Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Plano de Controle Seguro e Distribuído para Redes Definidas por Software

Jefferson Pereira da Silva

Dissertação apresentada como requisito parcial para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2020

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

SS586p

Silva, Jefferson Pereira da
Um Plano de Controle Seguro e Distribuído para Redes Definidas por Software / Jefferson Pereira da Silva; orientador Eduardo Adilio Pelinson Alchieri. -- Brasília, 2020.
96 p.

Dissertação (Mestrado - Mestrado em Informática) -- Universidade de Brasília, 2020.

1. Segurança. 2. SDN. 3. Controladores. I. Alchieri, Eduardo Adilio Pelinson , orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Plano de Controle Seguro e Distribuído para Redes Definidas por Software

Jefferson Pereira da Silva

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Odorico Machado Mendizabal
INE/UFSC

Prof. Dr. Jacir Luiz Bordim
CIC/UnB

Prof. Dr. Genaina Nunes Rodrigues
Coordenador do Programa de Pós-graduação em Informática

Brasília, 01 de Março de 2020

Dedicatória

Dedico esse trabalho primeiramente a Deus por ter me ajudado chegar ate aqui e a toda minha família, pelos momentos em que estive fora me dedicando a realização deste sonho. Em especial, agradeço a minha esposa que em todos os momentos do mestrado me ajudou me apoiando e dando forças nos momentos em que pensei que não seria capaz, sua ajuda em algumas disciplinas especificas foi fundamental para meu sucesso. Te amo.

Agradecimentos

Agradeço imensamente ao Prof. Dr. Eduardo Adilio Pelinson Alchieri por suas orientações e rico conhecimento em todas as etapas do desenvolvimento do trabalho, ele sempre esteve ali presente me apoiando e dando dicas e me ensinando de verdade o que eu deveria fazer e com seu auxilio pude chegar até aqui. Ele é um exemplo a ser seguido por mim. Obrigado professor! Também agradeço aos colegas de laboratório Ranyelson e Lucas por terem me auxiliado na etapa final do trabalho, no qual fizemos alguns avanços interessantes.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Redes Definidas por Software (SDN) surgiram como um novo paradigma para gerenciamento de redes, definindo uma arquitetura que separa os planos de dados e de controle. Uma arquitetura SDN baseada em um controlador centralizado não escala e nem tolera falhas, pois apresenta um ponto único de falhas. Controladores distribuídos baseados em um modelo de consistência eventual para gerenciamento do estado da rede também apresentam sérios problemas: um modelo de programação complexo para as aplicações de rede; e pode gerar anomalias (como *loops*) na rede. Consequentemente, soluções considerando um modelo de dados com consistência forte para o armazenamento das informações da rede SDN foram propostos. Nestas abordagens, os controladores distribuídos usam um armazenamento de dados consistente e tolerante a falhas para armazenar o estado relevante das aplicações e da rede. Infelizmente, estas propostas existentes não consideram requisitos fundamentais de segurança para a arquitetura SDN. Este trabalho apresenta nossos esforços no projeto, implementação e avaliação de um modelo seguro e consistente para o plano de controle, baseado no DepSpace, que é um espaço de tuplas com propriedades de segurança. Modificamos duas aplicações de rede (Aprendizado de *Switch* e Balanceador de carga) de um controlador de mercado (i.e., Floodlight), integrando ao DepSpace e verificamos os custos introduzidos nesta arquitetura. Além disso, verificamos os custos apenas com a utilização do DepSpace considerando os acessos necessários para cada aplicação. Por fim, verificamos o uso de um mecanismo de coordenação extensível, presente no DepSpace para melhorar o desempenho da nossa arquitetura. Experimentos foram realizados aplicando nossa proposta e foi possível verificar o ganho de benefícios associados a manutenção de informações sensíveis em um espaço seguro evitando acessos indevidos e garantindo uma alta disponibilidade com a utilização de um armazenamento distribuído e seguro. Resultados mostram, ainda, um custo de aproximadamente $0,4$ segundos para se conectar ao armazenamento de dados seguro, significando um custo muito pequeno comparado com as vantagens das propriedades de seguranças adquiridas com o uso de tal arquitetura.

Palavras-chave: Segurança, SDN, Controladores

Abstract

Software Defined Networks (SDN) emerged as a new paradigm for network management, defining an architecture that physically decouples the control and data planes. A SDN architecture based on a central controller does not scale and neither is fault-tolerant since it presents a single point of failure. Distributed SDN controllers based on eventually consistent model for the network state also brings serious drawbacks: a complex programming model for network applications; and it can lead to network anomalies. Consequently, solutions considering a strong consistent model for the network state are emerging. In these approaches, the distributed controllers use a consistent and fault-tolerant data store that keeps relevant network and applications state. Unfortunately, these approaches do not consider security requirements for the SDN network. This work aims to design, implement and evaluate a secure and consistent model for the control plane based on DepSpace, a secure tuple space implementation. Experimental results show the practical feasibility of the proposed architecture. We modified two network applications (switch learning and load balancer) from a market controller (i.e., Floodlight), integrated them in DepSpace and analysed the costs introduced in this architecture. In addition, we check costs only with the use of DepSpace considering the accesses necessary in each application. Finally, we verified the use of extensible coordination, present in DepSpace, to improve the performance of our architecture. Experiments were performed using our proposal to verify the benefits with the use of a distributed and secure storage. Results also show an approximate cost of 0.4 seconds to connect to secure data storage, meaning a very low cost when compared to the advantages of security properties provided by the architecture.

Keywords: Security, SDN, Controllers

Sumário

1 Introdução **1**
1.1 Motivação . . . 1
1.2 Objetivos . . . 3
1.2.1 Objetivos Específicos . . . 3
1.3 Organização do texto . . . 3

2 Fundamentação Teórica **5**
2.1 Segurança Computacional . . . 5
2.2 Sistemas Distribuídos . . . 8
2.2.1 Modelo de Sincronismo . . . 9
2.2.2 Modelos de Falhas . . . 10
2.2.3 Modelos de Consistência . . . 12
2.2.4 Espaço de Tuplas . . . 15
2.3 *Software Defined Network* (SDN) . . . 20
2.4 Considerações Finais . . . 27

3 Revisão do Estado da Arte **28**
3.1 Controladores . . . 28
3.1.1 Controladores Centralizados . . . 29
3.1.2 Controladores Distribuídos . . . 34
3.2 Propriedades de Segurança dos Controladores . . . 43
3.2.1 Outros Problemas de Segurança em SDN . . . 46
3.3 Considerações finais . . . 52

4 Proposta **54**
4.1 Plano de Controle Seguro e Distribuído baseado no DEPSPACE . . . 54
4.1.1 Aplicações de Rede . . . 56
4.2 Uso de Coordenação Extensível no DepSpace . . . 62
4.3 Integração no Controlador Floodlight . . . 63

4.4 Experimentos 64
4.4.1 Custo necessários para acessar o DepSpace 65
4.4.2 Desempenho da Solução Integrada ao FloodLight 67
4.5 Conclusão 70

5 Conclusão **72**
5.1 Visão Geral do Trabalho 72
5.2 Revisão dos Objetivos e Contribuições desta Dissertação 73
5.3 Perspectivas Futuras 74

Referências **75**

Lista de Figuras

2.1 Atributos de segurança e dependabilidade. 8
2.2 Generais bizantinos que enviam mensagens entre si para chegarem em um acordo sobre um valor booleano [1]. 15
2.3 Operações básicas do Espaço de Tuplas. 16
2.4 Arquitetura DepSpace, adaptado de [2] 17
2.5 Cálculo do *fingerprint*. 20
2.6 Rede tradicional. 22
2.7 Rede SDN. 23
2.8 Camadas SDN. 25

3.1 Arquitetura SDN com um controlador centralizado [3]. 30
3.2 Componentes de uma rede baseada em NOX: comutadores OpenFlow (OF), um servidor executando um processo do controlador NOX e um banco de dados que contém a visão da rede [4]. 31
3.3 Controlador Floodlight e suas relações com outras camadas. 33
3.4 Arquitetura controladora distribuída [3]. 36
3.5 Controladores horizontalmente Distribuídos. 37
3.6 Controladores verticamente Distribuídos. 37
3.7 Arquitetura do ONOS, adaptado de [5]. 39
3.8 Arquitetura do Kandoo. 39
3.9 Implantação do B4 ao redor do mundo [6]. 40
3.10 Arquitetura do Espresso, adaptado de [7]. 41
3.11 Controlador principal e controladores secundários. 41
3.12 Solicitação de controle. 42
3.13 Arquitetura do SMaRt-Light, adaptado de [8]. 43
3.14 Vetores de ameaças SDN, adaptado de [9]. 46

4.1 Plano de controle baseado no DepSpace. 55
4.2 Aprendizado de *Switch*. 57
4.3 Aprendizado de *Switch*. 59

4.4 Balanceador de Carga. 61
4.5 Integração de uma rede emulada no mininet com o controlador Floodlight e este com o DepSpace. O tráfego entre os *hosts* é interceptado pelo controlador e neste existe um módulo chamado *Integration* que manipula o *packet-in* e envia ao armazenamento seguro. 64
4.6 Latência medida em *ms*; a aplicação aprendizado de *switch broadcast* foi a aplicação que teve a menor latência, pois executa apenas uma operação no DepSpace, já a aplicação aprendizado de *switch unicast* apresentou maior latência pois executa 2 operações no DepSpace, a aplicação balanceador de carga apresentou a maior latência pois executa diversas operações no armazenamento distribuído, felizmente as aplicações aprendizado de *switch unicast* e balanceador de carga com extensões apresentaram latências próximas a da aplicação aprendizado de *switch broadcast* 66
4.7 Vazão medida em Kop/seg. Percebe-se que a vazão aumenta com o número de controladores e a aplicação com maior vazão foi o aprendizado de *switch* pois é a *workload* menos custosa, pois executa apenas uma operação no DepSpace; a *workload* com menor vazão foi o balanceador de carga que tem que lidar com a concorrência nos controladores. 66
4.8 Acessos subsequentes ao DepSpace. 68
4.9 Aprendizado de *Switch* - Floodlight vs DepSpace. Apesar do tempo de acesso ao DepSpace ser o mais custoso.As operações são menos custosas que as do Floodlight pois com a utilização do DepSpace as operações são realizadas no espaço seguro, necessitando, apenas, de uma chamada ao DepSpace. 69
4.10 Balanceador de carga - Floodlight vs DepSpace. Nota-se que os custos para armazenar os dados no DepSpace são similares ao armazenamento de dados padrão do controlador Floodlight, levemente maiores. Porém o custo é vantajoso pois os benefícios justificam essa diferença. 70

Lista de Tabelas

2.1 Falhas em Sistemas Distribuídos [10]. 12

3.1 Comparativo das Arquiteturas de Controladores SDN. 29
3.2 Classificação física das arquiteturas do plano de controle SDN distribuídas. . 42
3.3 Principais características de alguns controladores SDN. 45
3.4 Desafios de segurança no plano de controle SDN. 49
3.5 Ameaças e Soluções de segurança SDN. 52

4.1 Mapeamento Portas - MAC. 56

Lista de Abreviaturas e Siglas

API *Application Programming Interface*.

ASIO *Asynchronous I/O*.

DDoS *Distributed DoS*.

DoS *Denial of Service*.

ISP *Internet Service Provider*.

LLDP *Link Layer Discovery Protocol*.

MTBF *Mean Time Between Failures*.

NOS *Network Operating System*.

OF *OpenFlow*.

ONF *Open Networking Foundation*.

SA *Simulated Annealing*.

SDN *Software Defined Network*.

SOM *Self-Organizing Maps*.

VPN *Virtual Private Network*.

Capítulo 1

Introdução

1.1 Motivação

Redes Definidas por Software (SDN) surgiram como um novo paradigma que possibilita a elasticidade e o dinamismo na operação das redes, facilitando seu gerenciamento. Nestas redes, o plano de controle é fisicamente separado do plano de dados e a visão global da rede é logicamente centralizada, o que facilita o desenvolvimento de aplicações e serviços de rede. As redes SDN mudaram radicalmente a forma de gerenciamento das redes, trazendo uma maior capacidade de implementações de políticas e rígidos controles de segurança no plano de controle. O fato de dividir as responsabilidades das redes em dois planos, um plano de controle (que tem por funções coordenar os recursos da rede e gerenciar as políticas) e um plano de dados (responsável por encaminhar as mensagens e informações entre os nós que compõe a infraestrutura global da rede) traz diversas vantagens como, por exemplo, a facilidade na manutenção e um maior controle pelos gerentes e administradores de redes.

Inicialmente, o plano de controle destas redes era baseado em um controlador centralizado, considerado o "cérebro" da rede, que possui uma visão global consistente de toda a rede. Neste cenário, ainda é possível que os administradores de rede criem mecanismos de proteção para evitar acessos indevidos ao núcleo central da rede, i.e., ao controlador centralizado. Porém, com o aumento da rede, um único controlador pode tornar-se o gargalo da rede, i.e., o mesmo pode não ter capacidade computacional suficiente para atender todas as demandas encaminhadas para o mesmo. Por exemplo, sempre que um *switch* recebe um fluxo de dados para um caminho ainda desconhecido, o controlador precisa ser consultado para obtenção desta informação, tornando-se o gargalo de uma rede com milhares de nós interconectados através de vários *switches*. Além disso, um controlador centralizado representa um ponto único de falha [11, 12, 13] que pode ser explorado por um atacante, seja por tentativas fraudulentas de injeção de código malicioso

visando comprometer a rede ou por ataques de negação de serviço visando sobrecarregar o controlador.

Para contornar este problema, foram propostos planos de controle escaláveis baseados em controladores distribuídos e tolerantes a falhas, os quais usualmente adotam um modelo de consistência eventual (*eventual consistency*) para gerenciamento do estado da rede [13]. Apesar desta abordagem apresentar um bom desempenho, ela apresenta pelo menos dois sérios problemas [13]: leva a um modelo de programação complexo para as aplicações/serviços de rede, pois as mesmas devem lidar com a possibilidade de dados estarem inconsistentes por um determinado tempo; e pode gerar anomalias na rede, como *loops* no encaminhamento de pacotes que podem levar a vários problemas na rede, como a quebra de conexões.

Consequentemente, soluções considerando um modelo de dados com consistência forte para gerenciamento e armazenamento das informações sobre o estado da rede SDN começaram a aparecer na literatura [12, 13, 14]. Estas abordagens definem uma arquitetura para o plano de controle baseada em um *data store*: os controladores distribuídos usam um *data store* consistente e tolerante a falhas para armazenamento do estado relevante das aplicações e da rede. Infelizmente, estas propostas existentes não consideram requisitos fundamentais de segurança para a arquitetura SDN, o que pode levar a vários problemas de segurança, como por exemplo um atacante pode tentar capturar ou forjar regras de fluxos comprometendo a confidencialidade ou a integridade da rede, respectivamente. Com o objetivo de preencher esta lacuna, neste trabalho buscamos projetar, implementar e avaliar um modelo seguro e consistente para o plano de controle. Em nossa abordagem usamos o DepSpace [2, 15, 16], que é um espaço de tuplas com propriedades de segurança, para armazenamento seguro e consistente das informações relevantes da rede SDN, o qual funciona como um *data store*, além de permitir a coordenação entre processos (controladores), característica fundamental dos espaços de tuplas.

Com a utilização do DepSpace teremos as vantagens de um *data store* tolerante a falhas e seguro, mantendo o rede disponível e os dados replicados em n réplicas que suportam este *data store*, de tal forma que falhas (por parada ou maliciosas) em alguns destas réplicas (no máximo f, sendo $n \geq 3f+1$) não comprometem a disponibilidade e segurança destes dados. Por exemplo, se em uma rede SDN um controlador falhar, outro pode assumir rapidamente o seu lugar e manter o funcionamento da rede sem interrupção, pois os dados são buscados no DepSpace, ficando sempre disponíveis de forma segura, mesmo na falha de algum controlador.

Para avaliar a arquitetura proposta, na Seção 4.4 apresentamos duas aplicações que podem se beneficiar desta infraestrutura em uma rede SDN, a saber, balanceador de carga e aprendizado de *switch*, e avaliamos os custos introduzidos neste projeto, onde

todos os dados manipulados pelas aplicações são armazenadas de forma segura e confiável no DepSpace.

1.2 Objetivos

O objetivo geral desse trabalho é projetar, implementar e avaliar um modelo seguro e consistente para o plano de controle de uma rede SDN, utilizando o DepSpace, que é um espaço de tuplas com propriedades de segurança, para armazenamento seguro e consistente das políticas e informações relevantes de uma rede e suas aplicações.

1.2.1 Objetivos Específicos

Visando atender o objetivo geral, este trabalho possui os seguintes objetivos específicos:

- Estudar os conceitos relevantes para o trabalho, relacionados com Redes Definidas por Software, Segurança Computacional e Sistemas Distribuídos;
- Projetar e implementar uma arquitetura que introduz segurança aos dados manipulados pelo plano de controle através da utilização do DepSpace;
- Projetar, implementar e avaliar o desempenho de aplicações de rede que utilizem esta arquitetura, comparando-as com a arquitetura tradicional, sem mecanismos de segurança;
- Refinar a arquitetura e as aplicações anteriormente propostas, incluindo a utilização de mecanismos de coordenação extensível presentes no DepSpace;
- Avaliar o desempenho de aplicações de rede utilizando coordenação extensível.

1.3 Organização do texto

O restante deste texto está organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica sobre Segurança Computacional, Sistemas Distribuídos, tratando os modelos de falhas, o Espaço de Tuplas, o DepSpace e por fim as Redes Definidas por Softwares e os problemas relacionados.

O Capítulo 3 apresenta os conceitos de SDN com um detalhamento dos Sistemas operacionais de Rede, controladores, dividindo-os em centralizado e distribuído, e apresentando os conceitos de segurança, os problemas advindos com a arquitetura SDN e as soluções propostas na literatura.

Em seguida, o Capitulo 4 apresenta nossa proposta para um plano de controle seguro e distribuído baseado no DepSpace. A Seção 4.4 discute alguns experimentos executados com dois serviços de rede criados sobre a arquitetura proposta e apresenta os resultados alcançados.

Finalmente, no Capítulo 5 são apresentadas as conclusões do trabalho, na qual uma visão geral do trabalho é apresentada, além da revisão dos objetivos e contribuições literárias.

Por fim, as perspectivas futuras desta dissertação é exposta.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos fundamentais sobre o qual a proposta é aplicada, abordando desde os aspectos de segurança até a arquitetura SDN.

2.1 Segurança Computacional

Nos dias de hoje, a informação é essencial para todas as tarefas diárias da vida em sociedade, seja na indústria, no comércio ou em casa, com isso o tratamento da mesma deve ser feito de maneira adequada. Qualquer tentativa de obtenção, manutenção, falsificação ou disseminação de informações sem as devidas autorizações torna-se um grande risco para os negócios da coletividade em geral, ocasionando uma crescente expansão das formas de armazenamento e de distribuição das informações ao redor do globo. No cenário atual, tem-se observado o crescente impacto em garantir a segurança das informações.

Com a utilização crescente das informações na Internet para atividades essenciais, a preocupação e o risco de invasão dos sistemas cresceram de forma elevada nos últimos tempos. Manter sistemas que armazenam informações de maneira confiável é um desafio explorado cada vez mais. Os crimes digitais são muito difíceis de serem descobertos e até mesmo rastreados, muitas empresas sofrem invasões diariamente em seus sistemas e necessitam de mecanismos robustos para evitar que seus dados sejam comprometidos.

Tendo em vista este cenário, existem alguns mecanismos para tentar evitar ataques aos dados e informações como, por exemplo, *Firewalls*, Criptografia, Certificados Digitais, VPN (*Virtual Private Networks*), *Smart Cards* e Biometria. Não existe nenhum sistema que possa ser considerado invulnerável em matéria de proteção, muito embora, diversos recursos forneçam boa segurança, os sistemas ainda deixam alguma “brecha” que um atacante pode se utilizar para capturar as informações desejadas.

Segurança Computacional visa prevenir que atacantes alcancem seus objetivos através do acesso não autorizado ou uso não autorizado dos computadores e suas redes [17], sendo:

- **Acesso não autorizado**: ocorre quando um indivíduo acessa informações ou recursos sem a devida autoridade.
- **Uso não autorizado**: ocorre quando um indivíduo com autoridade para acessar informações ou recursos de um certo modo acessa de outras formas.

Ainda, segundo [18], a segurança em um Sistema de Informação (SI) visa protegê-lo contra ameaças à confidencialidade, à integridade e à disponibilidade das informações e dos recursos sob sua responsabilidade. Dessa forma, para que um sistema seja considerado seguro é necessário que atenda os três conceitos apresentados a seguir [19].

Confidencialidade

Confidencialidade significa a manutenção do sigilo das informações ou dos recursos. É a propriedade de que a informação não esteja disponível ou revelada a indivíduos, entidades ou processos não autorizados [20]. A violação da confidencialidade ocorre com a revelação não autorizada da informação ou dos recursos. A prevenção contra as ameaças à confidencialidade em SI pode ser alcançada com a aplicação de mecanismos de controle de acesso e com técnicas de criptografia e de segurança de redes. A confidencialidade também se aplica a mera existência de um dado, que pode ser mais importante do que o dado em si. Todos os mecanismos que impõem confidencialidade requerem serviços para suportá-los.

Abaixo dois exemplos onde a confidencialidade é aplicada:

- **Criptografia como mecanismo de controle de acesso**. O mecanismo aplicado para proteger as informações de um extrato de conta corrente evita alguma pessoa ou processo de ter tal informação. Caso o cliente, possuidor desta conta necessite verificar tal extrato, ele precisará decifrar tais dados. Com isso, somente quem possui a chave de criptografia poderá ler tais dados. Contudo, caso outros consigam ler estes dados, o mecanismo de confidencialidade será quebrado e a confidencialidade se torna comprometida.
- **Proteção da existência de um dado:** Muitas vezes, saber se um indivíduo é um cliente VIP de um banco pode ser mais importante que saber quais/quantos são os recursos que ele possui aplicados.

Integridade

Integridade é a propriedade de salvaguarda da exatidão e completeza de ativos [20]. Isso quer dizer que a integridade se refere a confiabilidade da informação ou dos recursos. A violação da integridade ocorre pela modificação imprópria ou não autorizada da informação ou dos recursos. Para manter a integridade dos dados, deve-se manter a integridade

na origem, quer dizer que quem criou o dado seja confiável e seu conteúdo também o seja. Alguns mecanismos permitem obter a confiabilidade das fontes, como o uso de autenticação. Ainda, a integridade se sustenta na acurácia e na credibilidade da fonte e na confiança que as pessoas depositam na informação. Abaixo alguns exemplos onde a integridade pode ser comprometida.

- **Corrupção da integridade de origem**: Um jornal pode noticiar uma informação obtida de um vazamento no Palácio do Planalto, mas atribuí-la a uma fonte errada. A informação impressa é recebida (integridade dos dados preservada), mas a fonte é incorreta (corrupção na integridade de origem).

- **Distinção entre acesso e uso não autorizados**: Num sistema contábil, o acesso não autorizado ocorre quando alguém quebra a segurança para tentar modificar o dado de uma conta, por exemplo, para quitar um débito, sem autoridade para tal. Agora, quando o contador da empresa tenta apropriar-se de dinheiro desviando-o para contas no exterior e ocultando respectivas transações, então ele está abusando de sua autoridade, embora possa, ele não deve fazê-lo.

Disponibilidade

Disponibilidade é a propriedade de estar acessível e utilizável sob demanda por uma entidade autorizada. Sistemas de consulta de base de dados *online*, servidores de rede, servidores de páginas web, são alguns exemplos de sistemas onde alta disponibilidade é requerida. Abaixo um exemplo onde o conceito de Disponibilidade é aplicado.

- **Manipulação de mecanismos de disponibilidade**: Suponha que Ana tenha comprometido o servidor secundário de um banco, que fornece os saldos das contas correntes. Quando alguém solicita informações ao servidor, Ana pode fornecer a informação que desejar. Caixas validam saques contatando o servidor primário. Caso ele não obtenha resposta, o servidor secundário é solicitado. Um cúmplice de Ana impede que caixas contactem o servidor primário, de modo que todos eles acessam o servidor secundário. Ana nunca tem um saque ou cheque recusado, independente do saldo real em conta. Note que se o banco tivesse apenas o servidor primário, este esquema não funcionaria – o caixa não teria como validar o saque.

Além desses atributos de segurança da informação, existe o conceito de dependabilidade, no qual é um conceito integrador que engloba os atributos abaixo [19]: A Figura 2.1 resume a relação entre dependabilidade e segurança em termos de seus principais atributos:

Disponibilidade: prontidão para o serviço correto.

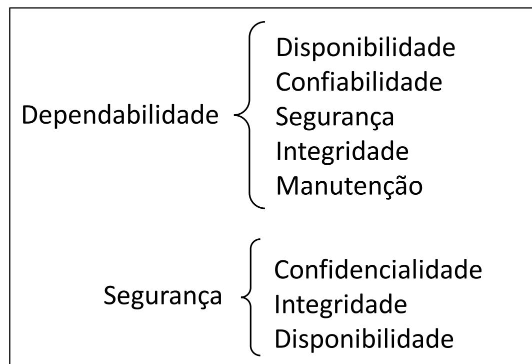

Figura 2.1: Atributos de segurança e dependabilidade.

Confiabilidade: continuidade do serviço correto.

Segurança: ausência de consequências catastróficas para o(s) usuário(s) e o meio ambiente.

Integridade: ausência de alterações inadequadas do sistema.

Capacidade de manutenção: capacidade de sofrer modificações e reparos.

2.2 Sistemas Distribuídos

Um Sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. Essa definição leva às seguintes características dos sistemas distribuídos: concorrência de recursos, falta de um relógio global e possibilidade de falhas de componentes independentes [10]. O compartilhamento de recursos é um forte motivo para construção de sistemas distribuídos. Entende-se por "recurso" um conjunto de objetos que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede. Esses recursos englobam desde componentes de hardware, como discos e impressoras, até entidades definidas pelo software, como arquivos, bancos de dados e objetos de dados de todos os tipos [10].

Ainda, segundo Kshemkalyani [21], um sistema distribuído é um conjunto de entidades independentes que cooperam para resolver um problema que não pode ser resolvido individualmente. Os sistemas distribuídos existem desde o início do universo. De um cardume de peixes a um bando de pássaros e ecossistemas inteiros de micro-organismos, há comunicação entre agentes inteligentes móveis na natureza. Com a proliferação generalizada da Internet e a aldeia global emergente, a noção de sistemas de computação distribuídos como uma ferramenta útil e amplamente implantada está se tornando realidade. Você sabe que está usando um quando a falha de um computador que você nunca ouviu falar o impede de trabalhar [22].

Abaixo são elencados alguns conceitos que exemplificam melhor o que pode ser chamado de Sistema Distribuído:

- Uma coleção de computadores que não compartilham memória comum ou um relógio físico comum, que se comunicam por mensagens que passam por uma rede de comunicação e onde cada computador tem sua própria memória e executa seu próprio sistema operacional. Normalmente, os computadores são semi-autônomos e são pouco acoplados enquanto cooperam para resolver um problema coletivamente.
- Uma coleção de computadores independentes que aparece para os usuários do sistema como um único computador coerente.
- Um termo que descreve uma ampla variedade de computadores, de sistemas fracamente acoplados, como redes de área ampla, a sistemas fortemente acoplados, como redes de área local, a sistemas muito fortemente acoplados, como sistemas de multiprocessadores.

2.2.1 Modelo de Sincronismo

Os modelos de sincronismo se referem aos sistemas onde a noção de tempo os definem, sendo divididos em síncronos, parcialmente síncronos e assíncronos, detalhados abaixo:

Síncrono

Um sistema é dito síncrono se tiver as seguintes propriedades [23]:

- Existe um limite superior conhecido no tempo exigido por qualquer processo para executar uma etapa;
- Todo processo possui um relógio local com uma taxa limitada de desvio conhecida em relação ao tempo real;
- Existe um limite superior conhecido no atraso da mensagem; isso consiste no tempo que leva para enviar, transportar e receber uma mensagem por qualquer link.

