionais

1. Escalonar cargas de trabalho utilizando arquitetura hierárquica de sistemas distribuídos, denominada *master/slave*;
2. Segmentar o *cluster*, cada segmento será gerenciado por um escalonador dedicado denominado *worker*;
3. *Workers* serão responsáveis por escalonar cargas de trabalhos nos segmentos do *cluster*;
4. Implementação de estratégias de escalonamento conhecidas na literatura, como por exemplo: *Round-Robin*, *Binpack* e *Easy Backfiling*; e
5. Possibilidade de intercambiar o algoritmo de escalonamento.

Requisitos não funcionais

1. O escalonador deverá ser compatível com o *Kubernetes*;
2. Executar os microsserviços virtualizados em contêineres;
3. O sistema não deve conter único ponto de falha, todos os componentes deverão serem distribuídos;
4. Desenvolver os principais componentes (*Master* e *Worker*) no formato de microsserviços *stateless*.
5. Desenvolver um sistema distribuído de forma transparente, ou seja, promover acesso aos recursos distribuídos de forma oculta, como se fosse um único sistema para o usuário; e

6. Utilizar a estratégia de múltiplos escalonadores, como discutido na seção 2.2.4.

3.3 Identificação dos Componentes

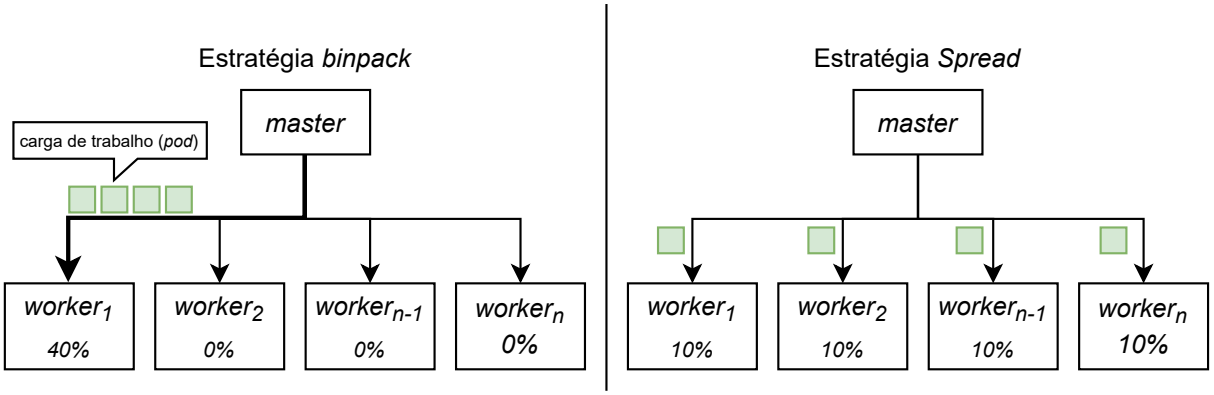
Em síntese, o *KMS* consiste em um modelo hierárquico de sistema distribuído, possuí 2 componentes principais: *Master* e *Worker*. O componente *Master* corresponde ao único ponto de centralização, isso não significa que haverá um único ponto de falha no sistema, mas sim que este componente manterá apenas uma instância ativa executando o trabalho dentre todas as suas réplicas. Ao fazer uma analogia com a arquitetura Produtor/Consumidor, o componente *Worker* corresponde ao Consumidor, pois é responsável por executar as ações de escalonamento do *KMS*, enquanto que o *Master* é interpretado como Produtor, em razão de lidar com a delegação de trabalho.

3.3.1 *Master*

Responsável por buscar cargas de trabalho (*pods*) na fila de escalonamento do *Kubernetes*, em seguida distribuir as cargas para os *Workers*. Ou seja, é um componente relacionado com a delegação de trabalho, considerado o **Produtor** em um sistema distribuído.

Um dos princípios do desenvolvimento, não só do componente *Master*, mas sim de todo o sistema, é implementar uma arquitetura modular a qual seja possível intercambiar entre as técnicas de escalonamentos. O primeiro ponto de escalonamento é encontrado na distribuição das cargas de trabalho do *Master* para os *Workers*. Por exemplo, duas estratégias são observadas na Figura 5.

Figura 5 – Estratégias de distribuição de cargas entre *master* e *workers*



Fonte: O autor

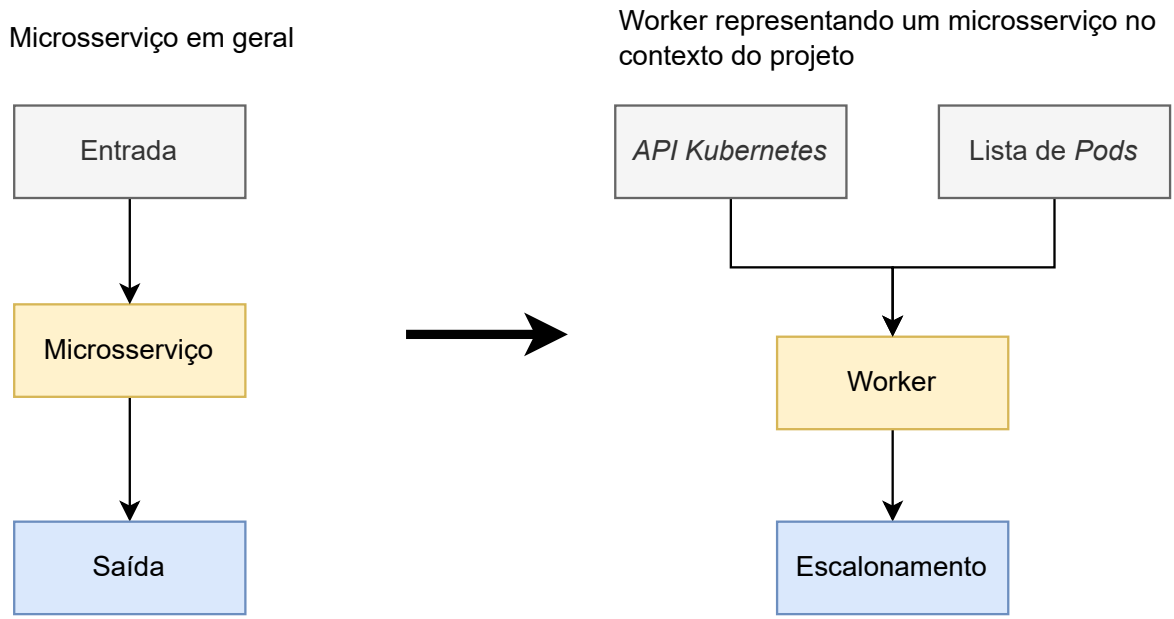
Na Figura 5 há a implementação de duas estratégias de escalonamento: *Binpack* e *Spread*. *Binpack* está relacionado com a minimização dos recursos, ou seja, o algoritmo selecionará o mesmo *Worker* enquanto existirem recursos nessa unidade computacional -

na imagem o $worker_1$ está apenas com 40% de utilização, por isso todos os *pods* estão sendo direcionados a ele. Já o *Spread* possui como objetivo otimizar a utilização de recursos, comumente utilizado *round-robin*, é possível notar que as cargas estão sendo distribuídas de forma igualitária entre os *workers*.

