UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO TÉCNICO INDUSTRIAL DE SANTA MARIA
CURSO DE SUPERIOR EM REDES DE COMPUTADORES

ANÁLISE DE DEPURADORES PARA REDES DEFINIDAS POR SOFTWARE

TRABALHO DE CONCLUSÃO DE CURSO

Heitor Scalco Neto

Santa Maria, RS, Brasil

2014

ANÁLISE DE DEPURADORES PARA REDES DEFINIDAS POR SOFTWARE

Heitor Scalco Neto

Trabalho de Conclusão de Curso apresentado ao Curso de Superior em Redes de Computadores, Área de Concentração em Ciência da computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Tecnólogo em Redes de Computadores.

Orientador: Prof. Renato Preigschadt de Azevedo

Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Colégio Técnico Industrial de Santa Maria
Curso de Superior em Redes de Computadores

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

ANÁLISE DE DEPURADORES PARA REDES DEFINIDAS POR SOFTWARE

elaborada por
Heitor Scalco Neto

como requisito parcial para obtenção do grau de
Tecnólogo em Redes de Computadores

COMISSÃO EXAMINADORA:

Renato Preigschadt de Azevedo, Ms.
(Presidente/orientador)

Celio Trois, Ms. (UFSM)
(co-orientador)

Bruno Augusti Mozzaquatro, Ms. (UFSM)

Simone Regina Ceolin, Dr. (UFSM)

Santa Maria, 8 de janeiro de 2014.

Dedico este trabalho aos meus pais, José Scalco Neto e Ana Regina Pinto Scalco, pelo carinho, incentivo e confiança em mim depositada durante todos estes anos. Do pai o espelho da dedicação, perseverança e do perfil empreendedor, buscando sempre encontrar a melhor solução para tudo. Da mãe o conselho de buscar na educação um futuro promissor, estando presente em todos os momentos nos quais precisei.

AGRADECIMENTOS

Em primeiro lugar, a Deus, por me proporcionar saúde, apoio e condições adequadas para seguir essa jornada tão importante.

Ao meu irmão, Henrique Scalco, pela amizade, parceria e felicidade a mim proporcionada durante todo este tempo. De uma forma ou outra, mesmo que indiretamente, ajudou a superar as barreiras e chegar aqui.

A minha namorada, Nicole Alessandra Engel, que morando debaixo do mesmo teto, foi em todas as horas companheira e incentivadora. Por suportar a minha ausente presença nos últimos meses da realização deste trabalho. Pela calma e compreensão. Por toda a felicidade a mim fornecida durante este tempo.

Ao meu orientador, Renato Preigschadt de Azevedo, pelo incentivo e conhecimento a mim repassado.

Ao meu co-orientador, Celio Trois, que mesmo distante, conseguiu dedicar seu tempo para que este trabalho fosse realizado.

Ao professor Claiton Pereira Colvero, pelo tempo a mim disponibilizado na escrita dos artigos acadêmicos durante o tempo de graduação.

A todos os meus colegas de aula, em especial Humberto Prado e Patricia Monego, pelas manhãs de domingo de estudo e todo o conhecimento repassado, pela amizade e companheirismo durante todo o tempo de curso.

Aos meus amigos, parentes e muitos nomes que não foram citados, mas que de alguma forma contribuíram para que eu chegasse até aqui.

RESUMO

Trabalho de Conclusão de Curso
Curso de Superior em Redes de Computadores
Universidade Federal de Santa Maria

ANÁLISE DE DEPURADORES PARA REDES DEFINIDAS POR SOFTWARE

AUTOR: HEITOR SCALCO NETO
ORIENTADOR: RENATO PREIGSCHADT DE AZEVEDO
Data e Local da Defesa: Santa Maria, 8 de janeiro de 2014.

A implementação de Redes Definidas por Software (*Software Defined Networks - SDN*) proporcionam vantagens aos profissionais da área de tecnologia de redes, como a possibilidade de gerenciamento centralizado da rede. Além disso, SDNs apresentam um controlador central programável, permitindo que novas aplicações de rede sejam desenvolvidas. Ao permitir a programabilidade do controlador, Redes Definidas por Software introduzem um problema: a possibilidade de ocorrência de erros nas aplicações (*bugs*), fazendo com que a rede possa apresentar comportamento diferente do desejado. Para auxiliar na solução desse problema, existem ferramentas chamadas depuradores que executam diversos testes a procura de erros, indicando-os para que o programador possa corrigí-los. Este trabalho tem como objetivo principal, pesquisar e analisar os depuradores existentes para SDNs, selecionando algumas propostas relevantes, das quais serão apresentadas as características, vantagens e desvantagens.

Palavras-chave: SDN. OpenFlow. Depurador.

ABSTRACT

Bachelor Thesis
Curso de Superior em Redes de Computadores
Universidade Federal de Santa Maria

ANALYSIS OF DEBUGGERS TO SOFTWARE-DEFINED NETWORKS

AUTHOR: HEITOR SCALCO NETO

ADVISOR: RENATO PREIGSCHADT DE AZEVEDO

Local and date: Santa Maria, January 8, 2014.

Software Defined Networks (SDN) provides benefits to professionals in the field of network technology, as the possibility of centralized network management. Furthermore, SDNs have a programmable central controller, allowing the development of new network applications . By enabling programmability of the controller, Software Defined Networks introduces a problem: the possibility of errors in applications (*bugs*), resulting in a network not performing as expected. To help solve this problem, there are tools called debuggers that perform several tests looking for errors, showing them to the programmer. The primary goal of this research is search existing debuggers to SDNs, selecting some relevant proposals, which will cover the features, advantages and drawbacks.

Keywords: SDN. OpenFlow. Debugger.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SDN	Software Defined Networks
API	Interface de Programação de Aplicativos
$ASIC$	Application Specific Integrated Circuit
$FPGA$	Field Programmable Gate Array
IP	Protocolo de comunicação
TCP	Protocolo de comunicação da camada de Transporte no modelo OSI
STP	Spanning Tree Protocol
bug	Erro no código
$host$	Terminal (computador)
$Liveness$	Eventualmente algo bom acontece
$blackhole$	"Buraco Negro" - Destino desconhecido
$Framework$	Conjunto de classes
$Firmware$	Conjunto de instruções operacionais
$VLAN$	Rede local virtual
$Broadcast$	Utilizado para distribuir a mensagem para todos os dispositivos da rede
$Ethernet$	Arquitetura de interconexão para redes locais

SUMÁRIO

INTRODUÇÃO

Com o crescimento das redes de computadores, no passar dos anos, tornou-se inviável efetuar testes de novas arquiteturas a nível de produção, pois isso poderia acarretar em grandes perdas. Percebe-se um crescimento acentuado nas redes de computadores que ao mesmo tempo estão restringidas pelas suas próprias limitações (COSTA, 2013), ou seja, grande parte dos dispositivos implementados são desenvolvidos em um software específico (e fechado) para um hardware próprio.

Estudos realizados sobre a Internet do futuro visam resolver os problemas atuais e atender as novas demandas de requisitos estabelecidos atualmente pela Internet. Nos equipamentos de redes utilizados normalmente, as tomadas de decisão ocorrem no seu próprio interior, ou seja, o encaminhamento dos pacotes é definido por algoritmos previamente calculados, geralmente de código fechado, de difícil ou impossível modificação (COSTA, 2013).

De acordo com (NASCIMENTO et al., 2011): "A infraestrutura das redes de pacotes é composta, atualmente, por equipamentos proprietários, fechados e de alto custo, cujas arquiteturas básicas são concebidas a partir da combinação de circuitos dedicados, responsáveis por garantir alto desempenho (Application-Specific Integrated Circuit), ao processamento de pacotes." Como consequência, as redes operam como um sistema de repasse de pacotes ou fluxos de dados genéricos. Assim, percebe-se que mecanismos baseados nas características das aplicações, como a alteração de dados, filtragem de informações, não ocorrem nas redes tradicionais.

As SDNs (Redes Definidas por Software) visam resolver esses problemas. Tendo em vista o controle da rede de forma centralizada, qualquer pessoa com conhecimento de redes e programação pode configurar o encaminhamento de pacotes da forma que melhor se adapta ao seu meio de trabalho ou suas necessidades.

De acordo com (GUEDES et al., 2012) , o surgimento do protocolo OpenFlow permitiu que os elementos de encaminhamento oferecessem uma interface de programação simples, que disponibiliza ao administrador da rede, estender o acesso e controle da tabela de consulta utilizada pelo hardware, para determinar o próximo passo de cada pacote recebido. Dessa forma, o encaminhamento continua sendo eficiente, pois a consulta à tabela de encaminhamento continua sendo tarefa do hardware, mas a decisão sobre como cada pacote deve ser processado pode ser transferida para um nível superior, onde diferentes funcionalidades podem ser implementadas. Essa estrutura permite que a rede seja controlada de forma extensível através de aplicações, expressas em software. A esse novo paradigma, deu-se o nome de Redes Definidas por Software, ou, *Software Defined Networks (SDN)*.

Chama-se depuração, o processo executado por um depurador, ou *debugger*, no

qual são isoladas as causas de comportamentos incorretos de sistemas de software. Uma operação de depuração envolve fases de coleta e análise de informações em uma ou mais execuções. Os resultados coletados podem variar de acordo com a aplicação e com as ferramentas empregadas. As ações executadas pelos depuradores são geralmente bem definidas: primeiramente obter informações, em quantidade suficiente, para que seja possível reconstruir uma aproximação da execução da aplicação e analisar essa reconstrução, em busca de alguma violação de premissas capazes de produzir um comportamento incorreto no software. (MEGA; KON, 2013)

O presente trabalho tem como objetivo principal, realizar a análise de depuradores para Redes Definidas por Software. Complementarmente, realizar um estudo sobre o protocolo OpenFlow e implementá-lo em uma rede, em ambiente simulado, para que seja possível executar testes utilizando as ferramentas de depuração existentes para SDNs. Estudar as características dos depuradores para SDNs e relatar critérios como: instalação, usabilidade, número de casos de teste, documentação disponível e eficiência nos testes de caso. Pretende-se indicar dentre as características estudadas, qual depurador melhor permite a verificação de programas implementados em Redes Definidas por Software.

Primeiramente será apresentada uma criteriosa revisão bibliográfica para o entendimento sobre o funcionamento de uma SDN. Após esse processo, será relatado como foram executadas as pesquisas e testes com os depuradores propostos e, ao final, os resultados obtidos através destes testes e suas respectivas referências bibliográficas.

1 JUSTIFICATIVA

Farias (FARIAS et al., 2011) afirma que a rede deve ser o mais flexível possível, para que essa infraestrutura permita a coexistência de múltiplos modelos paralelos. Nesse sentido, a virtualização (de rede, dispositivos, enlace e etc.) e a solução OpenFlow vem se tornando uma interessante alternativa para habilitar múltiplos experimentos com novas arquiteturas em um ambiente de produção. Uma rede OpenFlow admite a programação do comportamento da rede, além da criação de ambientes de rede virtuais chamados fatias (redes virtuais com nós, enlaces e recursos) para testar novos modelos de protocolo (MCKEOWN et al., 2008).

Redes Definidas por Software são controladas por meio de programação: o estado da rede é gerenciado por programas de controle logicamente centralizados, com visão ampla da rede, e escrevem no *switch* enviando tabelas através de *APIs* padrão (por exemplo: OpenFlow). Nesse novo conceito, pode-se começar a depurar redes assim como um software: escrever e executar programas de controle, usar o depurador para verificar se existem erros e definir sequências de eventos que levam as suas causas. Redes Definidas por Software possibilitam repensar o modo de depurar redes, desde o desenvolvimento do programa até a implementação em ambiente de produção (HANDIGOL et al., 2012).