Parcialmente Síncrono

De acordo com Hadzilacos [23], Sistemas parcialmente síncronos se referem a um modelo intermediário de sincronia onde pode haver limites conhecidos no desvio do relógio e no tempo de execução de um processo, mas os atrasos nas mensagens podem ser ilimitados; Ou pode haver limites no desvio do relógio, tempo de execução da etapa e atraso da mensagem, mas esses limites podem ser desconhecidos.

Assíncrono

Por fim, os Sistemas são ditos assíncronos, conforme [23], se não houver nenhuma premissa de tempo. Em particular, não há suposições sobre o atraso máximo da mensagem, desvio do relógio ou o tempo necessário para executar uma etapa.

2.2.2 Modelos de Falhas

Em um sistema distribuído, tanto os processos quanto os canais de comunicação podem falhar. O modelo de falhas define como uma falha pode se manifestar em um sistema de forma a proporcionar um entendimento dos seus efeitos e consequências [10]. É importante especificar claramente o modelo de falha porque o algoritmo usado para resolver qualquer problema específico pode variar drasticamente, dependendo do modelo de falha assumido. Para decidir de que tipo de resiliência precisamos, é necessário saber quais tipos de ataques são esperados. Muito disso virá da análise de ameaças específicas ao ambiente operacional do nosso sistema [24].

Um sistema é tolerante a t-falhas se continuar a satisfazer seu comportamento especificado desde que não mais do que t de seus componentes (seja processos, *links* ou uma combinação deles) falhe. O tempo médio entre falhas (MTBF) é geralmente usado para especificar o tempo esperado até a falha, com base na análise estatística do componente/sistema.

Abaixo serão detalhadas as falhas que ocorrem em Sistemas Distribuídos, dividindo-as em falhas de processos e de canais de comunicação com a taxonomia de falhas por omissão, falhas arbitrárias e falhas de sincronização [25].

Falhas por parada

Essas falhas se apresentam quando um processo "trava", ele não executa o serviço para o qual foi designado; isso é conhecido como *crash*. Outros processos podem depender deste que havia parado ou alguns processos podem executar normalmente seus serviços, não precisando de recursos do processo em *crash*.

A detecção destas falhas geralmente ocorre por meio de *timeouts*. Em sistemas assíncronos, quando ocorre um *timeout* quer dizer que o processo pode estar travado, lento ou não ter recebido as requisições por problemas de comunicação. Em sistemas síncronos, uma falha por parada pode ser detectada quando um processo p não responde a um processo q passado o *timeout* configurado para o processo q.

Falhas por omissão

Essas falhas referem-se aos casos em que um processo ou canal de comunicação deixa de executar as ações que deveria. É uma falha semelhante a falha por parada, a diferença é que o processo pode omitir um processamento e depois voltar a processar normalmente. Um exemplo dessa falha pode ser o caso de um servidor falhar ao responder a algumas requisições recebidas, classificando essas omissões em:

- Omissão de recebimento: O servidor falha ao receber alguma mensagem.
- Omissão de envio: O servidor falha ao enviar alguma mensagem.

Outros tipos de falhas por omissão não relacionadas com comunicação podem ser causadas por erros de software tais como laços infinitos ou gerenciamento inadequado da memória.

- **Falhas por parada na comunicação** É uma falha que é percebida na comunicação de uma mensagem de um processo p à um processo q. Podendo ser dividida em duas falhas de comunicação [25].
  - Falhas por omissão de envio: Perda de mensagens entre o processo remetente e o *buffer* de envio.
  - Falhas por omissão da recepção: Perda de mensagens entre o *buffer* de recepção e o processo destino.

Falhas arbitrárias

Em síntese, acontece uma falha arbitrária quando um servidor pode produzir respostas arbitrárias a qualquer momento. Também chamadas de falha Bizantina, neste caso, um servidor pode estar produzindo uma resposta que não havia sido planejada, e que não pode ser detectado como uma falha. Ou mesmo, um servidor pode estar trabalhando em conjunto com outros servidores, a fim de produzir respostas erradas. Uma falha arbitrária de um processo não pode ser detectadas simplesmente pelas respostas as solicitações, pois ele pode omitir a resposta.

Os canais de comunicação podem sofrer dessas falhas, como o conteúdo de uma mensagem pode ser alterado, mensagens podem ser criadas e enviadas ou mensagens podem nem, sequer, serem entregues.

Falhas de temporização

Essas falhas ocorrem quando existem limites de tempos estabelecidos. Essas falhas são aplicáveis em sistemas distribuídos síncronos. Outro exemplo são os Sistemas Operaci-

Tabela 2.1: Falhas em Sistemas Distribuídos [10].

Classe de falha	**Afeta**	**Descrição**
Falha por parada	Processo	O processo para e permanece parado. Outros processos podem detectar esse estado.
Colapso	Processo	O processo para e permanece parado. Outros processos podem não detectar esse estado
Omissão	Canal	Uma mensagem inserida em um *buffer* de envio nunca chega no *buffer* de recepção do destinatário.
Omissão de envio	Processo	Um processo conclui um envio, mas a mensagem não é colocada em seu *buffer* de envio.
Omissão de recepção	Processo	Uma mensagem é colocada no *buffer* de recepção de um processo, mas esse processo não a recebe efetivamente.
Arbitrária (Bizantina)	Processo ou Canal	O processo/canal exibe comportamento arbitrário: ele pode enviar/transmitir mensagens arbitrárias em qualquer momento, cometer omissões; um processo pode parar ou realizar uma ação incorreta.
Relógio	Processo	O relógio local do processo ultrapassa os limites de sua taxa de desvio em relação ao tempo físico.
Desempenho	Processo	O processo ultrapassa os limites do intervalo de tempo entre duas etapas.
Desempenho	Canal	A transmissão de uma mensagem demora mais do que o limite definido.

onais de tempo real. Aplicações multimídia exigem transferência de um grande volume de dados, para distribuir informações sem falhas de temporização pode impor exigências muito especiais sobre o sistema operacional e sobre o sistema de comunicação [10]. A Tabela 2.1 resume essas falhas acima apresentadas. Esses modelos de falhas aplicam-se a sistemas síncronos e assíncronos.

2.2.3 Modelos de Consistência

A consistência se refere a concordância entre os processos em um sistema distribuído em que tais processos trocam informações referentes a lógica das aplicações. A seguir serão tratados os conceitos sobre modelos de consistências, sendo dividido em consistência fraca ou eventual e consistência forte. Os dois modelos de consistência tem benefícios e malefícios que serão detalhados nas seções seguintes. Em geral, a consistência fraca gera maior dificuldade no modelo de programação das aplicações enquanto que na consistência forte, as aplicações se tornam mais simples apesar de ter uma maior latência na rede.

Consistência Fraca ou Eventual

Consistência fraca, também chamada de consistência eventual se refere ao fato de que o sistema irá possuir um período onde valores de propriedades poderão não estar atualizadas em todos os processos. Consequentemente, em algum instante, algumas propriedades podem estar com valores desatualizados. Em uma rede, as informações (e.g., novos fluxos)

são trocadas eventualmente entre os processos. No ambiente de uma rede SDN existe um período de tempo no qual os controladores possuem visões diferentes da rede. Esta abordagem é utilizada pela maioria dos controladores, principalmente por apresentar um bom desempenho.

Consistência Forte

Consistência Forte se refere ao fato de que, após uma atualização de uma propriedade, qualquer processo irá ler o valor atualizado. Em uma rede SDN, qualquer alteração na configuração é informada e sincronizada com todos os controladores, com isso, gera sobrecarga na rede, mas torna as aplicações mais simples. Esta abordagem é utilizada pelos controladores Onix [26], ONOS [5] e SMaRtLight [8]. Usualmente, consistência forte é garantida através da utilização de protocolos de consenso, de forma que todos os processos entram em acordo sobre o estado atual da rede. Os conceitos envolvendo estes protocolos são descritos a seguir:

O problema do acordo bizantino

O problema do acordo bizantino [27] requer que um processo designado, chamado de processo de origem, com um valor inicial, faça com que se chegue a um acordo com os outros processos sobre seu valor inicial, sujeito às seguintes condições:

- **Acordo**: Todos os processos não faltosos devem concordar com o mesmo valor.
- **Validade**: Se o processo de origem não for faltoso, o valor acordado por todos os processos não faltosos deve ser o mesmo que o valor inicial de origem.
- **Término**: Cada processo não faltoso deve, eventualmente, decidir sobre um valor.

Se o processo de origem for faltoso, os processos corretos poderão concordar com qualquer valor. É irrelevante o que os processos faltosos concordam - ou se eles terminam e concordam com qualquer coisa.

Existem dois outros sabores populares do problema do acordo bizantino - o problema de consenso e o problema de consistência interativa.

O problema do consenso

O problema de consenso [28] difere do problema do acordo bizantino onde cada processo tem um valor inicial e todos os processos corretos devem concordar com um único valor. Formalmente, as seguintes propriedades são definidas abaixo:

- **Acordo**: Todos os processos não faltosos devem concordar com o mesmo valor.

- **Validade**: Se um processo decide por um valor, então este valor foi proposto por algum processo.
- **Término**: Cada processo não faltoso deve, eventualmente, decidir sobre um valor.

O problema de consistência interativa

O problema de consistência interativa [29] difere do problema do acordo bizantino em que cada processo tem um valor inicial, e todos os processos corretos devem concordar com um conjunto de valores, com um valor para cada processo. Formalmente:

- **Acordo**: Todos os processos não faltosos devem concordar com a mesma matriz de valores A [V1 ... Vn].
- **Validade**: Se o processo i é não-faltoso e seu valor inicial é Vi, então todos os processos não-faltosos concordam em Vi como o i-ésimo elemento da matriz A. Se o processo j é faltoso, os processos não faltosos podem concordar com qualquer valor para A[j].
- **Término**: Cada processo não faltosos deve decidir sobre o *array* A.

Em geral, os requisitos formais para um protocolo de consenso podem ser:

- **Concordância**: Todo processo não faltoso deve concordar com o mesmo valor.
- **Validação fraca**: Se todo processo não faltoso receber o mesmo valor de entrada, então eles devem todos dar como saída aquele valor.
- **Validação forte**: Para cada processo não faltoso, a sua saída deve ser a entrada de algum processo não falsoso.
- **Término**: Todos processos devem, eventualmente, escolher um valor para dar como saída.

O exemplo a seguir ilustra a dificuldade para se chegar a um acordo, que é inspirado pelas longas guerras travadas pelo Império Bizantino na Idade Média [1]. Quatro acampamentos do exército atacante, cada um comandado por um general, estão acampados em torno do forte de Bizâncio. Eles só irão vencer se atacarem simultaneamente. Por isso, eles precisam chegar a um acordo sobre o tempo de ataque. A única maneira de se comunicar é enviando mensageiros entre si. Os mensageiros modelam as mensagens.

Um sistema assíncrono é modelado por mensageiros que levam um tempo ilimitado para viajar entre dois campos. Uma mensagem perdida é modelada por um mensageiro que é capturado pelo inimigo.

Um processo bizantino é modelado por um general sendo um traidor. O traidor tentará subverter o mecanismo de alcance do acordo, dando informações enganosas aos outros generais. Por exemplo, um traidor pode informar um general para atacar às 10h da manhã e informar aos outros generais para atacar ao meio-dia. Ou ele pode não enviar uma mensagem para algum general. Da mesma forma, ele pode adulterar as mensagens que recebe de outros generais, antes de transmitir essas mensagens.

Um exemplo simples de comportamento bizantino é mostrado na Figura 2.2. Quatro generais são mostrados e uma decisão de consenso deve ser alcançada sobre um valor booleano. Os vários generais estão transmitindo valores potencialmente enganosos da variável de decisão para os outros generais, o que resulta em confusão. Em face desse comportamento bizantino, o desafio é determinar se é possível chegar a um acordo e, em caso afirmativo, em que condições. Se o acordo for alcançável, os protocolos para alcançá-lo precisam ser planejados.

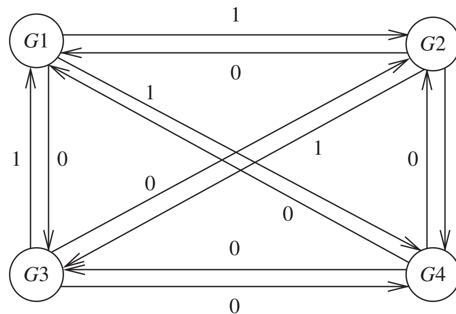

Figura 2.2: Generais bizantinos que enviam mensagens entre si para chegarem em um acordo sobre um valor booleano [1].

2.2.4 Espaço de Tuplas

Um espaço de tuplas [30] é um objeto de memória compartilhada que fornece operações para armazenar e recuperar conjuntos de dados ordenados chamados de tuplas, permitindo a coordenação de processos de um sistema distribuído desacoplada no espaço (os processos não precisam conhecer as localizações uns dos outros) e no tempo (os processos não precisam estar ativos ao mesmo tempo). Uma tupla t é uma sequência ordenada de campos, onde um campo que contém um valor é dito definido. Um tupla onde todos os campos são definidos é chamada de entrada. Uma tupla $\bar{t}$ é chamada molde (ou *template*) se algum de seus campos não tem valor definido. Diz-se que uma tupla t e um molde $\bar{t}$ combinam se e somente se ambos têm o mesmo número de campos e todos os valores e tipos dos campos definidos em $\bar{t}$ são iguais aos valores e tipos dos campos correspondentes em t. Por exemplo, a tupla $\langle \text{Jefferson}, Unb, \text{Mestrado} \rangle$ combina com o molde $\langle \text{Jefferson}, Unb, * \rangle$ (‘$*$’ denota um campo sem definição do molde).

Um espaço de tuplas funciona como uma memória associativa (os dados são acessados a partir de seu conteúdo, e não através de seu endereço), sendo manipulado através das seguintes operações [30]: $out(t)$ que adiciona a entrada t no espaço de tuplas; $in(\bar{t})$, que remove do espaço de tuplas uma tupla que combina com o molde $\bar{t}$; $rd(\bar{t})$, usada na leitura de uma tupla que combina com o molde $\bar{t}$, sem removê-la do espaço.

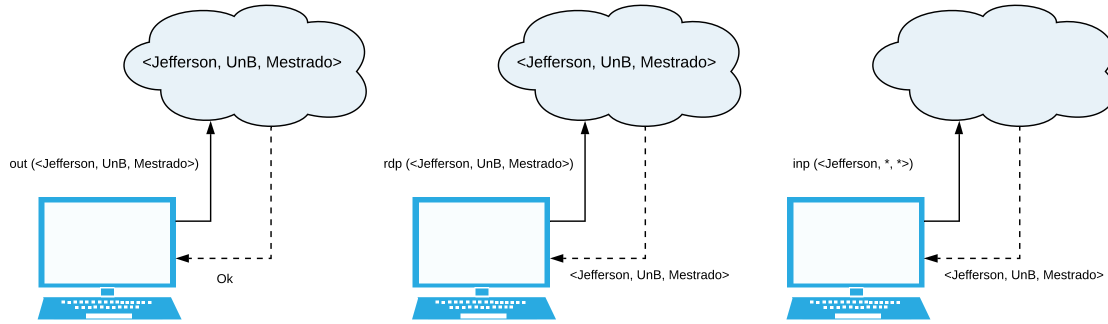

Figura 2.3: Operações básicas do Espaço de Tuplas.

As operações *in* e *rd* são bloqueantes, i.e., se não houver uma tupla que combine com o molde no espaço, o processo fica bloqueado até que uma esteja disponível. Também existem variantes não bloqueantes das operações de leitura, denominadas *inp* e *rdp*, que retornam nulo quando não existe uma tupla que combine com o molde.

A Figura 2.3 ilustra essas operações, mostrando a operação enviada pelo cliente, as respostas dos servidores e o estado final do espaço da tupla. Primeiramente, é ilustrado uma operação de escrita, em que é adicionado ao Espaço de tuplas a tupla <Jefferson, Unb, Mestrado> através da operação *out*. Em seguida, é feita uma leitura de uma tupla que combine com o molde passado para a operação *rdp*(<Jefferson, Unb, Mestrado>). Por fim, é feito a remoção de uma tupla que combine com o molde passado para a operação *inp*(<Jefferson, *, *>).

Uma extensão comum a este modelo é a inclusão das seguintes operações [2, 31, 32, 33]: $cas(\bar{t}, t)$ que insere t no espaço se não tiver uma tupla t' que combine com o molde $\bar{t}$ e retorna nulo, caso contrário retorna t'; $replace(t, t')$ que insere a tupla t' e remove (e retorna) a tupla t caso t esteja no espaço, caso contrário apenas retorna nulo; e $readAll(\bar{t})$ que retorna todas as tuplas que combinem com o molde $\bar{t}$.

DepSpace

No contexto de um espaço de tuplas, os seguintes atributos são necessários para segurança das operações: **confiabilidade** (as operações realizadas no espaço de tuplas fazem com que seu estado se modifique de acordo com a especificação), **disponibilidade** (o espaço

de tuplas sempre está pronto para executar as operações requisitadas por partes autorizadas), **integridade** (nenhuma alteração imprópria no estado de um espaço de tuplas pode ocorrer) e **confidencialidade** (o conteúdo dos campos das tuplas não pode ser revelado a partes não autorizadas). Segurança é uma característica fundamental de sistemas confiáveis [19]. O DepSpace [2, 15, 16] consiste na implementação de um espaço de tuplas que busca satisfazer estas propriedades por meio de diversas camadas (Replicação, Confidencialidade, Controle de acesso), descritas abaixo e ilustradas na Figura 2.4, que demonstra a comunicação entre um cliente e o DepSpace, onde um cliente interage com o sistema através da camada proxy, chamando funções com as assinaturas usuais das operações do espaço de tupla (i.e., out(t), rdp(t), inp(t)); logo abaixo exite uma camada responsável pela manipulação do controle de acesso no nível da tupla. Depois, há uma camada que cuida da confidencialidade e em seguida, uma que lida com a replicação. Já no lado do servidor é semelhante, exceto que há uma nova camada para verificar a política de acesso para cada operação solicitada.

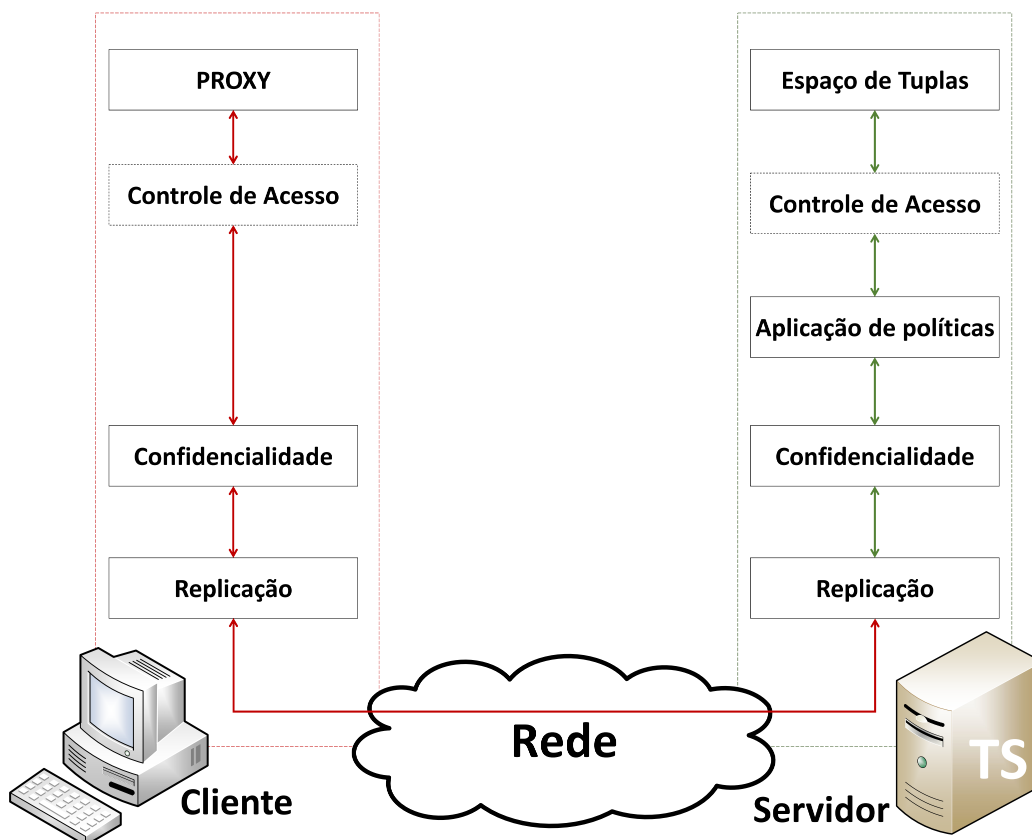

Figura 2.4: Arquitetura DepSpace, adaptado de [2]

.

Replicação. Para manter a consistência do espaço de tuplas, o DepSpace utiliza replicação de Máquina de Estados [34, 35, 36]. Este mecanismo está relacionado principalmente com as propriedades de disponibilidade e confiabilidade, pois para um número n de réplicas no sistema, garante que o espaço de tuplas executa as operações a ele endereçadas seguindo sua especificação, mesmo que até $f = (n-1)/3$ réplicas sejam maliciosas (as réplicas corretas *mascaram* o comportamento das maliciosas). Através deste protocolo, as réplicas corretas executam a mesma sequência de operações e retornam os mesmos valores, evoluindo de forma sincronizada.

Confidencialidade. Para evitar pontos únicos de falha, a preservação da propriedade de confidencialidade não é atribuída a um único servidor, mas a um conjunto deles. Sendo assim, a confidencialidade é implementada através do uso de um $(n,f+1)$ – esquema de compartilhamento de segredo publicamente verificável (*publicly verifiable secret sharing* – PVSS) [37]. Através deste esquema, os clientes cifram as tuplas com um segredo por eles gerado e geram um conjunto de n fragmentos (*shares*) deste segredo. O segredo pode ser remontado apenas com a combinação de $f + 1$ *shares*, o que torna impossível que um conluio de até f servidores faltosos revele o conteúdo de uma tupla. Como os servidores não conseguem acessar o conteúdo da tupla, que está cifrado, o DepSpace utiliza um *fingerprint* da tupla para possibilitar a comparação entre tuplas e moldes. Este *fingerprint* é computado de acordo com os seguintes tipos de campos escolhidos para a tupla [2, 15, 16]:

- Público (PU): o próprio valor do campo é o *fingerprint*, i.e., nenhum método criptográfico é aplicado ao conteúdo do campo que fica exposto.
- Comparável (CO): um *hash* do valor do campo é o *fingerprint* (para isso utiliza uma função de *hash* resistente a colisões), possibilitando a execução de buscas (comparações). Estes campos são implementados através do algoritmo SHA-1, que gera um *hash* de 20 *bytes*.
- Comparável Determinístico (CD): o conteúdo do campo é cifrado, através da função $encryptCD(key_{shared}, content)$, com um algoritmo determinístico de criptografia simétrica [38] e o resultado é usado como *fingerprint*. Por ser determinístico, este tipo de campo permite comparações nos servidores. A mesma chave deve ser compartilhada entre os clientes e é usada tanto para cifrar quanto para decifrar os dados. Estes campos são implementados através do algoritmo HMAC-SHA256 (*Hash-based Message Authentication Code with SHA-256*) e uma chave secreta de 256 *bits* para gerar uma saída pseudo-aleatória de 32 *bytes*.

- Ordenável (OR): neste caso, o conteúdo do campo é cifrado por meio da função $encryptOR(key_{shared}, content)$, com um algoritmo de criptografia simétrica que preserva a relação de ordem nas cifras [39, 40, 41] e o resultado é utilizado como *fingerprint*. Este tipo de campo permite que servidores definam a relação entre duas tuplas comparando a ordem de seus campos, sem acessar o conteúdo das tuplas. A mesma chave deve ser compartilhada entre os clientes e é usada tanto para cifrar quanto para decifrar os dados. Para implementar este tipo de campo, foi utilizado o algoritmo de *Order Revealing Encryption* [41]. Este algoritmo, baseado em AES, ao invés de preservar a ordem, gera textos cifrados não determinísticos que não podem ser comparados diretamente, mas sim submetidos a uma função de comparação que retorna a relação entre eles. Para este algoritmo foram utilizadas chaves simétricas de 128 *bits*.

- Operável (OP): para este campo é utilizado um par de chaves, sendo que o conteúdo do campo é cifrado por meio da função $encryptOP(key_{public}, content)$, utilizando um algoritmo homomórfico [42] ou um parcialmente homomórfico [43] e o resultado é usado como *fingerprint*. Este tipo de campo permite operações nos servidores (ex.: somas e multiplicações) acessando apenas os conteúdos cifrados. Como operações podem ter sido executadas, o *fingerprint* pode divergir da tupla a qual se refere, portanto, ao ler uma tupla e reconstruí-la através do esquema PVSS, o cliente deve atualizar o valor destes campos com os valores contidos no *fingerprint*. A chave privada deve ser compartilhada entre os clientes e é usada para decifrar os dados. Para implementar estes campos, foi utilizada a biblioteca *javallier* [44], que é uma implementação em Java do algoritmo de Paillier [45]. Esta biblioteca de criptografia parcialmente homomórfica permite a adição entre dois valores cifrados e através desta operação pode-se derivar a subtração. Adicionalmente, um valor cifrado pode ser multiplicado por um valor em claro usando repetidas operações de adição. Para este algoritmo assimétrico utilizamos um par de chaves (pública e privada) de 3072 *bits*, de modo a obter um nível de segurança de 128 *bits* [46].

- Privado (PR): um símbolo especial é o *fingerprint*, i.e., o conteúdo deste campo é mantido cifrado sem qualquer possibilidade de realização de comparações ou acesso aos dados.

A Figura 2.5 apresenta a função usada para computar o *fingerprint* $t_h = \langle h_1, ..., h_m \rangle$ de uma tupla $t = \langle f_1, ..., f_m \rangle$, de acordo com a classificação apresentada. Vale destacar que o mesmo procedimento é usado nos *templates*, permitindo a verificação se um *template* combina com uma tupla [2]. Esta classificação fornece a seguinte ordem para os campos, de acordo com o nível de segurança [15, 16]: PU $<$ CO $<$ CD $<$ OR $<$ OP $<$ PR. Esta

ordem está relacionada com a quantidade de informação revelada no *fingerprint* (ex.: a ordem entre os campos de duas tuplas).

$$h_i = \begin{cases} * & \text{if } f_i = * \\ f_i & \text{if } f_i \text{ is PU} \\ hash(f_i) & \text{if } f_i \text{ is CO} \\ encryptCD(key_{shared}, f_i) & \text{if } f_i \text{ is CD} \\ encryptOR(key_{shared}, f_i) & \text{if } f_i \text{ is OR} \\ encryptOP(key_{public}, f_i) & \text{if } f_i \text{ is OP} \\ \text{PR} & \text{if } f_i \text{ is PR} \end{cases}$$

Figura 2.5: Cálculo do *fingerprint*.

Controle de Acesso O controle de acesso é um mecanismo fundamental para manutenção da integridade e confidencialidade das informações (tuplas) armazenadas no DepSpace, pois previne que clientes não autorizados obtenham acesso as tuplas, além de impedir que clientes faltosos saturem o espaço de tuplas enviando uma grande quantidade de tuplas. O DepSpace implementa controle de acesso de duas formas:

- **Baseado em credenciais:** para cada tupla inserida no DepSpace pode-se definir quais são as credenciais necessárias para acessá-la, tanto para leitura quanto para remoção (acesso em nível de tuplas). Estas credenciais são definidas pelo processo que insere a tupla. Também é possível definir, quando o espaço de tuplas é criado, quais são as credenciais necessárias para inserir uma tupla no espaço (acesso em nível de espaço). A implementação desta funcionalidade é realizada através da associação de listas de controle de acesso a cada espaço e tupla.
- **Políticas de granularidade fina:** o DepSpace suporta a definição de políticas de acesso de granularidade fina [47], que devem ser especificadas no momento da criação do espaço de tuplas. Estas políticas controlam o acesso ao espaço e são definidas considerando três parâmetros: o identificador do cliente, a operação que será executada (juntamente com seus argumentos) e o estado atual do espaço de tuplas.