3.3.2 *Worker*

O *Worker* tem como objetivo resolver o escalonamento, ou seja, encontrar um *node* específico para executar o *pod*. A principal característica é ser um componente *stateless* - não armazena estado. Isso é possível, uma vez que foi desenvolvido no formato de microsserviço: recebe uma entrada, executa regras no escopo fechado da entrada e gera uma saída. Dessa forma, a entrada do *Worker* consiste no estado atual do *Cluster* e na lista de *Pods*, que são informados, respectivamente, pela *API* do *Kubernetes* e pelo componente *Master* do *KMS*. Após ler a entrada são executadas operações em relação ao escalonamento, por fim, a saída representa o resultado do escalonamento. A Figura 6 esboça o comparativo entre entre arquitetura de microsserviço genérica com o componente *Worker*.

Figura 6 – Comparativo Microsserviço x *Worker*

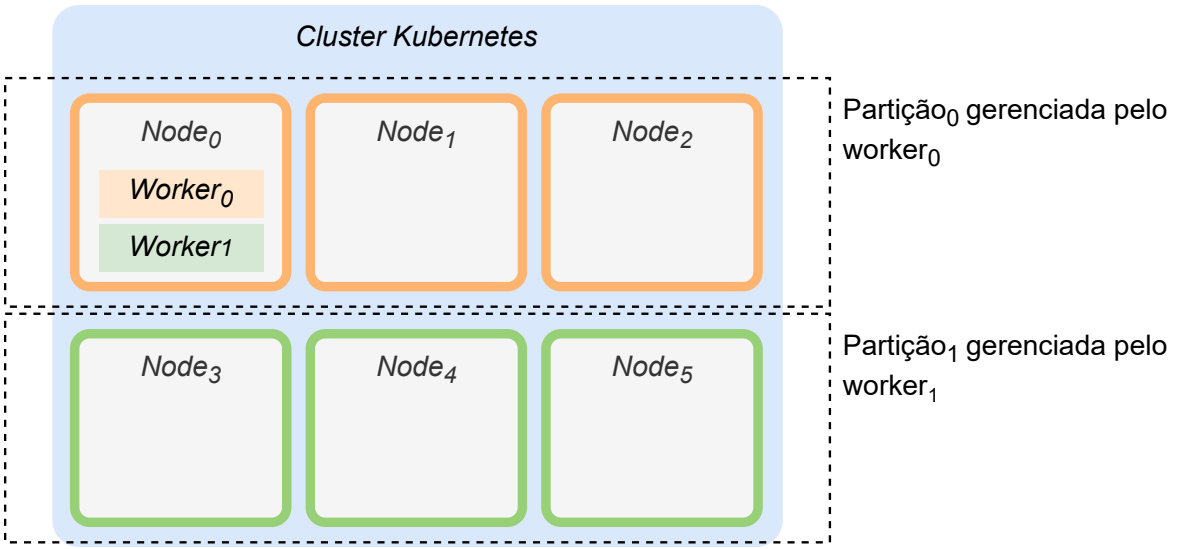


Fonte: O autor

No *KMS*, o *Worker* é um componente com réplicas, e todas ativas. A ideia é que cada instância gerencie uma partição do *cluster Kubernetes*. Por exemplo, considere um *cluster* com n nós computacionais e w *Workers*, nesse contexto, cada instância do *Worker* será responsável por m/w nós do *cluster*. Isto é, a instância executará o escalonamento no conjunto de nós que pertencem a sua partição.

Ao executar o *KMS* em um *cluster* com 6 nós computacionais e 2 instâncias do componente *Worker*, o *cluster* será seccionado em 2 conjuntos de nós de tamanho 3 (*Tamanho Cluster / quantidade réplicas Worker*). Logo, cada instância do *Worker* será responsável por processar o escalonamento da sua respectiva partição, como demonstra a figura 7.

Figura 7 – particionamento do *cluster* entre *workers*



Fonte: O autor

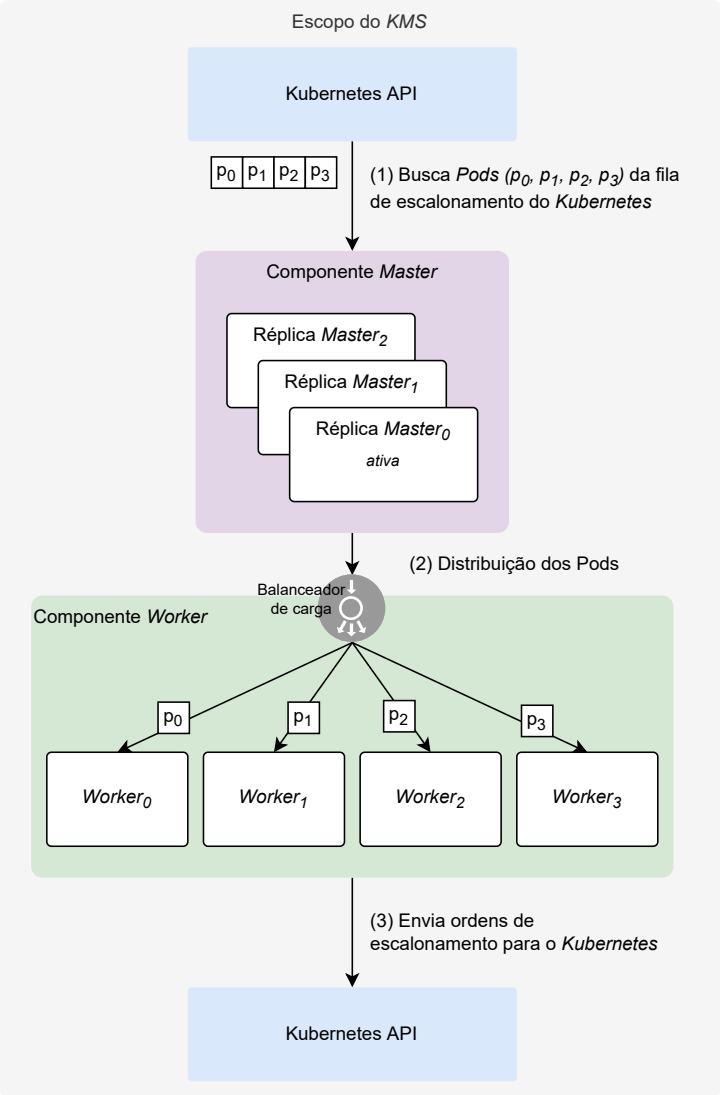
Na Figura anterior, nota-se que tanto o $Worker_0$ quanto o $Worker_1$ residem em $Node_0$. Deve-se ao fato que ambas as instâncias foram orquestradas pelo escalonador padrão do *Kubernetes*, que tomou a decisão, neste exemplo de forma hipotética, de escalonar as duas réplicas no mesmo nó computacional. Todos os componentes internos do *KMS* são gerenciados pelo escalonador padrão do *Kubernetes*.