Junto com esse novo paradigma de depuração das redes, surgem várias propostas de implementação de depuradores (NICE, STS, OFRewind, TestON, entre outros) (KUZNIAR; CANINI; KOSTIC, 2012; KUZNIAR et al., 2012; AL-SHAER; AL-HAJ, 2010; TestOn, 2013). O profissional de redes de computadores precisa ter o conhecimento sobre qual é a melhor ferramenta para a sua necessidade, chegando neste impasse, será feita uma análise, com base na documentação disponível, de alguns depuradores para SDN, permitindo assim, que os desenvolvedores possam escolher mais facilmente o depurador que melhor atende as suas necessidades.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo irá abordar os conceitos existentes no paradigma SDN, apresentando conceitos sobre os principais controladores utilizados, funcionamento do protocolo, divisores de recursos, simuladores de rede e depuradores de rede. A partir da leitura deste capítulo, será possível obter o conhecimento necessário para o entendimento do escopo principal deste trabalho, bem como os resultados aqui apresentados.

2.1 Redes Definidas por Software

Uma Rede Definida por Software (SDN), caracteriza-se pela existência de um software de controle que tem a capacidade de controlar o encaminhamento de pacotes pela rede, com uma interface de programação bem definida. Ou seja, os elementos de comutação exportam uma interface de programação que permite ao software gerenciador, inspecionar, definir e alterar entradas na tabela de roteamento do comutador.

Redes Definidas por Software apresentam alguns princípios e benefícios, se comparados às redes tradicionais (Juniper, 2013), são eles:

- Separar, de forma clara, o software das redes em quatro camadas (planos): Gerenciamento, serviços, controle e encaminhamento.
- Centralizar os aspectos relevantes dos planos de gerenciamento, serviços e controle para simplificar o *design* da rede e reduzir os custos operacionais.
- Criar uma plataforma para aplicações de rede, serviços e integração com sistemas de gerenciamento, permitindo novas soluções de negócios.
- Padronizar protocolos propiciando a interoperabilidade e o suporte heterogêneo de fornecedores com mais opções e redução de custos.

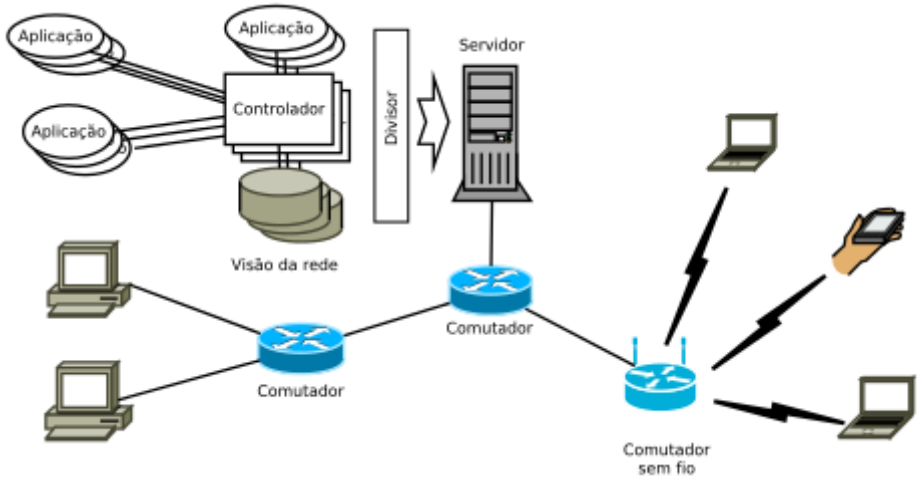

Figura 2.1: Estrutura geral de uma SDN

A Figura 2.1 esboça o modo que uma SDN é estruturada. O funcionamento de uma SDN segue um princípio simples, cada pacote recebido pelo *switch* gera uma consulta à tabela de encaminhamento do comutador. Caso não exista, na tabela de encaminhamento, uma ação definida para este pacote, o *switch* o encaminha para o controlador. Desta forma, o controlador poderá atribuir uma regra na tabela de encaminhamento do *switch*, para que o próximo pacote deste tipo, que passar pelo equipamento, não precise ser encaminhado novamente para o controlador (GUEDES et al., 2012). A seguir são apresentados conceitos relacionados a SDNs:

- Tabela de Fluxos: Cada entrada na tabela de fluxos no hardware de rede consiste em regra, ações e contadores (ROTHENBERG et al., 2010). A essas regras, é necessário atribuir ações para que o encaminhamento dos pacotes seja executado da forma correta. Os contadores são utilizados para fins estatísticos.

- Canal de comunicação seguro: Responsável por prover a proteção do canal de comunicação entre o *switch* e o controlador(ROTHENBERG et al., 2010).

- Protocolo OpenFlow: Protocolo que define as mensagens que serão trocadas entre os equipamentos da rede e os controladores(ROTHENBERG et al., 2010).

- Controlador: É o software responsável por tomar decisões, adicionar e remover as entradas na tabela de fluxos(ROTHENBERG et al., 2010).

Um pacote, ao chegar em um dispositivo com suporte ao protocolo OpenFlow, é analisado através do cabeçalho, que é comparado às regras da tabela de fluxos. Dessa forma as ações são devidamente realizadas. Caso o cabeçalho não corresponda a nenhuma regra da tabela, o pacote (por padrão) é enviado para ser tratado pelo controlador. Geralmente os pacotes que são encaminhados para o controlador são novos, ou seja,

correspondem ao primeiro pacote de um novo fluxo. De acordo com (ROTHENBERG et al., 2010) as ações associadas aos fluxos incluem:

- Modificar os campos do cabeçalho;
- Encaminhar o fluxo de pacotes para determinada porta (ou portas);
- Encapsular e transmitir o pacote para o controlador;
- Descartar os dados, como medida de segurança, com a implementação de *firewalls*, ou ainda para inibir ataques de negação de serviço;
- Encaminhar o pacote para o processamento normal do equipamento nas camadas 2 ou 3.

2.2 Controladores

De acordo com (MACAPUNA, 2012), um dos maiores problemas existentes hoje na Internet é o gerenciamento de redes, isto é causado em virtude do gerenciamento em baixo nível, configurado em cada ativo da rede, de forma individual. Na maioria das vezes, as configurações são realizadas de forma manual, tendo assim maior probabilidade de falhas, ocasionadas por erros de configuração dos componentes. Em virtude desses problemas, implementou-se um Sistema Operacional de Redes que oferece uma plataforma de desenvolvimento que permite a reutilização de aplicações de rede e possui a capacidade de análise e gerenciamento, permitindo assim que aplicações de rede sejam rapidamente desenvolvidas e aplicadas (GUDE et al., 2008).

Definida a interface de programação, seria possível desenvolver uma aplicação para fazer o controle de cada *switch* presente na rede, separadamente. Porém, existem algumas limitações, pois o programador teria de conhecer a fundo o hardware e fazer uma programação de baixo nível para que o software possa se comunicar com os dispositivos do comutador (GUDE et al., 2008).

Para que fossem resolvidos esses impasses, foi necessário o desenvolvimento de um novo nível na arquitetura, que oferecesse uma plataforma de programação de maior nível para o desenvolvedor. Esse nível é conhecido como controlador, ou sistema operacional de rede (GUEDES et al., 2012).

O controlador de rede, ou sistema operacional de rede, ou, ainda, *hypervisor* da rede (em alusão ao conceito derivado da área de sistemas virtualizados) pode, dessa forma, concentrar a comunicação com todos os elementos programáveis que compõem a rede e oferecer uma visão unificada do estado da rede (SHERWOOD et al., 2009).

Uma das virtudes das Redes Definidas por Software, é exatamente essa visão de controlador. Onde pode-se gerenciar um sistema inteiro através de um só centralizador, com visão das condições da rede, o que possibilita tomar decisões operacionais sobre como um sistema todo deve operar.

Um controlador pode ser implementado em um servidor comum da rede (ROTHENBERG et al., 2010).

A Figura 2.2 esboça o processo de instalação das regras, impostas pelo controlador nos dispositivos da rede. Um pacote, saindo do *host* A, é desconhecido pelo *switch* 1 que, por padrão, encaminha a decisão para o controlador. O controlador define as regras necessárias para aquele determinado tipo de pacote e retorna a regra aos dispositivos da rede. A partir deste processo, um pacote da mesma conexão, por exemplo, não precisará passar pelo controlador novamente.

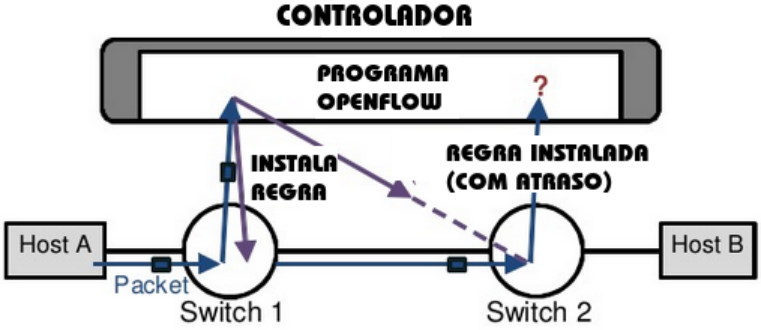

Figura 2.2: Instalação das regras na tabela de fluxos

A partir das informações disponibilizadas nesta seção, pôde-se compreender o funcionamento de um controlador SDN. Nas subseções seguintes, serão apresentados os tipos de controladores mais utilizados atualmente.

2.2.1 NOX

O NOX oferece uma interface de programação para desenvolvimento e/ou implementação de novas aplicações de rede, onde são utilizados dois diferentes conceitos que serão citados a seguir (FARIAS et al., 2011; NOX, 2013).

- Componente : É um conjunto de funcionalidades que são carregadas com o NOX. Pode-se citar, como exemplo, um *firewall*. Os componentes podem ser desenvolvidos em linguagem C e/ou Python
- Evento: Pode ser conceituado por ser uma ação realizada no fluxo da rede. As aplicações desenvolvidas em NOX utilizam um conjunto de manipuladores que são registrados para serem executados quando um evento ocorre.

NOX é o controlador original do OpenFlow, e tem como principal função o desenvolvimento de programas SDN na linguagem C++. O NOX funciona, assim como todos os controladores, sobre o conceito de fluxos de dados. O controlador gerencia a tabela de fluxos dos *switches* da rede reagindo a eventos de rede (GUEDES et al., 2012; NOX, 2013).

O NOX define uma série de eventos (GUEDES et al., 2012):

- ***packet in*** *(switch; port; packet)*: é acionado quando o *switch* envia um pacote recebido (por uma porta física) para o controlador.
- ***stats_in*** *(switch;xid; pattern; packets; bytes)*: é acionado quando o *switch* retorna os contadores de pacotes e bytes em resposta a um pedido pelas estatísticas associadas às regras contidas no padrão *pattern*. O parâmetro *xid* representa um identificador para o pedido.
- ***flow_removed*** *(switch; pattern; packets; bytes)*: acionado quando uma regra com padrão *pattern* ultrapassa o tempo limite estabelecido e é removido da tabela de fluxos do *switch*. Os parâmetros *packets* e *bytes* possuem os valores dos contadores para a regra.
- ***switch_join****(switch)*: acionado quando um *switch* se conecta à rede.
- ***switch_exit***(switch): acionado quando o switch sai da rede.
- ***port_change****(switch; port; up)*: acionado quando o dispositivo ligado ao switch é ligado ou desligado. O parâmetro *up* representa o novo estado do enlace.

O NOX também provê componentes para enviar mensagens aos *switches*:

- ***install*** *(switch; pattern; priority; time out; actions)*: insere uma regra com o dados padrão, prioridade, tempo limite e ações na tabela de fluxos do *switch*.
- ***uninstall****(switch; pattern)*: remove a regra contida em padrão da tabela de fluxos do *switch*.
- ***send*** *(switch; packet; action)*: envia o pacote para o *switch* e deixa que a ação seja tomada por ele.
- ***query_stats****(switch; pattern)*: gera uma requisição dos dados estatísticos contidos em uma regra no *switch* e retorna um identificador de requisição (*xid*) que pode ser usado para mapear uma resposta assíncrona do *switch*.

