K8ShMiR: um *framework* para replicação de máquinas de estado em contêineres gerenciados pelo Kubernetes

Lucas Borges Fernandes

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2021

Lucas Borges Fernandes

K8ShMiR: um *framework* para replicação de máquinas de estado em contêineres gerenciados pelo Kubernetes

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Lásaro Camargos

Uberlândia
2021

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

F363 2021

Fernandes, Lucas Borges, 1994-
K8ShMiR [recurso eletrônico] : Um framework para replicação de máquinas de estado em contêineres gerenciados pelo Kubernetes / Lucas Borges Fernandes. - 2021.

Orientador: Lásaro Jonas Camargos.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2022.583
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Camargos, Lásaro Jonas,1978-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Mestrado Acadêmico, 26/2021, PPGCO				
Data:	25 de novembro de 2021	Hora de início:	09:00	Hora de encerramento:	11:15
Matrícula do Discente:	11912CCP017				
Nome do Discente	Lucas Borges Fernandes				
Título do Trabalho:	K8ShMiR: Um framework para replicação de máquinas de estado em contêineres gerenciados pelo Kubernetes				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Paulo Rodolfo da Silva Leite Coelho - FACOM/UFU; Odorico Machado Mendizabal - INE/UFSC e Lásaro Jonas Camargos - FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Odorico Machado Mendizabal - Florianópolis/SC; Paulo Rodolfo da Silva Leite Coelho e Lásaro Jonas Camargos - Uberlândia/MG. O discente participou da cidade de Uberlândia/MG.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Lásaro Jonas Camargos, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos,

conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Lásaro Jonas Camargos**, **Professor(a) do Magistério Superior**, em 25/11/2021, às 14:46, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Odorico Machado Mendizabal**, **Usuário Externo**, em 25/11/2021, às 16:02, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Paulo Rodolfo da Silva Leite Coelho**, **Professor(a) do Magistério Superior**, em 26/11/2021, às 12:34, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3198329** e o código CRC **BF458FBB**.

Referência: Processo nº 23117.080215/2021-27 SEI nº 3198329

Dedico este trabalho aos meus pais Danilo e Rosana.

Agradecimentos

Resumo

Mecanismos para virtualização de infraestruturas de computação, como por exemplo máquinas virtuais e contêineres, são parte fundamental de sistemas de computação modernos. Suas características versáteis e baratas do ponto de vista computacional e financeiro permitiram que novas arquiteturas de sistemas fossem popularizadas. Uma delas são os microsserviços, onde módulos de um *software* executam em contêineres diferentes, mas que em conjunto funcionam como uma única aplicação. Neste trabalho, apresentamos uma arquitetura para replicação de máquinas de estado em sistemas baseados em contêineres, provendo garantias de tolerância a falhas de forma pouco intrusiva para a aplicação replicada e seus usuários. Essa arquitetura, além de pouco intrusiva é também extensível, de forma que pode ser instanciada de diversas formas e com tecnologias variadas. Após a descrevermos, apresentamos uma de suas possíveis implementações com um *framework* nomeado K8ShMiR, que funciona com contêineres Docker no Kubernetes, um orquestrador de contêineres. Para garantias de difusão totalmente ordenada de mensagens, utilizamos o *framework* Atomix. Por fim, realizamos testes integrados com um *cluster* Kubernetes local para validar que nossa implementação funciona.

Palavras-chave: Replicação de Máquinas de Estado, Contêiner, Difusão atômica, Microsserviços, Tolerância a falhas, Kubernetes.

Abstract

Mechanisms for computing infrastructure virtualization, such as virtual machines and containers, are a fundamental part of modern computing systems. Their versatility and low cost characteristics from a computational and financial standpoint have allowed new system architectures to be popularized. One of them are microservices, small pieces of software that run in different containers, while working together as a single application. In this work, we present an architecture for state machine replication in container-based systems, providing non-intrusive fault tolerance guarantees for the replicated application and its users. This architecture, in addition to being little intrusive, is also extensible, so that it can be instantiated in different ways and with varied technologies. After describing it, we present one of its possible implementations with a framework named K8ShMiR, which works with Docker containers in Kubernetes, a container orchestrator. To guarantee total order delivery of messages, we use the Atomix framework. Finally, we executed integration tests with a local Kubernetes cluster to validate that our implementation works.

Keywords: State Machine Replication, Container, Atomic Broadcast, Microservices, Fault Tolerance, Kubernetes.

Lista de ilustrações

Lista de siglas

API *Application Programming Interface*

CNCF *Cloud Native Computing Foundation*

CLI *Command line interface*

CAIUS Coordenação via Serviço no Kubernetes

DNS *Domain Name System*

DORADO *Ordering Over Shared Memory*

FIFO *First in, First out*

JVM *Java Virtual Machine*

ONF *Open Network Foundation*

PaaS *Platform as a Service*

RPC *Remote Procedure Call*

SMR *State Machine Replication*

SDN *Software Defined Networks*

SOA *Service-oriented architecture*

VM *Virtual Machine*

Lista de algoritmos

Sumário

Capítulo 1

Introdução

A construção e disponibilização de aplicações de forma distribuída, ou seja, em múltiplos processos que podem estar fisicamente separados, possibilita a obtenção de escalabilidade e tolerância a falhas, requisitos cada vez mais imprescindíveis na construção de aplicações disponibilizadas na Internet. Em termos de escalabilidade, a distribuição permite aproximar o serviço de seus usuários (escalabilidade geográfica (STEEN; TANENBAUM, 2017)) e dividir a carga de trabalho entre diversos componentes e obter uma maior vazão de operações (escalabilidade computacional (STEEN; TANENBAUM, 2017)). Em termos de tolerância a falhas, a distribuição permite a criação de réplicas de um serviço, tal que um serviço possa ser fornecido mesmo quando algumas réplicas estejam inoperantes. Uma das técnicas usadas neste sentido é a *State Machine Replication (SMR)*, em que sistemas são definidos como máquinas de estados determinísticas em que as transições são disparadas pela recepção de mensagens, implicando que a entrega confiável e ordenada de mensagens para múltiplas instâncias da máquina de estados resultará na replicação do estado do sistema. Dessa forma, a falha de alguns processos é tolerada, pois outros podem assumir no lugar sem perda de informação. O modelo de *SMR* é utilizado na academia (BURROWS, 2006; HUNT et al., 2010) e em grandes projetos na indústria[1], o que mostra sua relevância no ecossistema de desenvolvimento de *softwares*.

A disponibilização de sistemas na Internet é uma tarefa complexa, porém, ao longo dos anos, novas ferramentas e padrões de arquitetura auxiliaram no processo. Um exemplo é a virtualização de infraestruturas de computação, uma estratégia que possibilita que aplicações sejam disponibilizadas de forma agnóstica a servidores (M.; KANNAN, 2014), seja na forma de máquinas virtuais ou de contêineres. Em outras palavras, uma mesma aplicação pode executar em diferentes ambientes sem a necessidade de configurações distintas. Com essa virtualização, abriu-se um leque de possibilidades para arquiteturas de aplicações, como por exemplo os microsserviços (VURAL; KOYUNCU; GUNEY, 2017). Estes são "pequenos"*softwares* que residem em diferentes contêineres e que, ao trabalharem em conjunto, funcionam de forma coesa como uma única aplicação.

[1] https://etcd.io, https://ratis.apache.org/

Sistemas baseados em microsserviços têm uma complexidade inerente. Além da possibilidade dos contêineres residirem em diferentes servidores, eles também podem reiniciar, mudar de servidor, deixar de existir, ou serem adicionados ao *cluster*, tudo em tempo de execução (BASKARADA; NGUYEN; KORONIOS, 2020). Nesse cenário, mecanismos para descoberta de novos contêineres e conhecimento de seus endereços atuais são necessários. Para resolver estes problemas, ferramentas de orquestração de contêineres foram criadas (CASALICCHIO, 2019), sendo o Kubernetes provavelmente um dos mais conhecidos.

Implementar *SMR* em um ecossistema de microsserviços é um problema difícil, pois, além da complexidade inerente do modelo, é preciso lidar com particularidades da infraestrutura, como por exemplo a mobilidade de seus contêineres que dificulta tarefas simples como a troca de mensagens entre réplicas. Acoplar uma solução de *SMR* a um ecossistema existente de forma pouco intrusiva é outro problema difícil, pois microsserviços geralmente apresentam um número elevado de integrações entre serviços (VUČKOVIĆ, 2020), além de, em alguns casos, executarem em cima de ferramentas de orquestração de contêineres, como por exemplo o Kubernetes, que possuem uma *Application Programming Interface* (API) própria para disponibilização e execução de aplicações[2].

1.1 Proposta

Neste trabalho, propomos uma arquitetura para implementação da SMR em aplicações disponibilizadas em contêineres. A instanciação da arquitetura assume o uso do orquestrador de conteineres Kubernetes e é denominada K8ShMiR, *Kubernetes State Machine Replication*. K8ShMiR utiliza o modelo de rede do Kubernetes, além dos mecanismos para gerenciamento do ciclo de vida de contêineres, proporcionando garantias necessárias para aplicações modernas ao mesmo tempo que tolerância a falhas.

1.2 Objetivos e Desafios

O objetivo principal deste trabalho é a construção de uma ferramenta para *SMR* que funcione em contêineres no Kubernetes, e seja, sempre que possível, transparente para a aplicação e seus usuários. Para alcançar este objetivo, utilizamos ferramentas e padrões de projeto utilizados na indústria que possibilitaram que a aplicação se integrasse com o *framework* K8ShMiR sem a necessidade de muitas alterações. Como objetivo secundário, buscamos entregar uma ferramenta eficiente na utilização de recursos computacionais, como por exemplo memória, processamento, além de baixa latência e espera por E/S.

Garantir um nível mínimo de transparência ao mesmo tempo que eficiência na utilização de recursos é um grande desafio. Outro desafio, não menos importante, é a entrega

[2] https://kubernetes.io/docs/tutorials/kubernetes-basics/

de uma ferramenta para *SMR* que suporta falhas em contêineres. Nesse cenário, é preciso que mecanismos para recuperação de estado existam.

1.3 Hipótese

É possível construir uma ferramenta simples para implementação de *SMR* em ambientes de microsserviços baseados em contêineres, com baixo custo extra no desenvolvimento dos serviços.

1.4 Contribuição do Trabalho

Este trabalho contribui para o campo de sistemas distribuídos ao propor uma ferramenta para *SMR* que funciona em ambientes produtivos modernos. Entendemos que a união entre teoria e prática neste campo possui muito valor, pois, alguns problemas de desenvolvimento de *software* são cruciais para o funcionamento das soluções propostas. Alguns exemplos de problemas são: mecanismos de recuperação de falhas, impacto de decisões arquiteturais no desempenho, linguagens de programação, testes, entre outros. Neste trabalho, contribuímos com o estado da arte ao oferecer um *framework* que enfrentou alguns destes problemas que, na nossa visão, devem ser levados em consideração no momento de planejar uma aplicação distribuída.

Outra contribuição é a utilização de ferramentas consolidadas na indústria, como por exemplo o *Docker*[3], *Helm*[4], e *Atomix*[5]. Com isso, entendemos que diminuímos a distância entre a academia e a indústria de desenvolvimento de *software*, ao demonstrar que é possível construir uma solução de replicação de estado com desempenho aceitável utilizando ferramentas de mercado. Por fim, disponibilizamos o *framework* K8ShMiR gratuitamente em um repositório do GitHub[6]. Entendemos que ela pode ser utilizada pela comunidade para a construção de novas aplicações, além da melhoria contínua que contribuições *Open Source* podem oferecer.

1.5 Organização da Dissertação

Organizamos esta dissertação da seguinte forma: O Capítulo 2 explica os fundamentos teóricos necessários para o entendimento do problema que resolvemos, além dos conceitos que utilizamos durante o desenvolvimento da nossa ferramenta. Em seguida, no Capítulo 3, revisamos e posicionamos trabalhos correlatos em relação a esta proposta. No Capítulo 4, explicamos em detalhes a nossa ferramenta, desde suas ideias fundamentais, até trechos

[3] https://www.docker.com/
[4] https://helm.sh/
[5] https://github.com/atomix
[6] https://github.com/lucasbfernandes/k8shmir

de pseudocódigo com comentários. Também mencionamos como desenvolvemos testes integrados, que serviram para comprovar que nossa solução estava de fato correta. Por fim, no Capítulo 5, concluímos a dissertação com comentários acerca dos resultados, além de discutir direções para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados os conceitos necessários para o entendimento do *framework* **K8ShMiR**. Primeiramente, na Seção 2.1, definimos o modelo computacional usado, em termos de comunicação, sincronismo e falhas. Em seguida, nas Seções 2.2 até 2.5, explicamos os conceitos de ordenação de mensagens e consenso distribuído, além de exemplificá-los com a descrição do algoritmo Raft e o modelo de replicação de máquinas de estado (SMR). Por fim, a partir da Seção 2.6 discorreremos sobre virtualização de infraestruturas de computação, Kubernetes, e também apresentaremos o *framework* Atomix para criação de aplicações replicadas.

2.1 Modelo do sistema

Neste trabalho, entendemos que problemas enfrentados por sistemas computacionais podem ser classificados baseando-se no nível em que se apresentam. O nível mais básico é o de *falha*, que pode ser entendida como um erro no desenvolvimento do sistema, seja na forma de *bugs* ou problemas de fabricação. Uma falha pode levar a um estado incorreto da aplicação, fora do que foi especificado; este estado incorreto é um *erro*. Ao se externalizar, uma erro se manifesta como um *defeito* (AVIZIENIS et al., 2004).

Consideramos a possibilidade da existência de um número infinito de processos replicados. Estes podem apresentar falhas que levam a defeitos a qualquer instante. Existem vários tipos de defeitos que podem, por exemplo, ser consequência de uma corrupção de disco rígido, ou até mesmo de intervenções maliciosas.

Estes processos comunicam-se por troca de mensagens, uma premissa realista se considerarmos o desenvolvimento de sistemas utilizando o padrão *Service-oriented architecture* (SOA) (PERREY; LYCETT, 2003). Canais de comunicação podem sofrer vários tipos de defeitos, como perda e corrupção de mensagens. Contudo, corrupções são detectáveis e levam ao descarte de mensagens, que podem ser retransmitidas.

Quanto ao sincronismo, consideramos que operações podem ter atraso indeterminado, caso haja problemas com o relógio, mas funcionam de forma síncrona a maior parte do

tempo. Nas seções a seguir definiremos cada premissa do sistema em mais detalhes.

2.1.1 Modelo *crash-recovery* para processos com falha

Neste trabalho assumimos que processos podem se recuperar de defeitos e, portanto, devemos apresentar soluções para recuperação do estado deles. É importante ressaltar que não consideramos defeitos bizantinos (i.e., processos que podem tomar ações maliciosas e que desviam do algoritmo esperado) (LAMPORT; SHOSTAK; PEASE, 1982). Consideramos esta última premissa realista pois todos os processos executam em uma rede privada, protegida pelos mecanismos de segurança usuais.

2.1.2 Canais de comunicação *fair-loss*

Assumimos canais de comunicação *fair-loss*, que garantem que mesmo após uma perda de mensagens devido a falhas, estas falhas são temporárias e, se mensagens forem retransmitidas, serão entregues em algum momento (STEEN; TANENBAUM, 2017). Esta premissa é realista em termos de redes reais e é a base da garantia de confiabilidade dentro das sessões TCP/IP. Em resumo, neste trabalho assumimos um modelo onde a probabilidade de falha existe com um valor maior que 0. Na prática, isto significa que uma mensagem enviada para um processo pode não chegar até ele, seja por uma falha de *software* ou uma partição de rede, mas que dado um período de tempo, uma das retransmissões chegará ao destino.

2.1.3 Sistema parcialmente síncrono

Neste trabalho assumimos que mensagens podem ser atrasadas por algum período arbitrário sem nenhuma garantia de espera máxima, ao mesmo tempo que durante certos períodos as mensagens podem seguir um modelo síncrono, onde um atraso máximo existe, mesmo que não seja conhecido, de forma que o avanço do estado do processo replicado seja possível. Esta premissa é fundamental para o nosso trabalho devido a prova de impossibilidade FLP(FISCHER; LYNCH; PATERSON, 1985), que determina que é impossível conceber um algoritmo determinístico para consenso distribuído considerando um modelo totalmente assíncrono e com possibilidade de falhas de processos.

2.2 Estratégias para ordenação de mensagens

Além das abstrações de baixo nível para envio e recepção de mensagens, primitivas mais abstratas para a comunicação podem ser construídas para facilitar o desenvolvimento de sistemas. Estas primitivas de comunicação em grupo podem ser qualificadas pelas garantias providas quanto a ordem de entrega de mensagens, que se provam muito

úteis para sistemas distribuídos com replicação de estado e tolerantes à falhas. Como um ponto de partida, podemos dividir estas primitivas em dois grupos, o de Difusão Não-confiável (*Unreliable Broadcast*) e o de Difusão Confiável (*Reliable Broadcast*) (CACHIN; GUERRAOUI; RODRIGUES, 2011b). O primeiro não oferece garantias quanto ao recebimento da mensagem pelos destinatários, enquanto o segundo garante que, caso um processo (com ou sem falhas) receba uma mensagem, então todos os processos destinatários sem falha do sistema também a receberão em algum momento. Esta definição é encontrada na literatura sob o nome de Difusão Confiável Uniforme (*Uniform Reliable Broadcast*) e é descrita formalmente pelas seguintes propriedades (CACHIN; GUERRAOUI; RODRIGUES, 2011b):

- **Validade:** Caso um processo sem falhas *p1* envie uma mensagem m para um processo sem falhas *p2*, então este a receberá em algum momento.
- **Integridade:** Caso um processo qualquer *p1* receba uma mensagem m de um remetente *p2*, então m foi enviada previamente por *p2*.
- **Não-duplicação:** Nenhuma mensagem é entregue mais de uma vez.
- **Acordo:** Caso um processo qualquer (com ou sem falhas) receba uma mensagem m, então todos os processos corretos do sistema também irão receber a mesma mensagem m em algum momento.

Esta afirmação é um pouco mais forte do que a de Difusão Confiável (*Reliable Broadcast*), também encontrada na literatura, que não garante que processos sem falha receberão mensagens entregues a processos faltosos. Neste trabalho, assumiremos o modelo Uniforme como primitiva base para garantias de entrega de mensagens.

Nesta seção, iremos apresentar extensões à garantia de Difusão Confiável Uniforme. Em alguns casos, a entrega por si só não é suficiente, e a ordem das mensagens possui um papel decisivo no resultado das operações do sistema. Portanto, a seguir iremos apresentar alguns modelos de difusão confiável com ordenação de mensagens.