3.3.3 Relação *Master-Worker*

As subseções anteriores se encarregaram de apresentar os componentes de forma isolada, nesta subseção objetiva-se sintetizar o funcionamento do *KMS* como um todo, aqui o principal objetivo é relacionar os componentes *Master* e *Worker*. Em resumo, o *KMS* consiste em 3 passos objetivos: (1) Buscar os *Pods* na fila de escalonamento interna do *Kubernetes*, (2) Distribuir as cargas de trabalho entre os *Workers* e (3) executar o escalonamento.

Considere um cenário do *KMS* com 3 réplicas *Master* e 4 réplicas *Worker*, o funcionamento do sistema distribuído proposto pode ser visualizado na Figura 8.

Figura 8 – Diagrama de Fluxo



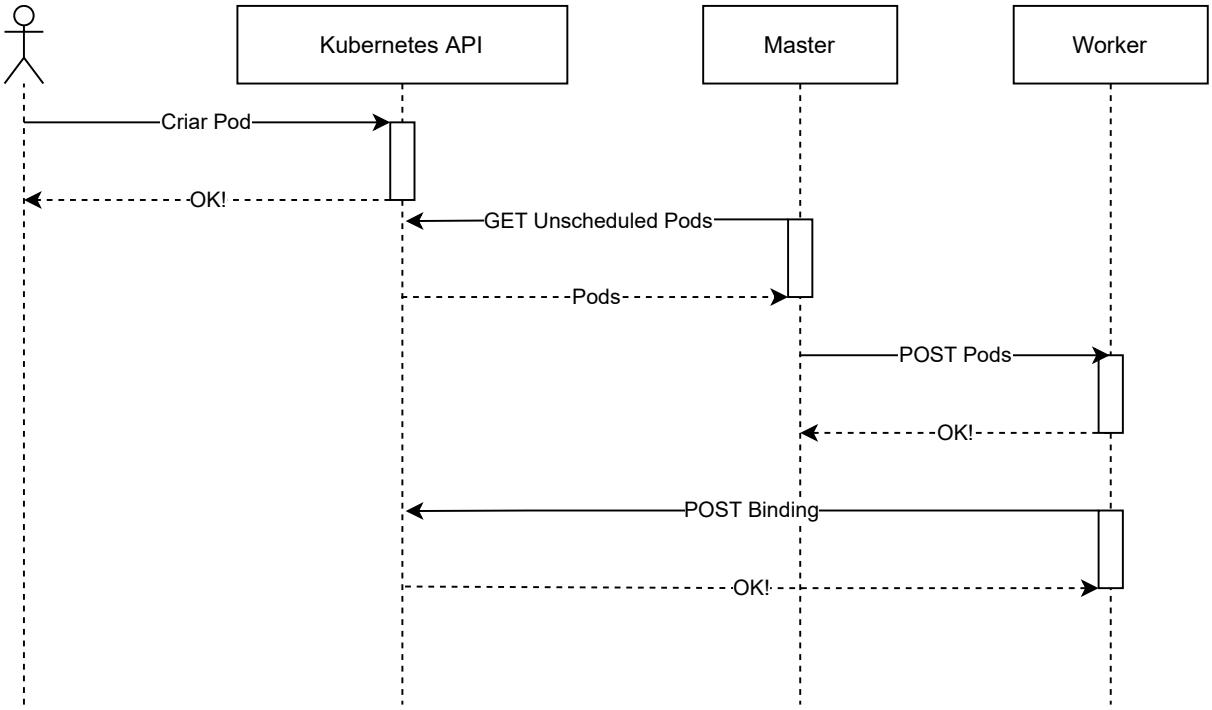
Fonte: O autor

O diagrama anterior apresenta a interação completa entre os módulos do *KMS*, desde a coleta do *Pod* pelo *Master*, até o envio da ordem de escalonamento pelo *Worker* para a *API* do *Kubernetes*. Além disso, é possível verificar que no topo do componente *Worker* há um balanceador de carga, que é responsável por distribuir as requisições de escalonamento de *Pods*. Com isso, o componente *Master* não necessita guardar estado das instâncias do *Worker*, comunicar-se diretamente com o balanceador de carga é o suficiente. No *Kubernetes* o balanceador de carga pode ser configurado tanto para a estratégia *Binpack* quanto para *Round-Robin*, no exemplo da imagem anterior a estratégia executada foi *Round-Robin*. Outro ponto de destaque são as réplicas do componente *Master*, considerado o **Produtor** do sistema distribuído, as instâncias do *Master* necessitam executar algoritmo de eleição para manter apenas uma ativa executando a tarefa de **Produtor**. O algoritmo de eleição será apresentado na Seção 3.5.

3.4 Trocas de Mensagens

A comunicação entre o sistema distribuído proposto e o *Kubernetes* é efetuada pelo protocolo *HTTP*, em resumo, o *Kubernetes* expõe um servidor *Web* que pode ser consumido por qualquer componente interno. No contexto do presente trabalho, o *Master* consumirá a *API* para buscar os contêineres que estão na fila de escalonamento. Na mesma linha de pensamento, o componente *Worker* se comunicará com a *API* para enviar ordens de escalonamento. A comunicação entre o componente *Master* e *Worker* também será efetivada via protocolo *HTTP*, em síntese, cada instância do *Worker* abrirá um servidor *Web* para consumir as cargas de trabalho enviadas pelo *Master*. Ao utilizar essa arquitetura, será possível definir novas rotas para o servidor *Web* do *Worker*, além daquelas relacionadas ao escalonamento, como por exemplo, rotas para verificar sobrecarga de recurso do contêiner. Para elucidar as trocas de mensagens, foi elaborado um diagrama de sequência com os eventos principais do *KMS* na Figura 9.

Figura 9 – Diagrama de Sequência *KMS*



Fonte: O autor

Como todos os módulos do *KMS* se comunicam por *HTTP*, o diagrama de sequência foi elaborado utilizando a nomenclatura padrão de requisições *Web - POST e GET*. O ponto de partida é no momento em que um novo *Pod* é inserido na plataforma pelo usuário, isso deve ser executado obrigatoriamente pela *API* do *Kubernetes*, na figura corresponde ao evento *Criar Pod*. O *Pod* permanecerá na fila de escalonamento interna até algum escalonador requisitar os *Pods* não escalonados, que é observado no evento *GET Unscheduled Pods*. O próximo passo é distribuir os *Pods* para os *Workers* por meio do evento

POST Pods, que é executado pelo *Master*. Ao fim, o *Worker* executa o escalonamento e envia uma ordem do tipo *binding* para a *API* do *Kubernetes*, que vinculará o *Pod* à algum nó eleito pelo *Worker* no processo de escalonamento. Essa última etapa é representada pelo evento *POST binding*.

