Lucas Vinicius Ruchel

Investigação sobre Tolerância a Falhas em Controladores SDN Distribuídos e Implementação de um Algoritmo de Difusão Atômica Hierárquico e Sem líder

Cascavel-PR

2022

Lucas Vinicius Ruchel

Investigação sobre Tolerância a Falhas em Controladores SDN Distribuídos e Implementação de um Algoritmo de Difusão Atômica Hierárquico e Sem líder

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Ciência da Computação (PPGComp) da Universidade Estadual do Oeste do Paraná – Unioeste, campus de Cascavel.

Universidade Estadual do Oeste do Paraná – Unioeste – Cascavel

Centro de Ciências Exatas e Tecnológicas – CCET

Programa de Pós-Graduação em Ciência da Computação – PPGComp

Orientador: Dr. Edson Tavares de Camargo

Coorientador: Dr. Rogério Correa Turchetti

Cascavel-PR

2022

Dedico este trabalho à minha esposa, meu filho Arthur e aos meus pais que sempre me incentivaram a estudar.

Agradecimentos

Agradeço primeiramente a Deus por toda a saúde, apesar da pandemia que assolou e continua a causar inúmeras mortes em todo o mundo.

Agradeço a minha esposa Fabiana que me acompanhou e me deu forças para que este trabalho fosse realizado. Durante o período do mestrado foram inúmeras as mudanças em nossas vidas. Além disso, surgiu o Arthur para alegrar os nossos corações e que contribui de forma muito especial na etapa final deste trabalho.

Agradeço aos meus orientadores Edson e Rogério, que sem dúvida conduziram um trabalho excelente de orientação. Foram inúmeras reuniões e tenho que certeza que várias horas de revisão de artigos, e não tenho como agradecer o empenho que dedicaram.

Agradeço aos meus colegas no IFPR, em especial a Juliana, por assumir as minhas atividades para que eu pudesse realizar exclusivamente o mestrado.

Agradeço aos membros da banca avaliadora, professores Luiz Antônio Rodrigues, Guilherme Galante e Eduardo Alchieri pelas sugestões e apontamentos realizados.

Agradeço aos meus pais, Vilson e Marta, por sempre me incentivarem a estudar e seguir o meu próprio caminho.

Agradeço a todos os professores do PPGComp que não mediram esforços para transmitir os seus conhecimentos, mesmo durante a pandemia.

Agradeço aos amigos que conheci no mestrado, apesar de breve o contato que pudemos estar juntos.

"Tarefa não é tanto ver aquilo que ninguém viu,
mas pensar o que ninguém pensou
ainda sobre aquilo que todo mundo vê."
Arthur Schopenhauer

Resumo

RUCHEL, Lucas Vinicius. **Investigação sobre Tolerância a Falhas em Controladores SDN Distribuídos e Implementação de um Algoritmo de Difusão Atômica Hierárquico e Sem líder**. Orientador: Dr. Edson Tavares de Camargo. 2022. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2022.

O avanço da Internet permitiu que uma imensa quantidade de dados fosse gerada a cada dia. A fim de absorver todo o crescimento da Internet, a infraestrutura da rede teve que aumentar tanto em capacidade quanto em número de dispositivos. Com isso, surgiram paradigmas para maximizar o gerenciamento e controle da infraestrutura de rede. Dentre os paradigmas existentes, as Redes Definidas por Software (SDN) permitem que as redes sejam flexíveis através da separação do plano de dados e do plano de controle. No entanto, distribuir o plano de controle em múltiplos controladores apresenta desafios relacionados à capacidade de tolerar falhas e a manutenção das regras e políticas de redes de maneira consistente entre os vários controladores. O objetivo desta dissertação é avaliar extensivamente o plano de controle SDN distribuído e os algoritmos para manter a consistência que permitam obter alto desempenho. Como resultado, este trabalho apresenta duas contribuições. A primeira contribuição realiza uma investigação dos mecanismos de tolerância a falhas e consistência de rede nos principais controladores distribuídos encontrados na literatura, ou seja, os controladores ONOS e Opendaylight (ODL). Ambos foram investigados e avaliados em relação aos diferentes algoritmos empregados para tolerar falhas e manter a consistência no plano de dados e no plano de controle. Os resultados obtidos demonstraram como cada um dos controladores lidou com os diferentes tipos de falhas, bem como métricas de desempenho entre eles. A segunda contribuição apresenta o projeto e avaliação de um algoritmo de difusão atômica hierárquico sem-líder (chamado de *AnyABCast*). Importante destacar que tanto ONOS quanto ODL utilizam uma abordagem de consenso em que existe um líder usando o algoritmo de consenso *Raft*. O algoritmo implementado visa distribuir o custo de ordenar as regras e políticas de rede entre todos os processos. Os experimentos foram executados em um ambiente simulado e em uma rede local. Na rede local, o algoritmo foi implementado com o apoio do *framework* distribuído *Akka.io* e comparado com uma implementação do algoritmo Raft presente no *Apache Ratis*. Os experimentos demonstraram que no ambiente simulado o *AnyABCast* apresenta desempenho superior tanto com e sem-falhas. Já no ambiente real, o *AnyABCast* apresentou desempenho superior somente na abordagem sem-falhas.

Palavras-chave: ONOS, Opendaylight, Tolerância a falhas, SDN, Difusão atômica.

Abstract

RUCHEL, Lucas Vinicius. **Investigation of Fault Tolerance in Distributed SDN Controllers and Implementation of a Hierarchical and Leaderless Atomic Broadcast Algorithm**. Orientador: Dr. Edson Tavares de Camargo. 2022. 97f. Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual do Oeste do Paraná, Cascavel – Paraná, 2022.

The growth of the Internet in recent years has meant that an immense amount of data is produced every day. The network infrastructure, due to the growth of the Internet, had to increase both in capacity and in the number of devices. As a result, new paradigms have emerged to maximize the management and control of the network infrastructure. Among the existing paradigms, Software Defined Networks (SDN) allow networks to be flexible through the separation of the data plane and the control plane. However, the control plane distributed across multiple controllers presents challenges related to being able to tolerate faults and maintaining network rules and policies consistently across multiple controllers. The objective of this master's thesis is to extensively evaluate the distributed SDN control plane and the algorithms to maintain consistency that allow achieving high performance. As a result, this work presents two contributions. The first contribution carries out an investigation of the fault tolerance and network consistency mechanisms in the main distributed controllers found in the literature, that is, the ONOS and Opendaylight (ODL) controllers. The controllers were investigated and evaluated in relation to the different algorithms used to tolerate failures and maintain consistency in the data plane and in the control plane. The results obtained showed how each of the controllers dealt with the different types of failures, as well as performance metrics between them. The second contribution presents the design and evaluation of a leaderless hierarchical atomic broadcast algorithm (called *AnyABCast*). It is important to note that both ONOS and ODL use a consensus approach in which there is a leader using the consensus algorithm *Raft*. The implemented algorithm aims to distribute the cost of ordering network rules and policies among all processes. The experiments were performed in a simulated environment and on a local network. On the local network, the algorithm was implemented with the support of the distributed *framework Akka.io* and compared with an implementation of the Raft algorithm in *Apache Ratis*. The experiments showed that in the simulated environment *AnyABCast* presents superior performance both with and without faults. In the real environment, *AnyABCast* presented superior performance only in the flawless approach.

Keywords: ONOS, ODL, fault tolerance, SDN, Atomic Broadcast

Lista de ilustrações

Figura 1 – Replicação passiva de eventos (*primary-backup*). 22
Figura 2 – Classificação dos algoritmos de ordem total (Atomic Broadcast). 25
Figura 3 – Camadas da Arquitetura SDN 27
Figura 4 – Arquitetura do Plano de Controle. 29
Figura 5 – Erros de encaminhamento causados por inconsistências entre controladores. 32
Figura 6 – Arquitetura de Comunicação do Controlador ONOS em Modo Distribuído. 35
Figura 7 – Processo de descoberta de novos controladores. 36
Figura 8 – Topologia utilizada no Caso 1. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo "X" representa o enlace interrompido. . . 40
Figura 9 – Falha de enlace sem caminhos alternativos. 41
Figura 10 – Topologia utilizada no Caso 2. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo "X" representa o enlace interrompido. . . . 42
Figura 11 – Falha de enlace com um caminho alternativo. 43
Figura 12 – Topologia utilizada no Caso 3. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo "X" representa o enlace interrompido. . . . 44
Figura 13 – Falha de enlace com dois caminhos alternativos. 45
Figura 14 – ODL - modo proativo 47
Figura 15 – Falha do *switch* 48
Figura 16 – Latência média para instalação de fluxos através da *Southbound*. . . . 52
Figura 17 – Vazão média para instalação de fluxos através da *Southbound*. 53
Figura 18 – Vazão para instalação de fluxos através da *Northbound*. 54
Figura 19 – Tempo para sincronização de regras nos controladores ONOS e ODL. . 56
Figura 20 – Tempo para sincronização de 2048 regras. 57
Figura 21 – Tempo para que novo líder seja eleito nos controladores ONOS e ODL. 59
Figura 22 – Percentual de regras instaladas após falha do controlador principal. . . 60
Figura 23 – Hipercubo com d-dimensões 68
Figura 24 – Hipercubo com dimensão $d = 3$ e falha do processo 4. 68
Figura 25 – Árvores de Difusão para um conjunto de 8 processos. 73
Figura 26 – Confirmação de recebimento (*ACKs*) na árvore de p_0. 75
Figura 27 – Encaminhamento de mensagens da origem. 77
Figura 28 – Execução sem processos falhos. 80
Figura 29 – Execução com falha do processo de origem. 82
Figura 30 – Execução com *broadcast* depois de falha em um processo ser detectada. 82
Figura 31 – Comparação da latência entre os algoritmos Raft e AnyABCast - sem falhas.´ 84

Figura 32 – Latência para entrega de mensagens após a falha de um processo comparando o Raft vs AnyABCast 85
Figura 33 – Cenário Sem Falhas. 86
Figura 34 – Cenário Com Falhas. 87

Lista de abreviaturas e siglas

SDN	Software Defined Network
ONOS	Open Networking Operating System
ODL	OpenDayLight
SBI	Southbound Interface
NBI	Northbound Interface
EWI	East/West Interface
ONF	Open Networking Foundation
API	Application Programming Interface
RESTFUL	Representational State Transfer
SNMP	Simple Network Management Protocol

Sumário

1 INTRODUÇÃO . . . 12

2 CONCEITOS BÁSICOS . . . 16

2.1 Sistemas Distribuídos . . . 16

2.1.1 Modelo de Sistema . . . 16

2.1.2 Modelos de Falhas . . . 18

2.1.3 Detectores de Falhas . . . 19

2.1.4 Consistência entre Réplicas . . . 20

2.1.4.1 O Consenso . . . 22

2.1.4.1.1 Algoritmo *Raft* . . . 23

2.1.4.2 Difusão Atômica . . . 24

2.2 SDN . . . 26

2.2.1 O Protocolo Openflow . . . 31

2.2.2 Falhas em Controladores . . . 31

2.3 Conclusão . . . 33

3 TOLERÂNCIA A FALHAS NOS CONTROLADORES SDN ONOS E ODL . . . 34

3.1 ONOS . . . 34

3.2 OpenDayLight . . . 37

3.3 Resultados . . . 38

3.3.1 Avaliação no Plano de Dados . . . 38

3.3.1.1 Caso 1 - Falha de enlace sem caminhos alternativos. . . . 40

3.3.1.2 Caso 2 - Falha de enlace com um único caminho alternativo. . . . 42

3.3.1.3 Caso 3 - Falha de enlace com dois caminhos alternativos. . . . 44

3.3.1.4 ODL - Modo proativo . . . 46

3.3.1.5 Falha do switch . . . 47

3.3.2 Avaliação no Plano de Controle . . . 50

3.3.2.1 Vazão e Latência através da *Southbound* . . . 50

3.3.2.2 Vazão através da *Northbound* . . . 54

3.3.2.3 Tempo para sincronização . . . 55

3.3.2.4 Tempo para recuperação de falha do master - Mastership Failover . . . 58

3.3.2.5 Análise de consistência na instalação de regras perante falha do controlador *master* 59

3.4 Trabalhos Relacionados . . . 61

3.5 Conclusão . . . 64

4 DIFUSÃO ATÔMICA SOBRE O VCUBE 66
4.1 Definições 66
4.1.1 Modelo de Sistema 67
4.1.2 VCube 67
4.1.3 Algoritmos de Difusão e Consenso no VCube 69
4.2 O Algoritmo 72
4.2.1 Construção da Árvore de Difusão 72
4.2.2 Algoritmo de Difusão Atômica 74
4.3 Resultados 79
4.3.1 Ambiente Simulado 79
4.3.1.1 Experimentos sem processos falhos 80
4.3.1.2 Experimentos com processos falhos 81
4.3.1.3 Comparação entre os algoritmos Raft e AnyABCast 83
4.3.2 Implementação do *AnyABCast* no *Akka.io* 85
4.4 Conclusão 88

5 CONCLUSÃO 90

REFERÊNCIAS 92

1 Introdução

A popularidade da Internet nos últimos anos fez com que a quantidade de dados gerados diariamente crescesse exponencialmente, isto exige que os dados sejam processados e transmitidos em tempo hábil. Devido aos limites do *hardware*, dispositivos computacionais podem não possuir os requisitos necessários para tratar sozinhos uma carga tão grande de trabalho (COULOURIS et al., 2011) e quando utilizado um *hardware* especializado o custo se torna elevado. Uma solução para este problema é distribuir o processamento em múltiplos dispositivos, o custo desta estratégia é a comunicação entre os processos, pois para obterem um fim específico devem trocar mensagens para coordenarem a execução das tarefas. Além disso, deve ser considerado diversos aspectos do sistema em que será implementado uma solução deste porte.

Em um ambiente que possui múltiplos dispositivos que cooperam para realizar uma tarefa específica é dito que temos um sistema distribuído. Implementar um sistema distribuído não é uma tarefa trivial, pois requer que os requisitos do sistema sejam elencados e avaliados extensivamente. Por exemplo, os requisitos de disponibilidade (CACHIN; GUERRAOUI; RODRIGUES, 2011) e escalabilidade (CHAIPET; PUTTHIVIDHYA, 2019) devem estar em conformidade com os objetivos da aplicação. Além disso, as aplicações distribuídas devem tratar diferentes tipos de falhas para que o sistema tenha a disponibilidade especificada nos requisitos.

O crescimento da Internet fez com que a infraestrutura de rede evoluísse, tanto em número de equipamentos quanto em novas tecnologias. Isso fez com que o gerenciamento de redes se tornasse cada vez mais complexo, visto que configurar manualmente os dispositivos é uma tarefa árdua e requer conhecimentos específicos de cada fabricante, principalmente em redes com dispositivos heterogêneos. Com isso, surgiram soluções que permitem o gerenciamento de forma facilitada destes dispositivos (TENNENHOUSE et al., 1997; LAZAR; LIM; MARCONCINI, 1996; MCKEOWN, 2008). No entanto, nos últimos anos o paradigma de Redes Definidas por Software (SDN) está cada vez mais sendo utilizada em grandes redes (KREUTZ et al., 2014).

O paradigma de Redes Definidas por Software, ou *Software-Defined Networking* (SDN) tem por objetivo aumentar a flexibilidade da rede ao separar o encaminhamento e o gerenciamento de pacotes realizado pelo plano de dados e o plano de controle, respectivamente (KREUTZ et al., 2014). Ao distribuir o plano de controle temos um sistema distribuídos em que cada controlador deve trocar informações para que o gerenciamento da rede seja logicamente centralizado. Devido aos diversos algoritmos distribuídos responsáveis pela comunicação entre os controladores, uma simples falha em qualquer componente da

rede pode afetar seriamente tanto a capacidade de gerenciamento da rede quanto o correto encaminhamento de pacotes (VIZARRETA et al., 2020).

Manter o gerenciamento logicamente centralizado da rede exige que sejam empregadas primitivas de sincronização para que a consistência entre os controladores seja mantida. Estas primitivas podem ser implementadas utilizando modelos de consistência fraca ou forte. Ao utilizar o modelo de consistência forte é garantido que operações de leitura em qualquer controlador irão resultar sempre no mesmo valor. Enquanto que, na consistência fraca os controladores podem retornar valores diferentes durante operações de escrita. O modelo de consistência fraca implica em períodos que os controladores terão uma visão inconsistente do estado da rede, podendo causar instabilidades temporárias como *loops* de encaminhamento ou buracos negros (BOTELHO et al., 2016).

Os principais controladores distribuídos destacados na literatura são o ONOS (ONF, 2020) e o ODL (LF, 2018). Ambos são controladores populares, de código-aberto, possuem uma comunidade de desenvolvimento ativa e são de propósito geral (SOARES et al., 2020), apesar de suas diferenças arquiteturais (DARIANIAN; WILLIAMSON; HAQUE, 2017; BADOTRA; PANDA, 2019). Além disso, diversos trabalhos os avaliam em relação ao tráfego gerado para sincronização (MUQADDAS et al., 2017), desempenho (ZHU et al., 2019; BADOTRA; PANDA, 2019), tempo para descoberta da topologia (T.BAH et al., 2019), falhas em enlaces (VILCHEZ; SARMIENTO, 2018), localização na infraestrutura (ZHANG et al., 2017), escalabilidade (CHAIPET; PUTTHIVIDHYA, 2019) e características de sincronização (SUH et al., 2017).

Tendo em vista os conceitos expostos, esta dissertação tem como objetivo avaliar extensivamente o plano de controle SDN distribuído e algoritmos para manter a consistência que diminuam o impacto no desempenho. As contribuições presentes neste trabalho foram divididas em duas partes: *(i)* investigação dos mecanismos de tolerância a falhas e consistência de rede nos controladores distribuídos ONOS e ODL; *(ii)* implementação de um algoritmo de difusão atômica hierárquico sem-líder (AnyABCast) já que tanto ONOS quanto ODL utilizam uma abordagem de consenso onde o líder é responsável por coordenar o consenso.

Na primeira contribuição a tolerância a falhas é avaliada tanto no plano de dados quanto no plano de controle. Com relação à consistência, é investigado o estado da tabela de encaminhamento dos *switches* após falhas nos controladores. Em relação ao plano de dados, além de falhas nos enlaces, considerou-se o comportamento dos controladores com a parada e o retorno de *switches*. Os controladores foram também avaliados com falhas nos enlaces utilizando tráfego TCP e UDP. Além disso, foi considerado como os controladores operam em relação a alterações na topologia (reativo e proativo) e o impacto na disponibilidade do plano de dados perante falhas. Os resultados mostram o tempo necessário para os controladores reagirem a falhas nos enlaces e o comportamento dos

controladores com a presença de enlaces redundantes na topologia. Foi observado que o controlador ODL falhou em tratar corretamente falhas de enlaces da topologia, não utilizando enlaces redundantes quando um enlace principal falhou, já o ONOS tratou em tempo hábil as falhas de enlaces, e mesmo de *switches* (RUCHEL; TURCHETTI; CAMARGO, 2020).

Em relação ao plano de controle, executou-se uma avaliação da tolerância a falhas dos controladores no modo distribuído. Foram avaliados o custo de manter o correto funcionamento do plano de controle distribuído em relação à latência e vazão durante a instalação de regras em duas perspectivas. A primeira perspectiva avalia a latência e vazão considerando a comunicação entre *switches* e controladores (*southbound*). A segunda perspectiva considera as aplicações que realizam a instalações de regras de rede diretamente através dos controladores (*northbound*). Também foi realizada uma avaliação do tempo para os controladores sincronizarem regras entre os controladores. Em relação à tolerância a falhas, foi avaliado o tempo necessário para um novo controlador assumir após a falha do controlador principal e o estado da tabela de regras dos dispositivos após o novo controlador assumir. Os resultados obtidos permitiram concluir que o ONOS apresenta maior desempenho no modo distribuído em relação ao ODL. No entanto, o ODL se mostrou mais estável conforme o número de controladores aumentou. Além disso, o modelo de consistência adotado pelo ODL permite que mesmo na presença de falhas de controladores, regras instaladas nos dispositivos não sejam perdidas.

A segunda atividade desta pesquisa, considerou que os controladores avaliados em modo distribuído precisam manter uma visão centralizada da rede. Para tanto, cada controlador ordena globalmente os eventos recebidos. Através do algoritmo *Raft*, os controladores entram em acordo na ordem dos eventos recebidos por cada controlador. No entanto, o acordo é implementado através de um processo que possui a função de coordenar, chamado de líder, as instâncias de consenso. Desta forma, foi implementado e avaliado um algoritmo de difusão atômica, chamado *AnyABCast*, para manter a consistência forte entre processos de forma que os eventos sejam ordenados globalmente sem utilizar um processo líder. Um algoritmo de difusão atômica garante que uma mensagem seja entregue a todos os processos e na mesma ordem (DéFAGO; SCHIPER; URBáN, 2004). O algoritmo faz uso da topologia virtual de um hipercubo para comunicação entre os processos, permitindo que as mensagens sejam enviados concorrentemente através de árvores dinamicamente construídas. Inicialmente, é apresentado o pseudocódigo do algoritmo proposto e realizado experimentos em diferentes situações de falhas e comparado com uma abordagem em que cada processo se comunica diretamente com os demais processos (chamada *All2All*), mostrando que utilizar a topologia virtual permite obter ganhos em relação a latência, apesar do aumento do tempo para detecção de falhas em comparação com a topologia comparada (RUCHEL et al., 2021).

A fim de obter uma linha de base de desempenho com outras soluções, o algoritmo *AnyABCast* também foi comparado com o algoritmo de consenso *Raft*. Para tal, foram utilizados duas implementações distintas do algoritmo. Na primeira implementação foi considerado um cenário simulado, comparando os dois algoritmos em relação à latência para entrega de mensagens em ambientes com e sem falhas, na qual o *Raft* foi implementado no simulador *Neko* para fins de comparação. Foi possível observar nos experimentos realizados que a latência de entrega de mensagens do *AnyABCast* foi consideravelmente menor em relação ao *Raft* em todos os conjuntos de testes, mesmo na presença de falhas, visto que no *Raft* um novo líder deve ser eleito na falha do líder. Na segunda implementação, o algoritmo *AnyABCast* foi codificado no *framework akka.io* (LIGHTBEND, 2020) que utiliza a linguagem JAVA, e comparado com o *Apache Ratis* (Apache Ratis™, 2021), uma implementação do *RAFT* em JAVA. Os experimentos foram conduzidos de forma a obter a vazão para entrega de requisições ordenadas pelos algoritmos e mostraram que o *AnyABCast* supera o desempenho do *Raft*. No entanto, conforme o número de clientes utilizados para envio de requisições passou de 256 a taxa de entrega de mensagens reduziu no *AnyABCast* em relação ao *Raft*. Isso se deve ao número de mensagens que a abordagem sem líder impõe, que em um ambiente simulado não é capaz de representar fielmente e à própria implementação do *Raft* com inúmeras otimizações. No cenário com falhas, foi observado que o *Raft* se sobressaiu em relação à implementação do *AnyABCast* devido a própria implementação, não mostrando impactos significativos.

O restante deste trabalho é organizado da seguinte forma. No capítulo 2 são abordados os conceitos básicos e protocolos em sistemas distribuídos, além disso é apresentado o paradigma SDN e seus componentes, bem como uma discussão sobre como falhas afetam o plano de controle. No capítulo 3 é apresentado os controladores distribuídos ONOS e ODL e contextualizado os algoritmos utilizados para manter a consistência em cada um deles. Também são explorados experimentalmente estes controladores em relação ao plano de dados e plano de controle. O capítulo 4 apresenta o algoritmo *AnyABCast* e os resultados obtidos com as duas implementações realizadas, além disso é introduzido o VCube e os trabalhos que o utilizam. O capítulo 5 apresenta uma conclusão dos trabalhos realizados.

2 Conceitos Básicos

Neste Capítulo serão apresentados os conceitos básicos dos temas abordados neste trabalho. Tais conceitos serão fundamentais para compreender os desafios a serem abordados ao implementar sistemas distribuídos e como podem ser aplicados em redes de computadores. Inicialmente na Seção 2.1.4 são apresentados os conceitos básicos de sistemas distribuídos. As Redes Definidas por Software são exemplo claro de sistema distribuído, desta forma, na Seção 2.2 é apresentada a arquitetura SDN e como conceitos utilizados em sistemas distribuídos são implementados intrinsecamente nesta arquitetura.

2.1 Sistemas Distribuídos

Em ambientes onde as aplicações necessitem trocar informações para coordenar suas tarefas é dito que temos um sistema distribuído. Um sistema distribuído é frequentemente estruturado em termos de **clientes** e **serviços**. Cada serviço é implementado em um ou mais dispositivo que expõem operações invocadas através de requisições. A forma mais simples de implementar um serviço é utilizar um único dispositivo, centralizado. No entanto, a tolerância a falhas que este serviço irá suportar depende exclusivamente da confiabilidade do *hardware* e canais de comunicação que este dispositivo possui. Em sistemas que exigem alta disponibilidade este nível de tolerância a falhas é inaceitável, uma solução é distribuir os serviços em múltiplos dispositivos, garantindo assim a confiabilidade dos serviços implementados (SCHNEIDER, 1990).

Em um sistema distribuído considera-se que as aplicações executam de forma concorrente, pois cada dispositivo possui seus próprios recursos computacionais. Devido a possibilidade dos dispositivos estarem geograficamente distantes, a noção de tempo de relógio entre eles não é confiável, visto que cada dispositivo pode ter pequenas variações na marcação de tempo (LAMPORT, 2019). É importante observar que mesmo sincronizando os relógios físicos de cada dispositivo, existe um limite de precisão que pode ser obtido.

2.1.1 Modelo de Sistema

Ao projetar aplicações distribuídas é fundamental que sejam definidas as características do ambiente em que as aplicações serão desenvolvidas. Uma forma de representar estas características é através de abstrações descritas no modelo de sistema. O modelo de sistema busca identificar e modelar como as propriedades e possíveis falhas observadas podem ser representados em modelos descritivos (COULOURIS et al., 2011). Ao descrever o modelo de sistema são considerados aspectos essenciais para entender o comportamento

do ambiente em que aplicações são implementadas.

Um aspecto fundamental descrito é a interação entre os processos, em que se destaca aspectos de sincronia e a noção de tempo que cada um dos processos possui (COULOURIS et al., 2011). Em (HADZILACOS; TOUEG, 1994) são definidos três modelos de sincronia entre processos:

- Síncrono: este modelo define um limite de tempo que um processo deve executar uma tarefa e cada processo possui um limite na variação de seu relógio local em relação ao tempo real. Além disso, em relação aos canais de comunicação cada mensagem possui um limite de tempo máximo e mínimo para recebimento e entrega de mensagens em cada um dos processos. Se uma mensagem não é entregue no limite de tempo esperado, é garantido que o processo esteja falho. Implementar o modelo síncrono em dispositivos é custoso, pois as garantias temporais devem ser sempre respeitadas;

- Assíncrono: por outro lado, no modelo assíncrono não é possível definir premissas de tempo sobre o tempo de processamento das mensagens e dos canais de comunicação ou do atraso dos relógios entre processos. Desta forma, implementar algoritmos no modelo assíncrono torna-se um desafio, pois as aplicações devem levar em consideração os aspectos temporais e o indeterminismo que o modelo assíncrono prevê.

- Parcialmente Síncrono: os dois modelos anteriores consideram lados extremos em relação a noção de tempo observada. No modelo parcialmente síncrono, a comunicação entre os processos pode variar entre momentos de sincronia e assincronia. Em (DWORK; LYNCH; STOCKMEYER, 1988) são definidos duas situações em que a comunicação entre processos será parcialmente síncrona: Na primeira situação é considerado que existe um limite máximo Δ para a entrega de mensagens, mas este limite não é conhecido inicialmente. Isto faz com que problemas que seriam insolúveis no sistema assíncrono (FISCHER; LYNCH; PATERSON, 1985) possam ser resolvidos neste modelo. No entanto, protocolos que utilizam o modelo parcialmente síncrono devem conhecer o valor de Δ, para que possam saber o quanto devem aguardar para cada etapa da troca de mensagens. Em consequência disso é possível escolher um valor arbitrário de Δ e se o tempo de envio da mensagem exceder Δ o processo de destino é considerado falho. No entanto, escolher um valor muito baixo implica em muitos falsos-positivos, por outro lado, um valor alto para Δ aumenta o tempo para detecção de falhas. Desta forma, é fundamental que as próprias aplicações executando entrem em acordo com o valor de Δ. Na segunda situação é considerado que o tempo limite de comunicação Δ é conhecido, mas os canais de comunicação são instáveis e algumas vezes as mensagens podem atrasar. Para permitir que os algoritmos possam obter progresso é necessário adicionar definições que restrinjam o tempo de instabilidade. Ou seja, em cada execução deve existir um tempo de estabilização

global (GST) em que a comunicação entre processos respeite o limite Δ após o GST ter sido alcançado, tornando-se síncrono (DWORK; LYNCH; STOCKMEYER, 1988).

2.1.2 Modelos de Falhas

Em um sistema distribuído com múltiplos componentes é comum que algum destes componentes possam falhar. A capacidade de continuar operante mesmo na presença de falhas é chamada de tolerância a falhas (AVIŽIENIS et al., 2004). Para determinar se um processo do sistema está falho ou não é necessário observar a especificação deste componente. Um processo é dito como **correto** se o seu comportamento está de acordo com a especificação, e se o seu comportamento desviar do especificado o processo é chamado de **falho**. Na literatura as seguintes propriedades são identificadas para sistemas tolerantes a falhas (AVIŽIENIS et al., 2004):

- Disponibilidade (*Availability*): garante que o sistema esteja pronto para uso assim que solicitado. Normalmente, é dito como a probabilidade de um sistema estar correto e disponível para realizar as suas funções para os usuários;
- Confiabilidade (*Reliability*): refere-se a propriedade do sistema executar continuamente correto e sem falhas. Por exemplo, se um sistema falha por um segundo a cada dia ele não é confiável, apesar de possuir alta disponibilidade. Por outro lado, se o sistema permanecer sempre correto, mas for desligado por 1 dia a cada ano o sistema é confiável;
- Segurança (*Safety*): a capacidade de evitar que o usuário ou o ambiente de execução sejam afetados por falhas;
- Manutenção (*Maintanability*): o quão fácil pode ser reparado um sistema após uma falha.

Ao descrever as propriedades de um sistema é importante que o modelo de falhas seja descrito. Pois, diferentes modelos de falhas possuem soluções completamente distintas. A função do modelo de falhas é restringir como os componentes do sistema podem falhar. A seguir é apresentado os principais modelos de falhas destacados na literatura (KSHEMKALYANI; SINGHAL, 2011).

No modelo ***fail-stop***, um processo ao falhar, para completamente de executar e não volta a ficar correto. No entanto, neste modelo é provido às aplicações meios de detectar que o processo está falho. Já no modelo de falhas *crash* os processos ao falharem nunca retornam, mas as demais aplicações executando no sistema não são capazes de detectar que o processo está falho. Isto ocorre porque nenhuma mensagem adicional é

enviada pelos processos falhos. Outro modelo de falhas é por **omissão**, nas quais algumas mensagens deixam de ser enviadas/recebidas ou ambas em determinados momentos. E englobando todos os modelos anteriores existe o modelo de falha **bizantina**. Neste modelo de falhas, um processo pode apresentar comportamento arbitrário ou malicioso.

2.1.3 Detectores de Falhas

A noção de tempo em sistemas distribuídos é importante para que aplicações sejam detectadas como corretas ou não. A detecção de processos falhos é importante para que algoritmos de difusão atômica e consenso possam ser resolvidos. No entanto, o modelo assíncrono de comunicação entre os processos não provê noção alguma de tempo, o que torna impossível que algoritmos de difusão atômica ou consenso sejam resolvidos (FISCHER; LYNCH; PATERSON, 1985). Isto ocorre porque no modelo assíncrono as aplicações não são capazes de distinguir se um processo está falho ou lento.

Com base no problema de resolver o consenso em sistemas assíncronos foi proposto por Chandra e Toueg (CHANDRA; TOUEG, 1996) a necessidade de utilizar mecanismos adicionais, surgindo a noção de detectores de falhas não confiáveis. Os detectores de falhas são módulos externos as aplicações e que podem cometer erros. Devido a possibilidade de os detectores de falhas cometerem erros, eles são chamados de não confiáveis. Além disso, foi provado que utilizar detectores de falhas não-confiáveis, que cometem erros até certo grau, permite que algoritmos de consenso ou difusão atômica sejam solucionados ao utilizar no mínimo o detector $\Diamond\mathcal{P}$ (CHANDRA; HADZILACOS; TOUEG, 1992).

Os detectores de falhas são classificados de acordo com propriedades de *precisão* (*accuracy*) e *completude* (*completeness*) que são garantidas por cada detector. A precisão garante que processos corretos não serão suspeitos de terem falho. Enquanto que, a completude define que todos os processos falhos serão suspeitos. De acordo com a classificação de Chandra e Toueg (CHANDRA; TOUEG, 1996), são definidas duas propriedades de **completude** e quatro propriedades de **precisão** que um detector de falhas pode garantir:

- **Strong Completeness:** em algum momento, *Todo* processo falho é permanentemente suspeito por *todos* os processos corretos em algum momento;
- **Weak Completeness:** em algum momento, *todo* processo falho é permanentemente suspeito por *algum* processo correto;
- **Strong Precision:** *nenhum* processo é suspeito antes que ele realmente falhe.
- **Weak Precision:** *Algum* processo correto nunca é suspeito de ter falhado;
- **Eventually Strong Precision:** em algum momento futuro, *todos* os processos corretos não serão suspeitos de estarem falhos por outros processos corretos;

- **Eventually Weak Precision:** em algum momento futuro, *algum* processo correto não é suspeito de ter falhado por outros processos corretos;

Ao considerar as propriedades de precisão e completude os detectores de falhas podem ser classificados em classes. Um detector de falhas é considerado da classe *perfeito*, representado por $\mathcal{P}$, se possui as propriedades de precisão e completude forte. Similarmente, outras classes são representadas pela combinação das propriedades de precisão e completude, formando 8 classes, representadas no Quadro 1.

quadro 1 Classes de detectores de falhas.

