Pós-Graduação em Ciência da Computação

MARCOS VINICIUS DE ARAÚJO ANDRADE

MIGRAÇÃO DE MICROSSERVIÇOS CONTAINERIZADOS EM TEMPO DE EXECUÇÃO

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2018

MARCOS VINICIUS DE ARAÚJO ANDRADE

MIGRAÇÃO DE MICROSSERVIÇOS CONTAINERIZADOS EM TEMPO DE EXECUÇÃO

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Sistemas Distribuídos
Orientador: Nelson Souto Rosa

Recife
2018

Catalogação na fonte
Bibliotecária Elaine Freitas CRB 4-1790

A553m Andrade, Marcos Vinicius de Araújo
Migração de microsserviços containerizados em tempo de execução / Marcos Vinicius de Araújo Andrade. – 2018.
58 f.: fig., tab.

Orientador: Nelson Souto Rosa
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação. Recife, 2018.
Inclui referências.

1. Sistemas distribuídos. 2. Microserviços. 3. Docker Swarm. 4. Scheduling. I. Rosa, Nelson Souto (orientador) II. Título.

004.36 CDD (22. ed.) UFPE-MEI 2018-129

Dissertação de Mestrado apresentada por **MARCOS VINICIUS DE ARAÚJO ANDRADE** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**MIGRAÇÃO DE MICROSSERVIÇOS CONTAINERIZADOS EM TEMPO DE EXECUÇÃO**" e aprovada pela Banca Examinadora formada pelos professores:

Djamel Fawzi Hadj Sadok
Centro de Informática/ UFPE

Prof. Robson Wagner Albuquerque de Medeiros
Departamento de Computação / UFRPE

Prof. **Orientador:** Nelson Souto Rosa
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 06 de Setembro de 2018.

Prof. Aluízio Fausto Ribeiro Araújo
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Decido este trabalho à minha família e em especial à minha esposa, mãe e avó, a quem devo eterna gratidão por terem sido porto seguro perante as dificuldades durante esse percurso.

Agradecimentos

Primeiramente, agradeço a Deus por todas os desafios que passei e com sua força e graça me abençoou para que eu pudesse supera-los e aprender com eles. Agradeço a todos os professores e ao orientador dessa dissertação, pelo seu exemplo de dinamismo e trabalho que é a maior lição que um professor pode dar a seu aluno. Gostaria de agradecê-lo por ter acreditado na minha caminhada. A todos os amigos, funcionários, professores e colegas que fazem parte do CIn da Universidade Federal da Pernambuco que contribuíram para a conclusão deste trabalho. A meus familiares que apoiaram e deram suporte a minha vida acadêmica. Em especial minha mãe, minha avó, minha tia e minha esposa.

Resumo

A arquitetura de microsserviços tem sido amplamente adotada para construir sistemas em ambientes de nuvem. Um aspecto crítico sobre aplicações baseadas em microsserviço (μApps) é sua implantação em nós/*hosts* num *cluster* devido a sua dinamicidade e facilidade. Ferramentas de gerenciamento de μApps como Kubernetes e Docker Swarm implantam microsserviços baseando-se em recursos disponíveis nos nós/*hosts* em tempo de implantação. Porém, operações para escalar/replicar microsserviços podem ocasionar, com o passar do tempo, uma queda de desempenho em relação ao que se tinha no momento da implantação. Por exemplo, uma operação para replicar um microsserviço pode sobrecarregar um nó/*host* ou intensificar significativamente a comunicação entre microsserviços implantados em nós/*hosts* diferentes. Essas operações acontecem sob demanda em tempo de execução. Neste trabalho, apresentamos uma solução capaz de alterar a implantação da μApp movendo os microsserviços considerando informações de tempo de execução que representam uma visão mais realista dos recursos disponíveis no *cluster*. Para avaliar a solução proposta, avaliamos o seu desempenho com as estratégias padrões usadas pelas ferramentas de gerenciamento de microsserviços, e em vários cenários diferentes.

Palavras-chaves: Microserviços. Docker Swarm. Kubernetes. *Scheduling*.

Abstract

The Microservice architectural style has been widely adopted for building distributed systems in cloud environments. A critical aspect of microservice-based applications (μApps) is their deployment throughout nodes of a cluster due their dynamism and ease. Commonly adopted management tools like Kubernetes and Docker Swarm deploy μApps based on the resources available in the nodes at deployment time. However, traditional scale in-/out operations of μApps may lead the initial deployment to a lousy performance over time. For example, scale-in/out operation can overload a given node or put high-intensive communicating microservices in different nodes. These operations happen on-demand during runtime. In this work, we present a solution capable of changing the deployment of the μApp by moving the microservices considering runtime information that represents a more realistic view of available resources in the cluster. To assess the proposed solution, we compare the performance of microservices from a μApps using the proposed solution against ones adopting the default strategies available in the mentioned management tools in several different scenarios.

Keywords: Microservices. Docker Swarm. Kubernetes. Scheduling

Lista de ilustrações

Lista de tabelas

Sumário

1 INTRODUÇÃO

Neste capítulo apresentamos a motivação desse trabalho, contextualizando o problema e as deficiências do estado da arte. Por fim, serão estabelecidos os objetivos e as contribuições esperadas.

1.1 Contexto e Motivação

O desenvolvimento e a execução de sistemas são atividades naturalmente complexas. E a complexidade aumenta se o sistema for distribuído (BRESCIANI et al., 2004; PRIKLADNICKI et al., 2003). Além disso, num sistema distribuído é comum que o software seja implantado na nuvem.

Softwares em ambientes de nuvem são normalmente monolíticos. Segundo Dragoni et al. (2017), softwares monolíticos são aplicações cujos módulos não podem ser executados independentemente. Villamizar et al. (2016) menciona que uma aplicação monolítica é um software com um repositório de código grande, com muitos serviços, e que usam diferentes meios para comunicação como HTTP. Sampaio et al. (2018) descreve serviço como uma grande parte de um software, como um serviço de pagamento em comércio online, onde várias funcionalidades são executadas: checar se existe o cliente, autorizar o pagamento e notificar o pagamento.

Sendo assim, um software monolítico é uma aplicação em que cada componente executa dependendo do pleno funcionamento dos demais componentes. Portanto, softwares monolíticos são fortemente acoplados visto que há grande dependência dos serviços providos.

No entanto, as aplicações monolíticas podem ser impactadas na nuvem de muitas maneiras afetando sua execução e seu desempenho. Por exemplo, a necessidade por atualização, uma correção num serviço com *bug* e assim por diante. Isso porque, geralmente, não se pode parar uma aplicação/serviço para, por exemplo, atualizá-la, ou para executar as atividades mencionadas.

O desempenho da aplicação também pode ser comprometido por operações sob demanda, como escalar um serviço. Se a aplicação é monolítica e há um serviço que é popular e precisa ser escalado, todos os serviços serão escalados, e aqueles que não são populares desperdiçarão recursos (VILLAMIZAR et al., 2015). Isso mostra o porquê empresas como Netflix, Facebook, Spotify, Amazon e LinkedIn fazem uso de microsserviços (VILLAMIZAR et al., 2016; VILLAMIZAR et al., 2015; THÖNES, 2015; ALSHUQAYRAN; ALI; EVANS, 2016; JAMSHIDI et al., 2018).

A escolha de microsserviços como estilo arquitetural no desenvolvimento de aplicações distribuídas tem sido uma tendência (JAMSHIDI et al., 2018). Tendo a modularização

como o principal princípio, aplicações baseadas em microsserviços (μApp) consistem em pequenos serviços (ou processos), desacoplados, monofuncionais, e integrados via algum protocolo de comunicação leve, como HTTP (LEWIS; FOWLER, 2014; SAMPAIO et al., 2017; KRATZKE, 2017). Com a crescente complexidade das aplicações distribuídas, microsserviços tem se mostrado uma abordagem que facilita a manutenção e escalabilidade nas grandes organizações (JAMSHIDI et al., 2018).

Além disso, as aplicações distribuídas fazem uso de diferentes tecnologias e ferramentas para prover seus serviços. Cada tecnologia demanda recursos e tem suas peculiaridades para funcionar. Por exemplo, as aplicações distribuídas fazem uso de vários servidores: servidor de aplicação, servidor de banco de dados e assim por diante. Geralmente, essa é uma abordagem pouco eficiente, já que havia desperdício de recursos.

Como alternativa à instanciação de vários servidores, a utilização de máquinas virtuais sobre um mesmo hardware demonstrou uma melhoria na eficiência da utilização de recursos para uma aplicação distribuída (MALHOTRA; AGARWAL; JAISWAL, 2014). Uma mesma máquina física pode conter várias máquinas virtuais (VMs). E nessas VMs, devem haver componentes (ferramentas/tecnologias) da aplicação distribuída. Em outras palavras, o que era feito em várias máquinas físicas, passou a ser feito com a instanciação de várias máquinas virtuais sobre uma única máquina física.

Mas, uma máquina virtual, exceto pelo hardware, é uma máquina tal qual uma maquina física já que desempenham a mesma função. VMs demandam, ainda que virtual, memória RAM, HD e Sistema Operacional. Portanto, a criação e manutenção dessas máquinas, mesmo que virtuais, podem gerar uma grande sobrecarga (DUA; RAJA; KAKADIA, 2014). Afinal, no contexto aqui descrito, as máquinas virtuais visariam unicamente atender as demandas dos componentes da aplicação distribuída. Para contornar o uso das VMs, contêineres têm sido aplicado e se mostrado promissor (KANG; LE; TAO, 2016; DUA; RAJA; KAKADIA, 2014).

Um contêiner é um ambiente virtual portável, leve, de alto desempenho e com bom isolamento. As imagens usadas na construção do ambiente virtual conteinerizado são menores do que as usadas na criação de VMs, contendo apenas o essencial para funcionar, e diminuindo a sobrecarga normalmente causada por instanciação de máquinas virtuais (ISMAIL et al., 2015; KANG; LE; TAO, 2016). Uma imagem é um pacote executável que contém o básico para executar a aplicação - o código, as bibliotecas, as variáveis de ambiente e os arquivos de configuração (DOCKER, 2018).

Para Thönes (2015) e Kang, Le e Tao (2016), a conteinerização de aplicações pode potencializar os microsserviços. Afinal, ao invés de uma VM para cada componente da aplicação, podemos ter um contêiner, para cada microsserviço.

Atualmente, existem ferramentas para construir um ambiente virtual conteinerizado, como Docker (DOCKER, 2018). O Docker usa imagens base que contém apenas o essencial para funcionar. A customização da imagem é permitida e pode ser necessária para

prover o ambiente virtual desejado; sempre a partir de uma imagem base. As imagens são disponibilizadas nos repositórios do Docker, de onde são buscadas automaticamente.

Existem ainda ferramentas para o gerenciamento de μApps (aplicações baseadas em microsserviços) como o Docker Swarm (DOCKER-SWARM, 2018) e o Kubernetes (KUBERNETES, 2018). Essas ferramentas proveem *clustering*, escalonamento e gerenciamento de μApps (FAZIO et al., 2016). Elas também abstraem o *scheduling* e o *deployment*. As ferramentas também podem escalar a μApp sob demanda, trocando, implantando e reimplantando nos nós/*hosts* os microsserviços em tempo de execução. Então, o desenvolvedor não precisa se preocupar com os *hosts* onde os microsserviços serão implantados.

No entanto, essas operações de escalar podem levar à perda de desempenho, pois o Docker Swarm e Kubernetes não consideram uma eventual concorrência por recursos nos *hosts* ou a latência de comunicação entre os microsserviços (SAMPAIO et al., 2017; SAMPAIO et al., 2018).

Em suma, esses problemas impactam no desempenho das μApps por conta do nó/*host* onde cada microsserviço é implantado e suas alterações em tempo de execução. Uma mudança de arranjo/posicionamento dos microsserviços conteinerizados em tempo de execução pode reverter esse quadro. Essa operação tem um custo pequeno em microsserviços conteinerizados, diferentemente do esforço empregado numa operação similar em máquinas virtuais.

1.2 O Problema

Visto que as (μApps) são altamente dinâmicas, uma vez que seus microsserviços são facilmente escalados, alterados, substituídos e assim por diante:

Qual é o impacto disto no desempenho das μApps?

Como os microsserviços são desacoplados e bem modularizados, as atualizações de cada microsserviço numa μApp podem ser facilmente implantadas e reimplantadas por um desenvolvedor. Uma μApp pode mudar várias vezes durante o dia (SAMPAIO et al., 2018), ou seja, pode ser atualizada, ou mesmo substituída, para corrigir algum problema, ou mesmo para melhorar em algum sentido.

Além disso, as μApps gerenciadas em *clusters* pelo Kubernetes e Docker Swarm podem ter algum dos seus microsserviços escalados para mais ou para menos em tempo de execução. Isso acontece à medida que determinado microsserviço torna-se mais ou menos requisitado.

Todas as alterações mencionadas provocam uma contínua reconfiguração do *cluster* onde a μApp está executando, podendo ocasionar uma dispersão dos microsserviços pelos nós/*hosts* do *cluster*. Soma-se a isso o fato de que não há qualquer tratamento em relação a eventuais afinidades criadas entre microsserviços que trocam muitas mensagens entre

si. Se eles forem implantados em nó/*hosts* diferentes, a latência de comunicação pode impactar no desempenho da μApp. Da mesma forma, se microsserviços que demandam muitos recursos forem alocados em nós/*hosts* com outros microsserviços, a concorrência por recursos também impactará no desempenho da μApp.

Portanto, a dinamicidade com que se implanta, reimplanta, altera, e substitui os microsserviços causa um impacto sobre o desempenho das μApps. Na prática, estas operações provocam mudanças nos nós/*hosts* onde o serviço é implantado (*placement*) que potencialmente levam à queda de desempenho. Por exemplo, dois serviços com grande demanda de processamento podem, após a reimplantação, serem colocados em um mesmo nó.

Então, como intervir na implantação dos microsserviços para melhorar o desempenho da μApp?

Um software que trate essa questão pode ser a resposta. Este software deve conectar ao *cluster* onde o microsserviço executa e reconfigurá-lo. Isso deve realizar um rearranjo na distribuição/alocação dos microsserviços sem parar a μApp.

Portanto, um software capaz de movimentar um microsserviço de um nó/*host* a outro num *cluster* em tempo de execução pode melhorar o desempenho da μApp.

1.3 Deficiências do Estado da Arte

O *scheduling* e o *deployment* de ambientes virtuais como VMs, microsserviços e aplicações baseadas em microsserviços (μApps) tem sido amplamente estudado na literatura (JI; LIU, 2016; KAEWKASI; CHUENMUNEEWONG, 2017; GAO et al., 2013; CHAISIRI; LEE; NIYATO, 2009). Todavia, o foco dos trabalhos existentes é em melhorar o *scheduling* e o *deployment* de microsserviços em tempo de desenvolvimento.

Ji e Liu (2016) propuseram um método dinâmico de *deployment* orientado a SLA (*Service Level Agreement*). Eles tentam resolver problemas causados pela frequência com que se faz *deployment* de microsserviços na nuvem, analisando a distribuição de recursos no *cluster*, o que também reduz custos operacionais.

Zhao et al. (2015) modelou e formulou o problema de *scheduling*, e desenvolveu uma solução *open-source* para PaaS (*Plataform as a Service*), que reduziu o tráfego de rede em 60% sem impactar o balanceamento de carga. Kaewkasi e Chuenmuneewong (2017) usaram ACO (*Ant Colony Optimization*) para desenvolver um novo *scheduler* para o Docker.

Os trabalhos citados focam em tempo de desenvolvimento e não há preocupação com o que ocorre enquanto a aplicação está em execução.

Observado isso, outros trabalhos propuseram um mecanismo de adaptação de microsserviços em tempo de execução, como o REMaP (*RuntimE Microservices Placement)*) (SAMPAIO et al., 2017; SAMPAIO et al., 2018). Resumidamente, com a μApp em produ-

ção, informações são coletadas, analisadas e, se for o caso, há um planejamento de como fazer a adaptação. E a adaptação consiste em migrar microsserviços para maximizar o desempenho da μApp. Tal trabalho se baseia no MAPE-K (*Monitor - Analyzer - Planner - Executor - Knowledge*), que é um modelo de referência para se desenvolver software autonômico (IBM, 2006). Quando o MAPE-K é usado, a adaptação é executada pelo *Executor*.