2.2.1 Ordenação FIFO

O modelo de difusão de mensagens *First in, First out* (FIFO), assim como o nome sugere, funciona formalmente segundo a seguinte propriedade (DÉFAGO; SCHIPER; URBÁN, 2004):

- Caso um processo qualquer envie uma mensagem *m1* antes de enviar outra mensagem *m2*, então nenhum processo receberá a mensagem *m2* sem ter antes recebido a mensagem *m1*.

Apesar disso, nenhuma garantia quanto a ordenação entre mensagens emitidas por dois processos diferentes é dada. Em outras palavras, é possível que uma mensagem *m3* emitida em um instante de tempo anterior ao da mensagem *m1* seja entregue primeiro a alguns destinatários e por último para outros, caso ambas tenham sido enviadas por processos diferentes.

2.2.2 Ordenação Causal

A ordenação causal especifica que, caso uma mensagem preceda outra em termos de causalidade, então ela deve ser entregue nesta mesma ordem em todos os processos destinatários. Para um melhor entendimento desta estratégia, é necessário explicar o que define a precedência de dois eventos (mensagens) sob esta ótica. Portanto, uma mensagem *m1* precede *m2* (denotado $m1 \implies m2$) em causalidade caso qualquer uma das afirmações abaixo seja verdadeira (CACHIN; GUERRAOUI; RODRIGUES, 2011b):

1. Algum processo p emitiu a mensagem *m1* antes de emitir a mensagem *m2* (Ordenação FIFO).

2. Algum processo p recebeu a mensagem *m1* e depois emitiu a mensagem *m2*. Neste caso, podemos afirmar que a mensagem *m1* pode ter causado a mensagem *m2*.

3. Existe uma mensagem *m3* tal que $m1 \implies$ m3 e $m3 \implies$ m2 (Transitividade).

Esta relação de potencial causalidade é capturada pela relação "aconteceu-antes"(*happened-before*), definida por Lamport (LAMPORT, 1978). Mais formalmente, as propriedades da ordenação causal são as mesmas da Difusão Confiável Uniforme, acrescentadas da propriedade de Entrega Causal definida a seguir:

- Para toda mensagem $m1$ que possui a relação "aconteceu-antes"com outra mensagem $m2$ (i.e., $m1 \implies m2$), nenhum processo recebe a mensagem $m2$ sem antes ter recebido a mensagem $m1$.

2.2.3 Ordenação Total (Difusão Atômica)

A ordenação causal explicada acima apenas garante a precedência entre mensagens envolvidas pelo princípio da causalidade. Em outras palavras, mensagens que não se relacionam sob esta propriedade não possuem restrições na sua ordem de entrega para os destinatários (DÉFAGO; SCHIPER; URBÁN, 2000). Por isso, podemos dizer que a ordenação causal é apenas parcial ao permitir que algumas mensagens sejam entregues em ordens diferentes.

Em contrapartida, a ordenação total, ou difusão atômica, é mais restrita ao especificar que todas as mensagens devem ser entregues na mesma ordem em todos os processos

destinatários. Como pode ser esperado, os custos computacionais para alcançá-la são mais altos, como veremos nas próximas seções. Algumas aplicações deste modelo são as *Blockchains* (CASINO; DASAKLIS; PATSAKIS, 2019), e aplicações que utilizam o modelo de SMR, que é utilizado pelo nosso trabalho e que também será explicado adiante neste capítulo. A ordenação total, ou difusão atômica, pode ser definida formalmente a partir das mesmas propriedades da Difusão Confiável Uniforme, acrescidas pela seguinte propriedade de Ordenação Total:

- Sejam $m1$ e $m2$ quaisquer 2 mensagens, e suponha que $p1$ e $p2$ são quaisquer 2 processos (com ou sem defeitos) que receberam $m1$ e $m2$. Se $p1$ recebe $m1$ antes de $m2$, então $p2$ também recebe $m1$ antes de $m2$.

É importante mencionar que a difusão atômica em sua definição pura é ortogonal aos conceitos apresentados anteriormente. Em outras palavras, a sua única garantia é de que todas as mensagens sejam entregues na mesma ordem, sem impor nenhuma restrição qualitativa, possibilitando que mensagens sejam entregues sem respeitar as ordenações FIFO ou Causal, por exemplo. Todavia, em alguns casos, como o deste trabalho, a ordenação FIFO é um requisito, fazendo ser necessária a criação de extensões a ordenação total a fim de garantir comportamentos específicos. Um exemplo na literatura é a Ordenação FIFO + Total, que especifica que a ordenação temporal de mensagens de um mesmo processo de origem devem ser respeitadas na ordem final de entrega aos destinatários (DÉFAGO; SCHIPER; URBÁN, 2004).

Uma das soluções para este problema é a adoção de algoritmos onde, em determinados períodos de tempo, um processo é responsável por distribuir as mensagens para todos os outros, inclusive para si mesmo. Este processo, ou nó, é costumeiramente chamado de Sequenciador ou Líder (DÉFAGO; SCHIPER; URBÁN, 2000), e ele facilita a tarefa de ordenação de mensagens ao ser o processo encarregado por este objetivo. O problema é que se este processo apesenta um defeito o algoritmo pode não mais progredir. Defeitos nos nós destinatários também devem ser tratadas e, portanto, algum mecanismo para tolerância a falhas deve ser utilizado. Neste trabalho, focaremos no mecanismo de consenso distribuído para prover ordenação total de mensagens de forma tolerante a falhas. Outras estratégias que não utilizam consenso distribuído foram apresentadas na literatura (DÉFAGO; SCHIPER; URBÁN, 2004) mas fogem do escopo deste trabalho.

2.3 Consenso distribuído

Diversos problemas em computação distribuída podem ser reduzidos ao problema do consenso distribuído, em que diversos processos devem concordar em algum valor em determinado instante (WOLFRAM, 2021). A dificuldade na resolução do consenso está no fato de que estes processos podem ou não apresentar defeitos e que eles se comunicam

por troca de mensagens sobre enlaces de rede também sujeitos a defeitos, que podem levar a atrasos de duração indeterminada na entrega das mensagens. Logo, algoritmos para o problema do consenso devem ser capazes de lidar com diversos cenários de erro, ou melhor, devem ser tolerantes a falhas.

No problema do consenso, um ou mais processos propõem valores e, dentre os valores propostos, um é decidido por todos os processos. Mais formalmente, um algoritmo para consenso distribuído deve atender às seguintes propriedades (CACHIN; GUERRAOUI; RODRIGUES, 2011a):

- **Término** - Todo processo (com ou sem defeitos) tomará uma decisão quanto ao valor em algum momento;
- **Validade** - Se todos os processos propuseram um valor, então todos os não defeituosos devem escolher este valor;
- **Integridade** - Todo processo (com ou sem defeito) deve escolher no máximo um valor, e esse valor deve ter sido proposto por algum processo envolvido no algoritmo;
- **Acordo** - Todos os processos que tomarem uma decisão, devem decidir pelo mesmo valor.

Para garantir a propriedade de término em caso de defeitos, algoritmos de consenso distribuído fazem uso de quóruns, colocando como pré-condição para algumas ações a concordância de uma maioria dos processos. Em outras palavras, em um sistema com N processos, pelo menos $\left\lceil \frac{N}{2} + 1 \right\rceil$ devem concordar com uma ação antes de sua execução ser autorizada. Por isso, é possível garantir que o resultado desta decisão será propagado para instâncias futuras do algoritmo, pois dois quóruns diferentes sempre terão uma interseção. No Raft, por exemplo, este mecanismo é utilizado para eleições de Líder (estado semelhante ao do Sequenciador na ordenação total), e para tarefas realizadas por ele. Em um modelo de sistema *crash-recovery*, algoritmos de consenso que utilizam quóruns podem tolerar até $\left\lfloor \frac{N}{2} - 1 \right\rfloor$ falhas (CHANDRA; TOUEG, 1996).

2.3.1 Equivalência com o problema de Difusão Atômica

O problema de difusão atômica (ou ordenação total) especifica que, em um sistema distribuído com vários processos, a entrega de mensagens deve respeitar a mesma ordem em todos eles. Em outras palavras, caso uma mensagem *m1* seja entregue antes de uma mensagem *m2* em um processo, então todos os outros devem respeitar esta mesma ordem. Este problema, como mencionado anteriormente, é costumeiramente solucionado com a eleição de um processo líder que exerce a função de garantir esta ordenação, o que cria um ponto único de defeito (*single point of failure*) que pode impedir o término do algoritmo. Uma das formas de solucionar este problema está no uso de algoritmos para consenso

distribuído, pois, essencialmente, o problema de difusão atômica pode ser reduzido a ele. Esta equivalência está no fato de que infinitas rodadas de consenso podem ser executadas para decidir cada posição da sequência de mensagens a serem entregues (CHANDRA; TOUEG, 1996; LAMPORT, 1998; CAMARGOS, 2008). (LAMPORT, 2005) modifica esta abordagem ao estender a definição de consenso em um único valor para um conjunto crescente de valores. Com isso, o problema se torna flexível ao ponto de solucionar diferentes problemas na área de sistemas distribuídos. Por exemplo, ao tratar estes valores como um conjunto não ordenado, o algoritmo de consenso resolve o problema de entrega confiável. Da mesma forma, caso esta sequência seja ordenada, resolvemos o problema de difusão atômica. Por fim, caso a cardinalidade deste conjunto de valores somente permita 1 elemento, então o problema volta a sua forma original de consenso em um único valor (PEDONE; SCHIPER, 1999).

2.4 Raft

Neste trabalho utilizamos o algoritmo Raft (ONGARO; OUSTERHOUT, 2014) para garantir difusão atômica. Este algoritmo segue o modelo de consenso distribuído explicado anteriormente e, utilizando um processo líder responsável por garantir as etapas do algoritmo. Além disso, o Raft também funciona baseado no sistema de votação majoritária (i.e., quórum), onde uma maioria de votos é necessária para a realização de algumas ações. Caso a obtenção do quórum seja inviabilizada, então o algoritmo não pode prosseguir e passa a não garantir novas etapas de consenso.

Em seu funcionamento, o Raft divide o tempo em termos de tamanho arbitrário, onde cada um começa com uma eleição de líder. Ao vencer uma eleição, um processo permanece líder até o fim do mandato, e caso a votação resulte em empate, então o termo finaliza sem um líder e a eleição é reiniciada. Este último cenário pode ser entendido como uma situação onde o quórum não foi obtido.

O termo atual é identificado por um número inteiro que é incrementado a cada nova iteração do algoritmo. Este número é guardado por cada processo e é utilizado na troca de mensagens do algoritmo, que tomam forma em dois serviços RPC diferentes. São eles:

1. **RequisitaVotos** - Utilizado pelos processos candidatos a líder durante cada termo;

2. **InsereEntradas** - Serviço utilizado pelo líder para replicar novas mensagens nos processos do sistema e também para a função de *heartbeat*, que será explicada adiante nesta seção.

Antes de prosseguirmos na descrição nas etapas do algoritmo, é necessário mencionar cada um dos 3 papéis que cada processo pode assumir em tempo de execução. São eles:

- **Seguidor** - Estado passivo onde os processos esperam por comandos do líder. Estes comandos servem para replicar as mensagens solicitadas ou para requisitar votos em uma nova eleição;
- **Candidato** - Estado onde um processo deseja se tornar o líder do termo. Para isso, ele solicita votos aos demais nós pertencentes ao *cluster*;
- **Líder** - Estado onde um processo é responsável por receber requisições para a inserção e replicação de novas mensagens. Este processo também deve enviar sinais *heartbeat* periodicamente para garantir a permanência da sua liderança.

Com isso, podemos prosseguir detalhando as etapas do algoritmo ao explicar os 2 sub-problemas que o compõem, a eleição do líder e a replicação de mensagens.

2.4.1 Eleição do líder

Uma nova eleição para líder acontece no início de todo termo, sendo que este dura enquanto a liderança não for desafiada por outro processo. Durante a sua execução, um processo líder perde sua titularidade quando um processo **Seguidor** não recebe uma mensagem *heartbeat* após o tempo máximo de espera. Após isso, este processo passa para o estado **Candidato** e incrementa o seu identificador de termo atual. Ele então vota para si mesmo, e envia uma mensagem do tipo **RequisitaVotos** para os demais processos. Esta solicitação possui 3 resultados possíveis:

1. O processo candidato obtém o quórum com respostas favoráveis a sua liderança. Após isso, ele começa a enviar as mensagens *heartbeat* para os demais e o seu termo começa;
2. Se outros candidatos a líder receberam a mensagem **RequisitaVotos**, então eles checam o número do termo recebido e caso este seja maior que o seu próprio, a requisição é aceita e o processo volta ao estado **Seguidor**. Caso o número do termo recebido seja menor que o próprio, então a requisição é recusada e o processo continua no estado **Candidato**.
3. Caso vários candidatos existam em uma mesma rodada de eleição, pode ser ela acabe sem um vencedor (i.e., nenhum obteve quórum). Se isto acontecer, a eleição recomeça após um dos candidatos receber um *timeout* na mensagem **RequisitaVotos**.

O algoritmo Raft garante que um processo líder sempre irá possuir todo o histórico de mensagens replicadas nos termos anteriores em seu estado local. Esta propriedade é alcançada durante a votação, pois um candidato só recebe votos caso o seu último índice de termo seja maior do que o do processo que realiza o voto. Em outras palavras, se um

processo do tipo **Seguidor** possuir mensagens mais recentes do que um candidato, este não receberá o voto.

2.4.2 Replicação de mensagens

As mensagens recebidas são processadas pelo processo líder do termo. Ao recebê-las, ele armazena internamente uma estrutura composta pela própria mensagem, o índice da nova entrada (i.e., valor responsável por ordenar as mensagens recebidas), e o número do termo atual do algoritmo. Após o armazenamento, o processo líder precisa garantir que os demais processos também recebam a mesma mensagem. Para isso, o serviço **InsereEntradas** é executado em todos os processos até que o número de recebimentos com sucesso atinja um quórum. Após isso, o processo líder insere o número do novo índice em uma lista interna cujo propósito é manter a ordenação das mensagens replicadas.

A requisição **InsereEntradas** contém o número do termo atual e o índice da última mensagem replicada. Com estes valores, o processo **Seguidor** verifica internamente qual o último índice de mensagem recebido por ele, e caso os valores sejam diferentes ele rejeita a nova requisição. Esta verificação garante ao líder que todo processo que respondeu com sucesso contém um estado exatamente igual ao seu.

Caso um processo responda a requisição **InsereEntradas** com falha devido a alguma inconsistência nos índices, então o líder realiza um processo de sincronização até que ambos possuam os mesmo estado. Este mecanismo permite que processos do tipo **Seguidor** se recuperem de falhas, o que em conjunto com o *heartbeat* do processo líder faz com que o algoritmo trate erros de ambas as partes. Esta propriedade garante que o modelo de sistema *crash-recovery* seja alcançado.

2.5 Replicação de máquinas de estado SMR

Replicação de máquinas de estado, ou SMR em inglês, pode ser entendida como um modelo para a construção de aplicações distribuídas e tolerantes a falhas (SCHNEIDER, 1990). Até agora, explicamos os conceitos de consenso distribuído e difusão atômica, ambos sob um ponto de vista abstrato de entrega ordenada de mensagens e acordo distribuído. Portanto, nesta seção exploraremos o problema sob uma ótica prática.

Uma aplicação que possui apenas um servidor centralizado e sem réplicas, possui o mesmo nível de tolerância à falhas de seu processador. Em outras palavras, caso este apresente defeitos, a aplicação fará o mesmo, cessando o fornecimento de serviços. Caso uma aplicação precise fornecer garantias mais fortes do que esta, então esta deve ser replicada em diferentes servidores, idealmente em localizações geográficas diferentes, para diminuir a probabilidade de um único defeito levar à incapacidade de fornecer um serviço (e.g., queda de energia, falhas de rede) (STEEN; TANENBAUM, 2017).

Estas réplicas devem possuir o seu estado replicado, ou seja, o resultado das operações realizadas em uma réplica também deve constar nas outras. Isso é importante pois caso uma das réplicas falhe, as outras devem assumir seu lugar sem perda de disponibilidade e consistência do sistema. O modelo de SMR especifica que a aplicação replicada deve poder ser definida formalmente como uma máquina de estados de transições determinísticas. Em outras palavras, o estado atual da aplicação deve ser produto apenas da combinação de uma entrada do sistema e de seu estado anterior, sendo que esta deve sempre resultar no mesmo estado (BESSANI; ALCHIERI, 2014). As transições de estado são, portanto, resultado das entradas do sistema, que no contexto de sistemas distribuídos podem ser entendidas como as mensagens recebidas pela aplicação. Ao garantir que estas mensagens sejam entregues na mesma ordem para cada réplica, temos como resultado máquinas de estado idênticas em cada processo, o que configura replicação de estado. Dessa forma, temos então um modelo prático para aplicações replicadas que utiliza os conceitos de difusão atômica, e, consequentemente, consenso distribuído.

2.5.1 Definição de uma máquina de estados

Mais formalmente, uma máquina de estados pode ser definida como uma estrutura com os seguintes atributos (SCHNEIDER, 1990):

- Um conjunto de estados;
- Um conjunto de entradas;
- Um conjunto de saídas;
- Uma função de transição determinística que leva uma entrada e um estado a um novo estado;
- Uma função de saída determinística que leva uma entrada e um estado a uma saída;
- Um estado inicial.

A máquina de estados começa no estado inicial. Após uma nova entrada, ela passa a um novo estado e gera uma saída, e este ciclo se repete enquanto novas entradas forem apresentadas. No contexto deste trabalho, entendemos as entradas como mensagens de rede e as saídas como as respostas emitidas para elas. Os estados são de responsabilidade da aplicação replicada, assim como o processamento das entradas e saídas.

2.5.2 O passo a passo da replicação de estado

De forma genérica, a replicação de máquinas de estado em um sistema distribuído pode ser resumida nas seguintes etapas:

1. Inicializar cópias da máquina de estado em múltiplos servidores;
2. Receber mensagens de clientes, que serão então interpretadas como entradas para as máquinas de estado;
3. Decidir a ordem das entradas;
4. Executar as entradas na ordem escolhida em cada um dos servidores distribuídos;
5. Responder aos clientes com a saída da máquina de estado;
6. Monitorar as réplicas para constatar possíveis defeitos.

Os passos 1, 2 e 5 são de responsabilidade da aplicação replicada, e sua implementação irá variar dependendo das regras de negócio e decisões arquiteturais do projeto (e.g linguagem de programação, *frameworks* de desenvolvimento etc). Como estas mensagens chegam até a aplicação replicada e como monitorar por falhas e inconsistências é responsabilidade da camada de replicação, que pode estar incorporada na própria aplicação ou externalizada como um serviço. Neste trabalho, criamos um serviço para replicação de máquinas de estado em aplicações disponibilizadas no Kubernetes. Estas aplicações residem em camadas de virtualização de *software* chamadas contêineres, que serão explicadas em detalhes adiante neste capítulo.