Completude	Precisão			
	Strong	Weak	Eventually Strong	Eventually Weak
Strong	$\mathcal{P}$	$\mathcal{S}$	$\Diamond\mathcal{P}$	$\Diamond\mathcal{S}$
Weak	$\mathcal{Q}$	$\mathcal{W}$	$\Diamond\mathcal{Q}$	$\Diamond\mathcal{W}$

Adaptado de: (CHANDRA; TOUEG, 1996)

Os detectores de falhas são implementados para proverem para as aplicações mecanismos de detectarem falhas nos demais processos. Os detectores podem oferecer valores binários, ou seja, somente informar se os demais processos estão suspeitos/falhos ou corretos, e também podem oferecer valores em uma escala contínua que informa o grau de confiabilidade da suspeita ou falho de um processo (HAYASHIBARA et al., 2004). Os detectores que oferecem uma escala contínua são chamados de *accrual* (HAYASHIBARA et al., 2004) e se baseiam no histórico de comunicação entre os processos para definir o grau de confiabilidade de cada suspeita às aplicações.

2.1.4 Consistência entre Réplicas

Em sistemas distribuídos outro conceito importante é a consistência, ela está ligada à visão que cada processo ou réplica possui de uma informação distribuída, independente se um cliente obteve, ou não, uma mensagem atualizada. Réplicas são cópias de um determinado serviço que permitem manter e aumentar a disponibilidade mesmo na presença de falhas em um serviço. Obter uma mensagem atualizada de uma réplica exige que as aplicações garantam determinados níveis de consistência, mas conforme definido pelo teorema CAP (GILBERT; LYNCH, 2002), existe uma relação entre consistência e alta disponibilidade. O teorema CAP aplicado à redes (PANDA et al., 2013) afirma que é impossível que as propriedades de consistência (C), disponibilidade (A) e tolerância a particionamento (P) sejam alcançadas sem que ao menos uma destas propriedades seja sacrificada. Desta forma, para garantir alta disponibilidade na presença de particionamentos na rede (AP) é necessário relaxar a propriedade de consistência. Já, ao relaxar a consistência, aplicações podem apresentar um comportamento incorreto. Em cenários que

é necessário garantir que políticas de rede sejam aplicadas atomicamente é fundamental utilizar consistência forte (CP).

O modelo de **consistência forte** garante que uma requisição feita a qualquer processo ou réplica retornará o mesmo valor, ou seja, a versão mais atualizada de informação (VIOTTI; VUKOLIć, 2016). Em aplicações que exigem níveis menores de consistência e que permitem que em alguns momentos os dados obtidos de alguns processos ou réplicas não sejam os mais recentes, surgiram modelos que relaxam a propriedade de consistência. É destacado o modelo de **consistência eventual**. Neste modelo, os processos convergem para o mesmo estado, ou seja, na ausência de atualizações para uma determinada informação, todos os processos terão a versão mais recente e seguindo uma ordem global (VIOTTI; VUKOLIć, 2016).

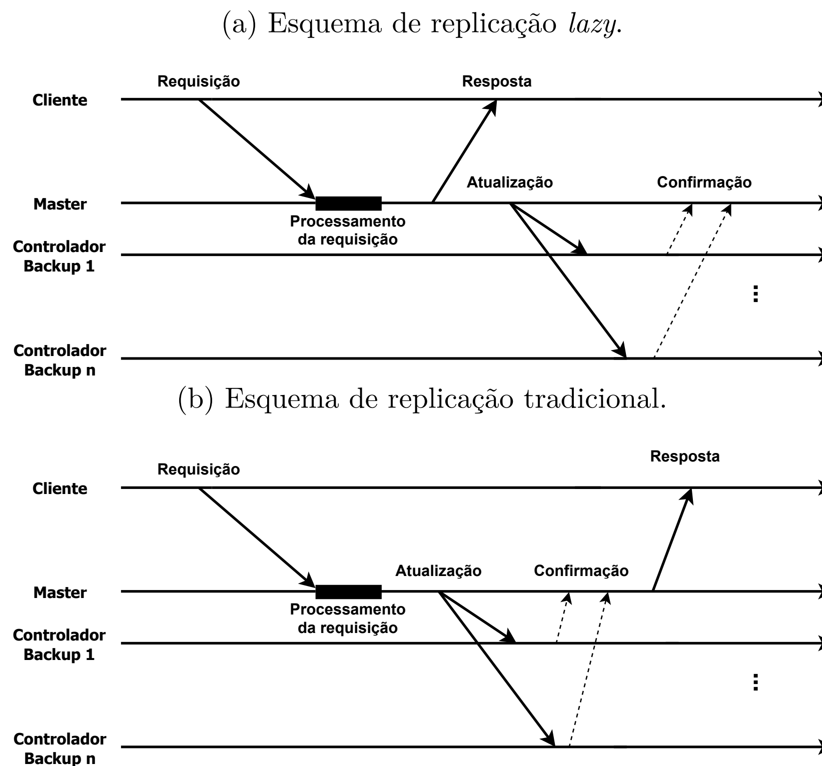
A seguir serão apresentados algoritmos comumente implementados em serviços distribuídos que utilizam os modelos de consistência elencados acima. Inicialmente, é destacado o algoritmo *primary-backup* que dependendo da variante implementada pode garantir a consistência forte ou consistência fraca. Em seguida, apresentamos o algoritmo de consenso *Raft*, dado a sua adoção em diversas ferramentas empresariais e por fim é apresentada a difusão atômica.

O algoritmo chamado de *primary-backup* emprega processos que possuem dois papéis: *primary* e *backup*. O processo *primary*, ou também chamado de *master*, possui a função de receber todas as operações de escrita e replicar aos processos *backups* as operações aceitas. Enquanto que, os processos com o papel de *backup* recebem as atualizações aceitas e podem se tornar o processo principal, caso o *primary* falhe. Este modelo de algoritmo permite aumentar a disponibilidade do sistema visto que múltiplos processos possuem cópias dos dados (BUDHIRAJA, 1993). A ordem que as mensagens são entregues em cada *backup* é definida pelo processo *primary*, pois ele coordena a replicação entre os processos.

Este algoritmo possui duas implementações com características distintas: *lazy* e a abordagem tradicional. O modo de replicação ***lazy*** garante que operações de escrita tenham maior desempenho, visto que são realizadas localmente e propagadas posteriormente aos demais processos. Desta forma, operações de leitura podem resultar em valores distintos entre os processos, pois uma operação de escrita pode estar sendo propagada enquanto ocorre a leitura de um valor. Na Figura 1a é representado o esquema de replicação *lazy*. Neste modo, logo que um processo recebe um evento uma resposta é enviada ao cliente. Na falha do *primary* operações de escrita podem ser perdidas antes de chegarem nos *backups*, mesmo o cliente obtendo a resposta. Outro modo de implementação do *primary-backup* é a abordagem tradicional (BUDHIRAJA, 1993), representado na Figura 1b. Neste modo o cliente somente recebe a resposta após todos os *backups* receberem a atualização. Esta característica possui um impacto no desempenho das aplicações, visto que possui um custo adicional, a latência para que o dado seja replicado entre os *backups*.

As duas implementações do algoritmo entregam modelos de consistência distintos. Enquanto que a replicação tradicional emprega a consistência forte. Ao utilizar a implementação *lazy* implica em momentos que os processos estarão com visões distintas sobre uma determinada informação, mas irão convergir para o mesmo estado na ausência de atualizações.

Figura 1 – Replicação passiva de eventos (*primary-backup*).



(a) Esquema de replicação *lazy*.

(b) Esquema de replicação tradicional.

Fonte: O Autor.

2.1.4.1 O Consenso

Outra forma de obter a replicação dos dados é através de algoritmos de consenso. O consenso é um problema fundamental em sistemas distribuídos e permite que múltiplos processos entrem em acordo, mesmo na presença de falhas. O consenso é comumente implementado no contexto de máquinas de estado replicadas, as máquinas de estado são uma coleção de processos que computam mensagens na mesma ordem, o que os permite alcançar o mesmo estado, mesmo na presença de falhas (ONGARO; OUSTERHOUT, 2014). Estes algoritmos permitem que um conjunto de processos decidam em um mesmo valor e que esta decisão seja unânime. Algoritmos de consenso garantem as seguintes propriedades: (CACHIN; GUERRAOUI; RODRIGUES, 2011):

- Terminação: em determinado momento os processos corretos irão definir um valor;
- Validade: nenhum valor é decidido se ele não houver sido proposto;

- Integridade: os valores decididos não são duplicados;
- Acordo: todos os processos definem no mesmo valor;

O consenso possui um custo associado à definição de um líder, responsável por manter o acordo entre os processos ativos. Este líder é sobrecarregado com a função de coordenar os demais processos. Com isso em condições de sobrecarga, propriedades do consenso podem ser violadas, levando à momentos de indisponibilidade (HANMER et al., 2018). Um importante algoritmo utilizado para a implementação do consenso é o Paxos (LAMPORT et al., 2001), no entanto devido a detalhes importantes ausentes na descrição original do algoritmo tornou sua implementação uma tarefa árdua. Com o avanço de pesquisas em sistemas distribuídos novos algoritmos mais fáceis de compreender e implementar surgiram. Entre eles o algoritmo que se destacou foi o *Raft* (ONGARO; OUSTERHOUT, 2014), tornando-se o principal algoritmo utilizado em aplicações empresariais para alcançar o consenso entre processos.

2.1.4.1.1 Algoritmo *Raft*

O *Raft* é um algoritmo de consenso baseado em líder. Durante a operação normal do algoritmo as atualizações aplicadas no *log* replicado partem do processo líder (*leader*) e os demais processos são *followers*. O papel dos *followers* é inserir as atualizações recebidas do líder em seu registro (*log*) após uma série de validações. A função do líder é executar todas as operações de escrita solicitadas pelos clientes. O funcionamento normal do algoritmo é relativamente simples, no entanto, quando o líder falha entram em cena requisitos importantes para garantir a consistência dos registros replicados e o funcionamento do algoritmo. Para entender o *Raft* é necessário relembrar que cada processo participante do consenso possui um registro replicado (*log*) e este registro é utilizado para construir uma máquina de estados, através da execução sequencial de cada entrada do registro. Isto faz com que todos os processos alcancem o mesmo estado em suas máquinas de estados, assim que o líder houver replicado todas as entradas que possui.

Na nomenclatura do *Raft* as chamadas de procedimento remoto (RPC) invocadas pelo líder para replicar entradas de seu registro são chamadas de ***AppendEntry***. Ao receber esta RPC o *follower* responde ao líder de que a entrada foi inserida com sucesso em seu registro. A resposta do *follower* pode ser de falha, no entanto ocorre somente quando o líder é suspeito de ter falhado. O líder ao receber a confirmação de sucesso da maioria dos processos envolvidos no consenso ($N/2 + 1$). Envia outra RPC *AppendEntry* confirmando a entrada no registro do *follower*.

A RPC *AppendEntry* pode conter registros para replicação, mas além disso, contém informações sobre o estado do registro replicado presente no líder. Outra função desta RPC é informar aos *followers* de que o líder não está falho, o chamado *heartbeat*. O líder

envia regularmente a RPC *AppendEntry* para informar aos *followers* de que está ativo. Cada *follower* possui um cronômetro com um valor escolhido aleatoriamente, sempre que uma RPC válida (com resposta de sucesso) é enviada ao líder esse cronometro é reiniciado. Se o líder deixa de enviar essas RPCs no tempo esperado, ou elas atrasam, cada *follower*, inicia uma votação para se tornar o líder e passa a ter o papel de *candidate*.

A eleição de um novo líder no *Raft* ocorre através da RPC *RequestVote*. Logo ao suspeitar de que o líder está falho o processo *follower* incrementa o **termo** atual e adiciona na RPC *RequestVote* o seu termo. O termo é um contador inteiro monotônico utilizado para limitar a autoridade de um líder e prevenir a presença de múltiplos líderes. Além disso, o termo é utilizado para prevenir que um antigo líder, que se recuperou de uma falha, possa enviar atualizações, sendo que um novo líder já foi eleito. O líder com o maior termo sempre prevalecerá se houverem conflitos em processos.

2.1.4.2 Difusão Atômica

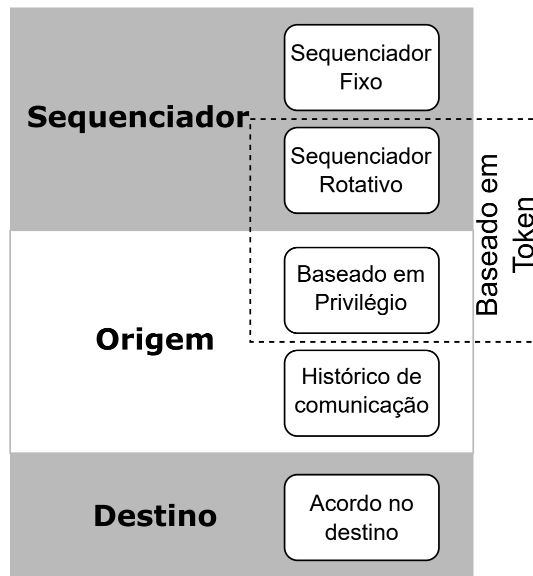
Diferente do consenso, outra forma de manter a consistência forte é utilizando primitivas de difusão de grupo que implementem o conceito de ordem total, também chamado de difusão atômica (CACHIN; GUERRAOUI; RODRIGUES, 2011). Esta primitiva garante que mensagens sejam entregues a todos os processos ativos, se ao menos um processo entregou a mensagem e está correto, além disso todas as mensagens são ordenadas. A difusão atômica garante as seguintes propriedades (DéFAGO; SCHIPER; URBáN, 2004):

- **Validade**: se um processo envia uma mensagem *m*, então em algum momento ele entrega *m*;
- **Acordo uniforme**: se um processo entrega uma mensagem *m*, então em algum momento todos os processos corretos entregarão *m*;
- **Integridade uniforme**: para qualquer mensagem *m*, cada processo entrega *m* no máximo uma vez somente se *m* foi entregue por algum processo, ou seja, mensagens não são criadas;
- **Ordem Total**: se os processos *p* e *q* entregam as mensagens *m* e *m'*, então *p* entrega *m* antes de *m'* se e somente se *q* entrega *m* antes de *m'*;

Em (DéFAGO; SCHIPER; URBáN, 2004) os autores classificaram os algoritmos de ordem total com base no papel do processo que realiza a ordenação das mensagens. Desta forma, foram definidos os papéis de remetente, destino e sequenciador. O remetente (*sender*) é o processo p_s que origina a mensagem *m*. O destino (*destination*) é o processo p_d em que a mensagem é destinada. O sequenciador não é necessariamente o destino ou

origem da mensagem, mas um processo envolvido na ordenação de mensagens. Na Figura 2 são apresentadas as classificações dos algoritmos de difusão atômica com base no papel de cada processo.

Figura 2 – Classificação dos algoritmos de ordem total (Atomic Broadcast).



Adaptado de: (DéFAGO; SCHIPER; URBáN, 2004)

Ao utilizar um sequenciador fixo um processo é eleito como responsável pela ordenação das mensagens. O sequenciador é único e a responsabilidade não é transferida para outros processos na ausência de falhas. O sequenciador rotativo permite que múltiplos processos sejam utilizados, balanceando a carga entre diferentes processos. O sequenciador baseado em privilégio utiliza *tokens* atribuídos a cada processo com permissão de envio, assim como no sequenciador rotativo. O sequenciador que utiliza o conceito de histórico de comunicação é similar ao baseado em privilégio, dado que a ordenação é baseada na origem, os processos podem enviar mensagens a qualquer momento. No entanto, a entrega de mensagens é atrasada para não violar a ordem total, aguardando a entrega em ordem das mensagens. Cada mensagem carrega um sinal de tempo (físico ou lógico) utilizado pelos processos para aprenderem quando devem entregar as mensagens. O acordo no destino, como o nome indica os processos de destino devem entrar em acordo na ordem de entrega das mensagens. Os autores em (DéFAGO; SCHIPER; URBáN, 2004) classificaram o acordo no destino com base em três variantes: (i) acordo em um número de sequência da mensagem; (ii) acordo em um conjunto de mensagens; ou (iii) acordo na ordem de mensagens proposta.

Em relação aos sequenciadores de acordo com o destino, descritos em (DéFAGO; SCHIPER; URBáN, 2004), ao utilizar a primeira variante (i) os processos entram em acordo para o número de sequência de cada mensagem recebida. Desta forma, as mensagens são entregues, de forma ordenada, com base em uma marcação temporal global. Na segunda

variante (ii) os processos devem entrar em acordo com o conjunto de mensagens a ser entregue, ou seja, devem ser ordenadas. Os autores destacaram que com base no algoritmo de (CHANDRA; TOUEG, 1996) a difusão atômica pode ser transformada em uma sequência de instâncias de consenso para entregar, de forma ordenada, um conjunto de mensagens. Na terceira variante (iii) um processo de destino pré-estabelece uma ordem e os demais processos devem entrar em acordo com a ordem das mensagens proposta.

2.2 SDN

Os conceitos apresentados acima são fundamentais para entender os desafios relacionados a implementação de sistemas distribuídos sob diversos aspectos. Observar como as redes de computadores funcionam permite entender como cada um dos conceitos apresentados se encaixam. Com isso, o advento de novas tecnologias de gerenciamento, como as Redes Definidas por Software (SDN), trouxeram flexibilidade e inovações (KREUTZ et al., 2014), mas também aprofundaram ainda mais a relação com sistemas distribuídos. Com o surgimento de SDN, uma nova arquitetura de rede foi definida, e para entender esta nova arquitetura se torna necessário observar como é realizado o gerenciamento e encaminhamento de dados nos dispositivos.

Em dispositivos de rede, há um plano de controle que é responsável por gerenciar o encaminhamento de pacotes e os algoritmos implementados nos dispositivos. Além disso, há também um plano de dados, que por sua vez realiza o encaminhamento de pacotes no meio físico. No entanto, embutir em um único equipamento o gerenciamento e o encaminhamento de dados faz com que aspectos como configuração, monitoramento e evolução da infraestrutura de rede como um todo sejam comprometidos (BENSON; AKELLA; MALTZ, 2009).

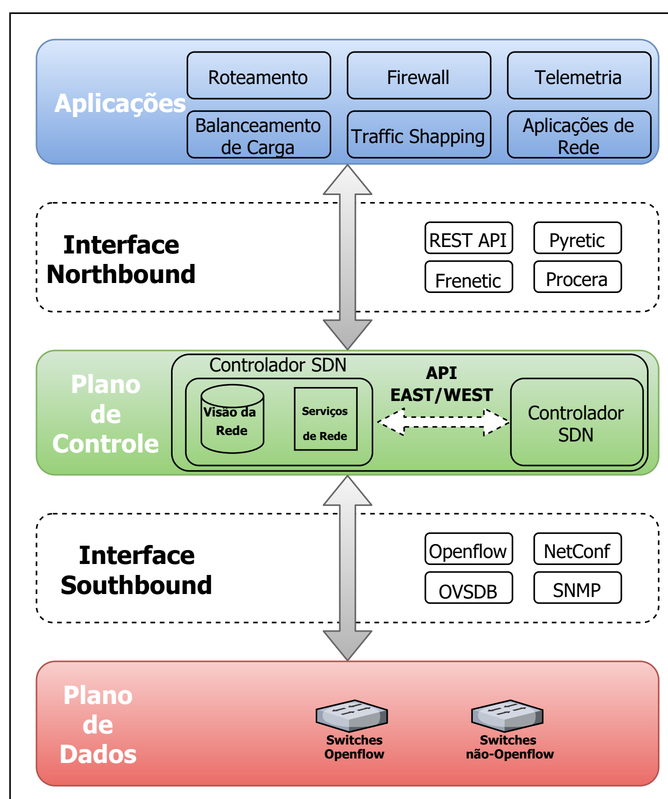
Por outro lado, o princípio básico do paradigma SDN se constituiu na separação do plano de dados e plano de controle que permite que a rede seja modular e que ambos os planos evoluam independentemente. No entanto, o plano de controle desacoplado, implica na necessidade de avaliar os requisitos de escalabilidade, confiabilidade e disponibilidade dos dispositivos SDN (BANNOUR; SOUIHI; MELLOUK, 2018b).

A definição de redes gerenciáveis por *software* não é um conceito novo e diversas soluções anteriormente haviam sido propostas (TENNENHOUSE et al., 1997; LAZAR; LIM; MARCONCINI, 1996; LAZAR, 1997; SHEINBEIN; WEBER, 1982; CAESAR et al., 2005), no entanto, foi com o surgimento do protocolo Openflow (MCKEOWN, 2008) que o termo SDN ganhou força. Com o Openflow a arquitetura SDN foi dividida em três camadas: plano de dados, plano de controle e aplicações. O plano de dados, também chamado infraestrutura, é responsável por realizar fisicamente o encaminhamento de pacotes, nele estão os *switches* e roteadores presentes na rede. O plano de controle é

responsável pela comunicação entre dispositivos no plano de dados, bem como traduzir os requisitos e políticas de rede impostas pelas aplicações para serem aplicadas nos dispositivos no plano de dados. Já as aplicações são responsáveis por instruir como os dispositivos devem realizar o encaminhamento de pacotes (KREUTZ et al., 2014).

Além da separação em camadas, na arquitetura SDN, foram propostas interfaces que facilitam o acesso à cada um dos recursos providos por cada camada, conforme visto na Figura 3. Com a implementação destas interfaces é possível a utilização de recursos sem que seja necessário conhecer a implementação de cada uma das funcionalidades presentes em cada camada. As interfaces utilizadas para comunicação entre camadas são divididas em: comunicação entre aplicações e controlador, chamada de *northbound interface* (NBI); comunicação controlador e dispositivos no plano de dados, chamada de *southbound interface* (SBI); e comunicação entre controladores, quando o plano de controle é distribuído, chamada de *East/West interface* (EWI).

Figura 3 – Camadas da Arquitetura SDN



Fonte: O Autor.

Os protocolos utilizados na NBI variam de acordo com a implementação de cada controlador, sendo mais comum a utilização de APIs RESTFUL (RICHARDSON; RUBY, 2007) e linguagens de alto nível como *frenetic* (FOSTER et al., 2011) ou pyretic (REICH et al., 2013) para definir as políticas de rede utilizadas e realizar o gerenciamento do plano de dados.

No controlador, o principal protocolo utilizado para comunicação com dispositivos no plano de dados através da SBI é o Openflow (KREUTZ et al., 2014). Além disso, outros protocolos como OVSDB (PFAFF; DAVIE, 2013), of-config (NARISETTY et al., 2013), Netconf (ENNS et al., 2011) e SNMP (FEDOR et al., 1990) podem coexistir juntamente com o protocolo Openflow, mas possuem funções distintas. A comunicação entre os controladores (EWI) não possui protocolos bem definidos, apesar de iniciativas de padronização em diferentes grupos de trabalho (BANNOUR; SOUIHI; MELLOUK, 2018b).

Na arquitetura SDN a função do plano de controle é gerenciar e monitorar os dispositivos presentes no plano de dados, porém o modo de sua implementação pode variar de acordo com os requisitos de cada rede. Separar o plano de controle dos dispositivos, permite que seja logicamente centralizado todo o gerenciamento, configuração e monitoramento dos dispositivos. Naturalmente, a forma mais simples de implementar o plano de controle é centralizar todas as funções em um único componente, seja de hardware ou software, o que chamamos de plano de controle centralizado. Contudo, centralizar todas as funções da rede em um componente traz questões relacionadas à requisitos de disponibilidade, escalabilidade e tolerância a falhas que cada rede possui (ZHANG et al., 2019; LINGER; MEAD; LIPSON, 1998). Em consequência disso, distribuir o plano de controle em múltiplos dispositivos se mostra necessário para ambientes de médio e grande porte.

Um exemplo são redes de centros de dados que utilizam milhares de dispositivos de encaminhamento que exigem do controlador o processamento de uma grande quantidade de eventos (BANNOUR; SOUIHI; MELLOUK, 2018b). Conforme a quantidade de eventos aumenta, o controlador se torna um gargalo, visto que os recursos físicos são limitados ao hardware utilizado. Na literatura não faltam esforços para obter maior desempenho e aumentar a escalabilidade de controladores (TOOTOONCHIAN et al., 2012) em modo centralizado, mas o desempenho acaba sendo limitado pelo hardware utilizado. Além dos limites de desempenho para processamento de eventos, obter baixa latência para comunicação com os dispositivos no plano de dados é fundamental para provedores de serviço, que possuem poucos dispositivos de encaminhamento. No entanto, estes dispositivos são geograficamente distribuídos, o que implica no aumento da latência de comunicação fim-a-fim, quando o controlador está afastado dos dispositivos (BANNOUR; SOUIHI; MELLOUK, 2018b).

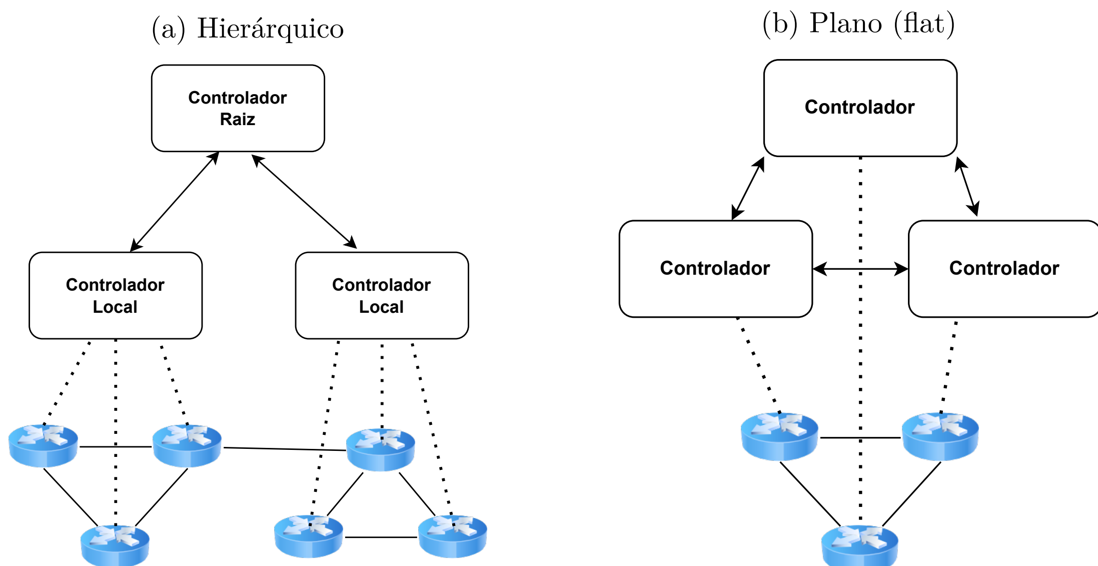
Devido à diferentes requisitos de rede necessários para cada cenário, distribuir o plano de controle fisicamente permite que múltiplos controladores sejam utilizados. Isto faz com que novos controladores possam ser ativados dinamicamente para tratar a carga de trabalho gerada por dispositivos de encaminhamento conforme são adicionados na rede. No entanto, ao utilizar múltiplos controladores surgem inúmeros desafios para manter

o plano de controle funcional. Além disso, a forma como o plano de controle deve ser implementado de forma distribuída não possui um padrão definido, e normalmente são classificadas em plano (*flat*) e hierárquico (BANNOUR; SOUIHI; MELLOUK, 2018b):

- Controle Hierárquico em SDN: nessa arquitetura assume-se que o plano de controle é verticalmente particionado em níveis de controladores. Na Figura 4a é possível observar um exemplo de organização hierárquica, em que um controlador raiz gerencia um conjunto de controladores locais. Desta forma, é possível que o controlador raiz utilize técnicas de engenharia de tráfego para otimizar a utilização dos enlaces na rede gerenciada por cada controlador local. A organização hierárquica pode aumentar a escalabilidade e desempenho dos controladores, visto que um número menor de mensagens é necessário para que haja uma visão global da rede (BANNOUR; SOUIHI; MELLOUK, 2018b).
- Controle plano (flat) em SDN, Figura 4b: neste modo os controladores estão organizados sob um único domínio administrativo e possuem as mesmas funções, é dito que são particionados horizontalmente (BANNOUR; SOUIHI; MELLOUK, 2018b). Ao ser identificada uma falha em um controlador, outros controladores podem assumir o controle dos dispositivos órfãos, aumentando a resiliência do plano de controle. O particionamento permite diminuir a latência de comunicação dos dispositivos do plano de dados com o plano de controle, visto que os controladores podem ser instalados próximo dos dispositivos.

Figura 4 – Arquitetura do Plano de Controle.

(a) Hierárquico (b) Plano (flat)



Fonte: O Autor.

A presença de múltiplos controladores implica que mecanismos adicionais sejam utilizados para manter o plano de controle logicamente centralizado. Entre as técnicas

empregadas estão (SPALLA et al., 2016): replicação ativa, replicação passiva, replicação com múltiplos *master/slave* ou a utilização de um armazenamento externo.

Na replicação ativa, também conhecida como replicação máquina de estado, os dispositivos do plano de dados se conectam a múltiplos controladores e cada um possui o papel mesmo papel. No papel *equal*, definido no protocolo Openflow, todos os controladores recebem todos os eventos gerados no plano de dados. Cada controlador processa os estes eventos recebidos e decide se deve responder ao dispositivo que originou o evento com base em um protocolo definido entre os controladores. Nesta abordagem, se todos os eventos forem recebidos e processados na mesma ordem o plano de controle terá o mesmo estado em todos os controladores, e caso um controlador falhe os demais podem assumir o controle da rede. No entanto, esta abordagem necessita da implementação de algoritmos que garantam a ordem total de eventos, visto que em uma rede a entrega não é determinística e com a utilização de múltiplos dispositivos estes algoritmos tornam-se complexos. Além de que, nesta abordagem são desperdiçados recursos como largura de banda para replicação dos eventos, processamento e memória para manter todo o estado da rede em cada controlador.

Já na replicação passiva ou *master/slave*, o custo gerado para replicação de eventos nos controladores é menor, visto que um controlador assume o papel *master* de todos os dispositivos e os demais controladores assumem o papel *slave*, e somente o controlador *master* processa e responde aos eventos recebidos do plano de dados e realiza a sincronização com os demais controladores. No entanto, esta abordagem possui a desvantagem de que o controlador *master* fica sobrecarregado com o processamento de todos os eventos da rede, apesar de diminuir o processamento nos controladores com o papel de *slave*. Além disso, se torna necessário que o estado do *master* seja monitorado em caso de falha, para que um novo *master* seja escolhido entre os demais controladores ativos.

Por outro lado, utilizando múltiplos *master/slave* é mesclado as vantagens de cada uma das abordagens anteriores. Nesta abordagem, cada controlador possui o papel *master* de um conjunto de dispositivos e os demais controladores assumem o papel *slave*. No entanto, esta abordagem necessita que os controladores troquem informações e implementem mecanismos que garantam a consistência da rede, visto que múltiplos controladores podem realizar alterações no estado da rede de forma concorrente. Uma técnica que torna mais simples a implementação desta abordagem é a utilização de um armazenamento de dados externo pois é delegado à esta entidade externa o armazenamento das informações da rede. No entanto, esta abordagem implica em um custo adicional de comunicação com o armazenamento de dados, podendo tornar inviável sua utilização. Os principais algoritmos utilizados para manter a consistência na rede utilizando está abordagem são o algoritmo de consenso *Raft* e o algoritmo de replicação *primary-backup* (Seção 2.1.4).

2.2.1 O Protocolo Openflow

O Openflow foi implementado com o objetivo de facilitar a experimentação de novos protocolos de rede nos dispositivos (MCKEOWN, 2008). O protocolo permite que equipamentos heterogêneos se comuniquem de forma uniforme sem que fabricantes exponham a detalhes arquiteturais dos dispositivos. Ele surgiu em 2008 como um protocolo de código-aberto, e foi designado à *Open Networking Foundation* (ONF) a especificação e evolução do protocolo.

Para realizar o encaminhamento de pacotes, o Openflow utiliza o conceito de regras de fluxo inseridas na tabela de fluxos. Uma entrada na tabela de fluxo é identificada por um par de campos de checagem e uma prioridade, cada uma das entradas é responsável por tratar o tráfego recebido que coincide com seus campos de checagem. O Openflow permite a utilização de máscaras coringas, permitindo a criação de regras genéricas. A entrada na tabela de fluxos que utiliza a máscara coringa em todos os campos de checagem e prioridade 0 é chamada de *table-miss*. Esta regra instrui o dispositivo à realizar alguma ação quando não houverem regras que coincidirem com o pacote.

Ao utilizar o protocolo Openflow nos *switches*, o controlador pode adicionar, atualizar ou remover regras nas tabelas de fluxo proativamente ou reativamente (OPENFLOW, 2015), de acordo com o modo de operação de cada controlador e suas aplicações. O modo reativo consiste no *switch* solicitar ao controlador como tratar um pacote sempre que não houver regra instalada posteriormente que coincide com o pacote atual. No modo proativo é papel do controlador instalar todas as regras necessárias para encaminhamento de pacotes e definir uma ação caso nenhuma regra seja aplicável. É possível aplicar um modo híbrido, em que são utilizados os modos proativo e reativo em conjunto.