Porém, o trabalho de Sampaio et al. (2017) não trata as réplicas dos microsserviços. Ou seja, a reconfiguração do *cluster* provida pelo REMaP acontece, mas as réplicas sempre acompanharão os seus repectivos microsserviços, pois o REMaP utiliza o *scheduler* padrão do Docker Swarm e do Kubernetes. Este fato pode sobrecarregar o nó/*host* de destino ou mesmo deixar de melhorar o desempenho da μApp ao não espalhar as réplicas desses microsserviços por vários nós/*hosts*.

1.4 Objetivos

O objetivo principal desta dissertação é prover uma solução capaz de mover microsserviços entre nós/*hosts* de um *cluster*, em tempo de execução, e sem uma parada completa da μApp. A solução proposta deverá ser capaz de tratar a migração de microsserviços com uma ou mais réplicas.

Para alcançamos este objetivo principal, alguns objetivos secundários precisam ser definidos: re-implementar o executor baseado em Sampaio et al. (2017), Sampaio et al. (2018); desenvolver um módulo que abstraia os demais componentes do MAPE-K; e tratar réplicas dos microsserviços.

1.5 Principais Contribuições

A contribuição principal deste trabalho é um software chamado CoMMAR (*Containerized Microservice Migration At Runtime*). Um componente de software capaz de mover microsserviços em tempo de execução, ou seja, um *scheduler* que pode ser usado em qualquer gerente de adaptação baseado no MAPE-K.

Além desta contribuição principal, o desenvolvimento do COMMAR possui também algumas contribuições secundárias:

- Um conector para *clusters* no Docker Swarm e no Kubernetes que estabelece a comunicação com o *cluster* para migrar os microsserviços de um nó/*host* para outro;
- Um *binder* que faz o microsserviço iniciar sua execução após o *scheduler* movê-lo; e
- Uma heurística simplificada que trata as réplicas de um determinado microsserviço, para permitir o *deployment* de réplicas em nós/*hosts* diferentes de modo customizado

1.6 Estrutura da Dissertação

Os próximos capítulos desta dissertação estão organizados como segue:

Capítulo 2 - Introduz os conceitos básicos necessários ao entendimento deste trabalho: softwares adaptativos, microsserviços, virtualização e conteinerização, e as ferramentas de gerenciamento Docker Swarm e Kubernetes.

Capítulo 3 - Apresenta a proposta desse trabalho, seus requisitos, arquitetura, projeto e implementação.

Capítulo 4 - Apresenta os experimentos realizados para a avaliação de desempenho do CoMMAR.

Capítulo 5 - Apresenta os trabalhos relacionados e faz uma análise comparativa com o que foi proposto nesta dissertação.

Capítulo 6 - Descreve as principais contribuições, limitações e trabalhos futuros desta pesquisa.

2 CONCEITOS BÁSICOS

Neste capítulo apresentaremos os conceitos usados neste trabalho. Nós começamos pelos softwares adaptativos, com ênfase na arquitetura MAPE-K, relacionando-a com o CoMMAR. Em seguida, serão discutidos os conceitos básicos de microsserviços, virtualização e conteinerização. Nós apresentaremos algumas ferramentas de gerenciamentos de aplicações baseadas em microsserviços e, sucintamente, abordaremos como o CoMMAR deverá atuar nelas. Por fim, nós descreveremos as estratégias padrões das ferramentas para realizar o *placement* de microsserviços.

2.1 Softwares Adaptativos

Um software adaptativo suporta mudanças comportamentais em tempo de execução, sem, no entanto, precisar ser reiniciado (TAYLOR; MEDVIDOVIC; OREIZY, 2009). Em outras palavras, não se deve parar a aplicação (VALETTO; KAISER, 2003). Kephart e Chess (2003) ressaltam que a adaptação é realizada através da computação autonômica. A computação autonômica, por sua vez, define um sistema computacional capaz de gerenciar a si mesmo dado os objetivos em alto nível de seu administrador (HUEBSCHER; MCCANN, 2008; KEPHART; CHESS, 2003).

Um software adaptativo precisa monitorar constantemente o ambiente em que executa e monitorar a si próprio, bem como analisar os dados, inferir se deve fazer alguma mudança e, se for o caso, atuar para fazê-la. Isso se repete indefinidamente como um *loop* de controle (SAMPAIO et al., 2018).

Da, Dalmau e Roose (2011) mencionam quatro razões que geralmente motivam a mudança (adaptação) num sistema:

1. *Mudanças nas condições do ambiente* - O sistema deve se adaptar, sem parar completamente, sempre que a infraestrutura do sistema mudar por conta de atualizações ou falhas no sistema;

2. *Detecção de Bugs* - Quando *bugs* são detectados, a aplicação deve ser atualizada para ajustar-se e corrigir-se;

3. *Evolução da Aplicação* - Quando uma nova versão de um componente da aplicação é desenvolvida, a aplicação é atualizada trocando a versão antiga do componente pela nova;

4. *Mudanças nos Requisitos da Aplicação* - A aplicação deve ser atualizada para satisfazer os novos requisitos.

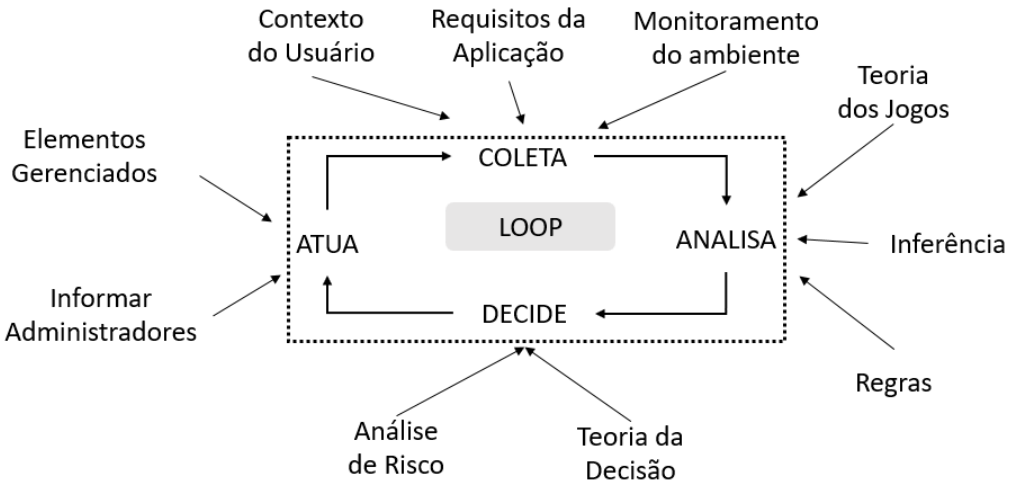

Figura 1 – *Loop* de adaptação

Essas adaptações podem ser reativas, corretivas ou evolucionárias (DA; DALMAU; ROOSE, 2011). Mudanças no ambiente disparam a adaptação reativa. Quando um problema é detectado, a adaptação corretiva pode ser executada. E a adaptação evolucionária é utilizada para suportar novas implementações ou novas tecnologias. A adaptação evolucionária também é utilizada para melhorar o atendimento aos requisitos.

Segundo Da, Dalmau e Roose (2011), as adaptações podem ser estruturais ou comportamentais. A adaptação estrutural modifica a aplicação adicionando, removendo ou substituindo os componentes. A adaptação comportamental, por sua vez, trata das mudanças comportamentais do sistema alterando alguma configuração, adicionando ou removendo alguma funcionalidade. De modo geral, essas adaptações estão estruturadas num *loop* de adaptação. Da, Dalmau e Roose (2011) estruturou um *loop*, como ilustrato na Figura 1. A IBM (2006), por sua vez, sistematizou uma arquitetura para softwares adaptativos: MAPE-K (*Monitor, Analyzer, Plan, Executor and Knowledge*).

2.1.1 *Loop* de Adaptação

Em sistemas auto-adaptativos, o *loop* de adaptação é parte do sistema. Mecanismos autônomos carregam a adaptação no sistema pelos sinais providos pelo ambiente e/ou pelo próprio sistema; o mecanismo autônomo decide o que fazer; e atua sobre o sistema gerenciado. Essa sequência de passos repete-se indefinidamente e é chamada de *loop* de controle ou *loop* de adaptação (SAMPAIO et al., 2018).

MAPE-K é um *loop* de adaptação que consiste numa arquitetura, ilustrada na Figura 2, para construir sistemas autonômicos e é composta por quatro atividades básicas que atuam num ciclo indefinidamente (WEYNS; MALEK; ANDERSSON, 2010; IBM, 2006; SALEHIE; TAHVILDARI, 2009). Essas atividades são:

- *Monitor* - responsável por coletar os dados do sistema gerenciado a partir de sen-

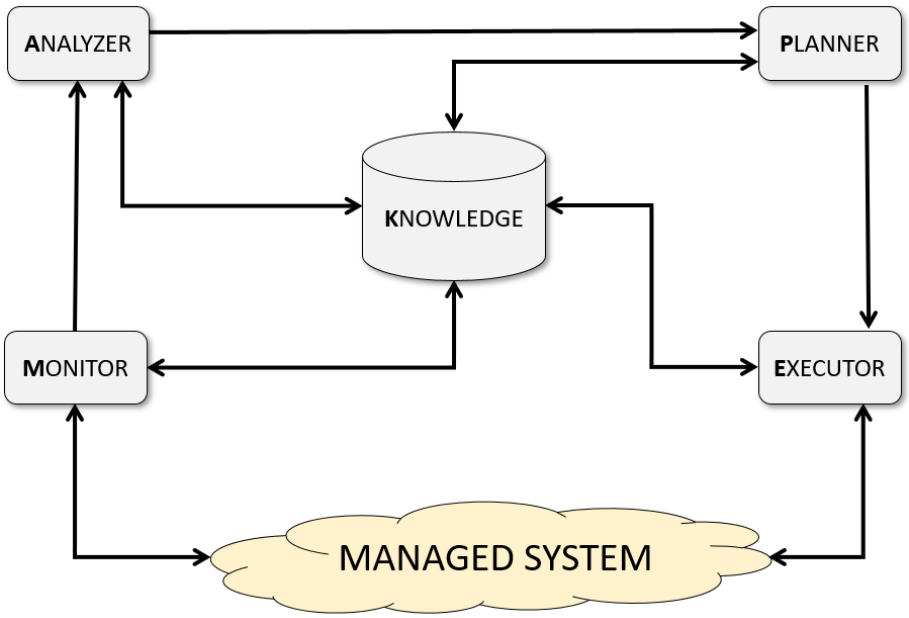

Figura 2 – Arquitetura MAPE-K

sores. Esses dados são despachado para o *Analyzer*;

- *Analyzer* - provê um mecanismo para interpretar os dados coletados e predizer futuras situações do μApp. Então, o *analyzer* encaminha o resultado dessa operação para o *Planner*;
- *Planner* - constrói as ações que forem necessárias para alcançar os objetivos, computando uma adaptação ou plano de gerenciamento;
- *Executor* - recebe o plano gerado no *Plan* e aplica ao sistema gerenciado.

Essas atividades compartilham o *Knowledge* que pode incluir um modelo do elemento gerenciado, uma descrição dos objetivos, propriedades, e outros tipos de dados que são usados para prover autonomia ao sistema (WHITE et al., 2004; SAMPAIO et al., 2018).

2.2 Microsserviços

Microsserviços é um estilo arquitetural que enfatiza a modularização e reforça a estrutura do time ao redor dos serviços (JAMSHIDI et al., 2018; BALALAIE; HEYDARNOORI; JAMSHIDI, 2016) como ilustra a Figura 3. Ou seja, a ideia é separar a aplicação em pequenas partes. Mas, ressalvamos que ainda há muita discussões que permeiam diversos aspectos dessa arquitetura.

Essas discussões passam por tópicos que vão desde a terminologia que define a arquitetura (*microservices*) até sobre o quão pequeno deve ser o microsserviço (THÖNES, 2015; JAMSHIDI et al., 2018; SAMPAIO et al., 2017), tangenciando outras arquiteturas como

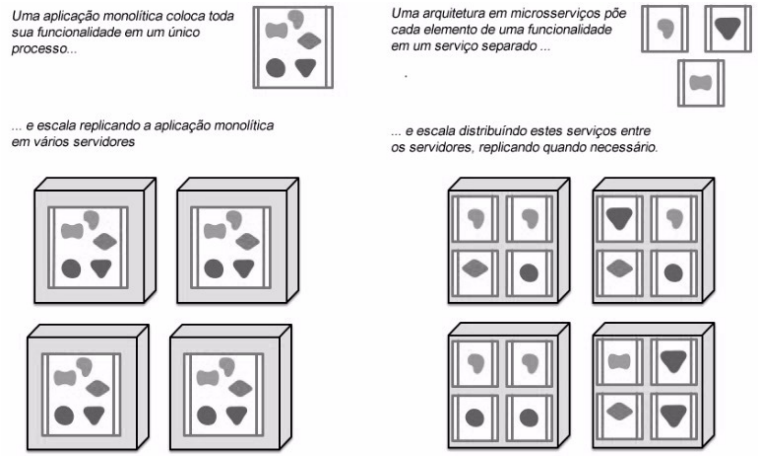

Figura 3 – Aplicações Monolíticas Versus Aplicações Baseadas em Microsserviços Lewis e Fowler (2014)

arquitetura orientada a serviços (SOA)(ZIMMERMANN, 2017; RICHARDS, 2015). Embora não se tenha ainda uma definição sobre o que são os microsserviços em todos os seus aspectos, a relação dos microsserviços com a conteinerização de aplicações tem tornado esse meio de se fazer software uma abordagem promissora.

2.3 Máquinas Virtuais e a Conteinerização

Uma aplicação distribuída, normalmente, depende de várias tecnologias para prover seus serviços. Banco de dados, servidores de aplicações, balanceador de carga e assim por diante. Cada serviço desse deve ser implantado numa servidor. Inicialmente, pode-se pensar em ter um nó/*host* para cada tecnologia. E cada nó/*host* pode demandar configuração, sistema operacional, infraestrutura diferentes. Esses nós/*hosts* devem suportar picos de acesso. Logo, há uma grande probabilidade de haver subutilização dos nós/*hosts* na maior parte do tempo.

Uma estratégia para resolver a subutilização dos nós/*hosts* seria implantando máquinas virtuais (VMs) em um único nó/*host*. Assim, custos seriam reduzidos como o de manutenção das máquinas, o de cabeamento e o de energia. Porém, ainda que sua estrutura seja virtualizada, uma máquina virtual demanda recursos tal qual uma máquina física, como memória RAM, disco (HD) e sistema operacional, por exemplo, como mostra a Figura 4. Além disso, o sistema operacional na máquina virtual também demanda recursos, como memória e processamento. Em suma, cada máquina virtual e seus respectivos sistemas operacionais consomem uma parcela significativa dos reais recursos disponibilizados pelo hardware. Ou seja, os recursos são tão ou mais utilizados para manter a máquina virtual do que para manter a aplicação que executa nessa VM.

Além disso, a manutenção de uma máquina virtual se aproxima bastante de uma máquina física, afinal, é preciso atualizar os sistemas operacionais, eventuais dependências

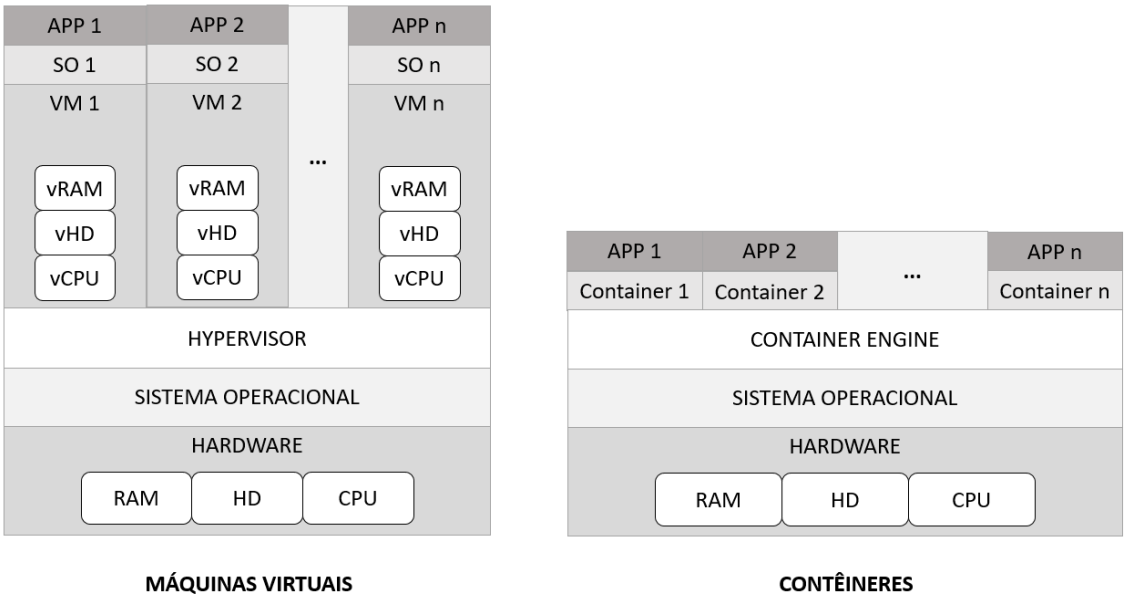

Figura 4 – VMs versus Contêineres

e assim por diante. Mais que uma aplicação distribuída, há uma preocupação em manter a estrutura como um todo.