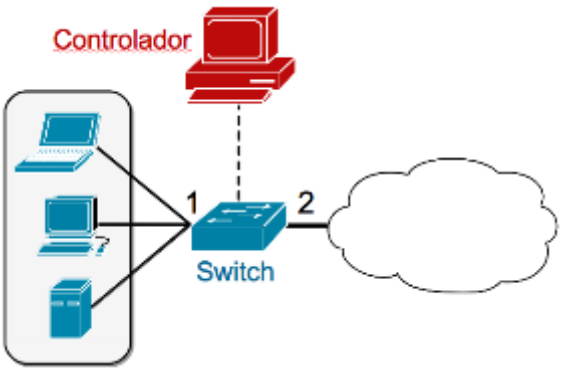

Figura 2.3: Topologia SDN simples

De acordo com (GUEDES et al., 2012), o programa em execução no controlador define um manipulador para cada um dos eventos construídos dentro do NOX, mas pode ser estruturado como um programa arbitrário. Por exemplo: Suponhamos que a rede tenha um *switch* ligado a um conjunto de *hosts* internos na porta 1 e uma rede (por exemplo: Internet) na porta 2, como esboçado pela Figura 2.3. Quando o *switch* é ligado, é feita uma requisição *switch_join*. Uma função instala as regras no *switch* para que os pacotes da porta 1 sejam enviados para a porta 2, e vice-versa. As regras (Figura 2.4 definem prioridade *Default* e tempo limite *None*.

```
def switch_join(switch):
   repeater(switch)

def repeater(switch):
   pat1 = {in_port:1}
   pat2 = {in_port:2}

install(switch,pat1,DEFAULT,None,[output(2)])
install(switch,pat2,DEFAULT,None,[output(1)])
```

Figura 2.4: Regras NOX

2.2.2 POX

O POX foi implementado tomando como base o controlador NOX. É uma plataforma para o desenvolvimento e prototipagem de software de rede utilizando a linguagem Python (POX, 2013).

O controlador POX surgiu com o objetivo de facilitar e contribuir em pesquisas e ensino sobre o protocolo OpenFlow. O POX disponibiliza uma interface mais simples e

uma pilha SDN mais organizada, o que facilita a execução de pesquisas e estudo(COSTA, 2013). O principal objetivo do POX é substituir o NOX em situações em que o desempenho não é algo para se preocupar (POX, 2013).

2.2.3 Trema

O Trema (Trema, 2013) é uma aplicação para o desenvolvimento de controladores OpenFlow. Este *framework* tem como característica o suporte as linguagens *Ruby* e C para que os pesquisadores desenvolvam e criem de maneira fácil seus próprios controladores OpenFlow. O Trema não tem como objetivo ser um controlador (especificamente), mas sim ajudar os desenvolvedores a criar implementações com *scripts* feitos em Ruby ou C.

O Trema também disponibiliza um emulador de rede OpenFlow próprio para a execução de testes, ou seja, o emulador simula toda a topologia da rede, não sendo necessário a utilização de *switches* ou *hosts* físicos para o teste do controlador desenvolvido.

2.3 Protocolo *OpenFlow*

De acordo com (COSTA, 2013), o OpenFlow é um protocolo promissor e inovador que proporciona um enorme leque de oportunidades de desenvolvimento tecnológico na área de redes de computadores. Empresas que são referências, quando o assunto é redes de computadores (HP, Cisco, Juniper), tem sido muito atraídas por essa tecnologia.

A tecnologia OpenFlow, inicialmente foi desenvolvida por pesquisadores da Universidade de Stanford e atualmente está sendo continuada pela *Open Networking Foundation (ONF)*. Em maio de 2012 a *Indiana University* em parceria com a ONF lançou o *SDN Interoperability Lab* que incentiva o desenvolvimento e a adoção de normas para as SDN com tecnologia OpenFlow (COSTA, 2013).

Com a aceitação das grandes empresas, o que define o rumo das tecnologias, está sendo possível ampliar o conceito de Redes Definidas por Software envolvendo-o a novos paradigmas de gerenciamento de redes integrado, desenvolvimento de protocolos e serviços baseados em redes virtualizadas ou SDNs (COSTA, 2013).

A pesquisa na área de arquiteturas de redes de computadores possui diversos desafios em relação à implementação e experimentação de novas propostas em ambientes reais. Isso ocorre devido à dificuldade que o pesquisador possui sobre obter uma rede de testes próxima de uma rede real. Para isso foi desenvolvido o protocolo OpenFlow (HANDIGOL et al., 2012), que propõe um mecanismo para permitir que redes reais se-

jam utilizadas como um ambiente de experimentos, sem interferir no tráfego a nível de produção (FARIAS et al., 2011).

O OpenFlow define um protocolo padrão para comunicação do controlador para com os dispositivos presentes na rede, como, por exemplo, comutadores, roteadores e *access points*(NASCIMENTO et al., 2011).

Esta proposta é fundamentada em comutadores *Ethernet* comerciais e define um protocolo padrão para controlar o estado destes comutadores. O OpenFlow também define um novo elemento de rede, o controlador, e o software que será executado nele(ROTHENBERG et al., 2010).

(COSTA, 2013) relata que o protocolo OpenFlow procura oferecer uma opção controlável e programável (SDN), sem atrapalhar o fluxo de produção. Nesse sentido o comutador OpenFlow encaminha os pacotes do tráfego experimental de acordo com regras definidas por um controlador centralizado. A arquitetura da plataforma OpenFlow pode ser vista da Figura 2.5.

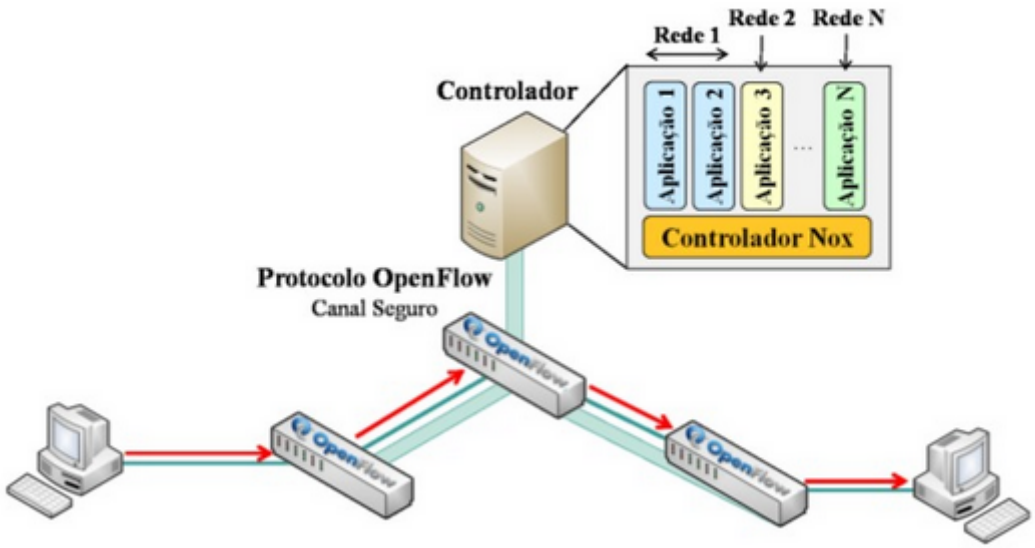

Figura 2.5: Protocolo OpenFlow com controlador NOX

De acordo com (ROTHENBERG et al., 2010), existem quatro razões fundamentais que contribuem para a aceitação da tecnologia OpenFlow:

- O OpenFlow pode ser incorporado em equipamentos de rede (roteadores, *switches*, pontos de acesso Wi-Fi) comerciais, sem modificação do hardware, mediante atualização do *firmware*, garantindo o desempenho de tecnologias consolidadas no encaminhamento de pacotes IP/*Ethernet*, como, por exemplo, ASICs, FPGAs, etc.;
- O protocolo OpenFlow separa o plano de controle do plano de dados, permitindo a utilização de controladores remotos baseados em servidores com sistemas operacionais e linguagens de programação comuns na indústria de tecnologia da infor-

mação. O software do controlador é responsável por definir o modo como os fluxos de pacotes são encaminhados e processados na rede. Dessa forma, o controle da rede deixa de estar embarcado nos equipamentos;

- Uma rede com suporte ao protocolo OpenFlow permite a definição de fatias de rede (*slices* ou *flow-spaces*), com garantia de isolamento entre os diferentes controladores que operam sobre a rede, permitindo que o tráfego operacional (conforme os protocolos tradicionais) e o tráfego experimental (conforme definido pelo usuário/operador da rede) operem em paralelo;
- O OpenFlow é compatível com a Internet atual, cujo tráfego pode continuar em operação em uma ou mais fatias da rede OpenFlow.

2.4 Virtualização da Rede

De acordo com (GUEDES et al., 2012) Redes Definidas por Software abriram espaço para que pesquisadores e desenvolvedores possam usar a rede como um ambiente de testes. Entretanto, ao se conectar os elementos de comutação de uma rede a um controlador único, a capacidade de se desenvolver novas aplicações e testá-las fica restrita ao responsável por aquele controlador. Mesmo que o acesso a este controlador seja compartilhado entre os pesquisadores, ainda há a questão da garantia de não-interferência entre as diversas aplicações sendo executadas na mesma interface.

Para se resolver esse problema, surgiu a possibilidade de dividir a rede em fatias (*slices*) e atribuir cada fatia a um controlador diferente. Ao considerar que um controlador SDN é um sistema operacional de rede, uma forma de implementar essas fatias é através da virtualização(GUEDES et al., 2012; GOMES et al.,).

O *FlowVisor* é um mecanismo desenvolvido especialmente para o *O*penFlow. Esta ferramenta funciona de forma transparente entre os dispositivos em uma rede OpenFlow e os controladores, como por exemplo, NOX e POX. O *Flowvisor* permite que mais de um controlador tenha funcionalidades sob a rede. A ferramenta utiliza o conceito de fatias (Figura 2.6), por exemplo, cada “pedaço” da rede é controlada por um controlador OpenFlow específico. As mensagens de controle trocadas entre os dispositivos e os controladores também passam pelo *Flowvisor*, que verifica para cada uma delas, quais políticas são aplicáveis. As políticas são definidas no próprio *Flowvisor* (SHERWOOD et al., 2009).

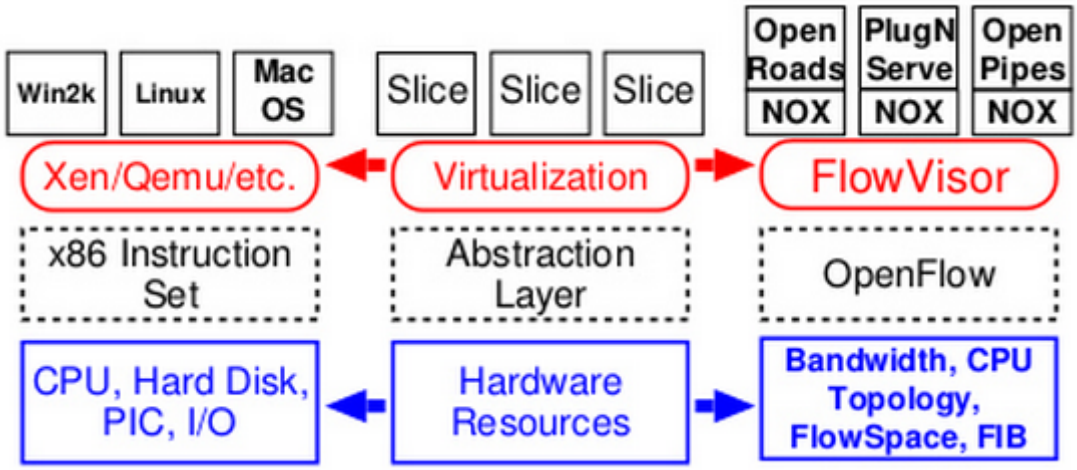

Figura 2.6: *Camada de virtualização da rede - Flowvisor*

Semelhante a virtualização do computador, *Flowvisor* é uma camada de virtualização de rede que reside entre o hardware e os componentes de arquitetura de software.