3.5 Eleição do Componente *Master*

Um dos princípios do *KMS* é desenvolver uma aplicação distribuída sem único ponto de falha, isso influência que todos os módulos do sistema necessitem alguma técnica de controle de falhas. As instâncias do *Worker* são considerados microsserviços e todas as sua réplicas trabalham simultaneamente no escalonamento. Entretanto, o módulo *Master*, mencionado nas seções anteriores como **Produtor** do sistema distribuído, demanda que apenas uma réplica esteja ativa trabalhando no escalonamento enquanto que as outras estarão em espera. Dado o contexto, verificou-se que uma das formas de resolver este problema é utilizar algoritmo de eleição para manter apenas uma instância ativa.

Visando simplificar o desenvolvimento, as réplicas do componente *Master* se comunicarão com o *Redis* para executar o ambiente de eleição. Ao contrário da abordagem padrão de banco de dados relacionais, o *Redis* é considerado não relacional do tipo Chave-Valor em memória. A principal utilização dessa tecnologia é no *caching* de informação devido ao seu rápido acesso aos dados. Além disso, é possível utilizar com persistência em disco, transmissão em tempo real de informação (*Streaming Engine*) e mensagens de inscrição em tópicos do tipo *Publish/Subscribe*, em conclusão, o *Redis* é uma eficiente tecnologia no desenvolvimento de sistemas distribuídos.

O principal objetivo da escolha do *Redis* é se beneficiar da funcionalidade de *Locks Distribuídos*. Os *Locks Distribuídos* são considerados uma primitiva de banco de dados, que auxiliam em cenários onde diferentes processos operam recursos compartilhados de forma mutuamente exclusiva (Redis Documentation, 2022).

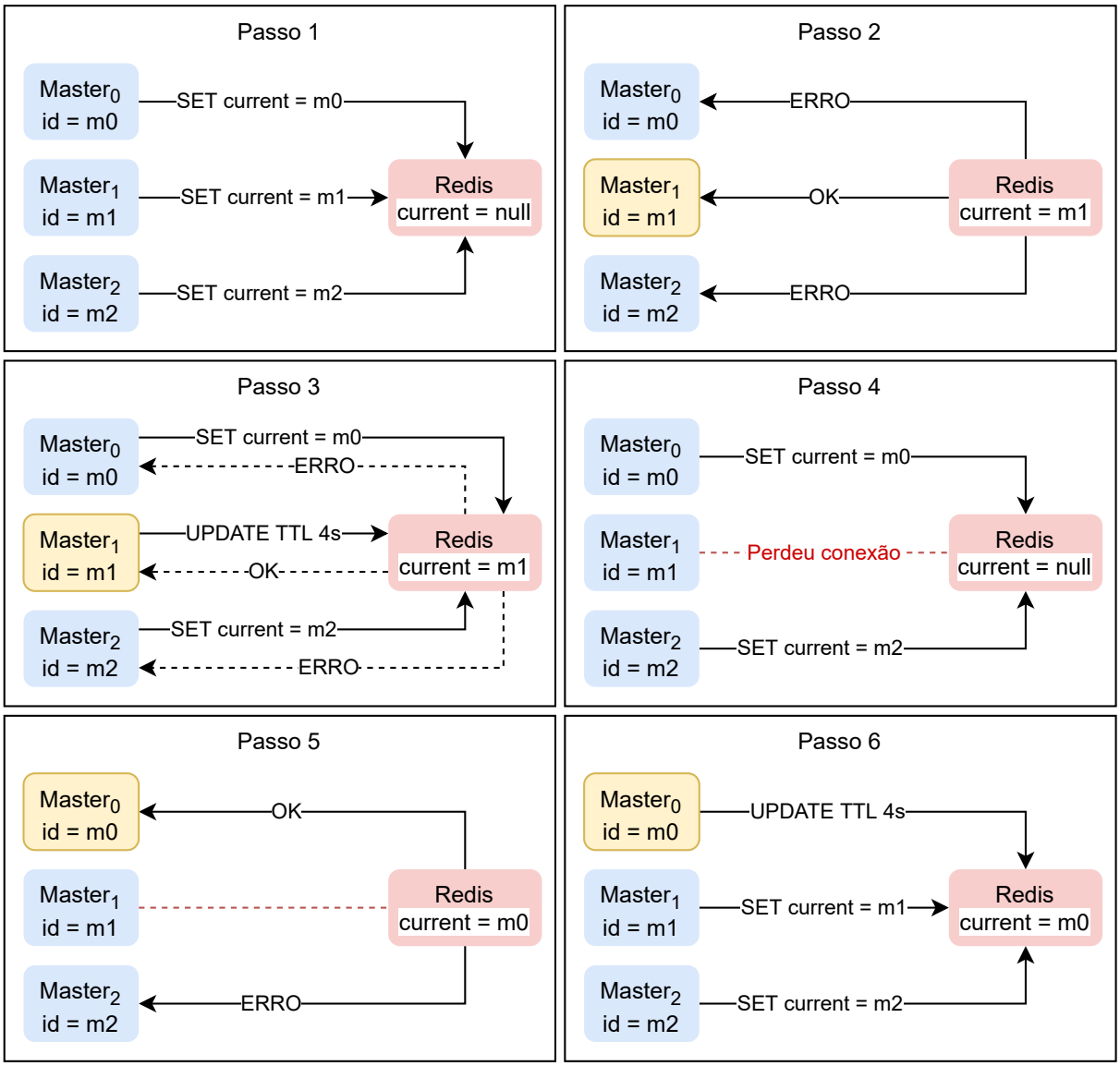
3.5.1 Algoritmo de eleição apoiado no *Redis*

Em síntese, o processo de eleição, baseado em *Locks Distribuídos*, consiste em reproduzir um cenário de corrida entre os processos, o primeiro que conseguir escrever a sua identificação no recurso compartilhado do *Redis* será eleito líder. O recurso compartilhado é no formato de um campo Chave-Valor e possui tempo de expiração (*TTL - Time to live*), o campo voltará a ser nulo no momento em que atingir a data de expiração, dessa forma, habilitando uma nova corrida entre os processos.

Para exemplificar a execução do algoritmo de eleição apoiado no *Redis*, foi desenvolvido um diagrama com 6 passos. Neste exemplo, considere que há 3 réplicas do componente *Master* que estarão disputando a eleição do líder. A réplica que for eleita executará o

escalonamento e as regras do *Master*, enquanto que as restantes permanecerão em espera por uma nova oportunidade de eleição. Os diagramas podem ser vistos na Figura 10.

Figura 10 – Processos de eleição do *Master*



Fonte: O autor

Passo 1: Todos os *Masters* tentarão escrever no campo *current* do Redis o próprio *id*. Por exemplo, $Master_2$ tentará escrever *m2* em *current*. O campo *current* foi configurado para ser compartilhado de forma mutuamente exclusiva e possui tempo de expiração de 4 segundos.