2.6 Virtualização da infraestrutura de computação

A tarefa de executar uma mesma aplicação em diferentes máquinas é complexa. Nem sempre a arquitetura computacional é suportada, além da inserção de novas variáveis que podem alterar o seu funcionamento esperado. Este problema sempre existiu, porém foi somente com o advento da computação em nuvem que ele foi colocado em evidência na escala que vemos hoje. Este modelo de computação sob demanda tornou imprescindível uma forma eficiente de garantir interoperabilidade de um sistema computacional em diversas infraestruturas, seja em servidores de provedores diferentes (e.g AWS[1], GCP[2], Azure[3] etc), ou até mesmo em *datacenters* proprietários.

A virtualização da infraestrutura de computação permitiu que este problema fosse resolvido, criando um ambiente favorável para o nascimento de novas arquiteturas de sistemas, como por exemplo os microsserviços (VURAL; KOYUNCU; GUNEY, 2017). Nesta seção iremos explicar o modelo tradicional de virtualização chamado de máquinas virtuais (VM) e também o mais recente e utilizado, os contêineres Docker.

[1] https://aws.amazon.com
[2] https://cloud.google.com
[3] https://azure.microsoft.com

2.6.1 Máquinas virtuais (VM)

Uma máquina virtual pode ser definida como uma cópia isolada do sistema operacional de uma máquina real (ROSENBLUM; GARFINKEL, 2005). Esta cópia executa como se estivesse interagindo com o *hardware* físico, quando na verdade não está. Este conceito foi introduzido na década de 1960 e desde então foi empregado em diversas soluções, como por exemplo a *Java Virtual Machine* (JVM) (PINILLA; GIL, 2003).

As máquinas virtuais foram originalmente concebidas para permitir que diversos operadores pudessem trabalhar simultaneamente em uma mesma máquina. Desta forma, era possível isolar o trabalho de cada um sem o risco de possíveis interferências.

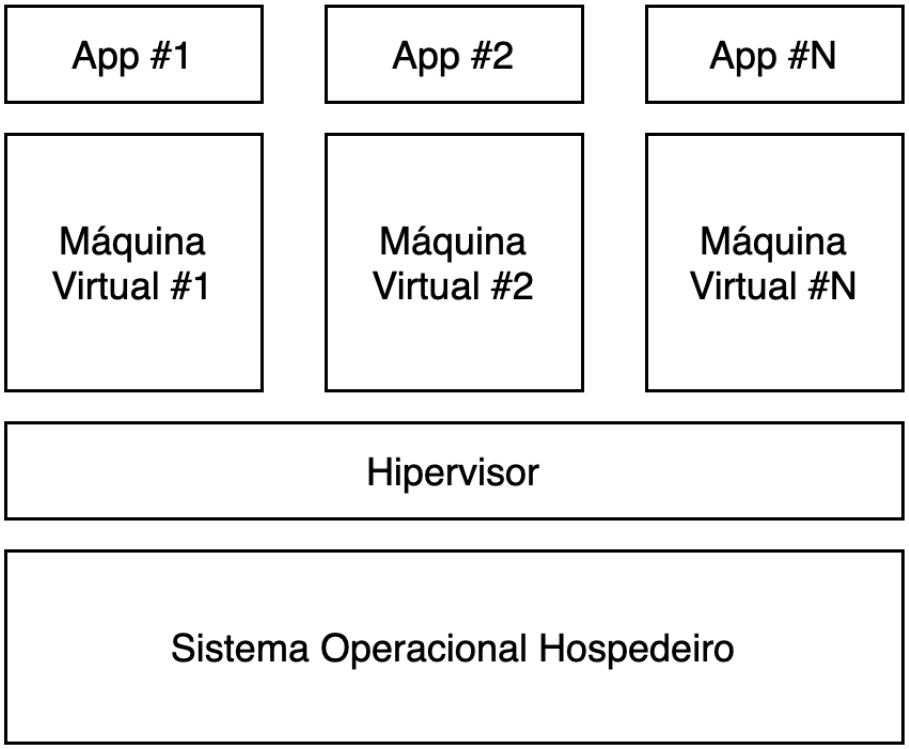

Figura 1 – Pilha simplificada de uma infraestrutura de máquinas virtuais

A Figura 1 ilustra um sistema operacional com 3 máquinas virtuais diferentes. Cada uma destas máquinas possui uma instância de uma determinada aplicação, todas completamente isoladas uma das outras. O responsável por gerenciar esta abstração é o sistema hipervisor, que será explicado a seguir.

2.6.1.1 Sistema hipervisor

O sistema hipervisor é um tipo de emulador responsável por criar uma camada lógica entre as máquinas virtuais e o sistema hospedeiro. Esta camada virtualiza os recursos do *hardware* de forma a permitir que todas as máquinas virtuais os utilizem simultaneamente (DESAI et al., 2013). Existem dois tipos de sistema hipervisor:

- **Nativos ou *bare-metal***: Executam diretamente no *hardware* hospedeiro, sem passar por um sistema operacional. Exemplos: Microsoft Hyper-V, VMware ESXi, Citrix XenServer etc.

- **Hospedados**: Executam em cima de um sistema operacional hospedeiro, da mesma forma que qualquer outro processo convencional. Alguns exemplos são: VirtualBox, VMWare Workstation, Parallels Desktop, etc.

2.6.2 Contêineres Docker

Docker é um conjunto de produtos no modelo *Platform as a Service (PaaS)* (PASTORE, 2013) que provê virtualização de nível de sistema operacional por meio de contêineres. Diferentemente das máquinas virtuais que emulam sistemas operacionais inteiros, os contêineres oferecem isolamento semelhante utilizando o *kernel* da máquina hospedeira e configurações de espaço de usuário (SILVA; KIRIKOVA; ALKSNIS, 2018). Do ponto de vista de uma aplicação, um contêiner atuará como se fosse uma máquina real com seus próprios recursos computacionais.

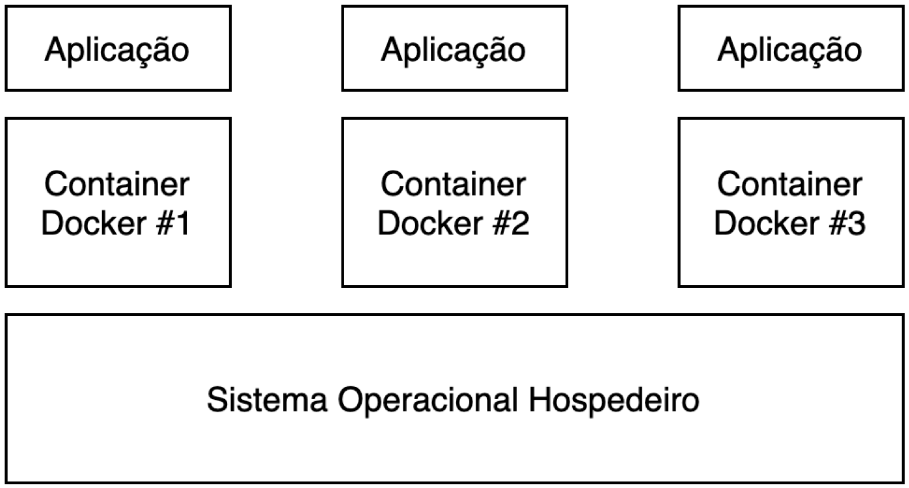

Figura 2 – Pilha simplificada de uma infraestrutura Docker

A Figura 2 ilustra uma infraestrutura Docker. Diferentemente de máquinas virtuais, nem sempre é necessário um hipervisor, uma vez que a virtualização é feita utilizando recursos do próprio sistema operacional. Feito originalmente para Linux, a tecnologia utiliza funcionalidades nativas para isolamento de processos (i.e., Linux *namespaces*, *cgroups* etc). No caso de outros sistemas operacionais como o Windows e o MacOS, são necessários hipervisores para garantir compatibilidade (RAD; BHATTI; AHMADI, 2017).

2.7 Kubernetes (Orquestrador de contêineres)

A revolução criada com o advento de tecnologias de virtualização mais baratas como o Docker possibilitou uma maior adoção de serviços de infraestrutura sob demanda (SONG; XIAO, 2013). Com isso, a rápida expansão desse mercado e sua natureza efêmera consequentemente aumentaram a complexidade do gerenciamento do ciclo de vida dos contêineres presentes. Desse problema surgiu a demanda por ferramentas que pudessem orquestrá-los de maneira simples e efetiva (KHAN, 2017).

O Kubernetes é um sistema de código aberto para orquestração de contêineres que foi desenvolvido originalmente pela Google, mas que hoje é mantido pela *Cloud Native Computing Foundation* (CNCF)[4]. Ele funciona com diversas tecnologias de virtualização, inclusive o Docker.

2.7.1 Arquitetura do sistema

A Figura 3 ilustra os componentes do Kubernetes. Nela, podemos ver 4 nós, sendo que 1 deles é o nó mestre, o responsável pelo plano de controle do sistema. Os outros 3 são utilizados para a disponibilização dos contêineres solicitados, que serão abstraídos para uma estrutura interna denominada *Pod*. O Kubernetes é um sistema distribuído que possui diversas componentes que comunicam entre si para gerenciar o *cluster* (MARTIN, 2020). As detalharemos a seguir.

2.7.1.1 API server

O API Server é o principal componente do plano de controle do Kubernetes. Ele é responsável por disponibilizar uma interface HTTP para a configuração do *cluster*. Ao receber comandos nesta interface, ele delega tarefas para os outros componentes a fim de realizar a tarefa solicitada. Uma ferramenta muito conhecida na indústria para execução de comandos no API Server é o *kubectl*, um programa do tipo *Command line interface (CLI)* que abstrai as requisições HTTP necessárias para interagir com o *cluster*, seja para criar, apagar, alterar ou apenas ler recursos.

2.7.1.2 Controller manager

Componente responsável por gerenciar o ciclo de vida dos recursos nativos do Kubernetes. É um *software* do tipo *daemon*, ou seja, executa no plano de fundo do sistema operacional. Ele funciona por meio de um laço de controle que verifica periodicamente com o API Server qual o estado desejado para um recurso, e realiza as ações necessárias para mover o estado observado para ele. Um exemplo simples é o caso de um recurso *Deployment* que espera que 2 *Pods* sejam criadas no *cluster*, com apenas 1 criada em um

[4] https://www.cncf.io

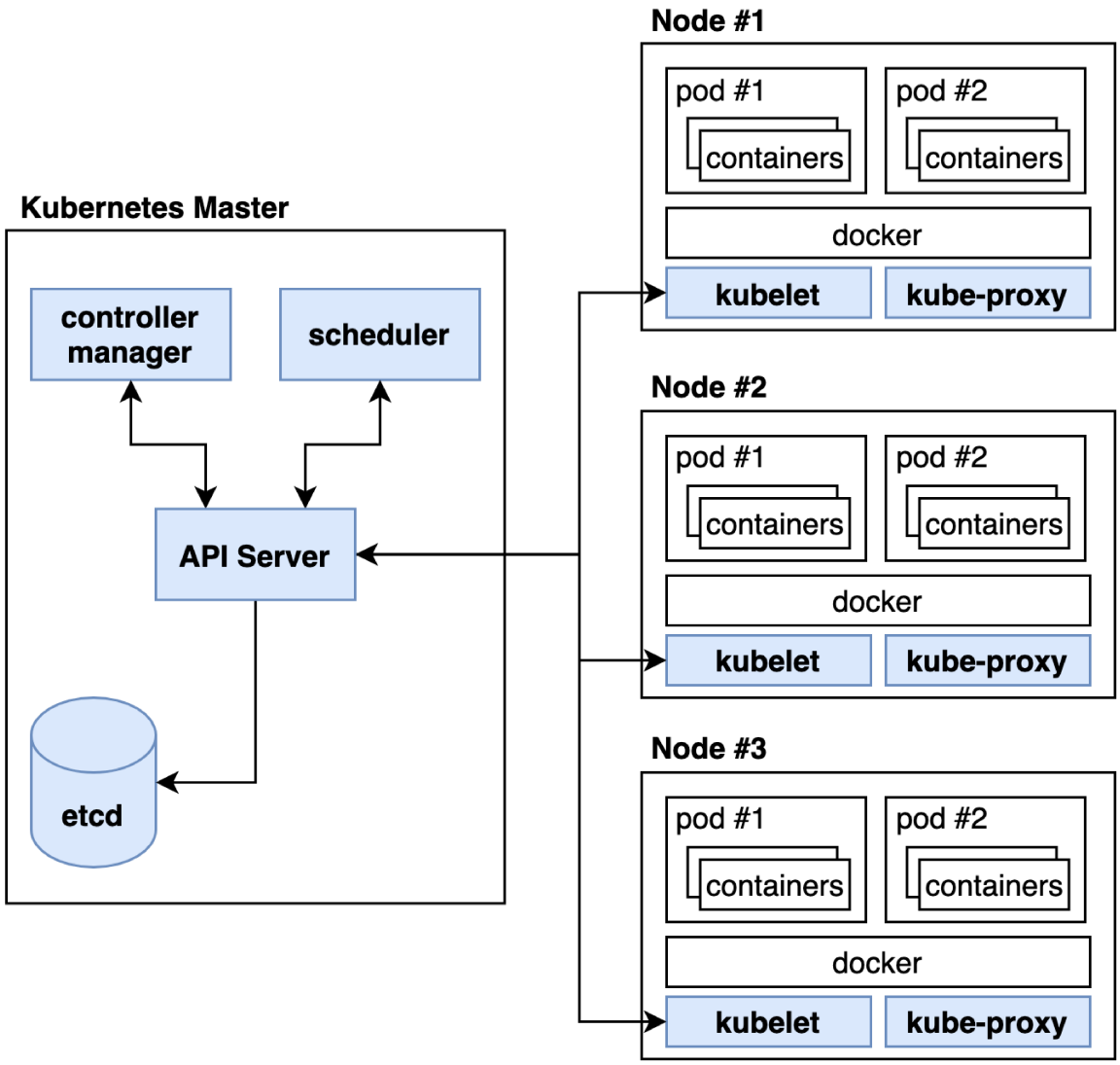

Figura 3 – Componentes do Kubernetes

dado instante de tempo. Neste caso, o *controller manager* irá realizar ações para criar a *Pod* faltante.

2.7.1.3 Scheduler

O *scheduler* é responsável por determinar se novos contêineres devem ser disponibilizados no *cluster* e, caso positivo, em qual nó eles devem ser criados. Para isso, o componente monitora o uso de recursos computacionais em cada nó e só aloca novos contêineres neles caso estes não sejam saturados.

O Kubernetes também possibilita que o operador do sistema defina quais os requisitos de memória, CPU, e até mesmo afinidade para certos nós na definição de uma aplicação. Portanto, este componente é crucial para o funcionamento correto do sistema, pois, além de garantir que os requerimentos do usuário serão atendidos, ele deve fazer isso sem

comprometer a saúde da infraestrutura como um todo.

2.7.1.4 Etcd

O Etcd é um banco de dados distribuído do tipo chave e valor tolerante à falhas. O Kubernetes o utiliza para armazenar o estado do *cluster*, por isso a necessidade de um banco de dados confiável e altamente disponível (LARSSON et al., 2020). Internamente o Etcd utiliza o algoritmo Raft para garantir consistência e disponibilidade.

2.7.1.5 Kubelet

O Kubelet é o *software* responsável por gerenciar o estado de um nó participante do *cluster* Kubernetes. É através dele que os recursos são criados e monitorados pelo API Server, o que o faz ser uma das componentes mais importantes do sistema. Ele pode ser entendido como o plano de dados do Kubernetes, enquanto o API Server e os demais componentes do nó mestre como o plano de controle.

2.7.1.6 Kube-proxy

O componente kube-proxy, assim como o nome sugere, funciona como um *proxy* de rede que permite que tráfego chegue e saia dos contêineres presentes em um nó. Ele pode funcionar de dois modos, roteando os pacotes através de si mesmo, ou através de recursos de rede do sistema operacional como o *iptables*. Além disso, ele também funciona como um distribuidor de carga tradicional, aliviando o processamento das aplicações caso mais de uma réplica exista. Na prática, o kube-proxy encaminha os pacotes para um recurso do tipo *Service*, que será explicado em detalhes mais adiante.

2.7.2 Pods e Deployments

O Kubernetes é um orquestrador de contêineres e, portanto, possui como tarefa fundamental organizar estes de maneira fácil e eficiente. A solução encontrada pela ferramenta foi a abstração de contêineres em um recurso chamado *Pod*. Este recurso pode conter 1 ou mais contêineres simultaneamente, compondo um bloco lógico que representa uma aplicação real. A capacidade de comportar mais de 1 contêiner é fundamental pois permite que aplicações auxiliares existam, abrindo um leque de possibilidades para padrões de arquitetura. Uma delas é o padrão *Sidecar*, que possibilitou a concepção da ferramenta proposta neste trabalho (BURNS; OPPENHEIMER, 2016).

Uma *Pod* pode ser criada de duas maneiras: ou através de um manifesto *yaml*, ou após a execução de um comando utilizando a CLI do Kubernetes. Após sua criação, ela será disponibilizada no *cluster* e o contêiner estará executando. Caso alguma falha ocorra, a *Pod* será reinicializada junto com seu contêiner para tentar resolver o erro. Este esforço é realizado pelo componente Controller manager descrito anteriormente. É importante

ressaltar que uma *Pod* sozinha não traz consigo funcionalidades para estratégias de *deployment*, criação de réplicas, entre outros. O recurso *Deployment* foi criado para suprir esta demanda.

Um *Deployment* pode ser entendido como um gerenciador de *Pods*. Através dele é possível definir como elas serão criadas, quantas replicas existirão inicialmente, além da possibilidade de realização de *rollbacks* para versões específicas, configuração de *auto-scaling*, entre outras funcionalidades. Sem um *Deployment*, seria necessário aplicar o manifesto de uma *Pod* 3 vezes para criar 3 réplicas, além de implementar a estratégia de atualização manualmente, o que é inviável do ponto de vista prático. Além do recurso *Deployment* também existem diversos outros como por exemplo o *StatefulSet*, *CronJob*, *PersistentVolume*, *ConfigMap*, entre outros (MARTIN, 2020). Para este trabalho não explicaremos todos eles, pois o número é muito extenso. Portanto, a seguir explicaremos como funciona o modelo de rede do Kubernetes, uma peça fundamental para a replicação de máquinas de estado.

