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

Jefferson Pereira da Silva, Eduardo Alchieri, Jacir Bordim e João Gondim

[1] Departamento de Ciência da Computação – Universidade de Brasília – UnB

Abstract. *Software Defined Networks (SDN) emerged as as new paradigm for network management, defining an architecture the 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 an 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 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.*

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 na rede. Consequentemente, soluções considerando um modelo de dados consistente 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. Resultados experimentais mostram a viabilidade prática da arquitetura proposta.*

1. Introdução

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

*Este projeto foi parcialmente financiado pelo MCTIC/RNP/CTIC através do projeto P4Sec.

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 da 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 divisão de responsabilidades, 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 proteções 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 possuir capacidade computacional suficiente para atender a todas as demandas encaminhadas para o mesmo. Por exemplo, sempre que um *switch* receber 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 *switchs*. Além disso, um controlador centralizado representa um ponto único de falha [Hu et al. 2018, Botelho et al. 2013, Botelho et al. 2016] 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 [Botelho et al. 2016]. Apesar desta abordagem apresentar um bom desempenho, ela apresenta pelo menos dois sérios problemas [Botelho et al. 2016]: 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 em 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 consistente para gerenciamento e armazenamento das informações sobre o estado da rede SDN começaram a aparecer na literatura [Eko Oktian et al. 2017, Botelho et al. 2013, Botelho et al. 2016]. 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, este trabalho apresenta nossos esforços no projeto, implementação e avaliação de um modelo seguro e consistente para plano de controle. Em nossa abordagem usaremos o DEPSPACE [Bessani et al. 2008, Floriano et al. 2017a, Floriano et al. 2017b], que é um espaço

de tuplas com propriedades de segurança, para armazenamento seguro e consistente das informações relevantes da rede SDN.

O restante deste texto está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica sobre redes SDN e o DEPSPACE. A Seção 3 apresenta nossa proposta para um plano de controle seguro e distribuído baseado no DEPSPACE. A Seção 4 discute alguns experimentos executados com dois serviços de rede criados sobre a arquitetura proposta. Finalmente, a Seção 5 conclui este trabalho.

2. Fundamentação Teórica

2.1. Redes Definidas por Software

Software Defined Network (SDN) ou Redes Definidas por Software é um novo paradigma em infraestrutura de redes que traz o conceito de virtualização da infraestrutura de redes, permitindo que ela seja programável e consciente, através desta característica, do estado geral da rede e das aplicações. 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 [Kreutz et al. 2014]:

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 interage com os dispositivos de plano de dados subjacentes.

O plano de dados contém os elementos de encaminhamento, aos quais diferentes hosts estão conectados. Os elementos de encaminhamento usam informações em sua chamada tabela de fluxo para descobrir para onde encaminhar pacotes.

O plano de controle é a camada em que todos os cálculos são feitos e as decisões são tomadas. O controlador tem conhecimento da topologia de rede, executa roteamento e informa os elementos de encaminhamento na camada abaixo, atualizando suas tabelas de fluxo. Isso é feito usando a interface *southbound*, através de um protocolo como o *OpenFlow* [McKeown et al. 2008]. Os controladores implementam a lógica de controle

que será traduzida em comandos a serem instalados no plano de dados, ditando o comportamento dos dispositivos de encaminhamento. A interface de comunicação entre o plano de controle e o plano de aplicação é chamada de *northbound*.

O protocolo *OpenFlow* é o responsável por toda a comunicação no plano de dados e deste com o plano de controle através da interface *southbound*. Este protocolo foi padronizado em 31 de dezembro de 2009 com a publicação de suas especificações pelo *Open Networking Foundation (ONF)*, uma organização dedicada à promoção e adoção de redes definidas por software.

2.1.1. Vulnerabilidades em Redes Definidas por Software

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, como as apresentadas abaixo [Kreutz et al. 2014].