Passo 2: *Redis* vai utilizar *lock* interno e apenas uma instância do *Master* conseguirá escrever o id no campo *current*. Só é possível escrever se o campo current for nulo. Nesse exemplo, $Master_1$ foi premiado e tournou-se líder. Neste passo, o *Redis* responde com sucesso apenas o $Master_1$.

Passo 3: O *Redis* está configurado para expirar o campo *current* a cada 4 segundos, assim permitindo a corrida entre os *Masters*. Ao passo que, o líder, no caso $Master_1$, a cada intervalo de 2 segundos estenderá o *TTL* do campo *current* em 4 segundos.

Passo 4: Considere que o $Master_1$ perdeu a conexão ou foi derrubado. Ou seja, não irá conseguir atualizar o *TTL* do valor *current*. Portanto, em 4 segundos o valor *current* voltará a ser nulo, dessa forma, habilitando uma nova corrida entre os *Masters*.

Passo 5: Como o campo *current* voltou a ser nulo, uma nova corrida é executada. Considere que o $Master_0$ conseguiu escrever *m1* em *current* e será o novo líder e responsável por atualizar o *TTL* enquanto ainda estiver disponível.

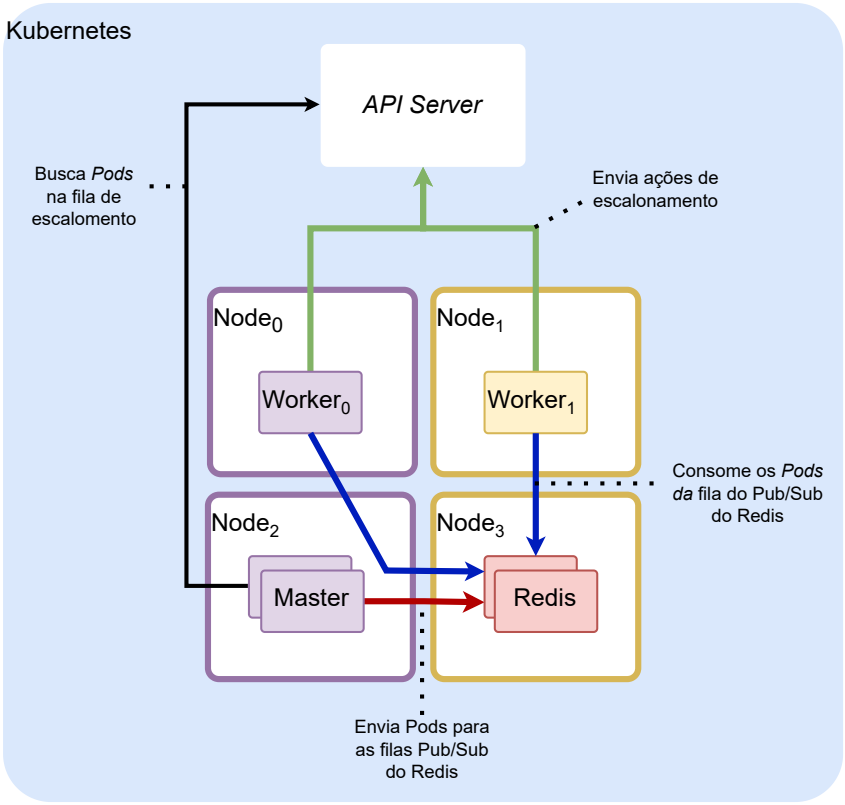
Passo 6: Este passo representa o reinicio do algoritmo, em que o último líder eleito, no caso o $Master_0$, será responsável por atualizar o *TTL* do campo *current*. No momento em que não for possível atualizar, o campo voltará a ser nulo dando início uma nova corrida entre as réplicas.

3.6 Implantação em *Kubernetes*

A *KMS* é executada no topo do *Kubernetes*, dessa forma, os componentes principais são conteinerizados e provisionados pelo próprio *Kubernetes*. Na Seção 2.2.4 foram apresentadas 4 formas distintas de customizar o escalonador padrão, após analisar a viabilidade, chegou-se a conclusão que o método **Múltiplos escalonadores** é o ideal para a implementação do projeto. Essa abordagem consiste no desenvolvimento de um escalonador que é executado no formato de *Pod* e toda a comunicação com o *Kubernetes* é realizada via troca de mensagens por meio da *API*. Com isso, o componente *Worker* é considerado *stateless*, pois as suas dependências (estado do *cluster* e a lista de *Pods* não escalonados) são considerados como parâmetros. Neste contexto, o problema de escalonamento, na visão do *Worker*, é da forma entrada e saída: a entrada é o estado do *cluster* que é informado pela *API*, a saída é a ação de escalonamento efetuada pelo *Worker*. Portanto, o escalonador proposto será executado ao lado do escalonador padrão e representado por um conjunto de *pods*. A Figura 11 demonstra um caso de uso com 2 instâncias de *Workers* que coordenam o escalonamento de 2 *Nodes* do *Kubernetes*, o $Worker_0$ é responsável pelo $Node_0$ e $Node_2$ (cor roxa) já o $Worker_1$ pelo $Node_1$ e $Node_3$ (cor amarela).

Na figura percebe-se que as instâncias do componente *Master* residem no $Node_2$, isso se deve ao fato que as instâncias dos componentes do *KMS* são conteinerizados e escalonadas pelo escalonador padrão do *Kubernetes*. Logo, não é tarefa do escalonador proposto pré definir os nodes em que as próprias instâncias serão executadas, isso é tarefa do *Kubernetes* por meio do escalonador padrão.

Figura 11 – Exemplo de implantação.



Fonte: O autor

Ao utilizar o método de múltiplos escalonadores, no momento de provisionar um novo *pod* para a plataforma é necessário discriminar o escalonador desejado, que é possível por meio do atributo *schedulerName*. Isso é alterado facilmente no arquivo de manifesto do *pod*, como ilustra a Figura 12.