A maior vantagem das Redes Definidas por Software são as diversas formas de se dividir os recursos encontrados na rede(MCKEOWN et al., 2008). Como citado anteriormente, a capacidade que o OpenFlow possui de permitir a divisão dos comportamentos entre o ambiente de produção e de pesquisa é bastante útil para o avanço das arquiteturas hoje existentes.

A capacidade de dividir a rede em fatias já ocorre na arquitetura atual da Internet, um exemplo é a utilização de *VLANs*, em que existe um cabeçalho exclusivo no pacote para a definição de qual rede virtual ele pertence. Porém, o uso de *VLANs* possui limitações definidas pela tecnologia *Ethernet*, e por esse motivo torna-se complexo sua aplicação em contextos nos quais essas fatias devam se estender por mais de uma tecnologia de rede(COSTA, 2013).

De acordo com (COSTA, 2013) “Considerando essa analogia do uso de *VLANs* com o contexto atual das redes SDN é possível estender essa divisão de um forma em que os recursos da rede sejam virtualizados e apresentados de maneira isolada para cada desenvolvedor que deseje ter o seu próprio controlador de rede.”

Dessa forma, é permitido a integração do ambiente de produção com vários tipos de pesquisas, de forma paralela, na mesma rede física. A implementação de diferentes tipos de controladores também é suportada, permitindo que o desenvolvedor faça a análise do desempenho de cada controlador com sua respectiva linguagem de programação. Do mesmo modo que um sistema operacional é virtualizado, pode-se efetuar a virtualização de controladores sobre uma rede física.

2.5 Simuladores de rede

Para que uma arquitetura SDN seja criada, é necessário a implementação de controladores, a utilização do protocolo OpenFlow e hardwares compatíveis. Com a finalidade de reduzir os custos para fins de pesquisas e desenvolvimento, foi implementado um sistema simulador de rede chamado Mininet (COSTA, 2013).

O sistema permite aos pesquisadores fazer a simulação de uma rede com inúmeros dispositivos, com o uso de um único computador. O sistema é capaz de criar uma rede de máquinas virtuais, *switches*, controladores e *links*. Os *hosts* simulados no Mininet são executados com base em uma plataforma Linux. Os *switches* simulados são totalmente compatíveis com o protocolo OpenFlow(Mininet, 2013).

O Mininet traz consigo as melhores características de emuladores, hardware, bancos de ensaio e simuladores. Em comparação aos sistemas virtualizados completos, o Mininet se sobressai nos seguintes aspectos (Mininet, 2013):

- Boot mais rápido: Segundos ao invés de minutos;
- Escalas maiores: Capacidade para centenas de *hosts* e *switches*;
- Fornece mais largura de banda;
- Instalação relativamente fácil;
- Compatível com os controladores NOX e POX;

2.6 Depuradores de rede

Redes de Computadores são notoriamente difíceis de depurar. Os administradores de rede, atualmente, só dispõe de um conjunto rudimentar de ferramentas, como *ping* e *traceroute*, ferramentas de monitoramento como o *tcpdump* nos dispositivos finais e *netflow* em *switches* e roteadores (HANDIGOL et al., 2012).

Depurar uma rede se torna difícil por uma razão: essas ferramentas tentam reconstruir o estado complexo e distribuído da rede de uma forma *ad-hoc*, porém protocolos como o *Mac-learning* na camada L2 e de roteamento da camada L3 estão em constante mudança de estado (HANDIGOL et al., 2012).

Todavia, Redes Definidas por Software, como já descrito anteriormente, são controladas por meio de programação. O estado da rede é gerenciado por programas de controle logicamente centralizados, os estados são escritos diretamente nas tabelas de fluxos do *switch* (HANDIGOL et al., 2012).

Neste paradigma de Redes Definidas por Software, é possível depurar uma rede da mesma forma que se depura um software: escrever e executar programas, usar um depurador para visualizar contexto em torno dos erros ocorridos e traçar sequências de eventos que levam a origem desses erros (HANDIGOL et al., 2012).

A partir dos conceitos aqui descritos, são apresentados a seguir alguns depuradores para Redes Definidas por Software.

- O **TestOn** é uma ferramenta de código aberto, para automação de testes em ambientes que utilizam o protocolo OpenFlow. O principal objetivo da ferramenta é fornecer aos usuários uma interface simples para criar aplicações de teste automatizados de modo eficiente. Este depurador foi desenvolvido pela Paxterra e está em constante evolução e amadurecimento como ambiente de teste. Além disso, a ON.LAB *(http://onlab.us/testing.html)* é responsável por desenvolver e disponibilizar uma série de casos de teste para *download* (TestOn, 2013).

- O **NICE** é uma ferramenta extremamente eficiente para depurar programas OpenFlow, seu processo consiste em uma combinação de verificação de um modelo e uma execução simbólica para encontrar erros de programação ou *bugs* em programas OpenFlow. A automatização do teste de aplicações OpenFlow é um dos maiores objetivos desta ferramenta(CANINI et al., 2012).

- O **STS** é um depurador que busca eliminar a necessidade de procurar em *logs* os erros do software do controlador. O depurador simula dispositivos na rede, permitindo gerar casos de testes complexos. Esta ferramenta também permite examinar o estado da rede em determinado ponto e automaticamente procurar as entradas ou erros que são responsáveis por gerar determinados *bugs* na rede(SCOTT et al., 2013a).

- O **OFRewind** tem por objetivo principal ajudar os administradores de rede a encontrar os erros existentes em redes com OpenFlow. O OFRewind não é apenas um depurador de rede, através dele também é possível fazer análises da rede, como: verificar a utilização de CPU nos *switches*, *broadcast storms* (excesso de *broadcast*), encaminhamentos de pacotes anômalos, erros de análise de pacotes no controlador NOX e ainda outras ações inválidas do controlador. Ou seja, esta ferramenta não está limitada a localizar erros específicos em controladores OpenFlow, mas também pode ser utilizada para reproduzir outras anomalias na rede(WUNDSAM et al., 2011).

3 MATERIAIS E MÉTODOS

Na busca de atingir os objetivos propostos nesse trabalho, buscou-se conhecer o cenário ao qual ele se aplica. A investigação dos métodos a serem utilizados e a aquisição de conhecimentos sobre SDNs e depuradores deste tipo foi um fator crucial para o desenvolvimento deste estudo. O presente trabalho foi realizado para analisar as vantagens e desvantagens da utilização de cada um dos depuradores para SDN apresentados neste estudo.

A análise será feita com base na documentação disponibilizada pelos desenvolvedores de cada uma das ferramentas aqui apresentadas e, como fator adicional, através da instalação, breve utilização e execução de casos de teste inclusos nestas ferramentas.

3.1 NICE

A palavra “correto” em depuradores para Redes Definidas por Software nem sempre é algo invariável. A possibilidade de ocorrência de falsos positivos em testes, através de definições de casos de teste, muitas vezes é elevada. Pensando nisso, os desenvolvedores do NICE permitem que os programadores especifiquem propriedades de corretude, através de alterações feitas nos códigos, em linguagem Python. Para testes padrões, são definidas bibliotecas e propriedades comuns em todos os testes, por exemplo: teste de *loop* de rede e *blackholes*(CANINI et al., 2012).

No NICE define-se propriedades de corretude baseado em coisas do tipo (“algo ruim nunca acontece”) e *liveness* (eventualmente algo bom acontece). Verificar as propriedades de segurança é uma tarefa relativamente simples, embora a escrita de um predicado apropriado para todas as variáveis de estado possíveis demandem muito tempo. Um simples exemplo de um predicado pode ser citado como: Um código para testar se ocorrem *loops* em uma rede e/ou um *blackhole*. Já, a verificação para propriedades *liveness* é normalmente mais difícil, pela necessidade de considerar um sistema em execução possivelmente infinito. No NICE, são definidas entradas finitas, por exemplo, número de pacotes, número de possibilidades dos valores de cabeçalhos e etc. O que permite mais facilmente verificar algumas propriedades *liveness*(CANINI et al., 2012).

Para verificar algumas propriedades de segurança e *liveness*, o NICE permite usar algumas propriedades de corretude para acessar o estado do sistema, registrar retornos de comunicação, para observar transações importantes em um sistema em execução, e manter o estado local(CANINI et al., 2012).

Esta ferramenta fornece uma biblioteca padrão de propriedades de corretude apli-

cáveis a uma ampla variedade de aplicações OpenFlow. Alguns módulos de teste do Nice serão citados e explicados abaixo(CANINI et al., 2012).

- *NoForwardingLoops* : Esta propriedade afirma que os pacotes sofrerão *loops* de encaminhamento, este módulo é implementado através da verificação de que cada pacote passa por qualquer endereço <*switch, input port*> no máximo uma vez.
- *NoBlackHoles*: Esta propriedade determina que nenhum pacote deve ser descartado na rede, e é implementado, verificando se cada pacote que entra na rede tem destino para a própria rede ou é consumido pelo controlador. Para dar conta de ataques ou erros do tipo *flooding* (enchente de pacotes na rede), a propriedade impõe um saldo zero entre as cópias de pacotes e os pacotes consumidos.
- *DirectPaths*: Esta propriedade verifica que, uma vez que um pacote atingiu com sucesso o seu destino, os pacotes futuros do mesmo fluxo, não irão para o controlador. Esta propriedade é útil para muitas aplicações OpenFlow, embora não se aplique ao *MAC-Learning Switch*, pois requer que o controlador saiba como alcançar ambos os *hosts*, evitando que sejam construídos caminhos uni-direcionais entre eles.
- *StrictDirectPaths*: Esta propriedade verifica que, depois de dois *hosts* entregarem com êxito pelo menos um pacote de um fluxo em cada sentido, outros pacotes da mesma conexão não serão entregues novamente ao controlador. A propriedade verifica que o controlador estabeleceu um caminho direto e bi-direcional entre os dois *hosts*.
- *NoForgottenPackets*: Esta propriedade verifica se todos os *buffers* dos *switches* estão vazios no final da execução do sistema. Um programa pode facilmente violar essa propriedade “esquecendo de dizer” como o *switch* deve tratar um pacote. Isso pode eventualmente consumir todo o espaço de *buffer* disponível. Depois de um tempo limite (*timeout*), o *switch* pode descartar esses pacotes. Um programa com execução curta pode não ser executado por tempo suficiente para preencher a fila pacotes à espera do controlador *awaiting-controller-response*. A propriedade *NoForgottenPackets* permite detectar esses *bugs*.

3.1.1 Arquitetura

O NICE consiste em três diferentes "camadas": Verificador de Modelo *(Model Checker)*, Execução simbólica *Concolic-execution engine* e uma biblioteca de modelos incluindo um *switch* e vários tipos de *hosts*. A partir desses parâmetros é possível fazer a detecção dos *bugs*(CANINI et al., 2012).

- *Model checker* (Verificador de Modelo)- Para fazer *checkpoints* e restaurar o estado do sistema, o NICE salva a sequência de transações que criaram o estado atual e restaura o sistema, baseando-se nesta sequência. As verificações de estado são realizadas armazenando e comparando os estados já explorados. A principal vantagem desse processo é que reduz o consumo de memória e, em grande parte dos casos, é simples de implementação. Para criar as chaves *hash*, o NICE serializa os estados através de um módulo chamado cPickle(CANINI et al., 2012).