1. Fluxos de tráfego forjados ou falsificados, nos quais um atacante pode usar elementos de rede para iniciar um ataque a um *switch* ou controlador.
2. Ataques a vulnerabilidades do *switch*, nos quais 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 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, respectivamente.
5. Falta de mecanismos para garantir a confiança entre o controlador e os aplicativos de gerenciamento, sendo necessário haver mecanismos para estabelecer comunicações seguras dos aplicativos para o controlador.
6. Ataques e vulnerabilidades em estações administrativas, as quais já são um problema de segurança das redes tradicionais e também estão presentes no ambiente SDN.
7. Falta de recursos confiáveis para análise forense e remediação, para entender as causas dos problemas e as formas de recuperação.

2.2. Espaço de Tuplas

Um espaço de tuplas [Gelernter 1985] 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 $\langle \text{WTF}, 2019, \text{Gramado} \rangle$ combina com o molde $\langle \text{WTF}, 2019, * \rangle$ ('$*$' 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 [Gelernter 1985]: $out(t)$ 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. 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. Uma extensão comum a este modelo é a inclusão das seguintes operações [Bessani et al. 2008, Distler et al. 2015, Bakken and Schlichting 1995, Segall 1995]: $cas(\bar{t}, t)$ 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 $\bar{t}$.

2.2.1. DepSpace

Segurança é uma característica fundamental de sistemas confiáveis [Avizienis et al. 2004]. No contexto de um espaço de tuplas, os seguintes atributos são necessários: *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), *confidencialidade* (o conteúdo dos campos das tuplas não pode ser revelado a partes não autorizadas). O DEPSPACE [Bessani et al. 2008, Floriano et al. 2017a, Floriano et al. 2017b] consiste na implementação de um espaço de tuplas que busca satisfazer estas propriedades por meio de diversas camadas, descritas abaixo.

Replicação. Para manter a consistência do espaço de tuplas, o DEPSPACE utiliza replicação Máquina de Estados [Schneider 1990, Castro and Liskov 2002, Bessani et al. 2014]. 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) [Schoenmakers 1999]. 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 [Bessani et al. 2008, Floriano et al. 2017a, Floriano et al. 2017b]:

- 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 [Boneh and Shoup 2015] 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 [Boldyreva et al. 2012, Boneh et al. 2014, Lewi and Wu 2016] 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* [Lewi and Wu 2016]. 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 [Tourky et al. 2016] ou um parcialmente homomórfico [Naehrig et al. 2011] 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* [Analytics 2017], que é uma implementação em Java do algoritmo de Paillier [Paillier 1999]. 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* [Alves and Aranha 2016].
- 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 1 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 [Bessani et al. 2008]. Esta classificação fornece a seguinte ordem para os campos, de acordo com o nível de segurança [Floriano et al. 2017a, Floriano et al. 2017b]: 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 1. 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 [Bessani et al. 2006], 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.

3. 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 2. 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). Como o DEPSPACE é construído de forma distribuída sobre uma camada de Replicação Máquina de Estados [Schneider 1990], um modelo de armazenamento com consistência forte e tolerante a falhas é fornecido para os controladores.