Tudo isso demonstra que há uma sobrecarga na criação e manutenção das máquinas virtuais, e a conteinerização se mostra promissora para diminuir a sobrecarga e aprimorar a utilização dos servidores (DUA; RAJA; KAKADIA, 2014; SCHEEPERS, 2014).

Um contêiner é um ambiente portável, leve, de alto desempenho e com bom isolamento. O tamanho de uma imagem base é menor do que aquelas usadas numa virtualização, contendo apenas o essencial para funcionar (ISMAIL et al., 2015; KANG; LE; TAO, 2016).

A Figura 4 ilustra um cenário comparativo entre máquinas virtuais e uma *container engine*. As máquinas virtuais consomem recursos e ainda demandam sistemas operacionais, que também consomem recursos. Isso gera sobrecarga. Mas, a conteinerização compartilha os recursos de máquina e sistema operacional em sua base. E cada contêiner gera o ambiente virtual leve que a aplicação necessita.

Em suma, contêineres compartilham os recursos disponibilizados pelo hardware, são formados a partir de uma imagem base que contém apenas o essencial para executar, são fáceis de escalar, atualizar, implantar, reimplantar e substituir (SEO et al., 2014; FELTER et al., 2015).

Visto que os contêineres são fáceis de gerenciar, implantar microsserviços em contêineres pode ser uma boa estratégia. Afinal, os contêineres podem prover um ambiente leve e com os recursos necessários para que um microsserviço execute. Thönes (2015), Pahl e Jamshidi (2016) afirmam que as μApps podem ter seu desempenho melhorado quando combinada à conteinerização.

Todavia, nós destacamos que toda essa facilidade provida pela conteinerização implica numa alta dinamicidade da aplicação conteinerizada em tempo de execução. Prin-

cipalmente quando a aplicação é gerenciada por uma ferramenta de gerenciamento de microsserviços.

2.4 Ferramentas de Gerenciamento de Microsserviços Conteinerizados

A junção de microsserviços e conteinerização pode significar às aplicações um incremento de desempenho, uma maior facilidade para se manter a aplicação e um melhor aproveitamento dos recursos.

Existem ferramentas para gerenciar as aplicações baseadas em microsserviços (μApps) conteinerizados, como o Docker Swarm e o Kubernetes. Vale destacar que o Kubernetes é a ferramenta mais popular no segmento, sendo utilizada por provedores de nuvem como Google, Amazon e Microsoft (SAMPAIO et al., 2018).

Essas ferramentas lidam com o aumento da demanda pelas μApps escalando automaticamente os conteineres implantados, como no exemplo da Figura 5, onde o microsserviço M3 está sendo escalado. O Docker Swarm e o Kubernetes também podem abstrair o nó/*host* onde se implanta cada microsserviço. Na prática, quando se implanta uma μApp num *cluster* gerenciado por estas ferramentas, não é preciso determinar onde cada microsserviço da μApp deve ser implantado.

Quando um desenvolvedor configura um microsserviço para ser implantado, informações são passadas para indicar às ferramentas algumas limitações dos microsserviços no uso dos recursos (SAMPAIO et al., 2018). Por exemplo, o limite na criação de réplicas de determinado microsserviço, a porcentagem máxima de uso da CPU, e assim por diante.

Enfim, o Kubernetes e o Docker Swarm mantém a μApp dentro dos parâmetros que foram determinados e ainda podem abstrair esses parâmetros ou outros que não foram atribuídos, como o uso de memória e a garantia da qualidade de serviço.

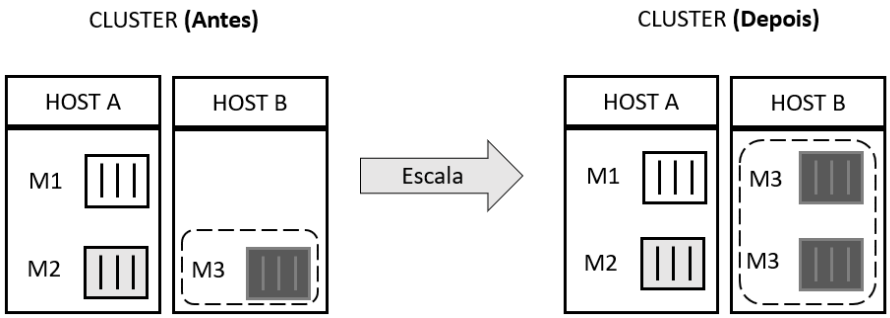

Figura 5 – Escalabilidade do Microsserviço M3

2.5 Objetos Kubernetes

O Kubernetes não trata os microsserviços conteinerizados diretamente, pois ele os abstrai em objetos próprios.

À medida que o Kubernetes foi sendo atualizado, alguns de seus objetos caíram em desuso e outros foram criados. Para o CoMMAR, os objetos mais importantes são: *Pod* e *Deployment*.

Na prática, um *cluster* Kubernetes abstrai os microsserviços conteinerizados em objetos *pods*. Esses objetos podem conter vários contêineres, mas o uso mais comum é um *pod* por contêiner.

Mas o objeto *pod* é volátil. Ou seja, não guarda o estado desejado do *cluster*. Por exemplo, se um desenvolvedor quer que sua μApp tenha um contêiner M2 no *cluster* com duas réplicas e um contêiner M1 com uma réplica, o objeto *pod*, usado para construir o *cluster*, não garante que o *cluster* permaneça no estado desejado/configurado.

A solução para garantir o estado do *cluster* é a abstração do objeto *pod* em outro objeto Kubernetes: o *deployment*. O objeto *deployment* garante que se o *pod* falhar por alguma razão, o *pod* será reiniciado para garantir o estado do *cluster* previamente definido em sua especificação.

A Figura 6 ilustra a abstração provida pelos objetos *deployments* (Configuração dos *Deployments*). Seguindo o exemplo da Figura 6, o *CLUSTER - Cenário 1* é atualizado para atuar com objetos *deployment*. Então, temos no *cluster* objetos *deployments* garantindo que os objetos *pods* se mantenham como foram especificados (*CLUSTER - Cenário 2*) previamente.

Quando um objeto *deployment* do Kubernetes é implantado num *cluster* gerenciado pelo próprio Kubernetes, geralmente está descrito o nome do microsserviço e, às vezes, número de réplicas. A partir disso, o Kubernetes cria objetos *pods* atendendo ao número

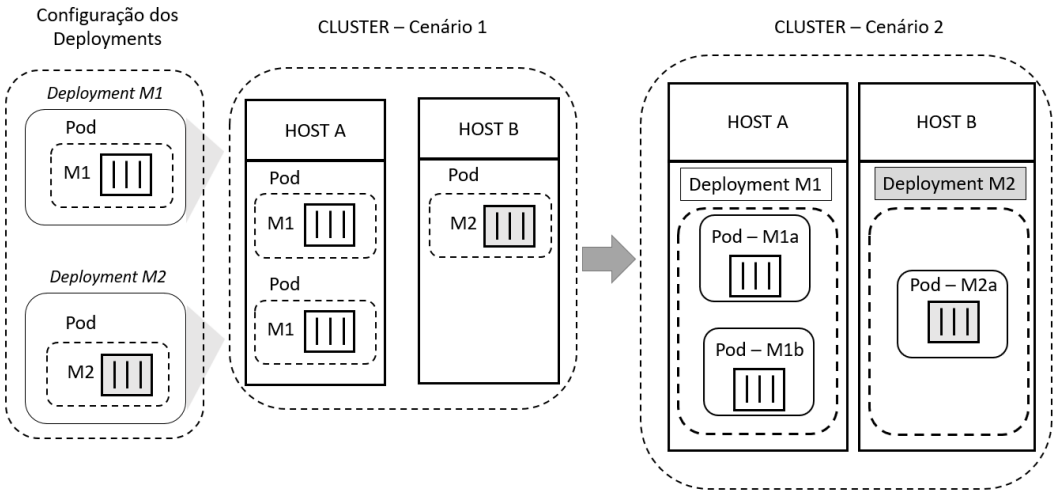

Figura 6 – Objeto *Deployment* num *Cluster* Gerenciando os Objetos *Pods*

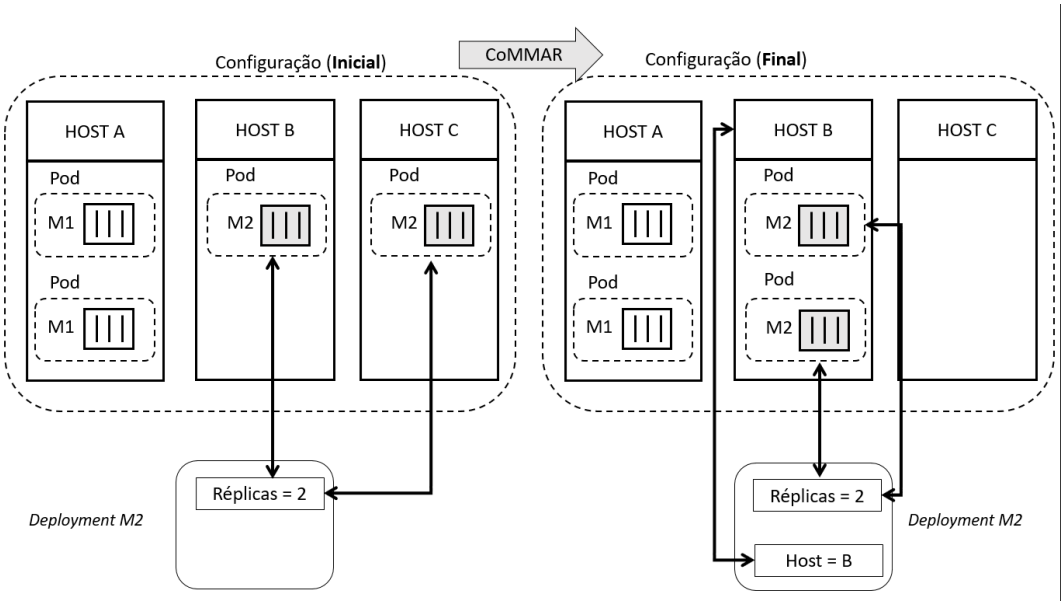

Figura 7 – CoMMAR reconfigurando os *Pods* no *Cluster* Kubernetes

de réplicas atribuídas no objeto *deployment*.

A partir desse ponto sempre que nos referimos à *migração de microsserviço* abstraímos os termos que se referam aos objetos Kubernetes, ou seja, mover um microsserviço no Kubernetes é, na verdade, mover um objeto *pod*. Também consideramos que microsserviços replicados no Kubernetes, são *pods* replicados.

Um exemplo de reconfiguração de um *cluster* Kubernetes é apresentado na Figura 7. Um objeto *Deployment* M2 mantém a informação sobre as réplicas do *cluster (Configuração inicial)*. Sucintamente, pode-se modificar informações sobre o *placement* dos microsserviços (*pods*). Uma restrição é atribuída, e isso faz com que uma das réplicas migre para o nó/*host* de destino (*HOST B*).

Nós destacamos ainda que em um *cluster* Kubernetes existem outros objetos, como o *Statefulset*, usado para conteinerização de banco de dados, e o *Replication Controller*, que caiu em desuso devido ao surgimento do objeto *deployment*.

2.6 *Placement* de Microsserviços

As ferramentas de gerenciamento de microsserviços possuem estratégias para implantar os microsserviços conteinerizados nos nós/*hosts* do *Cluster*. Sampaio et al. (2018) lista estas estratégias:

- *Estratégia Spread* - A ferramenta de gerenciamento aloca o mínimo de microsserviços por nó/*host*. Essa estratégia tenta evitar a contenção de recursos desde que poucos microsserviços disputem os mesmos recursos. Porém, isso pode degradar o desempenho da μApp adicionando latência na comunicação caso os microsserviços sejam implantados em nós/*hosts* diferentes. Além disso, essa estratégia pode desper-

diçar recursos uma vez que um microsserviço pode não necessitar de todo o recurso disponibilizado pelo nó/*host* em que foi implantado. Docker Swarm e Kubernetes usam essa estratégia;

- *Estratégia Bin-pack* - A ferramenta de gerenciamento usa o mínimo de nós/*hosts* para implantar os microsserviços, evitando o desperdício no *cluster*. Porém, alocar muitos microsserviços juntos pode causar contenção de recursos, degrandando o desempenho da μApp. Essa estratégia está disponível no Docker Swarm;

- *Estratégia de Etiquetagem* - Os microsserviços conteinerizados podem ser etiquetados com atributos que guiam à seleção do nó/*host*. Por exemplo, um microsserviço que demande um nó/*host* que contenha GPU, pode ser etiquetado com essa informação em tempo de implantação. Ou seja, isso pode estabelecer algumas restrições aos *schedulers* padrões do Docker Swarm e do Kubernetes;

- *Estratégia Randômica* - A ferramenta escolhe o nó/*host* onde o microsserviço será implantado randomicamente. Essa estratégia está disponível no Docker Swarm

2.7 Considerações Finais

Esse capítulo apresentou os principais conceitos básicos necessários ao entendimento deste trabalho. Primeiro, nós abordamos os softwares adaptativos estabelecendo as ideias formadas pela arquitetura MAPE-K. Depois nós discutimos sobre os microsserviços e como eles estão ligados à conteinerização, traçando um paralelo sobre sobre máquinas virtuais. Nós destacamos ainda a importância da compreensão dos objetos do Kubernetes. Por fim, nós terminamos com as estratégias de *placement* já disponíveis nas ferramentas de gerenciamento de aplicações conteinerizadas.

3 COMMAR

Este capítulo apresenta detalhes do desenvolvimento do CoMMAR. Inicialmente, serão discutidos os requisitos funcionais e não funcionais da solução. Em seguida, apresentaremos a arquitetura definida para atender os requisitos estabelecidos. Finalmente, na última parte do capítulo, apresentaremos detalhes do projeto e da implementação do CoMMAR.

3.1 Requisitos

O CoMMAR é um componente de software baseado no MAPE-K (descrito na Seção 2.1.1), cuja funcionalidade principal é mover, em tempo de execução, microsserviços conteinerizados entre nós/*hosts* do *cluster* tratando individualmente, quando for o caso, as réplicas dos microsserviços. Na prática, o CoMMAR implementa o *Executor* do MAPE-K. A seguir apresentamos os requisitos funcionais e não-funcionais da solução proposta.

3.1.1 Requisitos Funcionais

A migração de microsserviços entre nós, em tempo de execução, requer a implementação de várias funcionalidades, como mostrado na Tabela 1.

Para conseguir mover os microsserviços no *cluster*, inicialmente o CoMMAR precisa selecionar o nó/*host* para onde o microsserviços será migrado [RF01]. Num ambiente com

Tabela 1 – Requisitos Funcionais

Identificação	**Nome da Funcionalidade**	**Descrição**
[RF01]	Selecionar nó/*host*	Selecionar nó/*host* do *cluster* para o microsserviços ser movido.
[RF02]	Selecionar microsserviço	Fazer a seleção do microsserviço que será movido no *cluster*.
[RF03]	Tratar réplicas	Caso existam, tratar as réplicas selecionadas no *cluster* para espalha-lás.
[RF04]	Monitorar *status*	Monitorar os microsserviços (em alguns casos) a fim de verificar se ele foi implatado no nó/*host* de destino.
[RF05]	Realizar *bind*	Iniciar o microsserviço no novo nó/*host*, quando for necessário.