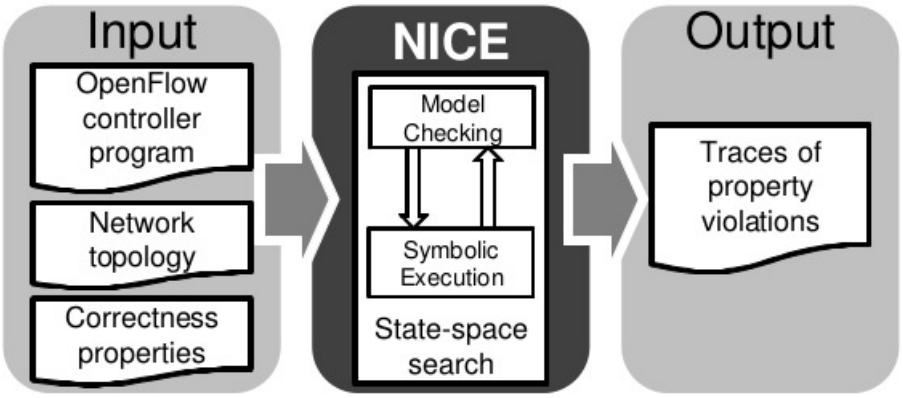

Figura 3.1: Camadas do NICE

Dado um programa OpenFlow, uma topologia de rede (definida pelo Anexo 1), e as propriedades de correção, o NICE realiza uma busca no espaço de estados e é capaz de identificar as propriedades errôneas ou violações (Figura 3.1).

3.1.2 Caso de teste

O caso de teste utilizado para validar o presente depurador foi o *Pyswitch*, que é uma aplicação OpenFlow que tem como objetivo implementar o *MAC learning* (registro do *MAC Address*), juntamente com o processo de *flooding* (enchente de pacotes) para destinos desconhecidos, muito comum em *Switches Ethernet*. Através da execução desta aplicação OpenFlow foi possível detectar três bugs. São eles(CANINI et al., 2012):

- **Host inacessível** (*blackhole*) - Esse é um *bug* muito relatado em testes de programas para SDN. Basicamente é quando um *host* B acaba se movendo de um local para outro. Antes de ocorrer essa movimentação, o *host* A inicia a comunicação com o *host* B, o que faz com que o controlador instale uma regra de encaminhamento para que os dispositivos da rede possam tratar esse tráfego. Quando o *host*

B se move de fato, a regra permanece da mesma forma, pois o tempo limite não expirou, e o *host* A continua enviando informações para B, que nunca vai receber esses pacotes.

- **Atraso no caminho direto** (*Delayed direct path*) - O Pyswitch também viola a propriedade *StrictDirectPaths*, ocasionando em uma significativa queda no desempenho do tráfego. A violação, ou *bug*, surge depois que um *host* "A"envia um pacote para o *host* "B", e "B"envia uma resposta para "A". Isso porque o Pyswitch instala uma regra de *forwarding* em uma direção (do remetente B para o destino A, linha 13 da Figura 3.2. O controlador não permite instalar uma regra de *forwarding* para a outra direção enquanto existir um pacote subsequente na comunicação de A para B. Em uma comunicação TCP, por exemplo, ao fazer o *Threeway Handshake*, esse *bug* ocasiona um tráfego até 50% maior. Após a correção deste erro, involuntariamente, ocorrerá outro. A correção anterior sugere que sejam instaladas regras nos dois sentidos, sendo que a segunda regra é reversa (para fazer o caminho de volta). Ao analisar o código, pode-se perceber que esta regra (A para B) é adicionada após a linha 14. No entanto, uma vez que as duas regras não são instaladas de uma só vez, a instalação desta regra, seguindo o código, pode permitir que o pacote do *host* B para o A chegue antes que a regra seja instalada. Esse *bug* fará com que este pacote seja redirecionado para ser tratado pelo controlador (desnecessariamente). A correção que melhor se adapta a esse problema é garantir que a regra seja instalada antes de permitir que o pacote do *host* B para A passe pelo *switch*. Efetuando essas correções, o programa irá satisfazer a propriedade *Strict-DirectPaths* e não retornará propriedades errôneas.

- **Excesso de tráfego** (Excess flooding) - Ao testar o *pyswitch* em uma topologia que consiste em um ciclo, o programa acusa uma violação na propriedade *NoForwardingLoops*. Isso ocorre em virtude do pyswitch não ter sido desenvolvido utilizando o protocolo STP *(Spaning Tree Protocol)*, que tem como principal função evitar a ocorrência de *loops* em redes com topologias semelhantes às testadas com o *pyswitch*.

```
1  ctrl_state = {} # State of the controller is a global variable (a hashtable)
2  def packet_in(sw_id, inport, pkt, bufid): # Handles packet arrivals
3    mactable = ctrl_state[sw_id]
4    is_bcast_src = pkt.src[0] & 1
5    is_bcast_dst = pkt.dst[0] & 1
6    if not is_bcast_src:
7      mactable[pkt.src] = inport
8    if (not is_bcast_dst) and (mactable.has_key(pkt.dst)):
9      outport = mactable[pkt.dst]
10     if outport != inport:
11       match = {DL_SRC: pkt.src, DL_DST: pkt.dst, ↩
               DL_TYPE: pkt.type, IN_PORT: inport}
12       actions = [OUTPUT, outport]
13       install_rule(sw_id, match, actions, soft_timer=5, ↩
               hard_timer=PERMANENT) # 2 lines optionally
14       send_packet_out(sw_id, pkt, bufid) # combined in 1 API
15       return
16   flood_packet(sw_id, pkt, bufid)
17 def switch_join(sw_id, stats): # Handles when a switch joins
18   if not ctrl_state.has_key(sw_id):
19     ctrl_state[sw_id] = {}
20 def switch_leave(sw_id): # Handles when a switch leaves
21   if ctrl_state.has_key(sw_id):
22     del ctrl_state[sw_id]
```

Figura 3.2: *Código-fonte Pyswitch*

3.2 TestOn

O TestOn foi projetado para usuários com diferentes níveis de habilidades em programação de scripts. Os casos de teste podem ser criados utilizando Python. O TestOn disponibiliza atualmente (TestOn, 2013):

- *Scripts* de fácil entendimento e manuseio: O TestOn disponibiliza uma interface abrangente (em desenvolvimento), algumas topologias não precisam de *scripts* e podem ser feitas utilizando o "plugin"*OF Automation Solution*, ou o usuário poderá fazer seu próprio *script* em Python. O TestOn disponibiliza uma interface gráfica do utilizador (GUI), ainda em desenvolvimento, para criar topologias de teste. Outra vantagem é a geração automática de elementos para unidades em teste, incluindo *switches* OpenFlow;

- Facilidade na execução de testes, a execução do teste é realizada em apenas um clique;

- Variedade de recursos de depuração;

- Capacidade de efetuar pausas no momento da execução;

- Depuração em pausa;

- Arquivos de *log*, relatórios e arquivos de sessão fáceis de pesquisar;

O TestOn tem como principal objetivo, interagir com os componentes do protocolo OpenFlow e automatizar a funcionalidade dos componentes. Esta ferramenta funciona de ponto a ponto para testar os componentes OpenFlow. A ferramenta também provê um *framework* automático que auxilia na depuração dos *scripts*.

3.2.1 Caso de teste em aplicações

Para um teste completo no TestOn, são necessários três diferentes componentes. São eles: Script de teste, Parâmetros a serem testados (casos de teste) e um arquivo com a topologia da rede a ser testada.

- Script de teste: O script de teste é responsável por fazer a rede funcionar "normalmente", com troca de pacotes de diferentes protocolos para ser testada posteriormente.

- Parâmetros a serem testados: É um *script* que contém todos os parâmetros da rede a serem testados. A Figura 3.3 mostra um *script* simples, que tem como finalidade executar testes em um acesso via SSH.

```
<PARAMS>
<testcases> 1</testcases>
<mail> paxweb@paxterrasolutions.com</mail>
<CASE1>

    <dl_type> 0x800 </dl_type>
    <nw_proto> 6 </nw_proto>
    <nw_src> 10.0.0.2 </nw_src>
    <tp_dst> 22 </tp_dst>
    <slice> SSH </slice>
    <permissions> 4 </permissions>

    <destination_host> 10.0.0.4 </destination_host>
    <destination_username> openflow </destination_username>
    <destination_password> openflow </destination_password>
    <destination_port> 22 </destination_port>
</CASE1>
</PARAMS>
```

Figura 3.3: *Script para teste de acesso SSH*

- Arquivo com a topologia de rede: Este arquivo funciona como base para execução de testes. Conforme a Figura 3.4, no *script* são configurados o número de dispositivos que estarão disponíveis na topologia e suas devidas ligações. Neste caso, incluiu-se um *FlowVisor*, um *host* emulado pelo Mininet e um controlador POX.

```
<TOPOLOGY>
<COMPONENT>
    <FlowVisor1>
        <host>192.168.56.101</host>
        <user> openflow </user>
        <fvadmin_pwd></fvadmin_pwd>
        <password>openflow</password>
        <type>FlowVisorDriver</type>
        <COMPONENTS>
        </COMPONENTS>

    </FlowVisor1>

    <Mininet1>
        <host>192.168.56.101</host>
        <user> openflow</user>
        <password> openflow</password>
        <type>MininetCliDriver</type>
        <COMPONENTS>
            # Specify the Option for mininet
            <topo> single</topo>
            <topocount>3</topocount>
            <switch> ovsk </switch>
            <controller> remote </controller>
         </COMPONENTS>
    </Mininet1>

    <POX2>
        <host> 192.168.56.102 </host>
        <user>  openflow </user>
        <password> openflow </password>
        <type> PoxCliDriver </type>
        <test_target> 1 </test_target>
        <COMPONENTS>
            <pox_lib_location> /home/openflow/pox/  </pox_lib_location>
            <samples.of_tutorial></samples.of_tutorial>
        </COMPONENTS>
    </POX2>
```

Figura 3.4: *Script para definição de topologia de teste - TestOn*

3.3 OFRewind

OFRewind é uma ferramenta que utiliza as propriedades de divisão na arquitetura de encaminhamento para permitir a praticidade na depuração de domínios de rede. Esta

divisão é realizada para permitir a gravação e reprodução de estados da rede. Em testes, a ferramenta revelou-se útil em diversos estudos de caso(WUNDSAM et al., 2011).

O OFRewind, por sua vez, não é muito intuitivo. A meta dos desenvolvedores foi fazer um software que ajude a encontrar a raiz do problema, porém, não faz isso automaticamente. O administrador da rede é responsável por informar detalhadamente o cenário que será depurado(WUNDSAM et al., 2011).

A ferramenta trabalha de modo passivo, onde existem duas situações: *recording* e *replay*. O *recording* tem como objetivo gravar o tráfego crítico da rede, por exemplo, mensagens de roteamento. As definições do que deve ser monitorado devem ser especificadas nas configurações. O *replay*, por sua vez, tem como objetivo reproduzir o tráfego gravado, de forma coordenada, para que assim, o administrador da rede possa identificar os problemas ali ocorridos(WUNDSAM et al., 2011).

3.3.1 Arquitetura

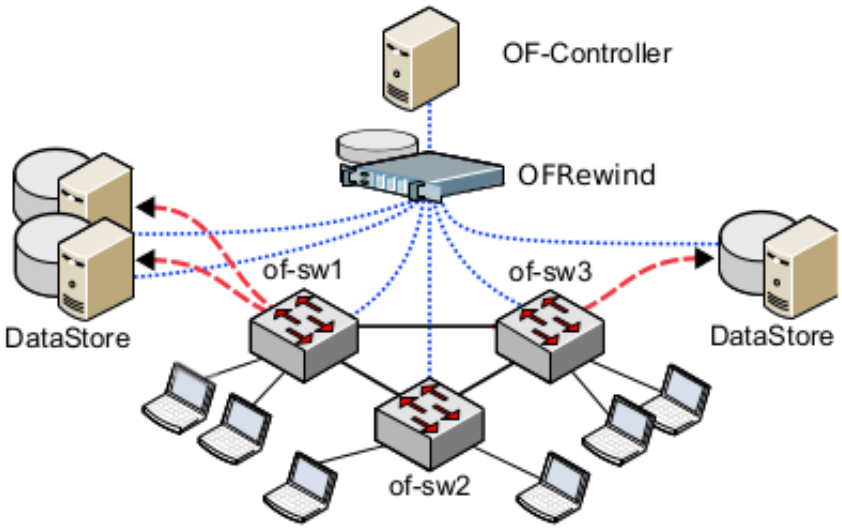

Figura 3.5: Ilustração básica do funcionamento do OFRewind

A partir da Figura 3.5, pode-se compreender como o OFRewind atua. Ele funciona como uma espécie de *proxy* para controle na camada de rede, entre o *switch* principal(*switch* que contém o OFRewind da Figura 3.5) e o controlador. Isso possibilita interceptar e modificar as mensagens do plano de rede para fazer o *recording* e *replay*. A partir deste ponto, as mensagens do plano de rede são gravadas nos *DataStore* instalados juntamente com o *switch* (WUNDSAM et al., 2011).