O Openflow, a partir da versão 1.2, possui o suporte à múltiplos controladores, introduzindo o conceito de papéis para cada um dos controladores conectados aos dispositivos de rede. Desta forma é possível aumentar a confiabilidade do plano de controle, pois na falha de um controlador outro pode assumir o controle (OPENFLOW, 2015).

2.2.2 Falhas em Controladores

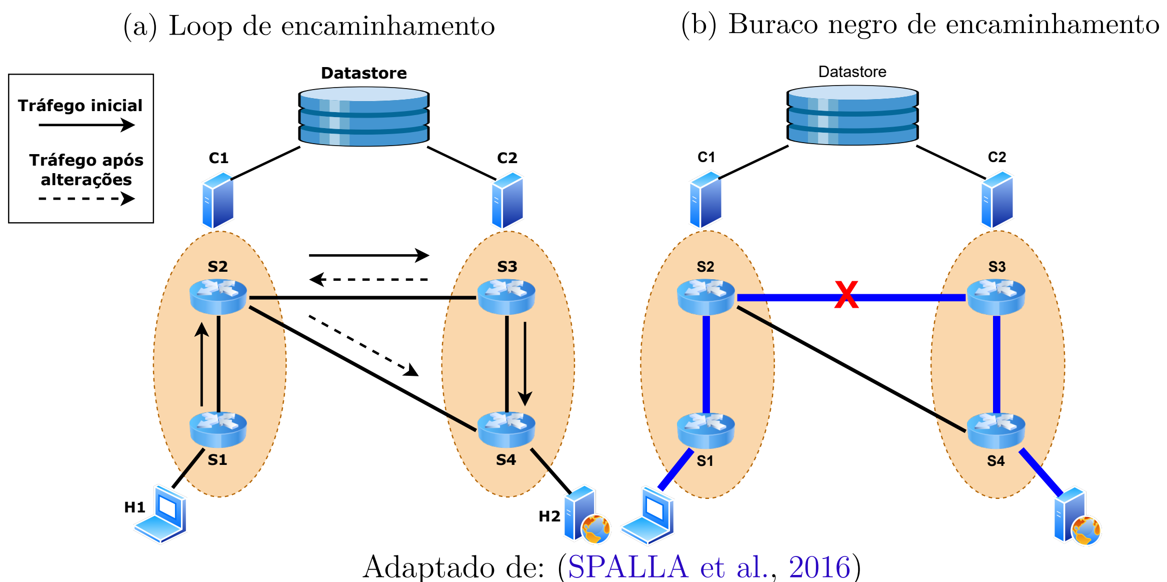
No plano de controle distribuído é necessário que os controladores troquem informações, visto que uma premissa de SDN é prover uma visão global da rede. Devido ao papel fundamental do controlador no funcionamento da rede, é importante que sejam implementados mecanismos que garantam a confiabilidade do plano de controle. Desta forma, é fundamental tolerar situações adversas de falhas, seja em enlaces do plano de dados ou até mesmo nos próprios controladores. Aliar desempenho e tolerância a falhas em SDN considerando uma visão consistente do estado da rede é desafiador.

Inconsistências entre os controladores podem gerar erros temporários que afetam

o plano de dados (BOTELHO et al., 2016), problemas comuns causados pela visão inconsistente entre os controladores são *loops* de encaminhamento e buracos negros. Na Figura 5a, temos o caso em que a visão inconsistente entre os controladores C1 e C2 gera um *loop* de encaminhamento ao realizar um balanceamento de carga para o enlace S2-S4 ao notar que o enlace S3-S4 começou a ficar congestionado. Nesta situação, o controlador C2 pode decidir encaminhar o tráfego recebido por S3 pelo enlace S2-S3. No entanto, se for implementado o modelo de consistência fraca a regra poderá ser aplicada antes que os dois controladores estejam cientes da alteração. Desta forma, ao receber pacotes o *switch* S2 realizará o encaminhamento pelo enlace S2-S3, criando assim um loop, pois o S3 é instruído a realizar o encaminhamento para S2. Somente após os controladores estarem consistentes que o *loop* é interrompido. Na Figura 5b, é apresentado um cenário com falha no enlace S2-S3, a falta de consistência entre os controladores gera um buraco negro de encaminhamento, pois pacotes são encaminhados para uma porta que não está ativa. Ao detectar a falha no enlace o controlador C2 instrui que os pacotes sejam encaminhados pelo enlace S4-S2, ao invés do caminho destacado. Se o controlador C1 não identificou a falha no enlace ao encaminhar o tráfego de S2 para S4, será utilizado o enlace S2-S3 que está falho e os pacotes serão descartados.

Figura 5 – Erros de encaminhamento causados por inconsistências entre controladores.

(a) Loop de encaminhamento (b) Buraco negro de encaminhamento



Adaptado de: (SPALLA et al., 2016)

Observando a consistência em SDN (SU; WANG; LIU, 2019) os autores realizaram um levantamento considerando três aspectos: (i) a visão da rede consistente e as decisões tomadas pelo plano de controle; (ii) a consistência e a eficiência durante atualizações no plano de dados; e (iii) a consistência da tabela de fluxos nos dispositivos do plano de dados. Devido a natureza assíncrona dos canais utilizados para controle do plano de dados os autores apresentam situações que devem ser tratadas, por exemplo, a presença de regras antigas durante a instalação de novas regras no plano de dados que pode ocasionar

violações de políticas de rede e problemas temporários na rede, além da troca de dados entre controladores.

Nota-se que de acordo com os níveis de consistência (fraca, forte ou adaptável) pode ser utilizado os conceitos de consistência *por pacote* em que para cada pacote que atravessa uma rede é processado por somente uma configuração de rede, por exemplo, em uma atualização de políticas de rede em que uma política antiga e uma nova possam estar instaladas no mesmo dispositivo. Por outro lado, é possível utilizar o conceito de consistência *por fluxo*, em que todos os pacotes pertencentes a um fluxos de dados são processados pela mesma política de rede. A falta de consistência impacta no encaminhamento de pacotes: A presença de regras antigas e regras novas podem causar problemas de encaminhamento durante um processo de atualização, por exemplo, loops de encaminhamento, buracos negros e violações de políticas de rede durante o período de atualizações. Devido ao fato de algumas atualizações necessitam de mais do que um passo para serem completadas existe um custo adicional para que estas atualizações sejam realizadas. Apesar de haver um custo para manter a consistência entre os controladores distribuídos, o desempenho da rede pode ser degradado devido à inconsistências no plano de controle (SU; WANG; LIU, 2019)

2.3 Conclusão

Neste Capítulo foram apresentados os conceitos fundamentais de sistemas distribuídos e seus principais componentes e definições. Além disso, foi mostrado um exemplo de sistema distribuído que está em constante desenvolvimento, as Redes Definidas por Software. Os desafios de comunicação entre os controladores SDN se tornam evidentes quando o plano de controle é distribuído, fazendo com que os controladores devam trocar mensagens para que possam controlar. Com isso, elencar os modelos de consistência é importante para que os controladores, quando no modo distribuído, possam decidir corretamente sobre as regras a serem instaladas nos dispositivos. Outro aspecto fundamental é a disponibilidade do plano de controle, a maneira de alcançar alta disponibilidade é através de replicação, na Seção 2.1.4 os algoritmos destacados permitiram entender o contexto e aplicabilidade para as aplicações, em especial em SDN, estes algoritmos são a base para obter o plano de controle distribuído. Desta forma, faz-se necessário avaliar as Redes Definidas por Software, seja em relação às métricas de desempenho aos seus aspectos distribuídos.

3 Tolerância a Falhas nos Controladores SDN ONOS e ODL

Atualmente os principais controladores distribuídos de código-aberto são o *Open Networking Operating System* (ONOS) e o OpenDayLight (ODL) (BADOTRA; PANDA, 2019). Estes controladores estão em desenvolvimento ativo e são utilizados como base para outras soluções comerciais. Por suas características distribuídas ONOS e ODL oferecem primitivas que permitem a troca de mensagens entre controladores de acordo com o caso de uso. São apresentados detalhes arquiteturais de cada um deles. Em seguida, são avaliados em relação a falhas de enlaces e dispositivos no plano de dados com diferentes tipos de fluxos e modos de operação dos dispositivos. Além disso, o plano de controle é avaliado em relação ao desempenho, consistência e tolerância a falhas.

3.1 ONOS

O ONOS é um controlador de código-aberto escrito em JAVA e mantido pela *Open Networking Foundation* (ONF). Este controlador foi desenvolvido com o intuito de aproveitar todas as funcionalidades SDN e atender à requisitos de provedores de serviço, permitindo que seja utilizado em modo centralizado ou em modo distribuído. Devido aos diferentes requisitos de consistência exigidos pelas aplicações, o controlador disponibiliza primitivas que implementam múltiplos níveis de consistência (forte e fraca) (Seção 2.1.4).

Para as aplicações que exigem que todos os controladores envolvidos possuam uma visão consistente da rede, é disponibilizado estruturas de dados para as aplicações que implementam o modelo de consistência forte, utilizando o *framework Atomix* (ATOMIX, 2020), que utiliza o algoritmo *Raft* (Seção 2.1.4.1.1) para a sincronização de estados entre as réplicas do *datastore* (BANNOUR; SOUIHI; MELLOUK, 2018b). Quando a consistência entre as aplicações pode ser relaxada, é disponibilizado estruturas de dados que implementam o modelo de consistência fraca, para tal é utilizado a replicação *lazy* com um método de sincronização otimista entre os controladores, e uma técnica chamada de anti-entropia com um protocolo epidêmico (*gossip*) para corrigir eventuais perdas de sincronia entre os controladores (BANNOUR; SOUIHI; MELLOUK, 2018b).

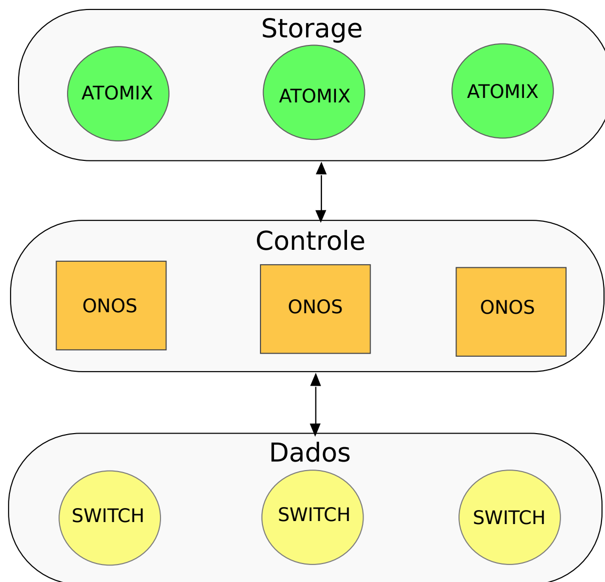
A partir da versão 1.14 do ONOS, publicada em 4 de Setembro de 2018, a arquitetura do controlador foi modificada, permitindo que o controlador e o *datastore* sejam desacoplados. Na Figura 6a é mostrado um esquema representativo da arquitetura SDN considerando cada um dos componentes da infraestrutura. Cada controlador ONOS conecta-se a n instâncias do Atomix (*datastore*). O *datastore* é localizado em processos distintos

do controlador, através de *containers*, máquinas virtuais ou dispositivos. O número de instâncias do controlador ou do *datastore* são independentes de quantos controladores estão ativos. Por exemplo, 3 controladores ONOS podem conectar-se à 5 instâncias do *datastore*. A identificação das instâncias do *Atomix* que o ONOS irá se conectar é realizada através de arquivos de configuração.

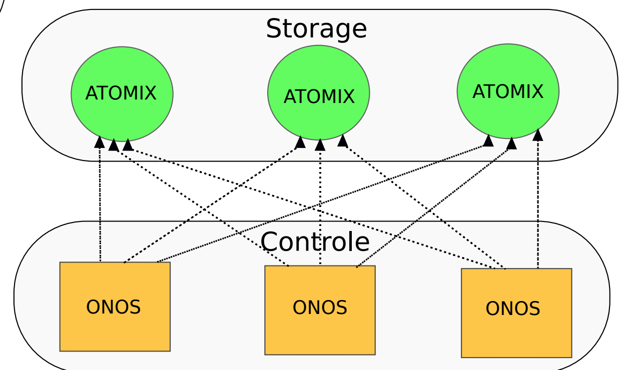
O *datastore* no ONOS provê serviços essenciais ao funcionamento do controlador. Dentre eles, o controle de membros ativos no *cluster* (controladores e instâncias do *datastore*) chamado de *membership* e a definição dos papéis de controladores para o protocolo Openflow, chamado de *mastership*. Por questões de disponibilidade o controlador se conecta à múltiplas instâncias do *datastore*, como visto na Figura 6b.

Figura 6 – Arquitetura de Comunicação do Controlador ONOS em Modo Distribuído.

(a) Representação da Arquitetura do ONOS.



(b) Conexões do ONOS com Datastore.

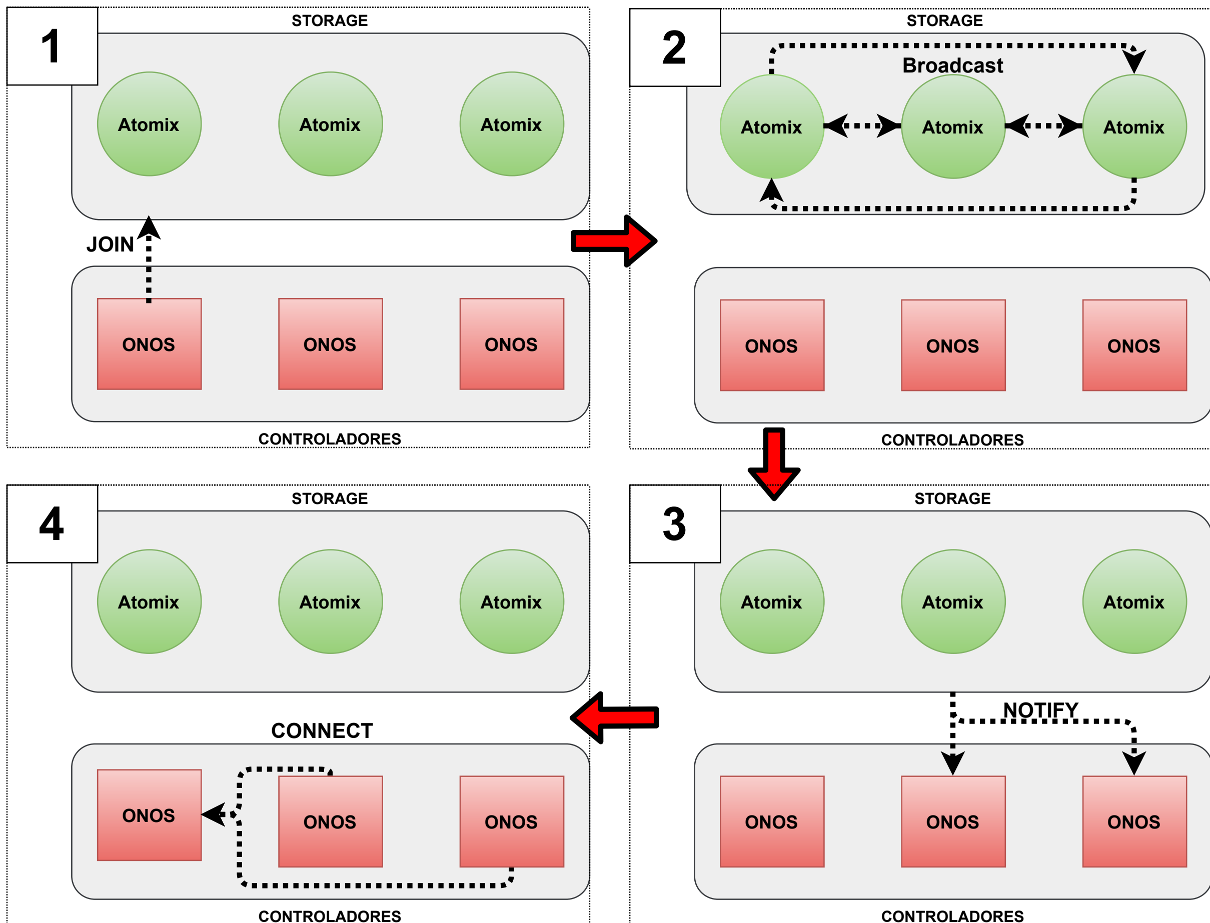


Adaptado de: (HALTERMAN, 2018).

O serviço de *membership* utiliza uma implementação do protocolo SWIM (DAS; GUPTA; MOTIVALA, 2002), que de forma escalável permite que os controladores detectem a atividade de outros controladores, sejam eles novos ou já ativos no *cluster*. Observa-se Figura 7 o funcionamento de forma simplificada do serviço de *membership* durante a descoberta de novos controladores. No passo 1, cada controlador conecta-se somente ao *datastore*. No passo 2, é propagado o evento de conexão a todas as instâncias do *datastore*. Após todas as instâncias do *datastore* possuírem o evento, notificam aos controladores ativos no *cluster* de que um novo controlador ingressou (passo 3). Desta forma, cada controlador ativo inicia uma conexão ponto-a-ponto com o novo controlador (passo 4).

O serviço de *mastership* provido pelo *datastore* considera a latência de comunicação e o número de dispositivos conectados a um controlador como critério para balanceamento

Figura 7 – Processo de descoberta de novos controladores.



Adaptado de: (HALTERMAN, 2018).

de carga. Para tal, é definido que o primeiro controlador a se conectar a um dispositivo Openflow e estabelecer a conexão terá o papel de *master*, ou seja, será o controlador principal. Os demais controladores conectados assumem o papel de *slave*.

Os dados da topologia e as regras de fluxo são sincronizados utilizando o algoritmo *primary-backup*. Com isso o ONOS armazena a visão global da topologia em memória. Isso permite que o acesso à topologia da rede possua baixa latência, visto que é utilizada uma visão armazenada localmente. No entanto, é possível que a visão da topologia seja brevemente diferente entre os controladores, de forma a gerar inconsistências na visão local, pois o mecanismo de replicação implementado não garante que as atualizações sejam entregues a todos os controladores. Um sistema puramente baseado em replicação otimista gradativamente pode perder a sincronia, necessitando mecanismos adicionais para correção de eventuais inconsistências. Para tal, um protocolo epidêmico (“gossip”) é utilizado para comparar, periodicamente, as informações entre dois controladores, corrigindo eventuais inconsistências.

Em um cenário onde o *master* falha logo após receber uma atualização da topologia

(uma porta foi desativada) um novo líder será eleito. No entanto, os controladores ativos não terão recebido a alteração na topologia e considerarão a porta como ativa se nenhuma ação for tomada pelo novo controlador. Para isso, o novo *master* consulta o estado de cada dispositivo em que se conecta. Regras que foram instaladas antes da falha do antigo controlador e não foram sincronizadas são excluídas para não gerar inconsistências. Desta forma, somente regras sincronizadas são mantidas nos dispositivos, apesar deste comportamento ser configurável.

É importante destacar que, no ONOS, se um controlador falhar, existe uma fila de controladores para cada dispositivo. Estes controladores recebem os mesmos eventos do controlador principal (*master*) com a diferença que não são processados, pois aguardam que o *master* realize a sincronização para que seja mantida a consistência na rede.

3.2 OpenDayLight

Outro controlador muito utilizado é o OpenDayLight (ODL), possui o código-aberto, e é administrado e mantido pela *Linux Foundation*. O ODL foi projetado para diversos domínios de aplicações (centros de dados, provedores de serviço, entre outros). Uma importante funcionalidade arquitetural do ODL é a *Model-Driven Service Abstraction Layer (MD-SAL)* que permite a fácil e flexível integração de serviços de rede, independente dos protocolos utilizados na SBI e a possibilidade de controlar dispositivos heterogêneos (SDN ou não).

O principal foco do ODL foi acelerar a integração de dispositivos SDN em ambientes com redes legadas, permitindo que dispositivos não-SDN se comuniquem com dispositivos Openflow. De tal forma, o projeto adotou soluções proprietárias para interoperar com dispositivos de diversos fabricantes. Assim como em outros controladores SDN, é utilizada a separação em camadas para gerenciamento da infraestrutura. As aplicações na camada superior consomem e alteram o estado dos elementos do plano de dados através da camada do controle. Uma importante diferença arquitetural entre o ONOS e o ODL é a localização do *datastore*. No ODL, o *datastore* é integrado ao controlador e provê a comunicação entre os controladores ativos.

Para que o ODL opere em modo distribuído é utilizado a aplicação *odl-mdsal-clustering*, responsável por sincronizar a *MD-SAL* através de todos os controladores disponíveis no *cluster*. O ODL provê a separação do *datastore* utilizando uma estrutura de árvore. A árvore de configurações representa o estado desejado do sistema e da rede, que é construída pelas aplicações, enquanto que a árvore de dados operacional é obtida por provedores de serviços utilizando a MD-SAL, e reporta o estado do sistema (LF, 2018).

A sincronização de estados utiliza consistência forte (Seção 2.2.2), provida pelo *framework* akka (LIGHTBEND, 2020). Desta forma, é garantido que todos os controladores

possuam uma visão consistente do estado dos dispositivos no plano de dados. As informações inseridas com sucesso, nas árvores de dados, não são perdidas em caso de falha no controlador. Para isso é importante que o *datastore* esteja replicado em mais do que um controlador. Para determinar a falha de um controlador é utilizado o serviço de detecção provido pelo *akka* que implementa um detector *accrual* chamado *Phi* (ϕ) (HAYASHIBARA et al., 2004), que devido a utilização do histórico da comunicação entre os controladores permite que seja reduzido o número de falsas suspeitas.

Tanto o ODL quanto o ONOS utilizam em seus *datastores* o conceito de *shards*, em que os dados são divididos em partições e cada partição armazena uma cópia dos dados. Desta forma, é garantido a disponibilidade dos dados, mesmo se um controlador ou (no caso do ONOS) uma instância do *datastore* falhar.

O número de partições do *datastore* é definido em arquivos de configuração carregados na inicialização dos controladores. No ODL são definidos por padrão três principais divisões do *datastore*: *default*, *topology* e *inventory*. Ainda nos arquivos de configuração é possível definir quais controladores irão armazenar cada uma das divisões do *datastore*. Com isso o número de partições e em quais controladores são alocadas estas partições impacta diretamente no desempenho dos controladores (SUH et al., 2017). Isto se deve, pois cada partição executa uma instância independente do algoritmo *Raft*.

3.3 Resultados

Esta Seção apresenta um estudo detalhado sobre o funcionamento do plano de dados e do plano de controle nos controladores ONOS e ODL. Em especial, a Seção 3.3.1 avalia o plano de dados e se concentra no comportamento dos controladores SDN considerando falhas em enlaces de comunicação e/ou *switches* com diferentes tipos de tráfego, levando em consideração o modo de instalação de regras proativo ou reativo. Logo após, na Seção 3.3.2 é avaliado o plano de controle em modo distribuído em relação as métricas de vazão e latência sob a perspectiva das aplicações (*northbound*) e dos dispositivos no plano de dados (*southbound*). Também é analisado o impacto do modelo de consistência adotado por cada controlador em relação as métricas de latência e vazão. Além disso, é avaliado o tempo de sincronização entre os controladores e o tempo após a falha do *master* para que o plano de controle fique estável. Com base nestes resultados foi avaliada a consistência da tabela de fluxos dos dispositivos após a falha do controlador principal (*master*).

3.3.1 Avaliação no Plano de Dados

Nesta seção são apresentados experimentos utilizados para avaliar o comportamento e desempenho dos controladores ODL e ONOS ao lidar com falhas que implicam no funcionamento do plano de dados. A avaliação foi realizada considerando os controladores

centralizados e sem qualquer mecanismo adicional para suportar falhas. Ou seja, deseja-se verificar como os controladores lidam com as falhas ocasionadas por interrupções de enlaces ou falhas nos próprios comutadores (*switches*).

Para realização dos experimentos foram utilizas duas máquinas com processador Intel Core i7-8700 de 3.20GHz, 16GB memória executando o sistema operacional Ubuntu 18.04.3. Foi utilizada uma topologia virtual provida pelo simulador Mininet(MININET, 2020). O tráfego entre os *hosts* da topologia foi gerado pela ferramenta *netperf*(HP, 2020). Os casos de testes avaliam o comportamento dos controladores diante de falhas no plano de dados e a continuidade dos serviços executados com a utilização de fluxos TCP e UDP.

Como critério de avaliação foi considerado o tempo para restabelecer cada conexão após a falha de um enlace ou *switch*. A execução de cada controlador foi realizada em uma máquina distinta do simulador.

As topologias utilizadas consideram a comunicação entre o dispositivo de origem e de destino da seguinte forma: (*i*) sem caminhos alternativos (Figura 8); (*ii*) um caminho alternativo (Figura 10); e (*iii*) dois caminhos alternativos (Figura 12). São consideradas falhas temporárias nos enlaces, pois estes são restabelecidos. Falhas do tipo *crash* são injetadas nos testes com o *switch*, o restabelecimento do serviço se dá pela inserção de um novo *switch*, restabelecendo a topologia inicial. O período de falha no *switch* é determinado pelo momento em que ocorre a parada do *switch* até que um novo seja iniciado. Os dados coletados foram extraídos capturando o tráfego durante a execução do *netperf* entre os *hosts H1* e *H2*. Para cada teste foram realizadas 3 repetições e analisado se realmente o comportamento se repete. Realizada esta análise com a constatação da igualdade entre os resultados, então os dados são representados graficamente utilizando uma das repetições obtidas no experimento.

Nos experimentos dos casos de 1 a 3 (Seções 3.3.1.1 a 3.3.1.3) os controladores estão no modo reativo. A seção 3.3.1.4 analisa o ODL no modo proativo. Foi definido um *timeout* de 5 segundos para que as regras instaladas nos *switches* expirem, fazendo com que as regras instaladas anteriormente não interfiram no comportamento do controlador. A Tabela 1 exibe as versões utilizadas na execução dos experimentos. As versões Beryllium-SR4 e Junco dos controladores ODL e ONOS, respectivamente, utilizadas em (VILCHEZ; SARMIENTO, 2018) são versões antigas destes controladores. A avaliação dos controladores foi realizada comparando as versões antigas com as versões mais recentes (Oxygen-SR4 e Sparrow, do ODL e do ONOS respectivamente) que possuem a funcionalidade de encaminhamento de pacotes na camada 2.

Nas subseções a seguir serão apresentados os dados coletados, evidenciando o tempo de resposta de cada um dos controladores. Em cada experimento, a falha de enlace ocorre exatamente aos 15s de execução e restaurada aos 30s. Perfazendo um período de 15s de inatividade. Para a interrupção foi utilizado o enlace em que os dados estavam sendo

Tabela 1 – Versões utilizadas dos controladores.

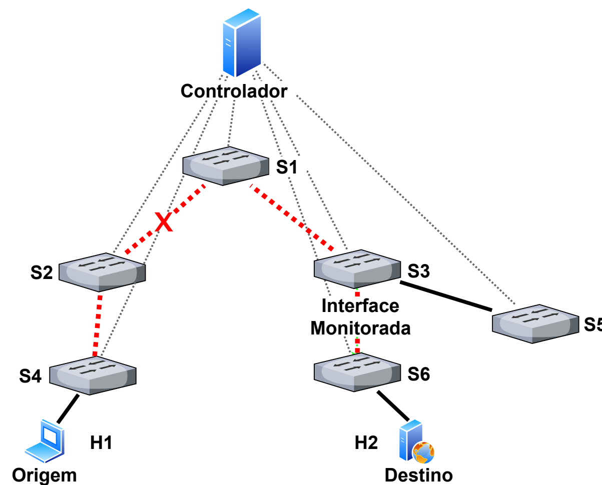
OpendayLight	**ONOS**
Beryllium-SR4 (0.4.4)	Junco (1.9.0)
Oxygen-SR4 (0.8.4)	Sparrow (2.2.0)

transmitidos, forçando o controlador a instalar novas regras para o caminho alternativo.

3.3.1.1 Caso 1 - Falha de enlace sem caminhos alternativos.

Uma falha nos enlaces de comunicação, quando não existem caminhos alternativos, implica na interrupção do serviço até que o enlace seja restaurado. A topologia da Figura 8 representa o cenário de falha com apenas um caminho entre origem (H1) e destino (H2) por onde o tráfego é encaminhado.

Figura 8 – Topologia utilizada no Caso 1. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo "X" representa o enlace interrompido.



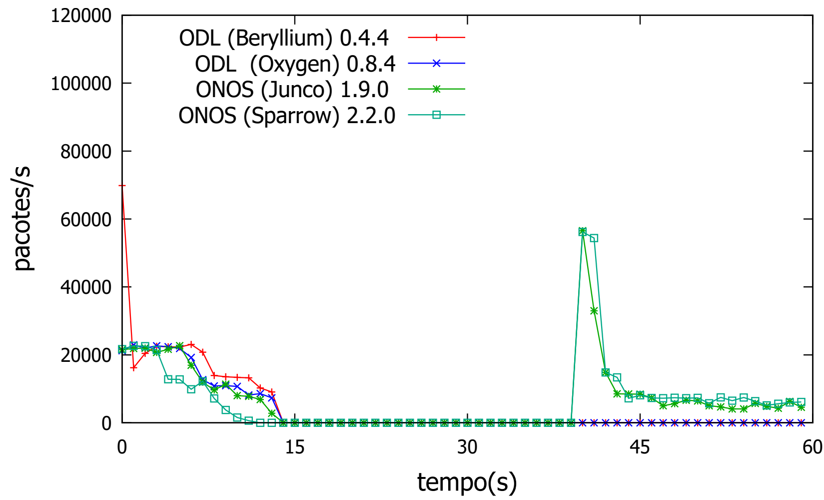
Fonte: O Autor.

Na Figura 9 é apresentada, no eixo y, a taxa de pacotes por segundo durante o período de captura, com a falha no enlace entre os *switches S2* e *S1*. No eixo x, é possível observar o instante da falha que ocorre entre o período de 15s até 30s. Conforme esperado, durante o período de inatividade não houve tráfego entre os *hosts H1* e *H2*. No entanto, depois de restabelecida a conexão, é possível notar a diferença no comportamento entre os experimentos. Utilizando fluxos TCP (Figura 9a) o ONOS teve um atraso de 10 segundos para restabelecer o serviço, independente da versão utilizada. O ODL, mesmo com a restauração do enlace, não conseguiu restabelecer o serviço para a aplicação em ambas as versões.

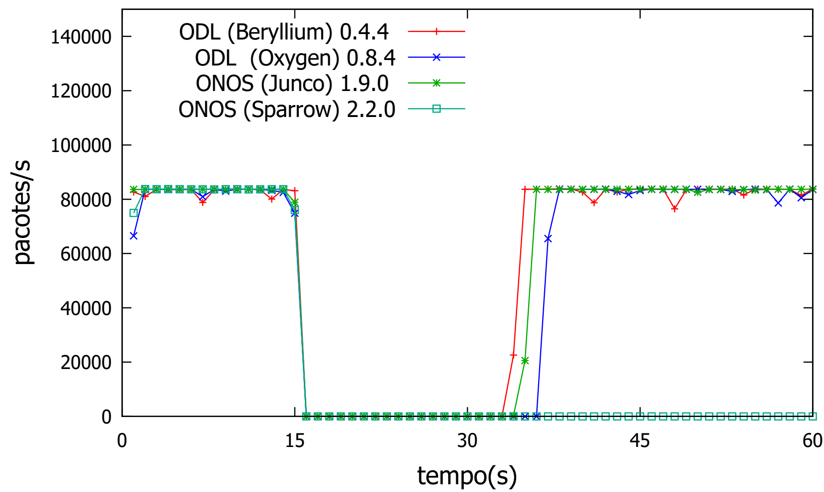
Com fluxos UDP (Figura 9b) o controlador ONOS, na versão Junco (1.9.0), foi capaz de restabelecer o serviço com um atraso de 4 segundos. Na versão Sparrow (2.2.0) do ONOS, após a falha do enlace, a conexão não foi restabelecida. A causa da interrupção

Figura 9 – Falha de enlace sem caminhos alternativos.

(a) Fluxo TCP



(b) Fluxo UDP



do ONOS foi estudada, mas não foi possível compreender o que levou o controlador a este comportamento. O controlador ODL, na versão Beryllium-SR4 (0.4.4) restabeleceu a conexão após 3 segundos do restabelecimento do enlace. Ao passo que, na versão Oxygen-SR4 (0.8.4), houve um atraso de 6 segundos para a restauração do serviço.

Para entender os resultados do ODL em relação ao TCP é importante relembrar o comportamento do protocolo TCP. Utilizando o protocolo TCP para comunicação entre as aplicações, quando ocorre a suspensão de um enlace, os pacotes perdidos são retransmitidos. Os pacotes são retransmitidos até que o destino confirme o recebimento ou que seja alcançado o tempo máximo definido no sistema para retransmissão. Cada retransmissão sem sucesso faz com que o tempo para a próxima retransmissão aumente exponencialmente. Além disso, para se comunicar na rede com outros dispositivos é necessário que a origem resolva o endereço MAC do destino através de requisições ARP. Uma vez obtido o endereço MAC do dispositivo de destino este é inserido em uma *cache*,

e só são geradas novas requisições ARP após a entrada na cache expirar.