2.3 *Software Defined Network* (SDN)

Redes Definidas por Software (do Inglês, *Software Defined Network*) é um novo paradigma em infraestrutura de redes que traz o conceito de virtualização da infraestrutura de redes. As redes SDN trouxeram a capacidade de se ter uma rede programável e gerenciável, através desta característica, obtêm-se uma visão do estado geral da rede e das aplicações.

Segundo [48], Redes definidas por software (SDN) é uma abordagem arquitetônica que otimiza e simplifica as operações de rede, vinculando mais de perto a interação entre aplicativos e serviços de rede e dispositivos, se eles são reais ou virtualizados. Muitas vezes é alcançado empregando um ponto de controle de rede logicamente centralizado — que muitas vezes é realizado como um controlador SDN — que então orquestra, media e facilita a comunicação entre aplicativos que desejam interagir com elementos de rede e elementos de rede que desejam transmitir informações para esses aplicativos. Em seguida, o controlador expõe e abstrai as funções e operações da rede através de interfaces programáticas modernas, amigáveis e bidirecionais.

A SDN desacopla dos dispositivos físicos de rede tradicional o controle e a decisão de encaminhamento dos fluxos de dados, abstraindo estas duas funcionalidades em planos que constituem camadas distintas dissociadas dos dispositivos de hardware de rede criando um novo conceito, protocolos e terminologia de infraestrutura de redes. Isso é semelhante com aquilo que ocorreu com o advento de virtualização de máquinas, onde o conhecimento em servidores e estações de trabalho teve que ser expandido para uma nova forma abstraída de lidar com estes dispositivos, configurá-los, disponibilizá-los, contingenciá-los, garantindo segurança e consistência dos dados neles hospedados. Os planos de que trata a SDN em sua proposta original são divididos em plano de dados, plano de controle e plano de aplicação, ordenados da camada mais baixa para a mais alta de abstração.

Uma rede SDN pode ser definida como uma arquitetura baseada em quatro pilares [49]:

1. Os planos de controle e de dados são desacoplados, o que resulta em dispositivos de rede se tornando elementos de encaminhamento simples (pacotes), regidos por um controlador.

2. As decisões de encaminhamento são baseadas em fluxo. No contexto SDN, um fluxo é uma sequência de pacotes entre uma origem e um destino, sendo que todos recebem políticas de serviço idênticas nos dispositivos de encaminhamento.

3. A lógica de controle é gerenciada por um controlador.

4. A rede é programável por meio de aplicativos de *software* executados sobre o controlador que interagem com os dispositivos de plano de dados subjacentes.

A arquitetura SDN separa as camadas de controle e dados dos dispositivos fazendo com que o componente de *software* (controlador) fique responsável pelo gerenciamento da rede (plano de controle), enquanto cabe ao *hardware* o transporte dos pacotes (encaminhamento) de acordo com as informações que estão definidas pelo *software* (controlador).

Com isso, a arquitetura SDN oferece abstração, simplificando o gerenciamento da rede e reduzindo os custos e a complexidade de hardware. Com esse nível de abstração, ganha-se um ambiente onde o tráfego, roteamento e controle de acesso podem ser implementados

de forma rápida e eficiente, sem a necessidade da criação de políticas aplicadas a baixo nível.

Esta abordagem também reduz a complexidade dos dispositivos da rede, uma vez que apenas uma simples interface é exposta para o administrador realizar as configurações, liberando-o de atuar manualmente em dispositivos de diferentes fornecedores. A arquitetura SDN simplifica significativamente as modificações na lógica de controle de rede (como é centralizada), permite que os planos de dados e controle evoluam e sejam escalonados independentemente, e diminui o custo dos elementos do plano de dados.

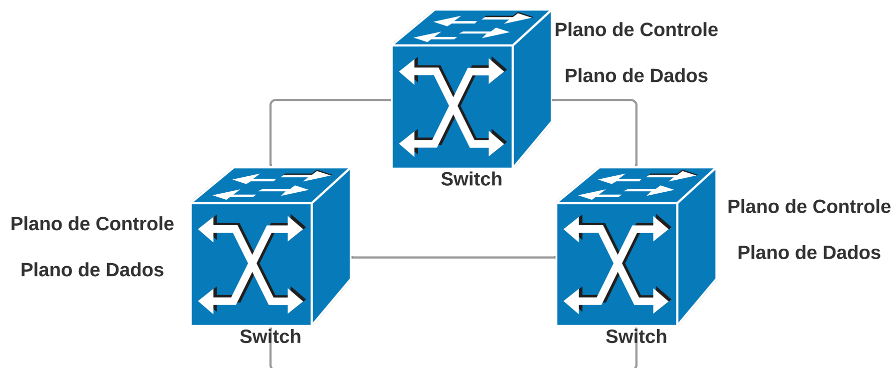

Figura 2.6: Rede tradicional.

A Figura 2.6 representa a infraestrutura tradicional de uma rede composta de dois planos i.e., um plano de dados e outro de controle. Conforme observamos na figura, o plano de controle é responsável pela lógica de roteamento e políticas de redes que foram definidas pelo administrador de rede e o plano de dados, responsável pelo encaminhamento de fluxos de dados e não possui nenhuma "inteligência" sendo capaz apenas de receber dados e encaminhar de acordo com sua tabela de encaminhamento.

Para ilustrar as mudanças trazidas pela SDN, a Figura 2.7 representa a arquitetura de uma rede SDN, na qual o conceito de separação de camadas é visível, verificando o plano de controle, responsável por toda lógica, políticas e segurança de rede e o plano de dados, responsável pelo encaminhamento dos dados. Com essa abordagem de separação de responsabilidade cada camada faz sua função de maneira mais transparente e problemas específicos de cada camada podem ser sanados pelos administradores de redes de maneira mais focada.

A arquitetura SDN é divida em plano de dados e plano de controle na qual os controladores coletam informações dos *switches*, calculam e distribuem as decisões de encaminhamento apropriadas ao plano de dados. Controladores e *switches* usam um protocolo aberto e padrão [9] para se comunicar e trocar informações.

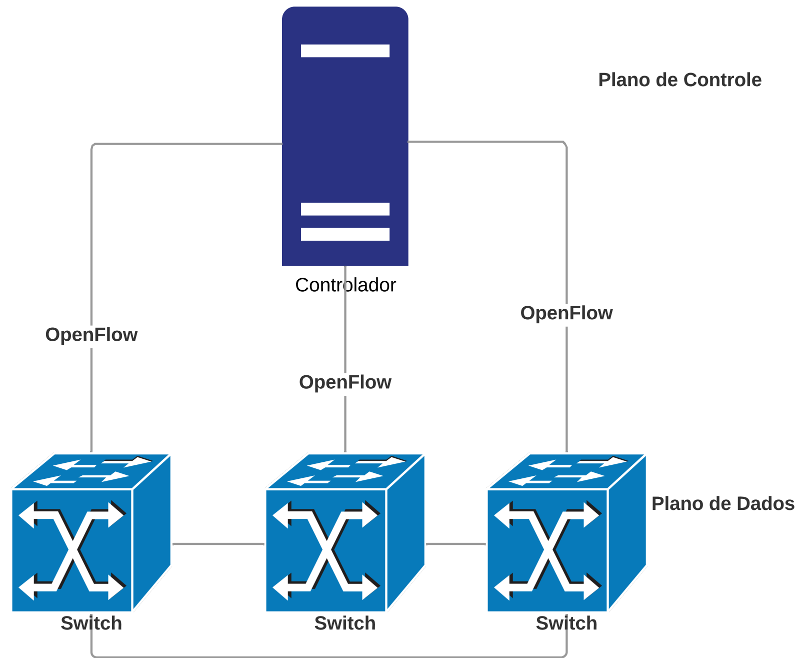

Figura 2.7: Rede SDN.

Os *switches* são modelados como uma tabela de fluxo onde contam com três colunas: regras, ações e contadores. A coluna de regras define o fluxo. As regras são comparadas com os cabeçalhos dos pacotes recebidos. Se uma regra corresponder, as ações correspondentes serão aplicadas ao pacote e os contadores na coluna do contador serão atualizados. A plataforma de controle desacoplada facilita a tarefa de modificar a lógica de controle de rede e fornece uma interface programática em que os desenvolvedores podem criar uma ampla variedade de novos protocolos e aplicativos de gerenciamento. Nesse modelo, os planos de dados e controle podem evoluir e escalar independentemente, enquanto o custo dos elementos do plano de dados é reduzido. A plataforma emergente OpenFlow [9] permite que os *switches* encaminhem o tráfego no plano de dados com base nas regras instaladas por um controlador.

Nas próximas seções serão detalhados os componentes e comportamentos fundamentais dos planos de dados, controle e aplicações.

Plano de Dados

Esse plano, também conhecido como camada de Infraestrutura é composto por equipamentos tradicionais de rede como *switches* e roteadores, sendo que esses dispositivos trabalham como simples elementos de encaminhamento sem controle ou software incorporado para tomar decisões autônomas. A inteligência de rede é removida dos dispositivos do plano de dados para um sistema de controle centralizado logicamente, isto é, o sistema

operacional e os aplicativos de rede, conforme mostrado na Figura 2.7. Os dispositivos de encaminhamento que compõem essa camada possuem conjuntos de instruções bem definidos (e.g., regras de fluxo) usados para executar ações nos pacotes recebidos (e.g., encaminhar para portas específicas, encaminhar para o controlador, reescrever algum cabeçalho). Essas instruções são instaladas nos dispositivos de encaminhamento pelos controladores SDN [49].

Plano de Controle

O plano de controle pode ser visto como o "cérebro da rede". Toda a lógica de controle repousa nos aplicativos e controladores, que formam o plano de controle. Nesta camada encontra-se o Controlador ou Sistema Operacional de Rede, do inglês *Network Operating System* (NOS), no qual, em uma arquitetura SDN, é a peça principal de suporte para a lógica de controle por gerar a configuração da rede com base nas políticas definidas pelos administradores de rede. Semelhante a um sistema operacional tradicional (i.e., Windows, Linux, Android), a plataforma de controle abstrai os detalhes de nível inferior de conexão e interação com dispositivos de encaminhamento. De forma similar aos sistemas operacionais elencados acima, o valor crucial de um NOS é fornecer abstrações, serviços essenciais e interfaces de programação de aplicativos comuns (APIs) para desenvolvedores.

Funcionalidades genéricas como estado da rede e informações de topologia de rede, descoberta de dispositivos e distribuição da configuração de rede podem ser fornecidas como serviços do NOS. Com os NOSs, para definir políticas de rede, um desenvolvedor não precisa mais se preocupar com os detalhes de baixo nível da distribuição de dados entre os elementos de roteamento, por exemplo.

De acordo com Kreutz [49], é possível que esses sistemas criem um novo ambiente capaz de promover a inovação em um ritmo mais rápido, reduzindo a complexidade inerente à criação de novos protocolos de rede e aplicativos de gerenciamento.

Plano de Aplicação

O plano de aplicação inclui os aplicativos como roteamento, *firewalls*, balanceadores de carga, monitoramento e assim por diante. Essencialmente, um aplicativo de gerenciamento define as políticas que são traduzidas em instruções específicas que programam o comportamento dos dispositivos de encaminhamento.

A Figura 2.8 ilustra as três camadas e a relação entre elas, através da interface *Northbound*, responsável por conectar a camada de aplicação à camada de controle e a interface *Southbound*, responsável pela ligação da camada de controle com a camada de dados.

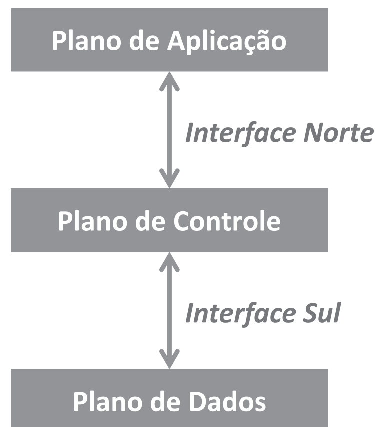

Figura 2.8: Camadas SDN.

OpenFlow

O OpenFlow é um protocolo padrão responsável pela comunicação entre o plano de controle e o plano de dados de uma rede SDN.

Em 2011, um consórcio sem fins lucrativos chamado *Open Networking Foundation* (ONF) foi formado por um grupo de prestadores de serviços para comercializar, padronizar e promover o uso do OpenFlow nas redes de produção [48].

Os principais componentes do modelo OpenFlow que ficaram definidos foi o uso de um protocolo padronizado entre o controlador e um agente no elemento de rede para instanciar o estado e a capacidade de fornecer a programação da rede a partir de uma visão centralizada através de uma API (do inglês, *Application Programming Interface*) extensível e moderna. OpenFlow, além das suas especificações, é um protocolo que permite que os administradores possam definir fluxos de dados e determinar quais caminhos esses fluxos devem percorrer independente do hardware em si (plano de dados). Com isso, o controle de tráfego e roteamento são retirados dos dispositivos de hardware e repassado para o plano de controle, através da interface sul, no qual os controladores tomam as decisões com base nas decisões do administradores de rede, usuários ou aplicações, através da interface norte. Com isso, ganha-se desempenho e segurança pois o tráfego diminui e as políticas são aplicadas de maneira gerenciada.

As tabelas de fluxo são configuradas nos *switches* e os controladores se comunicam com os *switches* por meio do protocolo OpenFlow, que impõe políticas sobre fluxos. As políticas podem permitir que os *switches* configurem caminhos através da rede para características específicas, como velocidade, menor número de saltos ou latência reduzida. O tráfego flui

pela rede em caminhos predefinidos pelo controlador e aplicados pelos *switches*.

A tabela de fluxo contém um conjunto de entradas de fluxos, em que cada entrada tem uma ação associada a ela, tal como encaminhamento, encapsulamento e descarte. Todos os pacotes processados pelo *switch* são comparados com esta tabela. Se uma entrada correspondente for encontrada, uma determinada ação para essa entrada será executada, como por exemplo, a ação de encaminhar um pacote para uma porta específica [9]. Se nenhuma correspondência for encontrada, o pacote é encapsulado e encaminhado para o controlador através do canal seguro, que conecta o *switch* ao controlador. O controlador é responsável por programar todas as regras de correspondência e encaminhamento de pacotes no *switch*, como por exemplo, manipular os pacotes sem entrada de fluxo correspondente e gerenciar a tabela de fluxo do *switch*, realizando a ação de adicionar ou remover entradas de fluxo. A ação de descarte dos pacotes pode ser usada para segurança como por exemplo, reduzir ataques de negação de serviço [9] [50].

O OpenFlow é o protocolo mais utilizado para realizar a comunicação dos planos de dado e controle, sendo a escolha ideal por ser simples e de grande utilização pela comunidade de redes, sendo assim, é viável enumerar seus principais problemas de segurança.

Abaixo é apresentado uma análise de segurança do protocolo OpenFlow, enumerando suas vulnerabilidades, utilizando a metodologia STRIDE [51], o qual utiliza mnemônicos para cada categoria de ataque, a saber:

- ***Spoofing*:** Também conhecido como falsificação, o *Spoofing* é um tipo de ataque no qual um atacante se passa por outro *host* ou usuário com o objetivo de roubar dados, disseminar *malware* ou contornar controles de acesso.
- ***Tampering*:** Também conhecido como violação, o objetivo desse ataque é permitir que o atacante intercepte e modifique os dados de uma aplicação através do canal de transmissão.
- ***Repudiation*:** O atacante consegue efetuar qualquer ação na aplicação sem que seja atribuído como autor das mesmas. Também é conhecido como repúdio.
- ***Information Disclosure***: Também conhecido como divulgação de informações, o objetivo do ataque é explorar o uso da agregação de fluxo para descobrir algum aspecto do estado da rede que, de alguma forma, não seria visível para um invasor. Essas informações podem ser usadas por um invasor para determinar a presença e a natureza dos serviços em uma rede. Este conhecimento também pode ser usado em um estágio posterior de um ataque.
- ***Denial of Service*:** Um atacante consegue impossibilitar que os utilizadores legítimos de uma aplicação ou serviço tenham acesso aos mesmos. Esse tipo de ataque é conhecido como ataque de negação de serviços.

- ***Elevation of Privilege***: Também conhecido como elevação de privilégios, um atacante tem possibilidade de ganhar privilégios elevados de acesso por meio não autorizado.

2.4 Considerações Finais

Este capítulo apresentou os principais conceitos sobre segurança computacional, envolvendo suas principais propriedades (confidencialidade, integridade e disponibilidade). Também discutiu conceitos envolvendo sistemas distribuídos e os modelos de falhas existentes. Além disso, foram apresentados os modelos de consistência, subdividindo-os em consistência forte/alta e consistência eventual/fraca, no qual os sistemas distribuídos utilizam para armazenar e recuperar informações de acordo com a política escolhida.

Também foi apresentado o conceito de Espaço de Tuplas e o DepSpace, uma implementação de espaço de tuplas com as propriedades de segurança, que será bastante explorado na pesquisa. Por fim, foi apresentado as Redes Definidas por Software (SDN), detalhando suas camadas e interações por meio do protocolo Openflow e descrevendo suas principais vulnerabilidades.

Estes conceitos são fundamentais para esta pesquisa, pois a segurança em SDN será baseada em múltiplos controladores distribuídos, que usará os aspectos relacionados a confidencialidade, integridade e disponibilidade por meio dos sistemas distribuídos, utilizando consistência forte para manter o estado da rede.

No capítulo a seguir será discutido com maior detalhe os conceitos de controladores e múltiplos controladores para manter uma infraestrutura confiável para redes SDN.

Capítulo 3

Revisão do Estado da Arte

Este capítulo apresenta a revisão do estado da arte, iniciando pelos controladores que foram divididos em centralizados e distribuídos. No final ainda aborda questões de segurança envolvendo o plano de controle de uma rede SDN, discutindo problemas de segurança e os principais desafios associados a estes problemas.

3.1 Controladores

Para um dispositivo funcionar de maneira adequada, é necessário um meio de gerenciar os recursos do hardware de tal dispositivo, como memória, disco rígido, CPU. Tal meio é o representado pleo sistema operacional que facilita a vida dos desenvolvedores de sistemas. Exemplo disso, temos o Sistema operacional Windows, Linux, Android, IOS. Em se tratando do ambiente de redes de computadores, existem os Sistemas Operacionais específicos para dispositivo de encaminhamento, e na maior parte das vezes, esses sistemas são proprietários de cada fabricante e não existe um padrão unificado para esses sistemas.

A SDN veio trazer a ideia de sistemas operacionais abertos, abstraindo características específicas de dispositivos e fornecendo, de forma transparente, funcionalidades comuns para qualquer dispositivo. A SDN está prometida para facilitar a gestão da rede e facilitar o trabalho de resolução de problemas de rede através do controle logicamente centralizado oferecido por um *Network Operating System* (NOS). Com isso, os NOS fornecem serviços essenciais aos desenvolvedores, funcionalidades genéricas como topologia da rede, estado da rede, descoberta de dispositivos e funcionalidades de baixo nível, facilitando ao administrador de rede a definição das políticas da rede. Ou seja, os NOS ou Controlador é uma plataforma de controle que abstrai os detalhes de nível inferior, sendo encarregado por criar funcionalidades para realizar as decisões operacionais de como o sistema deverá se comportar [52]. Ele exerce um controle direto sobre os elementos do plano de dados, através de sua programação em uma linguagem de alto nível.

Nas seções seguintes serão descritas as características e tipos das plataformas de controle em uma rede SDN. Além disso, é apresentada uma Tabela 3.1 comparando as principais diferenças entre as abordagens discutidas.

Tabela 3.1: Comparativo das Arquiteturas de Controladores SDN.

Arquitetura	**Vantagens**	**Desvantagens**
Centralizada	Simplicidade	Ponto único de falhas Limite de escalabilidade
Distribuída	Tolerância a falhas Escalabilidade	Complexidade

3.1.1 Controladores Centralizados

Um controlador centralizado é responsável por gerenciar todos os dispositivos do plano de dados em uma rede. Centralizando toda a lógica, fica mais fácil configurar a rede e o analista de rede terá maior facilidade em reparar defeitos ou problemas advindos de várias partes da infraestrutura. Esse projeto foi inicialmente proposto por uma questão de simplicidade.

A Figura 3.1 ilustra este cenário no qual existem três *sites* que devem ser gerenciados por um plano de controle, sendo que a plataforma de controle está centralizada e localizada em um dos *sites*, portanto, responsável por gerir, além de seu *site*, os outros dois.

Os controladores NOX [4], NOX-MT [53], Maestro [54], Beacon [55], POX [56], Ryu [57], Floodlight [58] e Meridian [59] são exemplos de controladores centralizados. Abaixo são descritas as suas principais características e arquitetura.

NOX

O **NOX** [4], desenvolvido inicialmente pela Nicira Networks e agora de propriedade da VMware, foi o primeiro controlador do OpenFlow (OF) sendo introduzido pela primeira vez na comunidade em 2009. Ele serve como uma plataforma de controle de rede, que fornece uma interface programática de alto nível para gerenciamento e desenvolvimento de aplicativos de controle de rede. Possui como principal objetivo permitir o gerenciamento da rede baseado em parâmetros de alto nível, como nome de usuário ou de servidor, ao invés de parâmetros de baixo nível como endereços MAC e IP. A Figura 3.2 mostra os principais componentes de uma rede baseada em NOX: um conjunto de *switches* e um ou mais servidores conectados à rede. O software NOX (e os aplicativos de gerenciamento executados no NOX) é executado nos servidores.

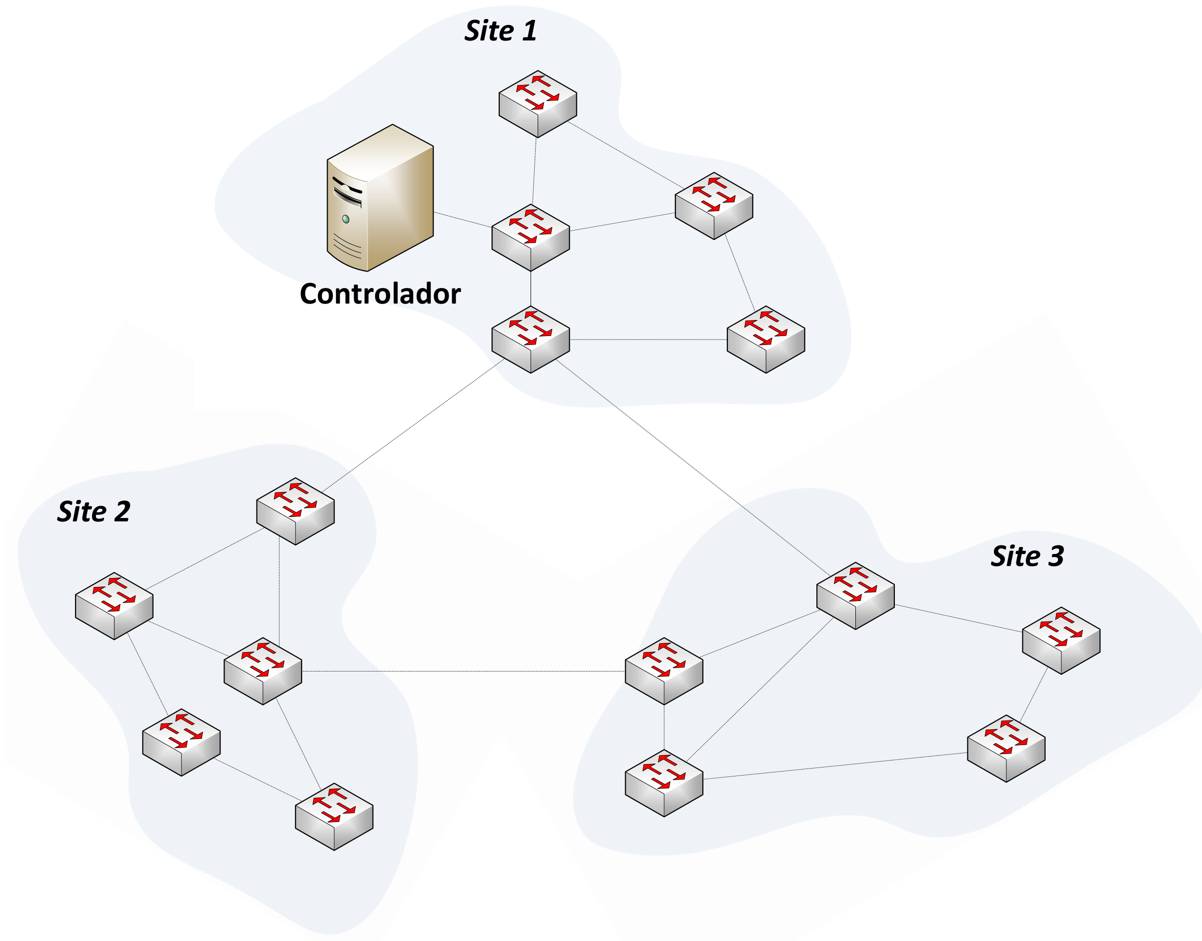

Figura 3.1: Arquitetura SDN com um controlador centralizado [3].

NOX-MT

O NOX-MT [53] é um sucessor *multithreaded* ligeiramente modificado do NOX que utiliza técnicas conhecidas de otimização, incluindo: *batch* de E/S para minimizar a sobrecarga de E/S, manipulação de E/S para a biblioteca ASIO (*Boost Asynchronous I/O*), e usa uma implementação *malloc* rápida que reconhece o multiprocessamento e se adapta bem a uma máquina com vários núcleos. Essas otimizações permitem que o NOX-MT supere o NOX em um fator de 33 vezes em um servidor com dois processadores quad-core de 2 GHz [53]. O NOX-MT foi o primeiro esforço para melhorar o desempenho do controlador e motivou outros controladores a implementar técnicas semelhantes para melhorar o desempenho.

Maestro

Uma característica fundamental de uma rede OpenFlow é que o controlador é responsável pelo estabelecimento inicial de cada fluxo entrando em contato com os *switches* relacio-

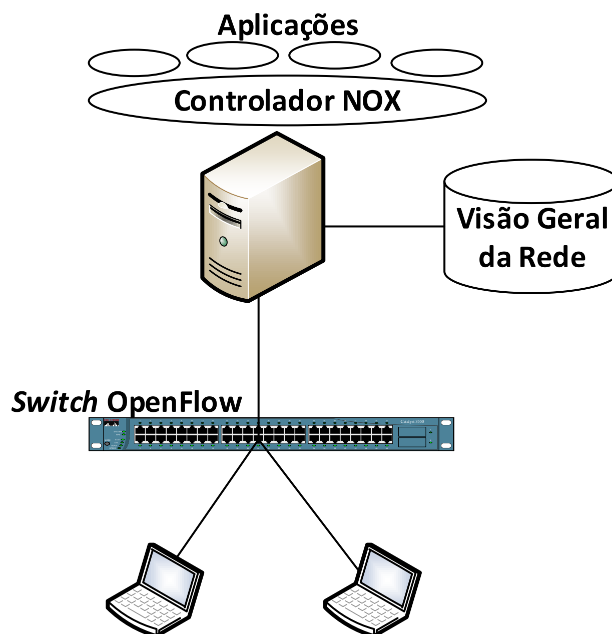

Figura 3.2: Componentes de uma rede baseada em NOX: comutadores OpenFlow (OF), um servidor executando um processo do controlador NOX e um banco de dados que contém a visão da rede [4].

nados. Assim, o desempenho do controlador pode ser o gargalo. Em relação a isso, o Maestro [54] é um sistema operacional de rede para orquestrar aplicativos de controle de rede que visa a melhoria do desempenho, em relação ao controlador NOX, base de seu projeto. O Maestro fornece interfaces para implementar aplicativos de controle de rede modulares para acessar e modificar o estado da rede e coordenar suas interações. O Maestro é uma plataforma para alcançar funções de controle de rede automáticas e programáticas usando esses aplicativos modularizados. O Maestro mantém um modelo simples de programação *single-thread* para programadores de aplicativos do sistema, e ainda permite e gerencia o paralelismo como um serviço para programadores de aplicativos, explorando tal paralelismo dentro de uma única máquina para melhorar o desempenho da taxa de transferência do sistema.