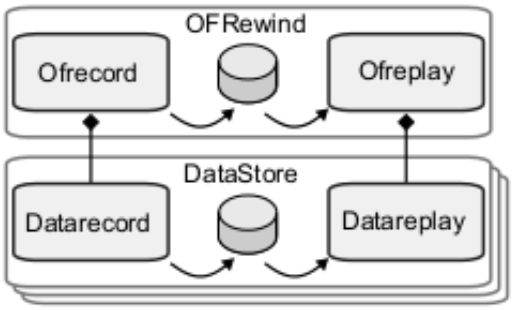

Figura 3.6: Subdivisão dos componentes do OFRewind

Os componentes podem ser subdivididos em dois módulos cada um, de acordo com a Figura 3.6. Eles consistem em um módulo para *recording*, e um módulo para *replay*, com um local comum de armazenamento, chamados *Ofrecord* e Ofreplay, com respectivos *Datarecord* e *Datareplay* (WUNDSAM et al., 2011).

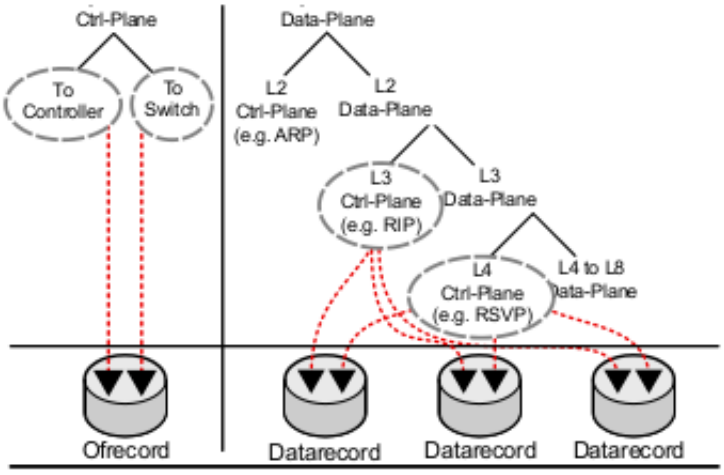

Figura 3.7: Relação entre as camadas do OFRewind

A Figura 3.7, representa a relação entre as camadas e o modo como as informações são armazenadas. Pode-se perceber que cada informação é armazenada em um *Datarecord* específico(WUNDSAM et al., 2011).

3.3.1.1 Gravação do Tráfego com *Ofrecord*

Como já discutido, o *Ofrecord* é um parâmetro do OFRewind para gravar todo e qualquer tráfego desejado em uma rede com *OpenFlow*. Ele atua de forma seletiva, ou seja, seleciona o que realmente é importante para gravar, pois gravar tudo ocuparia muito espaço(WUNDSAM et al., 2011). A seguir são apresentados fatores utilizados para reduzir o armazenamento de pacotes em excesso:

- **Seleção**: O fluxo pode ser classificado e selecionado conforme as necessidades do administrador de rede. Pode-se referir a seleção de tráfego sempre que seja necessário tomar uma decisão sobre um conjunto de tráfego para gravação. Entre as categorias de seleção estão inclusas: mensagens de protocolos de roteamento, mensagens de controle de tráfego entre outras. A Figura 3.7 pode ser novamente referenciada para o melhor entendimento do processo de seleção e gravação dos dados, onde são gravados, por exemplo, tráfego com protocolo RIP (protocolo de roteamento).

- **Amostragem**: Se o resultado da seleção não foi capaz de reduzir o tráfego suficientemente, pode-se aplicar pacotes ou fluxos de amostragem (filtros) de cada tipo de tráfego, como uma estratégia de redução.

- **Cut-offs**: Grava somente os primeiros *bytes* do pacote, que contêm informações relevantes.

3.3.1.2 Modos de operação do *Ofreplay*

Para dar suporte no teste de diferentes *switches*, controladores, *hosts* e também para oferecer suporte a diversos cenários, o *Ofreplay* suporta diversos modos de operação (WUNDSAM et al., 2011), tais como:

- **ctrl**: Nesse modo de operação, o *replay* é feito diretamente para o controlador. O *Ofreplay* reproduz os dados, de forma organizada, gravados no *storage* local. Esse modo permite depurar as aplicações do controlador em um único host, sem precisar de *switches*, vários *hosts* e etc. Para efetuar a depuração, não é necessário ter gravado outras mensagens (*Dataplane* Figura 3.7), apenas as mensagens de controle.

- **switch**: Este modo de operação reproduz as mensagens de controle para os *switches*. Neste modo não são necessários controladores.

- **datahdr**: Este modo usa os cabeçalhos dos pacotes capturados pelo Datarecord para regerar exatamente os mesmos os fluxos. O *payload* dos pacotes é criado com dados fictícios.

- **datafull**: Nesse modo, o tráfego de informações salvas pelos *DataStores* são reproduzidos com o *payload* do pacote completo, permitindo a inclusão, de modo selecionado, de tráfego para os terminais (*end-hosts*) nos testes.

O *Ofreplay* também permite o usuário refinar ainda mais o tráfego registrado para coincidir com o cenário de repetição. As mensagens reproduzidas podem ser sub-selecionadas com base no *host* de origem ou destino, porta e tipo da mensagem(WUNDSAM et al., 2011).

3.3.2 Caso de teste

O caso de teste utilizado para validação desta ferramenta será O *Anomalous Forwarding*, baseado na documentação disponibilizada pelos desenvolvedores da ferramenta. Para a elaboração dos testes, foram utilizados alguns *switches*, cada um com 2 *hosts* conectados, conforme ilustrado na Figura 4.2(WUNDSAM et al., 2011).

3.4 STS

Assim como todos os depuradores SDN, o STS tem como principal foco a solução de problemas em programas para controladores SDN.

O STS é o primeiro depurador a fazer, de forma programada, o isolamento das entradas que induzem a uma falha em um sistema distribuído. As técnicas de gravação e execução (*record* e *replay*), assim como o OFRewind, permitem que, de forma manual, o administrador percorra a execução original e verifique se um conjunto de entradas provocou um erro. O problema é que, na maioria das vezes, o arquivo gerado pelo funcionamento original é tão grande que o conjunto de entradas se torna confusa aos olhos do administrador de rede(STS, 2013).

O principal objetivo do STS é fornecer informações sobre o que exatamente fez a rede ter uma configuração inválida. Esta ferramenta foi aplicada e testada em várias plataformas de controladores, incluindo NOX e POX(SCOTT et al., 2013b).

3.4.1 Arquitetura

A principal abordagem do STS é simular o plano de controle da rede em uma única máquina. Após esta etapa, executar o software de controle junto com o simulador da rede e, após isso, conectar os *switches* aos controladores (simulados), como se fossem verdadeiros dispositivos de rede. Esta configuração permite que o simulador possa interpor no canal de comunicação e analisar o atraso, o descarte de pacotes ou reordenar as mensagens se necessário. A visão geral da arquitetura pode ser vista na Figura 3.8 (SCOTT

et al., 2013b).

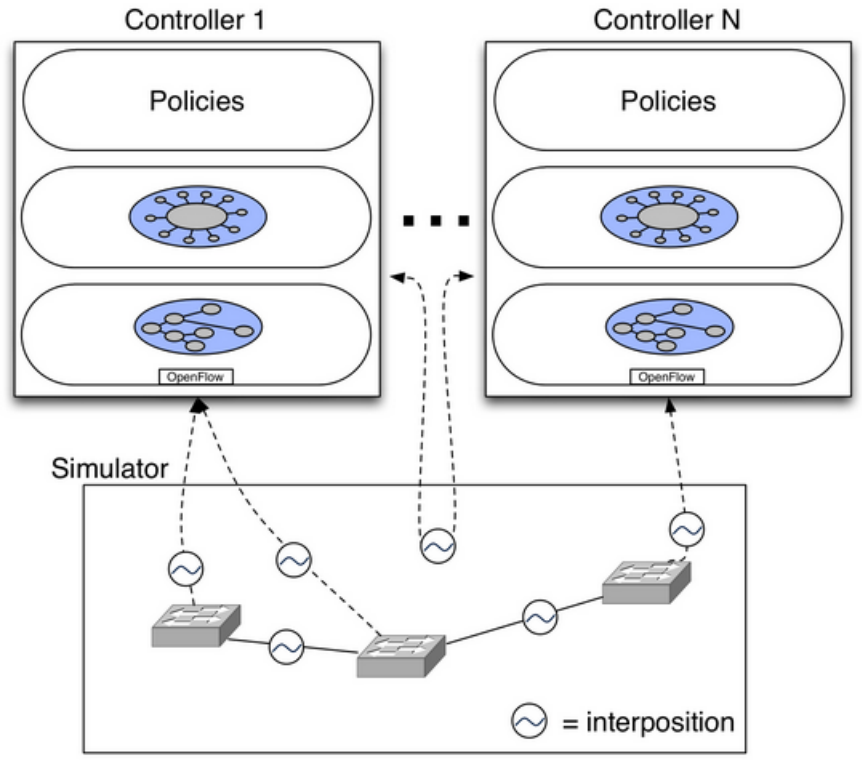

Figura 3.8: Arquitetura do STS

Infelizmente, o ato de transformar o *log* original (que reproduz a violação) para uma subsequência de eventos externos (que pode ou não se reproduzir como uma violação), não é trivial(SCOTT et al., 2013b).

Para obter resultados, é necessário processar a relação entre o que acontece no momento e o que aconteceu anteriormente, após isso é necessário verificar o que essa relação reflete no *log* original, e por último, fazer as inferências sobre as dependências restantes na subsequência(SCOTT et al., 2013b).

- **Equivalência Funcional**: O principal problema na hora de efetuar os testes necessários é manter a equivalência, a situação é que os eventos internos podem, e quase sempre, são diferentes sintaticamente. Por exemplo, os números da sequência de um pacote podem ser todos diferentes, quando repetido uma subsequência do *log* original. Observa-se que muitos eventos internos são funcionalmente equivalentes, no sentido em que eles têm o mesmo efeito sobre o estado do sistema, no que respeita a provocar a violação invariante (apesar das diferenças sintáticas). Por exemplo, as mensagens de modificação do fluxo podem fazer com que os *switches* alterem seu comportamento de encaminhamento, mesmo que o identificador da mensagem seja diferente.

- **Tratando Eventos Internos Ausentes**: Alguns eventos do *log* original podem "acontecer antes do teste"(*happen before*), ou seja, algum evento externo pode estar ausente quando repassado uma subsequência do *log*. O STS, por sua vez, tratou esse problema. A ferramenta espera até que aconteça um evento equivalente e usa para substituir o evento ausente. Para que não ocorram *dead-locks* foi definido um *time-out* ϵ . Caso o evento não ocorra, ele repete o registro desde o início para que haja este evento ausente, desta vez sabendo exatamente quais eventos internos no intervalo de entrada vão e não vão ocorrer antes de injetar a próxima entrada. O algoritmo utilizado está referenciado na Figura 3.9.