Figura 12 – Direcionamento do *pod* para o escalonador *my-scheduler*

Definição *pod*

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: pod-with-second-annotation-container
    image: k8s.gcr.io/pause:2.0
```

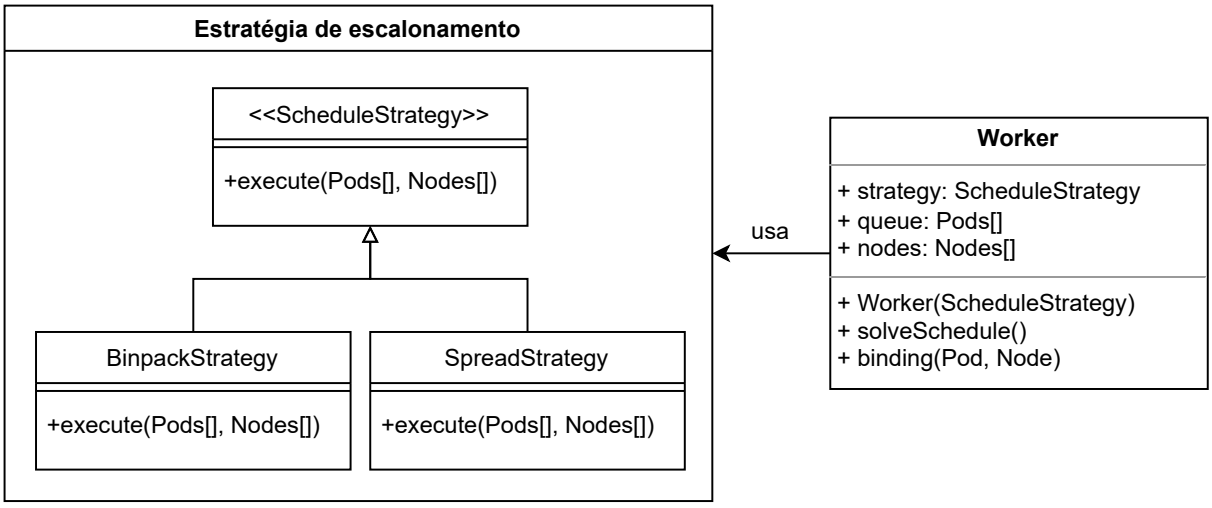
Fonte: O autor

3.7 Representação por Diagramas de Classes

O diagrama de classes da proposta é visualizado pelas figuras 13 e 14. No diagrama do *Worker* é utilizado o padrão de projeto *strategy* para alternar o método de escalonamento, além disso, há também outros dois métodos – *solve* e *binding*. *Solve* é responsável por resolver o escalonamento de acordo com a estratégia escolhida, e *binding* tem como objetivo enviar uma ordem de escalonamento para *Kubernetes* utilizando a *API*.

3.7.1 Worker

Figura 13 – Diagrama de Classes *worker*



Fonte: O autor

Interface *ScheduleStrategy*: Implementação do padrão de projeto *Strategy*, permite intercambiar o método de escalonamento.

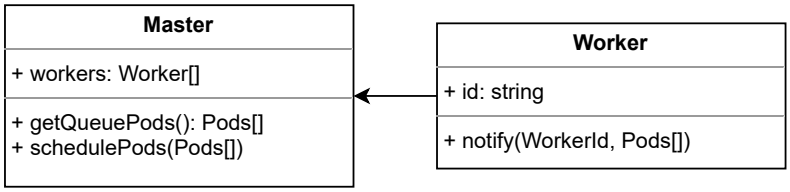
Classe *BinpackStrategy*: Executa o escalonamento utilizando técnica de agrupamento.

Classe *SpreadStrategy*: Executa o escalonamento utilizando técnica de espalhamento.

Classe *Worker*: Executa o escalonamento a partir da estratégia selecionada. A estratégia é escolhida no método de construção da classe *Worker(ScheduleStrategy)*. Também há a especificação da fila de escalonamento representado pelo atributo *queue* e dos nós – *nodes* – que o *Worker* está gerenciando. O método *solveSchedule* executa o escalonamento a partir da estratégia selecionada e *binding* envia ordem de escalonamento para *API* do *Kubernetes*.

3.7.2 Master

Figura 14 – Diagrama de Classes *Master*



Fonte: O autor

Classe *Worker*: Representa a entidade *Worker* no escopo do *Master*. Em resumo, consiste em um atributo para identificação – *id* – e um método para a comunicação de envio dos *Pod* denominado *notify*.

Classe *Master*: Possui uma lista de *Workers* vinculado, é responsável por buscar os *Pods* que estão na fila de escalonamento do *Kubernetes* – *getQueuePods*. Já o método *schedulePods* objetiva-se executar o particionamento e direcionar cada parte da fila de escalonamento para um *Worker* específico.

3.8 Considerações parciais

A proposta aqui elucidada visa o projeto de uma arquitetura distribuída, fundamentada em microsserviços, para a plataforma *Kubernetes*. Isso caracteriza um projeto multidisciplinar, pois fundamentos de engenharia de software são essenciais para levantamento de requisitos, documentação em *UML* e na identificação e construção dos microsserviços. Além disso, o desenvolvimento de sistemas hierárquicos está relacionado com a grande área de sistemas distribuídos, conceitos esses, essenciais na construção da arquitetura replicável dos módulos *master-worker*. Classifica-se também um projeto em redes de computadores, pois todas as interações entre os serviços se dão via comunicação, seja entre os microsserviços utilizando o protocolo *gRPC* e até mesmo com o próprio *Kubernetes* que é utilizado *HTTP*.

4 Análise Experimental

Nesta seção objetiva-se definir os detalhes de infraestrutura do ambiente de testes, isso inclui as versões dos sistemas que serão utilizados, tanto o *Kubernetes* quanto sistema operacional das unidades computacionais. Outro ponto a salientar é a definição dos parâmetros e as métricas de interesse, como também a carga de trabalho que irá estressar a arquitetura no ambiente de testes.

4.1 Métricas e Parâmetros

Um dos principais desafios do presente trabalho é a investigação e definição dos parâmetros da arquitetura proposta. A princípio, foram analisados dois parâmetros: (1) variação no número de réplicas do componente *Worker* e (2) quantidade de *nodes* que cada *Worker* gerenciará o escalonamento. Os parâmetros são diretamente proporcionais, pois ao variar a quantidade de réplicas será também variado a número de *nodes* vinculado a cada *Worker*. Por exemplo, considere um *cluster* com 8 *nodes* e 2 réplicas *workers*, cada réplica será responsável por executar o escalonamento em uma partição do *cluster* com 4 *nodes*. O cálculo feito para o tamanho de cada partição é a razão entre a quantidade de *nodes* e réplicas do *Worker*.

As métricas de interesse do presente trabalho estão relacionadas com o desempenho de escalonamento em cenários de falhas. Quanto menos tempo uma carga de trabalho permanecer na fila, maior é o desempenho da proposta de escalonamento. Portanto, há duas métricas que serão avaliadas de acordos com os parâmetros propostos: tempo de espera de escalonamento e *makespan*. As métricas de sobrecarga também serão importantes, principalmente relacionadas a sobrecarga do *node*, seja de espaço ou processamento.

4.2 Cenário experimental

Para o ambiente de testes será utilizada a Nuvem da UDESC, com isso, os *nodes* refletirão em máquinas virtuais gerenciadas pelo *Openstack*. Quanto ao sistema operacional não há muita influência no decorrer do presente trabalho, pois o único requisito do *Kubernetes* na versão *v1.23* é que o *node* seja executado em ambiente *linux*, portanto, o *SO* escolhido será *ubuntu cloud* que é projetado para nuvem. A princípio, os *nodes* são máquinas virtuais com recurso limitado em 4 núcleos e 4 gigas de memória *RAM*. No total, o ambiente experimental possui 16 núcleos e 16 gigas de *RAM*.

A configuração padrão do *KMS* no ambiente de teste irá contar com 2 réplicas

do componente *Master* e também 2 réplicas *Worker*. O objetivo do trabalho é analisar o desempenho em cenário de falhas de escalonamento, portanto, o baixo número de réplicas não influenciou nos resultados. Pois os testes consistiram em derrubar de forma parcial algum componente principal do *KMS*. Além das réplicas dos componentes internos, o banco de dados em memória *Redis* também será distribuído e contará com alta disponibilidade. Portanto, todos os módulos do sistema estão protegidos por alguma técnica de controle de falhas.

4.3 Protocolo Experimental