2.7.3 Modelo de rede

Nesta seção explicaremos a estrutura de rede do Kubernetes e as peças que a compõem. Para isso, dividimos o conteúdo em 3 partes: Primeiro falaremos sobre a rede no contexto das *Pods* e seus contêineres (BETZ, 2017b), depois sobre o recurso *Service* (BETZ, 2017c) e, por último, sobre o recurso *Ingress* (BETZ, 2017a), a ponte entre o mundo externo e os recursos do *cluster*.

2.7.3.1 Pods

Na Figura 4 podemos ver as interfaces de rede e componentes envolvidas na comunicação entre duas *Pods* que residem em nós diferentes. Cada uma dessas *Pods* possui 2 contêineres para a aplicação e um terceiro pausado. Este terceiro contêiner existe apenas para prover a interface de rede **veth0** que será compartilhada entre todos os contêineres da *Pod*. Esta se comunica com a interface Docker **cbr0** do nó, que por sua vez se comunica com a interface **eth0**, que funciona como a ponte para a rede externa.

O fato de todos os contêineres de uma *Pod* compartilharem a mesma interface de rede impossibilita que eles escutem pacotes em uma mesma porta. Por outro lado, isto também possibilita que aplicações residentes em dois contêineres diferentes de uma mesma *Pod* se comuniquem através do endereço 127.0.0.1 (i.e., localhost). Isto possibilita a criação de diversos padrões de construção de aplicações, como por exemplo o padrão *Sidecar* que foi utilizado para a ferramenta concebida neste trabalho.

Todos os pacotes oriundos de um nó Kubernetes são direcionados para um *gateway* (i.e., roteador) que possui uma tabela NAT que os redireciona para o nó correto, baseado em regras dinâmicas. Estas regras são configuradas de acordo com o estado atual do

cluster, que pode mudar frequentemente devido a interação humana ou ciclos de vida dos recursos. Essa combinação de endereços IP voláteis e regras de roteamento dinâmicas são características de uma rede *overlay* (i.e., uma rede virtual criada sobre de uma já existente) (GALÁN-JIMÉNEZ; GAZO-CERVERO, 2010).

Em resumo, a arquitetura descrita garante que duas *Pods* possam se comunicar dentro de um *cluster* Kubernetes. O problema é a premissa de que este endereço *ip* nunca irá mudar. Realisticamente isto não acontece na esmagadora maioria dos casos, pois uma série de fatores podem acontecer com uma *Pod*, como por exemplo uma reinicialização ou realocação em um novo nó, entre outros. Além disso, fixar um endereço *ip* impossibilita funcionalidades como por exemplo balanceamento de carga e descoberta de novas réplicas de uma aplicação. Para resolver este problema e garantir que *Pods* possam ser efêmeras, o Kubernetes disponibiliza um recurso que atua juntamente ao componente **kube-proxy** para alcançar este objetivo. O nome dele é Service e ele será explicado na próxima seção.

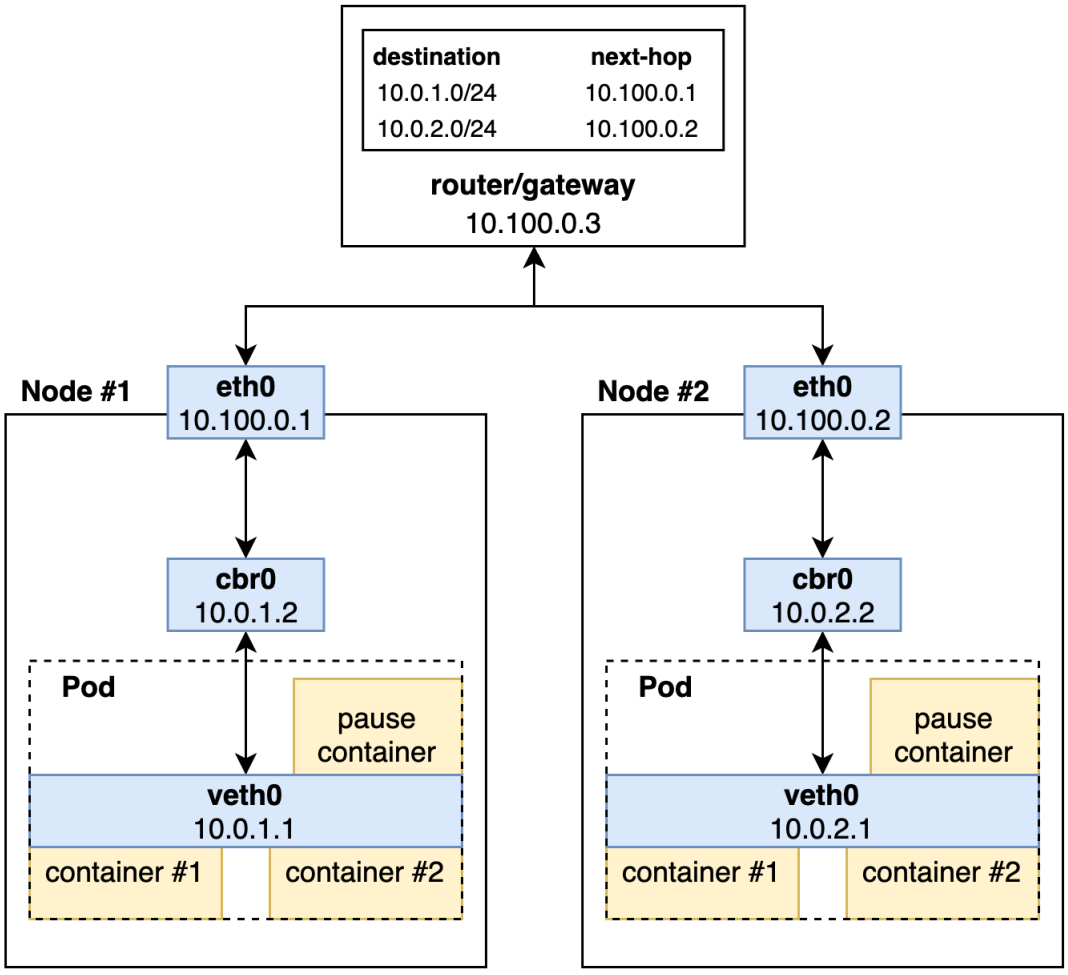

Figura 4 – Caminho de um pacote entre duas *Pods* em nós diferentes

2.7.3.2 Services

O recurso *Service* funciona como uma abstração de rede para um conjunto de *Pods*. Com ele, é possível definir quais as políticas de acesso a elas além criar uma camada que permite o desacoplamento do cliente e os endereços *ip* de destino atuais. A definição de quais *Pods* são representadas por um *Service* acontece por meio de um seletor que faz referência a uma *label*. Todas as *Pods* que possuem esta *label* serão utilizadas como destino do *Service*, e o componente API Server no nó mestre é responsável por notificar os Kubelets em cada nó caso haja alguma alteração no estado atual do *cluster* no que se refere a isto.

Cada *Service* possui um *ip* próprio e, portanto, todas as mensagens destinadas a uma *Pod* devem ser enviadas para este endereço. Para possibilitar que estes sejam efêmeros e que as aplicações Kubernetes não precisem utilizar *ips* fixos, o Kubernetes oferece uma solução nativa de DNS, que já cria uma entrada com o nome especificado no manifesto *yaml*. Caso mais de uma *Pod* possua a *label* especificada pelo *Service*, então o Kubernetes entregará as requisições seguindo o algoritmo *Round Robin*. Desta forma toda a carga será distribuída igualmente entre todas as réplicas. Caso uma réplica não esteja saudável, então o Kubernetes não entregará mensagens a ela.

O componente responsável pelas regras descritas acima é o kube-proxy, sendo que a fonte de informações sobre o estado atual do *cluster* é o API Server por meio do Kubelet do nó. O kube-proxy, assim como o nome sugere, funciona como um interceptador de mensagens. Ou seja, todas as mensagens com destino a algum *Service* serão alteradas para ter como destino o *ip* de alguma das *Pods* representadas, iniciando assim o fluxo descrito na seção anterior. Por isso, caso uma *Pod* tenha o seu *ip* alterado, a aplicação cliente não deixará de funcionar, pois o kube-proxy terá atualizado suas tabelas de roteamento para refletir o estado do *cluster*. Internamente, o kube-proxy utiliza a ferramenta *iptables*[5] para interceptar estes pacotes de rede, portanto seu único trabalho é atualizar suas regras corretamente.

O Kubernetes possui vários tipos de recursos *Service*, e nesta seção explicamos o funcionamento do tipo *ClusterIP*. Na próxima seção iremos explicar o *Service NodePort* e como ele e o recurso *Ingress* ajudam o Kubernetes a lidar com mensagens externas ao *cluster*.

2.7.3.3 Ingress

Até agora explicamos como o Kubernetes lida com tráfego de rede leste-oeste (i.e., tráfego entre aplicações do próprio *cluster*). Nesta seção, mostraremos como um pacote de rede com origem externa ao *cluster* consegue chegar em uma *Pod*.

[5] https://linux.die.net/man/8/iptables

Sabemos que um pacote originado de uma *Pod* com destino a um *Service* é interceptado pelo kube-proxy e redirecionado para o endereço correto. O que não foi mencionado ainda é que este comportamento ocorre independentemente da origem do pacote. Ou seja, caso qualquer pacote chegue no nó com endereço de destino para o *ip* de algum *Service*, então o kube-proxy o interceptará para aplicar suas regras de roteamento. Com isso foi possível construir uma solução onde o tráfego externo é redirecionado para um dos nós disponíveis em uma porta específica que está diretamente associada à porta da *Pod* exposta no *Service*. Após o recebimento do pacote nesta porta, o kube-proxy entende que o endereço *ip* de destino deve ser traduzido para o mesmo do *Service*, executando então o mesmo fluxo que foi descrito na seção anterior.

O recurso responsável por associar uma porta do nó a um *Service ClusterIP* é também um *Service*, porém de um outro tipo, chamado *NodePort*. Ao ser criado, este recurso cria automaticamente um *Service ClusterIP* e associa as portas de rede do nó e da *Pod*. Com isso, basta enviar uma mensagem para o endereço de algum dos nós do Kubernetes com a porta correta para acessar as *Pods* do *cluster*. Isso já resolve o problema apresentado e também cria um ambiente favorável para a utilização de ferramentas de distribuição de carga. O Kubernetes, como uma ferramenta altamente extensível, também possui um recurso para facilitar a configuração destes através de manifestos *yaml*. O nome dele é *Ingress* e possibilita que aplicações de mercado, como por exemplo o Nginx[6] e o HAProxy[7], sejam utilizadas no Kubernetes.

A Figura 5 ilustra de forma simples a trajetória de uma mensagem externa para dentro de um *cluster* Kubernetes até chegar em uma *Pod*. Nela, um balanceador de carga é responsável por captar as mensagens originais de um cliente para, após isso, redirecioná-la para um dos servidores em uma porta específica configurada pelo *Service NodePort*. Na mesma figura esta porta possui o número 32213 e é utilizada pelo componente *kube-proxy* para encaminhar a mensagem para a *Pod* correta, através de seu *Service ClusterIP*. Após a execução, o caminho inverso é realizado para devolver a resposta para o cliente.

Com isso, concluímos a explicação do modelo de rede, parte fundamental do orquestrador de contêineres que permitiu que nossa ferramenta fosse desenvolvida.

2.8 Atomix

O Atomix é um *framework* para coordenação de sistemas distribuídos que provê ferramentas para a resolução de vários problemas comuns na área, como por exemplo o gerenciamento de um *cluster* de nós, mensageria assíncrona, replicação de estado, entre outros. Sua API de primitivas abstrai a complexidade de criar estruturas de dados replicadas, funcionando quase como um banco de dados chave e valor tolerante a falhas.

[6] https://www.nginx.com
[7] http://www.haproxy.org

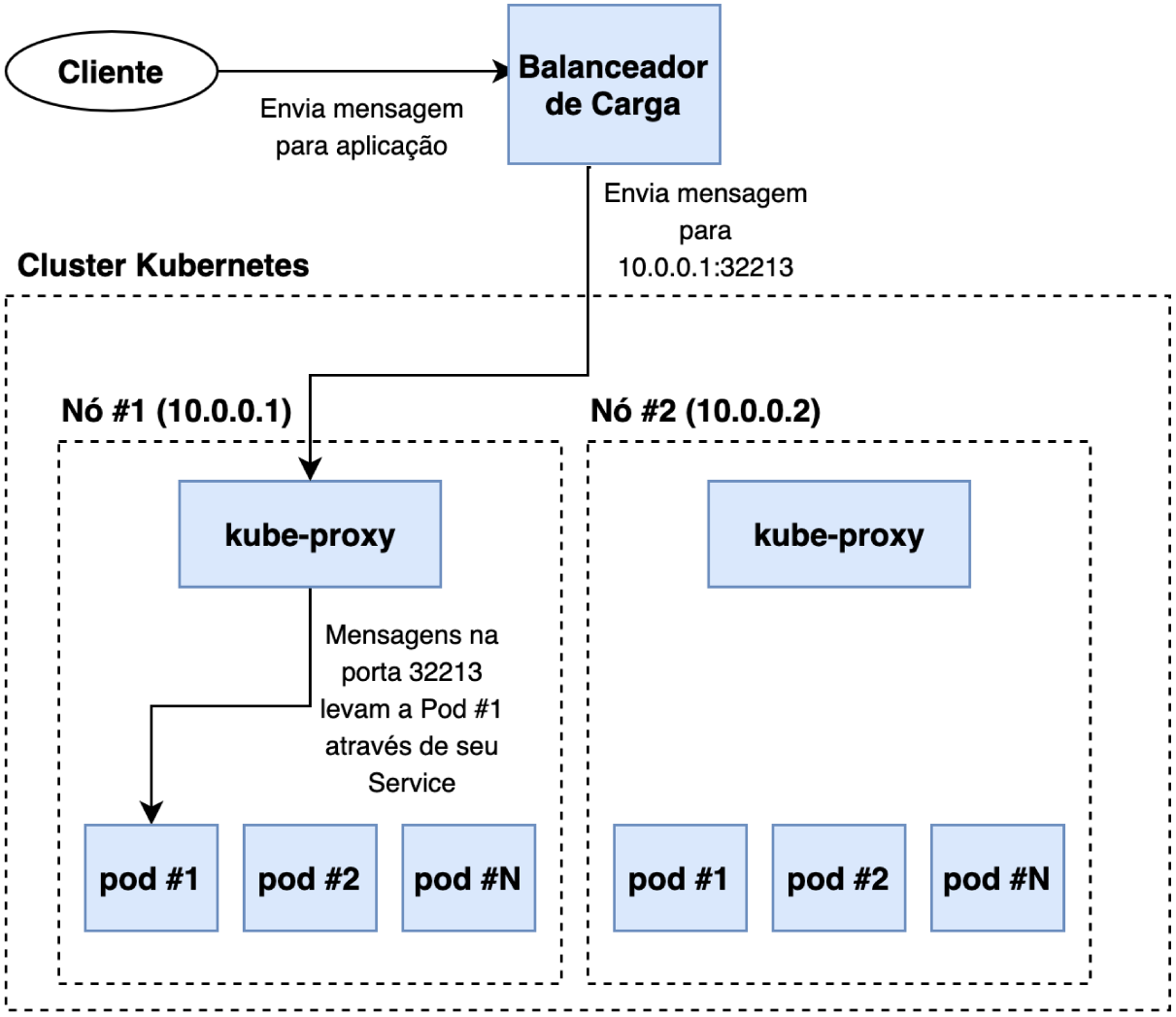

Figura 5 – O trajeto de uma mensagem externa ao *cluster* até uma *Pod*

Além disso, essas APIs também abstraem alguns problemas importantes de coordenação de serviços, como por exemplo eleições de líder e *locks* distribuídos.

Internamente as primitivas são classificadas em duas categorias: Dados e Coordenação. Alguns exemplos de primitivas de dados são:

- **Counter** - Um valor inteiro replicado com funções de leitura e escrita. Dois exemplos de funções disponíveis são: **Incrementar** e **Decrementar**.

- **List** - Lista replicada com valores arbitrários. Esta primitiva permite que o usuário interaja com a lista e escute eventos de inserção, remoção ou alteração.

- **Map** - Estrutura de dados do tipo chave e valor replicada. Esta primitiva também permite que os usuários escutem eventos de alteração na estrutura, possibilitando a criação de aplicações reativas.

- **Set** - Estrutura de dados do tipo *Set*. Esta estrutura funciona como um conjunto que não contém dois elementos idênticos. Da mesma forma que as primitivas anteriores, também permite que o usuário realize operações na estrutura além de se registrar para escutar eventos de alteração.

- **Value** - Primitiva que representa um valor qualquer na forma de um *array* de *bytes*. Ela oferece três operações: Uma de escrita, uma de leitura e outra para o registro de escuta de eventos de alteração do valor.

Exemplos de primitivas de coordenação:

- **Leader election** - Abstração do processo de eleição de líder no formato de uma API. Ela contém diversos métodos para inclusão ou remoção de novos membros no algoritmo, priorizar ou evitar certos nós, além de mecanismos para escuta de eventos que possam acontecer no decorrer de sua execução.

- **Distributed locks** - Implementação de um sistema de exclusão mútua distribuído. Com esta primitiva, vários processos em nós diferentes concordam sobre quem tem prioridade de acesso a determinados recursos em um dado instante. É um problema complexo que foi facilitado pelo *framework*.

O Atomix faz parte do núcleo do sistema operacional para SDNs ONOS (BERDE et al., 2014), um produto da *Open Network Foundation* (ONF)[8] utilizado em sistemas produtivos que necessitam de alto desempenho. Por esse motivo, pode-se dizer que o Atomix foi testado extensivamente e é confiável para o uso. Originalmente escrito em Java, a última versão do Atomix foi foi totalmente reescrito na linguagem Golang para se adequar às novas tecnologias para infraestrutura de sistemas, sendo uma delas o Kubernetes. Neste trabalho, usamos a versão escrita em Golang.

Para facilitar a instalação de seus módulos no Kubernetes, o Atomix disponibiliza *charts* Helm com configurações parametrizáveis. O módulo mais importante é o **Atomix-Controller**, responsável por gerenciar os seguintes recursos customizados no Kubernetes:

- **Database** - Responsável por disponibilizar bancos de dados no *cluster* Kubernetes. Estes são implementações de algoritmos de consenso, sendo o mais comum o **RaftDatabase**.

- **Partition** - Recurso responsável por realizar particionamento de bancos de dados.

- **Member** - Recurso utilizado para a implementação de protocolos *peer-to-peer*.

- **Primitive** - Utilizado para persistir metadados sobre primitivas distribuídas criadas no *cluster*.