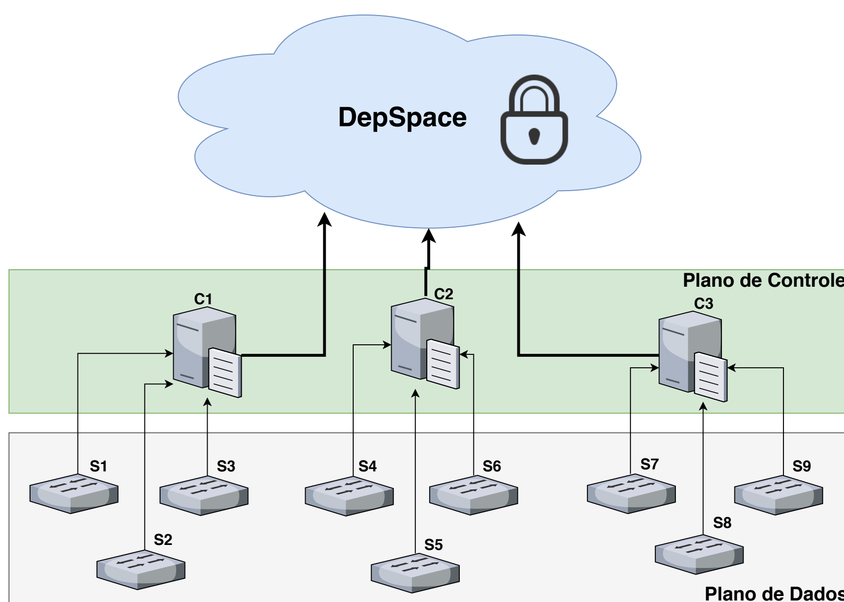

Figura 2. Plano de controle baseado no DEPSPACE.

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 2, 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 no estágio atual desta proposta, 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* utilizando as facilidades de um plano de dados programável (P4).

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.

3.1. Aplicações de Rede

Esta seção apresenta duas aplicações de rede que utilizam a arquitetura proposta: aprendizado de *switch* e balanceador de carga. Estas aplicações servem como prova de conceito a respeito de como funciona a arquitetura proposta.

3.1.1. 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 [Botelho et al. 2016]. 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 3 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 Figura 4 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.

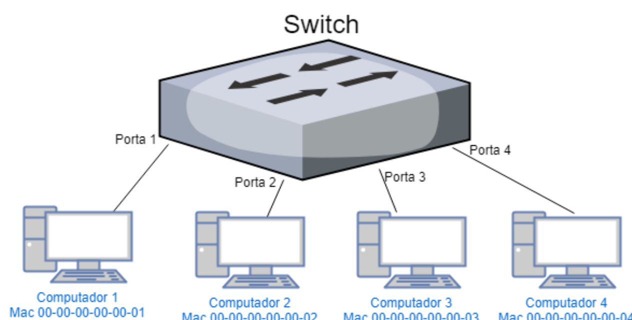

Figura 3. Aprendizado de *switch*.

O *switch* popula 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 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.

Porta	Endereço MAC
1	00-00-00-00-00-01
2	00-00-00-00-00-02

Figura 4. Tabela de mapeamento de endereços MAC para portas.

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 estiver 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. Ja 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, 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 irá verificar se conhece o endereço MAC de destino (computador 1). Neste caso, ele já possui essas informações, que foi a primeira requisição de envio. Como ele conhece as informações do computador 1, o controlador encaminhará o pacote a porta correta. Com isso ele aprendeu o endereço MAC e a porta do *switch* de dois computadores. Sabendo dessas informações, o controlador irá instalar 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.

1. *Workload* para *broadcast*: 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, armazena no DEPSPACE uma tupla que associa o endereço de origem com a respectiva porta de entrada.
2. *Workload* para *unicast*: Para um *unicast*, a aplicação aprendizado de *switch* primeiramente descobre o endereço de origem, executando um 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 devem ser configurados como comparáveis determinísticos (Seção 2.2.1), a fim de possibilitar a execução de buscas (operações rdp) e ao mesmo tempo ficarem protegidos.

3.1.2. 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 de forma exponencial, 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 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 poder implementar este serviço e decidirem como gerenciar o pacote, i.e., para qual destino encaminhá-lo. Consideramos o seguinte *workload* para esta aplicação:

- Buscar a entidade VIP associada ao endereço IP de destino. Para isso duas leituras (rdp) são necessárias: uma para buscar o identificador do VIP e outra para buscar as informações associadas com o VIP.
- Buscar o *Pool* de servidores escolhido. Com as informações do VIP, outra leitura (rdp) é necessária para buscar o *pool* de servidores.
- Atualizar as informações do *pool*. A partir das informações do *pool*, o algoritmo *round-robin* é executado para atualizar o membro atual (atributo *current-member* do *pool*) e os dados do *pool* são atualizados através da operação *replace* do DEPSPACE que troca 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).
- Ler o membro escolhido. O último passo é uma leitura (rdp) para ler as informações associadas com membro escolhido (*member-id*).