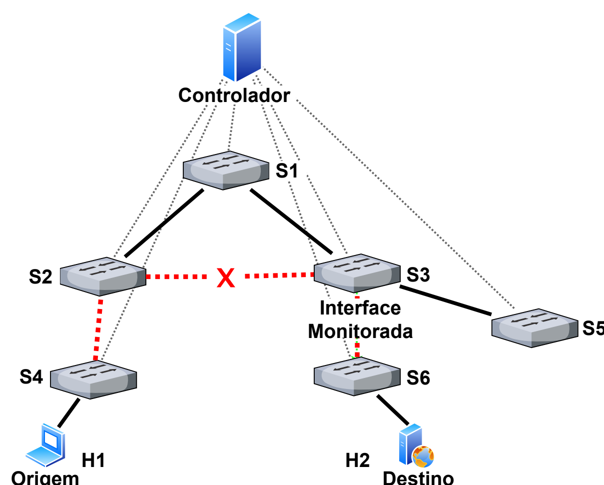
Levando isso em consideração, o ODL utiliza pacotes ARP para identificar a comunicação entre dois dispositivos, de tal forma, novas requisições ARP não são geradas quando o enlace é restabelecido. Além disso, quando uma aplicação está aguardando o tempo exponencial para enviar novos pacotes TCP, a entrada na tabela cache expira e uma nova requisição ARP é gerada. No entanto, devido a aplicação estar aguardando, as regras instaladas nos *switches* expiram, e ocorre novamente erro ao enviar o pacote. As regras de encaminhamento no *switch* expiram devido ao *idle_timeout* de 5 segundos.

3.3.1.2 Caso 2 - Falha de enlace com um único caminho alternativo.

Neste cenário existem dois caminhos entre os *hosts* avaliados (*H1* e *H2*), conforme a topologia da Figura 10. O controlador interliga os *hosts* pelo enlace S2-S3 indicado pela linha pontilhada em vermelho.

Em um cenários ideal, espera-se que na falha de um dos enlaces a comunicação ocorra através do caminho alternativo, de forma a garantir a continuidade do serviço oferecido. Neste experimento são avaliados: o enlace escolhido pelo controlador para comunicação entre os dois equipamentos, a utilização de um enlace alternativo, após uma falha e o comportamento do controlador com a restauração do enlace falho.

Figura 10 – Topologia utilizada no Caso 2. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo “X” representa o enlace interrompido.



Fonte: O Autor.

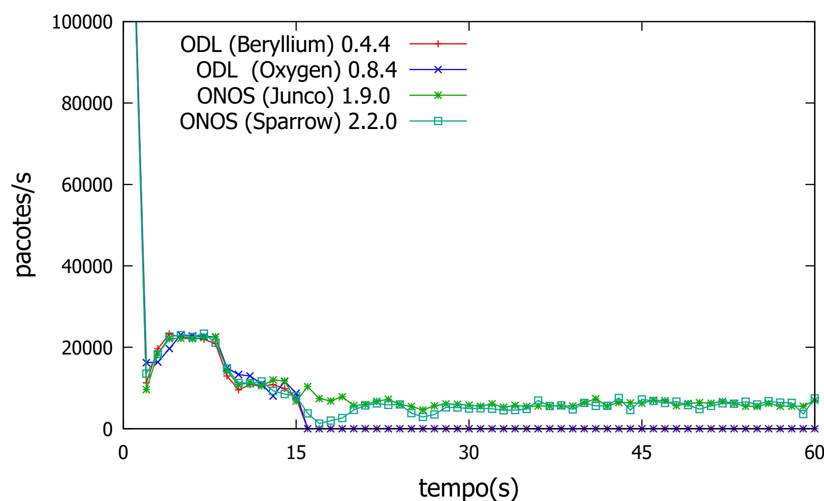
Ao avaliar fluxos TCP, é possível observar na Figura 11a o comportamento dos controladores durante a execução do experimento. Nas versões avaliadas do ONOS o tempo para restaurar o tráfego (detecção de falha no enlace e alteração na tabela de fluxos) foi menor que 1 segundo, não afetando a disponibilidade do serviço. Devido a presença de um enlace alternativo. Com o retorno do enlace falho o caminho alternativo instalado

no *switch*, pelo ONOS, continuou a ser utilizado. O ODL, além de interromper o serviço quando houve queda no enlace, não restabeleceu a conexão após o enlace ser restaurado.

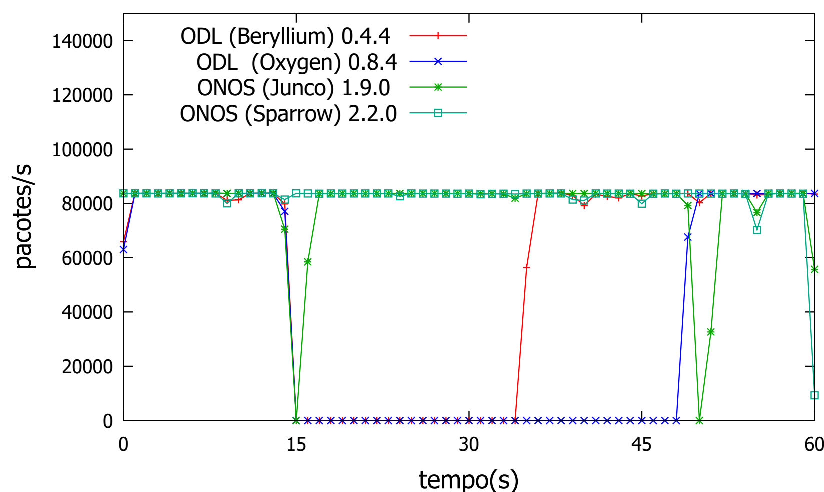
Com fluxos UDP, Figura 11b, o ONOS, na versão 1.9.0, detectou a falha do enlace em um tempo menor que 2 segundos, enquanto que na versão mais recente o tempo para detecção e restauração do serviço foi menor que 1 segundo, não interrompendo a execução do serviço. O ODL não foi capaz de manter a comunicação durante o período de falhas. Considerando a versão 0.4.4 do ODL, após 5 segundos da restauração do enlace, a comunicação foi restabelecida. Na versão 0.8.4 do ODL, foram necessários 9 segundos para que a comunicação fosse restabelecida. O ODL não utilizou o caminho alternativo durante ou após a falha, devido a implementação do módulo de encaminhamento no controlador que não trata corretamente atualizações na topologia.

Figura 11 – Falha de enlace com um caminho alternativo.

(a) Fluxo TCP



(b) Fluxo UDP



Como se observou, os resultados obtidos mostram que o ODL age diferente aos fluxos (TCP e UDP): com fluxos TCP o serviço não é restabelecido após a falha do

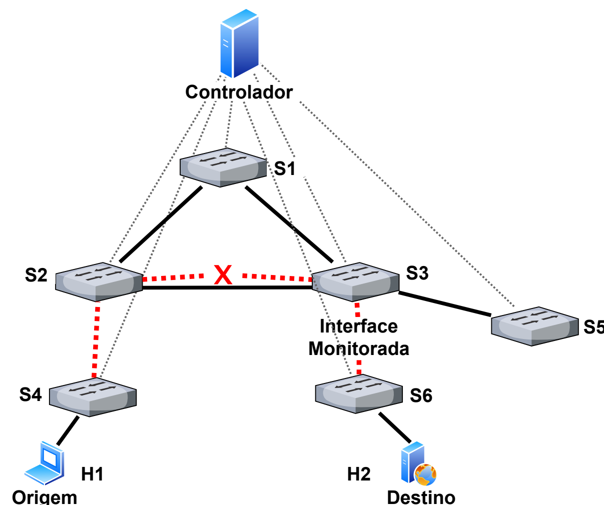
enlace; já com fluxos UDP o ODL restabelece a comunicação com atraso após o enlace ser restaurado, apesar de haver um caminho alternativo ocioso. O ONOS, por sua vez, utilizou o enlace alternativo em ambas versões, independente de fluxos TCP ou UDP e não interrompeu a continuidade do serviço.

3.3.1.3 Caso 3 - Falha de enlace com dois caminhos alternativos.

Este cenário avalia a queda de conexão e a possibilidade do controlador escolher mais do que um caminho alternativo (ver Figura 12). A topologia permite, de acordo com o instante do experimento, que o controlador escolha um dentre três enlaces para estabelecer a conexão. A Figura 13 ilustra o comportamento do tráfego durante o período de falhas.

Mesmo havendo a possibilidade de escolher entre os enlaces *S1* e *S3*, tanto o ONOS quanto o ODL escolheram utilizar o enlace com o *switch* mais próximo *(S3)*. No ODL com fluxos UDP, o tempo de restabelecimento de enlace foi menor do que o observado no caso da topologia da Figura 10. No ONOS, em ambas versões, utilizando fluxos TCP (Figura 13a), o serviço não foi interrompido. Com fluxos UDP (Figura 13b), a versão 1.9.0 do ONOS interrompeu o serviço até a detecção da falha e a instalação de regras nos *switches* para utilização de fluxos alternativos.

Figura 12 – Topologia utilizada no Caso 3. Setas tracejadas indicam a rota utilizada pelo controlador. O símbolo “X” representa o enlace interrompido.

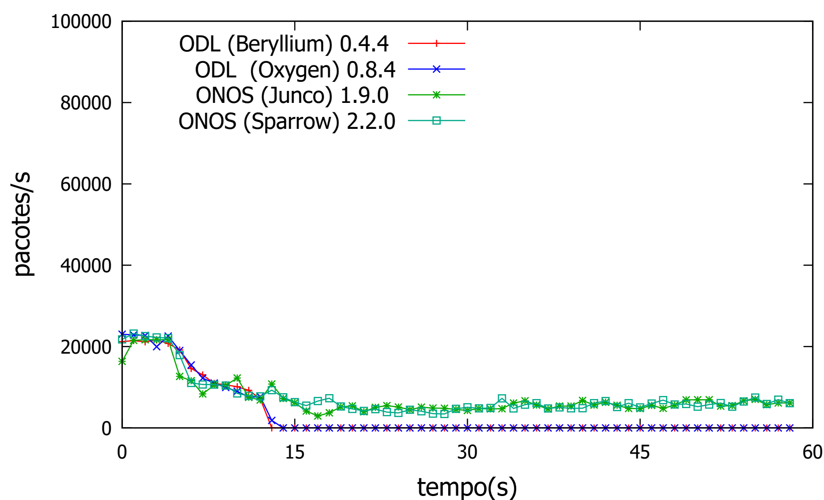


Fonte: O Autor.

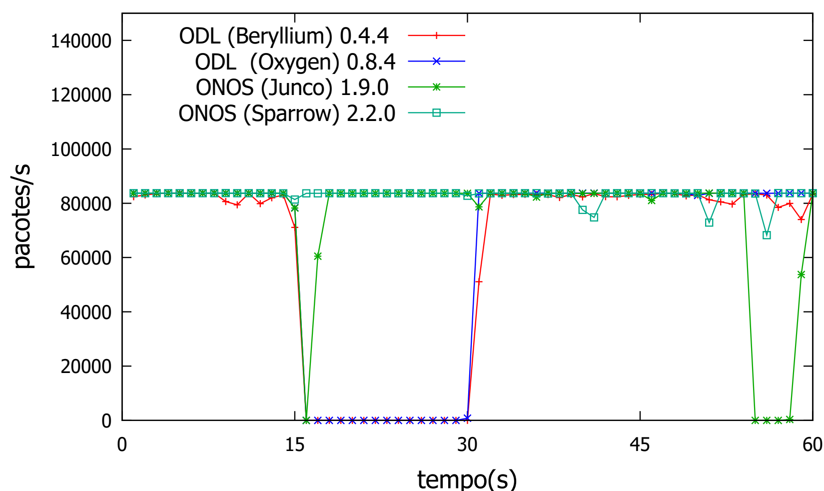
Na Figura 13a é possível observar na captura com fluxos TCP, o ODL, como nos casos anteriores, não manteve a continuidade do serviço após a queda do enlace, mesmo com um caminho alternativo e ocioso. O ONOS, por sua vez, com tempo de detecção inferior a 1 segundo, não interrompeu o serviço, visto a presença de enlaces alternativos.

Figura 13 – Falha de enlace com dois caminhos alternativos.

(a) Fluxo TCP



(b) Fluxo UDP



Com fluxos do tipo UDP, analisando a Figura 13b, observa-se que, em ambas as versões do ODL, retorna a comunicação somente após o restabelecimento do enlace. Observa-se também que após a restauração do enlace, com atraso menor que 1 segundo, o ODL restabelece a comunicação apesar de haver dois enlaces disponíveis e operacionais durante o período de falha. O ONOS, na versão 2.2.0, restabelece a conexão com atraso menor que 1 segundo. Ao passo que, a versão 1.9.0 do ONOS possui um atraso entre 1 e 2 segundos para detecção e instalação de novos fluxos, interrompendo o serviço durante este período.

A seguir compara-se os resultados obtidos, para os mesmos casos de teste realizados em (VILCHEZ; SARMIENTO, 2018), em que foi utilizado fluxos UDP, conforme descrito na Seção 3.4. Para o caso 1 do controlador ODL, o trabalho de Vilchez & Sarmiento descreve um tempo de 20 segundos para reação à falha, 17 segundos a mais do que o verificado nos experimentos deste trabalho. Para o ONOS, ainda no caso 1, o autores

indicaram que não houve atraso para o restabelecimento da conexão. Já os resultados obtidos neste artigo foram necessários 3 segundos para que o serviço fosse restabelecido. Para o caso 2 do ODL, os resultados foram semelhantes: o ODL falhou ao manter a continuidade do serviço após a falha. O ONOS, diferentemente do que observado na versão avaliada pelos autores, não restabeleceu a conexão instantaneamente, foi necessário um tempo menor que 2 segundos. No caso 3 o ODL reagiu da mesma forma que observado no trabalho de Vilchez & Sarmiento reagindo instantaneamente após a volta do enlace. O ONOS, ao contrário do que observado, apresentou um atraso menor do que 2 segundos. Ou seja, os resultados obtidos são semelhantes aos obtidos neste artigo, com diferenças para o tempo de resposta para restaurar a conexão após o restabelecimento dos enlaces, o que pode ser explicado pelo ambiente de execução.

3.3.1.4 ODL - Modo proativo

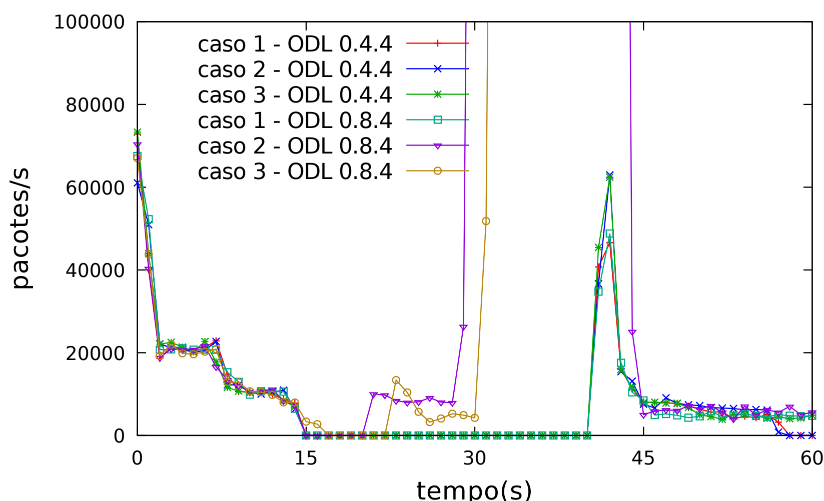
O ODL em modo proativo não recebe eventos *PACKET_IN*, ou seja, novas regras para tratamento de fluxos devem ser inseridas proativamente. São poucas as aplicações que suportam a utilização do modo proativo puro (OPENDAYLIGHT, 2019). Comparar o modo proativo é fundamental para analisarmos o comportamento do ODL em seus modos de funcionamento. Neste experimento é analisado comportamento do controlador diante de falhas no plano de dados e a continuidade dos serviços executados utilizando fluxos TCP. Além disso, é avaliada a evolução do ODL em relação a versão Berrylium-SR4 (0.4.4) com a versão Oxygen-SR4 (0.8.4). A execução dos experimentos foi realizada utilizando as topologias presentes nas Figuras 8, 10 e 12.

Observando a Figura 14, no caso 1 (Figura 8) em que não existem caminhos alternativos, durante a interrupção do enlace o tráfego deve ser interrompido, sendo avaliado o tempo necessário para que o serviço seja restabelecido. No experimento realizado, o ODL na versão 0.4.4 e na versão 0.8.4, foram necessários 10 segundos para que o serviço fosse restabelecido após a restauração do enlace.

No caso 2 (Figura 10) existe dois caminhos entre o host de origem (H1) e o host de destino (H2). Avaliou-se o comportamento do controlador e o tempo necessário para o restabelecimento do serviço. No experimento ambas as versões utilizaram o caminho *S3-S4* para encaminhar os fluxos. No ODL na versão 0.4.4 foram necessários 10 segundos para o restabelecimento do serviço após o enlace ser restaurado. Na versão 0.8.4, foram necessários 6 segundos para que o controlador identificasse a falha e redirecionasse para o caminho alternativo. Contudo, durante 10 segundos, após o restabelecimento do enlace, houve uma inundação de pacotes, com a taxa de pacotes, em média, 26 vezes maior do que a taxa normal.

No caso 3 (Figura 12), existem três caminhos entre o host de origem (H1) e o host de destino (H2), sendo que um caminho é sempre utilizado, restando dois de redundância.

Figura 14 – ODL - modo proativo



O ODL na versão 0.4.4, como no caso 2 (Figura 10), levou 10 segundos após a restauração do enlace para que o serviço fosse restaurado. Na versão 0.8.4 do ODL, foram necessários 8 segundos para que o serviço fosse restaurado. Conforme observado no caso 2, utilizando o modo reativo, houve um período de 10 segundos de inundação de pacotes após a restauração do enlace. A inundação de pacotes ocorre devido ao tempo para que a topologia de rede seja atualizada no controlador ao ser detectado novos enlaces, quando executado no modo proativo.

O ODL, mesmo em modo proativo, não reagiu a falhas de forma satisfatória, visto que o ONOS possibilita que com a presença de enlaces redundantes seja garantida a disponibilidade nos serviços do plano de dados. No ODL, as melhorias no modo proativo não foram capazes de fazer com que o serviço fosse mantido sem interrupção, mesmo com a presença de caminhos alternativos para os fluxos.

3.3.1.5 Falha do switch

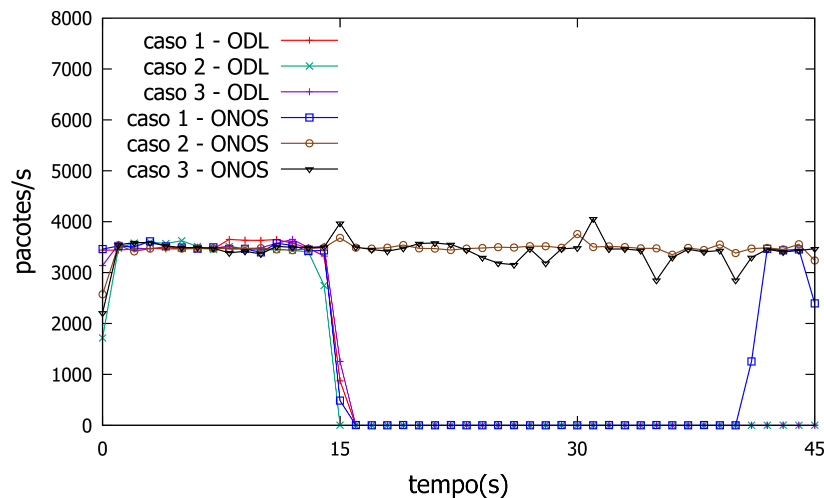
A falha em *switches* é semelhante à falha de enlace, visto que o controlador deve implementar mecanismos que detectam que houve uma falha no *switch* e redirecionar o tráfego para um caminho alternativo. No entanto, os *switches* iniciam com a tabela de fluxo vazia, podendo impactar no comportamento dos controladores. Os experimentos são executados utilizando as topologias descritas anteriormente. Neste experimento é avaliado o comportamento do ONOS e do ODL utilizando o modo reativo, nas versões Sparrow (2.2.0) e Oxygen-SR4 (0.8.4), respectivamente.

Nos experimentos com o fluxo TCP, Figura 15a , utilizando a topologia do caso 1 (Figura 8), o ONOS levou 11 segundos para que o serviço fosse restabelecido, após a restauração do enlace. O ODL não foi capaz de restaurar o serviço como observado na Seção 3.3.1.1 . Utilizando fluxos UDP, Figura 15a, o ODL levou 10 segundos para que o

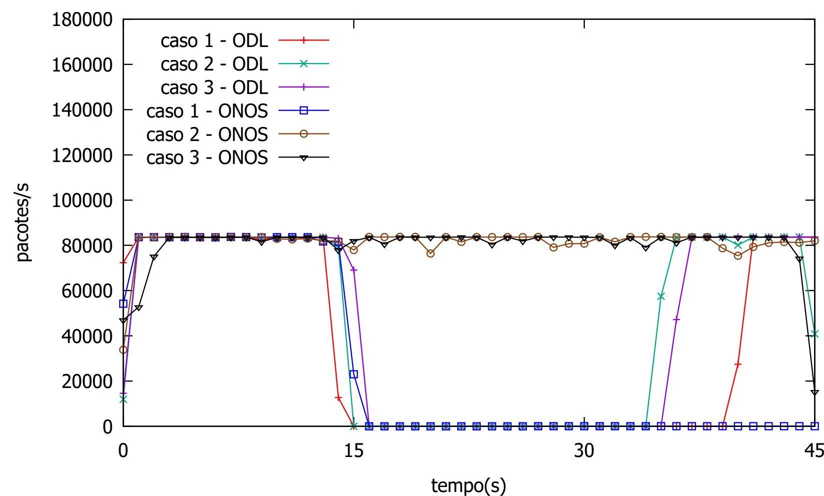
serviço fosse restaurado, após o retorno do enlace. A ferramenta *netperf* falhou ao realizar a geração de tráfego para o ONOS com fluxos UDP.

Figura 15 – Falha do *switch*

(a) Fluxo TCP



(b) Fluxo UDP



Avaliando o caso 2 (Figura 10) com fluxos TCP, o ODL não foi capaz de restabelecer o serviço, mesmo após o enlace ser restaurado. O ONOS, não interrompeu o serviço, com o tempo de detecção da falha do *switch* menor que 1 segundo. Utilizando fluxos UDP, o ODL não identificou o caminho alternativo e levou 5 segundos para identificar que o enlace antigo havia sido restaurado. O ONOS com fluxos UDP não interrompeu o serviço, com o tempo de detecção de falhas e reconfiguração dos equipamentos menor que 1 segundo.

No caso 3 (Figura 12) os resultados foram semelhantes ao caso 2, visto que o ONOS, em fluxos TCP e UDP, mantém a disponibilidade do serviço. O ODL, utilizando fluxos TCP não consegue restabelecer o serviço após a restauração do enlace. Utilizando fluxos UDP, o ODL levou 6 segundos para restabelecer o serviço após o enlace antigo ser restaurado.

quadro 2 Resumo dos experimentos realizados no plano de dados.

Controlador	Versão	Tipo de Tráfego	Tipo de Falha	Modo do controlador	Caso 1	Caso 2	Caso 3
ODL	0.4.4 (Beryllium)	TCP	Enlace	Reativo	Não se recuperou	Não se recuperou	Não se recuperou
			Enlace	Proativo	10 s	Não utilizou enlace alternativo	Não utilizou enlace alternativo
		UDP	Enlace	Reativo	3 s	Não utilizou enlace alternativo	Não utilizou enlace alternativo
	0.8.4 (Oxygen)	TCP	Enlace	Reativo	Não se recuperou	Não se recuperou	Não se recuperou
			Switch	Reativo	Não se recuperou	Não se recuperou	Não se recuperou
			Enlace	Proativo	10 s	6 s (inundação após enlace ser restaurado aos 30 s)	8 s (inundação após enlace ser restaurado aos 30 s
		UDP	Enlace	Reativo	6 s	Não utilizou enlace alternativo	Não utilizou enlace alternativo
			Switch	Reativo	10 s	Não utilizou enlace alternativo	Não utilizou enlace alternativo
ONOS	1.9.0 (Junco)	TCP	Enlace	Reativo	10 s	< 1 s	< 1 s
		UDP	Enlace	Reativo	4 s	< 2 s	< 2 s
	2.2.0 (Sparrow)	TCP	Enlace	Reativo	10 s	< 1 s	< 1 s
			Switch	Reativo	11 s	< 1 s	< 1 s
		UDP	Enlace	Reativo	Falha no netperf	< 1 s	< 1 s
			Switch	Reativo	Falha no netperf	< 1 s	< 1 s

Fonte: O Autor.

Nesta seção foi realizada uma avaliação de falhas no plano de dados utilizando os controladores ONOS e ODL. No Quadro 2 é apresentado um resumo dos resultados obtidos nos experimentos executados. A marcação "não se recuperou" indica que durante e após (enlace restabelecido) a falha do enlace não houve conectividade entre os *hosts* de origem e destino. A falha no *netperf* indica que não foi possível completar a execução utilizando a ferramenta *netperf* após as falhas nos enlaces. Na topologia sem redundância (caso 1) é apresentado o período para que o tráfego seja restabelecido entre os *hosts*. Nas demais topologias o período corresponde ao tempo necessário para que o tráfego seja restabelecido utilizando os enlaces alternativos. Nas topologias com redundância o ODL não foi capaz de utilizar outros caminhos, além do que continha o enlace que foi interrompido. Desta forma, o ONOS apresentou melhores resultados para detecção correção de falhas de enlace e alterações na topologia em relação ao ODL.

3.3.2 Avaliação no Plano de Controle

Nesta seção são avaliados experimentalmente os controladores ONOS e ODL em modo distribuído a partir da sua vazão (*throughtput*), latência e o custo para sincronizar novas regras nos múltiplos controladores. Os experimentos foram executados em um *cluster* formado por 14 máquinas com processador Intel Core i7-8700 CPU de 3.20GHz, cada máquina possui 12 núcleos e 16 GB de memória. O sistema operacional Ubuntu 18.04 com arquitetura x86_64 está instalado em cada computador. Todas as máquinas utilizadas nos experimentos estão conectadas a um *switch* com interfaces de 1Gbps. Os controladores foram conectados à *switches* virtuais (OPENVSWITCH, 2020) fornecidos pelo simulador Mininet (MININET, 2020). Todos os experimentos são executados variando de 3 até 13 controladores ativos. Cada controlador é executado em uma máquina física. As ferramentas Cbench (CBENCH, 2013) e o módulo *requests* do *python3* foram utilizadas para a geração de tráfego e são executados em uma máquina exclusiva e distinta dos controladores.

Os resultados de cinco experimentos são apresentados nas seções seguintes, sendo brevemente descritos a seguir. O primeiro experimento avalia a latência e a vazão dos controladores fazendo uso do software Cbench. O Cbench é uma ferramenta para análise de desempenho de controladores SDN através do protocolo Openflow que permite obter métricas de latência e vazão para a instalação de regras do plano de controle. Como visto na Seção 2.2, a sincronização do plano de controle implica em um custo adicional ao controlador devido à necessidade de manter uma visão logicamente centralizada do estado da rede. O segundo experimento avalia o impacto da sincronização dos controladores, mas sob a perspectiva das aplicações que fazem o controle da rede, visto que no experimento anterior os eventos gerados para instalação de regras são oriundos da *southbound* através da ferramenta Cbench. Aplicações instalam regras através de chamadas pela interface *northbound* do controlador com o objetivo de alterar o estado dos elementos de rede gerenciados pelo controlador. No terceiro experimento é analisado o tempo para os controladores sincronizarem seus estados de acordo com o modelo de consistência utilizando por cada controlador. No quarto experimento é avaliado o tempo necessário para que um novo controlador assuma os *switches* órfãos após a falha do controlador *master*. No quinto experimento é avaliada a consistência da tabela de fluxos dos *switches* após a falha do controlador *master*, visto que após a falha podem ocorrer inconsistências em relação às regras que já foram instaladas.

3.3.2.1 Vazão e Latência através da *Southbound*

Neste experimento é avaliada a latência e a vazão dos controladores em modo distribuído utilizando duas variáveis: quantidade de controladores e número de *switches*. Por latência entende-se o tempo necessário para um novo fluxo do plano de dados ser recebido pelo controlador e reencaminhado ao plano de dados com uma nova regra do

controlador. Nesse caso, no modo latência, o *Cbench* aguarda a instalação de uma regra de fluxo para que uma nova seja enviada ao controlador. Através desta medida é possível obter o tempo que uma regra leva para ser processada. A vazão é a medida que expressa a quantidade máxima de eventos que o controlador consegue processar, dado os recursos de hardware disponíveis. No modo vazão, o Cbench envia fluxos continuamente e conta a quantidade de regras recebidas do controlador (FLOW_MOD) em um determinado período de tempo.

As Figuras 16 e 17 apresentam os resultados da latência e vazão dos controladores ONOS e ODL obtidos com a ferramenta *Cbench*. É importante destacar que o *Cbench* se conecta somente a um dos controladores ativos. Nos experimentos foram utilizados 16, 32, 64, 128, 256 e 512 *switches* em cada execução e o número de controladores variou de 3 a 13. Por exemplo, quando o experimento conta com 3 controladores, o número *switches* variou de 16 a 512 e assim por diante. Em cada execução foram realizados 20 repetições e descartas as 10 primeiras (*warmup*), visto que existe um tempo para que os componentes dos controladores sejam iniciados.

Para o cálculo da latência e da vazão foi utilizado o total de regras recebidas do controlador e dividido pelo numero de *switches*. Obtendo, desta forma, o número médio de regras recebidos por *switch* do controlador em cada experimento. A latência corresponde à 1/L, tal que L é o número de respostas por *switch* do controlador. Deste modo é obtida a latência de resposta por regra. Com base nestes valores, a latência foi estimada para instalação de 10000 regras pelos controladores.

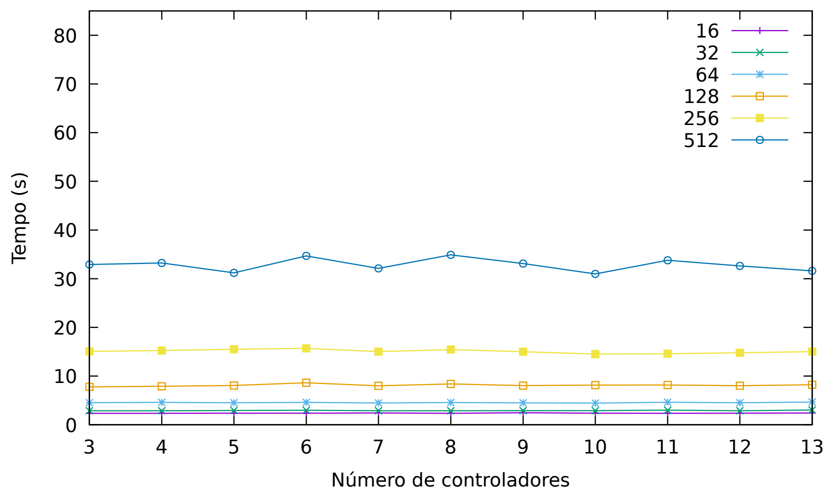
As Figuras 16a e 16b exibem a latência em milissegundos dos controladores ONOS e ODL para instalação de 10000 regras. No ODL, ao variar o número de *switches* houve um aumento na latência, visto que houve um número maior de requisições enviadas ao controlador. A latência média em segundos foi de 2.37, 2.91, 4.54, 8.11, 15.07, 32.82, respectivamente com 16, 32, 64, 128, 256 e 512 *switches*. Apesar do acréscimo na latência houve um aumento na quantidade de regras total recebidas pelo controlador, considerando todos os *switches*. Utilizando como base a execução com 16 *switches* na Figura 16a houve um aumento na quantidade de regras recebidas de 60,13%, 109,09%, 133,35%, 152,35% e 132,08%, respectivamente com 32, 64, 128, 256 e 512 *switches*.

No ONOS, o aumento de controladores apresentou taxas de latência inconstantes. A configuração com menor taxa de variação foi obtida utilizando 3 controladores (8.99%). A latência média em segundos foi de 3.53, 3.82, 10.51, respectivamente para 16, 32 e 64 *switches*, considerando as execuções que foram completadas em todas as configurações do *cluster*. É importante notar que com 128 e 256 *switches* foi possível executar o experimento somente até 8 controladores, enquanto que com 512 *switches* foi possível realizar a execução do experimento somente até 4 controladores. Nestes casos temos a latência média em segundos de 8.11, 15.07 e 32.82, respectivamente para 128, 256 e 512 *switches*.