Experimentalmente, a taxa de transferência do Maestro pode alcançar escalabilidade quase linear em uma máquina com oito processadores [54]. O Maestro é portátil e escalável sendo desenvolvido na plataforma Java ele é altamente portátil para vários sistemas operacionais e arquiteturas.

Beacon

O Beacon [55] é um controlador OpenFlow de código aberto baseado em Java criado em 2010. Ele tem sido amplamente usado para ensino, pesquisa e é a base do controlador

Floodlight. O Beacon fornece uma estrutura para controlar dispositivos de rede usando o protocolo OpenFlow e um conjunto de aplicativos internos que fornecem a funcionalidade geralmente necessária do plano de controle.

O Beacon explorou novas áreas do espaço de projeto do controlador OpenFlow, com foco em ser amigável ao desenvolvedor, apresentar alto desempenho e ter a capacidade de iniciar e parar aplicativos existentes e novos em tempo de execução.

POX

É um controlador OpenFlow derivado do NOX. O POX [60] é um controlador SDN de código aberto escrito na linguagem de programação Python. Ele é usado no desenvolvimento e prototipação de maneira rápida de novos aplicativos de rede. Uma característica interessante do POX é o fato de ele poder transformar dispositivos OpenFlow "burros" em *hubs*, *switches*, balanceadores de carga, dispositivos de *firewall* entre outros. O controlador POX permite uma maneira fácil de executar experimentos OpenFlow/SDN. O POX pode receber parâmetros diferentes de acordo com topologias reais ou experimentais, permitindo executar experimentos em hardware real, bases de teste ou emuladores.

Embora nenhum controlador SDN seja particularmente fácil de trabalhar, o POX tem uma das curvas de aprendizado mais baixas, possui documentação relativamente boa, exigindo alguma busca, suportando apenas o protocolo OpenFlow versão 1.0.

Ryu

Semelhante ao POX, o Ryu [61] também é controlador SDN baseado em Python. Suas vantagens incluem, entre outras, suporte do protocolo OpenFlow versão 1.3. Uma das suas desvantagens é a curva de aprendizado, sensivelmente mais longa que a do POX.

Além disso, o controlador Ryu segue uma estrutura baseada em componentes. O Ryu fornece componentes de software com APIs bem definidas que facilitam os desenvolvedores criarem novos aplicativos de gerenciamento e controle de rede. Os aplicativos consistem em componentes.

Floodlight

Floodlight [58] é um controlador de código aberto baseado na linguagem de programação Java e com suporte para OpenFlow versão 1.0, porém, também pode operar com equipamentos sem suporte ao OpenFlow. É administrado pela *Open Networking Foundation* (ONF) e pode ser executado em diferentes sistemas operacionais como Linux, Windows e Mac OS. O Floodlight trabalha com *switches* físicos e virtuais que utilizam o protocolo OpenFlow, sendo licenciado pela Apache. Além disso, o Floodlight é testado e aprimo-

rado ativamente por uma comunidade de desenvolvedores profissionais, incluindo vários engenheiros da Big Switch Networks [62].

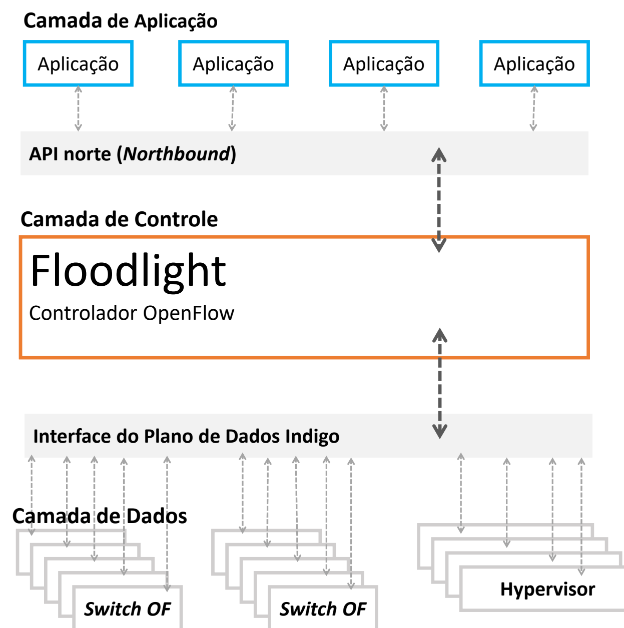

Figura 3.3: Controlador Floodlight e suas relações com outras camadas.

A Figura 3.3 apresenta o relacionamento entre o controlador Floodlight e os aplicativos. Os aplicativos criados são estruturados em módulos Java no quais são compilados com o Floodlight. Por *design*, o Floodlight é um controlador SDN que permite que diversos aplicativos sejam executados em cima dele (parte superior da figura); para tais aplicativos é recomendado a utilização de uma API REST. Para a comunicação do controlador Floodlight com as aplicações é utilizada a API norte (*Northbound*), e a comunicação do controlador Floodlight com os dispositivos físicos é utilizada a interface Indigo [63], que um projeto de código aberto para oferecer suporte ao OpenFlow em uma variedade de dispositivos físicos. Na parte inferior da figura encontram-se os dispositivos responsáveis pelo encaminhamento dos pacotes da rede e os Hypervisores para acesso ao Floodlight através de uma máquina virtual.

Por fim, o Floodlight é um controlador OpenFlow de classe empresarial, possuindo uma vasta documentação e é de fácil utilização, porém apresenta dificuldades para o aprendizado.

Meridian

O Meridian [59] é uma plataforma de controle SDN que suporta um modelo de nível de serviço para redes em nuvem.

Os sistemas atuais para o gerenciamento de uma estrutura de rede baseada em nuvem é projetado principalmente para o fornecimento de uma versão virtualizada dos componentes e funções semelhantes a uma rede física. No entanto o Meridian adota um modelo de nível de serviço de rede em nuvem, no qual lida com as políticas e funções de conectividade, integrando os processos de provisionamento e gerenciamento de aplicações em nuvem, seguindo um modelo de DevOps [64], no qual, em termos gerais, é uma infraestrutura de nuvem que visa a integração entre desenvolvedores de softwares e a equipe de infraestrutura.

Aproveitando o surgimento do paradigma SDN que fornece uma oportunidade para integrar de maneira transparente o provisionamento de aplicativos na nuvem com a rede por meio de interfaces programáveis, o Meridian aproveita tais recursos e se beneficia de alguns serviços (*e.g.*, escalabilidade, flexibilidade e desempenho). O foco do controlador Meridian está no uso de serviços de rede virtual flexíveis para suportar uma variedade de topologias de aplicativos.

O Meridian foi construído em cima da plataforma de código aberto do controlador Floodlight, no qual foram adicionados novos módulos para implementar as abstrações de integração de nuvem.

3.1.2 Controladores Distribuídos

Inicialmente a implementação do OpenFlow descrevia apenas um único controlador por uma questão de simplicidade. No entanto, como o aumento do tamanho das redes de produção que implementam o OpenFlow, confiar em um único controlador para toda a rede pode não ser viável por vários motivos [3]. Primeiro, a quantidade de tráfego de controle destinado ao controlador centralizado cresce com o número de *switches*. Em segundo lugar, se a rede tiver um diâmetro grande, não importa onde o controlador seja colocado, alguns *switches* encontrarão latências de configuração muito longas. Finalmente, como o sistema é limitado pela capacidade de processamento do controlador, os tempos de configuração podem crescer significativamente conforme a demanda cresce com o tamanho da rede.

Para contornar esses problemas que podem advir de uma arquitetura centralizada como ponto único de falha, escalabilidade e desempenho, a versão 1.2 do protocolo OpenFlow [65] trouxe a capacidade do *switch* com suporte a OpenFlow poder ser configurado para manter conexões simultâneas com vários controladores, atendendo ao requisito de alta disponibilidade. Essa especificação capacitou o controlador para assumir uma das três funções diferentes em relação a um *switch*:

- **Igual:** Capacidade total de programar o *switch*.

- **Escravo:** O controlador só pode solicitar dados do *switch*, como estatísticas, mas não pode fazer modificações.
- **Mestre:** Capacidade total de programar o *switch*, porém, o *switch* exige que apenas um controlador esteja no modo mestre e todos os outros estejam no modo escravo.

Apesar da especificação do OpenFlow trazer essa capacidade, a partir da versão 1.2, não há guias ou manuais de como utilizar os benefícios de múltiplos controladores em um ambiente de rede, cabendo aos analistas e projetistas de redes utilizarem técnicas e conhecimentos diversos para implementar tal serviço para garantir essa alta disponibilidade. Exemplo de tal abordagem poderia ser a replicação de controladores com alguma forma de replicação de máquina de estados, utilizando os conceitos de *Master*, *Slave* e *Equal* sugerido pela especificação.

Um controlador distribuído pode ser ampliado para atender aos requisitos de qualquer ambiente, de redes pequenas a grandes. Podendo ser um *cluster* de vários controladores ou controladores distribuídos sendo ligados por meio de uma rede. A Figura 3.4 demonstra tal cenário, no qual uma rede SDN possui vários *sites* sendo mantidos por múltiplos controladores, distribuídos geograficamente. Assim sendo, todas as solicitações são atendidas pelos controladores locais e o tráfego de controle entre sites é mínimo e os controladores são atualizados principalmente por seus vizinhos.

Um controlador distribuído pode melhorar a resiliência e a escalabilidade do plano de controle e reduzir o impacto de problemas causados pela partição de rede [49]. Os controladores distribuídos podem ser divididos em duas categorias principais de arquitetura com base na sua organização física, sendo Controladores Horizontalmente Distribuídos e Controladores Verticalmente Distribuídos

A Figura 3.5 exibe uma estrutura de controlador distribuída horizontalmente, na qual implica uma divisão horizontal da rede em várias áreas, cada uma das quais é manipulada por um único controlador encarregado de gerenciar um subconjunto de *switches.* Há várias vantagens em organizar os controladores em um estilo tão simples, incluindo a redução da latência de controle e a melhor resiliência [66].

Já a arquitetura de controle SDN distribuída verticalmente, mostrada na Figura 3.6, também conhecida como distribuição hierárquica, assume-se que o plano de controle de rede é particionado em vários níveis (camadas), com os quais podem melhorar a escalabilidade e o desempenho [66].

Onix

O Onix [26] é uma plataforma sobre a qual um plano de controle de rede pode ser implementado como um sistema distribuído. Planos de controle escritos dentro do Onix

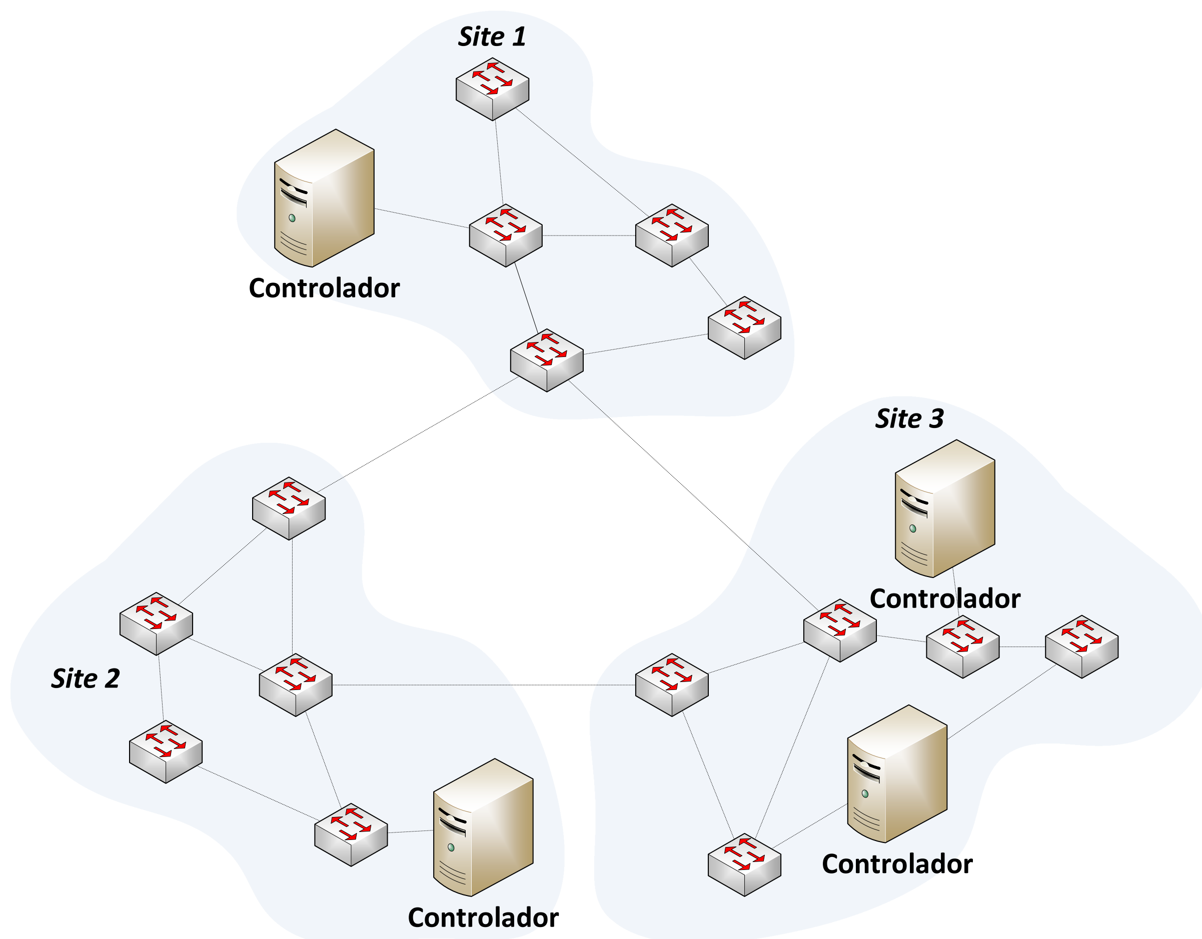

Figura 3.4: Arquitetura controladora distribuída [3].

operam em uma visão global da rede e usam primitivas básicas de distribuição de estado fornecidas pela plataforma. No Onix, a plataforma de controle manipula a distribuição de estado, coletando informações dos *switches* e distribuindo o estado de controle apropriado para eles, além de coordenar o estado entre os vários servidores da plataforma e fornecer uma interface programática na qual os desenvolvedores podem criar uma ampla variedade de aplicativos de gerenciamento.

HyperFlow

O HyperFlow [3] é um plano de controle distribuído baseado em eventos. Ele possibilita que as operadoras de rede implantem qualquer número de controladores em suas redes. Ele fornece escalabilidade enquanto mantém o controle de rede logicamente centralizado: todos os controladores compartilham a mesma visão consistente em toda a rede e atendem

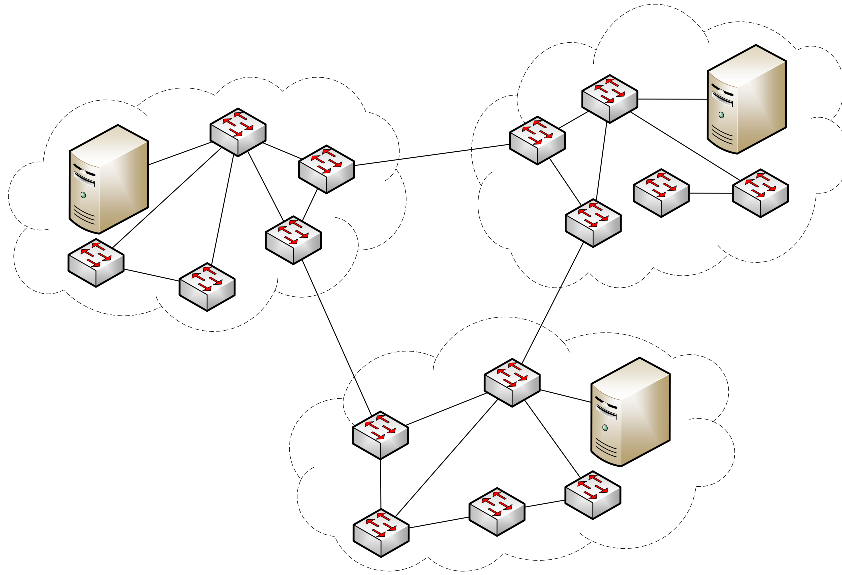

Figura 3.5: Controladores horizontalmente Distribuídos.

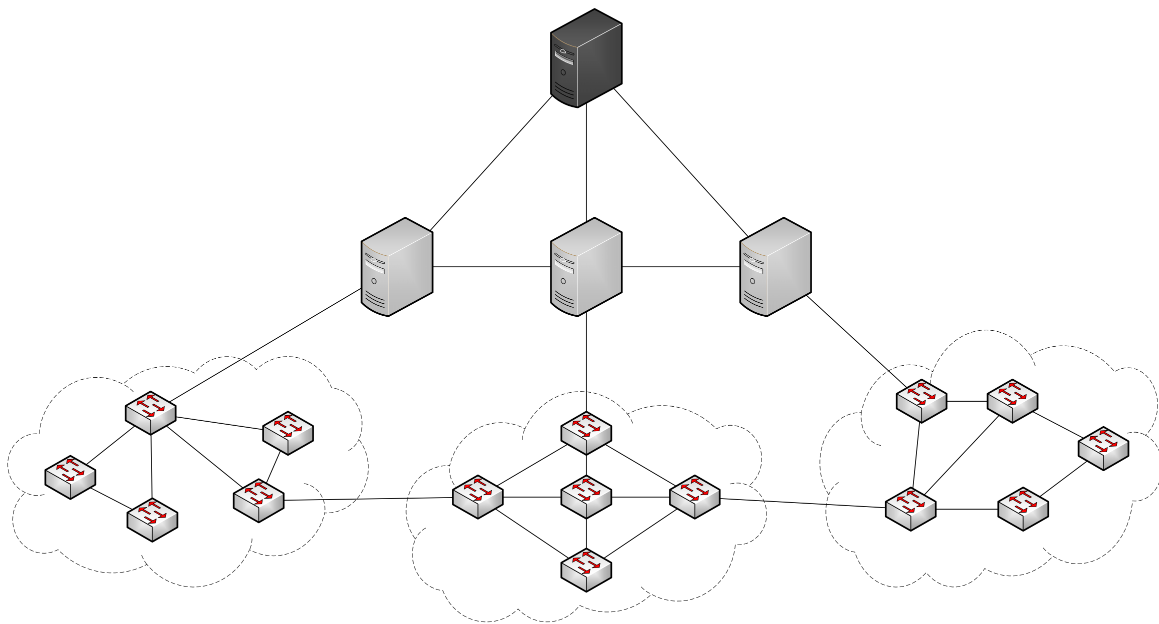

Figura 3.6: Controladores verticamente Distribuídos.

as solicitações localmente sem entrar em contato com nenhum nó remoto, minimizando, assim, os tempos de configuração.

OpenDaylight

O OpenDaylight (ODL) [67] é uma plataforma de controle aberta e modular para personalizar e automatizar redes de qualquer tamanho. Foi projetado desde o início como base para soluções comerciais que abordam uma variedade de casos de uso em ambientes

de rede existentes. A arquitetura do ODL é dividida em três camadas. A camada do topo que consiste de aplicações de rede e serviços. Posteriormente, se encontra a camada de controle onde as abstrações da SDN podem ser encontradas. Finalizando, temos a camada de interfaces *southbound* e *plugins* de protocolos de rede.

ONOS

O ONOS [5] é outra opção disponível para controladores, sendo baseado na linguagem de programação Java, possui uma boa escalabilidade e confiabilidade e tem consistência forte. Ele atribui para cada controlador um subconjunto de dispositivos que contém uma parte da rede inteira e gerencia esse subconjunto. A Figura 3.7 mostra a arquitetura do ONOS, apresentando seus componentes que são:

- Visão da rede: responsável por mantém uma visão global da rede, gerenciar e compartilhar essa visão entre outros controladores ONOS. Informações sobre *switches*, portas, *links* e *hosts* são mantidos aqui.
- Armazenamento Distribuído de chave-valor: O modelo de dados de exibição de rede é implementado usando o banco de dados baseado em grafos (Titan [68]), com armazenamento de *key-value* (chave-valor) do banco de dados Cassandra [69] para distribuição e persistência.
- Registro Distribuído: Para gerenciar a sincronização dos *switches* com os controladores o ONOS utiliza o serviço de coordenação ZooKeeper [70], incluindo a detecção e a reação à falhas.

Kandoo

O Kandoo [71] é um plano de controle altamente configurável e escalável com foco em melhorar a escalabilidade da rede sem fazer alterações nos *switches*, o que violaria os princípios básicos da SDN, no qual o plano de dados ficaria responsável apenas pelo encaminhamento dos pacotes e o plano de controle ficaria responsável pela lógica e gerenciamento da rede. Fato é que sobrecarregar o plano de controle com eventos frequentes torna a rede limitada em termos de escalabilidade, porém o Kandoo preza por outra filosofia adicionando duas camadas de controle (Figura 3.8): uma é a camada superior logicamente centralizada que mantém uma visão geral do estado da rede e outra é a camada inferior que é um grupo de controladores sem interconexão e sem conhecimento o estado global da rede.

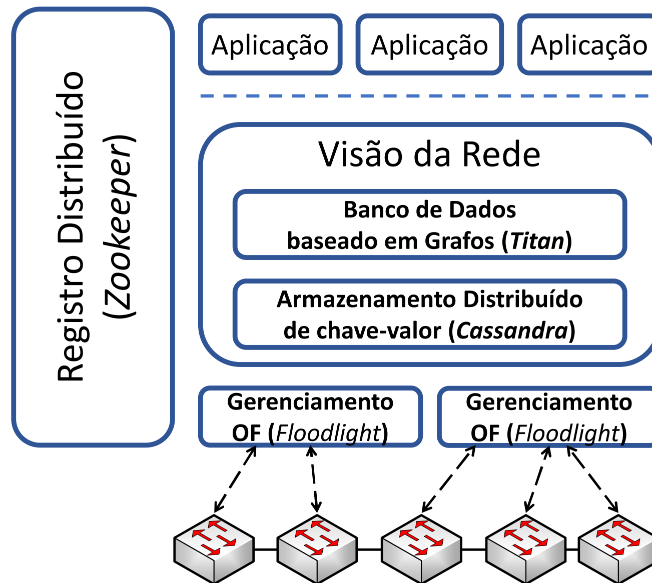

Figura 3.7: Arquitetura do ONOS, adaptado de [5].

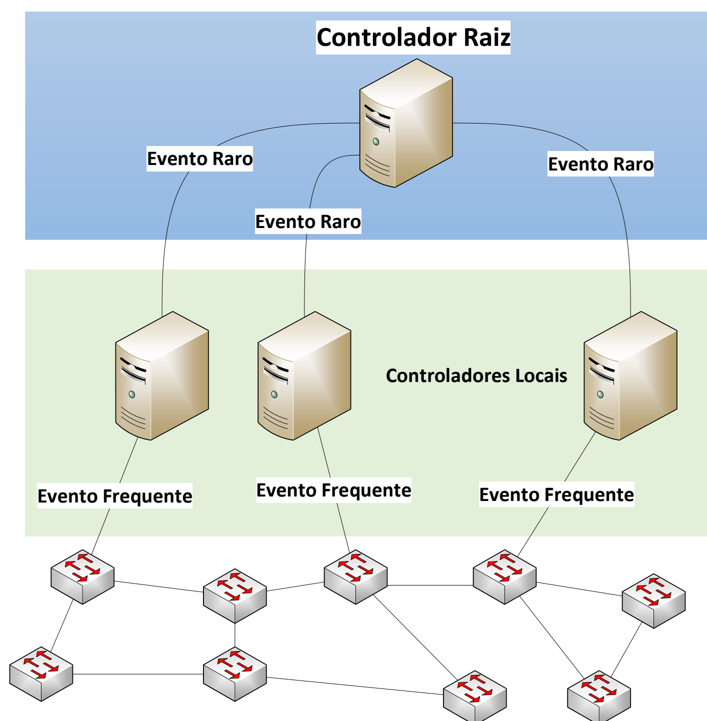

Figura 3.8: Arquitetura do Kandoo.

B4

O B4 [6] é uma rede WAN privada que conecta os *data centers* do Google em todo o planeta (Figura 3.9). O controlador B4 possui várias características únicas como requisitos massivos de largura de banda implantados em um número modesto de *sites*; demanda de tráfego elástico que busca maximizar a largura de banda; e controle total sobre os

servidores e a rede, o que, com essas características, permitiram a adoção da arquitetura de rede definida por software usando o OpenFlow para controlar seus roteadores e *switches*.

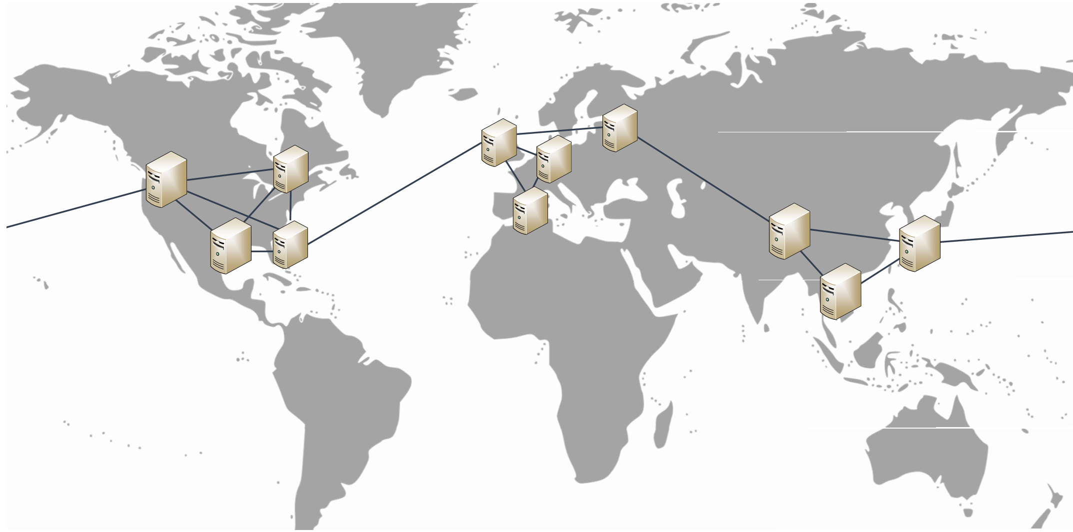

Figura 3.9: Implantação do B4 ao redor do mundo [6].

Espresso

O Espresso [7] é outra plataforma de controle da Google, que gerencia a infraestrutura de roteamento SDN de borda da Internet da Google. Essa arquitetura surgiu da necessidade de escalar exponencialmente e de maneira econômica a borda da Internet. No geral, o Espresso fornece ao Google uma vantagem de escalabilidade programável, confiável e integrada aos sistemas de tráfego global. Em 2017, o Espresso já atendia a mais de 22% do tráfego total do Google na Internet [7].

A Figura 3.10 ilustra a arquitetura do Espresso. Na qual, pode-se ver que o B4 conecta os *data centers* do Google, B2 é a rede pública, que se conecta aos ISPs (*Internet Service Provider*) para atender aos usuários finais. O Espresso é uma infraestrutura SDN implantada na borda do B2, destinada a permitir maior utilização da rede (+ 13%) [7] e implementação mais rápida de serviços de rede.