[8] https://opennetworking.org/

Além do **AtomixController**, também é necessário instalar outro módulo para a criação do bancos de dados. Este módulo também implementa o padrão *controller* para gerenciamento de recursos Kubernetes e varia de acordo com o tipo de banco de dados desejado. No caso do **RaftDatabase**, é necessário instalar o **RaftStorageController**. A função deste módulo é gerenciar o banco de dados de forma mais granular, enquanto o **AtomixController** gerencia a plataforma sob um ponto de vista mais geral.

Capítulo 3

Trabalhos Relacionados

Neste capítulo apresentaremos alguns trabalhos com objetivos semelhantes ao nosso ou que foram referência para as decisões arquiteturais tomadas.

3.1 Coordenação de Contêineres no Kubernetes: Uma Abordagem Baseada em Serviço

Segundo (NETTO et al., 2017b), a incorporação de um mecanismo de coordenação de mensagens em um sistema pode ser feito com 3 abordagens distintas:

- **Integração** - Construindo ou modificando um componente do sistema a fim de acrescentar a funcionalidade;
- **Interceptação** - As mensagens são interceptadas no caminho para os destinatários e mapeadas em um sistema de comunicação em grupo de forma transparente a aplicação. Um exemplo são as interfaces do sistema operacional que interceptam *system calls* realizadas por algum processo;
- **Serviço** - Uma camada de *software* é construída entre o cliente e a aplicação, provendo a coordenação das mensagens recebidas.

O trabalho apresenta uma solução baseada na abordagem de serviço para o problema de SMR. O nome do sistema proposto é Coordenação via Serviço no Kubernetes (CAIUS) e ele especifica que deve existir um contêiner para coordenação em cada nó do *cluster*, e que ele interceptará todas as mensagens recebidas na máquina. Após a interceptação, o *software* de coordenação comunica-se com suas réplicas utilizando o algoritmo Raft para coordenar o envio da mensagem recebida para todos os destinatários. É importante mencionar que este processo não é transparente para o usuário final, pois uma mensagem *ACK* é enviada para o cliente assim que a mensagem do cliente é interceptada, e em um segundo momento a resposta da aplicação é emitida para o cliente.

Já na proposta apresentada neste trabalho, o cliente não possui conhecimento de que internamente suas mensagens estão sendo replicadas através de um mecanismo de replicação de máquinas de estado, garantindo mais transparência. Além disso, nossa arquitetura não requer um contêiner de coordenação responsável por cada nó do *cluster* Kubernetes. No lugar, instanciamos um contêiner *proxy* que fica acoplado a cada réplica da aplicação, removendo um ponto único de falha para um nó qualquer do sistema.

3.2 State machine replication in containers managed by Kubernetes

Ainda seguindo as 3 abordagens para coordenação mencionadas na seção anterior, (NETTO et al., 2017a) apresenta uma solução de integração onde a componente utilizada para auxiliar na resolução do problema de SMR é o próprio banco de dados chave e valor etcd presente nas instalações do Kubernetes. No sistema proposto, um módulo de coordenação existe no contêiner da aplicação, interceptando mensagens e garantindo a entrega ordenada delas em cada réplica.

Podemos ver a arquitetura proposta na Figura 6. Primeiro, a mensagem é entregue pelo *firewall* a algum dos nós do *cluster*. Nele, o componente kube-proxy a roteia para a *Pod* desejada, e o módulo de coordenação intercepta a mensagem.

Ao interceptar a mensagem, o módulo de coordenação se comunica com o etcd através de um protocolo concebido pelos próprios autores chamado *Ordering Over Shared Memory* (DORADO). Após a execução da rodada do protocolo, as mensagens são entregues às réplicas da aplicação e uma resposta é emitida ao cliente através da *Pod* que a recebeu primeiro.

Este trabalho, apesar de possuir uma arquitetura similar a nossa, utiliza uma abordagem diferente para alcançar seus objetivos. Nossa arquitetura é menos intrusiva no sentido de que a aplicação não precisa conhecer os detalhes do módulo de coordenação, exceto pela construção de uma *API* necessária para recuperação de falhas. Outra diferença está na utilização de um protocolo próprio para comunicação com o etcd. Nosso trabalho não possui opinião quanto a solução utilizada para alcançar difusão totalmente ordenada. Ou seja, nossa arquitetura pode se integrar com qualquer serviço que garanta difusão totalmente ordenada, sendo o único requisito que este serviço ofereça as *API*s necessárias.

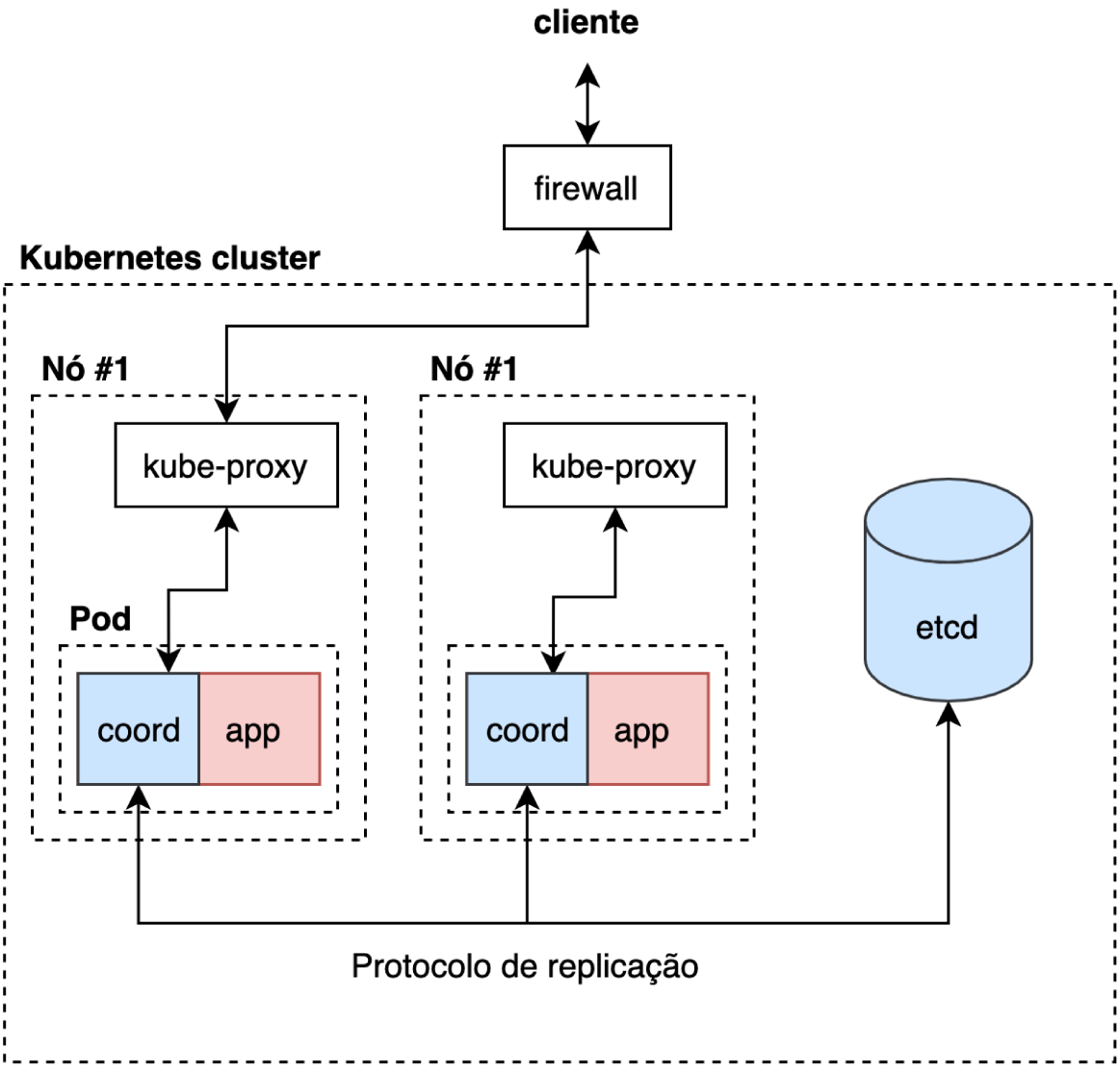

Figura 6 – Arquitetura do sistema para SMR que utiliza o etcd

3.3 Transparent State Machine Replication for Kubernetes

(BORGES et al., 2019) propõe uma arquitetura para SMR no Kubernetes utilizando a abordagem de serviço. A premissa era que ela fosse transparente para a aplicação final.

Na Figura 7 podemos ver uma ilustração da arquitetura proposta. Nela, existem 3 componentes chave:

- **Proxy** - Componente que encaminha as mensagens recebidas do cliente para algum dos SMaRtClients. A entrega é feita de forma balanceada a fim de não sobrecarregar nenhuma réplica.

- **SMaRtClient** - Responsável por executar o algoritmo para coordenação e distri-

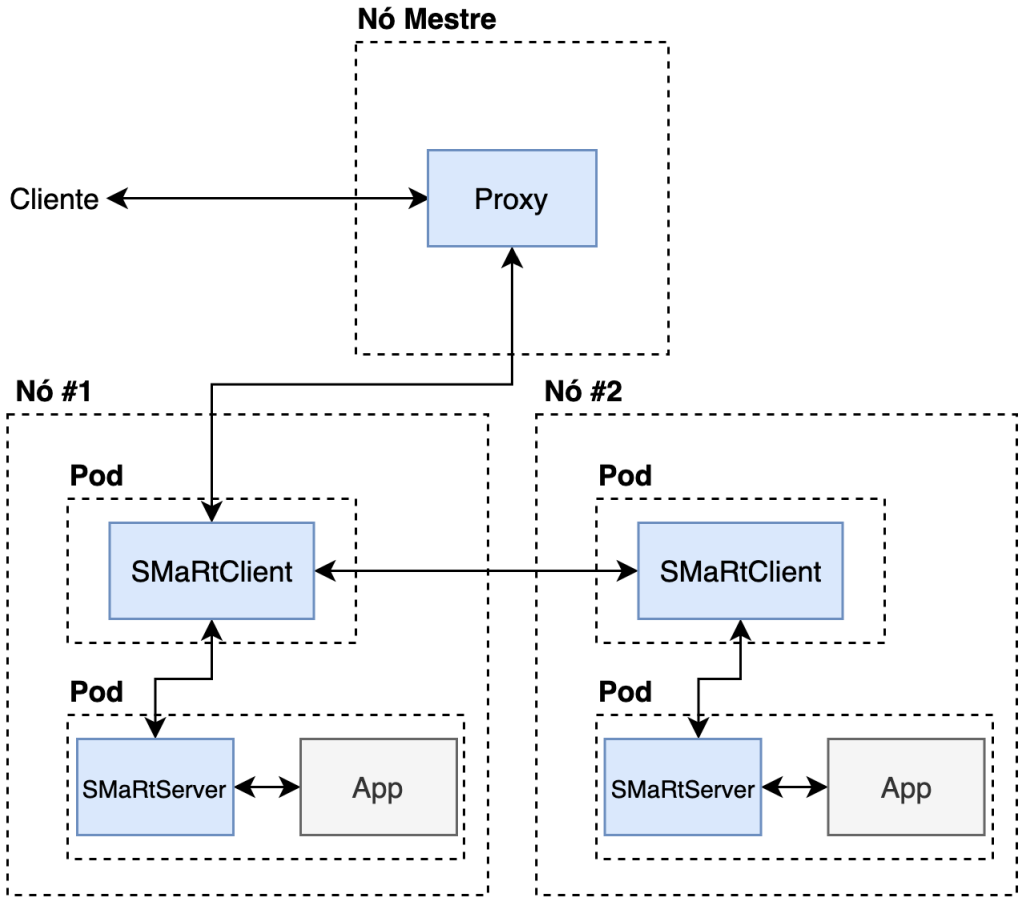

Figura 7 – Arquitetura transparente a aplicação para SMR no Kubernetes

buir as mensagens para todas as réplicas de SMaRtServers. Internamente, esta componente utiliza a biblioteca BFT-SMaRt para executar o algoritmo de difusão totalmente ordenada.

- **SMaRtServer** - Responsável por receber as mensagens encaminhadas pelo SMaRtClient e entregar à aplicação replicada. Também é responsável por processar a resposta da aplicação e devolvê-la ao SMaRTClient.

Este trabalho também possui semelhanças com o nosso, principalmente na transparência alcançada com a aplicação replicada, que não precisa alterar seu funcionamento para se integrar com a arquitetura de *SMR*. A diferença fundamental está na integração com o serviço que garante difusão totalmente ordenada. No caso do trabalho relacionado, a ferramenta BFT-SMART foi escolhida para essa tarefa, inclusive para o processo de recuperação de falhas e transferência de estado. Por outro lado, apesar da instanciação apresentada da nossa arquitetura utilizar o Atomix, não opinamos na escolha do serviço para difusão totalmente ordenada, sendo o único requisito que ele possua as *APIs* necessárias.

3.4 State Machine Replication for the Masses with BFT-SMART

(BESSANI; SOUSA; ALCHIERI, 2014) apresenta uma ferramenta para SMR que consegue lidar com o problema de falhas bizantinas. Conforme mencionado no Capítulo 2, uma falha bizantina acontece quando uma réplica do sistema de coordenação falha e as informações acerca disto são imprecisas. Em outras palavras, é um estado onde o erro de um componente pode não ser detectado pelo restante do sistema, comprometendo assim a sua execução.

A ferramenta proposta é moderna e contém diversas funcionalidades antes não vistas no estado da arte para ferramentas de SMR com suporte a falhas bizantinas. Duas dessas são a possibilidade de reconfiguração da máquina de estados (i.e., adição ou remoção de nós pertencentes ao sistema) e o suporte para serviços duráveis (i.e., que conseguem se recuperar de falhas e lidar com grandes quantidades de dados que podem comprometer o longo prazo). O *software* foi escrito inteiramente na linguagem Java e o seu código fonte está disponível no GitHub[1], o que nos ajudou muito na etapa de desenvolvimento da nossa ferramenta.

Nosso trabalho, ao contrário do BFT-SMART, foi construído com foco na execução em microsserviços que executam em infraestruturas modernas, como por exemplo orquestradores de contêineres. Portanto, direcionamos nosso esforço para uma arquitetura que funcione em tais ambientes. Um exemplo disso é a escolha do padrão *Sidecar* para composição de contêineres. Além disso, outra diferença está na forma que modelamos nossa arquitetura para que ela possa funcionar com quaisquer serviços de difusão totalmente ordenada, deixando esta decisão nas mãos dos desenvolvedores das aplicações replicadas. O BFT-SMART por si só já é um serviço para difusão totalmente ordenada.

3.5 On the Efficiency of Durable State Machine Replication

(BESSANI et al., 2013) explica em detalhes três técnicas utilizadas no desenvolvimento da ferramenta BFT-SMART para alcançar durabilidade. São elas:

- ***Parallel logging*** - Para permitir que réplicas com erro possam recuperar o estado mais atual do sistema, é necessário que todas as mensagens que cheguem sejam gravadas em disco na forma de um *log*. Este *log* é utilizado sempre que necessário para recuperar o histórico de operações, porém sua existência implica na inserção de um *overhead* nas operações devido a latência de escrita e leitura em disco. A técnica de *Parallel logging* visa diminuir este *overhead* ao realizar a inserção de grupos de

[1] https://github.com/bft-smart

operações (i.e., diminuindo o número de escritas em disco) e o processamento delas em paralelo.

- ***Sequential checkpointing*** - Ao amarzenar *logs* com as operações realizadas na máquina de estados, o espaço em disco utilizado tende a aumentar consideravelmente. Para mitigar este problema, uma técnica comum em SMR é a de *checkpointing*. A ideia é que o estado atual da máquina de estados seja compilado e salvo em disco de forma a eliminar a necessidade de armazenar os *logs* anteriores. O problema é que algumas abordagens na literatura realizam o processo de *checkpointing* em cada réplica simultaneamente, criando um momento de indisponibilidade na ferramenta de tempos em tempos. A técnica de *Sequential checkpoint* especifica que os *checkpoints* devem ser realizados em apenas uma réplica por vez, com intervalos fixos entre execuções. Com isso, é garantido que em todo instante de tempo um quórum pode ser alcançado, eliminando assim a indisponibilidade mencionada.

- ***Collaborative state transfer*** - Quando uma réplica está em um estado de erro ou entra no sistema pela primeira vez, é necessário transferir o estado correto e mais atual da máquina de estados para ela. Este processo é realizado após a transferência dos dados do último *checkpoint* para a réplica em questão. A forma mais usual de realizar isto é transferindo o *checkpoint* diretamente de uma réplica para outra, o que funciona porém degrada a performance do sistema, além de também não funcionar com a técnica de *Sequential checkpointing*. Este trabalho propõe um novo mecanismo onde a transferência de estado acontece de forma colaborativa entre mais de uma réplica do sistema, diminuindo a degradação em performance devido a sua natureza de divisão e conquista.

Apesar de não implementarmos todos as técnicas apresentadas neste trabalho, ele serviu de inspiração para a construção do nosso modelo de recuperação dos processos, explicados no Capítulo 4.

3.6 Paxos Made Live – An Engineering Perspective

(CHANDRA; GRIESEMER; REDSTONE, 2007) traz uma visão prática para o problema de construção de sistemas tolerantes da falhas. Tradicionalmente, artigos da área de sistemas distribuídos explicam suas soluções sob uma ótica teórica que não leva em consideração problemas inerentes da arte de desenvolvimento de *software*. Portanto, o artigo supriu esta deficiência em conteúdos práticos mencionando diversos desafios e soluções encontradas durante o desenvolvimento de um banco de dados distribuído que utiliza o algoritmo Paxos na Google (CHANG et al., 2008; BURROWS, 2006). Problemas como a corrupção de disco em um banco de dados, pertencimento a um grupo, *checkpointing*,

testes, entre outros foram abordados. Portanto, este trabalho foi essencial para a construção do nosso, pois pudemos ter contato com experiências prévias no assunto que nos ajudaram a não cometer erros conhecidos.

3.7 Considerações

Diversos trabalhos na literatura trouxeram soluções para replicação de máquinas de estado em contêineres e também no Kubernetes. Apesar da similaridade destes trabalhos com o nosso, entendemos que contribuímos com o estado da arte ao propor uma arquitetura extensível e que funciona em sistemas produtivos modernos sem muita intrusão nas suas aplicações. Nossa arquitetura aproveita ideias apresentadas em trabalhos anteriores e as condensa em uma solução que resolve um número maior de problemas que sistemas modernas precisam enfrentar.