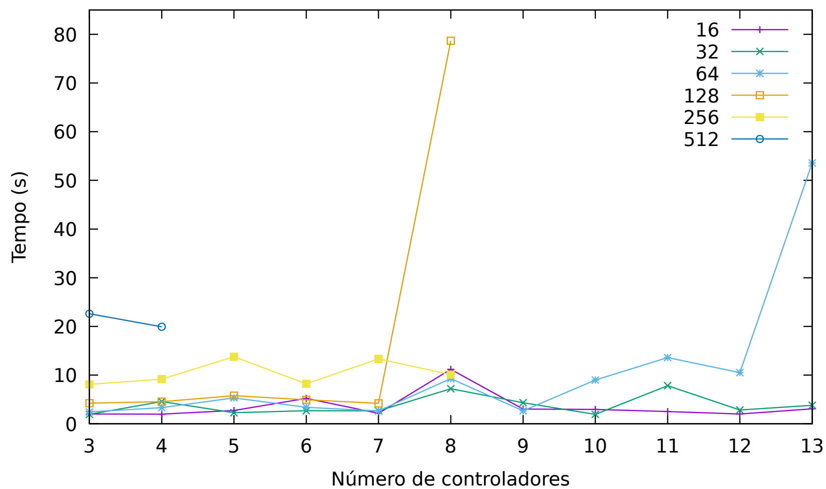
Apesar do aumento da latência, assim como no ODL, o total de respostas recebidas pelo controlador aumentou. De forma que, utilizando como base a configuração com 16 *switches* na Figura 16b, houve um aumento de 77,56% com 32 *switches* e de 135,65% com 64 *switches*, respectivamente. Nos cenários em que não foi possível executar os experimentos corretamente, considerando somente as execuções bem-sucedidas, temos que com base em 16 *switches* houve um aumento na quantidade de respostas de 296,42% com 128 *switches*, 339,81% com 256 *switches* e 314,41% com 512 *switches*.

Figura 16 – Latência média para instalação de fluxos através da *Southbound*.

(a) ODL.



(b) ONOS.

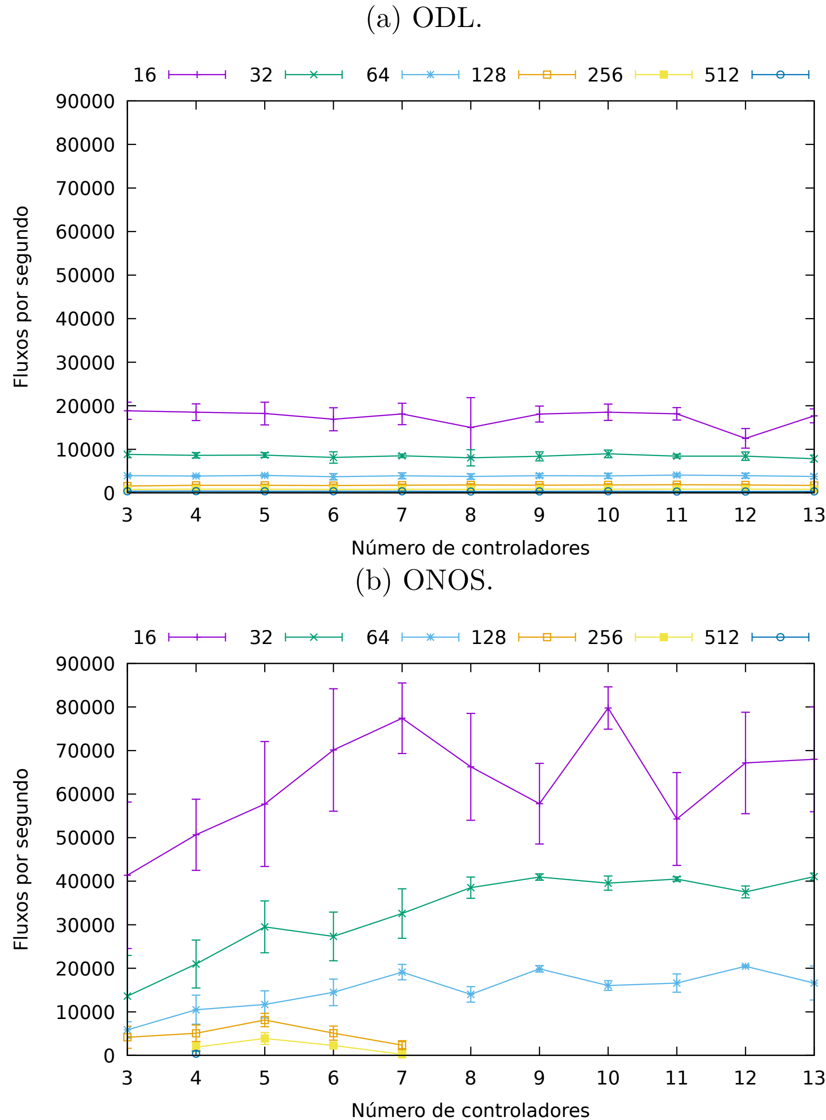


Podemos observar, comparando as execuções bem-sucedidas, nos controladores ONOS e ODL que a latência no ONOS foi maior do que no ODL. A latência média do ONOS foi de 48.66%, 31.20%, 131.26% maior do que no ODL, respectivamente com 16, 32 e 64 *switches*. No ONOS, a latência até 7 controladores é próxima à latência do ODL, no entanto, conforme o número de controladores aumentou houveram casos com aumento da latência. Mostrando desta forma, que conforme o número de controladores passa de 7, o

ONOS torna-se instável, mesmo na execução com 16 *switches*.

Figura 17 – Vazão média para instalação de fluxos através da *Southbound*.

(a) ODL.



(b) ONOS.

Nas Figuras 17a e 17b são exibidos os resultados da avaliação de vazão dos controladores. No ODL a vazão do controlador manteve-se pouco alterada com o aumento de controladores. A variação do número de *switches* teve impacto, visto que um número maior de fluxos são enviados ao controlador. Tomando como base a execução com 16 *switches* na Figura 17a houve uma diminuição na vazão por *switch* de 51.28%, 77.56%, 89.92%, 95.27% e 98.08%, respectivamente com 32, 64 , 128, 256, 512 *switches*. O número de controladores no ONOS não impactou na taxa de vazão, mas na estabilidade do controlador, conforme comentado anteriormente. Utilizando como base 16 *switches* houve diminuição na vazão de 47.56%, 76.07%, 92.10%, 96.71 e 99.39%, respectivamente com 32, 64, 128, 256 e 512 *switches*. É importante notar que nos experimentos com 128 e 256 *switches* não foram completadas as execuções a partir de 8 controladores, o controlador falhou ao tratar os fluxos gerados pela ferramenta. A execução com 512 *switches* somente foi possível

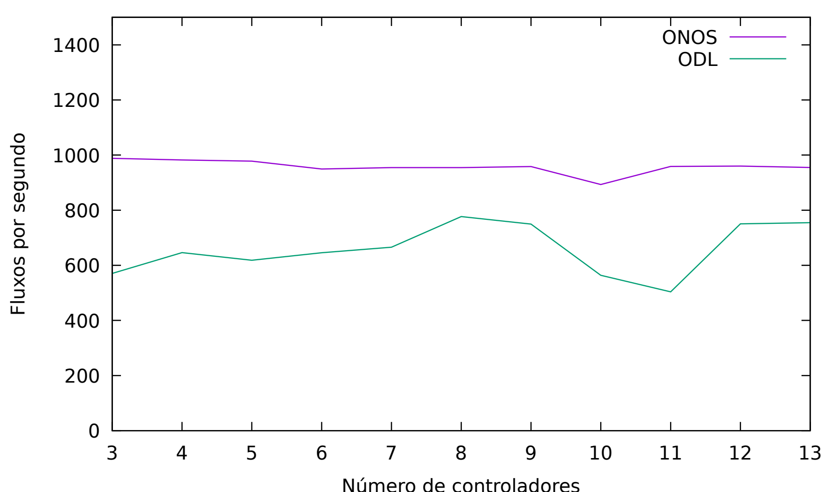
executar utilizando 4 controladores.

Ao analisarmos os experimentos executados com sucesso, a vazão do ODL foi 72.41%, 74.36% e 74.13% menor do que o ONOS em média, respectivamente para 16, 32 e 64 *switches*. Como observado nas figuras não foi possível avaliar o ONOS a partir de 8 controladores gerenciando 64 *switches* ou mais. Considerando os cenários que foram possíveis realizar experimentos sem quaisquer problemas, pode-se concluir que o ONOS superou consideravelmente o ODL quando analisamos a métrica vazão. No entanto, o ODL permite que seja utilizado um número maior de controladores, devido a sua estabilidade. Então, se a ideia é ter um ambiente escalável de controladores e *switches*, em que 7 controladores e 128 *switches* seja insuficiente, o ODL é o controlador indicado para tal ambiente.

3.3.2.2 Vazão através da *Northbound*

Neste experimento os controladores são avaliados na perspectiva das aplicações que realizam o controle da rede. Como as aplicações são responsáveis por aplicar políticas de rede nos dispositivos do plano de dados, é importante investigar se o número de controladores impacta na instalação de regras pelas aplicações utilizando a *northbound*. Neste experimento foi criada uma aplicação que envia regras de fluxo aos controladores utilizando a API REST durante 5 minutos. Para cada controlador foram criados 128 *threads* para envio de regras. E para cada configuração do grupo de controladores o experimento foi repetido 5 vezes.

Figura 18 – Vazão para instalação de fluxos através da *Northbound*.



A Figura 18 apresenta os resultados médios obtidos. É possível observar que aumentar o número de controladores não impactou na vazão da aplicação. Houve uma diferença média em que o ODL obteve 31% menos fluxos instalados do que o ONOS. É importante observar que como os controladores não estavam com a utilização máxima dos

recursos disponíveis, distribuir a carga de regras em vários controladores não diminuiu o número de regras instaladas. O gargalo observado foi na aplicação que aguardava o recebimento dos fluxos. É importante notar nesse experimento as características de sincronização de cada controlador. Como visto na Seção 3, o ODL utiliza consistência forte. O ONOS possui primitivas com os modelos de consistência forte e fraca, mas para instalação de regras Openflow é utilizado o modelo de consistência fraca.

Como apresentado na seção 3.1, o ONOS é um controlador que utiliza o modelo *lazy* para sincronização entre os controladores pertencentes ao *cluster*. Ao instalar fluxos pela API REST o controlador não aguarda a sincronização entre os demais controladores no *cluster*, respondendo às requisições realizadas através da API. Com isso a requisição é encaminhada ao *master* do dispositivo à ser instalado a regra de fluxo. Por outro lado, o ODL, como apresentado na seção 3.2, possui a característica de utilizar o *Raft* para sincronização das regras de fluxo entre os controladores pertencentes ao *cluster*. O que impacta em um modelo de consistência forte. A instalação de regras pela API REST é realizada após ser sincronizada entre todos os controladores. É importante notar que as regras não são aplicadas instantaneamente nos dispositivos do plano de dados ao serem instaladas nos controladores.

3.3.2.3 Tempo para sincronização

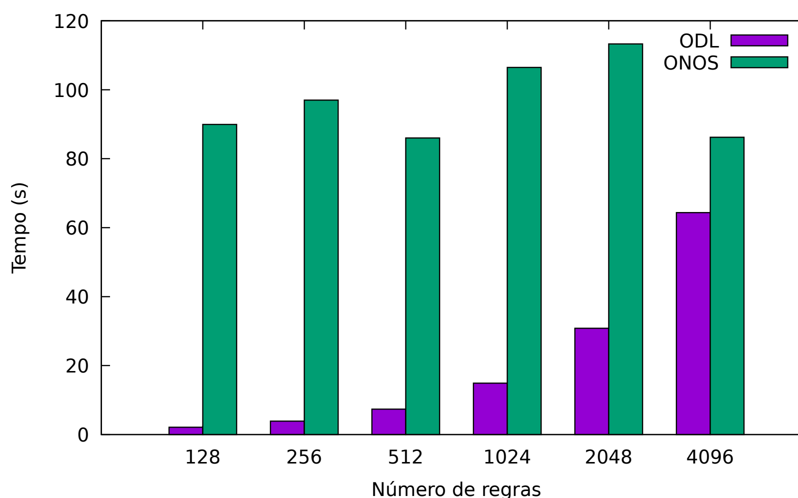
Os controladores em modo distribuído devem sincronizar informações entre os demais controladores do *cluster* para que possuam uma visão logicamente centralizada sobre as regras a serem instaladas no plano de dados. O modelo de consistência adotado por cada um dos controladores impacta no tempo para sincronização das regras. Vale lembrar que o ONOS adota um modelo de consistência fraca enquanto o ODL assume um modelo de consistência forte. Os controladores devem sincronizar o estado mesmo perante falhas no plano de controle, caso contrário regras instaladas pelas aplicações podem ser perdidas, causando inconsistências no plano de dados (KATTA et al., 2015; BOTELHO et al., 2016). O ONOS assume um modelo de sincronização *lazy* e o ODL aplica o algoritmo de consenso *Raft* entre todas as instâncias do controlador para garantir a consistência forte.

Dado os modelos de consistência e características próprias de cada controlador, neste experimento foi avaliado o tempo necessário para os controladores ONOS e ODL sincronizarem as regras entre si. De acordo com a documentação do ONOS, o período padrão para que um controlador inicie a sincronização com um de seus *backups* é de 2s (ONF, 2019), período mínimo esperado para sincronização entre os controladores. No ODL assim que uma regra é recebida, inicia-se o protocolo de sincronização e ao controlador responder à requisição a regra já está sincronizada entre todos os controladores. É importante destacar que as configurações padrões das aplicações instaladas nos controladores não

foram alteradas.

Para execução do experimento foi desenvolvido um *script* utilizando a biblioteca *requests* do python3 para envio de regras através da API REST dos controladores. Nas requisições através da API REST foram enviados 128, 256, 512, 1024, 2048 e 4096 regras. Para cada conjunto de regras foram realizadas 3 repetições do experimento. Para garantir a precisão de sincronização dos relógios dos computadores foi instalado nas máquinas o *PTPd* (*Precision Time Protocol daemon*) (PTPD, 2020).

Figura 19 – Tempo para sincronização de regras nos controladores ONOS e ODL.



Os resultados apresentados na Figura 19 são referentes à média do tempo de sincronização de regras nos controladores ONOS e ODL. Para obtenção dos dados no controlador ONOS foi modificada a aplicação *events* para que registrasse o momento de instalação das regras em cada controlador. O tempo para sincronizar cada regra foi obtido pela diferença de tempo entre o controlador que recebeu a regra (primário) e os demais controladores que receberam a regra (*backups*). O tempo apresentado corresponde à maior diferença observada para instalação da mesma regra. Por exemplo, se a regra foi instalada no instante X no controlador A e no instante Y e Z nos controladores B e C, então o tempo apresentado é a maior diferença entre Y-X e Z-X. O tempo de sincronização no ODL foi obtido registrando o tempo do envio e o tempo da resposta correspondente.

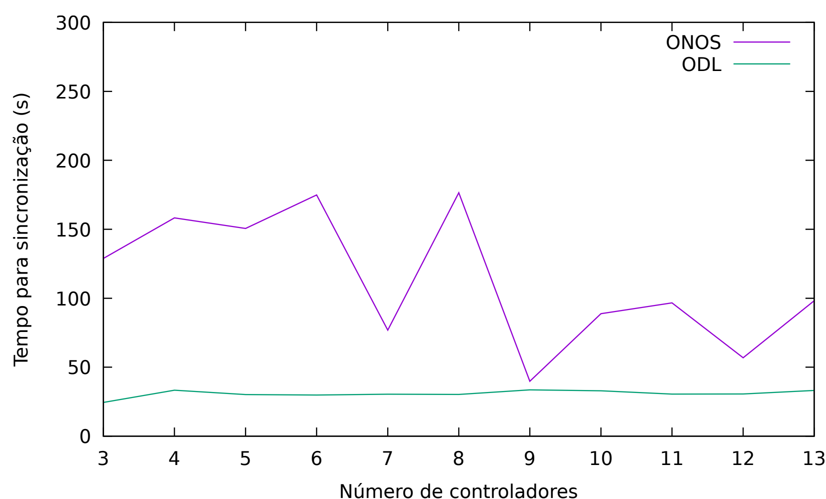
Conforme observado na Figura 19 a sincronização no controlador ONOS não ocorre conforme as especificações de 2 segundos. O tempo médio considerando a variação do número de controladores, em segundos, foi 89.93, 97.00, 85.99, 106.46, 113.29 e 86.20, respectivamente para 128, 256, 512, 1024, 2048, 4096 regras. O resultado é atribuído a constantes alterações no papel de controladores *backups*, visto que são eles que recebem as regras sincronizadas. Notou-se que o *Atomix*, responsável pelo serviço de *mastership*, detectava erroneamente que alguns controladores estavam falhos, fazendo com que os *backups* fossem removidos. É importante notar que as regras são sincronizadas somente

entre o primário e seus *backups*. Nas configurações padrões do ONOS o número máximo de controladores *backups* é 2, no entanto, devido às variações nos papéis, mais controladores receberam as regras.

Observou-se, no experimento com o ONOS, que houveram execuções sem *backups*, até, no máximo, 4 backups. Ou seja, para algumas execuções as regras não foram replicadas, permanecendo apenas no controlador primário. De forma que os dados exibidos são referentes somente às execuções que obtiveram a sincronização em pelo menos 1 backup. Considerando o total de 132 execuções do experimento, em 51.14% das execuções não houveram controladores backups disponíveis para sincronização.

Os tempos de sincronização, observados na Figura 19, significam constantes trocas de controladores *backups*. Tal fato faz com que o conjunto de regras precise ser sincronizado em novos controladores *backups* escolhidos durante a sincronização. Como exemplo, é exibido na Figura 20 os dados referentes à sincronização de 2048 regras nos controladores ONOS e ODL. No ONOS, é possível observar que houveram execuções com 9 e 12 controladores em que o tempo de sincronização, em segundos, foi de 39.77 e 56.81, respectivamente. De forma que os tempos de sincronização não são influenciados pelo número de controladores, mas sim pelas características de sincronização e estabilidade do controlador no ONOS. Enquanto que no ODL o tempo de sincronização é homogêneo, conforme aumenta o número de regras.

Figura 20 – Tempo para sincronização de 2048 regras.



Observando os dados obtidos do ODL na Figura 19, é possível notar que o tempo de sincronização, devido ao aumento de regras, é gradativo. Todas as regras são sincronizadas em todos os controladores, devido ao modelo de consistência forte empregado na sincronização. Considerando a variação na quantidade de regras, o tempo médio para sincronização, em segundos, foi de 2.11, 3.88, 7.35, 14.90, 30.83 e 64.37, respectivamente para 128, 256, 512, 1024, 2048 e 4096 regras. O aumento no número de regras impactou

no tempo de instalação visto que são mais regras à serem instaladas no controlador.

O ODL mostrou-se mais estável em relação ao ONOS para a sincronização de regras nos controladores. É fundamental destacar que o experimento foi realizado em máquinas físicas. O ONOS em sua arquitetura utiliza conexões com múltiplos controladores e com as instâncias do Atomix. A latência de comunicação entre os *hosts* impacta nestes serviços, fazendo com que ocorram sucessivas trocas de papéis entre os controladores (*role_requests*).

3.3.2.4 Tempo para recuperação de falha do master - Mastership Failover

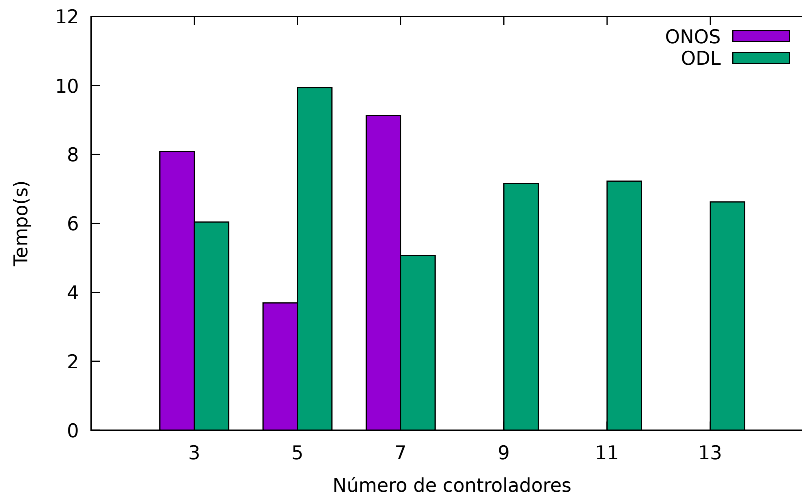
Nos *switches* Openflow em modo distribuído um dos controladores recebe o papel de *master*, sendo responsável pela instalação de regras no *switch*. Quando este controlador falha é necessário que os demais controladores detectem a falha e elejam um novo controlador *master*. Durante esse processo, mesmo que o *switch* esteja conectado à múltiplos controladores, eventos deixam de ser recebidos ou até mesmo processados durante a eleição de um novo *master*.

Neste experimento foi avaliado o tempo para que um novo controlador seja eleito, após induzir a falha do controlador *master*. Para falhar o *master* foi enviado um *SIGKILL* para o processo principal do controlador. A obtenção dos dados deste experimento foi realizada através da captura de tráfego do simulador durante a execução e analisado o último *packet_in* enviado pelo *switch* ao controlador que falhou e o primeiro *flow_mod* recebido pelo *switch* do novo controlador. Desta forma, foi possível observar o último evento enviado ao controlador antes da falha e o primeiro evento processado pelo novo controlador. Com essa abordagem, foi possível obter o tempo necessário para que o plano de controle volte a processar eventos. É importante destacar que após a falha do controlador, nenhum *packet_in* é enviado ao controlador que falhou, pois a conexão Openflow TCP é encerrada. Os resultados obtidos do experimento exibem o valor médio de sincronização de 5 repetições.

A Figura 21 representa os dados obtidos na execução do experimento, é possível observar que tanto para o ONOS quanto para o ODL o aumento no número de controladores não representou um aumento significativo no tempo para definição de um novo *master*. Considerando o cenário com 5 controladores o ONOS levou em média 3,69 segundos entre o último *packet_in* e o primeiro *flow_mod* do novo *master*. Enquanto que o ODL levou 9,93 segundos para definir um novo controlador principal e voltar a processar eventos do plano de dados. No entanto, o ODL com 3, 7, 9, 11 e 13 controladores levou, respectivamente, 6.03, 5.07, 7.15, 7.22, 6.61 segundos. No ONOS o tempo de sincronização com 3 e 7 controladores foi de 8.08 e 9.12. É importante notar que no ONOS, a partir de 9 controladores, não foi possível executar o experimento, visto que não foi eleito um novo *master* para processar eventos em todas as execuções.

Neste experimento o ONOS apresentou um tempo menor do que o ODL para voltar

Figura 21 – Tempo para que novo líder seja eleito nos controladores ONOS e ODL.



à estabilidade somente com 5 controladores. Em relação à estabilidade após a falha de um dos controladores o ODL apresentou melhores resultados.

3.3.2.5 Análise de consistência na instalação de regras perante falha do controlador *master*

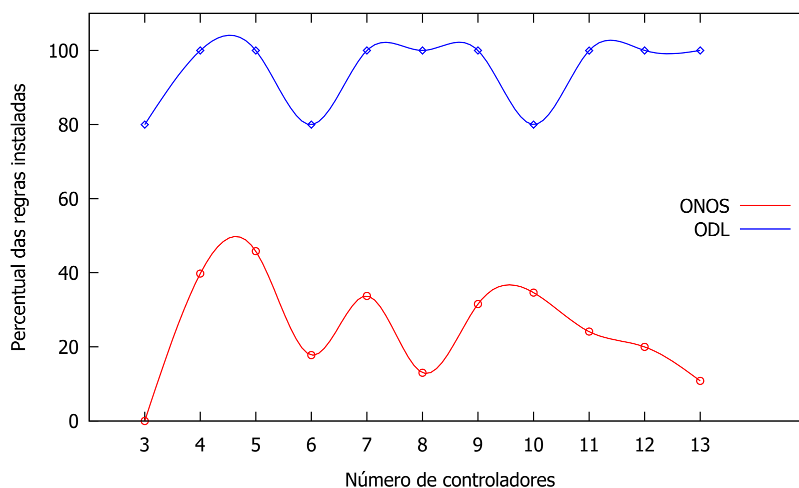
As aplicações de rede utilizam API *northbound* para instalar regras nos *switches* Openflow através dos controladores. No entanto, é possível que o controlador *master*, ou seja, aquele responsável por instalar as regras nos *switches*, falhe. Consequentemente, as aplicações precisam estar cientes de uma falha no *master* e assegurar que as regras solicitadas sejam devidamente instaladas. Desta forma é importante entender como os múltiplos controladores tratam as requisições no caso de falha do controlador master. É importante relembrar que no ONOS, devido ao seu modo de sincronização *lazy*, a aplicação pode receber uma resposta, mas a regra não ser instalada em todos os controladores devido à falha do controlador principal antes de realizar a sincronização.

Para tal, foi realizada a instalação de 1000 regras Openflow através da API *northbound* e logo após a conclusão do envio de todas as requisições foi enviando um *SIGKILL* ao processo principal do controlador, induzindo a uma falha. As regras da tabela de fluxos do *switch* foram removidas. Foi aguardado 30 segundos para um novo *master* assumir e as regras sincronizadas serem instaladas. Espera-se que ao obter um novo *master* as mesmas 1000 regras sejam novamente instaladas. Vale ressaltar que as regras são enviadas à somente um controlador pela API *northbound*. Assim é possível avaliar o estado das regras após a resposta dos controladores.

Na Figura 22 são apresentados os resultados no experimento. Os dados são exibidos na forma de percentual médio das execuções, ou seja, se em média foi sincronizado 100 regras do total de 1000 o valor exibido será 10%. É possível observar que o ONOS não reinstala todas as regras. Enquanto que no ODL o número de regras instaladas pelo novo

controlador é 100% na maioria das execuções.

Figura 22 – Percentual de regras instaladas após falha do controlador principal.



No ODL com 3 e 10 controladores em uma das execuções não houve a eleição de um novo líder para o *switch* após a falha do *master*, fazendo com as regras não fossem reinstaladas. Na configuração com 6 controladores, uma das execuções obteve um novo *master*, no entanto as regras não foram instaladas. De tal forma nas configurações com 3, 6 e 10 controladores foram obtidos o total de 80% das regras instaladas.

No controlador ONOS os resultados mostram o esperado para o modo de sincronização *lazy*, regras serão perdidas. No entanto, nota-se que o período de sincronização padrão dado pela documentação para os *backups* é de 2 segundos, menor do que o tempo necessário para instalação das 1000 regras (próximo de 7 segundos). Espera-se que apenas uma parte das regras instaladas serão perdidas. Observando a Figura 22 o percentual médio de regras instaladas se considerarmos todas as execuções é de 24,66%. A execução com 3 controladores não obteve a sincronização das regras em nenhum dos *backups*, visto que nenhuma regra foi reinstalada após a falha. As execuções com maior número de regras instaladas foram com 4, 5, 7, 9 e 10 controladores, respectivamente com 39,78%, 45.83%, 33.77%, 31.57% e 34.57% do total de regras. É importante destacar que em 56,36% das execuções não foram instaladas regras após a falha do *master*. Em 7,27% das execuções foram reinstaladas todas as regras enviadas ao controlador após a falha. Este resultado deve-se pelo fato de que o controlador *master* foi definido no início do experimento, e durante o envio de regras houve a troca de papéis entre os controladores. Desta forma, o controlador em que a falha foi induzida estava apenas encaminhando as regras para o *master*.

O Quadro 3 apresenta brevemente um resumo dos resultados obtidos dos experimentos realizados no plano de controle. Apesar de mostrarem que o ODL apresenta maior latência ao processar eventos, mostraram também que o ODL é mais estável em relação ao

quadro 3 Resumo dos experimentos realizados no plano de controle.

Experimentos	ONOS	ODL
Vazão e latência através da *southbound*	Menor latência, no entanto, apresentou falhas ao manter o plano de controle estável conforme o número de controladores aumentou.	Maior latência e menor vazão. Estável com o aumento de controladores e permitindo completar todos os experimentos realizados.
Vazão e latência através da *northbound*	Fluxos não são sincronizados entre os controladores antes de responder ao cliente fazendo que a latência seja menor (replicação *lazy*).	Fluxos são mantidos de forma consiste e sincronizados entre todos os controladores antes de responder ao cliente (consistência forte), impactando na latência e vazão do controlador.
Tempo de sincronização	Não sincroniza em todos os controladores, devido ao modo de replicação *lazy*, o tempo para as réplicas estarem consistentes é maior.	Sincroniza em todos os controladores para cada atualização, fazendo com que as réplicas estejam sempre consistentes.
Tempo de eleger um novo *master*	Em média levou menos tempo nos experimentos executados corretamente para eleger um novo *master*, no entanto, não escala bem a partir de 9 controladores.	Maior estabilidade em relação ao ONOS para eleição de um novo *master*, permitindo um número maior de controladores.
Consistência de regras após falha do *master*	Devido ao modelo de consistência fraca, regras são perdidas após falha do controlador. *master*	Regras instaladas não são perdidas, na maioria dos casos, após falha do *master*.

ONOS conforme o número de controladores aumenta. É possível notar que o controlador ODL possui melhor tolerância a falhas do que o ONOS, dado inclusive pelo modelo de consistência adotado entre as réplicas do controlador. Os resultados negativos observados no controlador ONOS devem-se inclusive ao fato observado nos experimentos anteriores, o serviço de *mastership* provido pelo Atomix. Os controladores *backups* não ficam estáveis, impossibilitando a sincronização.

3.4 Trabalhos Relacionados

Nesta seção são apresentados trabalhos que avaliam os controladores destacados sob diversos aspectos, partindo de avaliações em relação aos modelos de consistência até a implementação correta de suas funcionalidades e algoritmos. Os trabalhos destacados apresentam a importância dos controladores SDN escolhidos e as características que devem ser observadas em cada um deles, para que atendem aos requisitos do ambiente em que serão implantados.

Em (BANNOUR; SOUIHI; MELLOUK, 2018a), discute-se os modelos de consistência forte e fraca adotados pelos controladores SDN e é proposto a utilização de um modelo

de consistência adaptável, pois manter a consistência entre os controladores exige a troca de mensagens entre eles, mesmo na ausência de tráfego no plano de dados. Geralmente, para garantir o funcionamento consistente das aplicações em um controlador é utilizado o modelo de consistência forte, que sacrifica a escalabilidade e o desempenho da rede. Com isso, foi apresentado uma solução escalável e que possui bons níveis de desempenho que é a utilização de algoritmos de replicação que utilizam o modelo de consistência fraca, no entanto, foi destacado que esta abordagem possui a desvantagem de que eventos paralelos possam causar inconsistências temporárias na rede. Além disso, foi observado pelos autores que os protocolos utilizados para garantir a consistência entre as réplicas geram, mesmo na ausência de atualizações, tráfego adicional. Desta forma, foi proposto e avaliado experimentalmente um algoritmo para manter a consistência no controlador ONOS para diminuir o tráfego entre controladores e obter níveis aceitáveis de consistência.

Levando em consideração a implementação dos algoritmos de consistência que visam a tolerância a falhas em controladores SDN distribuídos os autores em (VIZARRETA et al., 2020) analisaram os repositórios dos controladores ONOS e ODL para identificar problemas de implementação nos mecanismos de consistência. Os autores realizaram a divisão dos problemas identificados nos repositórios em quatro categorias: defeitos na implementação dos algoritmos distribuídos, escalabilidade e desempenho, alta disponibilidade e problemas operacionais e concluíram que os principais problemas observados estão relacionados com a implementação dos algoritmos distribuídos nos controladores.

Em (HANMER et al., 2018) é analisado a disponibilidade do algoritmo de consenso *Raft*, utilizado no ONOS através do framework Atomix. Além de garantir que os controladores concordem na mesma visão da rede mesmo com a falha da minoria dos nós, também é necessário que seja estável com grande quantidade de tráfego. Foi identificado pelo autor que em momentos com aumento da latência devido ao volume de tráfego, ocorrem sucessivas eleições de um novo *master*. Os autores mostram que o algoritmo *Raft* é (silenciosamente) violado em condições de sobrecarga, estas condições aumentam o tempo de propagação de mensagens, pois as implementações que foram analisadas consideram um atraso fixo para eleição de um novo líder. A implementação do *Raft* no Atomix não garante a execução em tempo real, e em condições de sobrecarga devido o alto uso do CPU a execução pode sofrer atrasos. Com isso a utilização do *Raft*, que deveria garantir a alta disponibilidade, possui o efeito oposto.

Em (SAKIC; KELLERER, 2018) os autores investigaram o tempo de resposta total do sistema ao utilizar o plano de controle distribuído. Os autores utilizaram os controladores ONOS e ODL e analisaram o protocolo de consenso *Raft* utilizado para manter a consistência forte nos controladores. O tempo de resposta do sistema está relacionado a fatores como o número de controladores, a distância dos controladores em relação aos dispositivos no plano de dados, atrasos gerados pelo processamento e possíveis

falhas. No entanto, neste trabalho o tempo de resposta e a disponibilidade são apenas estimados, não sendo realizado experimentos práticos nos controladores.

Em (SOARES et al., 2020) é realizado uma avaliação dos controladores ONOS e ODL sob diferentes óticas, provendo uma classificação qualitativa e quantitativa dos controladores. Os cenários utilizados para a análise quantitativa consideraram a comunicação *out-of-band* (interface dedicada entre controlador e o dispositivo Openflow). Neste modo o ODL apresentou uma taxa menor de *packet-outs*, pois, as regras para encaminhamento são instaladas somente no inicio do experimento. Utilizando o modo de comunicação *in-band* (pacotes de controle através do plano de dados) o ODL se comportou melhor do que o ONOS, entregando todos os pacotes, enquanto no ONOS somente 34.23% dos pacote foram entregues. Em relação à resiliência dos controladores, o ODL apresentou problemas logo no início da execução, enquanto que o ONOS manteve-se estável em quase todos as repetições realizadas pelos autores. Os autores concluíram que o ONOS, em relação à resiliência, atende aos requisitos necessários para a finalidade avaliada devido à sua estabilidade. Mas, não são discutidos os diferentes modo de operação de cada controlador.