D-SDN

Decentralize-SDN ou D-SDN [72] é uma iniciativa brasileira de uma estrutura controladora que permite não apenas a distribuição física, mas também a distribuição lógica do plano de controle. Ele realiza a distribuição de controle de rede definindo uma hierarquia de controladores que podem "corresponder" à estrutura organizacional e administrativa de uma rede. O D-SDN permite o controle logicamente descentralizado por meio da delegação de controle entre diferentes níveis da hierarquia, subdividindo em controladores principais

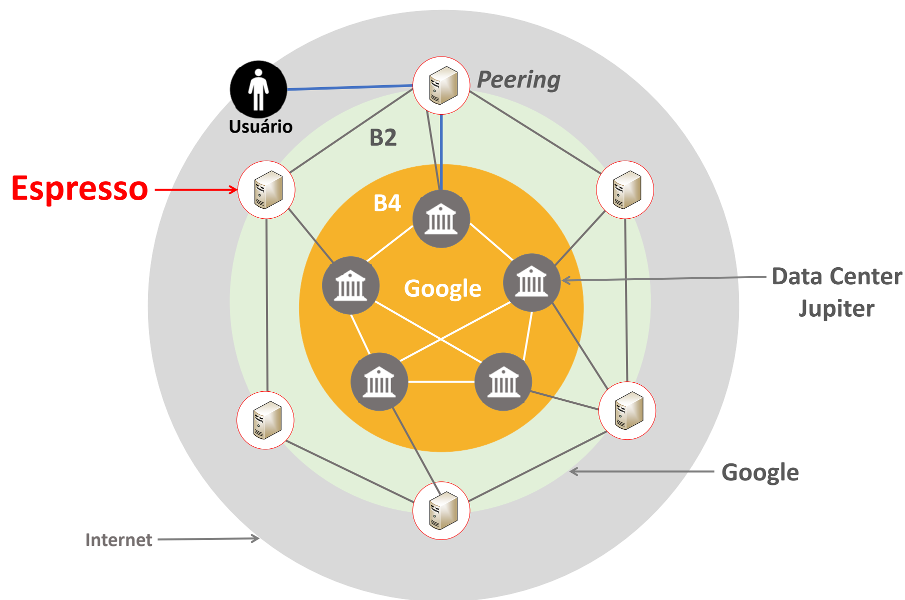

Figura 3.10: Arquitetura do Espresso, adaptado de [7].

MC (*Master Controller*) e controladores secundários SC (*Secundary Controller*), conforme pode ser observado na Figura 3.11.

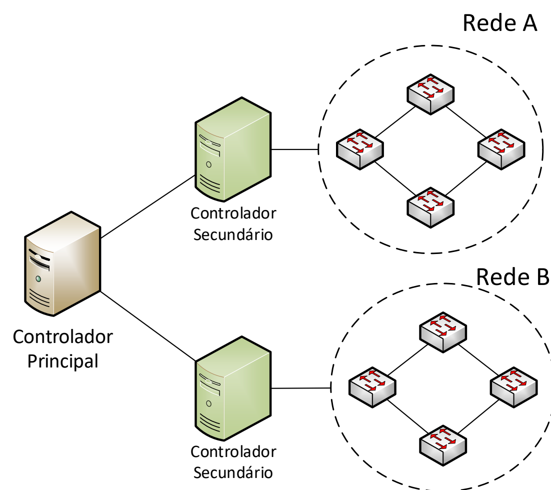

Figura 3.11: Controlador principal e controladores secundários.

A Figura 3.12 ilustra como é feita a delegação de controle, a qual é realizada de acordo com as seguintes etapas:

- Solicitação de check-in: um SC solicita autorização para gerenciar um dispositivo ativado por SDN específico.
- Resposta de check-in: o MC, ao acessar seu banco de dados, autoriza ou nega o acesso pelo SC solicitante.

O D-SDN também incorpora a segurança como parte integrante da estrutura.

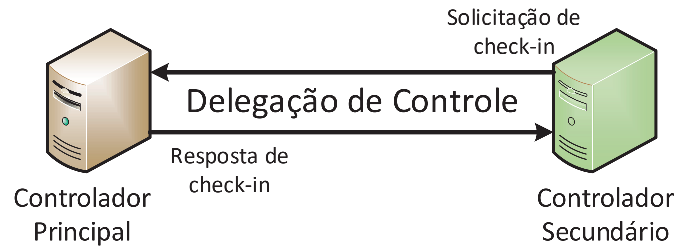

Figura 3.12: Solicitação de controle.

A Tabela 3.2 apresenta os controladores discutidos acima, dividindo as arquiteturas em horizontais e verticais.

Tabela 3.2: Classificação física das arquiteturas do plano de controle SDN distribuídas.

Horizontais	**Verticais**
Onix [26]	Kandoo [71]
HyperFlow [3]	B4 [6]
OpenDayLight [67]	Espresso [7]
ONOS [5]	D-SDN [72]

SMaRt-Light

O SMaRt-Light [8] é um controlador distribuído, tolerante a falhas, baseado na linguagem de programação Java, possuindo consistência forte e boa confiabilidade, porém, com escalabilidade limitada, pois um único controlador é responsável por todas as decisões da rede. A ideia fundamental do SMaRt-Light é ter um controlador *Master* e outros *Slaves* que servem de *backup* do principal.

A arquitetura do SMaRtLight é apresentada na Figura 3.13 no qual utiliza um armazenamento de dados compartilhados que é usado para armazenar o estado da rede e das aplicações e inclui dois aspectos adicionais. Primeiro, integra algoritmos de detecção de falhas e eleição de líderes usando o armazenamento de dados. Segundo, os controladores mantêm um cache local para evitar o acesso ao armazenamento de dados compartilhados no caso de operações de leitura.

O SMaRtLight é uma plataforma de controle destinada a redes SDN pequenas e médias porque utiliza uma abordagem simplificada Mestre-Escravo onde uma réplica de controlador deve recuperar todo o estado de rede em cenários de falhas e possui a desvantagem em termos de latência e *failover*.

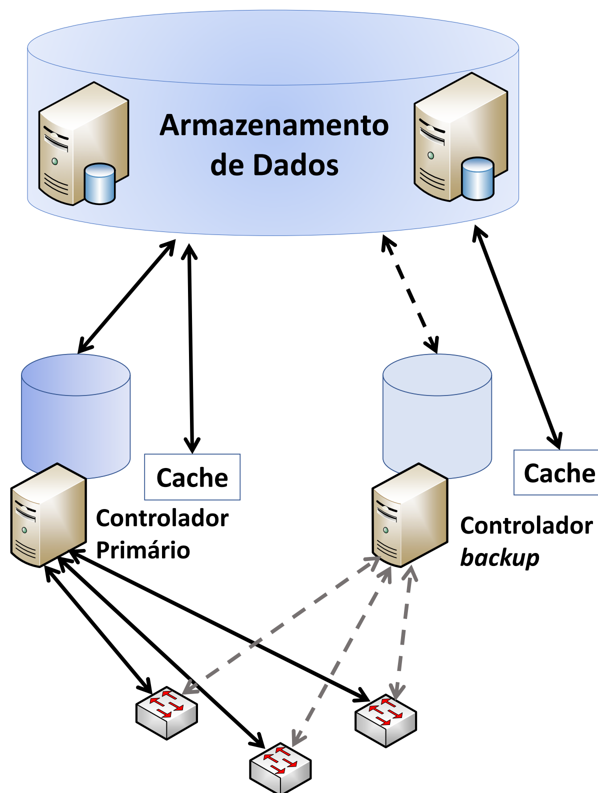

Figura 3.13: Arquitetura do SMaRt-Light, adaptado de [8].

3.2 Propriedades de Segurança dos Controladores

Confiabilidade

Em relação a confiabilidade, preocupações sérias tem sido consideradas no contexto SDN. O desacoplamento do plano de dados para controle tem, de fato, um impacto significativo na confiabilidade do plano de controle SDN. Em uma rede centralizada baseada em SDN, a falha do controlador central pode colapsar a rede geral. Em contraste, o uso de vários controladores em uma arquitetura de controlador fisicamente distribuída alivia a questão de um único ponto de falha.

A redundância do controlador pode ser obtida adotando diferentes abordagens para o processamento de atualizações da rede. Na abordagem de replicação ativa (*Active Replication*) [73], também conhecida como Replicação de Máquina de Estado (*State Machine Replication*), vários controladores processam os comandos emitidos pelos clientes conectados de maneira coordenada e determinística para atualizar simultaneamente o estado da rede replicada. Na abordagem de replicação passiva, conhecida como replicação de *backup*, um controlador (principal) processa as solicitações, atualiza o estado replicado e informa periodicamente as outras réplicas do controlador (*backups*) sobre as mudanças de estado.

Além disso, a distribuição de controle é um desafio central ao projetar uma plataforma controladora tolerante a falhas. A abordagem de controle centralizado que segue o conceito simples de Mestre/Escravo [74] depende de um único controlador (mestre) onde mantém todo o estado da rede e toma todas as decisões com base em uma visão de rede global. Controladores de *backup* (escravos) são usados para fins de tolerância a falhas.

A alternativa centralizada é geralmente considerada em redes de pequeno e médio porte. Por outro lado, na abordagem de controle distribuído, o estado da rede é particionado em vários controladores que assumem simultaneamente o controle da rede enquanto trocam informações para manter a visão de rede logicamente centralizada.

Finalmente, alguns trabalhos [75, 76] consideraram critérios de confiabilidade na colocação de controladores SDN distribuídos no qual o número e a localização dos controladores são determinados de maneira confiável para se alcançar um bom desempenho.

Consistência

Ao contrário dos projetos de SDN fisicamente centralizados, as plataformas de controle SDN distribuídas enfrentam grandes desafios de consistência [77, 78]. Para manter uma adequada visão da rede unificada, os controladores devem trocar dados que transitam pela rede e lidar com a consistência do estado global da rede no qual é armazenada em suas estruturas de dados compartilhadas. A maioria dos controladores distribuídos oferece semântica de consistência fraca, o que significa que as atualizações de dados em nós distintos serão eventualmente atualizadas em todos os nós do controlador. Isso implica que há um período de tempo no qual nós distintos podem ler valores diferentes (valor antigo ou novo valor) para a mesma propriedade. A consistência forte, por outro lado, garante que todos os nós do controlador lerão o valor da propriedade mais atualizada após uma operação de gravação. Apesar de seu impacto no desempenho do sistema, a consistência forte oferece uma interface mais simples para os desenvolvedores de aplicativos.

Manter a consistência de estado entre os controladores SDN é um desafio significativo de projeto que envolve trocas entre o cumprimento de políticas e o desempenho da rede. A questão é que conseguir consistência forte em um ambiente SDN propenso a falhas de rede é quase impossível sem comprometer a disponibilidade e sem adicionar complexidade ao gerenciamento de estado da rede. De acordo com Panda [79] os projetistas de SDN precisam aproveitar a flexibilidade oferecida pela SDN para selecionar os modelos de consistência apropriados para o desenvolvimento de aplicativos com vários graus de requisitos de consistência de estado e com políticas diferentes.

Em relação ao ambiente tão diversificado da SDN, adotar um modelo único de consistência para lidar com diferentes tipos de estados compartilhados pode não ser a melhor alternativa, sendo necessário alcançar diferentes níveis de consistência (i.e., consistência

forte e consistência fraca). Assim, Bannour [14] propõe que uma abordagem híbrida que mescle vários níveis de consistência deve ser considerada para encontrar o equilíbrio ideal entre consistência e desempenho.

Até o momento, existem alguns controladores que oferecem consistência forte para armazenamento de seus dados e estado da rede. Na arquitetura de controle centralizado, naturalmente tal consistência está presente em todos os controladores, por exemplo o NOX [53], POX [60] e o Floodlight [58]. Porém, na arquitetura distribuída a consistência forte está presente em apenas algumas plataformas como, por exemplo, o Onix [26], ONOS [5] e SMaRt-Light [8].

Comparação entre os Controladores

Para resumir os conceitos tratados acima, a Tabela 3.3 elenca algumas características de alguns controladores, subdividido-os em controladores centralizados e distribuídos. O caractere hífen significa que a informação não foi encontrada na documentação do controlador e onde estiver N/A significa que aquela característica não se aplica ao contexto de tal controlador.

Além das considerações de confiabilidade e consistência, os requisitos de escalabilidade (que diretamente impactam o desempenho) também são importantes e devem ser levados em consideração ao projetar uma arquitetura de controlador SDN com propriedades de segurança. Por isso, a tabela também apresenta uma coluna para a propriedade de escalabilidade.

Tabela 3.3: Principais características de alguns controladores SDN.

Nome	Linguagem de Programação	Arquitetura	Plataformas Suportadas	Licença	Documentação	Escalabilidade	Confiabilidade	Consistência
NOX [4]	C++	Centralizada	Linux, MacOS Windows	Apache 2.0	Limitada	Muito Limitada	Limitada	-
POX [60]	Python	Centralizada	Linux, MacOS Windows	Apache 2.0	Limitada	Muito Limitada	Limitada	-
Floodlight [58]	Java	Centralizada	Linux, MacOS Windows	Apache 2.0	Boa	Muito Limitada	Limitada	-
Beacon [55]	Java	Centralizada	Linux, MacOS Windows	GPL 2.0	Razoável	-	-	-
Meridian [59]	Java	Centralizada	Baseado em Nuvem	-	Limitada	-	-	-
Ryu [61]	Python	Centralizada	Linux, MacOS	Apache 2.0	Boa	-	-	Forte
Onix [26]	C / C++ Python	Distribuída	-	Proprietária	Limitada	Muito Boa	Boa	Fraca Forte
ONOS [5]	Java	Distribuída	Linux, MacOS Windows	Apache 2.0	Boa	Muito Boa	Boa	Fraca Forte
OpenDayLight [67]	Java	Distribuída	Linux, MacOS Windows	EPL 1.0	Boa	Muito Boa	Boa	Forte
HyperFlow [3]	C++	Distribuída	-	Proprietária	Limitata	Boa	Boa	Fraca
SMaRtLight [8]	Java	Distribuída	Linux	Proprietária	Limitada	Muito Boa	Muito Boa	Forte
B4 [36]	Python C	Distribuída	-	-	Limitada	Boa	Boa	N/A
Kandoo [71]	C / C++ Python	Distribuída	-	-	-	Muito Boa	Limitada	N/A

3.2.1 Outros Problemas de Segurança em SDN

As redes SDN trouxeram muitos benefícios como a possibilidade de se introduzir inovação na rede por meio do suporte a programabilidade e visão ampla da rede. Apesar destes benefícios, a arquitetura das redes SDN abriu as portas para diversas vulnerabilidades entre as suas camadas, conforme [49].

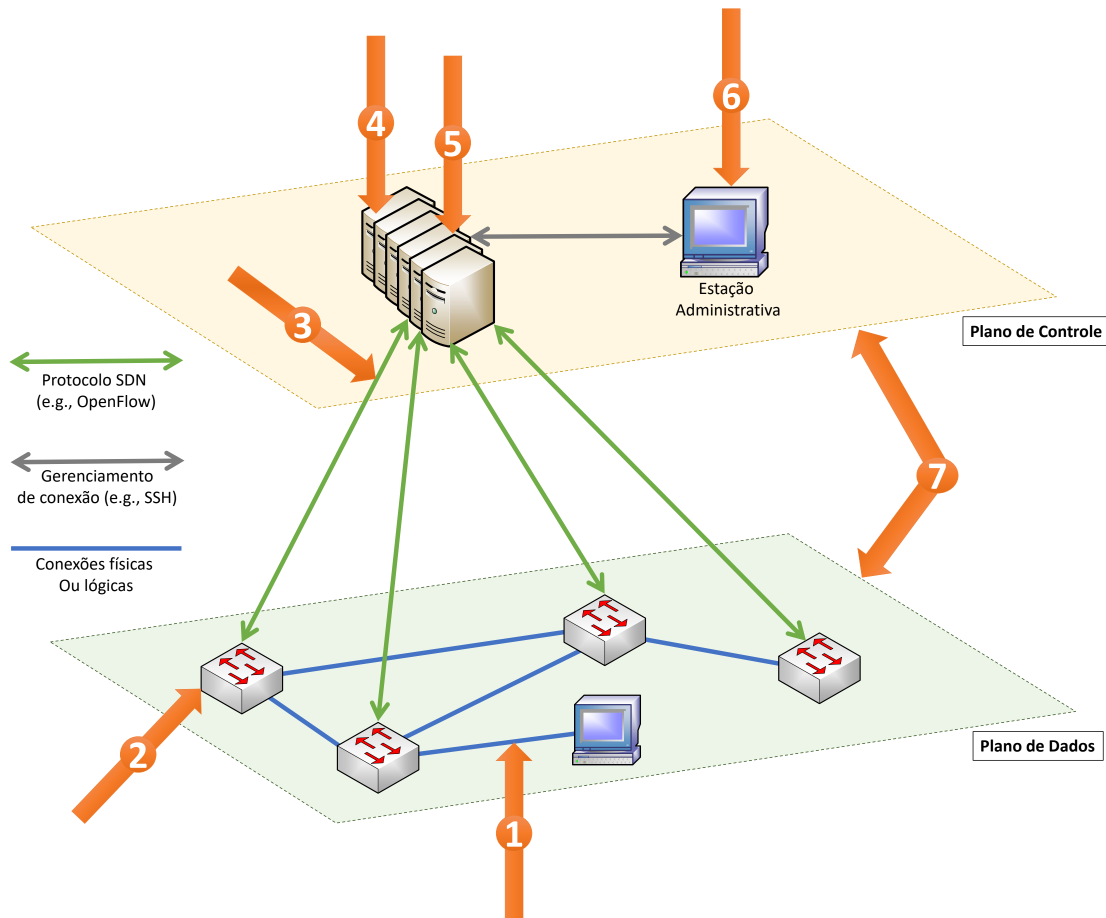

Figura 3.14: Vetores de ameaças SDN, adaptado de [9].

A Figura 3.14 ilustra os vetores de ataques enumerados abaixo.

1. Fluxos de tráfego forjados ou falsificados, no qual um atacante pode usar elementos de rede para iniciar um ataque a um *switch* ou controlador.

2. Ataques a vulnerabilidades do *switch*, no qual se um atacante detiver o controle de um simples *switch* na rede poderá causar muitas falhas, como injetar tráfego falso, inundar a comunicação com o controlador ou até derrubar a rede.

3. Ataques nas comunicações entre os planos de controle e de dados. O protocolo Openflow é responsável pela comunicação entre estes planos, a qual deve ser adequadamente gerenciada para ser segura. Atualmente essa comunicação trabalha

com o protocolo TLS/SSL, no entanto deve-se ter mais controle entre estes planos pois diversos trabalhos [80, 81] já demonstraram as fragilidades de tal protocolo.

4. Ataques em controladores, provavelmente o pior tipo de ataque, onde uma falha no controlador pode comprometer a rede inteira. Por exemplo, um atacante pode tentar capturar ou forjar regras de fluxos comprometendo a confidencialidade ou a integridade.

5. Falta de mecanismos para garantir a confiança entre o controlador e os aplicativos de gerenciamento, no qual deve haver mecanismos para estabelecer comunicações seguras dos aplicativos para o controlador.

6. Ataques e vulnerabilidades em estações administrativas, onde já é um problema de segurança das redes tradicionais e está presente também no ambiente SDN.

7. Falta de recursos confiáveis para análise forense e remediação, para entender as causas dos problemas e as formas de recuperação.

Desafios de segurança do plano de controle SDN

O plano de controle é uma entidade centralizada de tomada de decisão. Logo, o controlador pode ser um ponto altamente visado pelos atacantes para comprometer a rede ou realizar atividades maliciosas devido ao seu papel central [82]. Os principais desafios e ameaças à segurança existentes no plano de controle são elencados abaixo:

1. **Ameaças das aplicações**: os aplicativos implementados no topo do plano de controle podem representar sérias ameaças à segurança do plano de controle. Geralmente, a segurança do controlador é um desafio, do ponto de vista da capacidade do controlador em autenticar aplicativos e autorizar os recursos utilizados pelos aplicativos com isolamento, auditoria e rastreamento adequados [83].

   Portanto, é necessário um mecanismo de imposição de segurança personalizado para vários tipos de aplicativos na API *northbound*.

2. **Ameaças devido à escalabilidade**: No OpenFlow, a maior parte da complexidade é direcionada para um controlador no qual as decisões de encaminhamento são tomadas de maneira lógica centralizada [84]. Se for necessário que os controladores instalem regras de fluxo para cada novo fluxo no caminho de dados, o controlador pode facilmente se tornar um gargalo. De acordo Jarschel *et al.* [85], as implementações de controladores atuais não são capazes de lidar com o grande número de novos fluxos ao usar o OpenFlow em redes de alta velocidade com *links* de 10 Gbps.

Outro trabalho [86] descreve que a falta de escalabilidade permite que ataques direcionados causem saturação do plano de controle com resultados mais prejudiciais em SDNs do que as redes tradicionais.

Outro desafio para as implementações de controladores atualmente disponíveis é especificar o número de dispositivos de encaminhamento a serem gerenciados por um único controlador para lidar com as restrições de atraso. Se o número de fluxos no controlador aumentar, há uma alta probabilidade de aumento do tempo de resposta do controlador, o que depende profundamente da capacidade de processamento do controlador. Essa limitação dos recursos dos controladores pode levar a um ponto único de falha. Para evitar que um controlador sejam um ponto único de falha, é sugerido o uso de vários controladores.

3. **Ataques DoS**: Os ataques DoS e DoS distribuído(DDoS) são os desafios de segurança mais ameaçadores para o controlador SDN. O ataque DoS é uma tentativa de tornar um recurso (de rede) indisponível para usuários legítimos. Um ataque DoS no SDN é demonstrado em [87], que explora a lógica de separação de planos de dados de controle do SDN. Outro ataque DoS no controlador SDN também é demonstrado em [88], em que um invasor envia continuamente pacotes IP com cabeçalhos aleatórios para colocar o controlador no estado não responsivo.

4. **Plano de controle distribuído**: Para gerenciar um grande número e variedade de dispositivos que não podem ser gerenciados por um único controlador SDN, é necessário implantar vários controladores que dividam a rede em diferentes subdomínios. Porém, se a rede for dividida em várias sub-redes definidas por software, a agregação de informações e a manutenção de diferentes regras de privacidade em cada sub-rede serão um desafio.

5. **Outros desafios**: Como o controlador é responsável por diversas tarefas, como, por exemplo, aplicação de políticas em toda a rede, lidar com a lacuna semântica entre o *switch* e o controlador, a distribuição de controle de acesso que suportam fluxos agregados, controladores com vários *switches* e vários controladores em um único domínio podem criar diversos conflitos de configuração [89].

   Em várias infra-estruturas OpenFlow, a inconsistência nas configurações do controlador resultará em potenciais conflitos [89].

Os desafios mais comuns de segurança, descritos acima, são apresentados na Tabela 3.4.

Tabela 3.4: Desafios de segurança no plano de controle SDN.

Tipo de ameaça	Descrição
DoS e DDoS	Inteligência centralizada e recursos limitados no plano de controle são os principais motivos para atrair ataques de negação de serviço.
Acesso não autorizado ao controlador	Não há mecanismos obrigatórios para impor o controle de acesso as aplicações.
Escalabilidade e disponibilidade	A centralização da inteligência em uma entidade provavelmente terá desafios de escalabilidade e disponibilidade.

Trabalhos Relacionados

Nesta seção são discutidas medidas de segurança propostas para proteger o plano de controle de uma plataforma SDN. As soluções de segurança no plano de controle propõem abordagens para proteger o controlador de aplicações maliciosas como, por exemplo, assegurar a segurança e disponibilidade do controlador durante um ataque de negação de serviço. Elas se concentram em realizar medidas de seguranças por meio de filtros, classificação de fluxos, priorização de tarefas, limites de taxa de fluxos recebidos pelo controlador, regras de fluxos proativas e diversas outras medidas.

Ainda, as soluções de segurança do plano de controle são categorizadas em propostas e abordagens para proteger o plano de controle de (1) aplicativos mal-intencionados ou defeituosos, (2) contornar a segurança do SDN visando a escalabilidade do plano de controle, (3) proteger de ataques DoS ou DDoS e (4) garantir o controle, a segurança e a disponibilidade do plano através da colocação confiável do controlador. Abaixo será descrita cada uma dessas categorias de propostas de controle:

1. **Aplicações**: Como as aplicações acessam recursos e informações da rede através do plano de controle, é muito importante proteger o plano de controle contra aplicativos mal-intencionados ou defeituosos. Além disso, o plano de controle deve garantir o acesso a aplicativos legítimos de acordo com seus requisitos funcionais, mas dentro das restrições de segurança.

   O controlador SE-Floodlight [90] fornece mecanismos para separação de privilégios, adicionando uma API segura e programável entre aplicativos e plano de dados. Ele apresenta um módulo de verificação de aplicativo OpenFlow em tempo de execução para validar a integridade dos módulos de classe que produzem regras de fluxo. Para a resolução de conflitos com base em funções, o SE-Floodlight atribui funções

de autorização aos aplicativos de fluxo aberto para resolver conflitos de regras, comparando as funções autorizadas de produtores de regras conflitantes. Da mesma forma, ele pode restringir as mensagens *packet out* produzidas por vários aplicativos e, portanto, proteger a mediação de regras de fluxo.

2. **Escalabilidade do controlador**: No padrão OpenFlow do SDN, um controlador instala regras separadas para cada conexão do cliente, também denominada microfluxo, levando à instalação de um grande número de fluxos nos *switches* e de uma carga pesada no controlador. Portanto, várias abordagens são sugeridas para minimizar a carga em um controlador, distribuir as funcionalidades do plano de controle ou maximizar a potência de processamento e a memória dos controladores.

   Além disso, o OpenFlow suporta o uso de curingas (*wildcards*) para que o controlador direcione um agregado de solicitações de clientes para réplicas de servidores. A utilização de *wildcards* explora o suporte dos *switches* para regras de *wildcards* para obter maior escalabilidade, além de manter uma carga balanceada no controlador.

   Uma análise comparativa de vários paradigmas reativos e proativos do controlador OpenFlow para escalabilidade é apresentada em [91]. Controladores reativos recebem o primeiro pacote de fluxo do *switch* para preencher a tabela de fluxo no *switch* para esse fluxo específico, enquanto controladores proativos definem regras de fluxo antes que os fluxos cheguem ao *switch* com base em algumas regras de encaminhamento predefinidas.

   Fernandez [91] demonstra que controladores proativos são mais escaláveis que seus equivalentes. No entanto, um controlador proativo puro precisaria conhecer antecipadamente todos os fluxos de tráfego, o que não é possível na prática. Portanto, o autor sugere uma arquitetura de controlador híbrido na qual os controladores agem reativamente para configurar rotas e possuem alguma inteligência para agir proativamente para entender o comportamento do tráfego e definir um caminho com antecedência.

   Existem esforços para aumentar o poder de processamento dos controladores e compartilhar responsabilidades entre um conjunto de controladores. Entre esses esforços está o McNettle [92], que é um controlador SDN extensível com vários núcleos (podendo ser escalonado em até 46 núcleos) de CPU desenvolvidos para dimensionar e suportar algoritmos de controle.

   Finalmente, o Maestro [54] propõe o paralelismo através de processadores multinúcleo para aumentar o desempenho do processamento dos controladores, para maior escalabilidade e disponibilidade.

3. **Mitigação de DoS**: Os ataques de DoS ou DDoS podem ser mitigados analisando o comportamento do fluxo e as estatísticas de fluxo armazenadas nos *switches* OpenFlow. Como as estatísticas do comutador podem ser facilmente buscadas no controlador OpenFlow, o processo de coleta de estatísticas no OpenFlow é comparativamente econômico devido à baixa sobrecarga. Uma detecção leve de ataque de inundação DDoS usando Mapas Auto-Organizáveis (do inglês, *Self-Organizing Maps* - SOM) [93] é apresentada em [94]. O SOM é uma rede neural artificial usada para transformar um dado padrão de n-dimensões em um mapa de 1 ou 2 dimensões. O processo de transformação realiza a ordenação topológica, onde padrões de dados com recursos estatísticos semelhantes são reunidos para processamento adicional. Em [94], o mecanismo do SOM é usado para encontrar relações ocultas entre os fluxos que entram na rede.