Capítulo 4

K8ShMiR: Um *framework* para replicação de máquinas de estado em contêineres gerenciados pelo Kubernetes

Neste capítulo apresentamos K8ShMiR, um *framework* para replicação de microsserviços usando a técnica de replicação de máquinas de estados. Na Seção 4.1 apresentamos a arquitetura da solução, considerando mecanismos genéricos de conteinerização, orquestração de microsserviços, e difusão totalmente ordenada. Em seguida, na Seção 4.2, instanciamos nossa proposta usando contêineres Docker sob o sistema Kubernetes e usando a implementação do protocolo Raft provida pelo *framework* Atomix. Em seguida, na Seção 4.3, apresentamos uma aplicação que se integra com o *framework* K8ShMiR e, por fim, na Seção 4.4 descrevemos a infraestrutura de execução de testes para validação do *framework*.

4.1 Arquitetura

Concebemos uma arquitetura para replicação de máquinas de estado de forma modular, nos beneficiando da portabilidade da tecnologia de contêineres. Por este motivo, entendemos ser possível instanciá-la em diversas infraestruturas, seja em orquestradores de contêineres (e.g., Nomad, Docker Swarm, Kubernetes, etc.) ou em quaisquer servidores que possuam suporte para esse tipo de virtualização. Além disso, uma arquitetura baseada em contêineres também permite que, caso desejado, sejam utilizados padrões de projeto costumeiramente encontrados em microsserviços, como por exemplo o padrão *Sidecar*, para adição de novas funcionalidades a uma aplicação através de contêineres, e o padrão *Ambassador*, onde um contêiner acoplado a aplicação é responsável por realizar as integrações com serviços externos (BURNS; OPPENHEIMER, 2016). Partindo deste

ponto, a seguir explicaremos as principais características da arquitetura proposta, além de apresentar seus componentes e como eles interagem entre si.

O principal requisito da nossa arquitetura é que ela funcione com microsserviços sem exigir muitas alterações de código. Queremos que, sempre que possível, ela seja transparente para a aplicação e usuários que a utilizam. Almejar transparência total, ou seja, que ambos possam operar completamente alheios ao *framework* de *SMR*, é impraticável pois a inserção de camadas sujeitas a falhas entre clientes e servidores exige que tais falhas sejam tratadas. O mesmo raciocínio é usado por (STEEN; TANENBAUM, 2017) para justificar a impossibilidade de total transparência de distribuição. Ainda que almejar um grau de intrusão completamente nulo não seja factível, neste trabalho minimizamos a intrusão nas aplicações que utilizarão nossa arquitetura de *SMR*. Para alcançar tal objetivo, optamos pelo uso de padrões de arquitetura eficientes na utilização de recursos computacionais, como por exemplo memória, processamento e espera por E/S (e.g., rede e camada de persistência).

4.1.1 Visão Geral

Um sistema que utiliza nossa solução para obtenção de tolerância a falhas é composto por um *middleware* e por N réplicas da aplicação. Este *middleware* é instanciado como *proxies* em contêineres ao lado de cada uma das réplicas, e possui a função de interceptar mensagens destinadas a elas. Além disso, a arquitetura possui um serviço de difusão totalmente ordenada de mensagens. Mensagens enviadas pelo cliente para a aplicação são interceptadas pelo *proxy* e redirecionadas para o serviço de difusão totalmente ordenada, que entregará as mensagens para todos os *proxies* com a garantia de ordenação total. As mensagens recebidas do serviço de ordenação são então encaminhadas pelos *proxies* para todas as N réplicas (com ou sem falhas) da aplicação.

Nossa arquitetura não impõe restrições quanto ao algoritmo de difusão totalmente ordenada implementado pelo serviço. O único requisito é que ele seja capaz de prover as garantias de difusão totalmente ordenada e prover as API necessárias para envio de mensagens pelos *proxies* e pela entrega das mensages de volta aos *proxies*.

Em resumo, este mecanismo de envio de mensagens e espera de eventos com o serviço de difusão de mensagens garante que, em algum momento, todas as réplicas da aplicação receberão as mensagens na mesma ordem. Em caso de falha, tornamos a instância do *proxy* inacessível de forma a impossibilitar que a aplicação receba novas mensagens. Ela continuará isolada até que o mecanismo de recuperação de falhas seja executado com sucesso.

A Figura 8 ilustra uma infraestrutura de servidores que utiliza a nossa arquitetura para replicação de máquinas de estado. Nela, pode-se observar 2 contêineres de nome *app* para as réplicas da aplicação, e 2 contêineres para as instâncias do *proxy* de interceptação

de mensagens. Na mesma imagem, está ilustrado o ciclo de vida de uma mensagem em etapas enumeradas de 1 até 6. A seguir, explicaremos todas elas em detalhes.

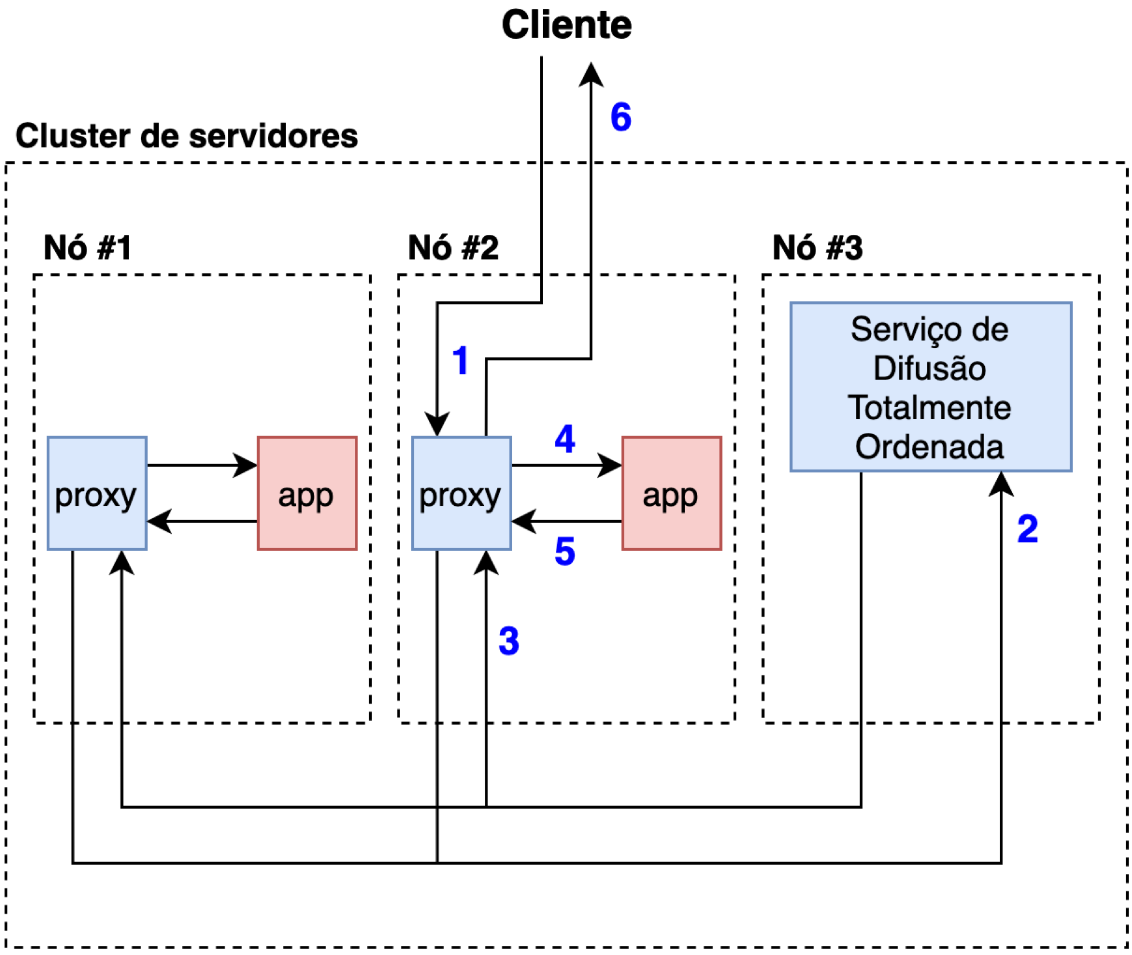

Figura 8 – A arquitetura proposta e o fluxo de mensagens no caso sem falhas.

4.1.2 O caminho de uma mensagem até o contêiner da aplicação

A etapa número 1 da Figura 8 representa o envio de uma mensagem do cliente para a aplicação replicada. Na figura, nenhum *middleware* existe entre o cliente e o *cluster* de servidores, porém, a existência de um é bastante provável em um cenário real (e.g., Balanceador de carga, *proxy* HTTP, etc) (PRADHAN; BISOY; MALLICK, 2020). Independente do mecanismo de entrega, em caso de sucesso esta mensagem sempre chegará em algum nó (i.e., máquina) e, após o seu recebimento, o sistema operacional vigente a redirecionará para o contêiner da aplicação (i.e., réplica).

A garantia de sucesso da etapa 1 não está no escopo do nosso trabalho, sendo ela de responsabilidade dos desenvolvedores da aplicação. Cada sistema requer políticas diferentes para entrada de mensagens, e nossa ferramenta não interfere nesta decisão.

4.1.3 Interceptação de mensagens pelo *proxy*

No recebimento da mensagem pelo contêiner da aplicação, esta é interceptada pelo *proxy* antes de ser encaminhada para seu destino original. Imediatamente antes de prosseguir com esta etapa, o *proxy* aciona o serviço de difusão de mensagens informando que uma nova mensagem foi recebida, de forma que ela possa ser armazenada para ser reproduzida no futuro. O serviço, então, ordena esta mensagem utilizando algum algoritmo de difusão totalmente ordenada, sendo que esta etapa é representada na Figura 8 pelo número 2.

A premissa para esta etapa funcionar é a existência de um caminho de rede entre o contêiner da aplicação e o serviço de difusão de mensagens. Portanto, não é necessário que ambos estejam no mesmo *cluster*, embora nossa recomendação seja que estejam próximos fisicamente para diminuir a latência de rede e, consequentemente, a sobrecarga introduzida pela arquitetura.

4.1.4 O encaminhamento da mensagem para a aplicação

Após ser acionado pelo *proxy*, o serviço de difusão de mensagens executa um algoritmo de difusão totalmente ordenada que resulta na ordem de entrega da nova mensagem. Após esta execução, a mensagem é emitida sob a forma de um evento em um canal aberto com cada uma das réplicas do *proxy*. Esta etapa é ilustrada pelo número 3 na Figura 8. Após serem notificadas com este evento, cada réplica do *proxy* encaminha a mensagem para a aplicação em que está acoplada, como pode ser observado na mesma figura no número 4.

Existem dois cenários possíveis para o ato de encaminhamento de uma mensagem para a aplicação. O primeiro ocorre quando a mensagem original enviada pelo cliente é recebida no mesmo *proxy* onde o encaminhamento ocorre. Neste caso, o *proxy* precisa devolver ao cliente a resposta da aplicação. Para isso, ele simplesmente a captura e a devolve de forma transparente, sendo que esta etapa pode ser vista na mesma figura nos números 5 e 6. É importante ressaltar que, enquanto a interação do *proxy* com o serviço de difusão de mensagens não terminar, a resposta não será enviada. Portanto, quanto menor a latência de rede entre ambos e menor o custo de processamento, melhor. Em caso de uma partição de rede entre o serviço de difusão de mensagens e o contêiner da aplicação, uma mensagem de erro deve ser entregue ao cliente informando que o resultado da operação não pôde ser detectado. O outro cenário ocorre quando a mensagem original foi recebida em outra réplica. Neste caso, o *proxy* encaminha a mensagem e não devolve sua resposta para o cliente. Em caso de erro de encaminhamento em ambos os cenários, então o *proxy* entra em um estado de erro e seu processo de recuperação inicia.

4.1.5 Recuperação de uma réplica fora de sincronia

Uma réplica da aplicação entra em um estado de erro quando pelo menos um de seus dois contêineres (*proxy* ou aplicação) apresenta um defeito, ou caso ocorra um erro no encaminhamento da mensagem do *proxy* para a aplicação. A nossa arquitetura, por meio deste mesmo *proxy*, realiza envios constantes de mensagens para a aplicação de forma a verificar sua disponibilidade. Estas mensagens são enviadas em intervalos de tempo fixo e também possuem como objetivo verificar qual foi a última mensagem recebida com sucesso. Internamente, o serviço de difusão totalmente ordenada possui identificadores únicos para cada mensagem, que também são utilizados pela nossa solução.

Quando um erro é detectado durante o encaminhamento de uma mensagem, quando o *proxy* ou aplicação são inicializados, ou após um evento de indisponibilidade da aplicação, a arquitetura entra no modo de recuperação e, baseado na última mensagem encaminhada com sucesso, inicializa o processo de envio das mensagens faltantes. Durante este processo, é possível que novas mensagens sejam publicadas no canal de eventos do serviço de difusão de mensagens e, por isso, elas ficam represadas até que a aplicação volte ao estado sem erros. Neste momento, todas as mensagens represadas são entregues e a aplicação volta a ficar acessível para o cliente.

É importante mencionar que após a inserção de novas réplicas, estas entram automaticamente no modo de recuperação de estado. O motivo é que o estado delas precisa alcançar o atual do sistema. Por isso, o mesmo mecanismo utilizado para recuperação de estado em caso de defeitos é utilizado neste cenário. Com isso, nossa arquitetura consegue se recuperar de diversos cenários de erro, além de permitir que a aplicação escale conforme seja necessário.

Para que a arquitetura conheça, durante o seu processo de recuperação, qual a última mensagem entregue com sucesso, é necessário que esta informação seja guardada e repassada de alguma forma. Optamos por deixar esta responsabilidade nas mãos dos desenvolvedores da aplicação replicada. O motivo é que, da mesma forma que os mecanismos de entrega de mensagens ao *cluster* podem variar em diferentes cenários, as garantias de durabilidade de estado também e, portanto, não interferimos nesta decisão. No entanto, precisamos desta informação e, devido a isso, um requisito para a utilização da nossa solução para *SMR* é a criação de uma API que entrega este dado sempre que necessário. Esta *API* é acionada pelas mensagens enviadas em intervalos de tempo fixo pelo *proxy* mencionadas nesta seção.

4.2 Implementação

Neste trabalho, instanciamos a arquitetura proposta na Seção 4.1 utilizando contêineres Docker e o orquestrador de contêineres Kubernetes. Nosso *proxy* foi desenvolvido utilizando a linguagem Golang e o serviço para difusão totalmente ordenada escolhido foi

o Atomix[1]. Além disso, optamos pela distribuição da ferramenta utilizando um gerenciador de pacotes para Kubernetes chamado Helm[2]. Este, por sua vez, não melhora o desempenho da aplicação, mas sim facilita sua adoção. Na Figura 9 é possível observar o ciclo de vida de uma mensagem em nossa implementação da arquitetura para *SMR*, com passos enumerados de 1 até 10. Da mesma forma que na Figura 8, a mensagem é enviada do cliente para o contêiner da aplicação, que agora está contido em uma *Pod* do Kubernetes juntamente com o *proxy*. Após o encaminhamento da mensagem para o serviço de difusão de mensagens Atomix, caso o recebimento do evento de inserção aconteça no *proxy*, a mensagem é encaminhada para a aplicação e a resposta é devolvida para o cliente, caso a conexão original tiver ocorrido no mesmo contêiner.

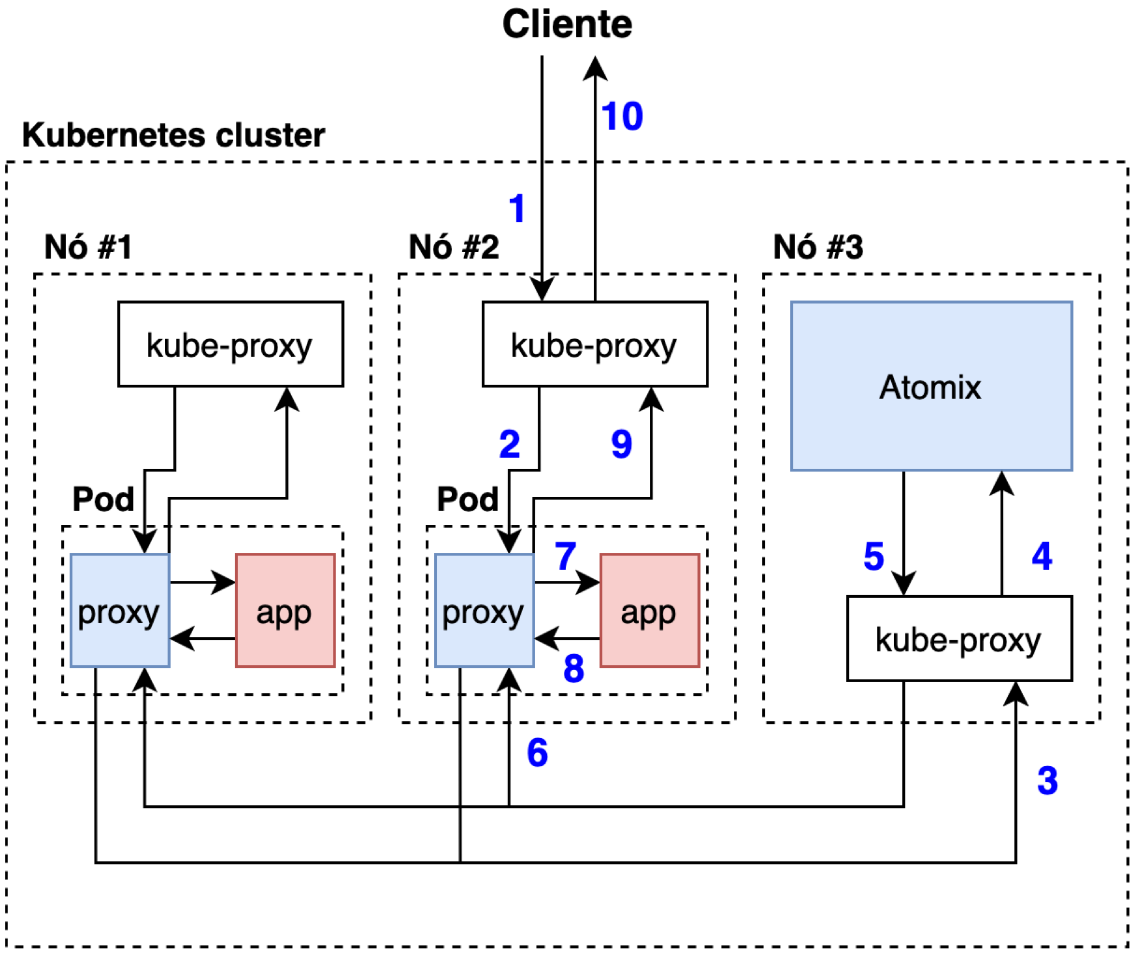

Figura 9 – O fluxo de mensagens no *framework* K8ShMiR no caso sem falhas

O nome dado para o *framework* construído é K8ShMiR, e para utilizá-lo basta adequar a aplicação replicada para conter a API necessária para recuperação de estado, além de realizar a instalação do serviço de difusão de mensagens e do *middleware* (i.e., *proxy*) para encaminhamento de mensagens. Preparamos alguns *charts* Helm para a instalação do

[1] https://github.com/atomix
[2] https://helm.sh

framework de forma simples, e de forma que seja fácil replicá-la em diferentes instalações do Kubernetes[3]. Partindo deste ponto, a seguir explicaremos a nível de código cada um dos passos descritos na Figura 9.