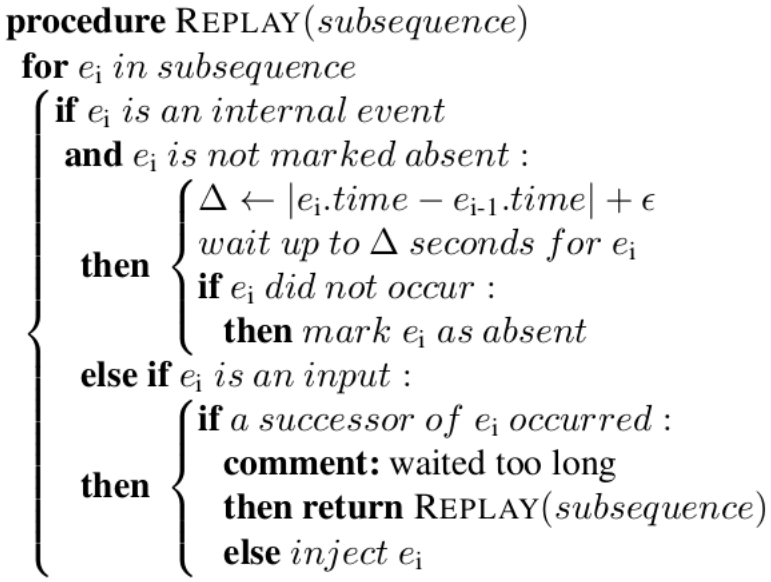

Figura 3.9: *Replay* no STS

- **Tratando Novos Eventos Internos**: Esta propriedade trata eventos internos que não haviam acontecido nos *logs* capturados. É utilizado um algoritmo para saber onde adicionar algum novo evento que não existia na captura inicial.

3.4.2 Caso de teste

O caso de teste utilizado para validar o presente depurador foi o *POX In-Flight Blackhole*, disponível na documentação dos desenvolvedores. Este caso de teste tem como objetivo encontrar *blackholes* nos softwares dos controladores (SCOTT et al., 2013b).

A partir da execução do caso de teste *POX In-Flight Blackhole*, foi possível encontrar alguns *blackholes*, ou *hosts* desconhecidos em sua execução. Após 20 execuções com entradas geradas aleatoriamente, foi possível detectar um *blackhole* persistente enquanto o controlador POX fazia a descoberta de links e locais em uma pequena topologia

com 2 *switches* e 2 *hosts*. No rastreamento inicial houveram 29 entradas, o STS retornou 11 possíveis entradas que poderiam causar este problema (SCOTT et al., 2013b).

4 RESULTADOS E DISCUSSÕES

Afim de atender os objetivos propostos neste trabalho, analisando os diferentes depuradores existentes para SDNs, bem como suas respectivas características, torna-se necessário, para o melhor entendimento, estabelecer critérios de comparação entre essas ferramentas. Os critérios definidos são: instalação, usabilidade, documentação disponível, número de casos de teste disponíveis e eficiência.

O estudo desenvolvido visa auxiliar profissionais e pesquisadores da área de Redes Definidas por Software a realizar seus projetos. Com base nos dados expostos aqui, os leitores deste trabalho terão toda a base necessária para a escolha do melhor depurador SDN para suas necessidades.

4.1 NICE

De acordo com os critérios definidos, os resultados do estudo e teste desta ferramenta são:

- **Instalação**: Um tutorial de instalação do NICE é encontrado na página dos desenvolvedores *(https://code.google.com/p/nice-of/wiki/DocRequirements)*, o que facilita a instalação. São necessários alguns *plugins* adicionais, como o Python com versão 2.6 ou superior.

- **Usabilidade**: Apesar do NICE não disponibilizar uma interface gráfica, a sua usabilidade é bastante satisfatória. Com apenas poucos comandos é possível fazer o teste da aplicação desejada. Por exemplo, para executar o caso de teste pyswitch, só é preciso executar o nice com a chamada para este arquivo através deste comando: *"./nice.py config/pyswitch.conf"*.

- **Quantidade de casos de teste**: A ferramenta disponibiliza apenas três casos de teste (não impedindo que o usuário crie seus próprios casos de teste), os principais exemplos a serem citados são pyswitch e o loadbalancer (balanceamento de carga).

- **Documentação disponível**: Os desenvolvedores do NICE disponibilizam um material completo sobre a ferramenta, o que facilita muito em pesquisas e desenvolvimento da ferramenta de forma organizada. São disponibilizados três artigos, em fóruns internacionais, sobre esta ferramenta. Também é disponibilizado no site *https://code.google.com/p/nice-of/* , um manual completo da utilização deste depurador e também uma Wiki *(https://code.google.com/p/nice-of/w/list)*.

- **Violações Encontradas**: A partir da execução do caso de teste padrão do NICE (pyswitch), foi possível comprovar a existência dos três *bugs* citados na documentação do NICE. O Resultado pode ser visto na Figura 4.1.

```
--- Results ---
Total states: 3102
Unique states: 1520
Revisited states: 1582
Maximum path length: 34
Invariant violations: 35
NoLoop              : 0 violations
internal check      : 0 violations
NoDrop              : 2 violations (first found after 13.17s, 1114 transitions)
StrictDirectRoute   : 28 violations (first found after 13.55s, 1155 transitions)
NoForgottenPackets  : 5 violations (first found after 13.17s, 1114 transitions)
ReturnContinueStop  : 0 violations
:: Symbolic engine quitting...
```

Figura 4.1: *Resultados execução Pyswitch.conf*

4.2 TestOn

- **Instalação**: A instalação do TestOn é feita através de alguns simples comandos e do *download* do código-fonte da ferramenta. É necessário ter os pacotes *python* instalados com versões acima de 2.6.

- **Usabilidade**: O TestOn, apesar de ainda não disponibilizar uma interface gráfica (em desenvolvimento), possui um *cli* para rodar as aplicações, o que facilita muito na hora de efetuar os testes. Na prática é um dos depuradores mais fáceis de utilizar.

- **Quantidade de casos de teste**: Os desenvolvedores não fornecem nenhum caso de teste que contenham *bugs*. Apenas o caso chamado "MininetTest", no qual a aplicação monta a topologia, chama um *script* de teste e aplica sobre esta tecnologia. Não foram encontrados *bugs* na execução deste caso de teste.

- **Documentação disponível**: A ferramenta disponibiliza apenas um *wiki*, pode ser acessado no *link* (*https://github.com/Paxterra/TestON*), com tutoriais de instalação e alguns exemplos para executar. Não há nenhum artigo acadêmico disponibilizado por seus desenvolvedores.

- **Violações Encontradas**: A partir do caso de teste executado, percebeu-se a rápida execução de todos os parâmetros, não sendo encontrado (conforme o esperado), algum erro nos *scripts*.

4.3 STS

De acordo com os critérios definidos, os resultados do estudo e teste desta ferramenta são:

- **Instalação**: A instalação do STS requer apenas a linguagem *Python* como *plugin* (A página dos desenvolvedores disponibiliza um tutorial de instalação). Após isto, só é preciso efetuar o download através do *git* no site dos desenvolvedores *(https://github.com/ucb-sts/sts)*.
- **Usabilidade**: O STS não possui uma interface gráfica, o que não complica em nada a relação com o usuário. Ele possui um *cli* para executar todos os comandos e depurar os arquivos desejados.
- **Quantidade de casos de teste**: A ferramenta disponibiliza alguns casos de teste que deixam a desejar. O que não impede do próprio administrador de rede desenvolver seus próprios casos de teste.
- **Documentação disponível**: Os desenvolvedores do STS disponibilizam dois artigos, na versão completa e outro na versão reduzidas. Também existe uma wiki no site *http://ucb-sts.github.io/sts/*. O material para desenvolvimento de novos casos de teste do STS é um pouco confuso, porém é disponibilizado.
- **Violações Encontradas**: A partir da execução do caso de teste padrão do NICE (pyswitch), foi possível comprovar a existência dos três *bugs* citados na documentação do NICE.

A equipe de desenvolvedores do STS, disponibilizou os *logs* gerados para que a falha encontrada no caso de teste seja corrigida no POX. Ao combinar a saída do console com o código, foi descoberto que os fatores que causavam esta falha eram dois *in-flight packets* (acionados por prioridade de tráfego)(SCOTT et al., 2013b).

4.4 OFRewind

- **Instalação**: Não se aplica.
- **Usabilidade**: O OFRewind, apesar de ser uma ferramenta *Open-source*, tem algumas limitações. Não foi possível obter o código-fonte para compilar a ferramenta pois não obteu-se resposta ao contato com o desenvolvedor.

- **Quantidade de casos de teste**: A documentação da ferramenta traz consigo três diferentes casos de teste, o que proporciona ao leitor o conhecimento das funcionalidades desta aplicação.

- **Documentação disponível**: Apesar de todos os outros impasses obtidos nesta ferramenta, esta possui uma documentação completa e bem detalhada. O que, a partir da obtenção do código-fonte, possibilita o melhor desenvolvimento de casos de teste para serem abordados por esta ferramenta.

- **Violações Encontradas**: A partir da documentação disponível, obteu-se a informação de alguns casos de teste já executados e devidamente documentados.

Para investigar o desempenho de dispositivos a partir de um novo fabricante, ilustrado pelo fabricante C (*Vendor C*), foi necessário registrar o tráfego de um fluxo específico. Para teste foi enviado um *ping* com 10 segundos de *delay* entre um par de *hosts* ligados ao *switch* fabricante B (*Vendor C*, of-sw3 na Figura 4.2). Em seguida foi utilizado o recurso de mapeamento de dispositivos e portas do *Ofreplay* para reproduzir o tráfego do *host* c7 para o c8 pela porta 8 e 42 pertencente ao *switch* fabricante C na Figura 4.2.

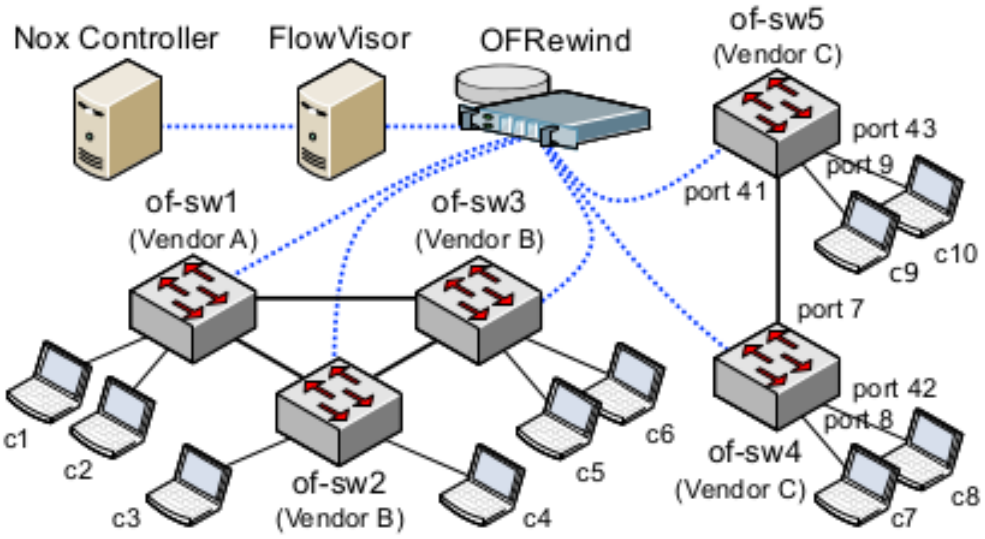

Figura 4.2: *Topologia para testes do OFRewind*

Após a repetição, foi possível observar uma limitação interessante no *switch* fabricante C. O fluxo do ping na fase de resolução do ARP. Os pacotes ARP transmitidos do *host* c7 foram recebidos pelo *host* c10, mas não pelo *host* c8 nem pelo c9. A entrada da tabela de fluxos criado no *switch* of-sw4 na Figura 4.2, é mostrada na Figura 4.3. Conclui-se que a ação de *Flooding*, ou seja, inundação de pacotes, não está devidamente aplicada pelo *switch* fabricante C da Figura 4.2.

CONTADOR	AÇÃO
duration=181s	in_port=8
n_packets=0	dl_type=arp
n_bytes=3968	dl_src=00:15:17:d1:fa:92
idle_timeout=60	dl_dst=ff:ff:ff:ff:ff:ff
hard_timeout=0	actions=FLOOD

Figura 4.3: *Entrada da tabela de fluxos switch of-sw4*

O teste também mostra que o *Flooding* de pacotes recebidos pelas portas baixas (até a porta 24) que devem ser enviados para *hosts* de portas altas (após a porta 24) não funcionam corretamente. Efetivamente, o *Flooding* de pacotes ARP é restrito apenas dentro de um "grupo"de 24 portas dentro do *switch* (inferior ou superior), considerando um *switch* com 48 portas. Este fato foi confirmado pela depuração, validando assim a ferramenta OFRewind.