4. **Posicionamento do controlador**: Em [75], é demonstrado que o número e a localização topológica dos controladores são dois desafios para a escalabilidade e resiliência da rede SDN. Dessa forma, o posicionamento ideal do controlador atraiu muita atenção da comunidade de pesquisa e vários algoritmos para o posicionamento ideal foram examinados e testados. O algoritmo Simulated Annealing (SA), um algoritmo probabilístico genérico, foi considerado o algoritmo ideal para a colocação de controladores em [95, 96, 97].

   O problema de posicionamento do controlador com sua natureza não determinística, *non deterministic polynomial time*, é descrito em [61] para maximizar a confiabilidade das operações de controle SDN, atendendo aos requisitos de tempo de resposta. Os autores sugerem que a sincronização de estado e a coordenação de controle entre controladores são necessárias e podem ser alcançadas com técnicas como a hierarquia do controlador. Para um posicionamento ideal, a porcentagem esperada de perda do caminho de controle é usada como uma métrica de confiabilidade, onde a perda do caminho de controle é considerada como o número de caminhos de controle interrompidos devido a falhas na rede.

   O posicionamento ideal do controlador baseado em Pareto é apresentado em [98] para aumentar a resiliência no SDN. Os autores sugerem que um único controlador pode ser suficiente para satisfazer restrições de latência; no entanto, muito mais controladores (pelo menos 20% de todos os nós precisam ser controladores) são necessários para atender aos requisitos de resiliência da rede.

5. **Troca de inteligência entre plano de dados e controle**: A troca de inteligência entre o controlador e os *switches* pode ser usada para aumentar a disponibilidade do controlador. O DevoFlow [81] é uma dessas abordagens que modifica o modelo

OpenFlow para minimizar a interação dos planos de controle e dados. A arquitetura impõe alguns mecanismos de controle aos *switches*, mantendo o controle central no controlador. Os autores sugerem que a visibilidade central de todos os fluxos pode não ser necessária, mas sim dispendiosa em termos de escalabilidade. Portanto, o DevoFlow é projetado de forma a usar regras OpenFlow com caracteres curingas, e os *switches* tomam suas decisões de roteamento local onde o controlador não conseguir verificar os fluxos. A maioria dos micro-fluxos no DevoFlow é manipulada no plano de dados; no entanto, os operadores de rede podem gerenciar ou examinar qualquer fluxo que seja importante para fins de gerenciamento.

No OpenFlow, um controlador obtém as informações da topologia de rede usando o LLDP, i.e., Protocolo de Descoberta de Camada de *Link*(Rede), *Link Layer Discovery Protocol*, para obter o status da conexão dos *switches* OpenFlow, incluindo o *switch* e as informações de conexão da porta [99]. Para gerenciamento de falhas, o controlador pode usar o LLDP para monitorar os *links* na rede. Nesse caso, é necessário que o controlador esteja envolvido em todas as mensagens de monitoramento LLDP, o que pode resultar em limitação de escalabilidade. Para superar essa limitação, uma arquitetura é proposta em [100] a fim de descarregar o recurso de monitoramento de *link* do controlador para o *switch*. Os autores propõem um gerador de mensagens e uma função de processamento nos *switches*, e uma extensão no protocolo OpenFlow versão 1.1 para suportar a função de monitoramento.

A Tabela 3.5 resume algumas soluções de segurança apresentadas acima, destacando o alvo da ameaça e o impacto gerado.

Tabela 3.5: Ameaças e Soluções de segurança SDN.

Solução de Segurança	**Alvo da ameaça**	**Tipo de Solução**
SE-Floodlight [90]	Autorização das aplicações	Segurança do Controlador fornecendo uma API segura entre os planos de Controle e Aplicação
HybridCtrl [91]	Escalabilidade do Controlador	Arquitetura de Controle Híbrida (reativa e proativa)
Ctrl-Placement [75, 96, 95]	Escalabilidade do Controlador	Posicionamento dos Controladores
HyperFlow [3]	Escalabilidade do Controlador	Plano de Controle Distribuído
DDoSDetection [94]	Ataques DoS e DDos	Framework de detecção

3.3 Considerações finais

Este capítulo abordou o estado da arte dos controladores SDN, subdividindo-os em centralizados e distribuídos e destacando seus principais pontos. Foi citado o problema dos

controladores centralizados serem um ponto único de falha e sofrerem de falta de escalabilidade e, também, foi apresentado os controladores distribuídos, sendo uma possível solução dos problemas citados anteriormente. Porém, vimos que apesar de os controladores distribuídos solucionarem os problemas dos controladores centralizados, eles trouxeram outros desafios a serem sanados, como os problemas de consistência de dados. Além disso, foi exposto um comparativo entre os controladores e os problemas de segurança que afeta essa nova arquitetura de gerenciamento de redes (SDN).

No próximo capítulo será discutido a proposta de segurança para o plano de controle de uma rede SDN, na qual se baseia em um plano de controle seguro e distribuído que utiliza um *data store* seguro para armazenamento do estado da rede e dos dados das aplicações em controladores distribuídos, mantendo requisitos rígidos de segurança, como, por exemplo, confidencialidade, integridade e disponibilidade.

Capítulo 4

Proposta

O presente capítulo apresenta a proposta de trabalho de estudo e pesquisa que se refere a utilização de um armazenamento de dados seguro e distribuído baseado no DepSpace para armazenar o estado da rede e das aplicações instaladas no plano de controle. A vantagem na utilização do espaço de tuplas com propriedades de segurança (DepSpace) é que ele provê mecanismos de tolerância a falhas e confidencialidade, baseados em políticas de acesso e consistência forte para manter o plano de controle livre de falhas e dados corrompidos.

4.1 Plano de Controle Seguro e Distribuído baseado no DEPSPACE

A arquitetura para um plano de controle distribuído com propriedades de segurança é apresentada na Figura 4.1. A ideia principal é fazer com que os controladores utilizem o DepSpace para armazenar e recuperar informações sobre o estado da rede (além de informações relevantes para as aplicações de rede). Consequentemente, os controladores se beneficiarão de um armazenamento de dados consistente que também fornece propriedades de segurança e tolerância a falhas (Seção 2.2.4), o que simplifica o desenvolvimento de aplicações de rede, pois os programadores podem se concentrar na complexidade inerente dos aplicativos que estão desenvolvendo e não precisam lidar, por exemplo, com a possibilidade de falha nos controladores, inconsistência entre os dados manipulados pelos controladores ou até ataques de segurança contra os dados armazenados. Outra característica importante dessa abordagem é que, sempre que um controlador falha, todos os dados relevantes para as aplicações e necessários para configurar os *switches* sob seu controle ainda estão disponíveis com segurança no DepSpace e podem ser usados pelo controlador em seu lugar.

Como o DepSpace é construído de forma distribuída sobre uma camada de Replicação Máquina de Estados [34], um modelo de armazenamento com consistência forte e tolerante a falhas é fornecido para os controladores.

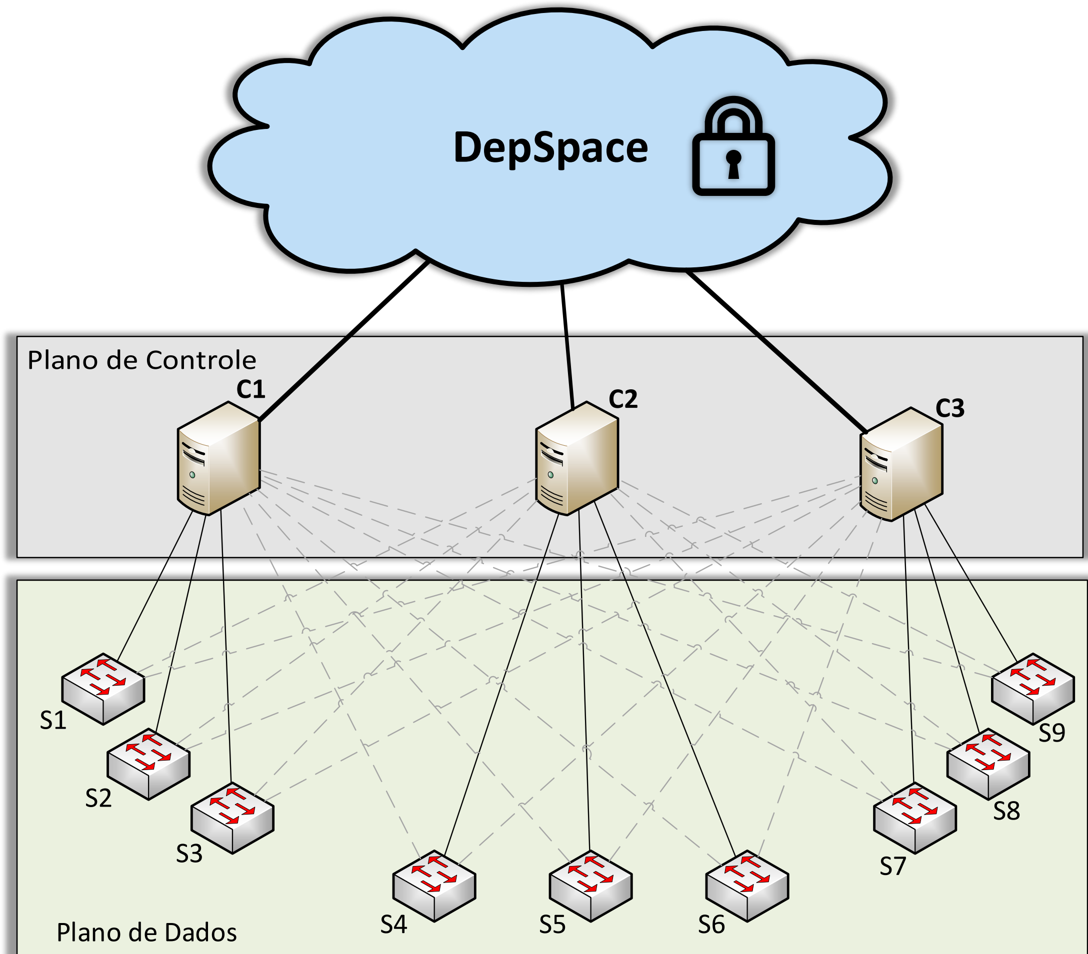

Figura 4.1: Plano de controle baseado no DepSpace.

Nesta arquitetura, cada controlador é responsável/mestre (*master*) por um subconjunto dos *switches*, sendo que outros controladores devem ser configurados como secundários/escravos (*slaves*) destes *switches*, a fim de manter o correto funcionamento destes *switches* em caso de falha do controlador mestre. Considerando a Figura 4.1, o controlador $C1$ é mestre dos *switches* $S1$,$S2$ e $S3$, sendo que por exemplo o controlador $C2$ pode ser configurado como escravo destes *switches*. Caso $C1$ falhe, $C2$ passa então a controlar $S1$,$S2$ e $S3$.

Note que, apesar das informações ficarem armazenadas e serem manipuladas de forma segura, evitando que ataques comprometam os dados armazenados no DepSpace (nosso *data store*), um controlador malicioso ainda pode encaminhar dados incorretos (e.g., regras de fluxo) para os *switches* que controlam. Porém, não podem forjar dados/regras

para serem instaladas em toda a rede SDN. Para solucionar este problema, é necessário que um quórum de controladores atestem a autenticidade destas regras (e.g., através de uma assinatura gerada através de um protocolo de criptografia de limiar), que deve ser verificada nos *switches*.

Para utilizar esta arquitetura, os dados a serem armazenados primeiramente devem ser modelados em forma de tuplas, cujos campos devem receber uma classificação adequada a fim de garantir a segurança destas informações e, ao mesmo tempo, possibilitar a realização de futuras buscas. Estes procedimentos são exemplificados nas seções seguintes que apresentam a implementação de duas aplicações de rede.

4.1.1 Aplicações de Rede

Esta seção apresenta duas aplicações de rede que utilizam a arquitetura proposta: aprendizado de *switch* e balanceador de carga. Essas aplicações servem como prova de conceito para mostrar a viabilidade da arquitetura proposta neste trabalho. As Figuras 4.3 e 4.4 apresentam os fluxogramas para essas aplicações, detalhadas abaixo.

Aprendizado de *Switch*

O aplicativo aprendizado de *switch* emula um processo de encaminhamento de *switch* de camada 2 com base em uma tabela de *switches* que associa endereços MAC a portas de *switch*. O *switch* preenche esta tabela escutando cada pacote de entrada que, por sua vez, é encaminhado de acordo com as informações presentes nesta tabela [12]. O *switch* de camada 2 toma suas decisões de encaminhamento com base apenas nas portas de origem e destino e nos endereços MAC.

Por exemplo, considere a Figura 4.2 que ilustra um novo *switch*, ainda não usado, e que foi adicionado na rede. Este *switch* ainda não sabe quais computadores estão conectados na rede, então ainda não aprendeu os endereços MAC dos quatro computadores conectados. A Tabela 4.1 ilustra a tabela de mapeamento de endereços MAC para portas, a qual fica armazenada no *switch*. Em uma rede SDN, estas informações ficam armazenadas na tabela de fluxos.

Tabela 4.1: Mapeamento Portas - MAC.

Porta	**MAC**
1	00-00-00-00-00-01
2	00-00-00-00-00-02
3	

O *switch* povoa a tabela de endereços dinamicamente. A cada fluxo que entra em um *switch* é verificado se o endereço MAC de origem e o número da porta em que o pacote

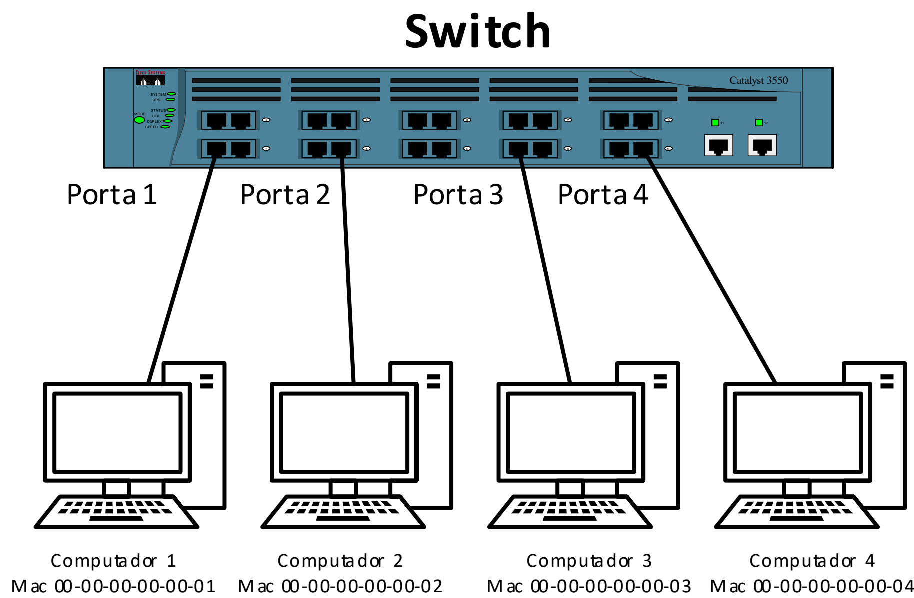

Figura 4.2: Aprendizado de *Switch*.

entrou no *switch* constam na tabela de endereços. Se o endereço MAC de origem não existe, é adicionado na tabela juntamente com o número da porta de entrada. Supondo que o computador 1 queira enviar um pacote para o computador 2, o *switch* adiciona o endereço MAC do computador 1 na tabela, juntamente com a porta de entrada. Em seguida, o *switch* procurará uma correspondência entre o endereço MAC de destino do pacote e uma porta na respectiva tabela. Caso o endereço MAC de destino esteja na tabela, ele encaminhará o pacote pela respectiva porta. Considerando o modelo de rede tradicional, se o endereço MAC de destino não estiver na tabela, o *switch* encaminhará o pacote para todas as portas a fim de descobrir a porta do destino (*broadcast*). Já em uma rede SDN, se o endereço MAC de destino não estiver na tabela de fluxos, o *switch* encaminhará um pedido para o controlador.

A partir daí, o controlador verifica se conhece o endereço MAC e a porta de quem deseja enviar o pacote (computador 1). Caso não conheça, ele armazena o endereço MAC do remetente e a porta de origem desse pacote. Logo depois, o controlador verifica se conhece o endereço MAC de destino. Caso não conheça, ele realiza uma inundação (*flooding*), enviando esse pacote por todas as portas do controlador.

Com isso, o destino encaminhará uma resposta ao remetente. Porém, a tabela de fluxos do destino não possui nenhuma informação para encaminhar diretamente a mensagem, então, novamente, o destino encaminhará uma mensagem para o controlador. O controlador então irá verificar se conhece o endereço MAC desse remetente (computador 2). Caso não conheça, ele irá armazenar no controlador o endereço MAC e a porta. Após isso, o controlador verificará se conhece o endereço MAC de destino (computador

1). Neste caso, ele já possui essas informações, que foi o resultado da primeira requisição de envio. Como ele conhece as informações do computador 1, o controlador encaminhará o pacote à porta correta. Com isso ele aprendeu o endereço MAC e a porta do *switch* de dois computadores.

Sabendo dessas informações, o controlador configurará na tabela de fluxo essas informações e quando o computador 1 ou o computador 2 precisarem se comunicar eles podem trocar informações diretamente, sem necessitar de auxilio do controlador.

Para esta aplicação de aprendizado de *switch*, consideramos dois tipos de *workloads*, um para *broadcast* e outro para *unicast* de pacotes.

- *Workload* para *broadcast* (1 *out*): Como em um *broadcast* o pacote deve ser encaminhado para todas as portas, a aplicação aprendizado de *switch* apenas descobre o endereço de origem e, através de uma operação *out*(espaço de tuplas), armazena no DepSpace uma tupla que associa o endereço de origem com a respectiva porta de entrada.
- *Workload* para *unicast* (1 *out* e 1 *rdp*): Para um *unicast*, a aplicação aprendizado de *switch* primeiramente descobre o endereço de origem, executando uma operação *out* como descrito anteriormente. Por fim, executa um *rdp* para ler a porta de saída associada com o endereço de destino.

Vale destacar que os campos das tuplas ⟨ endereço, porta ⟩ devem ser configurados como comparáveis determinísticos, $\langle CD, CD \rangle$, i.e., os dois campos são classificados como comparáveis determinísticos (Seção 2.2.4) para serem protegidos e permitir a execução de pesquisas (operação *rdp*). Os Algoritmos 2 e 3 apresentam os processamentos necessários no *workload* de *broadcast* e *unicast*, respectivamente. Antes de usar estes algoritmos, é necessário executar o Algoritmo 1 para compartilhamento das chaves a serem utilizadas nos algoritmos seguintes. Note que o próprio espaço de tuplas é utilizado para compartilhamento das chaves.

Balanceador de Carga

O aplicativo de balanceamento de carga emprega um algoritmo *round-robin* para distribuir as solicitações endereçadas a um endereço IP virtual (VIP) em um conjunto de servidores. Atualmente, o uso da rede vem aumentando e para manter a qualidade dos serviços de forma satisfatória para os usuários e aplicações deve haver meios para evitar problemas de sobrecarga de serviços e indisponibilidade de informações. O balanceamento de carga pode ajudar a economizar energia e melhorar a utilização dos recursos de uma rede SDN.

Uma técnica típica de balanceamento de carga é usar um balanceador de carga dedicado para encaminhar as solicitações do cliente para diferentes servidores, essa técnica

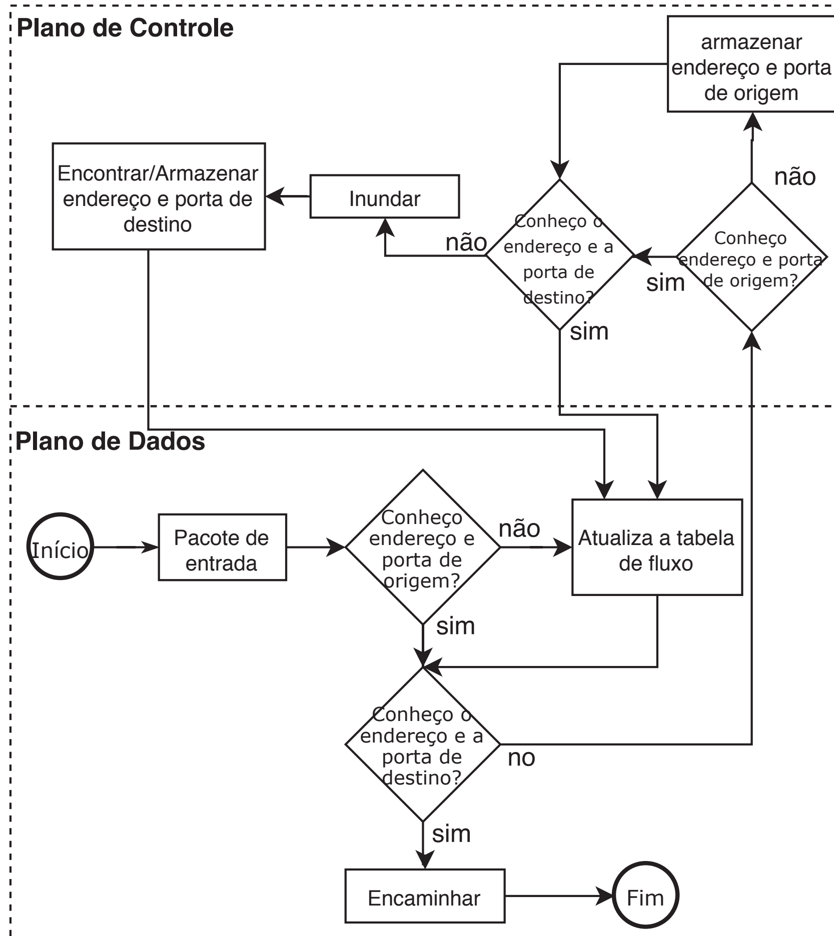

Figura 4.3: Aprendizado de *Switch*.

Algoritmo 1 Compartilhamento de chaves

1: void sharing() {
2: $shared_key_{priv}$ = gera chave privada;
3: $shared_key_{pub}$ = gera chave pública;
4: $t = \langle \text{“}keys\text{”}, shared_key_{pub}, shared_key_{priv} \rangle$ as $\langle PU, PU, PR \rangle$
5: $\bar{t} = \langle \text{“}keys\text{”}, *, * \rangle$ as $\langle PU, PU, PR \rangle$
6: $\langle \text{“}keys\text{”}, key'_{pub}, key'_{priv} \rangle = \mathbf{cas}(\bar{t}, t)$;
7: **if** $\langle \text{“}keys\text{”}, key'_{pub}, key'_{priv} \rangle$!= *null* **then**
8: $shared_key_{pub} = key'_{pub}$
9: $shared_key_{priv} = key'_{priv}$
10: **end if**
11: }

requer suporte de *hardware* dedicado que é caro, sem flexibilidade e fácil de se tornar um ponto único de falha. Considerando uma rede SDN, os controladores podem implementar este serviço e decidirem como gerenciar os pacotes, i.e., para qual destino encaminhá-lo. Consideramos a seguinte *workload* para esta aplicação (3 *rdp* e 1 *replace*):

Algoritmo 2 Learning Switch - Broadcast.

```
1: void broadcast(SourceMac, inPort){
2:     t = ⟨SourceMac, inPort⟩ as ⟨CD, CD⟩;                    // gera a tupla
3:     out(t);                           // adiciona a tupla ao DepSpace
4: }
```

Algoritmo 3 Learning Switch - Unicast.

```
1: int unicast(SourceMac, inPort, destMac) {
2:     t = ⟨SourceMac, inPort⟩ as ⟨CD, CD⟩;                    // gera a tupla
3:     out(t);                            //adiciona a tupla ao DepSpace
4:     t̄= ⟨destMac, ∗⟩ as ⟨CD, CD⟩;                        // define o template
5:     ⟨destMac, number⟩ = rdp(t̄);    // ler a porta de saída associada ao destino
6:     return number;
7: }
```

Algoritmo 4 LoadBalancer.

```
1: Address:Port LoadBalancer(vipAddress) {
2:     t̄_vip= ⟨vip, ∗, ∗, ∗, vipAddress⟩ as ⟨PU, PR, PR, PR, CD, PR, PR⟩;
3:     ⟨vip, vipId, name, protocol, vipAddress⟩ = rdp (t̄_vip);    // ler as info do Vip
4:     t̄_pool= ⟨pool, ∗, name, ∗, vipId, ∗, ∗⟩ as ⟨PU, PR, PR, PR, CD, PR, PR⟩;
5:     while true do
6:         ⟨pool, poolId, name, method, vipId, currentMember, numberOfMembers⟩ =
                                          rdp(t̄_pool); // ler o Pool de servidores
7:         nextMember = (currentMember + 1) mod numberOfMembers;
8:         upPoll = ⟨pool, poolId, name, method, vipId, nextMember, numberOfMembers⟩
                                          as ⟨PU, PR, PR, PR, CD, PR, PR⟩;
9:         oldPoll = ⟨pool, poolId, name, method, vipId, currentMember, numberOfMembers⟩
                                          as ⟨PU, PR, PR, PR, CD, PR, PR⟩;
10:        if replace (oldPool, upPool) ≠ null then
11:            t̄_member = ⟨member, currentMember, ∗, ∗, pollId⟩
                                          as ⟨PU, CD, PR, PR, CD⟩;
12:            ⟨member, currentMember, address, port, pollId⟩ = rdp(t̄_member);
13:            return address : port;
14:        end if
15:    end while
16: }
```

1. Buscar a entidade VIP associada com o destino VIP através de uma operação *rdp*. A tupla $\langle vip, vipId, name, protocol, vipAddress\rangle$ que deve ser classificada como $\langle PU, PR, PR, PR, CD\rangle$ é criada para cada VIP. O último campo é CD para permitir a execução da operação $rdp(\langle vip, *, *, *, vipAddress\rangle)$ e recuperar o $vipId$ (segundo campo).

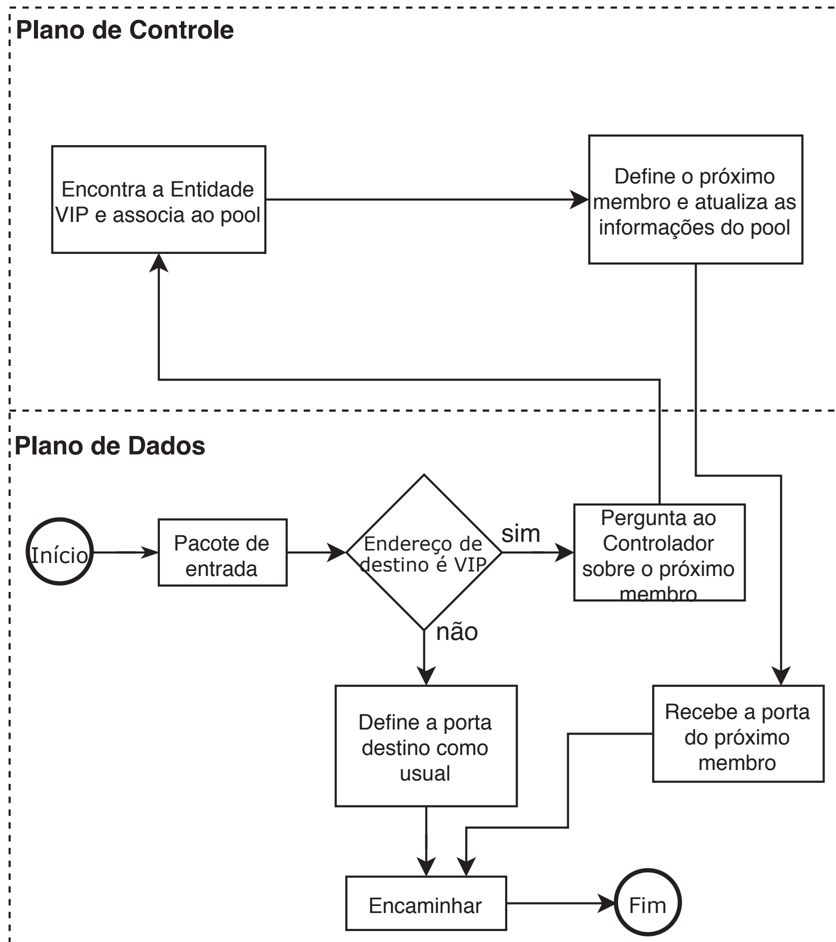

Figura 4.4: Balanceador de Carga.