O protocolo experimental é considerado um simulador de eventos discretos, possui como parâmetro o total de cargas de trabalhos que serão submetidas à plataforma e o tempo total de execução. As cargas serão submetidas de forma gradual respeitando a curva da distribuição normal, ou seja, o pico de submissão será exatamente na metade do tempo total de execução. Além disso, haverá também um simulador de eventos de erro, em síntese, consiste em simular erros de *crash* nos sistemas de escalonamentos derrubando os processos. Os eventos de erro também serão disparados respeitando a curva da distribuição normal, isto é, o pico de disparo de erros será na metade do tempo total de execução da bateria de testes.

4.4 Resultados

5 Conclusão

Escalonamento é considerado um processo de tomada de decisão, em resumo, no contexto do presente trabalho, o objetivo é alocar contêineres em nós computacionais denominados *nodes* com recursos finito. Há diversos parâmetros a ser considerados, por consequência, é possível alcançar as mais divergentes funções objetivos – reduzir consumo energético, otimizar utilização de recursos do *data center*. Logo, de maneira geral, os algoritmos de otimização de escalonamento se enquadram na classe *NP-difícil* (ULLMAN, 1975), visto que, a natureza do problema de escalonamento é combinatória.

A explosão da demanda de poder computacional dos *data centers* uniram, ainda mais, as áreas entre escalonamento e sistemas distribuídos. O objetivo desse trabalho, é, a partir da intersecção entre as duas áreas, resolver de forma elegante o problema de escalonamento para *data centers* de larga escala. De acordo com a literatura, *data centers* de larga escala necessitam de componentes internos sofisticados e adaptáveis que lidam com o gerenciamento de centenas de milhares de cargas de trabalho. Assim, a construção de uma arquitetura de escalonamento distribuída, até o momento, é a solução para que as empresas, como *Google* e *Microsoft*, tratem um grande volume de requisições de escalonamento (WANG et al., 2019; VERMA et al., 2015).

Ao unir as áreas entre sistemas distribuídos e escalonamento encontra-se quantidade significativa de estudos explorados, entretanto, a intersecção entre essas áreas junto com orquestradores de contêineres é escasso na literatura como visto no Capítulo 3. Este trabalho possui como objetivo suprir essa carência, uma vez que a principal arquitetura de escalonamento encontrada em *data centers* de larga escala, atualmente, é distribuída. Para validação da proposta, os resultados da arquitetura aqui desenvolvida serão comparados com o escalonador padrão do *Kubernetes*. Com isso, objetiva-se chegar no resultado positivo, cujo as métricas de tempo de espera de escalonamento e *makespan* sejam reduzidas. Além disso, haverá um estudo relacionado ao comportamento da métrica de *slowdown* nos dois cenários propostos: escalonamento padrão e distribuído.

5.1 Trabalhos futuros

A seguir é exposto as etapas de desenvolvimento do projeto, considerando em **verde** as etapas que já foram concluída no período esperado e em **preto** as etapas a ser desenvolvidas.

1. Revisão bibliográfica sobre contêineres e escalonamento;
2. Estudo sobre orquestração de contêineres;

3. Elaboração teórica do algoritmo de escalonamento proposto;
4. Implementação do algoritmo desenvolvido no gerenciador de contêiner;
5. Redação TCC-I;
6. Ajustes de implementação;
7. Desenvolver definições de análise;
8. Análise e comparação dos resultados; e
9. Redação TCC-II.

Tabela 1 – Calendário cronograma

Etapas	2021/2						2022/1					
	O	N	D	J	F	M	A	M	J	J	A	S
1												
2												
3												
4												
5												
6												
7												
8												
9												

Referências

ACETO, G. et al. Cloud monitoring: A survey. *Computer Networks*, Elsevier, v. 57, n. 9, p. 2093–2115, 2013. Citado na página 14.

ARUNDEL, J. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*. [S.l.]: O'Reilly Media, 2019. Citado 2 vezes nas páginas 10 e 16.

ASSUNÇÃO, M. D. de; VEITH, A. da S.; BUYYA, R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, Elsevier BV, v. 103, p. 1–17, fev. 2018. Disponível em: <https://doi.org/10.1016/j.jnca.2017.12.001>. Citado 2 vezes nas páginas 9 e 14.

BERG, T.; CRAMP, A.; SIEGEL, B. Guidelines and best practices for using docker in support of hla federations. SISO-Simulation Interoperability Standards Organization, 2016. Citado na página 15.

Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 2014. Citado na página 13.

BRUCKER, P. et al. Resource-constrained project scheduling: Notation, classification, models, and methods. *European journal of operational research*, Elsevier, v. 112, n. 1, p. 3–41, 1999. Citado na página 23.

CARASTAN-SANTOS, D. et al. One can only gain by replacing EASY backfilling: A simple scheduling policies case study. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019. Disponível em: <https://doi.org/10.1109/ccgrid.2019.00010>. Citado 3 vezes nas páginas 10, 19 e 25.

CASALICCHIO, E.; IANNUCCI, S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, p. e5668, 2020. Citado 2 vezes nas páginas 15 e 16.

DDD Community. *What is DDD*. 2007. <https://www.dddcommunity.org/learning-ddd/what_is_ddd/>, 28 mar. 2007. Citado na página 22.

EVANS, E. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. [S.l.]: Dog Ear Publishing, 2014. Citado na página 22.

FAZIO, M. et al. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, Institute of Electrical and Electronics Engineers (IEEE), v. 3, n. 5, p. 81–88, set. 2016. Disponível em: <https://doi.org/10.1109/mcc.2016.112>. Citado na página 9.

FEITELSON, D. G. *Metrics and Benchmarking for Parallel Job Scheduling*. 1998. Citado na página 24.

FOWLER MARTIN E LEWIS, J. Microservices. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Citado 3 vezes nas páginas 9, 21 e 22.

FRITZSCH, J. et al. From monolith to microservices: A classification of refactoring approaches. In: BRUEL, J.-M.; MAZZARA, M.; MEYER, B. (Ed.). *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Cham: Springer International Publishing, 2019. p. 128–141. ISBN 978-3-030-06019-0. Citado na página 9.

GOOGLE KUBERNETES. *Access Clusters Using the Kubernetes API*. 2019. <https://kubernetes.io/docs/tasks/administer-cluster/access-cluster-api/>, 21 ago. 2019. Citado 2 vezes nas páginas 10 e 17.

GOOGLE KUBERNETES. *Production-grade container orchestration*. 2019. <https://www.kubernetes.io>, 22 ago. 2019. Citado 2 vezes nas páginas 10 e 16.

GOOGLE KUBERNETES. *Kubernetes Scheduler*. 2020. <https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/>, 7 mar. 2020. Citado 2 vezes nas páginas 10 e 19.