Note que consideramos que as configurações necessárias para o balanceamento de carga já tinham sido inseridas no espaço. Finalmente, vale destacar que os campos das tuplas devem ser configurados como comparáveis determinísticos (Seção 2.2.1), a fim de possibilitar a execução de buscas (operações rdp e $replace$) e ao mesmo tempo ficarem protegidos.

4. Experimentos

Visando analisar o desempenho da arquitetura proposta, protótipos das aplicações acima discutidas foram implementados e experimentos preliminares foram realizados no Emulab [White et al. 2002]. O objetivo destes experimentos não é determinar o desempenho geral da rede, mas sim determinar os custos associados ao acesso ao novo componente do plano de controle (DEPSPACE), i.e, o *overhead* introduzido no sistema.

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. A Figura 5 apresenta os custos introduzidos na arquitetura SDN, considerando as aplicações aprendizado de *switch*, tanto para o *workload broadcast* (*as-bcast*) quanto para o *workload unicast* (*as-ucast*), e balanceador de carga (*bc*). Conforme já comentado, estes custos referem-se ao *overhead* introduzido no plano de controle para manter os dados armazenados de forma segura e compartilhada no DEPSPACE, seguindo um modelo de consistência forte.

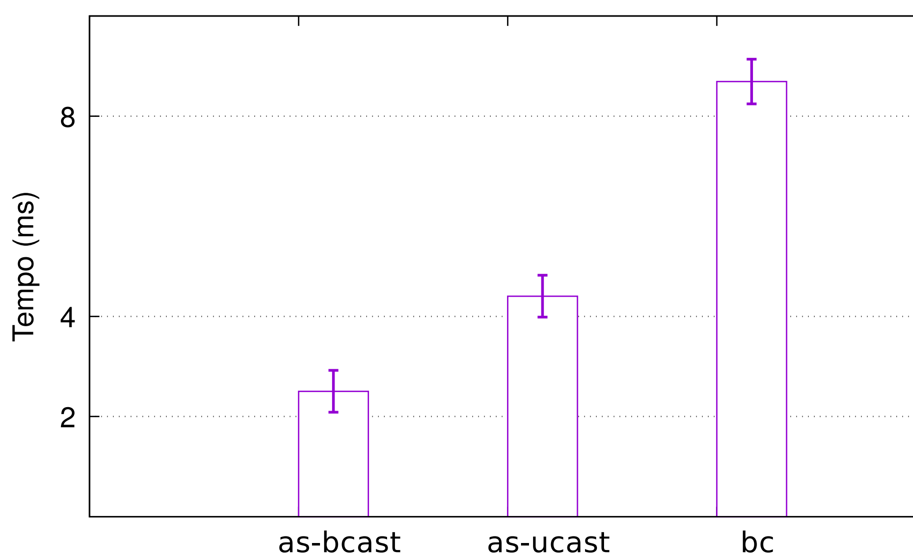

Figura 5. Custos de acesso aos dados.

A aplicação *as-bcast* apenas precisa executar um out no DEPSPACE, por isso apresenta o menor custo. Já a aplicação *as-ucast* necessita de dois acessos, um out para armazenar as dados do endeço de origem descoberto e um rdp para buscar as informações do endereço de destino, introduzindo aproximadamente o dobro do *overhead* quando comparado com o *as-bcast*. Finalmente, a aplicação *bc* precisa acessar o DEPSPACE 4 vezes (3 operações de rdp e 1 operação $replace$), introduzindo o maior *overhead*. É importante destacar que algumas aplicações de rede possuem uma característica tal que sua execução no controlador não é frequente, como por exemplo o aprendizado de *switch* que é executado apenas durante a instalação de um novo fluxo.