Tabela 2 – Requisitos Não-Funcionais

Identificação	Nome do Requisito	Descrição
[RNF01]	Interoperabilidade	O CoMMAR deve funcionar tanto no Docker Swarm quanto no Kubernetes.
[RNF02]	Desempenho	O desempenho do CoMMAR deve se aproximar dos *schedulers built-in* do Kubernetes e do Docker Swarm.
[RNF03]	Confiabilidade	Caso aconteça algum problema durante a migração, a μApp não pode falhar.

um gerente baseado no MAPE-K, essa escolha vem do processo resultante de operações realizadas pelo *monitor*, *analyzer* e pelo *planner*.

Além de selecionar o nó, é preciso definir qual microsserviços precisa ser movido [RF02]. Da mesma forma, a definição do microsserviços depende de operações do gerente baseado no MAPE-K.

O requisito RF03 trata da possibilidade de direcionamento das réplicas dos microsserviços. Ao realizar a migração, o CoMMAR deve permitir o espalhamento das réplicas dos microsserviços entre diferentes nós/*hosts*.

Nós ressaltamos que houveram situações em que o microsserviços precisava ser monitorado. Após um microsserviços ser movido, por exemplo, algumas situações o faziam não ser inicializado. Então, um módulo para observar o *status* do microsserviços foi desenvolvido [RF04] e, caso ele não estivesse inicializado no novo nó/*host*, um outro módulo desenvolvido o inicializava [RF05].

3.1.2 Requisitos Não-Funcionais

O CoMMAR, por tratar da migração em tempo de execução de microsserviços em *clusters*, precisa satisfazer requisitos de desempenho e confiabilidade para reduzir o seu impacto sobre as aplicações baseadas em microsserviços (μApp). Um requisito adicional está relacionado à interoperabilidade [RNF01], pois a solução proposta deve funcionar com duas das tecnologias mais utilizadas para gerenciar aplicações conteinerizadas: Docker Swarm e Kubernetes (SAMPAIO et al., 2018). Esses requisitos estão apresentados na Tabela 2.

O desempenho do CoMMAR [RNF02] deve ser observado sob duas diferentes perspectivas. A primeira delas diz respeito ao desempenho do μApp sobre a qual o CoMMAR irá atuar. Há aplicações sensíveis a impactos no desempenho. Mas, as μApps da Internet, majoritariamente, lidam bem com o impacto da migração de contêineres em tempo de execução. A segunda perspectiva trata do desempenho do próprio CoMMAR. Afinal, se o CoMMAR puder ter um tempo de execução próximo ao padrão do Docker Swarm e do Kubernetes, ele pode se tornar *transparente* às aplicações durante sua atuação.

Quanto à confiabilidade [RNF03], o CoMMAR não pode finalizar a execução da μApp

em nenhuma hipótese. Não faz sentido o CoMMAR tentar reconfigurar o *cluster* e acabar terminando a μApp.

Nós destacamos que o CoMMAR *migra* o microsserviços de um nó/*host A* para outro nó/*host B*. E esse processo consiste em: primeiro - instanciar o microsserviços no novo nó/*host* B; e segundo - finalizar a instância do nó/*host* de origem A. Se algo sair errado nesse processo, a μApp não pode falhar em sua execução [RNF03].

3.2 Arquitetura

A partir dos requisitos apresentados na seção anterior, a arquitetura do CoMMAR foi definida e é apresentada na Figura 8. A arquitetura é composta por quatro componentes principais: *Conexão e Setup*, *Core*, *Estratégia de Scheduling* e o *Scheduler Customizado*.

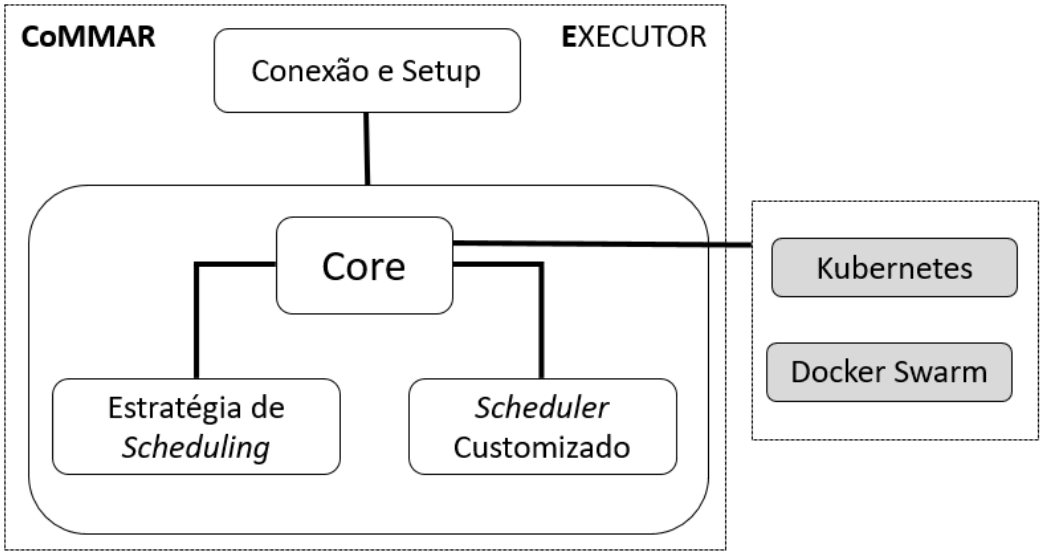

Figura 8 – Arquitetura do CoMMAR

O componente *Conexão e Setup* é responsável por estabelecer a conexão com o *cluster* e definir os dados necessários para realizar a operação *mover*. Para isto, este componente utiliza o nome do nó/*host* onde o microsserviços será implantado e o nome do microsserviços conteinerizado.

O *Core* é responsável por migrar os microsserviços, mas a execução da operação de migração depende do ambiente de execução, i.e., depende se o mecanismo é o Docker Swarm ou Kubernetes. Caso seja o Docker Swarm, o *Core* utiliza o componente *Estratégia de Scheduling*. Esse Componente é responsável por espalhar as réplicas dos microsserviços nos nós/*hosts* de destino.

Por outro lado, caso o ambiente seja o Kubernetes, o *Scheduler Customizado* é utilizado. Esse componente tem a mesma função que a *Estratégia de Scheduling*, mas é mais complexo e robusto. Estes componentes serão detalhados nas Seções 3.4, 3.5 e 3.6.

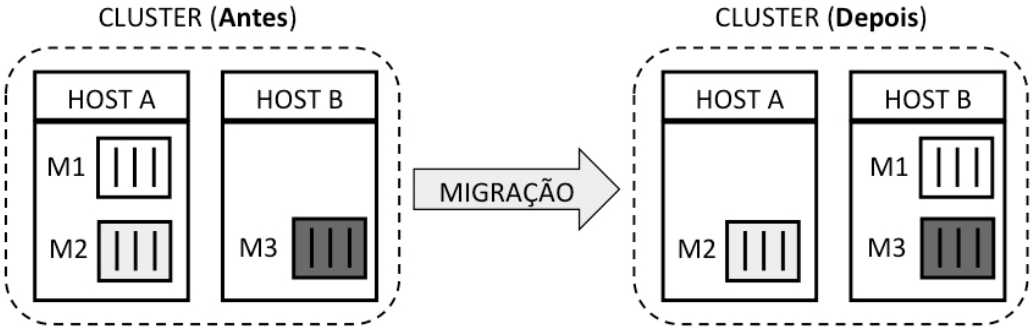

Figura 9 – Reconfiguração do *Cluster*

A Figura 9 exemplifica o processo de migração de um microsserviços. O microsserviços *M1* é movido do *Host A* para o *Host B* pelo CoMMAR. A decisão pela migração ocorre como consequência da análise de dados monitorados no *cluster* (feita pelo *Monitor)* e por uma decisão do *Analyzer*. Após a decisão pela migração, o *Planner* constrói um plano de adaptação que é executado pelo CoMMAR (*Executor*). Na implementação proposta por Sampaio et al. (2017), a adaptação é disparada analisando as afinidades entre os microsserviços.

As informações sobre o microsserviços que será migrado e o nó/*host* destino que chegam ao CoMMAR são repassados ao componente *Conexão e Setup*, responsável pela conexão com o *cluster* onde o CoMMAR atuará. Em seguinda, as informações chegam no *Core*. O *Core* é responsável por efetivamente migrar o microsserviços. Mas, se o microsserviços tiver réplicas, os componentes *Estratégia de Scheduling* e *Scheduler Customizado* da arquitetura podem ser executados a depender do ambiente de execução em que o CoMMAR estiver atuando. A *estratégia de scheduling* e o *scheduler customizado* têm a capacidade de espalhar as réplicas do microsserviços M1 pelos nós/*hosts*.

3.3 Conexão e *Setup*

Como mencionado anteriormente, o componente *Conexão e Setup* é responsável por realizar a conexão com o *cluster* em que o CoMMAR vai atuar. Isso significa que vários dados devem ser informados, tais como: privilégios de acesso e dados para a configuração da API de conexão com o *cluster*. Num software MAPE-K, esses dados normalmente seriam repassados pelo *Planner*.

3.3.1 Implementação

O componente *Conexão e Setup* funciona de maneira semelhante no Docker Swarm e no Kubernetes. No Kubernetes, um arquivo de configuração que contém privilégios de acesso é passado para se obter acesso ao *cluster*. Com esse acesso, diversas operações podem ser realizadas através da API disponibilizada, e.g., alterar os dados de cada microsserviços

como o nome de seu *scheduler* e dados sobre o *placement*. Essencialmente é isso que o CoMMAR faz. No Docker Swarm, apenas a URI do *cluster* é informada para a conexão ao *cluster*. Mas, no Kubernetes e Docker Swarm o acesso é realizado através de uma rede privada.

Por serem, na prática, dois *Executors* baseados no MAPE-K, temos duas implementações do CoMMAR. O Código 3.1 descreve como é feita a conexão ao *cluster* do Docker Swarm.

Listing 3.1 – Conexão no Docker Swarm

```
final String uri = "http://localhost:80";
static DockerClient connGFADS() {
        DockerClient dockerConn = null;
            docker = DefaultDockerClient.builder()
            .uri(URI.create(uri))
            .build();
        return dockerConn;
}
```

Nesta implementação foi utilizada a API Java do Docker (Docker *client*)[1], versão *3.5.12*.

Em relação à implementação da versão CoMMAR para o Kubernetes, a conexão (e demais operações no *cluster*) depende da API *Kubernetes Java Client*[2] e da leitura do arquivo de configuração que contém os privilégios de acesso ao *cluster*, como mostrado no Código 3.2.

Listing 3.2 – Conexão no Kubernetes

```
    //algum caminho para o arquivo de configura  o
    final String filepath = "PATH_TO_FILE";
    public static ApiClient SetEnv() {
        //diret rio do arquivo mais o nome do arquivo
        String pathToGfadsConfigFile = filepath;
        FileReader gfadsConfigFile;
        try {
            gfadsConfigFile = new FileReader(
               pathToGfadsConfigFile);
            KubeConfig kc = KubeConfig.loadKubeConfig(
               gfadsConfigFile);
            ApiClient client = Config.fromConfig(kc);
```

[1] https://github.com/spotify/docker-client
[2] https://github.com/kubernetes-client/java

```
12              Configuration.setDefaultApiClient(client);
                return client;
14          } catch (FileNotFoundException e) {
                e.printStackTrace("file not found");
16          } catch (IOException e) {
                e.printStackTrace("something went wrong");
18          }
            return null;
20      }
```

A versão da API do *Kubernetes Java Client* é *1.0.0-beta1*. A implementação atual do CoMMAR é sensível à esta versão e a outras dependências, tais como: o conjunto de bibliotecas *jackson*; e a biblioteca *json patch*. Desta forma, caso haja uma atualização das APIs, o CoMMAR precisa ser atualizado. As outras dependências serão descritas à medida que forem explorados os componentes diretamente relacionado a elas.

3.4 *Core*

O *Core* efetivamente migra o microsserviços escolhido para o nó/*host* de destino. A exemplo do componente *Conexão e Setup*, o *core* atua de forma diferente se o *cluster* é gerenciado pelo Docker Swarm ou Kubernetes.

3.4.1 Projeto e Implementação

Com a conexão com o *cluster* estabelecida e o *setup* dos dados realizados, deve-se iniciar a migração (função *Mover*). A função *Mover* recebe os dados do componente *Conexão e Setup* (nome do nó/*host* de destino e o nome do microsserviço a ser movido) e valida as informações verificando se dados como o nome do nó/*host* de destino (*hostname*) e o microsserviços a ser movido existem no *cluster*. Nós destacamos que essa validação não se faz necessária num ambiente onde todos os componentes do MAPE-K estão implementados. Afinal, se, por exemplo, o nome microsserviços que será movido não existe no *cluster*, uma excessão seria lançada pelos demais componentes, interrompendo o fluxo de execução antes de chegar ao *executor*.

A Figura 10 mostra o passo-a-passo de como a função *Mover* é realizada. Dada uma configuração inicial dos microsserviços no *cluster* (*Configuração 1*), uma nova instância do microsserviços que será movido é criada no *host* destino (*Configuração 2)*. Em seguida, após a nova instância ficar pronta para receber as requisições (*Configuração 3*), a antiga instância é desativada (*Configuração 4*).

É importante observar que apenas microsserviços *stateless* devem ser movidos com o CoMMAR. Isso também foi observado no trabalho de Sampaio et al. (2018), pois as ferramentas de gerenciamento de microsserviços conteinerizados não têm mecanismos para

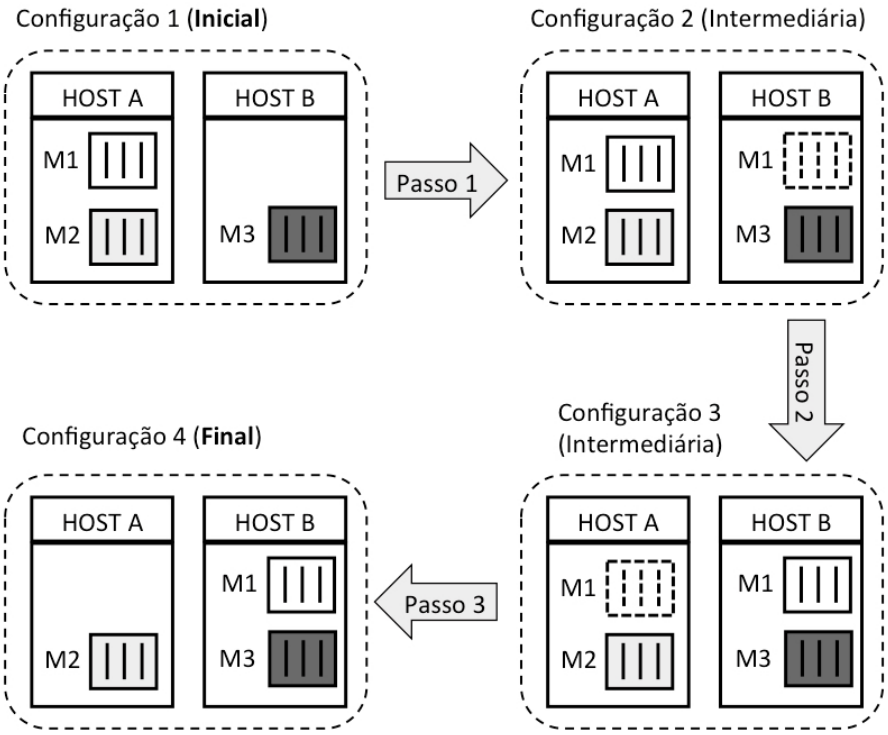

Figura 10 – Passo a Passo da Migração de microsserviços

lidar com a migração e/ou replicação de dados. Logo, não se deve mover, por exemplo, um microsserviço conteinerizado que seja um banco de dados.

No cenário de migração de microsserviços implementado pela função *Mover*, é importante ressaltar como se lida com microsserviços com réplicas. Na implementação do Docker Swarm e Kubernetes, quando um microsserviços possui réplicas, todas as réplicas são migradas para um único nó. Para evitar a migração desta forma, optou-se por estender o escalonamento padrão implementado pelas ferramentas de gerenciamento. A extensão consiste basicamente em implementar escalonadores que consigam espalhar as réplicas por outros nós do *cluster*.

Estas extensões são apresentadas nas subseções seguintes.

3.5 *Scheduler* Customizado

A extensão proposta para o escalonamento dos microsserviços no Kubernetes foi implementada em um *Scheduler Customizado* (como mostrado na Figura 8). Ele é capaz de lidar com as réplicas, espalhando-as nos nós/*hosts* informados pelo componente *Conexão e Setup*. Isso contrasta com o *scheduler default* do Kubernetes, onde os microsserviço e todas as suas réplicas são movidas para o mesmo nó/*host*. Para compreender sua implementação e funcionamento, nós faremos uso de informações previamente descritas na Seção 2.5.