4.2.1 Proxy

O componente *proxy* é responsável por interceptar mensagens que possuem como destinatário a aplicação em que está acoplado. Este acoplamento acontece no contexto de uma *Pod* no Kubernetes, através do padrão *Sidecar* para composição de contêineres. A ferramenta *iptables* foi utilizada para realizar esta tarefa, e o comando executado pode ser visto a seguir:

- **iptables -t nat -A PREROUTING -p tcp -i eth0 –dport <porta-aplicacao> -j REDIRECT –to-port <porta-proxy>**

Após a mensagem ser interceptada, ela é processada e enviada ao Atomix, sendo que assim que possível, ela é encaminhada para a aplicação. No Algoritmo 4.1 podemos visualizar este processo na forma de pseudocódigo. No contexto de uma *Pod* que recebeu a mensagem original, precisamos garantir que a mensagem só será encaminhada após o aval do Atomix. Para isso, na linha 13, a execução do *proxy* é bloqueada até que uma notificação para prosseguimento aconteça. Isto garante que entregaremos todas as mensagens na ordem correta, em um cenário onde a mensagem original pode ou não ser entregue a *Pod* atual. Esta notificação acontence após a emissão do evento de inserção de mensagem no Atomix, através de um canal aberto em cada uma das réplicas do *proxy*.

No mesmo Algoritmo 4.1, caso um erro de encaminhamento ocorra na linha 14, então a réplica é movida para o estado de erro. Enquanto estiver neste estado, o *proxy* e consequentemente a aplicação não receberão mensagens do cliente diretamente. Este mecanismo é possível devido a uma funcionalidade do Kubernetes chamada *readinessProbe*[4]. Com ela, é possível definir condições para o recebimento de mensagens em uma *Pod*. No nosso caso, esta condição é definida pelo estado atual do *proxy*, que pode ser um de dois possíveis: com, ou sem erro.

Como mencionado anteriormente, o encaminhamento de uma mensagem para a aplicação só ocorre após o aval do Atomix. Este aval é dado através de sua API de eventos, que emite uma notificação de inserção após uma rodada do algoritmo Raft decidir, por quórum, pela incorporação de uma nova mensagem. O Algoritmo 4.2 ilustra em forma de pseudocódigo o módulo do *proxy* responsável pela integração com esta API. Para cada novo evento, caso este seja um de inserção, é verificado se a mensagem foi recebida originalmente pela *Pod* atual. Caso positivo, a notificação necessária para o prosseguimento do Algoritmo 4.1 é criada. No outro cenário, quando a mensagem não foi recebida pela

[3] https://github.com/lucasbfernandes/k8shmir
[4] https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

Algoritmo 4.1 Interceptação da mensagem pelo *proxy*

```
1: função INTERCEPTAMENSAGEM
2:     idMensagem ← uuidv4()
3:     mapaMensagens[idMensagem] ← criaCanal()          ▷ Variável global para notificações de prosseguimento

4:     objetoMensagem, erro ← constroiObjMensagem(idMensagem)
5:     se erro ≠ nulo então
6:         retornaErro()                                ▷ Retorna erro caso construção do objeto falhe
7:     fim se

8:     erro ← enviaAtomix(objetoMensagem)
9:     se erro ≠ nulo então
10:        retornaErro()                                ▷ Retorna erro caso o envio ao Atomix não funcione
11:    fim se

12:    canalEspera ← mapaMensagens[idMensagem]
13:    esperaAtomix(canalEspera)                        ▷ Aguarda a notificação de prosseguimento para a mensagem

14:    resposta, erro ← encaminhaMensagem(objetoMensagem)
15:    se erro ≠ nulo então
16:        moveReplicaEstadoErro()
17:        retornaErro()                                ▷ Retorna erro caso encaminhamento falhe
18:    fim se

19:    devolveResposta(resposta)                        ▷ Devolve resposta da aplicação para o cliente original
```

Pod atual, primeiro é verificado se o estado do *proxy* é de erro. Caso positivo, a mensagem do evento é represada em uma estrutura de fila que será utilizada posteriormente. Caso não haja erros no *proxy*, a mensagem é encaminhada para o contêiner da aplicação.

Por fim, o Algoritmo 4.3 ilustra o processo de monitoramento de estado e disponibilidade de uma réplica. Nele, é possível observar na linha 2 a configuração que faz com que a função *heartbeat* execute em intervalos de 1 segundo. Nesta função, uma mensagem é enviada para o contêiner da aplicação, cujo propósito é verificar sua disponibilidade, além de receber a informação de qual foi a última mensagem recebida com sucesso. Caso haja uma falha neste passo, então o *proxy* é movido para o estado de erro e a execução da função é interrompida. Caso contrário, e a aplicação estiver em um estado de erro, então o processo de recuperação é iniciado. Este processo é ilustrado pelo Algoritmo 4.4 onde, primeiro, é feita uma consulta ao Atomix para verificar qual a última mensagem conhecida por ele. Após isso, todas as mensagens entre a última recebida pela aplicação, e a última do Atomix, são encaminhadas. Durante esta etapa, é possível que novas mensagens sejam enviadas para o Atomix através de outras réplicas do *proxy*. Por isso, as réplicas em estado de erro represam as mensagens entrantes pela API de eventos em uma estrutura de dados do tipo fila, como pode ser observado no Algoritmo 4.2. O passo final da recuperação é percorrer esta fila e encaminhar cada uma de suas mensagens. Feito isso, a réplica está sincronizada e *proxy* sai do estado de erro.

Algoritmo 4.2 Escuta de eventos de inserção do Atomix

```
1: função escutaEventos
2:     canalEventos ← atomix.canalEventos()
3:     para todo evento ∈ canalEventos faça
4:         se evento.Tipo ≠ Insercao então
5:             retorna                          ▷ Somente eventos de inserção interessam
6:         fim se

7:         idMensagem ← evento.Valor.Id
8:         se idMensagem ∈ mapaMensagens então
9:             canalEspera ← mapaMensagens[idMensagem]
10:            emiteNotificacaoProsseguimento(canalEspera)
11:            retorna                          ▷ Mensagem original foi recebida nesta réplica
12:        fim se

13:        se replicaEmEstadoErro() então
14:            concatena(filaMensagens, evento.Valor)
15:            retorna                          ▷ Armazena mensagem para depois
16:        fim se

17:        mensagem ← reconstroiMensagem(evento.Valor)

18:        erro ← encaminhaMensagem(mensagem)
19:        se erro ≠ nulo então
20:            marcaReplicaEstadoErro()         ▷ Réplica passa ao estado de erro
21:        fim se
22:    fim para
```

Algoritmo 4.3 Algoritmo de monitoramento da réplica

```
1: função inicia
2:     executaComIntervalo(heartbeat, 1)        ▷ Executa função heartbeat a cada segundo

3: função heartbeat
4:     ultimaMensagem, erro ← perguntaUltimaMensagem()
5:     se erro ≠ nulo então
6:         marcaReplicaEstadoErro()             ▷ Réplica passa para o estado de erro
7:         retorna
8:     fim se

9:     se replicaEmEstadoSemErro() então
10:        retorna                              ▷ Réplica já está no estado sem erros
11:    fim se

12:    erro ← sincronizaReplica(ultimaMensagem)
13:    se erro ≠ nulo então
14:        marcaReplicaEstadoErro()             ▷ Réplica passa ao estado de erro
15:        retorna
16:    fim se

17:    marcaReplicaEstadoSemErro()              ▷ Réplica passa ao estado sem erros
```

4.2.2 Integração com o Atomix

Neste trabalho, utilizamos a versão 4 do Atomix, a mais recente, e que trouxe uma reescrita completa para a linguagem Golang. Nossa única recomendação para sua insta-

Algoritmo 4.4 Algoritmo de sincronização de estado

```
1: função SINCRONIZAREPLICA(ULTIMAMENSAGEM)
2:     ultimaMensagemAtomix, erro ← atomix.ultimaMensagem()
3:     se erro ≠ nulo então
4:         retornaErro()
5:     fim se                                  ▷ Busca a última mensagem recebida pelo Atomix

6:     para todo idMensagem ∈ (ultimaMensagem, ultimaMensagemAtomix] faça
7:         objetoAtomix, erro ← atomix.buscaMensagem(idMensagem)
8:         se erro ≠ nulo então
9:             retornaErro()
10:        fim se

11:        erro ← encaminhaMensagem(objetoAtomix)
12:        se erro ≠ nulo então
13:            retornaErro()
14:        fim se
15:    fim para        ▷ Mensagens entre a última encaminhada e o última conhecida pelo Atomix

16:    para todo objetoAtomix ∈ filaMensagens faça
17:        se objetoAtomix.Id ≤ ultimaMensagemAtomix.Id então
18:            removeDaFila(objetoAtomix)
19:            continua                        ▷ Devemos encaminhar apenas novas mensagens
20:        fim se

21:        erro ← encaminhaMensagem(objetoAtomix)
22:        se erro ≠ nulo então
23:            retornaErro()
24:        fim se

25:        removeDaFila(objetoAtomix)
26:    fim para          ▷ Encaminhamento de mensagens represadas pelo Algoritmo 4.2
```

lação é que ela seja feita em um local fisicamente próximo ao *proxy*. Isso se dá devido aos requisitos de baixa latência do *framework* K8ShMiR. Para facilitar o processo de instalação, construímos um pacote Helm com os 3 componentes que compõem o Atomix. São eles:

- **atomix-controller** - Gerencia os recursos Kubernetes do Atomix e disponibiliza a API que é utilizada pelo *proxy*;
- **raft-storage-controller** - Gerencia os recursos Kubernetes relacionados ao Raft no Atomix;
- **raft-database** - Processo que implementa o algoritmo Raft.

Após a instalação, basta configurar as seguintes variáveis de ambiente no contêiner do *proxy*:

- **ATOMIX_DB_NAME** - Nome do recurso *Service* do Kubernetes criado para o componente *raft-database*. Tem como valor padrão o mesmo nome;

- ❑ **ATOMIX_CONTROLLER_ADDRESS** - Endereço do componente *atomix-controller*. Tem como valor padrão: *atomix-controller.default.svc.cluster.local:5679*;
- ❑ **ATOMIX_LOG_PRIMITIVE_NAME** - Nome da primitiva de *log* a ser utilizada no Atomix. Tem como valor padrão o nome *request-logs*.

Uma vez que estas variáveis estejam preenchidas e o Atomix instalado corretamente, o *proxy* irá conseguir utilizar as APIs necessárias para seu funcionamento.

4.2.3 Organização do *framework* e seu impacto na adoção

Um dos objetivos principais deste trabalho foi conceber uma ferramenta pouco intrusiva e de fácil adoção. Naturalmente, ao utilizar o Kubernetes e um gerenciador de pacotes como o Helm, já ganhamos o benefício de uma infraestrutura modular com vários pontos de customização. Aliado a isso, nos inspiramos na arquitetura de malhas de serviço como o Istio[5], que também utiliza o padrão *Sidecar* com o Envoy[6] para interceptação de mensagens destinadas aos microsserviços gerenciados. Por fim, a utilização de contêineres Docker permite que, caso necessário, o Kubernetes seja substituído por outra solução de infraestrutura.

O desenvolvimento do *framework* K8ShMiR foi feito de tal forma a diminuir o número amarras técnicas. O Atomix, talvez, seja sua maior dependência. No entanto, construímos uma camada de serviço a nível de código no *proxy*, para facilitar a troca desta ferramenta, caso necessário.

4.3 Aplicação

Para se integrar com o *framework* K8ShMiR, uma aplicação deve implementar a API para recebimento das mensagens constantes do *proxy*, durante o processo ilustrado no Algoritmo 4.3. Essa API, como mencionado anteriormente, possui como finalidade verificar a disponibilidade da aplicação, além de informar qual a sua última mensagem recebida com sucesso. O Algoritmo 4.5 ilustra em forma de pseudocódigo esta API e, como pode ser observado, na linha 4 existe uma função que a implementa. Esta função, após ser acionada, realiza uma busca interna para recuperar a informação de qual a última mensagem recebida. Esta busca pode ou não ser realizada em disco rígido, e entendemos que não cabe a nós tomar esta decisão para a aplicação, uma vez que requisitos de durabilidade podem variar.

No mesmo algoritmo, na linha 2, uma API qualquer do domínio das regras de negócio da aplicação é ilustrada. Esta pode tomar qualquer forma, sendo sua única responsabilidade salvar o identificador da mensagem que está recebendo, enviado pelo *proxy* em

[5] https://istio.io/
[6] https://www.envoyproxy.io

algum campo de metadado da mensagem. Novamente, a aplicação tem liberdade total na decisão de como armazenar esta informação.

Algoritmo 4.5 Implementação exemplo de uma aplicação replicada

```
1: função RECEBEMENSAGEM(MENSAGEM)
2:     salvaUltimaMensagem(mensagem)              ▷ Guarda a última mensagem
3:     executaRegraDeNegocio(mensagem)            ▷ Executa regras de negócio da aplicação
4: função RESPOSTAHEARTBEAT()
5:     mensagem ← buscaUltimaMensagem()
6:     respondeComUltimaMensagem(mensagem)        ▷ Responde qual a última mensagem ao proxy
```

Para auxiliar na visualização desta integração com o *framework* K8ShMiR, a Figura 10 mostra uma aplicação real escrita na linguagem Javascript que implementa a API mencionada acima. Por simplicidade, optamos por não realizar integrações com bancos de dados, portanto a informação da última mensagem recebida é mantida em uma variável global nomeada *lastIndex*. A API acionada pelo *proxy* é identificada pelo nome "/last-index", logo no início da Figura. Por fim, temos duas APIs nomeadas "/integer"e "/integer/reset"que representam as regras de negócio da aplicação, e também atualizam o valor da variável *lastIndex* com o valor recebido do *proxy* no cabeçalho da mensagem HTTP.

4.4 Validação

Para a validação de que o *framework* K8ShMiR funciona sob situações de estresse, foi necessário criar uma infraestrutura de testes integrados. Para isso, utilizamos um *cluster* Kubernetes local e a biblioteca Testify da linguagem Golang. A ferramenta Kind[7] foi a escolhida para a criação do *cluster*, uma vez que ela possibilita que estes sejam criados em contêineres Docker, de uma forma econômica para o sistema operacional.

Um *script* de configuração foi criado para preparar o ambiente de testes. Sua função é instalar uma aplicação exemplo replicada junto com o serviço Atomix. A aplicação é muito simples e possui uma API para o manuseio de uma variável em memória que guarda um valor inteiro. Este valor é inicializado com o valor 0 e pode ser incrementado ou decrementado dependendo da mensagem recebida. Esta aplicação é a mesma ilustrada pela Figura 10.

Em nossos testes, criamos 2 réplicas para a aplicação e as nomeamos de *counter1* e *counter2*. Em cada teste, interagimos com ambas as réplicas de forma a verificar o valor do inteiro após diferentes situações enumeradas a seguir:

[7] https://kind.sigs.k8s.io/

1. Uma série de mensagens entregues à réplica *counter1* devem ser propagadas para a outra réplica;

2. Mensagens enviadas em paralelo que incrementam ou decrementam o valor do inteiro nas duas réplicas, concorrentemente, devem resultar no mesmo valor final para ambas;

3. A réplica *counter2* deve se recuperar de uma falha proposital em seu contêiner da aplicação, enquanto uma bateria de mensagens para a outra réplica acontece;

4. Da mesma forma que o teste anterior, neste caso, a réplica *counter2* deve se recuperar de uma falha em seu *proxy*, enquanto a outra réplica recebe mensagens; e,

5. Durante um erro em um dos dois contêineres, a réplica *counter2* deve se recuperar enquanto ela, e a outra réplica, recebem mensagens em paralelo.

4.4.1 Descrição dos testes

O teste de número 1 é o mais simples e visa garantir que uma série de mensagens recebidas pela réplica *counter1* também serão recebidas na mesma ordem pela réplica *counter2*. Este teste não se preocupa com possíveis defeitos e consequentemente não testa o módulo de recuperação de falhas. Além disso, neste teste as mensagens são enviadas sequencialmente para uma mesma réplica, simulando um cenário sem muitas adversidades para o *framework* de replicação de máquinas de estado.

No teste 2 também verificamos a ordem de recebimento de mensagens, porém as enviando em paralelo para ambas as réplicas. O objetivo deste teste é garantir que condições de corrida não aconteçam e que a propriedade de difusão totalmente ordenada seja garantida nessa situação. Além do envio em paralelo, também alternamos as operações realizadas no valor inteiro de forma probabilística, removendo a previsibilidade do teste.

O teste 3 simula um defeito no contêiner da aplicação na réplica *counter2*. Durante a etapa de recuperação de falhas, novas mensagens são enviadas para a réplica *counter1*. O objetivo deste teste é garantir que, mesmo durante o processo de recuperação, novas mensagens serão contabilizadas pela réplica em estado de erro.

Da mesma forma, o teste 4 também simula um defeito em um contêiner enquanto envia mensagens para a outra réplica. A diferença deste em comparação com o teste 3 é que o defeito se dá no contêiner do *proxy*. Com isso, garantimos que defeitos em ambos os contêineres resultam no estado correto após o processo de recuperação da réplica.

Por fim, o teste 5 simula um defeito em um contêiner de uma réplica, enquanto uma bateria de mensagens é enviada em paralelo para ambas as réplicas. Dessa forma, garantimos que, caso uma réplica receba mensagens durante o seu processo de recuperação, isto não comprometerá o estado final do sistema como um todo.

4.4.2 Execução e resultado dos testes

Os cenários de teste foram executados programaticamente com o auxílio de *scripts* que simulam erros individuais em cada um dos contêineres. A Figura 11 mostra um teste unitário real correspondente ao teste de número 3, descrito anteriormente acima. É possível observar que um *script* é executado para forçar a falha do contêiner. Além disso, este *script* também aguarda a reinicialização do contêiner antes de prosseguir com o teste. No mesmo teste, mensagens alternadas para incrementar ou decrementar o valor do inteiro em 7 ou 11 são enviadas para a réplica *counter1*. Caso uma falha aconteça, o teste falhará imediatamente. Por fim, após o envio das mensagens e a reinicialização da réplica *counter2*, é verificado se seu estado é idêntico ao da réplica *counter1*. Caso não seja, a recuperação de falhas não terá funcionado e um erro acontece.