Em (SUH et al., 2017) foi avaliado a leitura/escrita de regras Openflow em controladores SDN distribuídos utilizando a API *northbound*. Foi realizado a separação de leitura e escrita no ODL em réplicas do controlador que possuem fragmentos do armazenamento de dados com o papel de *follower* e *master*. Os autores realizaram a avaliação de tempo para recuperação de falhas variando o número de dispositivos e alterando os parâmetros do detector de falhas de ambos os controladores. A conclusão obtida foi de que conforme o número de dispositivos no plano de dados aumenta, o tempo necessário para que um novo *master* seja eleito também aumenta. O tempo de escrita de regras em controladores que possuíam fragmento com o papel de *follower* teve uma uma diminuição considerável no número de regras instaladas. Neste trabalho é possível observar o impacto de operações de escrita em partições com o papel de *follower*, diminuindo o desempenho dos controladores.

Em (KIM; MYUNG; YOO, 2019) os autores analisaram a distribuição de líderes para os fragmentos do armazenamento de dados, visto que nos controladores distribuídos os dados são particionados. Cada partição possui um líder e a forma de distribuição não é homogênea, podendo todos os líderes estarem concentrados em um único controlador. Desta forma, foi proposto um algoritmo com a função de realizar a distribuição justa dos líderes em todos os controladores, aumentando assim a vazão de operações de escrita.

Em (VILCHEZ; SARMIENTO, 2018) é avaliado a tolerância a falhas no plano de dados levando um consideração o tempo de resposta a falhas de enlaces dos controladores ONOS e ODL. Os autores avaliaram o comportamento do plano de dados interrompendo enlaces com base em três topologias para comunicação entre dois dispositivos. Na primeira topologia não havia enlaces que proviam redundância na infraestrutura de rede, causando o interrompimento da comunicação entre os dispositivos durante a falha do enlace. Nas

segunda e terceira topologias foram utilizados um e dois caminhos para prover redundância, respectivamente. No entanto, os controladores avaliados apresentaram diferenças no modo que trataram as falhas em cada uma das topologias. Foi observado pelos autores que o tempo para o ODL restabelecer a comunicação entre os dispositivos, quando o enlace foi reconectado, na topologia sem redundância foi muito maior em relação ao ONOS. Além disso, os autores identificaram que nas topologias que haviam enlaces redundantes o ODL não foi capaz de utilizar estes enlaces para manter a comunicação entre os dispositivos durante a falha do enlace. Desta forma, os autores concluíram que a resiliência do ONOS a falhas no plano de dados era maior e atribuíram o comportamento do ODL ao módulo L2 de encaminhamento de pacotes. No entanto, este trabalho considerou apenas falhas em enlaces e a utilização de tráfego UDP entre os dispositivos.

3.5 Conclusão

Foram destacadas as características de consistência de cada controlador e discutido os algoritmos utilizados em cada um deles e o contexto em que estão inseridos. Conforme visto neste capítulo o ONOS possui os modelos de consistência forte e consistência fraca. No entanto, o ODL possui somente o modelo de consistência forte disponível para as aplicações. Isso permite ao ONOS maior flexibilidade para as aplicações definirem os níveis de consistência a serem utilizados.

Além disso, neste capítulo foi avaliado os controladores ONOS e ODL e o modo que tratam falhas no plano de dados e no plano de controle. Com os resultados preliminares pudemos identificar problemas nos controladores no tratamento de falhas no plano de dados, além de problemas de sincronização e estabilidade do plano de controle.

Em relação a falhas no plano de dados observamos que o controlador ONOS se destacou em relação ao ODL. Ao ocorrer falhas de enlaces, quando possível, o controlador restaurava a conectividade utilizando enlaces alternativos. No entanto, ao observarmos os experimentos no plano de controle o ONOS mostrou-se instável, apesar de possuir maior vazão e menor latência para o processamento de eventos.

No ONOS, o tempo para sincronização de regras entre os controladores foi consideravelmente maior, mesmo para um número reduzido de regras. Além disso, aumentando a quantidade de controladores tornou-se instável. Os resultados obtidos nos experimentos realizados no ODL permitiram observar que com o aumento do número de controladores o plano de controle manteve-se estável. Apesar disso, a latência para instalação de regras foi maior e consequentemente afetou a vazão.

Os controladores distribuídos avaliados utilizam algoritmos de consenso para manter a consistência forte ou técnicas de replicação *lazy*, quando implementado o modelo de consistência fraca. Com base na necessidade de manter a visão da rede consistente

os controladores implementam algoritmos de consenso. No entanto, utilizam um processo/controlador responsável por coordenar a realização do consenso. Desta forma, este processo recebe uma carga de trabalho adicional, causando impacto no desempenho e disponibilidade dos controladores.

Com base nos resultados obtidos e no funcionamento dos controladores surge a questão de diminuir o custo para processamento de eventos nos controladores sem comprometer a estabilidade do plano de controle.

4 Difusão Atômica Sobre o VCube

Os controladores SDN avaliados, ODL e ONOS, utilizam a replicação ativa através de algoritmos de consenso e a replicação passiva (somente no ONOS) através do algoritmo de *primary-backup* para manter a consistência do seu plano de controle distribuído. Esses algoritmos são peças fundamentais e mesmo perante eventos adversos, como falhas, devem garantir a correta sincronização entre os controladores sem impactar no desempenho. Uma desvantagem dos algoritmos apresentados está em delegar a um único processo, chamado de coordenador, a função de coordenar a sincronização de regras ou eventos no plano de controle distribuído. Com isso, foi implementado um algoritmo de difusão atômica hierárquico e sem a presença de um coordenador para manter a consistência no plano de controle distribuído. O algoritmo faz uso de uma topologia virtual, chamada VCube, para disseminar as mensagens e detectar as falhas.

Até o momento, o VCube foi empregado como topologia de algoritmos de difusão de melhor esforço, difusão confiável (RODRIGUES; ARANTES; DUARTE JR., 2014) e também para executar uma instância do Paxos (TERRA; CAMARGO; DUARTE JR., 2020). Ha ainda o trabalho em (POKE; HOEFLER; GLASS, 2017) que propõe um algoritmo de difusão atômica sobre uma topologia virtual de um dígrafo (grafo dirigido). Esses trabalhos serão apresentados na Seção 4.1.3. Este é o primeiro trabalho a implementar e avaliar um algoritmo de difusão atômica sobre o VCube com o objetivo de executar a sincronização entre processos para manter a consistência forte. Em outras palavras, o algoritmo deverá provisionar a entrega confiável e ordenada das mensagens mantendo a consistência forte com a expectativa de não sobrecarregar os processos que executam a sincronização de estados. O algoritmo proposto utiliza a variante de acordo no destino na qual os processos escolhem um número de sequência que acompanha a mensagem para definir a ordem em que cada mensagem será entregue, conforme descrito na Seção 2.1.4. Desta forma, dois processos distintos entregam mensagens na mesma ordem. Este algoritmo nasceu no grupo de pesquisa de sistemas distribuídos do mestrado de ciência da computação da UNIOESTE.

4.1 Definições

Esta seção apresenta o modelo de sistema utilizado para definir o algoritmo de difusão atômica através do VCube, bem como outras definições importantes de sistemas distribuídos. Logo após, o VCube também será descrito. Os trabalhos relacionados são apresentados também nesta subseção.

4.1.1 Modelo de Sistema

O sistema é definido por um conjunto finito P com $n > 1$ processos. A comunicação entre os processos é realizada através de troca de mensagens. A rede é totalmente conectada e cada processo está conectado a todos os demais processos através de enlaces ponto-a-ponto bidirecionais, ou seja, podem enviar e receber mensagens de qualquer processo. Os processos são organizados utilizando uma topologia de hipercubo virtual, chamada de VCube, descrita logo a seguir. Desta forma, os processos se comunicam utilizando a topologia virtual. Na ausência de falhas os processos formam um hipercubo completo. A presença de falhas implica na reorganização da topologia virtual. Os enlaces são confiáveis, assim, mensagens trocadas entre dois processos nunca são perdidas, corrompidas ou duplicadas.

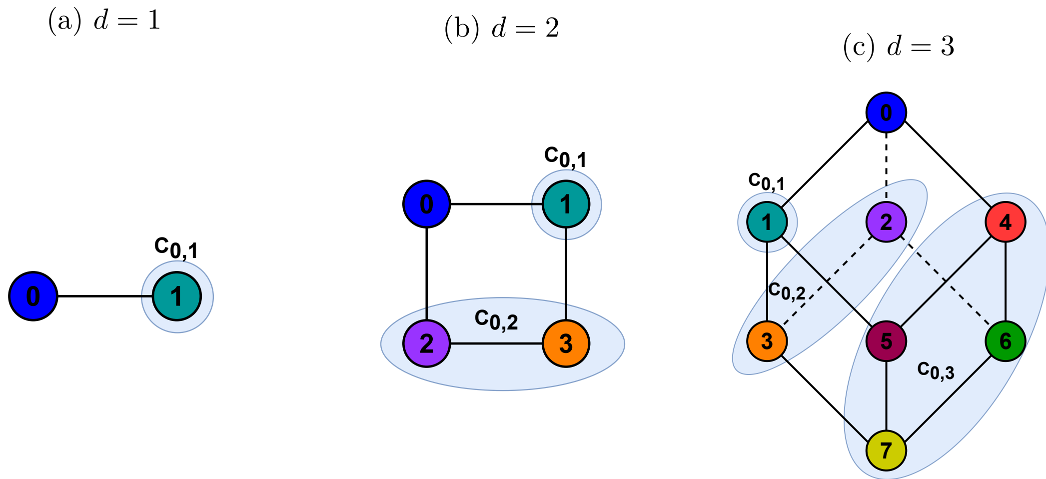
Em um hipercubo de d-dimensões, cada processo é identificado por um endereço binário. Dois processos estão virtualmente conectados se seus endereços binários diferem somente por um bit. Processos podem falhar por *crash*. Ou seja, uma vez que um processo falha, não se recupera. Se um processo nunca falha durante a sua execução, então é considerado correto ou sem-falha; ao contrário o processo é considerado falho. O modelo de sistema é síncrono, ou seja, os limites temporais para processamento e transmissão de mensagens são conhecidos (CHANDRA; TOUEG, 1996).

4.1.2 VCube

O VCube é um algoritmo de diagnóstico distribuído que organiza os processos em grupos progressivamente maiores em uma topologia de hipercubo virtual. Um algoritmo de diagnóstico é utilizado para determinar quais processos de um sistema estão corretos (funcionando de acordo com a especificação) ou falhos (o comportamento diverge da especificação). O VCube permite que os processos utilizem múltiplos caminhos para obter informações de diagnóstico de outros processos. Com isso, utiliza marcações lógicas em cada mensagem para determinar quais informações são mais recentes. O processo i ao testar um grupo de tamanho 2^{s-1}, tal que $s = 1... \log_2 n$, utiliza uma lista ordenada de processos obtida pela função $c_{i,s}$ (Equação 4.1). Os processos no hipercubo são testados em rodadas de 1 até $\log_2 n$.

O número de processos (n) corretos no sistema definem as dimensões que o hipercubo formado pelo VCube irá possuir. Com isso, para obter o número de dimensões é executado a operação $\log_2 n = d$, por exemplo, com 8 processos o hipercubo será organizado em 3 dimensões ($\log_2 8 = 3$). Para ilustrar a formação do VCube de acordo com sua dimensão, na Figura 23 são apresentados hipercubos com 2, 4 e 8 processos, representados, respectivamente, nas Figuras 23a, 23b e 23c. Note que nas figuras a elipse envolvendo os processos representa um agrupamento de processos (*cluster*) de tamanho progressivamente maior. Um *cluster* é representado pela letra s e um processo pela letra i na equação $c_{i,s}$,

Figura 23 – Hipercubo com d-dimensões

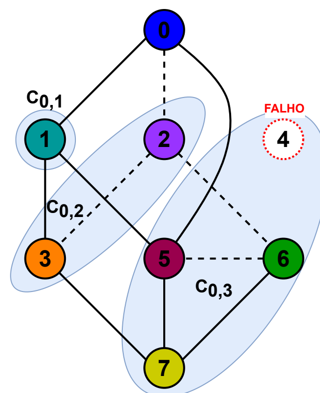


descrita a seguir.

$$c_{i,s} = \left\{ i \oplus 2^{s-1},\ c_{i \oplus 2^{s-1},\,1},\ \dots\ ,\ c_{i \oplus 2^{s-1},\,s-1} \right\} \tag{4.1}$$

Na Equação 4.1, o símbolo $\oplus$ corresponde à operação bit-a-bit ou exclusivo (xor). Inicialmente, para criar a topologia virtual, cada vizinho do processo i no grupo s é computado. Os identificadores de dois processos vizinhos diferem somente em um bit, na representação binária. Os demais processos de cada grupo pertencem aos grupos 1, 2, ..., $s-1$ de cada vizinho do processo i, por exemplo, $c_{i\oplus 2^{s-1},1}$, $c_{i\oplus 2^{s-1},2}$, ... , $c_{i\oplus 2^{s-1},_{s-1}}$. Caso o processo i identifique um vizinho no grupo s como falho será buscado o próximo processo no grupo s, se houver. Por exemplo, ilustrando a reorganização da topologia, na Figura 24 o p_4 está falho, o próximo vizinho de p_0 no grupo $s = 3$ é o p_5.

Figura 24 – Hipercubo com dimensão $d = 3$ e falha do processo 4.



Em um sistema com 8 processos, para obter os vizinhos do processo p_0 (representado pelo número binário 000) no grupo $s = 1$ é alterado o primeiro bit menos significativo,

desta forma, utilizando a Equação 4.1, temos que $c_{0\oplus 2^{1-1},1} = 1$ (001). Identificado o p_1 como vizinho de p_0, nenhum outro processo é adicionado ao grupo $s = 1$, pois p_0 pertence ao grupo $s = 1$ de p_1. O próximo passo é identificar o processo vizinho de p_0 no grupo $s = 2$, desta forma, o segundo bit menos significativo é alterado ($c_{0\oplus 2^{2-1},2} = 2$ (010)), obtendo o p_2 como vizinho. O vizinho de p_2 (010) no grupo $s = 1$ ($c_{2\oplus 2^{1-1},1} = 3$) é o processo p_3 (011) que é adicionado ao grupo $s = 2$ de p_0, finalizando os processos deste grupo. No grupo $s = 3$ o p_4 (100) é vizinho de p_0 ($c_{0\oplus 2^{3-1},3} = 4$). Os demais membros do grupo $s = 3$ de p_0 são $c_{4\oplus 2^{1-1},1} = 5$, $c_{4\oplus 2^{2-1},2} = 6$ e $c_{6\oplus 2^{1-1},1} = 7$, respectivamente os processos p_5, p_6 e p_7. Note que é identificado os vizinhos de p_4 dos grupos 1 e 2, e como p_6 pertence ao grupo $s = 2$ de p_4, é obtido o vizinho de p_6 (110) no grupo $s - 1$ que corresponde a p_7 (111). Assim, na ausência de falhas o hipercubo é construído com 3 dimensões. No Quadro 4 é apresentado os resultados da função $c_{i,s}$ para os demais processos de um hipercubo de 3 dimensões.

quadro 4 Membros de cada grupo lógico da topologia do VCube com 3 dimensões.

s	C0,s	C1,s	C2,s	C3,s	C4,s	C5,s	C6,s	C7,s
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

A seção seguinte descreve algoritmos de difusão e consenso definidos para executar sobre o VCube.

4.1.3 Algoritmos de Difusão e Consenso no VCube

Ao realizar a difusão de mensagens cada algoritmo deve garantir determinadas propriedades para as aplicações em relação à entrega e ordem de cada mensagem. Inicialmente, na difusão por melhor esforço a garantia de entrega das mensagens é feita pela origem. Desta forma, se o processo de origem falhar, não existe garantia de que as mensagens sejam entregues a todos os processos. As propriedades do algoritmo de difusão não confiável são: uma mensagem m é entregue em algum momento se um processo correto realizou o envio de m (validade); nenhuma mensagem é entregue mais do que uma vez (sem duplicação); um processo entrega uma mensagem somente se ela houver sido enviada (*broadcast*) por outro processo de origem, ou seja, nenhuma mensagem é criada (não cria mensagens) (CACHIN; GUERRAOUI; RODRIGUES, 2011). Com isso, temos algoritmos de difusão confiável que estendem o melhor esforço com uma nova propriedade de terminação (*liveness*): Se uma mensagem m é entregue por algum processo correto, então m é entregue, em algum momento, a cada processo correto (acordo) (CACHIN; GUERRAOUI; RODRIGUES, 2011). Desta forma, mesmo com uma falha na origem a mensagem é entregue.

Os primeiros algoritmos de difusão sobre o VCube foram proposto em (RODRIGUES; ARANTES; DUARTE JR., 2014). No trabalho são propostos algoritmos de difusão por melhor esforço (não-confiável) e confiável em um modelo de sistema síncrono. Nestes algoritmos, são utilizados árvores com base na topologia de hipercubo com d-dimensões do VCube para realizar a difusão das mensagens. No caso do algoritmo de difusão confiável, se a origem falhar, os processos que receberam a mensagem devem completar a transmissão para os demais processos. Além disso, ao utilizarem a topologia hierárquica do VCube o número de mensagens enviada por cada processo é menor, fazendo com que ao coletar confirmações de recebimento não haja implosão de *feedback*. Desta forma, cada vértice é responsável por encaminhar as mensagens aos nós filhos e coletar as respostas. As árvores são construídas dinamicamente por cada processo. Com isso, ao ser detectado uma falha cada processo reconstrói a sua árvore com base nos processos corretos. Para detecção de falhas em processos os autores utilizaram as informações providas pelo algoritmo de diagnóstico do VCube.

Em um trabalho posterior (JEANNEAU et al., 2016), os autores usaram uma ideia semelhante para criar um algoritmo hierárquico de difusão confiável para sistemas assíncronos. Nesse modelo de sistema os limites temporais não são conhecidos. O encaminhamento de mensagens é realizado com base na árvore criada, descrita acima. O VCube é utilizado para detecção de falhas, no entanto, devido a possibilidade de mensagens atrasarem por um tempo indefinido um processo pode ser detectado como falho mesmo estando correto (falsa suspeita). Desta forma, o algoritmo considera um detector de falhas imperfeito. Ou seja, o detector de falhas pode cometer enganos. Os processos suspeitos de estarem falhos recebem mensagens especiais. Fazendo com que o processo falho não encaminhe mensagens aos demais processos de sua árvore hierárquica caso esteja correto.

Recentemente, o trabalho de (TERRA; CAMARGO; DUARTE JR., 2020) utilizou a difusão de melhor esforço, definida em (RODRIGUES; ARANTES; DUARTE JR., 2014) para implementar uma instância do algoritmo de consenso Paxos. Ou seja, a disseminação de mensagem ocorre de acordo com a árvore de difusão. Devido à organização do VCube em grupos progressivamente maiores, em (TERRA; CAMARGO; DUARTE JR., 2020) as mensagens são enviadas para o maior grupo provido na topologia virtual, que consiste de $n/2$ processos. Com isso, os autores foram capazes de reduzir o custo para implementação do algoritmo Paxos, reduzindo o número de mensagens trocadas entre os processos, mesmo em cenários com uma quantidade elevada de falhas.

Investigando os ganhos de desempenho obtidos por utilizar uma estratégia de replicação sem líder, os autores em (POKE; HOEFLER; GLASS, 2017) implementaram um algoritmo de difusão atômica baseado em rodadas chamado de *AllConcur*, em que os processos trocam mensagens de forma concorrente. Ao invés de usar o VCube, o algoritmo utiliza um digrafo G (grafo dirigido) como rede sobreposta. É utilizado uma

técnica chamada de *early termination* que minimiza a latência de difusão. Em algoritmos de acordo que utilizam um líder (consenso) os processos enviam atualizações ao líder, responsável por replicar estas atualizações e disseminar aos outros processos participantes. No algoritmo proposto cada processo é tratado igualmente. O funcionamento básico do algoritmo consiste nas seguintes etapas: Em cada rodada do algoritmo cada processo correto realiza a difusão de uma mensagem m (possivelmente vazia); registra as mensagens na rodada utilizando o mecanismo de *early termination* e quando termina de acompanhar as mensagens realiza a entrega das mensagens recebidas na rodada de forma ordenada e determinística. Ao utilizar a abordagem sem líder para realizar a implementação do algoritmo de difusão atômica o desempenho obtido foi 17x maior do que o algoritmo de consenso comparado (Paxos através da *libpaxos*).

O AllConcur considera em sua implementação um modelo de sistema com um detector de falhas $\mathcal{P}$, no entanto, em ambientes práticos sua implementação é custosa. Em redes *Ethernet* o atraso de comunicação entre dispositivos é variável, sendo afetado por congestionamento ou até mesmo falhas em enlaces. Para tal, é discutido que é necessário implementar o AllConcur com um detector de falhas $\diamond\mathcal{P}$ em que a propriedade de precisão do detector é relaxada. Ao relaxar a propriedade de precisão uma notificação de falha não indica necessariamente que um processo esteja falho, pois pode haver uma falsa suspeita. Isso fez com que os autores sugerissem utilizar mensagens adicionais indicando a suspeita de um processo, estas mensagens de falhas ao serem recebidas de todos os processos predecessores na topologia garante que o processo suspeito realmente está falho ou isolado de todos os demais processos no *cluster*. No entanto, os processos isolados ainda podem entregar mensagens fora de ordem, violando a propriedade de *safety* do algoritmo. Com isso, foi adicionado a restrição de que somente a partição com a maioria dos membros é capaz de entregar as mensagens.

Estendendo o trabalho de (POKE; HOEFLER; GLASS, 2017), o algoritmo *AllConcur* foi implementado em (PAZNIKOV; GURIN; KUPRIYANOV, 2020) utilizando *framework akka.net*. Ao utilizar o conceito de atores provido pelo *framework* os aspectos relacionados à implementação de métodos concorrentes são abstraídos, visto que cada ator possui uma "mailbox" que recebe todas as mensagens que são processadas utilizando uma ordem FIFO. Para avaliar a implementação proposta os autores conduziram experimentos em ambientes com memória compartilhada e com memória distribuída. Os autores mostraram que o *AllConcur* pode ser implementado em um ambiente com premissas de sincronia mais relaxada e mesmo usando o detector de falhas padrão, provido pelo próprio *framework*. No entanto, os autores não se preocuparam em comparar a implementação com outros algoritmos, para que fosse obtido uma linha de base para comparação.

4.2 O Algoritmo

Com base nos algoritmos de difusão já implementados utilizando a topologia virtual do VCube, é implementado um algoritmo de difusão atômica. O algoritmo suporta falhas por parada de processos. Inicialmente serão descritas as funções necessárias para criação de árvores de difusão utilizando a topologia de hipercubo gerada pelo VCube, e logo a seguir será apresentado o pseudocódigo do algoritmo de difusão atômica proposto.

4.2.1 Construção da Árvore de Difusão

A construção das árvores de difusão é realizada através de um conjunto de funções criadas em (RODRIGUES; ARANTES; DUARTE JR., 2014). A função $FF_neighbor_i(s) = j$ identifica o primeiro processo correto j ($j \in correct_i$) no *cluster* s do processo i. Por exemplo, para um hipercubo de três dimensões sem-falhas, $FF_neighbor_0(1) = 1$, $FF_neighbor_0(2) = 2$ e $FF_neighbor_0(3) = 4$. Em caso de falha do processo p_2, $FF_neighbor_0(2) = 3$. Se não houverem processos corretos em um grupo, falha de p_1, $FF_neighbor_0(1) = \emptyset$.

A função $neighborhood_i(d)$, definida como

$$neighborhood_i(d) = \{k | k = FF_neighbor_i(s), 1 \leq s \leq d\} \tag{4.2}$$

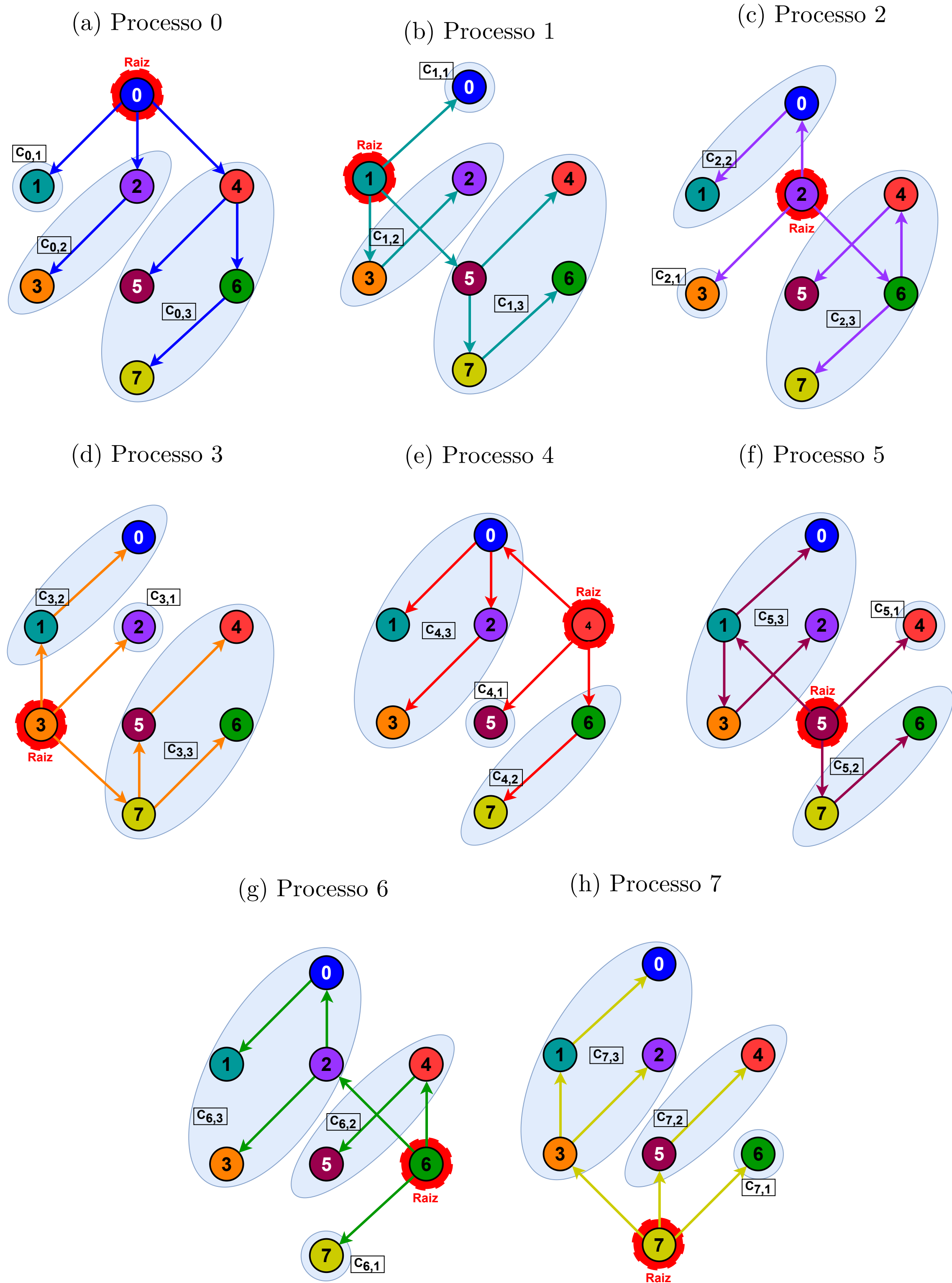
é utilizada para calcular os vizinhos de um processo i considerando o nível d da árvore de difusão. Se i é a raiz, a função retorna todos os vizinhos sem-falha em cada cluster $s = 1..d$, $d = \log_2(n)$ (a dimensão do hipercubo). Se i é raiz de uma sub-árvore do processo fonte que originou m, a função calcula os vizinhos considerando os clusters internos que ele tem conexão no nível d. Por exemplo, para um hipercubo de três dimensões sem falhas com difusão de uma mensagem pelo processo p_0, $neighborhood_0(d = \log_2(n)) = \{1, 2, 4\}$. No processo p_4, que está no nível $d = 2$, $neighborhood_4(2) = \{5, 6\}$. Além disso, caso $d = 0$ indica que o processo não possui vizinhos, considerando a sub-árvore de p_0 ($neighborhood_1(0) = \emptyset$), tornando o processo uma folha na árvore.

A função $cluster_i(j) = s$ é utilizada para determinar o *cluster* s que um processo i pertence em relação a um processo j. Esta informação é fundamental para determinar: a) o nível de i na árvore de j; b) o *cluster* de um processo j detectado como falho, para o qual a mensagem deve ser reencaminhada ao próximo processo correto do grupo. Como exemplo, $cluster_4(0) = 3$. Portanto, existem outros dois níveis na sub-árvore de p_4 que está ligada à p_0.

Definido as funções e dado que i e j são processos distintos, a árvore é construída adicionando o processo i como raiz. Os filhos de i são obtidos através da função $neighborhood_i(d)$. Desta forma, cada processo j, vizinho de i, é adicionado como filho na árvore. Para cada processo j, são obtidos os vizinhos k de j através da função

$neighborhood_j(cluster_i(j) - 1)$ e adicionados à arvore. O passo anterior é repetido até que todos os processos estejam na árvore. As setas na Figura 25 representam as árvores de difusão geradas por este algoritmo.

Figura 25 – Árvores de Difusão para um conjunto de 8 processos.



Para exemplificar a construção da árvore de cada processo considere p_0 como

raiz em um hipercubo de três dimensões. Inicialmente, o processo p_0 é adicionado como raiz de sua árvore. Os filhos são adicionados através da função $neighborhood_i(d)$, onde d é a dimensão do hipercubo. Nesse caso, $neighborhood_0(3) = \{1, 2, 4\}$ que retorna os processos p_1, p_2 e p_4. Para cada processo adicionado como filho do processo p_0, é chamada a função $neighborhood_j(cluster_i(j) - 1)$, que retorna os vizinhos do processo j em relação ao grupo que j pertence do processo i. Desta forma, para o processo p_1 em relação a p_0, $neighborhood_1(cluster_0(1) - 1) = \emptyset$, visto que p_1 pertence ao *cluster* 1, o *cluster* anterior (ou seja, -1 da equação), será o *cluster* de tamanho 0. Com isso, o processo p_1 é adicionado como folha na árvore de p_0.

No processo p_2, é chamada a função $neighborhood_2(cluster_0(2) - 1) = 3$, com isso o processo p_3 é adicionado como filho de p_2. No processo p_3, é chamado a mesma função trocando o valor de i, que corresponde ao processo pai, $neighborhood_3(cluster_2(1) - 1) = \emptyset$. Voltando ao processo p_4, é executada a função $neighborhood_4(cluster_0(4) - 1) = \{5, 6\}$, de forma que os processos p_5 e p_6 são adicionados como filhos de p_4. No processo p_5, é chamada a função $neighborhood_5(cluster_4(5) - 1) = \emptyset$ e identificado que não possui processos filhos. No processo p_6 é chamada a função $neighborhood_6(cluster_4(6) - 1) = \{7\}$, assim o processo p_7 é adicionado como filho de p_6. Em p_7, a função $neighborhood_7(cluster_6(7) - 1) = \emptyset$ não retorna nenhum processo. Desta forma, todos os processos foram adicionados na árvore. Na Figura 25a, as setas indicam a árvore construída neste exemplo.

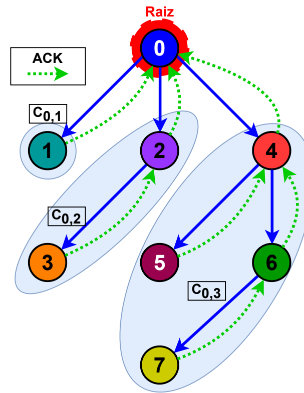
4.2.2 Algoritmo de Difusão Atômica

O algoritmo de difusão atômica implementado neste trabalho transmite suas mensagens através da topologia virtual do VCube. Cada processo, ao difundir uma mensagem m, utiliza uma árvore geradora com raiz em si mesmo para encaminhar um contador lógico (*timestamp*) com a finalidade de ordenar a entrega das mensagens. As árvores são dinamicamente construídas e autonomicamente mantidas utilizando a estrutura hierárquica dos grupos e o conhecimento de processos falhos. Uma vez que um processo detecta a falha de outro processo, sua árvore é reconfigurada sem a necessidade de mensagens adicionais.

Em linhas gerais o algoritmo funciona da seguinte forma: processo de origem (fonte) difunde a mensagem para seus vizinhos (filhos). Ao receber a mensagem, cada processo se torna a raiz de uma sub-árvore e a encaminha para seus próprios grupos considerando a árvore geradora da origem. Depois que uma mensagem é propagada com sucesso em suas sub-árvores, o processo envia uma mensagem de confirmação (ACK) para o processo pai, ou seja, o processo do qual a mensagem foi recebida (Figura 26). Desta forma, um processo envia um ACK somente se recebeu confirmações de todos os seus processos filhos. Assim, ao receber ACK de um processo filho, é possível afirmar que todos os processos corretos daquele *cluster* também receberam a mensagem. Além disso, cada processo, ao

receber uma mensagem pela primeira vez, atualiza o *timestamp* local e o replica em uma árvore com raiz nele próprio. Uma mensagem só é entregue ao receber o *timestamp* de todos os processos corretos para a mensagem propagada.

Figura 26 – Confirmação de recebimento ($ACKs$) na árvore de p_0.