Embora o *scheduler* customizado direcione as réplicas para vários nós/*hosts*, essas ré-

plicas não são iniciadas nos seus respectivos destinos. Desta forma, foi preciso implementar um componente adicional para ativá-las, chamado *Watch*.

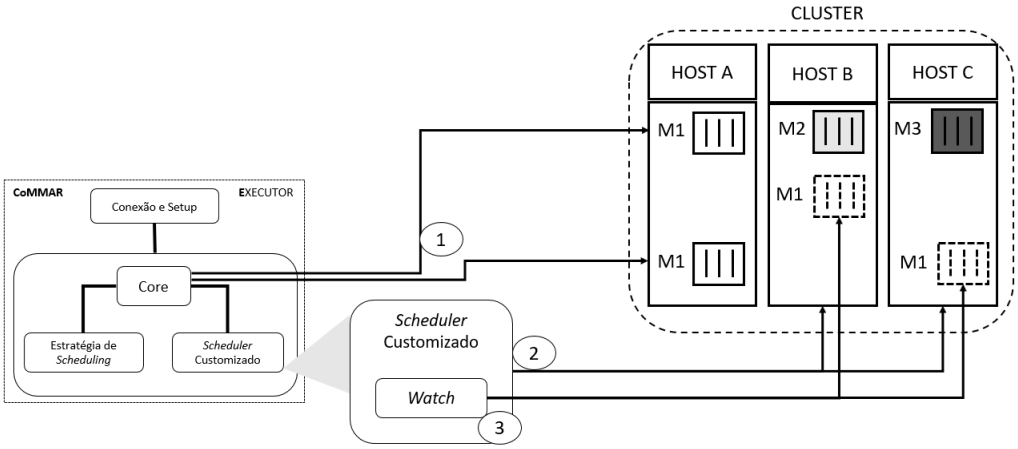

Figura 11 – Módulo Interno do *Scheduler* Customizado - *Watch*

A Figura 11 mostra como a migração das réplicas acontece. Primeiro o *Core* determina o microsserviço que será movido (M1) (1) para o novo nó/*host*. Então, o *Scheduler Customizado* direciona réplica a réplica o destino dos microsserviços (2). O *Watch* monitora brevemente os nós/*hosts* de destino do microsserviço para ativar o microsserviço e/ou suas réplicas, inicializando-os nos nós/*hosts* de destino (3). O *Watch* é, de fato, quem implanta e possibilita a inicialização do microsserviço (3) no seu respectivo novo nó/*host* (atendendo ao RF05). Então, tem-se a nova configuração do *cluster*, ilustrado na Figura 12. M1 é composto por duas réplicas, visto o exemplo ilustrado na Figura 11. Cada réplica é direcionada para um nó/*host* (*HOST B , HOST C*) diferente de sua origem (*HOST A*).

Como mencionando na Seção 2.5, e observando que trataremos a partir daqui de detalhes do funcionamento do CoMMAR no Kubernetes, na prática, o *cluster* Kubernetes

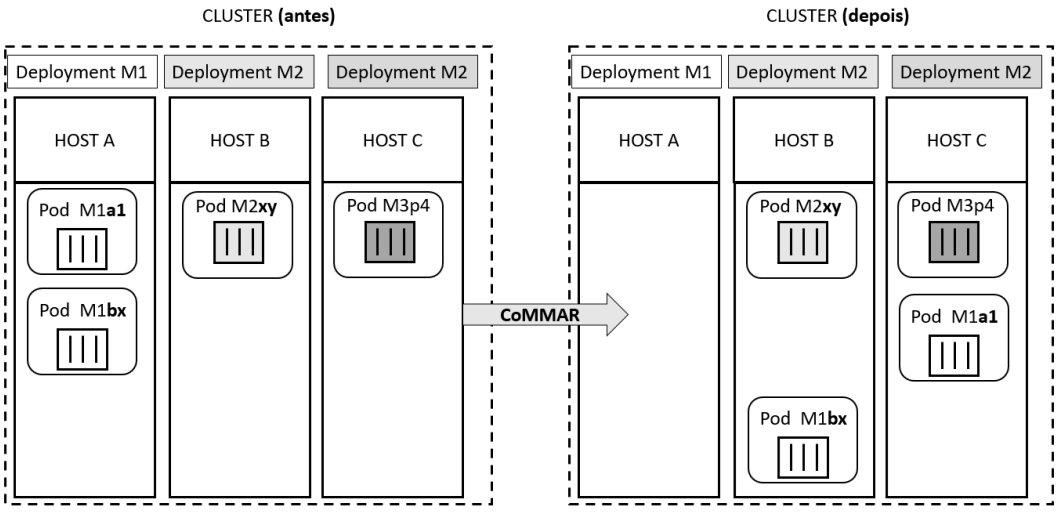

Figura 12 – Nova Configuração do *Cluster* no Kubernetes com o *Scheduler* Customizado

têm microsserviços conteinerizados abstraídos em objetos *pods* que são, por sua vez, gerenciados por objetos *deployment*.

Os objetos *pods* tem seu nome derivado dos nomes especificados para os objetos *deployments*. Por exemplo, na Figura 12, o objeto *deployment M1* gerou dois *pods*: M1a1 e M1bx. Seguindo a ilustração da Figura 11, foi determinado que o microsserviços M1 deveria ser migrado (1). Ná prática, os *pods M1* derivados do *deployment M1* serão migrados. O direcionamento individualizado de cada réplica para nó/*host* de destino é feito através de uma abstração simplificada do que seria a atuação dos demais componentes do MAPE-K.

Então, na nossa abstração simplificada, nós determinamos que as réplicas/*pods* com nomes terminados em números deveriam migrar para o nó/*host C* e os terminados em letras para o nó/*host B* ((2) da Figura 11).

Por fim, esses objetos *pods* não inicializam após migrados. Por isto, desenvolvemos o *Watch* que monitora brevemente os nós/*hosts* de destino dos microsserviços migrados e realiza o *bind* (RF05), o que inicializa o *pod* (3). O resultado da operação executada pelo CoMMAR num *cluster* com réplicas é apresentado na Figura 12.

Para implementar essas funcionalidades nós usamos a linguagem Java na versão oito, e APIs externas. A primeira API é *json patch*[3] na versão *1.9*. Temos ainda um conjunto de APIs *Jackson*: *Jackson-core*[4] versão *2.9.2*, *Jackson-annotation*[5] na versão *2.9.2* e *Jackson-databind*[6] também na versão *2.9.2*. Essas APIs fazem um mapeamento do objeto Kubernetes de maneira abstrata e retorna um objeto num formato aceito pela função responsável por atualizar o *cluster*. Essa função está disponibilizada pela *Kubernetes Java Client*.

Em seguida, o CoMMAR verifica qual *scheduler* está configurado no microsserviço. Se for o *scheduler default* ele executa o mover, e termina a execução. A real implantação fica a cargo do Kubernetes. Se for um outro *scheduler*, no nosso caso chamado de *gfads*, ele executa o *scheduler* customizado. Esse fluxo de execução está descrito na Figura 13.

[3] https://mvnrepository.com/artifact/com.github.fge/json-patch/1.9
[4] https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core
[5] https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-annotations/2.2.3
[6] https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind/2.0.0

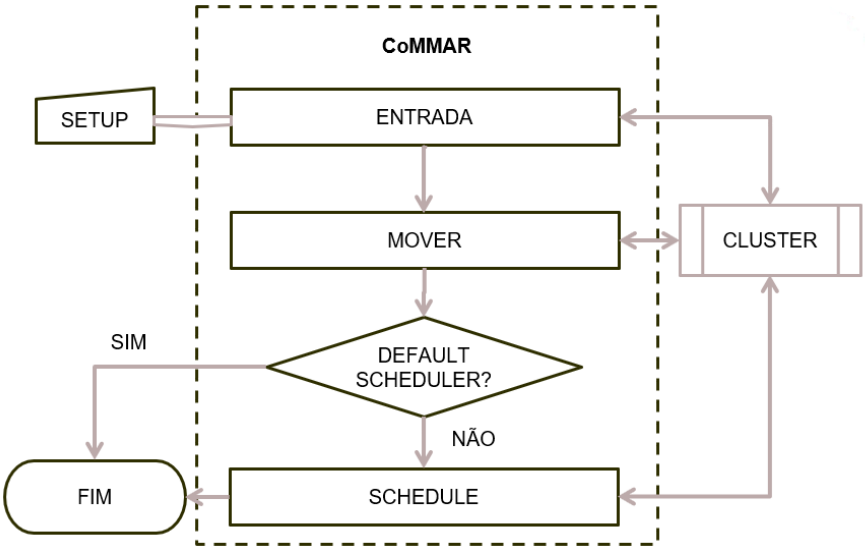

Figura 13 – Fluxo de Execução do Kubernetes

3.6 Estratégia de *Scheduling*

No Docker Swarm, uma abordagem diferente foi adotada e uma estratégia de *scheduling* foi desenvolvida. A estratégia consiste em usar o *scheduler default* do Docker Swarm para espalhar as réplicas pelos nós/*hosts* passados no setup. Ao contrário do Kubernetes, no entanto, não é possível garantir o direcionamento individualizado de cada réplica. Mesmo assim, a solução é mais robusta, pois espalha as réplicas apenas entre os nós/*hosts* que foram determinados pelo *Core*.

3.6.1 Projeto e Implementação

O Docker Swarm fornece algumas primitivas que facilita a implementação do *scheduling* de microsserviços conteinerizado. Usando estas primitivas é possível determinar diretamente qual nó/*host* do *cluster* se quer implantar determinado microsserviço. Além disso, temos a opção de determinar quais nós/*hosts* não podem receber um determinado microsserviço.

A Figura 14 mostra um cenário de uso da solução proposta. O *Core* determina qual microsserviço vai ser movido (1) e quais os possíveis nós/*hosts* de destino (2). Estas informações são passadas para o componente *Estratégia de scheduling*, que faz a migração negando a implantação nos nós/*hosts* não escolhidos (3). Essa negação pode incluir o nó/*host* de origem, como ilustra a Figura. Caso o microsserviço tenha réplicas, as réplicas também serão movidas, e provavelmente espalhadas entre os nós/*hosts* determinados.

É importante observar que o componente *Estratégia de Scheduling* usa funcionalidades providas pelo Docker Swarm para manipular as réplicas. Desta forma, não necessariamente as réplicas vão se espalhar pelos nós/*hosts* escolhidos pois, o *scheduler default* ainda usará sua respectiva estratégia *spread* (estratégia padrão, que foi explicada na Seção 2.6) para determinar onde implantar cada réplica atendendo apenas a restrição de não implantar em

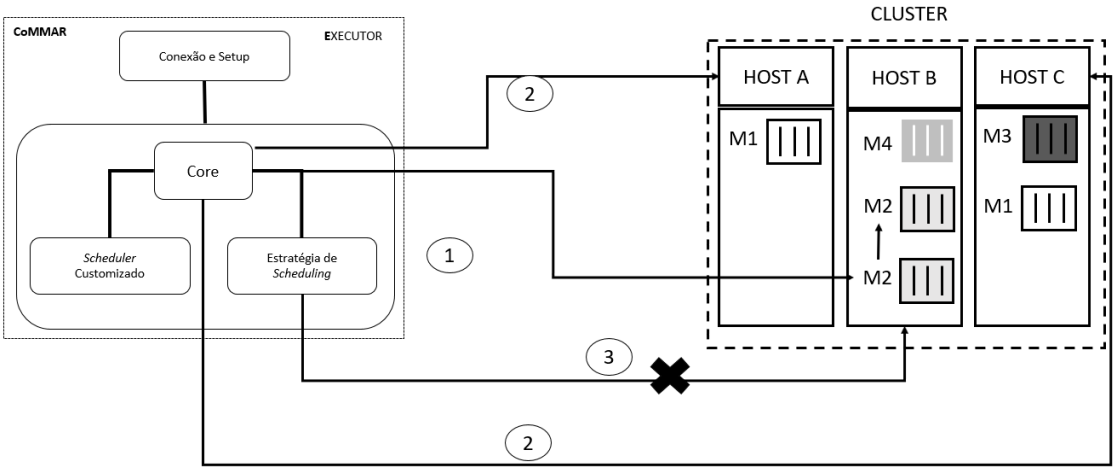

Figura 14 – Estratégia de *Scheduling*

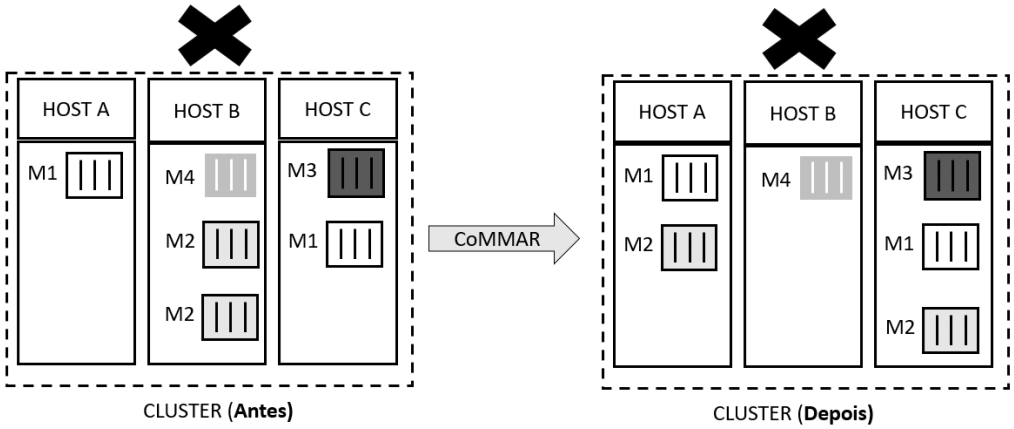

Figura 15 – Nova Configuração no *Cluster* Docker Swarm após a execução da Estratégia de *Scheduling* ter Atuado

determinados nós/*hosts*. Porém, de maneira geral, o *espalhamento* de réplicas acontece. Principalmente quando há várias réplicas.

Seguindo o exemplo ilustrado na Figura 14, o cenário inicial do *cluster* nos levará, após a atuação do CoMMAR, ao que é apresentado na Figura 15 (*Cluster depois*): O microsserviços M2 é escolhido para migrar e ele tem duas réplicas; Os nós/*hosts* de destino escolhidos são os *HOST A* e *HOST C*. Na prática, a Figura 14 destaca que a informação passada é sobre os *hosts não escolhidos*. Isto significa que o *HOST B* foi negado (*CLUSTER Antes*), e o microsserviços *M1* migrou para os *hosts* escolhidos (*CLUSTER Depois*).

A implementação do que é descrito na Figura 15 está exemplificada no Código 3.3. A função descrita no código recebe como entrada os possíveis nós/*hosts* de destino; compara com todos os nós/*hosts* do *cluster* gerenciado pelo Docker Swarm; e retorna uma lista com os nós/*hosts* não escolhidos. O resultado dessa operação é usado para adicionar as

propriedades de restrições (*constraints*) disponibilizadas pelo Docker Swarm através de sua API.

Listing 3.3 – nós/*hosts* negados

```
    /**
2   * this method is used to set constraints using "!=".
    * It gets the nodes the user want and
4   deny all the user do not want
    */
6   List<String> denyNodes(List<String> destinationNodes) {
        List<String> nodesNotWanted = new LinkedList<String>();
8       for (Node node : this.docker.listNodes()) {
            if(!destinationNodes.contains(node.description().
               hostname())) {
10              nodesNotWanted.add(node.description().hostname()
                    );
            }
12      }
        return nodesNotWanted;
14  }
```

O Código 3.3 retorna uma lista dos nós/*hosts* que não podem ser utilizados para implantar o microsserviços que será migrado. Tudo isso, a partir do que foi definido como nós/*hosts* de destino.

Para implementar o CoMMAR para Docker Swarm, bem como o componente *Estratégia de Scheduling*, usamos a API Java para o Docker (*Docker Java Client*)[7] na versão *3.5.12* em conjunto com o Java versão oito.

3.7 Considerações Finais

Nesse capítulo foram apresentados o projeto e a implementação do CoMMAR. Primeiramente, foram apresentados os requisitos necessários à implantação. Então, foi proposta uma arquitetura que define o CoMMAR. Os componentes da arquitetura foram projetados e implementados, descrevendo suas relações com os requisitos, e suas particularidades nos ambientes Docker Swarm e Kubernetes.