Os testes integrados foram configurados para executar assim que uma nova alteração de código fosse sugerida para o *framework* K8ShMiR. Esta sugestão é feita por meio de um *pull request* na ferramenta de versionamento GitHub[8]. Dessa forma, foi possível garantir que todas alterações de código passariam pelo crivo dos testes, possibilitando assim um controle maior de qualidade. Em relação a aplicação escolhida, entendemos que ela não possui as complexidades e nuances do mundo real. Porém, ainda assim ela possui todas as características fundamentais de uma aplicação web qualquer. Um benefício de sua escolha para os testes realizados é a sua velocidade de resposta, que faz com que a troca de mensagens seja mais rápida e, portanto, mais propensa a condições de corrida no momento do processamento das mensagens pelo *framework* K8ShMiR. A versão atual do K8ShMiR[9] passa em todos os testes propostos.

Para fazer uma avaliação inicial do *overhead* introduzido pelo nosso *framework*, executamos um teste simples na em que 10000 mensagens foram enviadas para cada uma das duas réplicas da aplicação de testes descrita anteriormente, onde um número inteiro é armazenado em memória primária e APIs para sua adição e subtração existem. Também executamos o mesmo teste com a mesma aplicação mas com clientes fazendo invocações diretamente a uma instância do servidor, sem o uso do K8ShMiR. No cenário com uso do K8ShMiR o tempo médio de resposta foi de 14,495ms e, no caso sem o *framework* 3.76 ms. Ou seja, o *framework* K8ShMiR introduziu um *overhead* de aproximadamente 10 milissegundos no tempo de processamento da mensagem, o que pode ou não ser ignorável, dependendo da aplicação, e que deve ser considerado na análise do custo/benefício da implementação de tolerância a faltas. Os testes foram executados em um *cluster* Kubernetes local utilizando a ferramenta Kind, e o serviço Atomix foi instanciado neste mesmo *cluster*, diminuindo a latência de rede entre ele e o *proxy*. Sendo assim, entendemos que caso os usuários de nosso *framework* mantiverem esta proximidade física entre o Atomix e as aplicações replicadas, o custo introduzido pelo *framework* não será alto.

[8] https://github.com/lucasbfernandes/k8shmir/tree/master/test
[9] https://github.com/lucasbfernandes/k8shmir/releases/tag/v1.0.0

```
const app = express()
const port = 3000

let integer = 0
let lastIndex = 0

app.get('/last-index', (req, res) => {
    res.status(200).send({ index: lastIndex })
})

app.get('/integer', (req, res) => {
    lastIndex = parseInt(req.headers['log-index'])
    res.status(200).send({ value: integer })
})

app.post('/integer', (req, res) => {
    lastIndex = parseInt(req.headers['log-index'])
    const { op, value } = req.body
    switch (op) {
        case 'INC':
            integer += value
            break
        case 'DEC':
            integer -= value
            break
        default:
            res.status(500).send({ error: 'Invalid operation' });
            break
    }

    res.status(201).send()
})

app.post('/integer/reset', (req, res) => {
    lastIndex = parseInt(req.headers['log-index'])
    integer = 0
    res.status(200).send()
})

app.listen(port, () => {
    console.log(`App running on port ${port}`)
})
```

Figura 10 – Exemplo de aplicação Javascript que utiliza o *framework* K8ShMiR

```
func (suite *CounterE2ETestSuite) TestAppFailureShouldCatchUpStateCorrectly() {
        counter1Chan := make(chan struct{}, 1)

        go func() {
                for i := 0; i < 250; i++ {
                        err := counter.DoAlternateRequest(i, counter.URL1, 7, 11)
                        assert.Nil(suite.T(), err, "request error should be nil")
                }
                counter1Chan <- struct{}{}
        }()

        err := counter.ExecuteAndWaitScriptFile(restartCounter2AppScriptPath)
        assert.Nil(suite.T(), err, "restart error should be nil")

        <-counter1Chan
        close(counter1Chan)

        c1Resp, err := counter.DoGetCounterRequest(counter.URL1)
        assert.Nil(suite.T(), err, "request error should be nil")

        c2Resp, err := counter.DoGetCounterRequest(counter.URL2)
        assert.Nil(suite.T(), err, "request error should be nil")

        assert.Equal(suite.T(), c1Resp.Value, c1Resp.Value, "values should be equal")
}
```

Figura 11 – Teste de integração que valida a recuperação de uma réplica com erro

Capítulo 5

Conclusão

Conforme mencionado nos Capítulos 1 e 4, o objetivo principal deste trabalho foi entregar uma ferramenta para *SMR* que fosse, sempre que possível, transparente para a aplicação replicada e os usuários que a utilizam. Com relação a este objetivo, ao utilizarmos o padrão *Sidecar* para composição de contêineres, conseguimos que todas as mensagens destinadas a aplicação fossem interceptadas automaticamente pelo *proxy*, sem necessidade de reconfigurar a aplicação, ou que os clientes soubessem desse processo. A aplicação precisou ser estendida para implementar uma API para que lidasse com defeitos e recuperação após os mesmos, mas a API é mínima e exige essencialmente que aplicação registre o progresso do processamento para que possa indicar um ponto de recuperação.

Como objetivo secundário, nos propomos a entregar um *framework* que introduz um *overhead* aceitável no tempo de processamento de uma mensagem, sem inviabilizar sua utilização em sistemas produtivos modernos. Entendemos que alcançamos ambos objetivos, pois, após a conclusão do desenvolvimento e execução de testes de estresse, a replicação de estado funcionou da forma esperada, mesmo em cenários onde falhas acontecem. Além disso, mesmo com o envio simultâneo de mensagens para várias réplicas diferentes, o estado final sempre foi o mesmo, assegurando que as garantias necessárias de entrega ordenada foram cumpridas. Em relação ao *overhead* introduzido, nossos testes detectaram que aproximadamente 10 milissegundos são adicionados no tempo de resposta de uma mensagem, caso as recomendações de instalação sejam cumpridas. Entendemos que este custo não é caro dado o nível de garantias oferecidas. Sendo assim, concluímos que os objetivos propostos foram alcançados e a hipótese colocada provada.

Como contribuição prática, consideramos que o *framework* K8ShMiR pode ser facilmente estendido, seja para adição de novas funcionalidades, ou para reaproveitamento de sua arquitetura pouco intrusiva. Por isso, o disponibilizamos em um repositório Git público[1], para que a comunidade possa, caso seja de seu interesse, contribuir com seu desenvolvimento.

[1] https://github.com/lucasbfernandes/k8shmir

Principais Contribuições

Em resumo, a principal contribuição deste trabalho para o estado da arte foi a construção do *framework* K8ShMiR para aplicações replicadas em contêineres no Kubernetes. De forma mais granular, o desenvolvimento dela possibilitou as seguintes contribuições:

- ❏ Prova de conceito que mostra ser possível aliar ferramentas modernas com o rigor teórico do modelo de *SMR*, sem perda de desempenho e de forma simples;
- ❏ Ferramenta extensível e agnóstica a servidores, facilitando sua adoção;
- ❏ Solução pouco invasiva para a aplicação replicada e usuários que a utilizam. A única alteração necessária é a construção da API para recuperação de falhas, mencionada no Capítulo 4;
- ❏ Ambiente de testes integrados que utiliza um *cluster* Kubernetes real instalado na máquina do desenvolvedor. Criamos *scripts* que simulam falhas individuais em contêineres e disparam mensagens automáticas para as *Pods* replicadas.

Trabalhos Futuros

Após o desenvolvimento deste trabalho, alguns pontos de melhoria ficaram claros para nós. Primeiro, apesar de nossos testes com uma aplicação Javascript indicarem que de fato há pouca intrusão, entendemos ser necessário que terceiros realizem este mesmo teste para dar mais força a nossa afirmação. No entanto, estamos seguros de que alcançamos o objetivo de prover *SMR* através de um *framework* que não requer muitas alterações nas aplicações que o utilizam, além de ser transparente para seus clientes. O motivo para esta segurança é o fato de que o *framework* K8ShMiR necessita apenas da construção de uma API e do armazenamento do identificador da última mensagem recebida com sucesso. Este armazenamento, conforme discutido anteriormente neste trabalho, pode ser feito de acordo com as necessidades da aplicação replicada, sem nenhuma interferência do nosso *framework* nesta decisão.

O *framework* K8ShMiR, hoje, funciona apenas em *clusters* Kubernetes. Como mencionado anteriormente, entendemos que a migração para outros orquestradores de contêineres não seria complexa, mas ainda assim um trabalho a ser feito. Tornar a ferramenta genérica e compatível com diversos orquestradores de contêineres é um dos trabalhos futuros possíveis. Além disso, outro trabalho possível é tornar a ferramenta genérica e compatível com outros serviços de algoritmos de consenso. Hoje, nossa solução funciona apenas com o serviço Atomix.

Outra direção que a ferramenta pode tomar é a sua incorporação em algum *framework* para microsserviços, como o Dapr[2]. Ele oferece diversos módulos que resolvem proble-

[2] https://dapr.io/

mas conhecidos de arquiteturas de microsserviços, como por exemplo: *Service Discovery*, *Observability*, Serviços *Publisher/Subscriber*, entre outros. Na nossa visão, *SMR* com o *framework* K8ShMiR poderia ser um dos serviços ofertados pelo *framework*.

Finalmente, compreendemos a necessidade da comunicação científica e por isso estamos já preparando uma submissão que descreve a arquitetura e outros resultados sendo derivados da mesma. É importante notar que apesar de nenhuma publicação sobre K8ShMiR ter sido ainda feita, uma publicação sobre o tema inicial deste mestrado, Redes Definidas por Software utilizando a linguagem P4, já foi alcançada (FERNANDES; CAMARGOS, 2020), inclusive aparecendo em citações de trabalhos recentes na literatura (HAUSER et al., 2021).

Formação Humana

Por fim, no aspecto da formação do pesquisador, o desenvolvimento deste trabalho possibilitou que aplicássemos o rigor teórico da área de sistemas distribuídos em um cenário prático, com uma ferramenta que pode ser utilizada em arquiteturas modernas que executam em *clusters* Kubernetes. Também entendemos que, devido a arquitetura ser baseada em contêineres Docker, não seria complexo realizar uma migração para outro orquestrador de contêineres, como por exemplo o Nomad [3]. Devido a natureza dos problemas que a área de sistemas distribuídos se propõe a resolver, acreditamos que ela pode se beneficiar muito de trabalhos com um viés prático como este que apresentamos.

[3] https://www.nomadproject.io/

Referências

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, v. 1, n. 1, p. 11–33, 2004.

BASKARADA, S.; NGUYEN, V.; KORONIOS, A. Architecting microservices: Practical opportunities and challenges. **Journal of Computer Information Systems**, v. 60, p. 428 – 436, 2020.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: **Proceedings of the third workshop on Hot topics in software defined networking**. [S.l.: s.n.], 2014. p. 1–6.

BESSANI, A. et al. On the efficiency of durable state machine replication. In: **2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)**. [S.l.: s.n.], 2013. p. 169–180.

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with bft-smart. In: IEEE. **2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. [S.l.], 2014. p. 355–362.

BESSANI, A. N.; ALCHIERI, E. A guided tour on the theory and practice of state machine replication. In: CITESEER. **Tutorial at the 32nd Brazilian symposium on computer networks and distributed systems**. [S.l.], 2014.

BETZ, M. **Understanding kubernetes networking: ingress**. 2017. <https://medium.com/google-cloud/understanding-kubernetes-networking-ingress-1bc341c84078>. [Online; accessed 09-November-2021].

_____. **Understanding kubernetes networking: pods**. 2017. <https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727>. [Online; accessed 09-November-2021].

_____. **Understanding kubernetes networking: services**. 2017. <https://medium.com/google-cloud/understanding-kubernetes-networking-services-f0cb48e4cc82>. [Online; accessed 09-November-2021].

BORGES, F. et al. Transparent state machine replication for kubernetes. In: SPRINGER. **International Conference on Advanced Information Networking and Applications**. [S.l.], 2019. p. 859–871.

BURNS, B.; OPPENHEIMER, D. Design patterns for container-based distributed systems. In: **8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)**. [S.l.: s.n.], 2016.

BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In: **Proceedings of the 7th symposium on Operating systems design and implementation**. [S.l.: s.n.], 2006. p. 335–350.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. Consensus. In: _____. **Introduction to Reliable and Secure Distributed Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 203–279. ISBN 978-3-642-15260-3. Disponível em: <https://doi.org/10.1007/978-3-642-15260-3_5>.

_____. Reliable broadcast. In: _____. **Introduction to Reliable and Secure Distributed Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 73–135. ISBN 978-3-642-15260-3. Disponível em: <https://doi.org/10.1007/978-3-642-15260-3_3>.

CAMARGOS, L. J. **Multicoordinated agreement protocols and the log service**. Tese (Doutorado) — Università della Svizzera italiana, 2008.

CASALICCHIO, E. Container orchestration: A survey. **Systems Modeling: Methodologies and Tools**, Springer, p. 221–235, 2019.

CASINO, F.; DASAKLIS, T. K.; PATSAKIS, C. A systematic literature review of blockchain-based applications: Current status, classification and open issues. **Telematics and informatics**, Elsevier, v. 36, p. 55–81, 2019.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: **Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing**. [S.l.: s.n.], 2007. p. 398–407.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 43, n. 2, p. 225–267, 1996.

CHANG, F. et al. Bigtable: A distributed storage system for structured data. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 2, jun. 2008. ISSN 0734-2071. Disponível em: <https://doi.org/10.1145/1365815.1365816>.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Totally ordered broadcast and multicast algorithms: A comprehensive survey. 01 2000.

_____. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys (CSUR)**, ACM, v. 36, n. 4, p. 372–421, 2004.

DESAI, A. et al. Hypervisor: A survey on concepts and taxonomy. **International Journal of Innovative Technology and Exploring Engineering**, Citeseer, v. 2, n. 3, p. 222–225, 2013.

FERNANDES, L. B.; CAMARGOS, L. J. Bandwidth throttling in a p4 switch. **2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)**, p. 91–94, 2020.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 32, n. 2, p. 374–382, 1985.

GALÁN-JIMÉNEZ, J.; GAZO-CERVERO, A. Overlay networks: overview, applications and challenges. **IJCSNS**, v. 10, n. 12, p. 40, 2010.

HAUSER, F. et al. A survey on data plane programming with p4: Fundamentals, advances, and applied research. **ArXiv**, abs/2101.10632, 2021.

HUNT, P. et al. Zookeeper: Wait-free coordination for internet-scale systems. In: **USENIX annual technical conference**. [S.l.: s.n.], 2010. v. 8, n. 9.

KHAN, A. Key characteristics of a container orchestration platform to enable a modern application. **IEEE cloud Computing**, IEEE, v. 4, n. 5, p. 42–48, 2017.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, ACM, v. 21, n. 7, p. 558–565, 1978.

_____. The part-time parliament. **ACM Trans. Comput. Syst.**, v. 16, p. 133–169, 1998.

_____. Generalized consensus and paxos. In: . [S.l.: s.n.], 2005.

LAMPORT, L.; SHOSTAK, R. E.; PEASE, M. C. The byzantine generals problem. **ACM Trans. Program. Lang. Syst.**, v. 4, p. 382–401, 1982.

LARSSON, L. et al. Impact of etcd deployment on kubernetes, istio, and application performance. **Software: Practice and Experience**, v. 50, p. 1986 – 2007, 2020.

M., D.; KANNAN, P. A study on virtualization techniques and challenges in cloud computing. **INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH 2277-8616**, v. 3, p. 147–151, 11 2014.

MARTIN, P. Kubernetes resources. **Kubernetes**, 2020.

NETTO, H. V. et al. State machine replication in containers managed by kubernetes. **Journal of Systems Architecture**, Elsevier, v. 73, p. 53–59, 2017.

_____. Coordenação de containers no kubernetes: Uma abordagem baseada em serviço. In: **Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2017. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc/article/view/2634>.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)**. [S.l.: s.n.], 2014. p. 305–319.

PASTORE, S. The platform as a service (paas) cloud model: Opportunity or complexity for a web developer? **International Journal of Computer Applications**, v. 81, n. 18, p. 29–37, 2013.

PEDONE, F.; SCHIPER, A. Generic broadcast. In: SPRINGER. **International Symposium on Distributed Computing**. [S.l.], 1999. p. 94–106.

PERREY, R.; LYCETT, M. Service-oriented architecture. In: IEEE. **2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.** [S.l.], 2003. p. 116–119.

PINILLA, R.; GIL, M. Jvm: platform independent vs. performance dependent. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 37, n. 2, p. 44–56, 2003.

PRADHAN, A.; BISOY, S. K.; MALLICK, P. K. Load balancing in cloud computing: Survey. In: SHARMA, R. et al. (Ed.). **Innovation in Electrical Power Engineering, Communication, and Computing Technology**. Singapore: Springer Singapore, 2020. p. 99–111. ISBN 978-981-15-2305-2.

RAD, B. B.; BHATTI, H. J.; AHMADI, M. An introduction to docker and analysis of its performance. **International Journal of Computer Science and Network Security (IJCSNS)**, International Journal of Computer Science and Network Security, v. 17, n. 3, p. 228, 2017.

ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: Current technology and future trends. **Computer**, IEEE, v. 38, n. 5, p. 39–47, 2005.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 22, n. 4, p. 299–319, 1990.

SILVA, V. G. da; KIRIKOVA, M.; ALKSNIS, G. Containers for virtualization: An overview. **Appl. Comput. Syst.**, v. 23, n. 1, p. 21–27, 2018.

SONG, W.; XIAO, Z. An infrastructure-as-a-service cloud: On-demand resource provisioning. In: **Principles, methodologies, and service-oriented approaches for cloud computing**. [S.l.]: IGI Global, 2013. p. 302–324.

STEEN, M. V.; TANENBAUM, A. S. **Distributed systems**. [S.l.]: Maarten van Steen Leiden, The Netherlands, 2017.

VUČKOVIĆ, J. You are not netflix. In: _____. **Microservices: Science and Engineering**. Cham: Springer International Publishing, 2020. p. 333–346. ISBN 978-3-030-31646-4. Disponível em: <https://doi.org/10.1007/978-3-030-31646-4_13>.

VURAL, H.; KOYUNCU, M.; GUNEY, S. A systematic literature review on microservices. In: SPRINGER. **International Conference on Computational Science and Its Applications**. [S.l.], 2017. p. 203–217.

WOLFRAM, S. The problem of distributed consensus: A survey. **arXiv preprint arXiv:2106.13591**, 2021.