O Algoritmo 1 apresenta o pseudocódigo da solução. As variáveis locais mantidas pelos processos são:

- LC_i: relógio lógico local que identifica unicamente as mensagens enviadas por i. É usado para controlar quais mensagens já foram entregues de um determinado processo fonte;
- TS_i: *timestamp* utilizado para a ordem total;
- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida e entregue de cada processo fonte em ordem;
- $pending_ack_i$: o conjunto de ACKs pendentes no processo i. Para cada mensagem $TREE\langle m,T\rangle$ (re)-transmitida por i de um processo j para um processo k, um elemento $\langle j,k,\langle m,T\rangle\rangle$ é adicionado a este conjunto;
- $received_i$: conjunto de mensagens $\langle m, ts_k(m)\rangle$ recebidas pelo processo i dos processos k que ainda não podem ser entregues à aplicação;
- $stamped_i$: conjunto de mensagens m para as quais já foram recebidos todos os *timestamps* $\langle m, ts_k(m)\rangle$, mas que ainda não podem ser entregues porque estão fora de ordem em relação às demais mensagens $m' \neq m$.

As mensagens utilizadas no algoritmo são:

- $TREE\langle m,T\rangle$, que contém a informação m sendo propagada e o conjunto T de *timestamps* $ts(m)$. Sendo que o conjunto T possui 1 até $log_2 n$ *timestamps*. Para identificar para qual processo k um *timestamp* pertence, $ts_k(m)$ pode ser utilizado;

Algorithm 1 Difusão Atômica no Processo i

1: **procedure** INITIALIZATION( )
2: $LC_i \leftarrow TS_i \leftarrow 0$
3: $correct_i \leftarrow \{0, .., n-1\}$
4: $last_i[n] \leftarrow \{\perp, .., \perp\}$
5: $pending_ack_i \leftarrow \emptyset$
6: $received_i \leftarrow stamped_i \leftarrow \emptyset$

7: **procedure** A-BROADCAST(m)
8: $m.src \leftarrow i$; $m.seq \leftarrow LC_i$
9: $ts_i(m) \leftarrow TS_i$
10: $LC_i \leftarrow LC_i + 1$
11: $TS_i \leftarrow$ MAX(TS_i, LC_i)
12: $received_i \leftarrow received_i \cup \{\langle m, ts_i(m)\rangle\}$
13: FORWARD($\log_2 n$, $TREE\langle m, \{ts_i(m)\}\rangle$)

14: **procedure** FORWARD(s, $TREE\langle m, T\rangle$)
15: **for all** $p \in neighborhood_i(s)$ **do**
16: SEND($TREE\langle m, T\rangle$) **to** p
17: $pending_ack_i \leftarrow pending_ack_i \cup \{\langle from, p, \langle m, T\rangle\rangle\}$

18: **upon** `receive` $TREE\langle m, T\rangle$ **from** p_j
19: **if** $j \notin correct_i$ **then**
20: RETURN
21: $TS_i \leftarrow \max(\forall ts_k(m) \in T, TS_i + 1)$
22: **if** $m.seq > last_i[m.src]$ **and** $m \notin \{received_i \cup stamped_i\}$ **then**
23: $ts_i(m) \leftarrow TS_i$
24: $T \leftarrow T \cup \{ts_i(m)\}$
25: **for all** $p \in neighborhood_i(\log_2 n) \setminus neighborhood_i(cluster_i(j) - 1)$ **do**
26: SEND($TREE\langle m, \{ts_i(m)\}\rangle$) **to** p
27: $pending_ack_i \leftarrow pending_ack_i \cup \{\langle i, p, \langle m, ts_i(m)\rangle\rangle\}$
28: **for all** $ts_k(m) \in T$ **do**
29: $received_i \leftarrow received_i \cup \{\langle m, ts_k(m)\rangle\}$
30: FORWARD($cluster_i(j) - 1$, $TREE\langle m, T\rangle$)
31: CHECKDELIVERABLE(m)
32: CHECKACKS(j, $\langle m, T\rangle$)

33: **upon** `receive` $ACK\langle m, T\rangle$ **from** p_j
34: $p \leftarrow x : \langle x, j, \langle m, T\rangle\rangle \in pending_ack_i$
35: $pending_ack_i \leftarrow pending_ack_i \setminus \langle p, j, \langle m, T\rangle\rangle$
36: CHECKDELIVERABLE(m)
37: CHECKACKS(p, $\langle m, T\rangle$)

38: **procedure** CHECKDELIVERABLE(m)
39: **if** recebeu $\langle m, ts_k(m)\rangle$ de todo $k \in correct_i$ **and** $pending_ack_i \cap \langle *, *, \langle m, *\rangle\rangle = \emptyset$ **then**
40: $sn(m) \leftarrow \max(ts_k(m) \mid \langle m, ts_k(m)\rangle \in received_i)$
41: DELIVER(m, $sn(m)$)
42: **for all** $\langle m' = m, ts(m)\rangle \in sorted(received_i)$ **do**
43: **if** $m'.seq = last_i[m.src] + 1$ **then**
44: $last_i[m.src] \leftarrow m'$
45: $received_i \leftarrow received_i \setminus \{\langle m', *\rangle\}$

46: **procedure** DELIVER(m, $sn(m)$)
47: $stamped_i \leftarrow stamped_i \cup \{\langle m, sn(m)\rangle\}$
48: $deliverable \leftarrow \emptyset$
49: **for all** $\langle m', sn(m')\rangle \in stamped_i \mid \forall \langle m'', ts(m'')\rangle \in received_i : sn(m') < ts(m'')$ **do**
50: $deliverable \leftarrow deliverable \cup \{\langle m', sn(m')\rangle\}$
51: Entrega todas as mensagens em *deliverable* na ordem $(sn(m), m.src)$
52: $stamped_i \leftarrow stamped_i \setminus deliverable$

53: **procedure** CHECKACKS(p, $\langle m, T\rangle$)
54: **if** $pending_ack_i \cap \langle p, *, \langle m, T\rangle\rangle = \emptyset$ **then**
55: **if** $m.src \neq i$ **and** $\{p\} \in correct_i$ **then**
56: SEND($ACK\langle m, T\rangle$) **to** p

57: **upon notifying** `crash`(j)
58: $correct_i \leftarrow correct_i \setminus \{j\}$
59: **for all** $p = x, m = y : \langle x, j, \langle y, T\rangle\rangle \in pending_ack_i$ **do**
60: **if** $\{p\} \in correct_i$ **then**
61: **if** $k = FF_neighbor_i(cluster_i(j)) \neq \emptyset$ **then**
62: SEND($TREE\langle m, T\rangle$) **to** p
63: $pending_ack_i \leftarrow pending_ack_i \cup \langle p, k, \langle m, T\rangle\rangle$
64: $pending_ack_i \leftarrow pending_ack_i \setminus \langle p, j, \langle m, T\rangle\rangle$
65: CHECKACKS(p, $\langle m, T\rangle$)
66: **for all** $\langle m, *\rangle \in received_i$ **do**
67: CHECKDELIVERABLE(m)

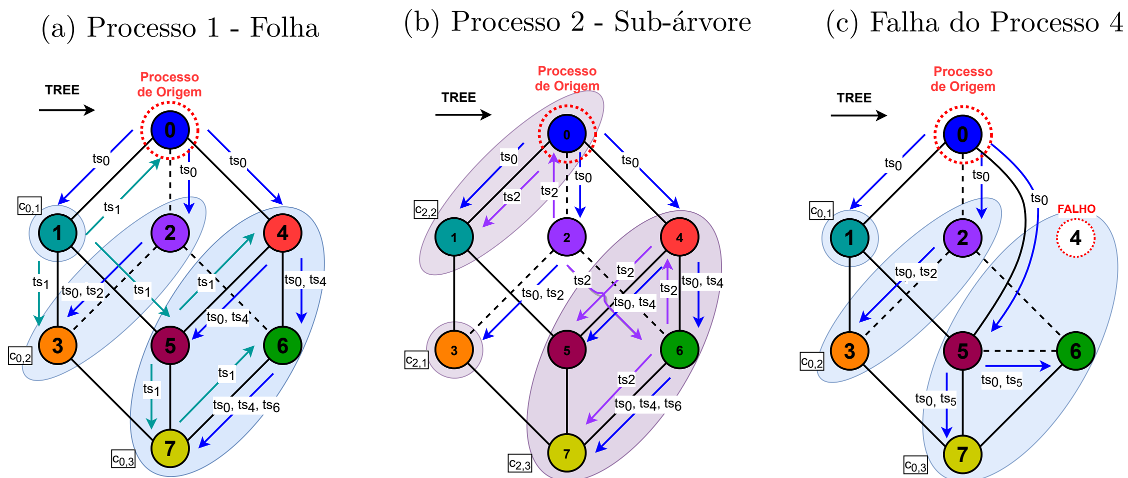
- $ACK\langle m, T\rangle$, para confirmar a recepção de uma mensagem $TREE\langle m, T\rangle$.

Toda mensagem m possui dois parâmetros: (i) o identificar do processo de origem $m.src$; e (ii) um número de sequência $m.seq$, obtido do relógio lógico local LC_i do processo de origem i, utilizado para identificar, de maneira única e sequencial, as mensagens enviadas por cada processo.

A difusão é iniciada com a função A-Broadcast(m) (linha 7), que adiciona à mensagem o *timestamp* $ts_i(m)$ do processo de origem. A função Forward é utilizada para encaminhar $TREE$ aos vizinhos do processo i. Os parâmetros da função Forward são o número de *clusters* s de i que a mensagem deverá ser encaminhada e a mensagem contendo o $ts_i(m)$. Ao iniciar a difusão, o processo i utiliza $s = \log_2 n$, que representa a dimensão do hipercubo e corresponde ao número de *clusters* que cada processo possui. Na função Forward a mensagem m é encaminhada aos processos vizinhos de i com base no valor de s (linha 15).

Ao receber a mensagem $TREE$, i verifica se j está na lista de processos corretos (linha 19). Se estiver falho seu *timestamp* deve ser desconsiderado. Se j estiver correto o processo i atualiza seu *timestamp* local TS_i com base nos *timestamps* recebidos do processo j (linha 21). Ao verificar que a mensagem foi recebida pela primeira vez, ou seja, não foi entregue e não está no conjunto de mensagens recebidas ou marcadas, a mensagem é replicada na árvore do processo i com $T = ts_i(m)$, aos processos que não sobrepõem a árvore do processo j (linha 25). Para os processos que sobrepõem a árvore de j a mensagem m é encaminhada com *timestamp* T (linha 30), tal que se m foi recebida pela primeira vez, $ts_i(m)$ é agregado a T (linha 24).

Figura 27 – Encaminhamento de mensagens da origem.



Considerando um hipercubo de três dimensões, a Figura 27 apresenta o esquema de difusão de mensagens do processo p_0 em sua árvore (ver seta azul escura) e o encaminhamento da mensagem m aos vizinhos dos *clusters* 1, 2 e 3. A Figura 27a destaca

o comportamento do processo p_1 ao receber a mensagem m pela primeira vez. O processo p_1, identifica que está no último nível da árvore de p_0 (folha), através da função $\{cluster_1(0) - 1 = 0\}$. Desta forma, p_1 apenas encaminha seu *timestamp* aos processos $\{0, 3, 5\}$ (representado por ts_1 nas setas). A Figura 27b destaca o processo p_2, que é um ramo na árvore de p_0 ($cluster_2(0) - 1 = 1$). Com isso, encaminha m com seu *timestamp* aos processos $\{0, 6\}$. Na árvore do processo p_0, p_2 agrega seu próprio *timestamp* à m e encaminha ao processo p_3. Assim m passa a conter os *timestamps* de p_0 e p_2.

Ao receber a mensagem $TREE$ e encaminhar as mensagens aos próximos processos, cada *timestamp* contido em T é adicionado à lista de *timestamps* recebidos de m (linha 29) e é chamada a função CheckDeliverable(m). Esta função verifica se os *timestamps* de todos os processos corretos do sistema foram recebidos para m (linha 39). Caso seja tenham sido recebidos, é obtido o maior *timestamp* para m (linha 40), representado por $sn(m)$ e chamada a função Deliver($m, sn(m)$) (linha 41). Em seguida, atualiza-se o registro $last_i$ de mensagens do processo de origem da mensagem e remove-se todos os *timestamps* associados à m da lista de mensagens recebidas $received_i$.

Na função Deliver, a mensagem m é adicionada à lista de mensagens marcadas ($stamped_i$) com o maior *timestamp* recebido $sn(m)$ (linha 47). A mensagem é adicionada para entrega se, e somente se, para toda mensagem m' em $stamped_i$ não existir mensagem m'' recebida ($received_i$) com *timestamp* menor que m'. Caso contrário m' é entregue somente após m'' (linha 49). As mensagens marcadas para entrega (*deliverable*) são ordenadas pelo *timestamp* associado à m e o processo de origem de m ($m.src$) (linha 51). Desta forma, as mensagens são entregues ordenadas e removidas de $stamped_i$.

A confirmação de recebimento de mensagens é realizada através da função CheckAcks. A tupla $\langle p, *, \langle m, T \rangle \rangle$ expressa os ACKs que estão sendo aguardados para a mensagem m com *timestamp* T, recebidas de p e encaminhadas a qualquer processo ($*$) (linha 54). Por exemplo, na Figura 27a, p_1 é uma folha da árvore de p_0, portanto, não encaminhará nenhuma mensagem com o *timestamp* de p_0, o que permite o envio imediado do ACK para p_0. O mesmo ocorre com p_3 em relação à p_2. No entanto, p_2, ao receber o ACK de p_3, remove a mensagem pendente de sua lista e obtém o processo p que inicialmente enviou a mensagem com *timestamp* T. Com isso, ao executar CheckAcks e verificado que não existe mensagem pendente de p_0, que deveria ter sido encaminhada por p_2, é enviado um ACK para p_0.

A detecção e notificação de falhas em processos é realizada pelo algoritmo de diagnóstico do VCube. Para todos os processos, quando i recebe a notificação de que j está falho (linha 58), j é removido da lista de processos corretos. Com isso, para cada mensagem m encaminhada a j não confirmada, ou seja, existe m enviada a j em $pending_acks_i$ (linha 59), estas mensagens devem ser reencaminhadas ao próximo processo correto, se houver, no *cluster* que j pertence (a função $FF_neighbor_i(s)$ determina esta informação,

sendo $s = cluster_i(j)$) (linha 61). Por exemplo, na Figura 27c p_4 está falho e p_0 é a origem da mensagem em um hipercubo de três dimensões, se p_4 falhar, durante a difusão de uma mensagem, p_0 não receberá a confirmação de recebimento do grupo três e, ao ser detectada a falha, encaminhará seu *timestamp* ao próximo processo correto deste grupo (p_5). Se não houver mais processos corretos naquele *cluster*, a função CheckAcks é chamada para verificar se existe alguma mensagem pendente a ser encaminhada do processo p que enviou m ao processo i (linha 65). Além disso, o processo i pode não ter recebido o *timestamp* de j, logo ficará aguardando até que seja recebido. No entanto, j foi detectado como falho. Portanto, para cada mensagem m em $received_i$ é chamada a função CheckDeliverable (linha 67) para verificar se m pode ser entregue, visto que o *timestamp* de j não é mais necessário.

4.3 Resultados

Nesta Seção são discutidos os resultados obtidos pela execução do algoritmo em dois ambientes distintos. No primeiro ambiente, através do simulador Neko (URBAN; DEFAGO; SCHIPER, 2001), é avaliada a implementação do algoritmo utilizando duas estratégias de comunicação: através de uma topologia virtual hierárquica em relação à comunicação tradicional, em que cada processo se comunica com todos os demais. No segundo ambiente, o algoritmo *AnyABCast* foi implementado utilizando o framework Akka (LIGHTBEND, 2020) em um ambiente real. Para tal, foram utilizadas 16 máquinas com 16GB de memória e processador AMD A8-5500B, com 4 *threads* de processamento, as máquinas estão interligadas com interfaces de 1*Gbps* através de um *switch*. Os experimentos avaliam o desempenho do algoritmo implementado em relação ao algoritmo de consenso *Raft*, afim de obter um paralelo de desempenho entre estes dois algoritmos e avaliar se o *AnyABCast* pode ser uma alternativa viável para redes definidas por software.

4.3.1 Ambiente Simulado

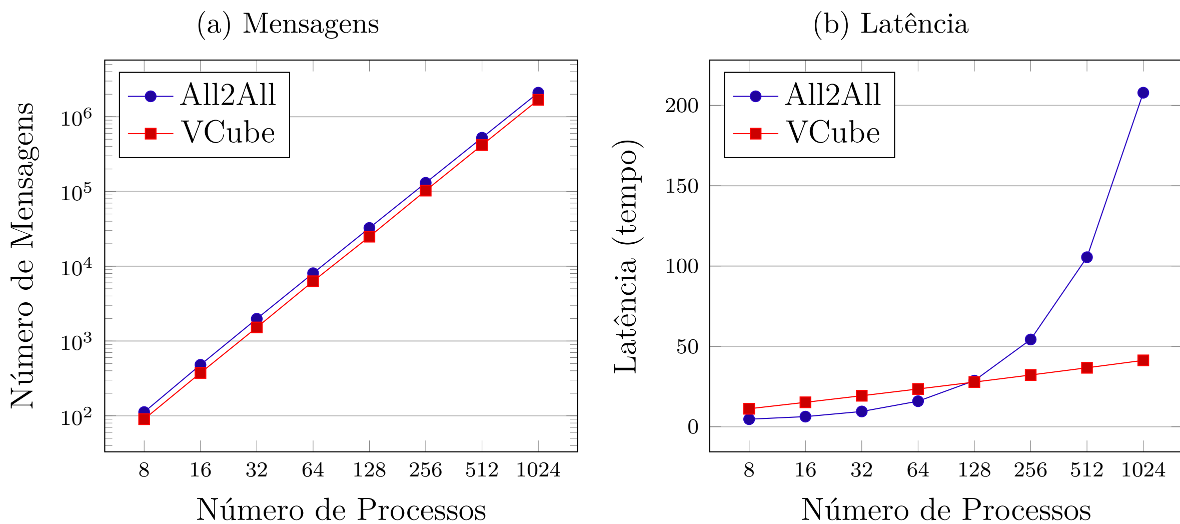
A estratégia hierárquica de difusão de mensagens foi comparada com uma estratégia todos-para-todos (chamada *All2All*). Na estratégia todos-para-todos, cada processo envia o *timestamp* (ts) para a mensagem recebida a $n - 1$ processos, assim cada processo comunica-se diretamente com os demais processos. Foram considerados diferentes cenários e tamanhos de sistema. Em cada experimento, o número de processos n variou de 8 a 1024 em potência de 2. Ao utilizar a estratégia All2All o detector de falhas se comunica diretamente com os processos e, portanto, possui menor latência para detecção de falhas. A estratégia de difusão hierárquica de mensagens conta com a estrutura do próprio VCube, que proporciona maior escalabilidade ao custo de aumentar a latência para detecção de falhas. O número de mensagens corresponde a todas as mensagens trocadas entre

os processos, incluindo as confirmações de entrega (*ACKs*). A latência corresponde à t unidades de tempo para que uma mensagem seja entregue a todos os processos corretos.

Nos experimentos, considera-se que cada processo envia somente uma mensagem por vez com um atraso t_s. Ainda um processo leva t_r unidades de tempo para receber uma mensagem. Se uma mensagem é enviada para mais do que um processo, t_s deve ser computado para cada cópia da mensagem a ser enviada. O tempo de transmissão na rede de uma mensagem é dado por t_t. Desta forma, definimos que $t_r = t_s = 0.1$ e $t = 0.8$ unidades de tempo para todas as execuções. O intervalo de teste do detector em cada experimento foi definido para 30.0 unidades de tempo. Um processo é considerado falho se o detector não obtiver a resposta em $4 * (t_s + t_t + t_s)$[1] unidades de tempo. A seguir, são apresentados os resultados dos experimentos sem falhas e com falhas de processos.

4.3.1.1 Experimentos sem processos falhos

Figura 28 – Execução sem processos falhos.



A Figura 28 apresenta o número de mensagens e a latência do envio de uma única mensagem pelo processo p_0. Na estratégia hierárquica, ao enviar a mensagem e seu *timestamp*, o caminho mais longo em uma sub-árvore gerada no hipercubo é dado por $\log_2 n$. Desta forma, no pior caso, um processo leva $2log_2n$ para encaminhar seu *timestamp*, visto que é necessário o recebimento de *ACKs* para garantir que o *timestamp* e a mensagem foram recebidos. A entrega de uma mensagem é dada pelo tempo que o último processo da maior sub-árvore de p_0 leva para encaminhar seu *timestamp*. Na estratégia *All2All*, a latência é dada pelo tempo que um processo leva para encaminhar seu *timestamp* aos demais processos após receber uma mensagem. Com isso, para poucos processos a latência da estratégia hierárquica é maior. No entanto, quando o número de processos é maior ou igual a 128, a estratégia hierárquica apresenta menor latência (ver Figura 28b). Desta

[1] Este valor representa o dobro do tempo para o envio até a recepção de uma mensagem.

forma, quanto maior o número de processos maior será o tempo para o envio das mensagens. A latência na difusão hierárquica utilizando o hipercubo mostrou-se linear. Já a latência da estratégia *All2All* apresentou comportamento exponencial.

A ordem total das mensagens é garantida com o recebimento do *timestamp* de todos os processos para a mensagem m. Ao utilizar a estratégia *All2All* todos os processos enviam diretamente o seu *timestamp* aos demais processos. Com a estratégia hierárquica, é possível que uma mensagem agregue mais do que um *timestamp*. Desta forma, o número de mensagens no algoritmo de difusão atômica proposto foi em média 21.45% menor (Figura 28a).

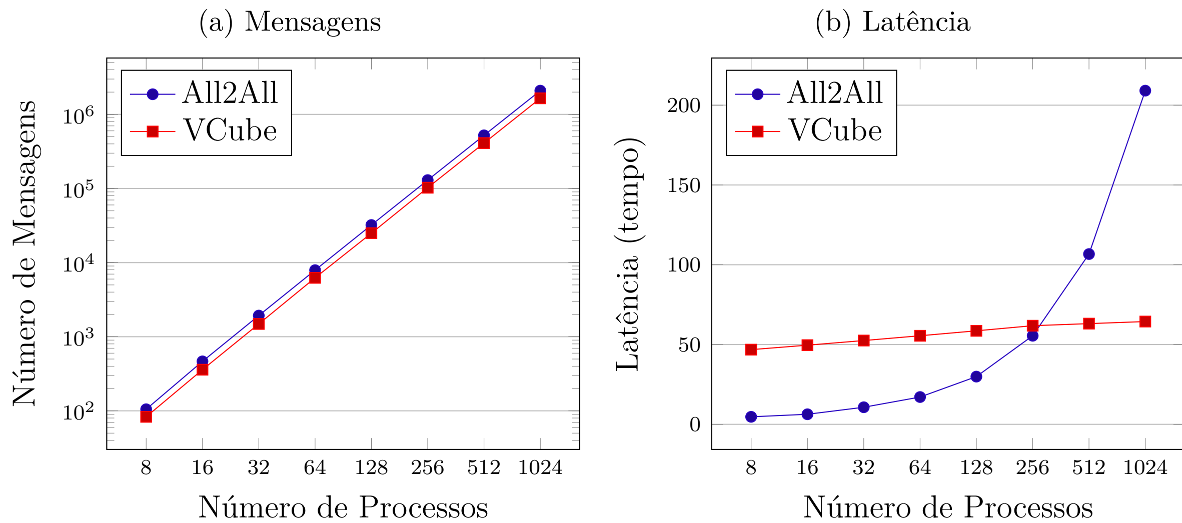
4.3.1.2 Experimentos com processos falhos

A seguir são apresentados resultados para a latência e número de mensagens a partir de três cenários distintos de falhas de processos. O primeiro cenário apresenta a falha do processo de origem da mensagem. O segundo cenário descreve o impacto da falha de um processo qualquer do sistema, antes de ter sido detectado como falho. Por fim, no terceiro cenário o algoritmo de difusão é executado após a falha de um processo ser detectada por todos os demais processos do sistema.

Execução com falha da origem. Considera-se que ao menos um processo correto recebeu a mensagem antes do emissor falhar, conforme a propriedade de terminação (*liveness*) herdada da difusão confiável. No algoritmo de difusão atômica proposto a falha da origem implica em todos os processos que receberam a mensagem desconsiderarem o *timestamp* do processo falho. Além disso, como o algoritmo utiliza difusão hierárquica de mensagens, os processos que não recebem a confirmação de recebimento das mensagens enviadas ao processo falho, devem reencaminhar a mesma mensagem ao próximo processo correto no *cluster* do processo falho, se houver. Para execução do experimento considerou-se a origem p_0 enviando mensagens no tempo 0 e falhando logo em seguida.

Conforme apresenta a Figura 29a, houve pequena variação no número de mensagens, pois a falha de um processo livra-o de obter os *timestamps*. Desta forma, o número de mensagens da abordagem hierárquica foi 21.74% menor em relação à topologia *All2All*, resultado semelhante à execução sem falhas. Já a latência para entrega de mensagens, ilustrada na Figura 29b, apresenta uma diferença significativa na presença de falhas. O algoritmo de diagnóstico do VCube executa em rodadas. Assim, para todos os processos corretos identificarem uma falha é necessário a execução de mais do que uma rodada do algoritmo. Desta forma, o intervalo para execução de rodadas do detector de falhas, definido em 30 unidades de tempo, impacta diretamente na latência para entrega de mensagens. A diferença de tempo para entrega de mensagens entre a abordagem hierárquica sem falha e na presença de uma falha é, em média, 29.63 unidades de tempo. Em relação à abordagem *All2All*, a partir de 512 processos apresentou maior latência para entrega de mensagens,

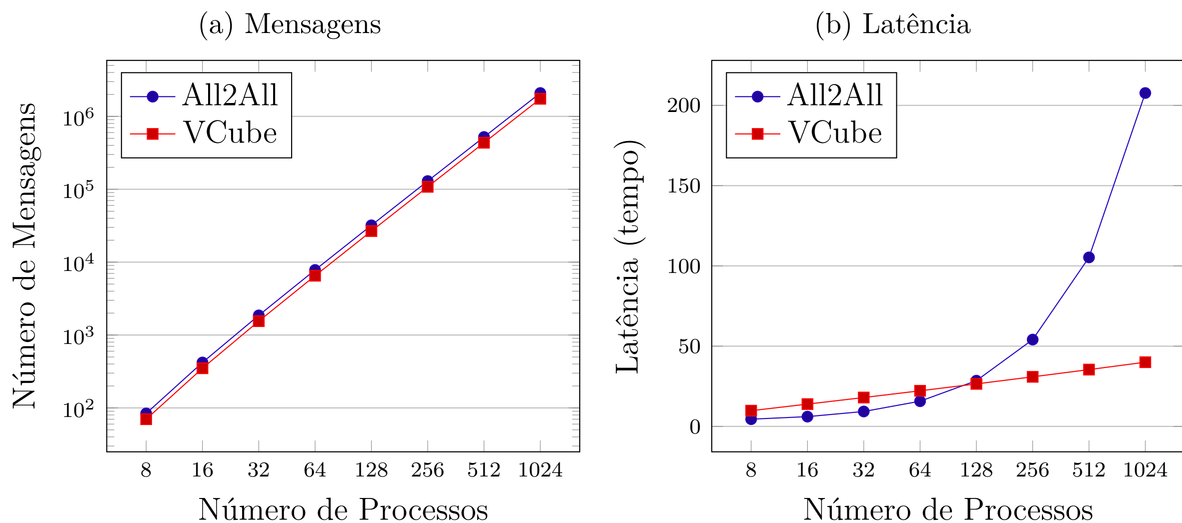
Figura 29 – Execução com falha do processo de origem.



visto que o tempo de envio e recebimento (t_s e t_r) das mensagens é mais significativo quando o número de processos é maior.

Execução com envio antes de processo ser detectado como falho. Considera-se a falha do processo p_1 durante a difusão de uma mensagem m por p_0. Um processo ao encaminhar m à um processo falho, nunca receberá a confirmação de recebimento deste processo e acabará por detectá-lo como falho. A falha de um processo qualquer, assim como a falha da origem, implica na remoção do *timestamp* do processo falho na variável $received_i$ e o reencaminhamento das mensagens com *ACKs* pendentes ao próximo processo correto do grupo, se houver. Desta forma, a latência e o número de mensagens para que uma mensagem m seja entregue é idêntica à execução com falha na origem.

Figura 30 – Execução com *broadcast* depois de falha em um processo ser detectada.



Execução com envio após detecção de processo falho. Neste experimento o processo p_1 falha no tempo 0. Somente após todos os processos detectarem a falha de p_1 a mensagem

é difundida pelo origem (processo p_0). Com isso, o algoritmo se reorganiza para que o processo falho seja removido da topologia. Os resultados obtidos são apresentados na Figura 30 e apontam que na topologia *All2All* o número de mensagem foi menor em relação a execução sem falhas, visto que existem menos processos ativos. No entanto, a abordagem hierárquica apresenta menor número de mensagens em relação à abordagem *All2All* (Figura 30a).

Devido a redução no número de processos corretos no sistema, a latência para entrega de mensagens é menor. Na abordagem *All2All* a latência é reduzida em 0.2 unidades de tempo para todas as execuções. Na abordagem hierárquica a latência para entrega é reduzida em 1.3 unidades de tempo. Assim como na execução sem falhas, a partir de 128 processos a latência utilizando a abordagem hierárquica é menor do que a latência utilizando a topologia *All2All*.

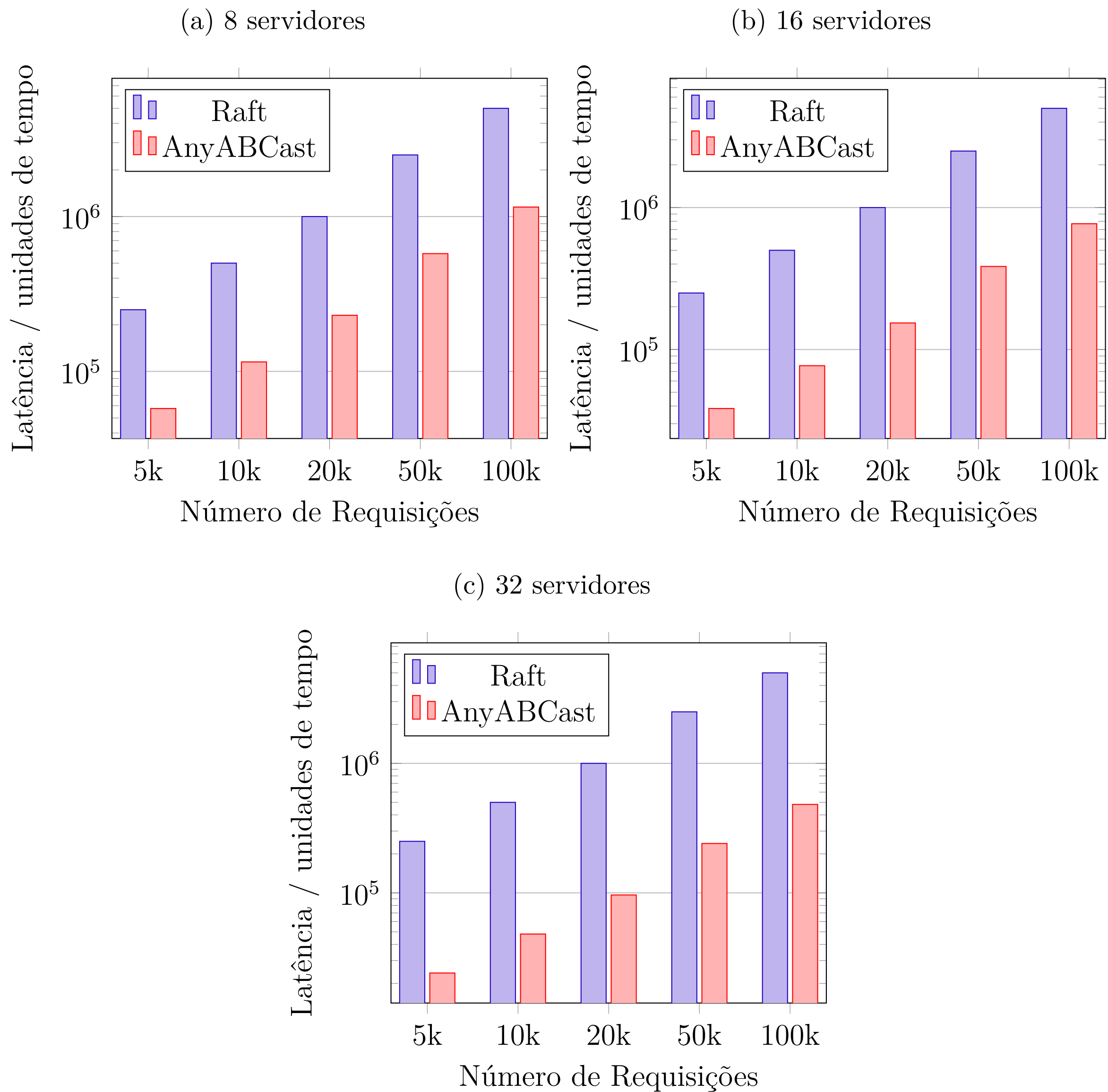
4.3.1.3 Comparação entre os algoritmos Raft e AnyABCast

Considerando a importância de conhecer o desempenho do algoritmo proposto frente ao algoritmo empregado para manter consistência forte nos controladores, foi realizada a simulação do algoritmo proposto com o algoritmo de consenso *Raft* (Seção 2.1.4.1.1). O Raft foi implementado no simulador *Neko*. Apesar de abordagens distintas para solucionar o problema de entrega ordenada, é possível obter uma linha de base entre estes dois algoritmos. Os experimentos foram realizados em dois cenários. No primeiro, é comparado um cenário sem falhas, com o número de processos participantes variando progressivamente com uma carga crescente de requisições, sendo obtido a latência total para entrega destas requisições. No segundo, é considerado um cenário com falhas, sendo avaliado a latência dos dois algoritmos para entrega das mensagens, mesmo após a falha de um processo qualquer no *AnyABCast* ou do líder no Raft.