[7] https://github.com/spotify/docker-client

4 AVALIAÇÃO DE DESEMPENHO

Este capítulo apresenta as etapas de avaliação de desempenho do CoMMAR. Inicialmente, apresentamos os objetivos da avaliação e os experimentos realizados. Por fim, o capítulo apresenta os resultados e uma análise dos experimentos.

4.1 Objetivos

O objetivo dos experimentos é comparar o desempenho do CoMMAR em suas duas implementações: Docker Swarm e Kubernetes. Isto será feito com medições às chamadas realizadas ao CoMMAR.

4.2 Experimentos

As etapas para a avaliação realizadas seguiram o passo-a-passo proposto por Jain (1990) e incluem:

1. *Definir os objetivos* - O objetivo é comparar o desempenho entre o CoMMAR desenvolvido para atuar no Kubernetes, que faz uso de um *scheduler* customizado, e o CoMMAR desenvolvido para o Docker Swarm que usa o *scheduler default* em conjunto com uma estratégia de *scheduling*;

2. *Listar os Serviços do Sistema* - Os serviços providos pelo CoMMAR, tanto no Kubernetes quanto no Docker Swarm, é o de mover microsserviços conteinerizados em tempo de execução, podendo espalhar as réplicas (quando houver) entre os nós/*hosts*;

3. *Escolher as Métricas de Desempenho* - Considerando apenas as execuções com saídas corretas, a métrica usada será o tempo para movimentar um microsserviço entre um nó/*host* de origem para o nó/*host* de destino. Essa métrica foi escolhida para validar a sensibilidade do *cluster* ao impacto causado pela atuação do CoMMAR;

4. *Listar os Parâmetros* - Os parâmetros do sistema que podem afetar o desempenho do CoMMAR são: Velocidade da CPU, velocidade da memória RAM, e o sistema operacional. Estes parâmetros estão descritos na Tabela 3 para o Docker Swarm[1] e na Tabela 4 para o Kubernetes. Quanto aos parâmetros da carga de trabalho temos: número de invocações; e o número de microsserviços conteinerizado.

[1] Há um *host 3* para o Docker Swarm, modificando apenas a memória RAM que é de 2GB em relação ao que é apresentado na Tabela 3.

Tabela 3 – *Hosts* 1 e 2 do Docker Swarm

Parâmetro	**Valor**
Processador	2 CPUs Virtuais
Memória Ram	4 GB
Sistema Operacional	Ubuntu 16.04 LTS

5. *Escolher os Fatores* - Os fatores são os dois CoMMAR (para o Docker Swarm e o Kubernetes); o microsserviço conteinerizado e o número de réplicas de cada microsserviço.

6. *Escolher a Técnica de Avaliação* - Como implementamos o CoMMAR, escolhemos como técnica de avaliação a medição;

7. *Escolher a Carga de Trabalho* - Uma chamada remota ao *cluster* feita pelo CoMMAR irá gerar sucessivas migrações de microsserviços. Fixamos em cinquenta o número de chamadas. Também fizemos cópias dos microsserviços delimitando que cada microsserviço tenha outros quatro iguais. Na prática, o *cluster* sempre tem microsserviços múltiplos de cinco. Ou seja, no cenário do experimento, quando for migrado um microsserviço qualquer, suas cópias também serão movidas; sempre preservando a igualdade do número de réplicas. Por exemplo, no *cluster* que havia dois microsserviços NGINX, cada um com duas réplicas, para realizar o experimento, nós copiamos cada microsserviço multiplicando-os por cinco. Então, teremos para a avaliação de desempenho, nesse cenário, dez microsserviços, cada um com duas réplicas.

8. *Projetar o Experimento* - O experimento foi realizado com cinquenta invocações do CoMMAR. Um *script* foi desenvolvido para realizar as chamadas e um arquivo de *log* foi exportado enquanto o experimento era realizado;

9. *Analisar e Interpretar os Resultados* - Um gráfico com o desempenho será ilustrado para comparar o CoMMAR do Kubernetes e do Docker Swarm;

O serviço provido pela função mover do CoMMAR é o de mover os microsserviços entre os nós/*hosts*. Os retornos esperados são: *mover realizado com sucesso* ou *erro na execução do mover*.

A métrica será o tempo para a execução do Mover, afinal, busca-se comparar o desempenho do CoMMAR no Kubernetes e no Docker Swarm.

Na escolha dos níveis dos fatores, os microsserviços conteinerizados foram: NGINX, que é bastante difundido como *load-balancer* e *API Gateway*; e o weaveworksdemos/user (weaveworks) que é um contêiner customizado construído para o *sock-shop*.

Tabela 4 – *Hosts* 4 e 5 - Kubernetes

Parâmetro	**Valor**
Processador	2 CPUs Virtuais
Memória Ram	2 GB
Sistema Operacional	Ubuntu 16.04 LTS

Sock-shop é um *benchmark µApp* de código aberto que simula um site que vende meias, amplamente usado para validar microsserviços (SAMPAIO et al., 2018). O microsserviço conteinerizado waveworks compõe o *sock-shop*.

Nos experimentos, foram utilizados um contêiner puro (NGINX) e outro customizado (waveworks), o que representa as condições comumente encontradas numa μApp. Esses fatores serão utilizados no Docker Swarm e Kubernetes.

Contêiner puro são contêineres construídos utilizando apenas a imagem base, sem instalar nenhuma biblioteca adicional. Isso significa que o contêiner é o mais leve possível para um microsserviço construído a partir de determinada imagem. O contêiner customizado é formado a partir de uma imagem base e mais algumas bibliotecas podem ser adicionadas, para aumentar a sua robustez. Mas,vale salientar que a customização pode também aumentar o seu tamanho.

Tabela 5 – Fatores e Níveis do CoMMAR

Fatores	**Níveis**
Microsserviços	NGINX, Waveworks
Número de Réplicas	1,2,3
Número de Chamadas	50

Um outro fato importante é que no Docker Swarm, foi implementado a *estratégia de scheduling* para espalhar as réplicas nos nós/*hosts* apresentado na Seção 3.6 como *estratégia de scheduling*. Como visto naquela seção, estamos usando o *scheduler default* do Docker Swarm.

No Kubernetes, foi implementado um novo *scheduler* como apresentado na Seção 3.5. Este algoritmo é capaz de direcionar as réplicas para o nó/*host* de destino. Ressaltamos isso porque, para o Kubernetes, temos mais um parâmetro de carga de trabalho: *scheduler default* e o *scheduler customizado*.

Vale destacar que foram movidos cinco microsserviços a cada chamada, todos com números iguais de réplicas. Esse número foi escolhido devido à capacidade das máquinas em suportar os experimentos. Sobretudo, à medida em que se variava os níveis dos fatores. Afinal, se exigiria mais das máquinas quando se aumentava o número de réplicas dos microsserviços.

Na prática, quando o experimento é executado com um microsserviço conteinerizado *NGINX*, a migração ocorre em todos os microsserviços com a mesma configuração. É

Tabela 6 – Fatores e Níveis do CoMMAR no Kubernetes

Fatores	**Níveis**
Microsserviços	NGINX, Waveworks
Scheduler	Customizado, *Default*
Número de Réplicas	1,2,3
Número de Chamadas	50

importante observar que todos os microsserviços com a mesma configuração também tem o mesmo número de réplicas. As réplicas também eram movidas.

Logo, ao todo, sempre foram executadas duzentas e cinquenta migrações de microsserviços a cada chamada, independentemente da quantidade de réplicas que cada microsserviço possuía.

Por fim, vale observar que na Tabela 6 temos um fator a mais em relação a Tabela 5 e mais dois níveis. Isso ocorreu por conta de uma observação dos primeiros resultados dos experimentos. Portanto, esse fator não estava previsto inicialmente.

4.3 Resultados

A Figura 16 mostra o tempo médio para mover um microsserviço no Docker Swarm e no Kubernetes. No Kubernetes foi utilizado o *scheduler* customizado e no Docker Swarm foi utilizada a estratégia de *scheduling*.

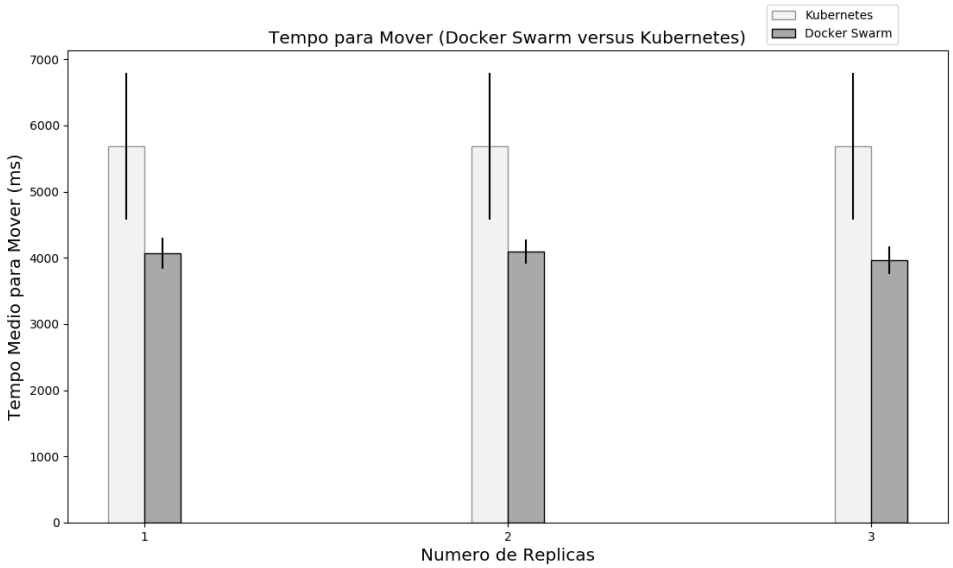

Figura 16 – Tempo Médio do Move - NGINX

Percebe-se que o Docker Swarm teve um tempo médio de execução menor que o Kubernetes.

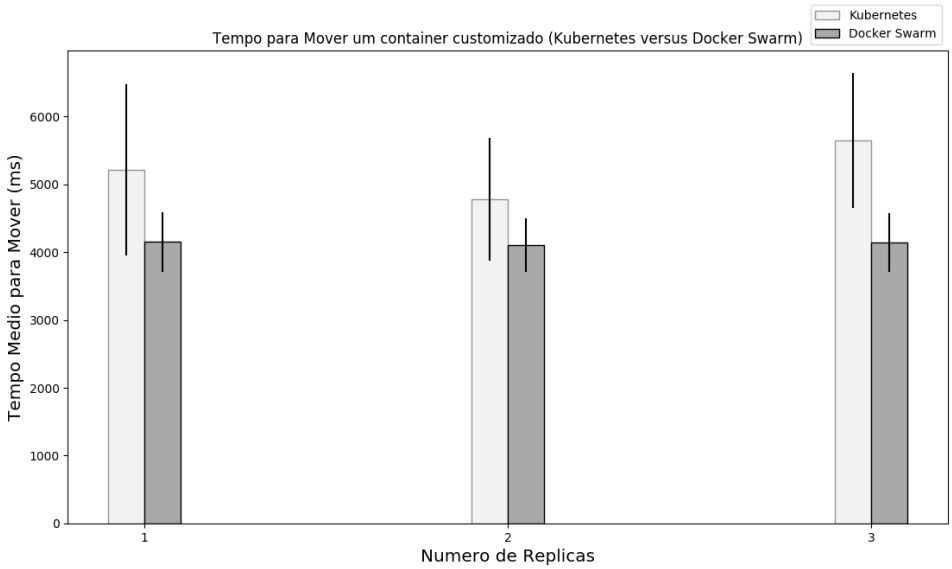

Figura 17 – Tempo para Mover um Container Customizado (waveworks)

A Figura 17 mostra o tempo médio para mover um microsserviço containerizado e customizado (waveworks). Também é comparado o desempenho do Docker Swarm e sua estratégia de *scheduling* e do Kubernetes com o *scheduler* customizado.

Mais uma vez, baseado na Figura 17 temos o Docker Swarm mais rápido que o Kubernetes com o *scheduler* customizado.

Percebe-se nas Figuras 16 e 17 que o tempo para mover quase não se altera em função do número de réplicas. Tanto o Kubernetes quanto o Docker Swarm se mantém com a mesma média de tempo para mover.

Mas o Kubernetes teve um desvio padrão muito maior que o apresentado pelo Docker Swarm. Então, observamos o comportamento das chamadas que realizam a migração dos microsserviços. As Figuras 18, 19 e 20 mostram que o Kubernetes é bem mais volátil. Em suma, essas figuras mostram o tempo de cada chamada do CoMMAR para mover o microsserviço.

Decidimos então, comparar o *scheduler default* do Kubernetes com o *scheduler* customizado do CoMMAR. Fixamos o fator *microsserviço* com o nível *NGINX*. O resultado, ilustrado na Figura 21, mostrou que há uma perda de desempenho ao se utilizar o *scheduler* customizado do CoMMAR. Porém, o *scheduler* customizado é capaz de direcionar às réplicas de cada microsserviço individualmente. Uma das prováveis razões para tal é que o *watch* é um submódulo do *scheduler* customizado que demanda processamento e faz monitoramento, ainda que breve, do *cluster* para garantir a inicializado dos microsserviços após sua migração. Em suma, isso significa mais processamento.