GOOGLE KUBERNETES. *Set up High-Availability Kubernetes Masters*. 2020. <https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/>, 7 mar. 2020. Citado na página 20.

Jason McGee. *The 6 steps of the container lifecycle*. 2016. <https://www.ibm.com/blogs/cloud-computing/2016/02/08/the-6-steps-of-the-container-lifecycle/>, 5 mar. 2020. Citado na página 15.

KRAUTER, K.; BUYYA, R.; MAHESWARAN, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, Wiley Online Library, v. 32, n. 2, p. 135–164, 2002. Citado na página 24.

Kubernetes Documentation. *Kubernetes Components*. 2019. <www.kubernetes.io/docs/concepts/overview/components/>, 26 nov. 2021. Citado na página 18.

Kubernetes Documentation. *Kubernetes Components*. 2020. <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework>, 1 fev. 2022. Citado na página 20.

KUBIAK, W. Completion time variance minimization on a single machine is difficult. *Operations Research Letters*, Elsevier BV, v. 14, n. 1, p. 49–59, ago. 1993. Disponível em: <https://doi.org/10.1016/0167-6377(93)90019-d>. Citado na página 25.

LEWIS, J. Microservices-java, the unix way. In: *Proceedings of the 33rd Degree Conference for Java Masters*. [S.l.: s.n.], 2012. Citado na página 21.

LIU, B. et al. A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, Springer, v. 22, n. 23, p. 7741–7752, 2018. Citado na página 27.

LOCH, W. J. et al. A novel blockchain protocol for selecting microservices providers and auditing contracts. *Journal of Systems and Software*, Elsevier, p. 111030, 2021. Citado na página 26.

LU, G.; ZENG, W. H. Cloud computing survey. In: TRANS TECH PUBL. *Applied Mechanics and Materials*. [S.l.], 2014. v. 530, p. 650–661. Citado na página 14.

MACCIO, V. J.; HOGG, J.; DOWN, D. G. On slowdown variance as a measure of fairness. *Operations Research Perspectives*, Elsevier BV, v. 5, p. 133–144, 2018. Disponível em: <https://doi.org/10.1016/j.orp.2018.05.001>. Citado na página 25.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National, 2011. Citado na página 13.

MENOUER, T.; DARMON, P. New scheduling strategy based on multi-criteria decision algorithm. In: IEEE. *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. [S.l.], 2019. p. 101–107. Citado na página 27.

MENOUER, T. et al. Power efficiency containers scheduling approach based on machine learning technique for cloud computing environment. In: SPRINGER. *International Symposium on Pervasive Systems, Algorithms and Networks*. [S.l.], 2019. p. 193–206. Citado na página 28.

MERSON, P.; YODER, J. Modeling microservices with ddd. In: IEEE. *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. [S.l.], 2020. p. 7–8. Citado na página 23.

NESI, L. L. et al. Tackling virtual infrastructure allocation in cloud data centers: a gpu-accelerated framework. In: *2018 14th International Conference on Network and Service Management (CNSM)*. [S.l.: s.n.], 2018. p. 191–197. Citado na página 26.

NEWMAN, S. *Building microservices: designing fine-grained systems*. [S.l.]: "O'Reilly Media, Inc.", 2015. Citado na página 22.

PINEDO, M. *Scheduling*. [S.l.]: Springer, 2012. v. 29. Citado 3 vezes nas páginas 23, 24 e 25.

RAYMOND, E. S. *The Art of UNIX Programming (The Addison-Wesley Professional Computng Series)*. Addison-Wesley, 2003. ISBN 0131429019. Disponível em: <https://www.xarg.org/ref/a/0131429019/>. Citado na página 21.

RED HAT. *What is container Linux?* 2019. <https://www.redhat.com/pt-br/topics/containers/whats-a-linux-container>, 15 abr. 2020. Citado na página 14.

RED HAT. *What is DOCKER?* 2019. <https://www.redhat.com/en/topics/containers/what-is-docker>, 19 ago. 2019. Citado na página 9.

RED HAT. *What's the difference between cloud and virtualization?* 2020. <redhat.com/pt-br/topics/cloud-computing/cloud-vs-virtualization>. Citado na página 13.

Redis Documentation. *Distributed Locks with Redis*. 2022. <https://redis.io/docs/reference/patterns/distributed-locks/>. Citado na página 37.

RODRIGUEZ, M. A. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, Wiley, v. 49, n. 5, p. 698–719, nov. 2018. Disponível em: <https://doi.org/10.1002/spe.2660>. Citado na página 9.

SCHEEPERS, M. J. Virtualization and containerization of application infrastructure: A comparison. In: *21st twente student conference on IT*. [S.l.: s.n.], 2014. v. 21. Citado na página 13.

SURESHKUMAR, M.; RAJESH, P. Optimizing the docker container usage based on load scheduling. In: IEEE. *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*. [S.l.], 2017. p. 165–168. Citado na página 27.

THÖNES, J. Microservices. *IEEE software*, IEEE, v. 32, n. 1, p. 116–116, 2015. Citado na página 21.

ULLMAN, J. D. Np-complete scheduling problems. *Journal of Computer and System sciences*, Academic Press, v. 10, n. 3, p. 384–393, 1975. Citado na página 45.

VAUCHER, S. et al. SGX-aware container orchestration for heterogeneous clusters. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. [S.l.: s.n.], 2018. p. 730–741. ISSN 2575-8411. Nenhuma citação no texto.

VERMA, A. et al. Large-scale cluster management at Google with Borg. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France: [s.n.], 2015. Citado 2 vezes nas páginas 26 e 45.

WANG, K. et al. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. *Concurrency and Computation: Practice and Experience*, v. 28, p. 70 – 94, 2016. Citado na página 26.

WANG, Z. et al. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In: *Proceedings of the ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2019. (SoCC '19), p. 246–258. ISBN 9781450369732. Disponível em: <https://doi.org/10.1145/3357223.3362728>. Citado 2 vezes nas páginas 26 e 45.

Wei Huang. *Create a custom Kubernetes scheduler*. 2019. <https://developer.ibm.com/technologies/containers/articles/creating-a-custom-kube-scheduler/>, 5 mar. 2020. Citado na página 19.

YE, N. et al. Job scheduling methods for reducing waiting time variance. *Computers & Operations Research*, Elsevier BV, v. 34, n. 10, p. 3069–3083, out. 2007. Disponível em: <https://doi.org/10.1016/j.cor.2005.11.015>. Citado 2 vezes nas páginas 10 e 25.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, Springer Science and Business Media LLC, v. 1, n. 1, p. 7–18, abr. 2010. Disponível em: <https://doi.org/10.1007/s13174-010-0007-6>. Citado na página 13.