2. Buscar o pool de servidores associados com o $vipId$, através da operação rdp. A tupla $\langle pool, poolId, name, method, vipId, currentMember, numberOfMembers\rangle$, definida como $\langle PU, PR, PR, PR, CD, PR, PR\rangle$, é criada para cada pool.

   O campo $vipId$ é classificado como CD para permitir a execução da operação $rdp(\langle pool, *, *, *, vipId, *, *\rangle)$ usando o $vipId$ obtido na etapa anterior, obtendo todas as informações do pool.

3. Atualizar as informações do pool, através da operação *replace*. Usando as informações do pool, o algoritmo *round-robin* é executado para atualizar o atributo *current_member* (com o identificador do próximo membro do pool) e as informações do pool são atualizadas através da operação *replace*, no qual substitui as informações antigas pelas novas. Devido a acessos concorrentes, esta operação pode retornar nulo e neste caso os procedimentos devem ser repetidos a partir do item anterior.

4. Ler o membro escolhido, através da operação rdp. Finalmente, é necessário obter

o endereço e a porta do $currentMember$, obtidos na etapa 2. A requisição será encaminhada a este membro. A tupla $\langle member, memberId, address, port, pollId\rangle$, classificada como $\langle PU, CD, PR, PR, CD\rangle$, é criada para cada membro. Os campos $memberId$ e $poolId$ são classificados como CD para permitir a execução da operação $rdp(\langle member, currentMember, *, *, poolId\rangle)$ e obter todas as informações do membro. Note que o $currentMember$ e o $poolId$ foram definidos na etapa 2.

O Algoritmo 5 apresenta um exemplo dos processamentos necessários para popular o espaço de tuplas com as informações necessárias para o balanceador de carga considerando um *vip* e um *pool* com três membros associados, i.e., a carga endereçada para este *vip* será distribuída entre estes três membros. Após isso, o Algoritmo 4 apresenta os processamentos necessários para executar o balanceador de cargas.

Algoritmo 5 Load Balancer - Inserir Informações no Espaço de Tuplas.

```
1:  void populateSpace() {
2:      t_vip = ⟨vip, vipId, name, protocol, vipAddress⟩ as ⟨PU, PR, PR, PR, CD⟩;
3:      t_pool = ⟨pool, poolId, name, method, vipId, currentMember, numberOfMembers⟩
                                              as ⟨PU, PR, PR, PR, CD, PR, PR⟩;
4:      t_member1 = ⟨member, memberId1, address1, port1, pollId⟩
                                              as ⟨PU, CD, PR, PR, CD⟩;
5:      t_member2 = ⟨member, memberId2, address2, port2, pollId⟩
                                              as ⟨PU, CD, PR, PR, CD⟩;
6:      t_member3 = ⟨member, memberId3, address3, port3, pollId⟩
                                              as ⟨PU, CD, PR, PR, CD⟩;
7:      out(t_vip);                           //adiciona o vip ao DepSpace
8:      out(t_pool);                          //adiciona o pool ao DepSpace
9:      out(t_member1);                       //adiciona os membros ao DepSpace
10:     out(t_member2);
11:     out(t_member3);
12: }
```

4.2 Uso de Coordenação Extensível no DepSpace

Usando extensões seguras, implementadas no DepSpace [101], é possível executar qualquer carga de trabalho com um único acesso (operação) ao armazenamento de dados, melhorando o desempenho geral. A ideia básica é que, através de extensões instaladas nos servidores, as solicitações são interceptadas, processadas atomicamente e os dados armazenados no DepSpace são atualizados em um único acesso (operação única). Usando esquemas de criptografia robustos, como criptografia homomórfica (ou parcialmente homomórfica), os servidores podem processar sobre dados criptografados. Na *workload* unicast do aprendizado de *switch*, uma única operação armazena informações relacionadas

ao endereço de origem e retorna as informações relacionadas ao destino. Já o *workload broadcast* não pode ser melhorada, pois já acessa o DepSpace apenas uma vez. No balanceador de carga, os membros de um pool devem ser organizados em uma lista cíclica e os controladores enviam uma solicitação com o endereço VIP (como na etapa 1) e, no lado dos servidores, o pool é recuperado e as informações relacionadas ao próximo membro da lista cíclica é atualizado e retornado.

4.3 Integração no Controlador Floodlight

A seguir serão apresentados alguns experimentos da integração do DepSpace ao controlador Floodlight. Como o Floodlight é composto por módulos Java, foi necessária a criação de um módulo (chamado *integration*) para realizar a conexão com o DepSpace. Após a criação do módulo Java, foi feita a criação de métodos que manipulassem os dados trafegados em uma rede, no qual esses métodos capturaram os pacotes de entrada que chegam a algum elemento do plano de dados de uma rede SDN, processando-os e subdividindo em diversos campos. Após a captura do fluxo de rede foi feita a conexão com o espaço seguro através de uma chamada dentro do módulo *integration*, explicado abaixo.

O módulo *integration*, representado pela figura 4.5, é composto pelos métodos: *receive, init, startUp, processPacketInMessage* e os métodos de cada aplicação (*broadcast, unicast* e *loadBalancer*) que se comunicam para armazenar de forma segura as informações trafegadas entre os *hosts* da rede. O processamento deste módulo inicia-se pelo método *receive*, o qual captura o fluxo enviado entre os *hosts*, onde verifica o *switch* remetente, a mensagem enviada e o contexto; a partir daí verifica o tipo de mensagem (*broadcast, unicast, Vip*) e envia ao método *init*.

O método *init*, responsável por mapear e armazenar o estado aprendido por cada *switch* e inicializar alguns serviços para registrar os eventos de entrada e configurações de *debugs*, invoca o método *startUp*, que é responsável por registrar e ler as configurações dos pacotes, como carga útil de mensagens, nome dos serviços dos canais de comunicação, *id* e endereço IP do controlador atual e outras configurações. Após essas configurações iniciais, o método *processPacketInMessage* é chamado, no qual, a partir dele, é feito a leitura de diversos atributos do cabeçalho do pacote e subdividindo os campos em diversos atributos (*sourceMac, destinationMac, vlan, inPort, outPort, Vips, Members, Pool*) para a realização da lógica da aplicação (i.e., aprendizado de *switch* ou balanceador de carga). Depois do processamento do pacote, são chamados os métodos *learningSwitch* ou *loadBalancer* que modela tais atributos em formas de tuplas e realiza o envio de tais informações ao DepSpace, conforme os fluxogramas 4.3 e 4.4 para cada aplicação.

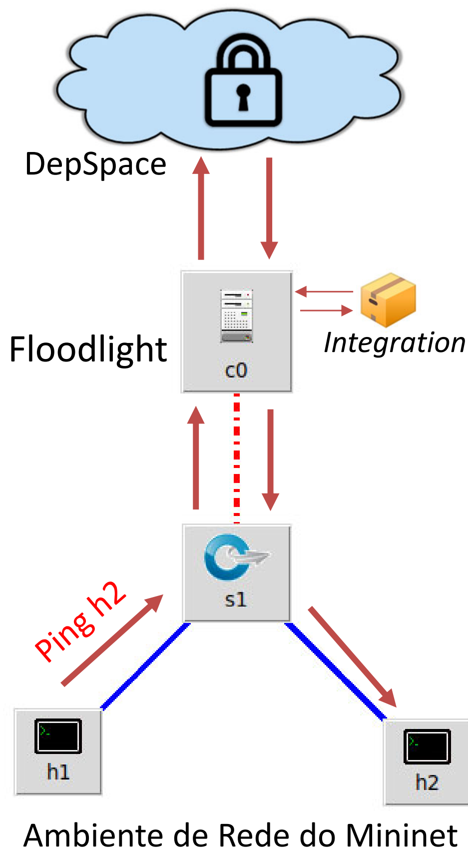

Figura 4.5: Integração de uma rede emulada no mininet com o controlador Floodlight e este com o DepSpace. O tráfego entre os *hosts* é interceptado pelo controlador e neste existe um módulo chamado *Integration* que manipula o *packet-in* e envia ao armazenamento seguro.

Para cada *workload* descrito foram criados métodos específicos para cada aplicação. Todos os códigos referentes ao projeto está hospedado no github, através da URI `https://github.com/jefferson31jan/DepSpaceFloodlight`.

4.4 Experimentos

Visando analisar o desempenho da arquitetura proposta, protótipos das aplicações acima discutidas foram implementados e experimentos foram realizados no Emulab [102]. Os experimentos podem ser divididos em dois grandes conjuntos. Um primeiro conjunto de experimentos com o objetivo de determinar os custos associados ao acesso ao novo componente do plano de controle (DepSpace), i.e., o *overhead* introduzido no sistema; e um segundo conjunto de experimentos com o objetivo de verificar o desempenho da solução proposta integrada ao controlador Floodlight.

4.4.1 Custo necessários para acessar o DepSpace

Esta seção apresenta os custos relacionados com os acessos ao DepSpace para cada uma das aplicações e *workloads*, considerando as soluções com e sem o uso de coordenação extensível.

Configuração dos Experimentos. O ambiente para os experimentos foi constituído por 5 máquinas *d430* (2.4 GHz E5-2630v3, com 8 núcleos e 2 *threads* por núcleo, 64GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O DepSpace foi configurado com 4 servidores para tolerar até uma falha maliciosa. Cada servidor foi executado em uma máquina separada, enquanto que um controlador (que funciona como cliente do DepSpace) foi executado na máquina restante. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 14 64-*bit* e máquina virtual Java de 64 *bits* versão 1.8.0_131.

Resultados e Análises. As figuras 4.6 e 4.7 mostram os resultados de desempenho para a arquitetura proposta, considerando o custo para acessar o armazenamento de dados no DepSpace. A avaliação considera ambos *workloads* das aplicações de aprendizado de *switch* e balanceador de carga. Além disso, também avaliamos as aplicações implementadas com otimizações, discutidas na seção anterior, a saber, extensões seguras. Note que os custos se referem ao *overhead* introduzido pelo plano de controle para manter os dados armazenados de forma segura e compartilhada no DepSpace, seguindo o modelo de consistência forte.

A *workload broadcast* necessita apenas executar uma operação *out*, apresentando uma latência menor que a *workload unicast*, que necessita duas operações para acessar o DepSpace. Além disso, a taxa de transferência aumenta com o número de controladores, mas a latência permanece quase constante. Esta é uma característica importante do DepSpace [2, 101]: a maioria dos custos de criptografia está localizada no lado do cliente; consequentemente, o sistema é escalável com o número de controladores. Em uma implementação de *unicast* usando extensões, o desempenho é semelhante para o *broadcast*, já que tudo é computado dentro do DepSpace com apena um acesso. Uma observação final sobre a aplicação aprendizado de *switch* é que ela é executada apenas quando os *switches* não têm as informações sobre o destino do pacote em sua tabela de fluxo, principalmente quando um novo *switch* ou computador é conectado à rede. Consequentemente, esse custo adicional pode ser negligenciado na maioria dos cenários.

A aplicação de balanceamento de carga apresentou a maior latência, que aumentou com o número de controladores. Além disso, a taxa de transferência permaneceu quase constante. Isso acontece porque aumentar o número de controladores executando simulta-

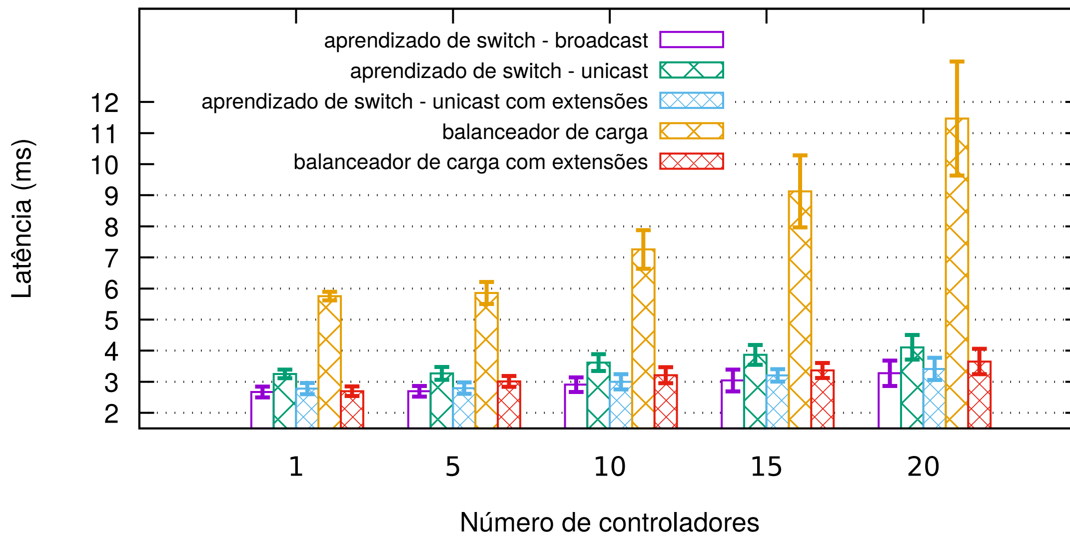

Figura 4.6: Latência medida em *ms*; a aplicação aprendizado de *switch broadcast* foi a aplicação que teve a menor latência, pois executa apenas uma operação no DepSpace, já a aplicação aprendizado de *switch unicast* apresentou maior latência pois executa 2 operações no DepSpace, a aplicação balanceador de carga apresentou a maior latência pois executa diversas operações no armazenamento distribuído, felizmente as aplicações aprendizado de *switch unicast* e balanceador de carga com extensões apresentaram latências próximas a da aplicação aprendizado de *switch broadcast*

.

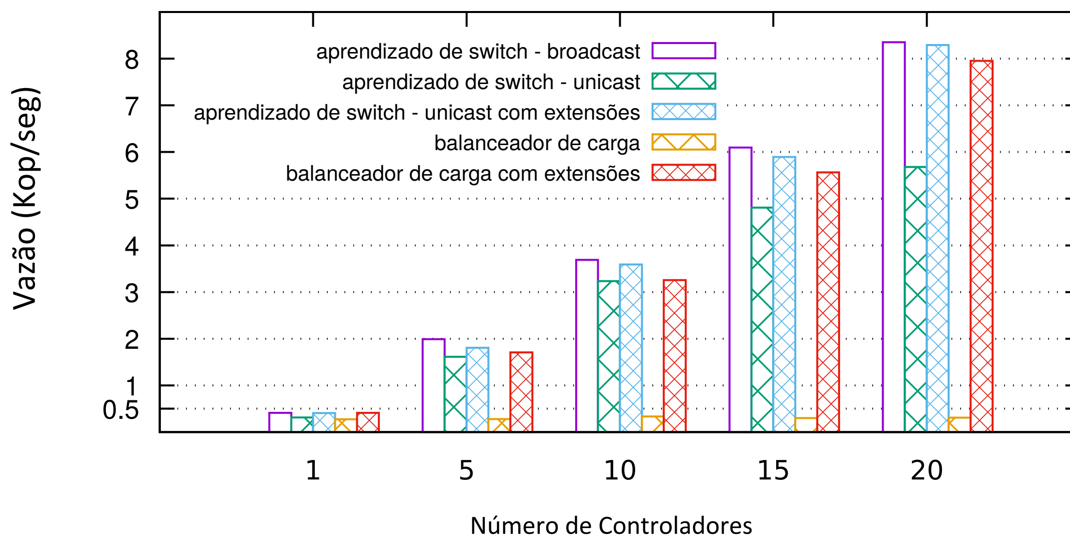

Figura 4.7: Vazão medida em Kop/seg. Percebe-se que a vazão aumenta com o número de controladores e a aplicação com maior vazão foi o aprendizado de *switch* pois é a *workload* menos custosa, pois executa apenas uma operação no DepSpace; a *workload* com menor vazão foi o balanceador de carga que tem que lidar com a concorrência nos controladores.

neamente essa carga de trabalho também aumenta a probabilidade de falha na operação de substituição (executada na etapa 3) (ou seja, retornando nulo porque as informações antigas do conjunto já foram atualizadas por um controlador simultâneo que acessa o armazenamento de dados) [101]. Felizmente, quando consideramos essa aplicação usando extensões, o desempenho é bastante aprimorado e se torna semelhante à carga de trabalho de *broadcast*. A latência introduzida por este aplicativo (aproximadamente $2,7ms$) pode ser importante ao executar solicitações leves, que exige pouco tempo para execução, mas pode ser negligenciada quando o tempo para executar uma solicitação for dominante no sistema, ou seja, não afetará o desempenho geral percebido pelos clientes.

4.4.2 Desempenho da Solução Integrada ao FloodLight

Esta seção discute o desempenho das aplicações quando a arquitetura proposta é integrada ao controlador FloodLight [58]. Primeiramente, apresentamos um experimento que demonstra que o primeiro acesso é mais custoso, visto que várias configurações devem ser adicionadas ao espaço de tuplas e conexões precisam ser estabelecidas. Depois disso, analisamos o desempenho de cada aplicação (aprendizado de máquina e balanceador de carga) nesta solução integrada.

Configuração das aplicações de teste. O ambiente para os experimentos foi constituído por 5 máquinas i5-5200 (2.2 GHz, com 4 núcleos e 2 threads por núcleo, 8GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O DepSpace foi configurado com 4 servidores para tolerar até uma falha maliciosa. Cada servidor foi executado em uma máquina separada, enquanto que um controlador (que funciona como cliente do DepSpace) foi executado na máquina restante. O ambiente de software utilizado foi o sistema operacional Ubuntu 19.10 64-bit e máquina virtual Java de 64 bits versão 1.8.0_231.

Para verificar o desempenho da arquitetura proposta foi utilizado um ambiente de rede na rede virtual do Mininet [103] e o conectamos ao controlador Floodlight [58], ao qual se conectou ao DepSpace. O trabalho executado pelo Mininet foi o envio de mensagens no plano de dados para o Floodlight (plano de controle), no qual se conectava ao DepSpace. Essas mensagens OpenFlow disparadas dos *switches* para o controlador foram mensagem de solicitação ICMP, como, por exemplo, *pings* disparados de um host para outro da rede.

Para tal teste configuramos um *switch* e dois *hosts*, para simplificar a troca de mensagens e a comunicação com o controlador e o DepSpace. Além disso, utilizamos o Mininet e o controlador Floodlight sem o DepSpace para avaliar a diferença de desempenho entre a arquitetura original e a do DepSpace.

Custos para Acessar o DepSpace. A Figura 4.8 mostra a latência para o controlador Floodlight [58] acessar o espaço de tuplas seguro, sem realizar nenhuma operação, apenas se conectar ao DepSpace. Nota-se que o primeiro acesso tem a maior latência, caindo drasticamente nos acessos subsequentes. O primeiro acesso é o de maior duração pois para acessar o DepSpace exige-se que o controlar se conecte a um armazenamento de dados distribuído, no exemplo, tal armazenamento de dados é construído sobre 4 servidores, no qual é atribuído as configurações que o DepSpace necessita, como o nome de quem está acessando, o *id*, as propriedades de confidencialidade, as configurações de armazenamento e replicação entre os servidores, e entre outras, garantido a coordenação entre todos os servidores. Percebe-se que depois do primeiro acesso, o custo para acessar o DepSpace se torna muito menor, devido as configurações já estarem armazenadas nos servidores bem como as conexões já estarem estabelecidas.

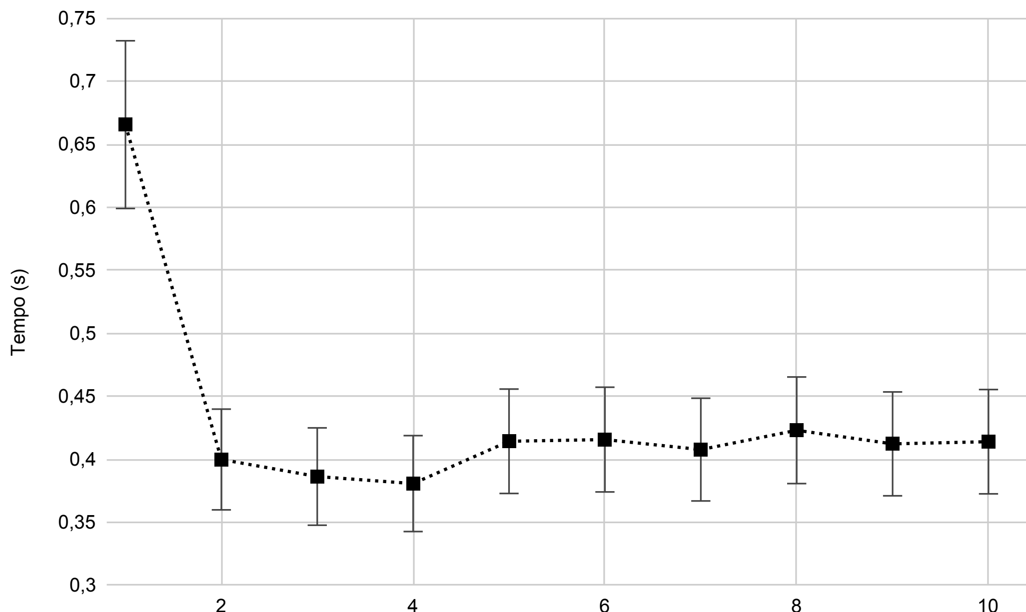

Figura 4.8: Acessos subsequentes ao DepSpace.

Aprendizado de *Switch*. A figura 4.9 apresenta os custos referentes a cada operação da aplicação Aprendizado de *Switch*. Os resultados da execução da aplicação consideram que o DepSpace já foi acessado pelo menos uma vez, como já discutido no parágrafo acima, para evitar configurações iniciais e aberturas de conexões que tem um custo mais elevado.

Observando a Figura 4.9, a barra azul se refere ao custo para, a partir do Floodlight, se conectar ao DepSpace, no qual é feito as conexões com servidores e o espaço seguro é

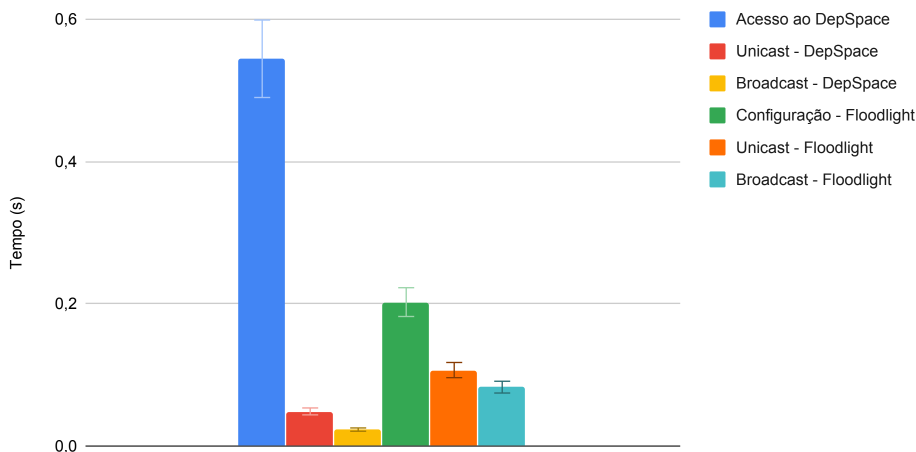

Figura 4.9: Aprendizado de *Switch* - Floodlight vs DepSpace. Apesar do tempo de acesso ao DepSpace ser o mais custoso.As operações são menos custosas que as do Floodlight pois com a utilização do DepSpace as operações são realizadas no espaço seguro, necessitando, apenas, de uma chamada ao DepSpace.

criado e mantido. As duas barras seguintes (vermelha e laranja) referem-se as operações *Unicast* e *Broadcast* da aplicação aprendizado de *switch*, respectivamente. A barra verde se refere as configurações que o Floodlight tem que realizar para configurar as opções de armazenamento e as barras laranja e azul são, respectivamente, referentes a *workload unicast* e *broadcast*. Percebe-se na Figura 4.9 que o custo das operações *Unicast* e *Broadcast* são mínimos e o maior custo é o acesso ao armazenamento de dados. Nota-se ainda que o custo para associar o endereço de origem à porta do *switch* é pequeno (cerca de 0,02 segundos) e a operação de *Unicast*, por ter uma busca adicional tem o custo mais elevado (cerca de 0,04).

Os custos referentes ao DepSpace são menores que os custos no armazenamento tradicional do Floodlight; isso ocorre porque o Floodlight utiliza operações complexas de mapeamento de dados em tabelas, utilizando operações custosas de *HashMap* disponíveis em bibliotecas do Java, verificando para cada operação de processamento de pacotes as tabelas disponíveis e atualizando as informações, enquanto que, com a utilização do DepSpace, todas as operações são realizadas no espaço seguro, necessitando, apenas, de uma chamada ao DepSpace.

A maior latência está na etapa de configuração e acesso ao DepSpace que necessita a conexão a um ambiente externo (nosso *data store*). No entanto esses custos são irrelevantes pois tais operações são executadas apenas no início da configuração da rede. Além disso, é notável que os custos para armazenar os dados no DepSpace são similares ao armazenamento de dados padrão do controlador Floodlight.

Balanceador de Carga. A Figura 4.10 apresenta os custos referentes da aplicação balanceador de carga. Os resultados da execução da aplicação considerou que o DepSpace já foi acessado pelo menos uma vez, como já discutido nos parágrafos acima, para evitar configurações iniciais que tem um custo mais elevado. Nota-se que os custos para armazenar os dados no DepSpace são similares ao armazenamento de dados padrão do controlador Floodlight, levemente maiores.

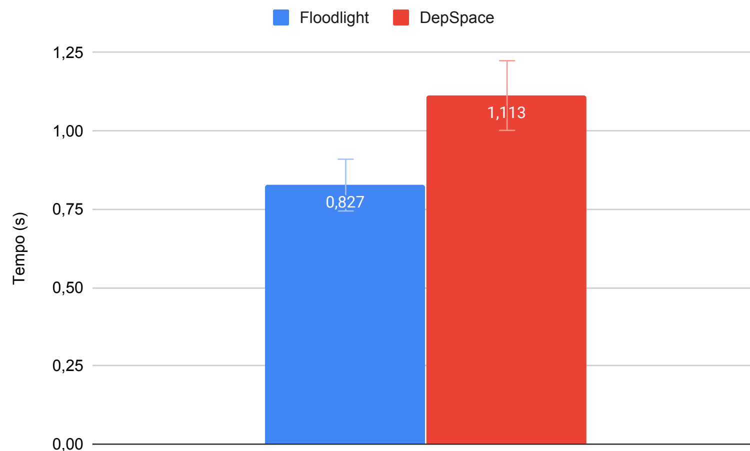

Figura 4.10: Balanceador de carga - Floodlight vs DepSpace. Nota-se que os custos para armazenar os dados no DepSpace são similares ao armazenamento de dados padrão do controlador Floodlight, levemente maiores. Porém o custo é vantajoso pois os benefícios justificam essa diferença.

4.5 Conclusão

Este capítulo apresentou uma proposta de um armazenamento de dados seguro e distribuído baseado em um espaço de tuplas que garante segurança, conhecido como DepSpace. O DepSpace foi utilizado para armazenar o estado da rede e das aplicações (i.e., Aprendizado de máquina e Balanceador de carga) instaladas no plano de controle.

A vantagem na utilização do DepSpace foi que ele proveu mecanismos de tolerância a falhas e confidencialidade, baseando-se em políticas de acesso e consistência forte para manter o plano de controle livre de falhas e dados corrompidos.