Cenário sem falhas: Neste experimento os algoritmos avaliados foram submetidos a diferentes cargas de trabalho com requisições a serem ordenadas. Foram utilizados para os experimentos conjuntos de 5k, 10k, 20k, 50k e 100k requisições e analisado o impacto de aumentar o número de processos participantes na ordenação das requisições. Cada processo somente envia uma requisição quando a anterior enviada por ele houver sido entregue. Ao executar o *AnyABCast* as requisições a serem processadas foram distribuídas igualmente entre todos os processos, enquanto que no *Raft* somente o processo com o papel de líder recebe as requisições.

Na Figura 31 são apresentados os resultados dos experimentos realizados sem a presença de falhas entre os algoritmos avaliados. É possível observar uma clara diferença na latência para entrega das mensagens, visto que no *AnyABCast* todos os processos são capazes de processar as mensagens e iniciar o ordenamento. Como observado, a latência do *Raft* em relação ao *AnyABCast* foi 4.34, 6.49 e 10.36 vezes maior, nas configurações com 8,

Figura 31 – Comparação da latência entre os algoritmos Raft e AnyABCast - sem falhas.´

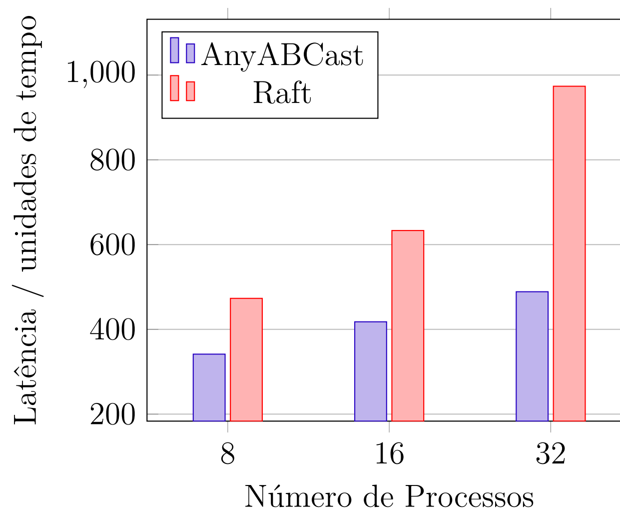


16 e 32 processos, respectivamente. Os resultados obtidos se devem pelo fato do processo líder no *Raft* receber todas as requisições, enquanto que no *AnyABCast* as mensagens são divididas igualmente entre todos os processos, permitindo que haja um balanceamento da carga de trabalho empregada.

Cenário com a falha de 1 processo: Neste experimento é considerada a falha de um dos processos participantes na ordenação das mensagens. É importante relembrar que no *Raft* os processos com o papel *followers* não causam impactos significativos se falharem, no entanto, o *leader* ao falhar faz com que um novo líder seja eleito. A simulação da falha de um processo é induzida no processo *leader*, no Raft, enquanto que no *AnyABCast* é escolhido um processo qualquer para induzir a falha, visto que todos os processos podem enviar e receber requisições. O experimento considera que os clientes estão desacoplados dos processos executando os algoritmos, fazendo com que haja um atraso adicional para envio e

recebimento de requisições. Um cliente ao identificar que sua requisição não foi processada, realiza a retransmissão da requisição identificada. Ao retransmitir uma requisição, todas as instâncias do Raft recebem a solicitação, mas somente o novo líder realiza o processamento desta requisição. Por outro lado, no *AnyABCast*, novas requisições são enviadas ao próximo processo correto, em relação ao processo falho. Cada processo executando os algoritmos possui um cliente associado que envia requisições a serem ordenadas.

Figura 32 – Latência para entrega de mensagens após a falha de um processo comparando o Raft vs AnyABCast



Na Figura 32 é apresentado o resultado do experimento com a simulação da falha de 1 processo. É possível observar um aumento na latência de entrega de mensagens no *Raft* em relação ao *AnyABCast*, este aumento foi 38,54%, 51,55% e 99,20% para as configurações com 8, 16 e 32 processos, respectivamente. Este resultado é devido à necessidade dos clientes retransmitirem as requisições a um novo líder e pela própria característica do algoritmo *Raft*, que avança sequencialmente o índice do *log* replicado, conforme os dados são enviados pelo líder aos demais processos.

4.3.2 Implementação do *AnyABCast* no *Akka.io*

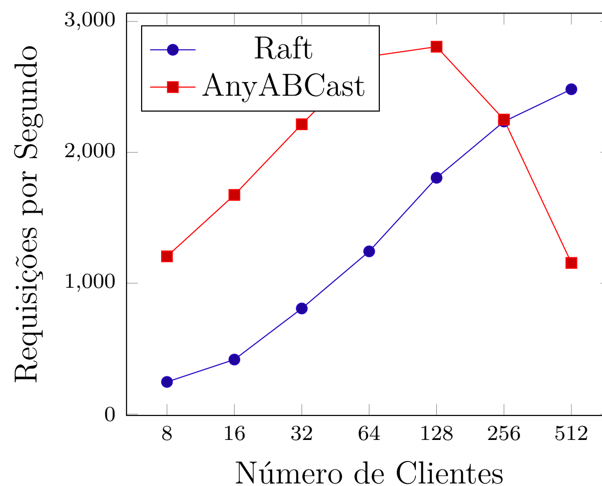
Implementar algoritmos distribuídos não é uma tarefa trivial, pois devem ser considerados inúmeros aspectos do sistema que será implementado. Com base no algoritmo proposto foi realizado a implementação do *AnyABCast* no *framework Akka.io* (LIGHTBEND, 2020). O *Akka* permite que a implementação do algoritmo não precise se preocupar com concorrência, pois cada componente do sistema é um ator. Cada ator possui uma *mailbox* que é utilizada para receber as mensagens de outros processos, cada mensagem recebida é retirada e processada da *mailbox* sequencialmente. O algoritmo foi implementado na linguagem JAVA. Foi utilizado o detector de falhas padrão do *framework*, que utiliza um detector *phi accrual* (HAYASHIBARA et al., 2004). Além disso, o algoritmo implementado foi comparado com uma implementação em Java do *Raft*, chamada *Apache*

Ratis (Apache Ratis™, 2021). A seguir serão apresentados resultados experimentais do desempenho obtido da implementação do *AnyABCast*.

Nos experimentos foram utilizadas 16 máquinas físicas, das quais 8 delas foram destinadas para execução dos algoritmos *AnyABCast* e *Raft*, e o restante utilizado para enviar requisições a serem replicadas e ordenadas aos algoritmos (clientes). Tanto os clientes quanto os algoritmos de replicação se comunicam somente através de mensagens enviadas pela rede. Os canais de comunicação permitem que qualquer dispositivo se comunique com qualquer outro. No entanto, nos experimentos utilizando o *AnyABCast* os dispositivos se comunicam através de uma topologia virtual hierárquica (VCube) capaz de se reorganizar na presença de falhas. Inicialmente, é apresentada uma comparação do desempenho dos dois algoritmos em um ambiente sem falha e com um número progressivo de clientes. Em seguida, é apresentada uma comparação do desempenho dos algoritmos em relação à falhas nos processos.

Experimento Sem Processos Falhos: Neste experimento o algoritmo *AnyABCast* é avaliado em relação ao número de requisições processadas por cada algoritmo. Para realização do experimento cada cliente submeteu requisições por 5 minutos, cada requisição somente é enviada por cada cliente após a requisição anterior houver sido processada. O número de dispositivos executando os algoritmo foi fixado em 8 e avaliado o impacto do aumento do número de clientes na vazão de cada algoritmo. O número de clientes variou de 8 a 512, sendo que cada máquina possuía de 1 a 64 clientes, visto que foram utilizados 8 máquinas. Ou seja, com 1 cliente em cada máquina, o total de clientes era 8, bem como, para 64 clientes por máquina, o total de clientes era 512.

Figura 33 – Cenário Sem Falhas.



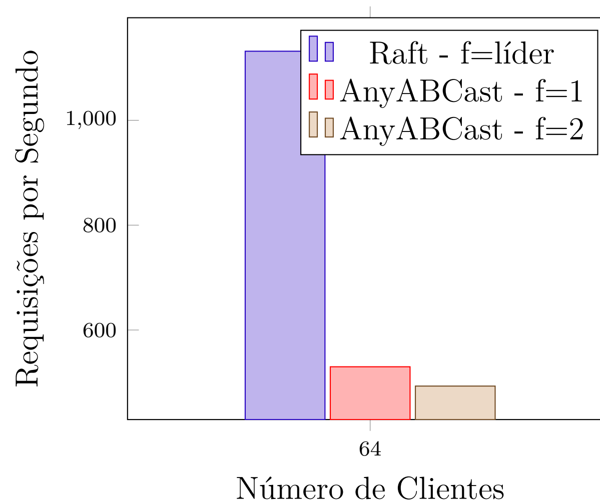
Nos experimentos foi observado um ganho de desempenho do *AnyABCast* expressiva quando executado em um cenário sem falhas. Ao observar na Figura 33 o número de clientes, temos que em relação à implementação do *Raft* houve um ganho de 390,01%, 301,72%, 174,38%, 119,78%, 55,45% e 0,66%, respectivamente para 8, 16, 32, 64, 128 e

256 clientes. No entanto, a partir de 512 clientes o desempenho em relação ao *Raft* fica 53,45% menor.

A diminuição no desempenho ocorre devido ao número de mensagens recebidas por cada cliente e o número de mensagens processadas pelos algoritmos de replicação. Conforme o número de clientes aumenta, mais mensagens são recebidas simultaneamente por cada cliente, visto que cada cliente recebe todas as requisições ordenadas, isto faz com que o desempenho de cada cliente seja afetado. Além disso, não utilizar um líder faz com que haja custo adicional para que as requisições sejam ordenadas, no *AnyABCast* este custo está relacionado com o número de mensagens trocadas pelo algoritmo. Conforme o número de clientes aumenta, consequentemente mais requisições serão enviadas, fazendo com que o *AnyABCast* necessite trocar mais mensagens entre os demais processos executando o algoritmo para que as mensagens sejam entregues.

Experimento Com Processos Falhos: Neste experimento o *Raft* e o *AnyABCast* são avaliados em relação ao impacto de falhas no processamento de requisições dos clientes. Foi utilizado o mesmo número de máquinas executando os algoritmos e fixado o número de clientes em 64. Em relação aos clientes, cada máquina utilizada para enviar requisições (8 ao total) possui 8 clientes, totalizando os 64 clientes do experimento. Esta configuração foi escolhida por apresentar uma das maiores taxas de entrega de requisições no cenário sem falhas do algoritmo *AnyABCast*.

Figura 34 – Cenário Com Falhas.



Na Figura 34 é apresentado os resultados obtidos da execução do algoritmo *AnyABCast* com a falha de 1 e 2 processos executando o algoritmo em relação a falha do líder no *Raft*. É fundamental relembrar que no *Raft*, os processos podem assumir um dentre 3 papéis (*follower*, *leader* e *candidate*), o papel do líder (*leader*) é receber as requisições e replicar aos demais processos para que as requisições sejam ordenadas. A falha no líder implica em um custo adicional relacionado a eleição de um novo líder, já o algoritmo

proposto não possui o custo de eleger um novo líder, visto que todos os processos podem enviar mensagens para serem ordenadas.

Em relação aos resultados obtidos é possível observar na Figura 34 que o *Raft* apresentou pouca variação na taxa de entrega de mensagens, mesmo com a falha do líder. Enquanto que o *AnyABCast*, em relação ao *Raft*, obteve uma taxa 53.18% e 56.44% menor, respectivamente para 1 e 2 processos falhos. Além disso, comparado a execução sem falhas, enquanto o *Raft* apresentou uma redução de 8.97%, no *AnyABCast* a diminuição da taxa de entrega de requisições foi de 80.61% e 81.96%, respectivamente com a falha de 1 e 2 processos.

No algoritmo *AnyABCast*, a diminuição na taxa de entrega de requisições se deve ao tempo para detecção de cada falha pelos demais processos, que é influenciada pelos parâmetros do detector de falhas do *framework akka.io*. Estes parâmetros indicam o quão conservador o detector de falhas é, quanto mais conservador menor é a taxa de falsas suspeitas. No entanto, apesar de serem utilizados os valores recomendados para redes locais, não foi obtido um desempenho satisfatório. Já no *Raft*, a detecção de falhas é realizada pelo próprio algoritmo. No *Raft*, os processos que detectam a falha do líder, iniciam a eleição de um novo líder e assim novas requisições podem ser processadas pelo novo líder eleito.

4.4 Conclusão

Neste capítulo foi apresentada uma solução hierárquica, autonômica e totalmente descentralizada para difusão atômica. A principal contribuição é uma estratégia escalável que permite aos processos entregarem as mensagens em ordem sem a necessidade de um líder. A solução proposta foi avaliada em cenários simulados e implementados em uma rede *Ethernet*. No cenário simulado é considerado duas estratégias de comunicação entre processos: todos para todos (chamada de *All2All*) em que os processos se comunicam diretamente, e a estratégia hierárquica, chamada VCube. Já a implementação foi realizada utilizando o *framework Akka.io* e comparada com o algoritmo de consenso *Raft*.

Os resultados simulados apresentaram cenários com e sem falhas de processos. Na presença de falhas, a abordagem hierárquica aproveita as propriedades do VCube para permitir que os processos da árvore se reorganizem automaticamente. Nos experimentos sem falhas o número de mensagens da estratégia hierárquica foi significativamente menor. A partir de 128 processos a latência foi maior na estratégia *All2All*. Nos cenários com falhas, não houve diferenças significativas no número de mensagens e observou-se que a latência é diretamente influenciada pelo intervalo de execução de rodadas do detector de falhas. Ainda assim, a abordagem hierárquica apresenta menor latência conforme o número de processos aumenta e supera a abordagem *All2All* quando a difusão atômica

apresenta mais de 256 processos, apresentando uma tendência de maior escalabilidade à solução proposta. Ao comparar com o algoritmo *Raft*, foram obtidos ganhos consideráveis em relação a latência para entrega de requisições, apesar de exigir que mais mensagens seja trocadas entre os processos para ordenação.

Os experimentos realizados com a implementação do *AnyABCast* e do *Raft* consideraram cenários com e sem falhas. Em ambos os cenários para execução do *AnyABCast* foi utilizada a topologia do VCube para organizar logicamente a comunicação entre os dispositivos. Na execução sem falha, o *AnyABCast* apresentou ganhos consideráveis de desempenho em relação ao *Raft*, no entanto, a partir de 256 clientes foi notado uma diminuição no número de requisições processadas pelo algoritmo em relação ao *Raft*. Na execução com falhas, o desempenho do *AnyABCast* foi afetado pelo atraso na detecção de falhas, apresentando uma queda considerável no desempenho do algoritmo em relação ao *Raft*.

5 Conclusão

As redes de computadores são um exemplo claro de sistema distribuídos e que possuem inúmeros algoritmos em execução para garantir a conectividade fim-a-fim de dispositivos remotos. Neste contexto, novos mecanismos utilizados para gerenciamento da rede e algoritmos devem ser constantemente avaliados para garantir seu correto funcionamento. Desta forma, esta dissertação propôs duas contribuições para sistemas distribuídos em geral. Em complemento, foram submetidos e publicados dois artigos com os resultados obtidos. O primeiro artigo foi publicado no *XXV Workshop de Gerência e Operação de Redes e Serviços (WGRS)* sob o título "Uma Avaliação sobre a Tolerância a Falhas no Plano de Dados em Controladores SDN". O segundo artigo foi publicado no *XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (SBRC 2021) sob o título "Abaixo o Líder: Um Algoritmo Hierárquico Sem-Líder para Difusão Atômica". A seguir são apresentadas cada uma das contribuições e os trabalhos futuros a serem realizados.

Na primeira contribuição foram apresentadas as SDN e os principais controladores distribuídos, o ONOS e ODL. Estes controladores foram extensivamente avaliados em relação ao plano de dados e ao plano de controle. Em relação ao plano de dados os experimentos realizados mostraram que o ONOS foi capaz de tratar diferentes tipos de falhas, enquanto que o ODL em inúmeros experimentos não executou suas ações como esperado. Por outro lado, ao avaliar o plano de controle distribuído os experimentos mostraram que o ODL foi mais estável conforme aumentou o número de controladores na rede e de dispositivos. Já no plano de controle o ONOS se mostrou instável. Desta forma, o ODL pode ser uma alternativa viável para o gerenciamento de redes de grande porte. Apesar de exigir que seja utilizada uma aplicação específica para manter a conectividade entre os dispositivos no plano de dados.

Na segunda contribuição foi proposto e implementado um algoritmo de difusão atômica sem líder. A difusão atômica é um problema fundamental para a criação de máquinas de estado replicadas e para garantir a consistência forte. O algoritmo desenvolvido adiciona nas mensagens trocadas entre cada processo marcações lógicas (*timestamps*) que são utilizadas para manter ordem. Para avaliar o algoritmo foi considerada a topologia de comunicação entre os processos executando o algoritmo e o desempenho da solução proposta em relação ao algoritmo de consenso *Raft*. Em relação à topologia, os experimentos mostraram que utilizar a topologia hierárquica em relação a uma topologia *All2All* permitiu obter uma redução no número de mensagens, apesar do impacto na latência para entrega de requisições. Ao comparar o algoritmo proposto com o *Raft* foi possível obter ganhos de desempenho tanto na versão simulada quanto em um ambiente não simulado. Os resultados

simulados apontaram que o *AnyABCast* se sobressaiu em ambientes sem falhas e com falhas. Já no ambiente não simulado foram obtidos resultados distintos, visto que conforme o número de clientes aumentou o desempenho foi se degradando e o impacto de falhas causou um redução drástica no número de mensagens entregues.

Trabalhos futuros no âmbito desta dissertação incluem implementar melhorias no módulo de encaminhamento do ODL que garanta o correto funcionamento do plano de dados; integrar o algoritmo proposto (*AnyABCast*) no controlador SDN ODL para prover um mecanismo que garanta a consistência forte sem a presença de um líder; estudar otimizações para reduzir o número de mensagens necessárias para a ordenação de requisições no *AnyABCast* e aumentar o desempenho perante falhas; avaliar o impacto de alterar o modelo de sistema do *AnyABCast* para que as premissas de sincronia sejam relaxadas; provar formalmente a terminação e corretude do *AnyABCast*.

Referências

Apache Ratis™. *Open source Java implementation for Raft consensus protocol.* 2021. Disponível em: <https://ratis.apache.org/>. Acesso em: 08 dez. 2021. 15, 86

ATOMIX. *Atomix Framework*. 2020. Disponível em: <https://atomix.io/>. Acesso em: 09 jan. 2021. 34

AVIŽIENIS, 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. ISSN 15455971. 18

BADOTRA, S.; PANDA, S. N. Evaluation and comparison of opendaylight and open networking operating system in software-defined networking. *Cluster Computing*, Springer, p. 1–11, 2019. 13, 34

BANNOUR, F.; SOUIHI, S.; MELLOUK, A. Adaptive State Consistency for Distributed ONOS Controllers. *2018 IEEE Global Communications Conference, GLOBECOM 2018 - Proceedings*, n. September, 2018. 61

BANNOUR, F.; SOUIHI, S.; MELLOUK, A. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys Tutorials*, v. 20, n. 1, p. 333–354, 2018. 26, 28, 29, 34

BENSON, T.; AKELLA, A.; MALTZ, D. A. Unraveling the complexity of network management. In: *NSDI*. [S.l.: s.n.], 2009. p. 335–348. 26

BOTELHO, F. et al. Design and implementation of a consistent data store for a distributed sdn control plane. In: *2016 12th European Dependable Computing Conference (EDCC)*. [S.l.: s.n.], 2016. p. 169–180. 13, 32, 55

BUDHIRAJA, N. Chapter 8 : The Primary Backup Approach. *New York*, p. 1–18, 1993. Disponível em: <http://dl.acm.org/citation.cfm?id=302438>. 21

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to reliable and secure distributed programming*. [S.l.: s.n.], 2011. 1–367 p. ISBN 9783642152597. 12, 22, 24, 69

CAESAR, M. et al. Design and implementation of a routing control platform. In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. [S.l.: s.n.], 2005. p. 15–28. 26

CBENCH. *Cbench Controller Benchmarker*. 2013. Disponível em: <https://github.com/mininet/oflops/tree/master/cbench>. Acesso em: 06 jan. 2021. 50

CHAIPET, S.; PUTTHIVIDHYA, W. On studying of scalability in single-controller software-defined networks. In: *2019 11th International Conference on Knowledge and Smart Technology (KST)*. [S.l.: s.n.], 2019. p. 158–163. 12, 13

CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The weakest failure detector for solving consensus. In: *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: Association for

Computing Machinery, 1992. (PODC '92), p. 147–158. ISBN 0897914953. Disponível em: <https://doi.org/10.1145/135419.135451>. 19

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 43, n. 2, p. 225–267, mar. 1996. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/226643.226647>. 19, 20, 26, 67

COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011. 12, 16, 17

DARIANIAN, M.; WILLIAMSON, C.; HAQUE, I. Experimental evaluation of two openflow controllers. In: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. [S.l.: s.n.], 2017. p. 1–6. 13

DAS, A.; GUPTA, I.; MOTIVALA, A. Swim: scalable weakly-consistent infection-style process group membership protocol. In: *Proceedings International Conference on Dependable Systems and Networks*. [S.l.: s.n.], 2002. p. 303–312. 35

DéFAGO, X.; SCHIPER, A.; URBáN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 36, n. 4, p. 372–421, dez. 2004. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/1041680.1041682>. 14, 24, 25

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 35, n. 2, p. 288–323, abr. 1988. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/42282.42283>. 17, 18

ENNS, R. et al. *Network Configuration Protocol (NETCONF)*. RFC Editor, 2011. RFC 6241. (Request for Comments, 6241). Disponível em: <https://rfc-editor.org/rfc/rfc6241.txt>. 28

FEDOR, M. et al. *Simple Network Management Protocol (SNMP)*. RFC Editor, 1990. RFC 1157. (Request for Comments, 1157). Disponível em: <https://rfc-editor.org/rfc/rfc1157.txt>. 28

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 32, n. 2, p. 374–382, abr. 1985. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/3149.214121>. 17, 19

FOSTER, N. et al. Frenetic: A network programming language. *ACM Sigplan Notices*, ACM New York, NY, USA, v. 46, n. 9, p. 279–291, 2011. 27

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, Association for Computing Machinery, New York, NY, USA, v. 33, n. 2, p. 51–59, jun. 2002. ISSN 0163-5700. Disponível em: <https://doi.org/10.1145/564585.564601>. 20

HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. USA, 1994. 17

HALTERMAN, J. *Distributed Systems in ONOS with Atomix 3*. 2018. Disponível em: <https://opennetworking.org/wp-content/uploads/2018/12/Distributed-Systems-in-ONOS-with-Atomix-3.pdf>. Acesso em: 01 dez. 2021. 35, 36

HANMER, R. et al. Friend or Foe: Strong Consistency vs. Overload in High-Availability Distributed Systems and SDN. *Proceedings - 29th IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW 2018*, IEEE, p. 59–64, 2018. 23, 62

HAYASHIBARA, N. et al. The /spl phi/ accrual failure detector. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* [S.l.: s.n.], 2004. p. 66–78. 20, 38, 85

HP. *Netperf: Github source-code*. 2020. Disponível em: <https://github.com/HewlettPackard/netperf>. Acesso em: 06 j. 2021. 39

JEANNEAU, D. et al. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector. *Proceedings - 7th Latin-American Symposium on Dependable Computing, LADC 2016*, p. 91–98, 2016. 70

KATTA, N. et al. Ravana: Controller fault-tolerance in software-defined networking. In: ACM. *SIGCOMM*. [S.l.], 2015. p. 12. 55

KIM, T.; MYUNG, J.; YOO, S. Load balancing of distributed datastore in opendaylight controller cluster. *IEEE Transactions on Network and Service Management*, v. 16, n. 1, p. 72–83, 2019. 63

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, Ieee, v. 103, n. 1, p. 14–76, 2014. 12, 26, 27, 28

KSHEMKALYANI, A. D.; SINGHAL, M. *Distributed Computing: Principles, Algorithms, and Systems*. 1. ed. [S.l.]: Cambridge University Press, 2011. ISBN 0521189845. 18

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. In: _____. *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019. p. 179–196. ISBN 9781450372701. Disponível em: <https://doi.org/10.1145/3335772.3335934>. 16

LAMPORT, L. et al. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18–25, 2001. 23

LAZAR, A. A. Programming telecommunication networks. *IEEE Network*, IEEE, v. 11, n. 5, p. 8–18, 1997. 26

LAZAR, A. A.; LIM, K.-S.; MARCONCINI, F. Realizing a foundation for programmability of atm networks with the binding architecture. *IEEE Journal on Selected Areas in Communications*, IEEE, v. 14, n. 7, p. 1214–1227, 1996. 12, 26

LF. *ODL Controller Documentation*. 2018. Disponível em: <https://docs.opendaylight.org/en/stable-oxygen/developer-guide/controller.html>. Acesso em: 09 jan. 2021. 13, 37

LIGHTBEND. *Akka: Documentation*. 2020. Disponível em: <https://akka.io/docs/>. Acesso em: 09 jan. 2021. 15, 37, 79, 85

LINGER, R.; MEAD, N.; LIPSON, H. Requirements definition for survivable network systems. In: *Proceedings of IEEE International Symposium on Requirements Engineering: RE '98*. [S.l.: s.n.], 1998. p. 14–23. 28

MCKEOWN, N. OpenFlow: Enabling Innovation in Campus Networks. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, 2008. Disponível em: <http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf>. 12, 26, 31

MININET. *Mininet: Ferramenta de simulação de Rede*. 2020. Disponível em: <http://mininet.org/>. Acesso em: 06 jan. 2021. 39, 50

MUQADDAS, A. S. et al. Inter-controller traffic to support consistency in onos clusters. *IEEE Transactions on Network and Service Management*, v. 14, n. 4, p. 1018–1031, 2017. 13

NARISETTY, R. et al. Openflow configuration protocol: implementation for the of management plane. In: IEEE. *2013 second GENI research and educational experiment workshop*. [S.l.], 2013. p. 66–67. 28

ONF. *ONOS: Osgi Property Constants*. 2019. Disponível em: <https://github.com/opennetworkinglab/onos/blob/2.2.0/core/store/dist/src/main/java/org/onosproject/store/OsgiPropertyConstants.java>. Acesso em: 12 jan. 2022. 55

ONF. *ONOS: Controller Wiki*. 2020. Disponível em: <https://wiki.onosproject.org/>. Acesso em: 14 abr. 2021. 13

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. 22, 23

OPENDAYLIGHT. *SDN Controller Platform (OSCP):Overview*. 2019. Disponível em: <https://wiki.opendaylight.org/view/OpenDaylight_SDN_Controller_Platform_(OSCP):Overview>. Acesso em: 06 jun. 2020. 46

OPENFLOW. *OpenFlow Switch Specification Version 1.5.1*. 2015. Disponível em: <https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>. Acesso em: 19 mai. 2021. 31

OPENVSWITCH. *OpenVSwitch: website*. 2020. Disponível em: <https://www.openvswitch.org/>. Acesso em: 06 jan. 2021. 50

PANDA, A. et al. Cap for networks. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 91–96. ISBN 9781450321785. Disponível em: <https://doi.org/10.1145/2491185.2491186>. 20

PAZNIKOV, A. A.; GURIN, A. V.; KUPRIYANOV, M. S. Implementation in actor model of leaderless decentralized atomic broadcast. In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. [S.l.: s.n.], 2020. p. 1–4. 71

PFAFF, B.; DAVIE, B. *The Open vSwitch Database Management Protocol*. RFC Editor, 2013. RFC 7047. (Request for Comments, 7047). Disponível em: <https://rfc-editor.org/rfc/rfc7047.txt>. 28

POKE, M.; HOEFLER, T.; GLASS, C. W. Allconcur: Leaderless concurrent atomic broadcast. In: . New York, NY, USA: Association for Computing Machinery, 2017. (HPDC '17), p. 205–218. ISBN 9781450346993. Disponível em: <https://doi.org/10.1145/3078597.3078598>. 66, 70, 71

PTPD. *Precision Time Protocol Daemon*. 2020. Disponível em: <https://sourceforge.net/projects/ptpd/>. Acesso em: 06 jan. 2021. 56

REICH, J. et al. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013. 27

RICHARDSON, L.; RUBY, S. *Restful Web Services*. First. [S.l.]: O'Reilly, 2007. ISBN 9780596529260. 27

RODRIGUES, L. A.; ARANTES, L.; DUARTE JR., E. P. An autonomic implementation of reliable broadcast based on dynamic spanning trees. In: *2014 Tenth European Dependable Computing Conference*. [S.l.: s.n.], 2014. p. 1–12. 66, 70, 72

RUCHEL, L. et al. Abaixo o líder: Um algoritmo hierárquico sem-líder para difusão atômica. In: *Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Porto Alegre, RS, Brasil: SBC, 2021. p. 15–28. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc/article/view/16708>. 14

RUCHEL, L.; TURCHETTI, R.; CAMARGO, E. de. Uma avaliação sobre a tolerância a falhas no plano de dados em controladores sdn. In: *Anais do XXV Workshop de Gerência e Operação de Redes e Serviços*. Porto Alegre, RS, Brasil: SBC, 2020. p. 153–166. ISSN 2595-2722. Disponível em: <https://sol.sbc.org.br/index.php/wgrs/article/view/12458>. 14

SAKIC, E.; KELLERER, W. Response time and availability study of raft consensus in distributed sdn control plane. *IEEE Transactions on Network and Service Management*, v. 15, n. 1, p. 304–318, 2018. 62

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/98163.98167>. 16

SHEINBEIN, D.; WEBER, R. Stored program controlled network: 800 service using spc network capability. *The Bell System Technical Journal*, Nokia Bell Labs, v. 61, n. 7, p. 1737–1744, 1982. 26

SOARES, A. A. Z. et al. SDN-based teleprotection and control power systems: A study of available controllers and their suitability. *International Journal of Network Management*, n. April, p. 1–20, 2020. ISSN 1055-7148. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.2112>. 13, 63

SPALLA, E. S. et al. AR2C2: Actively replicated controllers for SDN resilient control plane. *Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, p. 189–196, 2016. 30, 32

SU, J.; WANG, W.; LIU, C. A survey of control consistency in Software-Defined Networking. *CCF Transactions on Networking*, Springer Singapore, v. 2, n. 3-4, p. 137–152, 2019. ISSN 2520-8462. Disponível em: <https://doi.org/10.1007/s42045-019-00022-w>. 32, 33

SUH, D. et al. Toward highly available and scalable software defined networks for service providers. *IEEE Communications Magazine*, v. 55, n. 4, p. 100–107, 2017. 13, 38, 63

T.BAH, M. et al. Topology discovery performance evaluation of opendaylight and onos controllers. In: *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. [S.l.: s.n.], 2019. p. 285–291. 13

TENNENHOUSE, D. L. et al. A survey of active network research. *IEEE communications Magazine*, IEEE, v. 35, n. 1, p. 80–86, 1997. 12, 26

TERRA, A.; CAMARGO, E.; DUARTE JR., E. A caminho de uma alternativa hierárquica para implementação do algoritmo de consenso paxos. In: *Anais do XXI Workshop de Testes e Tolerância a Falhas*. Porto Alegre, RS, Brasil: SBC, 2020. p. 15–28. ISSN 2595-2684. Disponível em: <https://sol.sbc.org.br/index.php/wtf/article/view/12484>. 66, 70

TOOTOONCHIAN, A. et al. On controller performance in software-defined networks. In: *2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*. [S.l.: s.n.], 2012. 28

URBAN, P.; DEFAGO, X.; SCHIPER, A. Neko: a single environment to simulate and prototype distributed algorithms. In: *Proceedings 15th International Conference on Information Networking*. [S.l.: s.n.], 2001. p. 503–511. 79

VILCHEZ, J. M. S.; SARMIENTO, D. E. Fault Tolerance Comparison of ONOS and OpenDaylight SDN Controllers. In: *2018 4th IEEE Conference on Network Softwarization and Workshops, NetSoft 2018*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2018. p. 384–392. ISBN 9781538646335. 13, 39, 45, 63

VIOTTI, P.; VUKOLIć, M. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 49, n. 1, jun. 2016. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/2926965>. 21

VIZARRETA, P. et al. Dason: Dependability assessment framework for imperfect distributed sdn implementations. *IEEE Transactions on Network and Service Management*, v. 17, n. 2, p. 652–667, 2020. 13, 62

ZHANG, T. et al. The role of the inter-controller consensus in the placement of distributed sdn controllers. *Comput. Commun.*, Elsevier Science Publishers B. V., NLD, v. 113, n. C, p. 1–13, nov. 2017. ISSN 0140-3664. Disponível em: <https://doi.org/10.1016/j.comcom.2017.09.007>. 13

ZHANG, Z. et al. 6g wireless networks: Vision, requirements, architecture, and key technologies. *IEEE Vehicular Technology Magazine*, v. 14, n. 3, p. 28–41, 2019. 28

ZHU, L. et al. Sdn controllers: Benchmarking & performance evaluation. *arXiv preprint arXiv:1902.04491*, 2019. 13