4.5 Comparações

As comparações aqui apresentadas foram feitas com base na utilização das ferramentas (com algumas exceções) e com base na documentação disponível por seus desenvolvedores. Para que o leitor obtenha compreenda de melhor forma os resultados aqui apresentados, as comparações foram organizadas em diferentes tabelas.

A Tabela 4.1 mostra quais são os controladores (aqui apresentados) compatíveis com cada depurador (* Precisa de *plugin*).

Tabela 4.1: Funcionalidades dos depuradores estudados

Depurador	NOX	POX	Trema	Teste de *Switches*
NICE	Sim	Sim*	Não	Não
STS	Sim	Sim	Não	Não
TestOn	Sim	Sim	Não	Não
OFRewind	Sim	Não	Não	Sim

A Tabela 4.2 mostra as características dos depuradores obtidas através das pesquisas aqui realizadas.

Com base na pesquisa realizada, a Tabela 4.3 mostra as características dos controladores com suas respectivas plataformas suportadas e linguagens de desenvolvimento.

Tabela 4.2: Critérios de análise dos Depuradores

Depurador	Eficiência	Instalação	Usabilidade	Documentação
NICE	Ótimo	Bom	Ótimo	Ótimo
STS	Bom	Bom	Regular	Ótimo
TestOn	Regular	Bom	Bom	Ruim
OFRewind	Bom	Ruim	Não se aplica	Ótimo

Tabela 4.3: Comparação entre controladores SDN

Controlador	Linguagem	Plataforma	Características
NOX	C++	Linux	Principal controlador para OpenFlow
POX	Python	Windows, Mac e Linux	Versão para desenvolvimento em Python
Trema	C e Ruby	Linux	Possui emulador para *scripts*

5 CONSIDERAÇÕES FINAIS

A partir do estudo realizado, foi possível observar que o paradigma Redes Definidas por Software teve uma grande aceitação em ambientes acadêmicos e corporativos. O desenvolvimento do protocolo OpenFlow tem muita importância para a correta expansão das Redes de computadores. A proposta oferecida pelo protocolo OpenFlow permite inúmeros avanços em pesquisas envolvendo redes de computadores, a capacidade de dividir o plano de dados e o plano de controle, é a maior vantagem do uso de Redes Definidas por Software. A partir deste recurso é possível fazer o teste de ambientes e configurações de redes sem que a rede, em nível de produção, fique instável ou indisponível.

O presente trabalho buscou apresentar uma criteriosa revisão bibliográfica dos aspectos envolvidos no desenvolvimento e configuração em Redes Definidas por Software. O estudo foi realizado com ênfase nos controladores de rede e nos depuradores com propostas mais relevantes. O trabalho mostrou que o controlador tem fundamental importância em um ambiente SDN, nele estão contidas todas as configurações e comportamentos que a rede deve seguir para cada tipo de pacote recebido. Para garantir que o programa que define as configurações do controlador estivesse funcionando da forma mais correta possível, foram, então, apresentados alguns controladores existentes e bastante utilizados, e também depuradores para SDN, tais como o NICE, TestOn, STS e OFRewind. A partir da revisão bibliográfica feita neste trabalho e baseado na documentação disponível de cada ferramenta apresentada, foi possível obter o conhecimento sobre qual o depurador utilizar para determinadas situações, tais como facilidade de uso e eficiência.

Este trabalho serve como base para profissionais e pesquisadores que desejam ampliar seus conhecimentos sobre depuradores para Redes Definidas por Software, tanto para âmbito acadêmico quanto corporativo.

6 TRABALHOS FUTUROS

Como sugestão para o enriquecimento desta pesquisa, pode-se incluir o desenvolvimento de um caso de teste específico, com topologia e propriedades de corretude para testar igualmente a eficiência dos depuradores existentes para SDNs. Utilizando como base este trabalho, é possível incrementá-lo com sugestões de melhorias para cada depurador e relatar possíveis *bugs* nos projetos aqui apresentados.

7 ANEXOS

```
class PyswitchModel(Model):
    def initTopology(self, topo):
        # instantiate the controller application, passing additional options
        self.controller = PySwitchController(name="ctrl", \
                    ctxt=self.of_context, version="pyswitch")
        # give the controller to the OF context
        self.of_context.setController(self.controller)
        # create two switches with 2 ports each
        # of_id is the openflow ID (the dp_id)
        sw1 = OpenflowSwitch(name="s1", port_count=2, of_id=1)
        sw2 = OpenflowSwitch(name="s2", port_count=2, of_id=2)
        mac1 = (0x00, 0x01, 0x02, 0x03, 0x04, 0x00)
        mac2 = (0x00, 0x01, 0x02, 0x03, 0x05, 0x05)
        # instatiate an Arrival client with mac1 that sends pkts ethernet frames to
        # mac2, sequentially (or not)
        cl1 = Arrival(name="h1", mymac=mac1, dstmac=mac2, \
                    pkts=self.config.get("pyswitch_model.pkts"), \
                    sequential=self.config.get("pyswitch_model.sequential"))
        # cl2 is a replier (ping server) that generates a reply whenever a packet
        # is received
        cl2 = Replier(name="h2", mymac=mac2)
        # Topology for sw1
        # port 0 is connected to port 0 on cl1
        # port 1 is connected to port 0 on sw2
        sw1.initTopology({0: (cl1, 0), 1: (sw2, 0)})
        # Topology for sw2
        # port 0 is connected to port 1 on sw1
        # port 1 is connected to port 0 on cl2
        sw2.initTopology({0: (sw1, 1), 1: (cl2, 0)})
        cl1.initTopology({0: (sw1, 0)})
        cl2.initTopology({0: (sw2, 1)})
        # Connect the switches to the controller
        sw1.setController(self.controller)
        sw2.setController(self.controller)
        # Add the clients
        self.clients.append(cl1)
        self.clients.append(cl2)
        # Add the switches
        self.switches.append(sw1)
        self.switches.append(sw2)
        self.switches_idx[sw1.getOpenflowID()] = sw1
        self.switches_idx[sw2.getOpenflowID()] = sw2
        # invariants (see the Invariants section)
        self.invariants = [NoLoopInvariant(), NoDropInvariant(), \
                    DirectRouteInvariant(), StrictDirectRouteInvariant()]
        # callbacks to call whenever a new path exploration starts
        self.controller.start_callbacks.append(lambda: self.controller.install())
        self.controller.start_callbacks.append(lambda: self.controller.addSwitch(sw1))
        self.controller.start_callbacks.append(lambda: self.controller.addSwitch(sw2))
```

Figura 1: *Script para definição de topologia de teste do Pyswitch- NICE*

REFERÊNCIAS BIBLIOGRÁFICAS

AL-SHAER, E.; AL-HAJ, S. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In: ACM. **Proceedings of the 3rd ACM workshop on Assurable and usable security configuration**. [S.l.], 2010. p. 37–44.

CANINI, M.; VENZANO, D.; PERESINI, P.; KOSTIC, D.; REXFORD, J. A nice way to test openflow applications. **NSDI, Apr**, 2012.

COSTA, L. R. Openflow e o paradigma de redes definidas por software. **CEP**, v. 70910, p. 900, 2013.

FARIAS, F. N.; JÚNIOR, J. M. D.; SALVATTI, J. J.; SILVA, S.; ABELÉM, A. J.; SALVADOR, M. R.; STANTON, M. A. Pesquisa experimental para a internet do futuro: Uma proposta utilizando virtualização eo frame-work openflow. **Minicursos**, v. 1, p. 1–61, 2011.

GOMES, V. S.; ISHIMORI, A.; PERES, I. M. A.; FARIAS, F. N.; CERQUEIRA, E. C.; ABELÉM, A. J. Flowvisorqos: aperfeiçoando o flowvisor para aprovisionamento de recursos em redes virtuais definidas por software.

GUDE, N.; KOPONEN, T.; PETTIT, J.; PFAFF, B.; CASADO, M.; MCKEOWN, N.; SHENKER, S. Nox: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 3, p. 105–110, 2008.

GUEDES, D.; VIEIRA, L.; VIEIRA, M.; RODRIGUES, H.; NUNES, R. V. Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores. **Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2012**, v. 30, n. 4, p. 160–210, 2012.

HANDIGOL, N.; HELLER, B.; JEYAKUMAR, V.; MAZIÈRES, D.; MCKEOWN, N. Where is the debugger for my software-defined network? In: ACM. **Proceedings of the first workshop on Hot topics in software defined networks**. [S.l.], 2012. p. 55–60.

Juniper. **Decodificando a SDN**. 2013. Acesso em 9 set. 2013. Disponível em: <http://www.juniper.net/br/pt/dm/sdn-wp/>.

KUZNIAR, M.; CANINI, M.; KOSTIC, D. Often testing openflow networks. In: IEEE. **Software Defined Networking (EWSDN), 2012 European Workshop on**. [S.l.], 2012. p. 54–60.

KUZNIAR, M.; PERESINI, P.; CANINI, M.; VENZANO, D.; KOSTIC, D. A soft way for openflow switch interoperability testing. In: ACM. **Proceedings of the 8th international conference on Emerging networking experiments and technologies**. [S.l.], 2012. p. 265–276.

MACAPUNA, C. A. B. Openflow e nox: Propostas para experimentaçao de novas tecnologias de rede. 2012.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008.

MEGA, G.; KON, F. God: Um depurador simbólico para sistemas de objetos distribuídos. **12º Salão de Ferramentas do Simpósio Brasileiro de Engenharia de Software**, 2013.

Mininet. **Sobre o Mininet**. 2013. Acesso em 08 out. 2013. Disponível em: <http://mininet.org/>.

NASCIMENTO, M. R.; ROTHENBERG, C. E.; DENICOL, R. R.; SALVADOR, M. R.; MAGALHAES, M. F. Routeflow: Roteamento commodity sobre redes programáveis. **XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos-SBRC**, 2011.

NOX. **Sobre o NOX**. 2013. Acesso em 2 nov. 2013. Disponível em: <http://www.noxrepo.org/nox/about-nox/>.

POX. **Sobre o POX**. 2013. Acesso em 22 set. 2013. Disponível em: <http://www.noxrepo.org/pox/about-pox/>.

ROTHENBERG, C. E.; NASCIMENTO, M. R.; SALVADOR, M. R.; MAGALHÃES, M. F. Openflow e redes definidas por software: um novo paradigma de controle e inovação em redes de pacotes. **Cad. CPqD Tecnologia**, v. 7, n. 1, p. 1–6, 2010.

SCOTT, C.; WUNDSAM, A.; WHITLOCK, S.; OR, A.; HUANG, E.; ZARIFIS, K.; SHENKER, S. Automatic troubleshooting for sdn control software. 2013.

____. **How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software**. [S.l.], 2013.

SHERWOOD, R.; GIBB, G.; YAP, K.-K.; APPENZELLER, G.; CASADO, M.; MCKEOWN, N.; PARULKAR, G. Flowvisor: A network virtualization layer. **OpenFlow Switch Consortium, Tech. Rep**, 2009.

STS. **Sobre o STS**. 2013. Acesso em 3 dez. 2013. Disponível em: <http://ucb-sts.github.io/sts/>.

TestOn. **A ferramenta TestOn**. 2013. Acesso em 12 out. 2013. Disponível em: <http://onlab.us/testing.html>.

Trema. **Trema - Full-Stack OpenFlow Framework in Ruby and C**. 2013. Acesso em 21 dez. 2013. Disponível em: <http://trema.github.io/trema/>.

WUNDSAM, A.; LEVIN, D.; SEETHARAMAN, S.; FELDMANN, A. Ofrewind: enabling record and replay troubleshooting for networks. In: **USENIX ATC**. [S.l.: s.n.], 2011.