Para prova de conceito, alguns experimentos foram realizados com os mecanismos desenvolvidos e abordados no decorrer deste trabalho. Estes experimentos contemplaram algumas operações de aplicações que estão presentes em todos os ambientes de rede. A principal contribuição deste trabalho foi introduzir uma camada adicional de segurança para o cenário SDN, no qual as aplicações armazenam os dados por ela manipulados em um espaço externo que possui propriedades de segurança. Diversas aplicações podem

vir a utilizar as soluções propostas para prover um maior nível de segurança para as aplicações que necessitem de altos níveis de segurança. Experimentos mostram que com a utilização da arquitetura proposta ganha-se em segurança mantendo níveis de desempenho aceitáveis.

O principal indicador de desempenho investigado nestes testes foi a latência do sistema (tempo necessário para uma aplicação executar uma operação no sistema) com a utilização de um armazenamento seguro e distribuído de dados, o DepSpace. Os resultados experimentais mostram que a sobrecarga introduzida pela arquitetura proposta pode ser negligenciada para muitos aplicativos e cenários, mostrando sua viabilidade prática.

Capítulo 5

Conclusão

Neste capítulo é apresentado a conclusão do trabalho. Dessa forma, será apresentado uma visão geral resumida de todo o trabalho com suas principais contribuições, logo em seguida será demonstrado como foi alcançado os objetivos propostos no início e finalizando com as perspectivas de trabalhos futuros.

5.1 Visão Geral do Trabalho

O objetivo geral desse trabalho é projetar, implementar e avaliar um modelo seguro e consistente para plano de controle de uma rede SDN, utilizando o DepSpace, que é um espaço de tuplas com propriedades de segurança, para armazenamento seguro e consistente das políticas e informações relevantes de uma rede.

O plano de controle seguro é importante para manter os dados das aplicações SDN de forma segura, evitando diversos problemas de segurança que podem comprometer a confiabilidade de uma rede. Com a utilização do DepSpace, o serviços SDN tem uma camada adicional de segurança para manter as aplicações que dele utilizam dentro das suas especificações.

Para subsidiar os trabalhos da dissertação, esta foi dividida em três partes fundamentais. A primeira parte compreende a fundamentação teórica, explanando os aspectos de segurança da informação, sistemas distribuídos, modelos de falhas, modelos de consistência, espaço de tuplas e as características das redes SDN.

A segunda parte aborda o "Estado da Arte", no qual foi exposto as características mais detalhadas sobre os controladores, subdividindo as arquiteturas entre controladores centralizados e controladores distribuídos. Além disso, foi discutido as propriedades de segurança dos controladores e outros problemas de segurança em SDN. Todas essas características dos controladores e os aspectos de segurança foram pesquisadas nos trabalhos e artigos mais relevantes da área.

A terceira parte apresentou a proposta do trabalho, compreendendo a arquitetura de um plano de controle seguro e distribuído baseado no DepSpace. Além disso, foram avaliadas duas aplicações, a saber, aprendizado de *switch* e balanceador de carga (*load balancer*), verificando os custos de tais aplicações no sistema. Também foi utilizado um esquema de coordenação extensível para melhorar o desempenho da arquitetura proposta através da diminuição da quantidade de acessos ao DepSpace.

5.2 Revisão dos Objetivos e Contribuições desta Dissertação

Abaixo será revisitado os objetivos propostos no início da dissertação e elencando como foram alcançados. Iniciando pelos objetivos específicos temos que:

- *Estudar os conceitos relevantes para o trabalho, relacionados com Redes Definidas por Software, Segurança Computacional e Sistemas Distribuídos.*

  O Capítulo 2 apresenta os conhecimentos e conceitos fundamentais relacionados com o trabalho. Além disso, o Capítulo 3 apresenta uma revisão do estado da arte.

- *Projetar e implementar uma arquitetura que introduz segurança aos dados manipulados pelo plano de controle através da utilização do* DepSpace.

  O capítulo 4 apresentou o projeto e a implementação da arquitetura proposta para introduzir segurança aos dados manipulados pelos controladores e aplicações de rede.

- *Projetar, implementar e avaliar o desempenho de aplicações de rede que utilizem esta arquitetura, comparando-as com a arquitetura tradicional, sem mecanismos de segurança.*

  O capítulo 4 apresentou o projeto e implementação de duas aplicações de rede que utilizam a arquitetura proposta: aprendizado de *switch* e balanceador de carga. Além disso, este capítulo apresentou alguns experimentos que compararam o desempenho da arquitetura proposta com o que já acontece em um controlador sem utilizar tal armazenamento.

- *Refinar a arquitetura e as aplicações anteriormente propostas, incluindo a utilização de mecanismos de coordenação extensível presentes no* DepSpace.

  O capítulo 4 também tratou da utilização de coordenação extensível, presente no DepSpace, visando aumentar o desempenho do sistema.

- *Avaliar o desempenho de aplicações de rede utilizando coordenação extensível.*

  Por fim, também no Capítulo 4 foi avaliado o desempenho da utilização da nossa arquitetura proposta com a utilização das extensões, verificamos um aumento considerável no desempenho das aplicações.

Com base nesses objetivos específico acima, acreditamos que o objetivo geral foi alcançado. Corroborando isso, uma parte das contribuições acima listadas foram publicadas nos seguintes artigos:

1. da Silva J.P., Alchieri E., Bordim J., Costa L. (2020) A Secure and Distributed Control Plane for Software Defined Networks. In: Barolli L., Amato F., Moscato F., Enokido T., Takizawa M. (eds) Advanced Information Networking and Applications. AINA 2020. Advances in Intelligent Systems and Computing, vol 1151. Springer 2020 [104]; e

2. da Silva J.P, Alchieri E., Bordim, Jacir L e Gondim João J. Um Plano de Controle Seguro e Distribuído para Redes Definidas por Software. Workshop de Testes e Tolerância a Falhas. SBC 2019. [105].

5.3 Perspectivas Futuras

Em geral, acreditamos que este trabalho pode render diversos insumos para pesquisas com foco em segurança e desempenho em redes definidas por *software* que utilizam a abordagem de múltiplos controladores. Além disso, como alternativas para pesquisas futuras, uma realização de testes maiores, envolvendo mais aplicações e maiores tráfegos de rede seria interessante para verificarmos o desempenho em outros cenários. Por exemplo, um trabalho viável seria a execução dos testes do Capítulo 4 em um sistema de larga escala como a Internet.

Outra proposta para trabalhos futuros é a definição e implementação de um protocolo no plano de dados para verificar assinaturas criadas através de um conjunto de controladores, garantindo assim a confiabilidade entre os dois planos mesmo na presença de alguns controladores maliciosos (invadidos).

Referências

[1] Kshemkalyani, Ajay D. e Mukesh Singhal: *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, USA, 1ª edição, 2008, ISBN 0521876346. x, 14, 15

[2] Bessani, Alysson, Eduardo Alchieri, Miguel Correia e Joni da Silva Fraga: *DepSpace: A byzantine fault-tolerant coordination service.* European Conference on Computer Systems, 2008. x, 2, 16, 17, 18, 19, 65

[3] Tootoonchian, A. e Y. Ganjali: *Hyperflow: A distributed control plane for openflow.* in Proc. Internet Netw. Manag. Conf. Res. Enterprise Netw. (INM/WREN), página 3, 2010. x, 30, 34, 36, 42, 45, 52

[4] Gude, Natasha, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick Mckeown e Scott Shenker: *Nox: towards an operating system for networks.* ACM SIGCOMM Computer Communication Review, 2008. x, 29, 31, 45

[5] al., P. Berde et: *Onos: Towards an open, distributed sdn os.* in Proc. 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN), páginas 1–6, 2014. x, 13, 38, 39, 42, 45

[6] Jain, Sushant, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu *et al.*: *B4: Experience with a globally-deployed software defined wan.* ACM SIGCOMM Computer Communication Review, 43(4):3–14, 2013. x, 39, 40, 42

[7] Castro, Flavio: *Espresso – more insights into google's sdn.* `https://sdn-lab.com/2017/08/16/espresso-more-insights-into-googles-sdn/`. x, 40, 41, 42

[8] Botelho, Fábio, Alysson Bessani, Fernando Ramos e Paulo Ferreira: *Smartlight: A practical fault-tolerant sdn controller.* arXiv preprint arXiv:1407.6062, 2014. x, 13, 42, 43, 45

[9] Foundation, Open Network: *Openflow switch specification 1.0.* ONF, 1:1–44, december 2009. x, 22, 23, 26, 46

[10] George Couloris, Jean Dollimore, Tim Kindberg: *Sistemas Distribuídos - Conceitos e Projeto.* Bookman, fourth edição, 2007. ISBN 978-85-60031-49-8. xii, 8, 10, 12

[11] Hu, Tao, Zehua Guo, Thar Baker e Julong Lan: *Multi-controller based software-defined networking: A survey.* IEEE Access, PP:1–1, março 2018. 1

[12] Botelho, F. A., F. M. V. Ramos, D. Kreutz e A. N. Bessani: *On the feasibility of a consistent and fault-tolerant data store for sdns.* Em *2013 Second European Workshop on Software Defined Networks*, páginas 38–43, Oct 2013. 1, 2, 56

[13] Botelho, F., T. A. Ribeiro, P. Ferreira, F. M. V. Ramos e A. Bessani: *Design and implementation of a consistent data store for a distributed sdn control plane.* Em *2016 12th European Dependable Computing Conference (EDCC)*, páginas 169–180, Sep. 2016. 1, 2

[14] Eko Oktian, Yustus, SangGon Lee, HoonJae Lee e JunHuy Lam: *Distributed sdn controller system: A survey on design choice.* Computer Networks, 121, abril 2017. 2, 45

[15] Floriano, Edson, Eduardo Alchieri, Diego Aranha e Priscila Solis: *Privacidade em dados armazenados em memória compartilhada através de espaços de tupla.* Em *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2017. 2, 17, 18, 19

[16] Floriano, Edson, Eduardo Alchieri, Diego Aranha e Priscila Solis: *Providing privacy on the tuple space model.* Journal of Internet Services and Applications, 8(19):1–16, 2017. 2, 17, 18, 19

[17] Howard, John Douglas: *An Analysis of Security Incidents on the Internet 1989-1995.* Tese de Doutoramento, Carnegie Mellon University, Pittsburgh, PA, USA, 1998. UMI Order No. GAX98-02539. 5

[18] BRINKLEY, D. L.; SCHELL, R. R.: *Concepts and terminology for computer security.* IEEE Computer Society Press, Information security: an integrated collection of essays:40–97, 1995. Los Alamitos, CA. 6

[19] Avizienis, Algirdas, J C Laprie, Brian Randell e Carl Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE transactions on dependable and secure computing, 1(1):11–33, 2004. 6, 7, 17

[20] ISO-IS: *7498-2 information processing systems open systems interconnection basic reference model.* ISO, 1989. Geneva, Switzerland. 6

[21] Kshemkalyani, Ajay e Mukesh Singhal: *Distributed Computing Principles, Algorithms, and Systems.* Cambridge, 2008. 8

[22] Lamport, Leslie e Nancy Lynch: *Chapter on Distributed Computing.* Cambridge, 1989. 8

[23] Hadzilacos, Vassos e Sam Toueg: *A modular approach to fault-tolerant broadcasts and related problems.* Relatório Técnico, Cornell University, 1994. 9, 10

[24] Anderson, Ross J.: *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley Publishing, 2ª edição, 2008, ISBN 9780470068526. 10

[25] Vassos Hadzilacos, Sam Toueg: *A modular approach to fault-tolerant broadcasts and related problems.* Computer science; technical report, 1994. 10, 11

[26] Koponen: *Onix: A distributed control platform for large-scale production networks.* in Proc. 9th USENIX Conf. Oper. Syst. Design Implement. (OSDI), páginas 351–364, 2010. 13, 35, 42, 45

[27] Lamport, Leslie, Robert Shostak e Marshall Pease: *The byzantine generals problem.* ACM Trans. Program. Lang. Syst., 4(3):382–401, julho 1982, ISSN 0164-0925. `https://doi.org/10.1145/357172.357176`. 13

[28] Castro, Miguel e Barbara Liskov: *Practical byzantine fault tolerance and proactive recovery.* ACM Trans. Comput. Syst., 20(4):398–461, novembro 2002, ISSN 0734-2071. `https://doi.org/10.1145/571637.571640`. 13

[29] Diamantopoulos, Panos, Stathis Maneas, Christos Patsonakis, Nikos Chondros e Mema Roussopoulos: *Interactive consistency in practical, mostly-asynchronous systems.* Em *2015 IEEE 21St International Conference On Parallel And Distributed Systems (ICPADS)*, páginas 752–759. IEEE, 2015. 14

[30] Gelernter, David: *Generative Communication in Linda.* ACM Transactions on Programing Languages and Systems, 7(1):80–112, janeiro 1985. 15, 16

[31] Distler, Tobias, Christopher Bahn, Alysson Bessani, Frank Fischer e Flavio Junqueira: *Extensible distributed coordination.* Em *Proc. of 10th European Conference on Computer Systems*, 2015. 16

[32] Bakken, David E. e Richard D. Schlichting: *Supporting Fault-Tolerant Parallel Programing in Linda.* IEEE Transactions on Parallel and Distributed Systems, 6(3):287–302, março 1995. 16

[33] Segall, Edward J.: *Resilient distributed objects: Basic results and applications to shared spaces.* Em *Proceedings of the 7th Symposium on Parallel and Distributed Processing*, 1995. 16

[34] Schneider, Fred B.: *Implementing fault-tolerant service using the state machine aproach: A tutorial.* ACM Computing Surveys, 22(4):299–319, dezembro 1990. 18, 55

[35] Castro, Miguel e Barbara Liskov: *Practical Byzantine fault-tolerance and proactive recovery.* ACM Transactions Computer Systems, 20(4):398–461, novembro 2002. 18

[36] Bessani, Alysson, João Sousa e Eduardo Alchieri: *State machine replication for the masses with BFT-SMaRt.* Em *International Conference on Dependable Systems and Networks*, 2014. 18, 45

[37] Schoenmakers, Berry: *A simple publicly verifiable secret sharing scheme and its application to electronic voting.* Em *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, páginas 148–164, agosto 1999. 18

[38] Boneh, Dan e Victor Shoup: *A graduate course in applied cryptography.* `https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf`, 2015. 18

[39] Boldyreva, Alexandra, Nathan Chenette, Younho Lee e Adam O'Neill: *Order-preserving symmetric encryption.* Cryptology ePrint Archive, Report 2012/624, 2012. 19

[40] Boneh, Dan, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry e Joe Zimmerman: *Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation.* Cryptology ePrint Archive, Report 2014/834, 2014. 19

[41] Lewi, Kevin e David J. Wu: *Order-revealing encryption: New constructions, applications, and lower bounds.* Em *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, páginas 1167–1178, 2016, ISBN 978-1-4503-4139-4. 19

[42] Tourky, D., M. ElKawkagy e A. Keshk: *Homomorphic encryption the "holy grail" of cryptography.* Em *2nd IEEE Conference on Computer and Communications*, 2016. 19

[43] Naehrig, Michael, Kristin Lauter e Vinod Vaikuntanathan: *Can homomorphic encryption be practical?* Em *Proceedings of 3rd Workshop on Cloud Computing Security Workshop*, 2011. 19

[44] Analytics, N1: *A java library for paillier partially homomorphic encryption.* GitHub, 2017. `https://github.com/n1analytics/javallier`. 19

[45] Paillier, Pascal: *Public-key cryptosystems based on composite degree residuosity classes.* Em *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, páginas 223–238, 1999, ISBN 3-540-65889-0. `http://dl.acm.org/citation.cfm?id=1756123.1756146`. 19

[46] Alves, Pedro Geraldo Morelli Rodrigues e Diego F. Aranha: *A framework for searching encrypted databases.* Em *XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2016)*, páginas 142–155. SBC, 2016. 19

[47] Bessani, Alysson Neves, Miguel Correia, Joni Silva Fraga e Lau Cheuk Lung: *Sharing memory between Byzantine processes using policy-enforced tuple spaces.* Em *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, julho 2006. 20

[48] Nadeau, T.D. e K. Gray: *SDN: Software Defined Networks.* O'Reilly, 2013, ISBN 9781449342302. `https://books.google.com.br/books?id=l01LmwEACAAJ`. 21, 25

[49] Kreutz, Diego, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky e Steve Uhlig: *Software-defined networking: A comprehensive survey.* Proceedings of the IEEE, 103:14–76, 2014. 21, 24, 35, 46

[50] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker e Jonathan Turner: *Openflow: Enabling innovation in campus networks*. SIGCOMM Comput. Commun. Rev., 38(2):69–74, março 2008, ISSN 0146-4833. `http://doi.acm.org/10.1145/1355734.1355746`. 26

[51] Klöti, Rowan, Vasileios Kotronis e Paul Smith: *Openflow: A security analysis*. Em *OpenFlow: A Security Analysis*, outubro 2013. 26

[52] Casado, M., T. Koponen, R. Ramanathan e S. Shenker: *Virtualizing the network forwarding plane*. WORKSHOP ON PROGRAMMABLE ROUTERS FOR EXTENSIBLE SERVICES OF TOMORROW, PRESTO, 2010. 28

[53] Tootoonchian, Amin, Sergey Gorbunov, Yashar Ganjali, Martin Casado e Rob Sherwood: *On controller performance in software-defined networks*. Em *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, San Jose, CA, abril 2012. USENIX Association. `https://www.usenix.org/conference/hot-ice12/workshop-program/presentation/tootoonchian`. 29, 30, 45

[54] Cai, Zheng, Alan L. Cox e T. S. Eugene Ng: *Maestro: A system for scalable openflow control*. Em *Rice*, 2010. 29, 31, 50

[55] Erickson, David: *The beacon openflow controller*. Em *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, páginas 13–18, 2013. 29, 31, 45

[56] *Documentation pox*. `https://noxrepo.github.io/pox-doc/html/`. 29

[57] *Sdn series part four: Ryu, um controlador sdn de código aberto de recursos avançados, suportado pela ntt labs - the new stack*. `https://thenewstack.io`. 29

[58] Networks, Big Switch: *Floodlight openflow controller*. `http://www.projectfloodlight.org/floodlight/`. 29, 32, 45, 67, 68

[59] Banikazemi, Mohammad, David Olshefski, Anees Shaikh, John Tracey e Guohui Wang: *Meridian: An sdn platform for cloud network services*. Communications Magazine, IEEE, 51:120–127, fevereiro 2013. 29, 33, 45

[60] Wang, F., H. Wang, B. Lei e W. Ma: *A research on high-performance sdn controller*. Em *2014 International Conference on Cloud Computing and Big Data*, páginas 168–174, Nov 2014. 32, 45

[61] *Ryu sdn framework*. `https://osrg.github.io/ryu/`. 32, 45

[62] *Big switch - networks*. `https://www.bigswitch.com/company/contact?source=footer&source-page=labs`. 33

[63] Switch, Big: *Indigo*. URL: http://www. projectfloodlight. org/indigo, 2020. 33

[64] Media, O'Reilly: *What is devops?* `http://shop.oreilly.com/product/0636920026822.do`. 34

[65] Foundation, Open Network: *Openflow switch specification*, 2011. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf, acesso em 2019-03-28. 34

[66] Bannour, Fetia, Sami Souihi e Abdelhamid Mellouk: *Distributed sdn control: Survey, taxonomy, and challenges.* IEEE Communications Surveys & Tutorials, 20(1):333–354, 2018. 35

[67] Foundation, OpenDaylight: *Opendaylight*, 2019. https://www.opendaylight.org/, acesso em 2019-03-20. 37, 42, 45

[68] Mishra, Vivek: *Titan graph databases with cassandra.* Em *Beginning Apache Cassandra Development*, páginas 123–151. Springer, 2014. 38

[69] Foundation., The Apache Software: *Apache cassandra documentation v4.0-alpha4.* https://cassandra.apache.org/doc/latest/. 38

[70] Hunt, Patrick, Mahadev Konar, Flavio Paiva Junqueira e Benjamin Reed: *Zookeeper: Wait-free coordination for internet-scale systems.* Em *USENIX annual technical conference*, volume 8, 2010. 38

[71] Hassas Yeganeh, Soheil e Yashar Ganjali: *Kandoo: a framework for efficient and scalable offloading of control applications.* Em *Proceedings of the first workshop on Hot topics in software defined networks*, páginas 19–24, 2012. 38, 42, 45

[72] Santos, Mateus AS, Bruno AA Nunes, Katia Obraczka, Thierry Turletti, Bruno T De Oliveira e Cintia B Margi: *Decentralizing sdn's control plane.* Em *39th Annual IEEE Conference on Local Computer Networks*, páginas 402–405. IEEE, 2014. 40, 42

[73] Spalla, Eros S, Diego R Mafioletti, Alextian B Liberato, Gilberto Ewald, Christian E Rothenberg, Lasaro Camargos, Rodolfo S Villaca e Magnos Martinello: *Ar2c2: Actively replicated controllers for sdn resilient control plane.* Em *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, páginas 189–196. IEEE, 2016. 43

[74] Yeganeh, Soheil Hassas e Yashar Ganjali: *Beehive: Simple distributed programming in software-defined networks.* Em *Proceedings of the Symposium on SDN Research*, páginas 1–12, 2016. 44

[75] Heller, B, R Sherwood e N McKeown: *The controller placement problem, proceedings of the first workshop on hot topics in software-defined networks.* Hot topics in software-defined networks, 2012. 44, 51, 52

[76] Yao, Guang, Jun Bi, Yuliang Li e Luyi Guo: *On the capacitated controller placement problem in software defined networks.* IEEE Communications Letters, 18(8):1339–1342, 2014. 44

[77] Schiff, Liron, Stefan Schmid e Petr Kuznetsov: *In-band synchronization for distributed sdn control planes.* ACM SIGCOMM Computer Communication Review, 46(1):37–43, 2016. 44

[78] Botelho, Fábio, Tulio A Ribeiro, Paulo Ferreira, Fernando MV Ramos e Alysson Bessani: *Design and implementation of a consistent data store for a distributed sdn control plane*. Em *"2016 12th European Dependable Computing Conference (EDCC)"*, páginas 169–180. IEEE, 2016. 44

[79] Panda, Aurojit, Colin Scott, Ali Ghodsi, Teemu Koponen e Scott Shenker: *Cap for networks*. Em *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, páginas 91–96, 2013. 44

[80] Holz, Ralph, Thomas Riedmaier, Nils Kammenhuber e Georg Carle: *X. 509 forensics: Detecting and localising the ssl/tls men-in-the-middle*. Em *european symposium on research in computer security*, páginas 217–234. Springer, 2012. 47

[81] Canvel, Brice, Alain Hiltgen, Serge Vaudenay e Martin Vuagnoux: *Password interception in a ssl/tls channel*. Em *Annual International Cryptology Conference*, páginas 583–599. Springer, 2003. 47

[82] Ahmad, I., S. Namal, M. Ylianttila e A. Gurtov: *Security in software defined networks: A survey*. IEEE Communications Surveys Tutorials, 17(4):2317–2346, 2015. 47

[83] Hartman, S, M Wasserman e D Zhang: *Security requirements in the software defined networking model*. Internet Engineering Task Force, Internet-Draft draft-hartman-sdnsec-requirements-01, 2013. 47

[84] Naous, Jad, David Erickson, G Adam Covington, Guido Appenzeller e Nick McKeown: *Implementing an openflow switch on the netfpga platform*. Em *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, páginas 1–9, 2008. 47

[85] Jarschel, Michael, Simon Oechsner, Daniel Schlosser, Rastin Pries, Sebastian Goll e Phuoc Tran-Gia: *Modeling and performance evaluation of an openflow architecture*. Em *2011 23rd International Teletraffic Congress (ITC)*, páginas 1–7. IEEE, 2011. 47

[86] Shin, Seungwon, Vinod Yegneswaran, Phillip Porras e Guofei Gu: *Avant-guard: Scalable and vigilant switch flow management in software-defined networks*. Em *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, páginas 413–424, 2013. 48

[87] Shin, Seungwon e Guofei Gu: *Attacking software-defined networks: A first feasibility study*. Em *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, páginas 165–166, 2013. 48

[88] Fonseca, Paulo, Ricardo Bennesby, Edjard Mota e Alexandre Passito: *A replication component for resilient openflow-based networking*. Em *2012 IEEE Network operations and management symposium*, páginas 933–939. IEEE, 2012. 48

[89] Al-Shaer, Ehab e Saeed Al-Haj: *Flowchecker: Configuration analysis and verification of federated openflow infrastructures*. Em *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, páginas 37–44, 2010. 48

[90] *Controlador do floodlight openflow - projeto floodlight.* `http://www.projectfloodlight.org/floodlight/`. 49, 52

[91] Fernandez, Marcial P: *Comparing openflow controller paradigms scalability: Reactive and proactive.* Em *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, páginas 1009–1016. IEEE, 2013. 50, 52

[92] Voellmy, Andreas e Junchang Wang: *Scalable software defined network controllers.* Em *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, páginas 289–290, 2012. 50

[93] Kohonen, Teuvo: *The self-organizing map.* Proceedings of the IEEE, 78(9):1464–1480, 1990. 51

[94] Braga, Rodrigo, Edjard Mota e Alexandre Passito: *Lightweight ddos flooding attack detection using nox/openflow.* Em *IEEE Local Computer Network Conference*, páginas 408–415. IEEE, 2010. 51, 52

[95] Hu, Yannan, Wendong Wang, Xiangyang Gong, Xirong Que e Shiduan Cheng: *On reliability-optimized controller placement for software-defined networks.* China Communications, 11(2):38–54, 2014. 51, 52

[96] Hu, Yannan, Wang Wendong, Xiangyang Gong, Xirong Que e Cheng Shiduan: *Reliability-aware controller placement for software-defined networks.* Em *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, páginas 672–675. IEEE, 2013. 51, 52

[97] Bari, Md Faizul, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed e Raouf Boutaba: *Dynamic controller provisioning in software defined networks.* Em *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, páginas 18–25. IEEE, 2013. 51

[98] Hock, David, Matthias Hartmann, Steffen Gebert, Michael Jarschel, Thomas Zinner e Phuoc Tran-Gia: *Pareto-optimal resilient controller placement in sdn-based core networks.* Em *Proceedings of the 2013 25th International Teletraffic Congress (ITC)*, páginas 1–9. IEEE, 2013. 51

[99] Ge, JingGuo, Hanji Shen, E Yuepeng, Yulei Wu e Junling You: *An openflow-based dynamic path adjustment algorithm for multicast spanning trees.* Em *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, páginas 1478–1483. IEEE, 2013. 52

[100] Kempf, James, Elisa Bellagamba, András Kern, David Jocha, Attila Takács e Pontus Sköldström: *Scalable fault management for openflow.* Em *2012 IEEE International Conference on Communications (ICC)*, páginas 6606–6610. IEEE, 2012. 52

[101] Junior, Edson Floriano S., Eduardo Alchieri, Diego F. Aranha e Priscila Solis: *Building secure protocols for extensible distributed coordination through secure extensions.* Computers & Security, 87:101583, 2019, ISSN 0167-4048. 62, 65, 67

[102] White, Brian, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb e Abhijeet Joglekar: *An Integrated Experimental Environment for Distributed Systems and Networks.* Em *Proc. of 5th Symp. on Operating Systems Design and Implementations*. ACM, 2002. 64

[103] Mininet: *Mininet: Rapid prototyping for software defined networks.* `https://github.com/mininet/mininet`, 2020. 67

[104] Silva, Jefferson Pereira da, Eduardo Alchieri, Jacir Bordim e Lucas Costa: *A secure and distributed control plane for software defined networks.* Em Barolli, Leonard, Flora Amato, Francesco Moscato, Tomoya Enokido e Makoto Takizawa (editores): *Advanced Information Networking and Applications*, páginas 994–1006, Cham, 2020. Springer International Publishing, ISBN 978-3-030-44041-1. 74

[105] Silva, Jefferson, Eduardo Alchieri, Jacir Bordim e Joao Gondim: *Um plano de controle seguro e distribuído para redes definidas por software.* Em *Anais do XX Workshop de Testes e Tolerância a Falhas*, páginas 20–33. SBC, 2019. 74