5. Conclusões

Apesar da arquitetura SDN trazer enormes benefícios para o gerenciamento de redes, muitas vulnerabilidades também foram introduzidas nesta arquitetura. Este trabalho apre-

sentou nossos esforços no desenvolvimento de um plano de controle distribuído e seguro, seguindo um modelo de consistência forte provido pelo DEPSPACE. Como trabalhos futuros, pretendemos explorar a camada de coordenação extensível do DEPSPACE para introduzir extensões neste sistema, as quais permitirão que as aplicações necessitem de apenas um acesso ao sistema para realizar suas operações.

Referências

Alves, P. G. M. R. and Aranha, D. F. (2016). A framework for searching encrypted databases. In *XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2016)*, pages 142–155. SBC.

Analytics, N. (2017). A java library for paillier partially homomorphic encryption. GitHub. `https://github.com/n1analytics/javallier`.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Bakken, D. E. and Schlichting, R. D. (1995). Supporting Fault-Tolerant Parallel Programing in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302.

Bessani, A., Alchieri, E., Correia, M., and da Silva Fraga, J. (2008). DepSpace: A byzantine fault-tolerant coordination service. *European Conference on Computer Systems*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *International Conference on Dependable Systems and Networks*.

Bessani, A. N., Correia, M., Fraga, J. S., and Lung, L. C. (2006). Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. (2012). Order-preserving symmetric encryption. Cryptology ePrint Archive, Report 2012/624.

Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., and Zimmerman, J. (2014). Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. Cryptology ePrint Archive, Report 2014/834.

Boneh, D. and Shoup, V. (2015). A graduate course in applied cryptography. `https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf`.

Botelho, F., Ribeiro, T. A., Ferreira, P., Ramos, F. M. V., and Bessani, A. (2016). Design and implementation of a consistent data store for a distributed sdn control plane. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 169–180.

Botelho, F. A., Ramos, F. M. V., Kreutz, D., and Bessani, A. N. (2013). On the feasibility of a consistent and fault-tolerant data store for sdns. In *2013 Second European Workshop on Software Defined Networks*, pages 38–43.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461.

Distler, T., Bahn, C., Bessani, A., Fischer, F., and Junqueira, F. (2015). Extensible distributed coordination. In *Proc. of 10th European Conference on Computer Systems*.

Eko Oktian, Y., Lee, S., Lee, H., and Lam, J. (2017). Distributed sdn controller system: A survey on design choice. *Computer Networks*, 121.

Floriano, E., Alchieri, E., Aranha, D., and Solis, P. (2017a). Privacidade em dados armazenados em memória compartilhada através de espaços de tupla. In *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Floriano, E., Alchieri, E., Aranha, D., and Solis, P. (2017b). Providing privacy on the tuple space model. *Journal of Internet Services and Applications*, 8(19):1–16.

Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Hu, T., Guo, Z., Baker, T., and Lan, J. (2018). Multi-controller based software-defined networking: A survey. *IEEE Access*, PP:1–1.

Kreutz, D., Ramos, F. M. V., Veríssimo, P., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2014). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103:14–76.

Lewi, K. and Wu, D. J. (2016). Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., J. Rexford, S. S., and Turner, J. (2008). Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication*, 38.

Naehrig, M., Lauter, K., and Vaikuntanathan, V. (2011). Can homomorphic encryption be practical? In *Proceedings of 3rd Workshop on Cloud Computing Security Workshop*.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164.

Segall, E. J. (1995). Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing*.

Tourky, D., ElKawkagy, M., and Keshk, A. (2016). Homomorphic encryption the "holy grail" of cryptography. In *2nd IEEE Conference on Computer and Communications*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*. ACM.