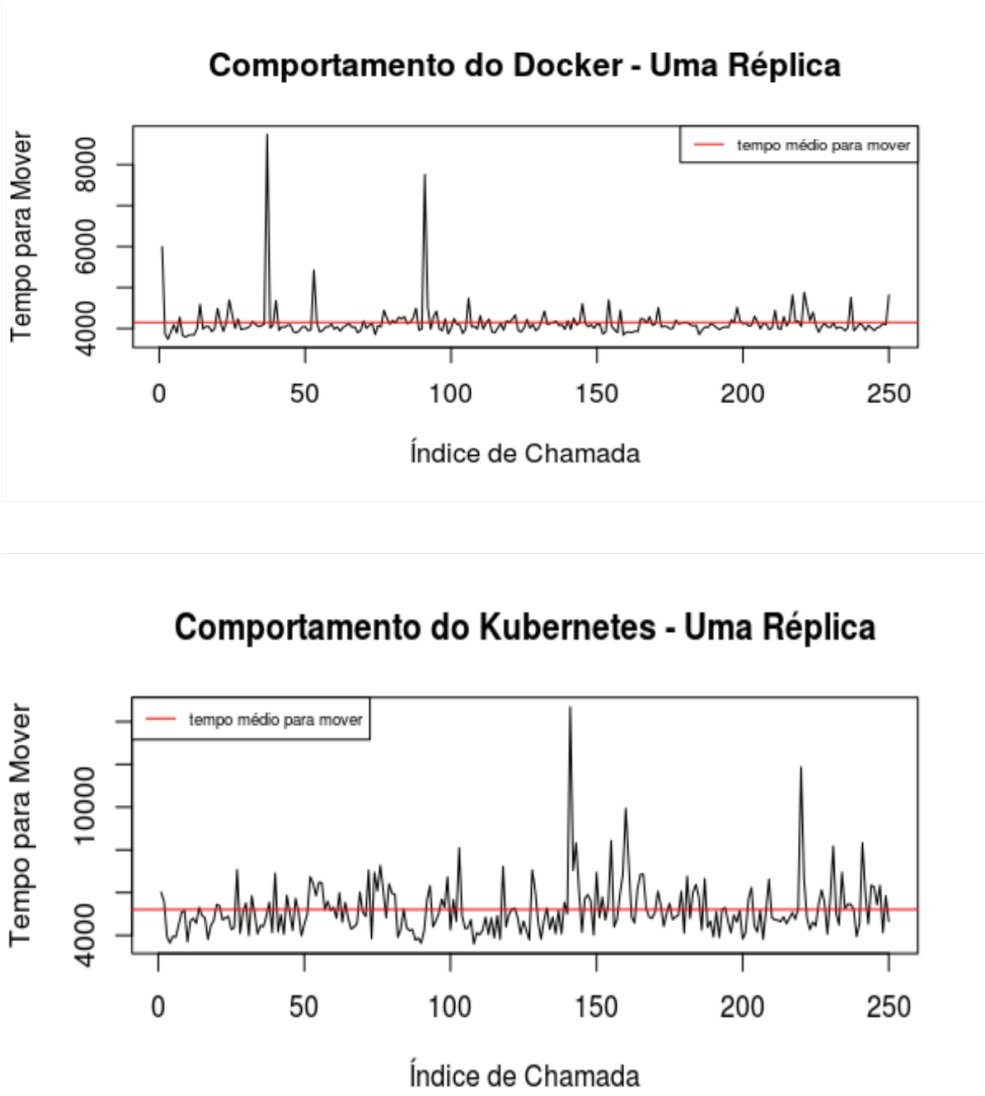

Figura 18 – Migração do waveworks com uma réplica

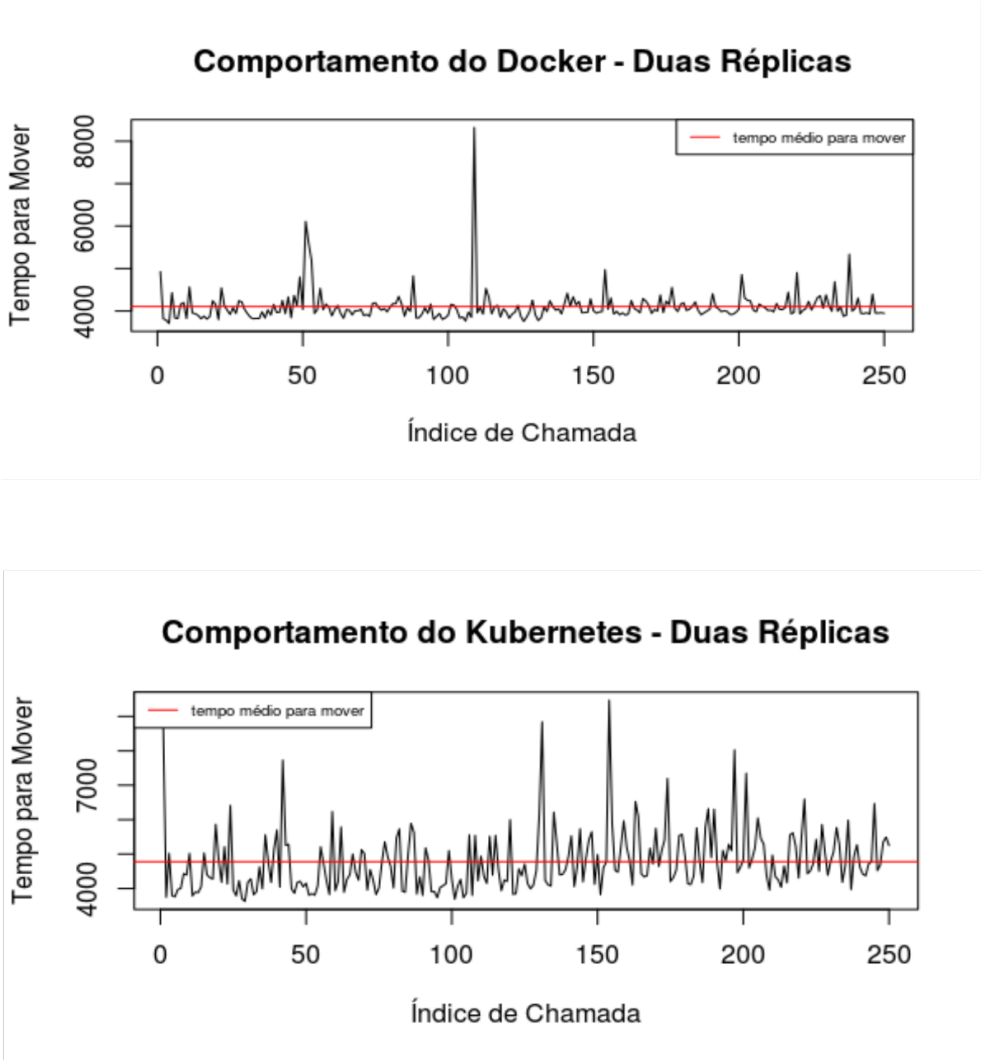

Figura 19 – Migração do waveworks com duas réplicas

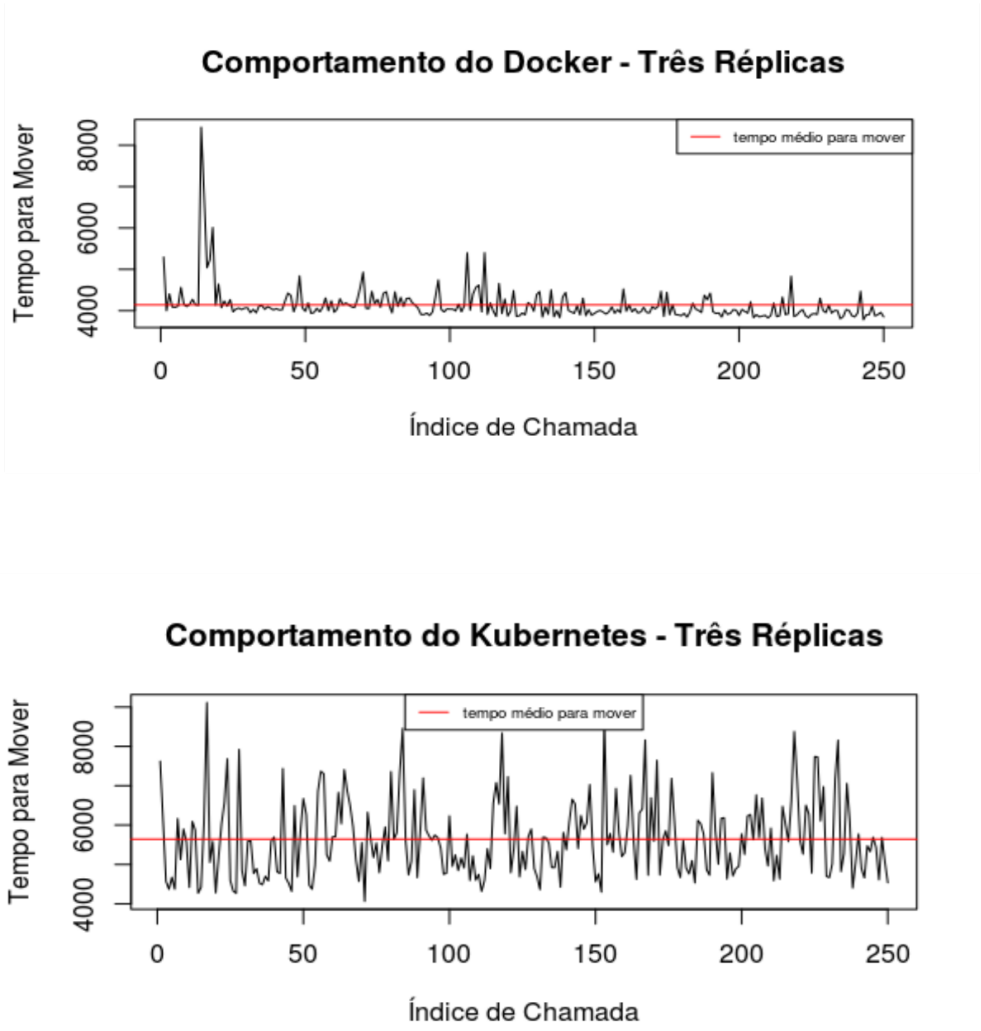

Figura 20 – Migração do waveworks com três réplicas

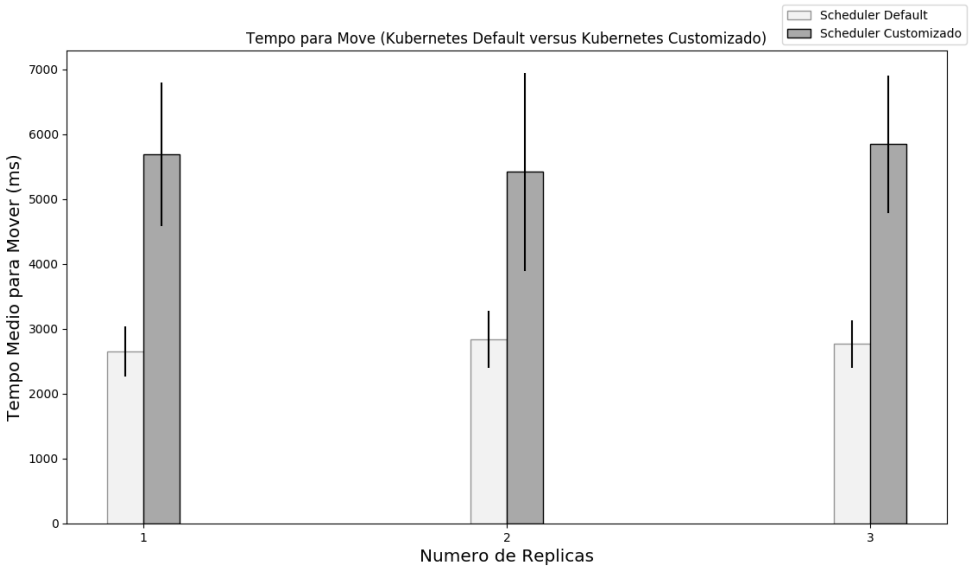

Figura 21 – *Scheduler Default* versus *Scheduler* Customizado

4.4 Considerações Finais

Nesse capítulo apresentamos à avaliação de desempenho do CoMMAR. Após selecionarmos os parâmetros, apresentamos os fatores que impactam no desempenho, e fizemos os experimentos variando os níveis também foram expostos. Os resultados demonstraram que a solução do Docker Swarm é mais rápida que no Kubernetes. E que o CoMMAR no kubernetes tem mais flexibilidade, porém, ao custo do desempenho.

5 TRABALHOS RELACIONADOS

Neste capítulo, serão abordados trabalhos relacionados ao CoMMAR. Para isto, os trabalhos foram organizados em dois grupos: trabalhos sobre escalonamento de contêineres e máquinas virtuais, e adaptação em sistemas distribuídos.

5.1 *Scheduling* de Containers

Ji e Liu (2016) abordam o escalonamento no contexto em que provedores de IaaS (infraestrutura como Serviço) devem atender a qualidade de serviços definida no SLA (*Service Level Agreement*), e.g., *tempo de resposta não deve exceder 100ms*, ou *a vazão não deve ser menor que 200 requisições por segundo*, e assim por diante. Esses provedores devem otimizar o uso dos recursos para garantir a qualidade do serviço e evitar o desperdício.

Neste cenário, foi então proposto um método dinâmico de *deployment* orientado a SLA. Busca-se resolver problemas causados pela frequência em que se faz *deployment* de microsserviços na nuvem sob diferentes condições, analisando a distribuição de recursos no *cluster*. Isso reduziu os custos operacionais, o consumo de recursos e garantiu o atendimento ao SLA.

Kaewkasi e Chuenmuneewong (2017) argumentam que o *scheduler* do Docker é subótimo quando os recursos são não uniformes. Na prática, o algoritmo de *scheduling* do Docker baseado no *round-robin*, funciona bem quando os nós/*hosts* do *cluster* possuem a mesma especificação. O algoritmo, chamado de *spread*, tenta distribuir igualmente os containers através dos nós/*hosts*.

Sendo assim, os autores propuseram o uso de ACO (*Ant Colony Optimization*) para otimizar o *scheduling* do Docker. ACO é um algoritmo meta-heurístico para problemas de otimização e combinação (DORIGO; BLUM, 2005).

Após a implementação do Docker *scheduler* baseado em ACO, o autor demonstrou que conseguiu, através dos experimentos, fazer o *scheduling* 15% mais rápido do que o Docker faz com seu *scheduler default*.

Os trabalhos de Ji e Liu (2016) e Kaewkasi e Chuenmuneewong (2017) visam otimizar a μApp através de melhorias no processo de *deployment*. Diferentemente deles, aqui nos visamos μApp em tempo de execução.

5.2 *Scheduling* de Máquinas Virtuais

Gao et al. (2013) assume que otimizar o *placement* das VMs tem impacto direto no desperdício de recursos e no consumo de energia.

Ele propôs um algoritmo multi-objetivo baseado em ACS (*Ant Colony System*) para lidar com o problema de *placement* de máquinas virtuais.

Os resultados, quando comparados à outras abordagens e outros algoritmos se mostraram bem competitivos.

Mas, esse trabalho visa o desperdício de recursos e o gasto de energia com um *placement* não-ótimo de VMs. Isso tudo em fase de implantação. Este trabalho visa μApp e seu desempenho em tempo de execução, permitindo um rearranjo/reconfiguração do μApp com a redistribuição dos microsserviços através dos nós/*hosts*.

Da mesma forma, Chaisiri, Lee e Niyato (2009) inicia seu trabalho destacando os problemas em otimização de recursos em *cloud provider* via máquinas virtuais. Os autores destacam que os planos oferecidos de pagamento pela infraestrutura/recursos na nuvem, que são dois, apresentam problemas.

O pré-pago, mais barato, necessita de uma reserva de recursos de antemão. Isso pode implicar em dois cenários. Subutilização dos recursos, caso a aplicação possua poucos usuários, ou escassez de recursos caso esses não atenda à demanda da aplicação. Mas, esses problemas podem serem evitados com o plano sob demanda, que é mais caro.

Para evitar esses problemas de recursos subutilizados (*overprovisioning*) ou a escassez dos recursos (*underprovisioning*), foi proposto um algoritmo OVMP *Optimal Virtual Machine Placement* para minimizar os custos sob os planos sob demanda e pré-pago.

O algoritmo apresentou resultados promissores, diminuindo os custos e considerando o *tradeoff* entre os planos. A solução ótima usada pelo OVMP foi obtida através da formulação e resolução de uma programação estocástica (*stochastic integer programming*).

No problema descrito no trabalho, elementos como incerteza são considerados. Afinal, há casos em que não se sabe qual será a necessidade de recursos. Mas, busca-se inferir via cálculos estocásticos qual será a necessidade.

O que difere do trabalho aqui proposto é a previsibilidade na alocação dos recursos/-microsserviços e a ausência de mudanças em tempo de execução.

5.3 Adaptação em Sistemas Distribuídos

Florio (2015) propõe um sistema decentralizado auto adaptativo, chamado Gru, baseado num sistema multi-agente auto organizável. Um conjunto de agentes autônomos tomam decisões locais impactando no comportamento global da aplicação. A adaptação é alcançada pela implementação em cada agente de um *loop* de adaptação baseado no MAPE-K.

No entanto, esse trabalho é parcial. O autor visava ainda evoluir o trabalho para implementar o Gru que é a realização da ideia apresentada.

Já o trabalho de Biyani e Kulkarni (2008) argumenta sobre a necessidade de adaptar sistemas, como os de longa execução. De modo geral, a adaptação acontece adicionando ou removendo algum componente de software dinamicamente.

Isso, segundo o autor, pode levar à um cenário onde, durante a adaptação, processos podem ser compostos por novos e velhos componentes ao mesmo tempo, causando uma sobreposição.

Então, foi proposto uma abordagem para verificar a exatidão de uma adaptação. Também foram introduzidos conceitos para verificar a presença/ausência de falhas. Uma arquitetura de *framework* foi descrita e usada para realizar adaptações.

Este trabalho se diferencia do CoMMAR porque tem o objetivo de garantir a exatidão da adaptação. O CoMMAR, embora altere a configuração do *cluster*, é um componente de um software adaptativo, e não foca na *corretude* da migração de microsserviços. Mas, vale ressaltar que o CoMMAR atua no Docker Swarm e no Kubernetes, e essas ferramentas já tem mecanismos semelhantes para garantir a integridade de seus *clusters*, mesmo após a reconfiguração promovida pelo CoMMAR.

Rajagopalan e Jamjoom (2015) apresentam o *App-Bissect* que é mecanismo semiautomático usado para detectar degradação no desempenho de uma aplicação e aplicar uma adaptação corretiva, revertendo o microsserviço a uma versão prévia.

App-Bissect carrega as dependências descritas por desenvolvedores num arquivo *manifest* que informa as dependências de cada microsserviço. Então, a ferramenta monitora o desempenho da μApp em execução a partir da informação do arquivo *manifest*. Quando *App-Bissect* detecta que a μApp teve seu desempenho degradado após uma atualização, por exemplo, a ferramenta reverte a aplicação, ou parte dela, para uma versão prévia. A reversão não se aplica à toda μApp. Em suma, a reversão privilegia apenas a nova versão do microsserviço e suas dependências.

Apesar do *App-Bissect* modificar a aplicação em tempo de execução, essa ferramenta não trata de melhorar o *placement* dos microsserviços. Afinal, mesmo uma nova versão de um microsserviço podendo impactar no desempenho da μApp, um *placement* melhor de cada microsserviço pode evitar os problemas descritos por Rajagopalan e Jamjoom (2015).

Diferente do *App-Bissect*, o CoMMAR não lida com o impacto gerado pela evolução/atualização dos microsserviços. Isso mostra que as abordagens das duas ferramentas podem se complementar para melhorar o desempenho da μApp.

5.4 Evolução de Microsserviços

Esta subseção se baseia no trabalho de Sampaio et al. (2017), Sampaio et al. (2018). É uma intersecção entre adaptação de microsserviços e *scheduling* de microsserviços containerizados.

O autor propôs uma ferramenta de adaptação de microsserviço em tempo de execução, chamada REMaP (*RuntimE Microsservice Placement*). É um sistema autonômico para fazer à adaptação do μApp em tempo de execução. Esses sistemas, por sua vez, são sistemas capazes de se auto gerenciar (KEPHART; CHESS, 2003) e pode ser separado em sistemas capazes de auto configuração, auto otimização e auto recuperação, entre outros (WHITE et al., 2004). A ferramenta se baseia no MAPE-K (*Monitor, Analyzer, Plan, Executor and Knowledge*) descrita na Seção 2.1.1.

Como a evolução do REMaP, aqui se propôs aperfeiçoamento do *Executor* implementado por Sampaio et al. (2017). CoMMAR é capaz de tratar as réplicas dos microsserviços, direcionando-as individualmente aos nós/*hosts* para serem implantadas.

5.5 Considerações Finais

Nesse capítulo foram apresentados trabalhos relacionados ao CoMMAR. Nós discutimos sobre o *scheduling* de máquinas virtuais e contêineres. Nós também discutimos sobre a adaptação em sistemas distribuídos e finalizamos com a evolução de microsserviços.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta as contribuições do trabalho proposto, as suas limitações e possibilidades de próximos passos.

6.1 Conclusões

A flexibilidade e a abstração promovida por ferramentas de gerenciamento de microsserviços conteinerizados escondem o impacto no desempenho da aplicação baseada em microsserviços (μApp) causado pelo *placement* dos microsserviços e mudanças na μApp em tempo de execução. Para melhorar o desempenho das μApps, Sampaio et al. (2018) propôs o REMaP (*RuntimE Microservice Placement*) para reconfigurar o *placement* dos microsserviços de forma que o novo arranjo dos microsserviços melhore o desempenho das μApps. Porém, o ReMAP não trata os microsserviços com réplicas.

O CoMMAR é responsável por aplicar as reconfigurações ao *cluster* onde está a μApp. Essa ferramenta pode tratar as réplicas dos microsserviços direcionando-as aos nós/*hosts* individualmente.

O CoMMAR é formado por quatro componentes: *Conexão e Setup, Core, Scheduler Customizado e Estratégia de Scheduling*.

O componente *Conexão e Setup* abstrai os componentes *Monitor, Analyzer* e *Planner* do MAPE-K, estabelecendo a conexão com o *cluster* e definindo os dados necessários para que a migração ocorra.

O Componente *Core* usa os dados passados pelo componente *Conexão e Setup* para determinar qual microsserviço deverá ser movido e qual será o nó/*host* de destino. Essas informações são repassadas aos componentes *estratégia de scheduling* e *scheduler customizado* dependendo da ferramenta (Docker Swarm ou Kubernetes) que estiver gerenciando o *cluster*.

O componente *Scheduler Customizado* foi desenvolvido para atuar num *cluster* Kubernetes. Ele é um novo *scheduler* capaz de tratar réplica a réplica a migração dos microsserviços, direcionando-as individualmente aos nós/*hosts* de destino.

O Componente *Estratégia de Scheduling* foi desenvolvido para atuar num *cluster* Docker Swarm. Ele faz uso de primitivas disponibilizadas pelo próprio Docker Swarm para *espalhar* as réplicas dos microsserviços após a migração. Embora não se garanta o *espalhamento* das réplicas isso tende a acontecer, principalmente quando o microsserviço tem muitas réplicas. Afinal, a *Estratégia de Scheduling* atua em combinação ao *spread* do Docker Swarm previamente abordado na Seção 2.6.

Em suma, o uso do CoMMAR pode:

- Melhorar o desempenho das μApps modificando o *Placement* dos microsserviços;

- Reconfigurar o *cluster* não só mudando o *placement* dos microsserviços, mas também espalhando as réplicas dos microsserviços por vários nós/*hosts*; e
- Ser implantado em ambientes MAPE-K para ser o atuador (*Executor*) capaz de tratar as réplicas.

6.1.1 Contribuições

O REMaP foi idealizado a partir da necessidade de adaptar e modificar os arranjos de microsserviços baseado na arquitetura MAPE-K (SAMPAIO et al., 2018).

A principal contribuição no CoMMAR é a capacidade de tratar as réplicas. Essa contribuição está fundamentada em algumas outras contribuições secundárias.

- *Abstração dos Componentes MAPE-K.* Como o trabalho foi fundamentado sobre o *Executor* do MAPE-K, os demais componentes do MAPE-K precisaram ser abstraídos. Resumidamente, o processamento de uma eventual adaptação foi simplificado através de atribuição de dados. Nós atribuímos dados como o nome do microsserviço e o nó/*host* de destino para verificar a atuação do CoMMAR garantindo que essa ferramenta fizesse a migração de microsserviços tratando suas respectivas réplicas;
- *Heurística Simplificada de Reimplantação das Réplicas.* Esta contribuição trata como o *Scheduler Customizado espalha* as réplicas do microsserviço que deve ser migrado nos nós/*hosts* de destino. Nesse caso, a heurística atua usando o nome dos objetos *pods* do *cluster* Kubernetes. No caso do Docker Swarm são usados os próprios recursos dessa ferramenta para *espalhar* as réplicas;
- *Um Scheduler para Kubernetes.* A implementação do CoMMAR para Kubernetes implementou a *heurística simplificada*. Ela foi desenvolvida para atuar no *Scheduler Customizado*. Qualquer microsserviço da μApp pode aplicar esse novo *scheduler*.

6.2 Limitações da Solução

O CoMMAR tem algumas limitações:

- *Scheduler Customizado.* Quando a avaliação de desempenho foi executada, nós notamos que quando sob uma grande carga de trabalho, a implantação de microsserviços e suas réplicas nos nós/*hosts* de destino poderiam falhar. Embora isso não pare a execução da μApp, não é desejável que ocorra. Sob circunstâncias normais onde se migre o microsserviço uma vez, empiricamente, isso não ocorreu.
- *Migração de Microsserviços Stateful.* Migrar microsserviços *stateful* seria algo desafiador. Além de migrar o microsserviço em si, os dados também precisariam migrar. *O que acontece se forem muitos dados? Essa operação levaria quanto tempo? Como*

ocorreria a operação de leitura de dados durante a migração? E quanto a escrita de novos dados na μApp durante a migração? Além disso, Sampaio et al. (2018) cita que as ferramentas Kubernetes e Docker Swarm não possuem mecanismos para migrar os dados.

- *Scheduler Customizado versus Estratégia de Scheduling.* A *Estratégia de Scheduling* não é poderosa como o *Scheduler Customizado*. Desenvolver um *Scheduler Customizado* para Docker Swarm seria o ideal.

6.3 Trabalhos Futuros

A partir do trabalho desenvolvido, alguns trabalhos futuros podem ser definidos:

- *Implantação do Scheduler Customizado no Cluster Kubernetes.* Implantar o *Scheduler Customizado* diretamente no *cluster*. O Kubernetes permite múltiplos *schedulers*. Isso substituiria o *scheduler customizado* como um todo.

- *Múltiplos Schedulers para Kubernetes.* O Kubernetes permite múltiplos *schedulers* no *cluster*. *Schedulers* previamente implantados num *cluster* Kubernetes podem ser usados para determinar *como* o microsserviço e suas réplicas devem migrar e *onde* devem ser implantados. Cada *scheduler* no *cluster* pode ter sua própria característica de como tratar os microsserviços e réplicas.

- *Manutenção da Migração Quando o Microsserviço Escala.* Há um cenário onde um software adaptativo decide realizar uma migração de um microsserviço e, durante essa decisão, a ferramenta de gerenciamento decide escalar o mesmo microsserviço. Isto é um problema sobre qual das duas ações deve ser realizada. O fato é que em ambos os casos busca-se melhorar o desempenho da μApp. Mas, o estudo futuro precisa determinar qual das duas atividades precisa ser realizada para atender melhor a μApp. Esse estudo determinará se uma migração será suficiente para atender a μApp ou se realmente será preciso fazer um escalonamento. Além disso, definirá como essa interversão será feita.

REFERÊNCIAS

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: IEEE. *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*. [S.l.], 2016. p. 44–51.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016.

BIYANI, K. N.; KULKARNI, S. S. Assurance of dynamic adaptation in distributed systems. *Journal of Parallel and Distributed Computing*, Elsevier, v. 68, n. 8, p. 1097–1112, 2008.

BRESCIANI, P.; PERINI, A.; GIORGINI, P.; GIUNCHIGLIA, F.; MYLOPOULOS, J. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, Springer, v. 8, n. 3, p. 203–236, 2004.

CHAISIRI, S.; LEE, B.-S.; NIYATO, D. Optimal virtual machine placement across multiple cloud providers. In: IEEE. *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*. [S.l.], 2009. p. 103–110.

DA, K.; DALMAU, M.; ROOSE, P. A survey of adaptation systems. *International Journal on Internet and Distributed Computing Systems*, v. 2, n. 1, p. 1–18, 2011.

DOCKER. *Docker Home Page*. 2018. Disponível em: <https://www.docker.com/>.

DOCKER-SWARM. *Docker Swarm Mode*. 2018. Disponível em: <https://docs.docker.com/engine/swarm/>.

DORIGO, M.; BLUM, C. Ant colony optimization theory: A survey. *Theoretical computer science*, Elsevier, v. 344, n. 2-3, p. 243–278, 2005.

DRAGONI, N.; GIALLORENZO, S.; LAFUENTE, A. L.; MAZZARA, M.; MONTESI, F.; MUSTAFIN, R.; SAFINA, L. Microservices: yesterday, today, and tomorrow. In: *Present and Ulterior Software Engineering*. [S.l.]: Springer, 2017. p. 195–216.

DUA, R.; RAJA, A. R.; KAKADIA, D. Virtualization vs containerization to support paas. In: IEEE. *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. [S.l.], 2014. p. 610–614.

FAZIO, M.; CELESTI, A.; RANJAN, R.; LIU, C.; CHEN, L.; VILLARI, M. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, IEEE, v. 3, n. 5, p. 81–88, 2016.

FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An updated performance comparison of virtual machines and linux containers. In: IEEE. *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. [S.l.], 2015. p. 171–172.

FLORIO, L. Decentralized self-adaptation in large-scale distributed systems. In: ACM. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2015. p. 1022–1025.

GAO, Y.; GUAN, H.; QI, Z.; HOU, Y.; LIU, L. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *Journal of Computer and System Sciences*, Elsevier, v. 79, n. 8, p. 1230–1242, 2013.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, ACM, v. 40, n. 3, p. 7, 2008.

IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, Citeseer, v. 31, p. 1–6, 2006.

ISMAIL, B. I.; GOORTANI, E. M.; KARIM, M. B. A.; TAT, W. M.; SETAPA, S.; LUKE, J. Y.; HOE, O. H. Evaluation of docker as edge computing platform. In: IEEE. *Open Systems (ICOS), 2015 IEEE Confernece on*. [S.l.], 2015. p. 130–135.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: John Wiley & Sons, 1990.

JAMSHIDI, P.; PAHL, C.; MENDONÇA, N. C.; LEWIS, J.; TILKOV, S. Microservices: The journey so far and challenges ahead. *IEEE Software*, IEEE, v. 35, n. 3, p. 24–35, 2018.

JI, Z.-L.; LIU, Y. A dynamic deployment method of micro service oriented to sla. *International Journal of Computer Science Issues (IJCSI)*, International Journal of Computer Science Issues (IJCSI), v. 13, n. 6, p. 8, 2016.

KAEWKASI, C.; CHUENMUNEEWONG, K. Improvement of container scheduling for docker using ant colony optimization. In: IEEE. *Knowledge and Smart Technology (KST), 2017 9th International Conference on*. [S.l.], 2017. p. 254–259.

KANG, H.; LE, M.; TAO, S. Container and microservice driven design for cloud infrastructure devops. In: IEEE. *Cloud Engineering (IC2E), 2016 IEEE International Conference on*. [S.l.], 2016. p. 202–211.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, IEEE, n. 1, p. 41–50, 2003.

KRATZKE, N. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049*, 2017.

KUBERNETES. *Kubernetes Home Page*. 2018. Disponível em: <https://kubernetes.io/>.

LEWIS, J.; FOWLER, M. Microservices: a definition of this new architectural term. *MartinFowler. com*, v. 25, 2014.

MALHOTRA, L.; AGARWAL, D.; JAISWAL, A. Virtualization in cloud computing. *J. Inform. Tech. Softw. Eng*, v. 4, n. 2, 2014.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: *CLOSER (1)*. [S.l.: s.n.], 2016. p. 137–146.

PRIKLADNICKI, R. et al. Munddos: um modelo de referência para desenvolvimento distribuído de software. Pontifícia Universidade Católica do Rio Grande do Sul, 2003.

RAJAGOPALAN, S.; JAMJOOM, H. App-bisect: Autonomous healing for microservice-based apps. In: USENIX ASSOCIATION. *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*. [S.l.], 2015. p. 16–16.

RICHARDS, M. *Microservices vs. service-oriented architecture*. [S.l.]: O'Reilly Media, 2015.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, ACM, v. 4, n. 2, p. 14, 2009.

SAMPAIO, A. R.; KADIYALA, H.; HU, B.; STEINBACHER, J.; ERWIN, T.; ROSA, N.; BESCHASTNIKH, I.; RUBIN, J. Supporting microservice evolution. In: IEEE. *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. [S.l.], 2017. p. 539–543.

SAMPAIO, A. R.; RUBIN, J.; BESCHASTINIKH, I.; ROSA, N. *Improving Microservice-based Applications with Runtime Placement Adaptation*. 2018.

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.

SEO, K.-T.; HWANG, H.-S.; MOON, I.-Y.; KWON, O.-Y.; KIM, B.-J. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, v. 66, n. 105-111, p. 2, 2014.

TAYLOR, R. N.; MEDVIDOVIC, N.; OREIZY, P. Architectural styles for runtime software adaptation. In: IEEE. *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. [S.l.], 2009. p. 171–180.

THÖNES, J. Microservices. *IEEE Software*, IEEE, v. 32, n. 1, p. 116–116, 2015.

VALETTO, G.; KAISER, G. Using process technology to control and coordinate software adaptation. In: IEEE. *Software Engineering, 2003. Proceedings. 25th International Conference on*. [S.l.], 2003. p. 262–272.

VILLAMIZAR, M.; GARCÉS, O.; CASTRO, H.; VERANO, M.; SALAMANCA, L.; CASALLAS, R.; GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. *Computing Colombian Conference (10CCC), 2015 10th*. [S.l.], 2015. p. 583–590.

VILLAMIZAR, M.; GARCES, O.; OCHOA, L.; CASTRO, H.; SALAMANCA, L.; VERANO, M.; CASALLAS, R.; GIL, S.; VALENCIA, C.; ZAMBRANO, A. et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: IEEE. *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. [S.l.], 2016. p. 179–182.

WEYNS, D.; MALEK, S.; ANDERSSON, J. Forms: a formal reference model for self-adaptation. In: ACM. *Proceedings of the 7th international conference on Autonomic computing*. [S.l.], 2010. p. 205–214.

WHITE, S. R.; HANSON, J. E.; WHALLEY, I.; CHESS, D. M.; KEPHART, J. O. An architectural approach to autonomic computing. In: IEEE. *Autonomic Computing, 2004. Proceedings. International Conference on*. [S.l.], 2004. p. 2–9.

ZHAO, D.; MANDAGERE, N.; ALATORRE, G.; MOHAMED, M.; LUDWIG, H. Toward locality-aware scheduling for containerized cloud services. In: IEEE. *Big Data (Big Data), 2015 IEEE International Conference on*. [S.l.], 2015. p. 263–270.

ZIMMERMANN, O. Microservices tenets. *Computer Science-Research and Development*, Springer, v. 32, n. 3-4, p. 301–310, 2017.