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

OpenFlow e o Paradigma de Redes Definidas por Software

Lucas Rodrigues Costa

Brasília
2013

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

OpenFlow e o Paradigma de Redes Definidas por Software

Lucas Rodrigues Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. André Costa Drummond

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Dr. André Costa Drummond (Orientador) — CIC/UnB
Prof. Dr. Jacir Luiz Bordim — CIC/UnB
Prof. Dr. Wilson Henrique Veneziano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Costa, **Lucas Rodrigues**.

OpenFlow e o Paradigma de Redes Definidas por Software / **Lucas Rodrigues Costa**. Brasília : UnB, 2013.
313 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. Redes Definidas por Software (RDS), 2. OpenFlow, 3. POX, 4. MiniNet

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

OpenFlow e o Paradigma de Redes Definidas por Software

Lucas Rodrigues Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. André Costa Drummond (Orientador)
CIC/UnB

Prof. Dr. Jacir Luiz Bordim
CIC/UnB

Prof. Dr. Wilson Henrique Veneziano
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 8 de março de 2013

Dedicatória

Dedico a todos os professores que passaram na minha vida dos quais pude tirar os conhecimentos necessários que contribuíram para este e outros trabalhos em minha vida. Dedico a minha família e meus amigos que me ajudaram com apoio moral nessa longa jornada.

Agradecimentos

Agradeço primeiramente a Deus por ter me ajudado nessa caminhada, a minha família pelo apoio e ajuda, a minha noiva Adriana, pela paciência e compreensão nos momentos difíceis e principalmente a meus professores da universidade, em especial ao professor André Costa Drummond que me apresentou à área de redes de computadores, a qual me identifiquei muito.

Resumo

As redes de computadores se tornaram parte da infraestrutura crítica de nossa sociedade, participando do cotidiano de bilhões de pessoas. O sucesso das redes de computadores se deve, em grande medida, a simplicidade de seu núcleo. Na arquitetura atual, a inteligência da rede está localizada nos sistemas de borda, enquanto o núcleo é simples e transparente. Embora essa abordagem tenha tido sucesso, viabilizando a Internet, também é a razão para sua inflexibilidade e incapacidade de se atender as necessidades das novas aplicações que deverão surgir no futuro próximo.

A inflexibilidade da arquitetura das redes de computadores também traz um desafio para os pesquisadores da área, pois seus experimentos dificilmente podem ser avaliados em redes reais. Sendo assim, em geral, testes de novas tecnologia são realizadas em simuladores de rede, o que implica em uma simplificação da realidade. O paradigma de Redes Definidas por Software (*Software Defined Networks* - SDN) e a arquitetura OpenFlow oferecem um caminho para a implementação de uma arquitetura de rede programável, capaz de ser implementada de forma gradativa em redes de produção, que oferece a possibilidade de separação dos mecanismos de controle dos diversos fluxos de tráfego atendidos, de forma que, por exemplo, um experimento científico possa ser executado em uma rede real (adaptada para o SDN) sem interferir em seu funcionamento.

O presente trabalho contextualiza os problemas existente nas redes de computadores atuais, e apresenta o paradigma de redes SDN como uma das principais propostas para a viabilização da Internet do Futuro. Nesse contexto, o trabalho discute a arquitetura OpenFlow que permite a criação de aplicações para redes SDN. O trabalho também apresenta o simulador de redes SDN MiniNet, que implementa a interface OpenFlow. Finalmente são implementados exemplos de uso da arquitetura OpenFlow no MiniNet com o intuito de preparar um conjunto de cenários que sirva como base para a realização de pesquisas na área de redes SDN, ou como ferramenta didática para o ensino de conceitos complexos em redes de computadores.

Palavras-chave: Redes Definidas por Software (RDS), OpenFlow, POX, MiniNet

Abstract

The computer networks have become an important element of the critical infrastructure of our society. The success of these networks is mainly due to the simplicity of its core. In the current architecture the network intelligence is located at the edge devices while the core remains simple and transparent. Although this approach has been successful, making the Internet a reality, it is also responsible for the inflexibility and the inability to cope with the needs of the novel network applications that will arise in the near future.

The inflexibility of today networks brings a challenge for the researchers in the field, because it makes the evaluation of scientific experiments almost impossible to carry on in real networks. Thus, frequently, new technologies are test in simulation environments, which might be an over simplification of the reality. The Software Defined Networks (SDN) approach and the OpenFlow architecture provide means for the implementation of a programmatic network, which allows its gradual deployment in production networks, and offer ways to separate different control mechanisms for different network flows which enables, among other things, a scientific experiment to be done in a real network (SDN capable one) without interfering with its operation.

This paper the main issues of today's networks and presents the SDN approach as one of the main proposals to enable the Future Internet. It discusses the OpenFlow architecture as a way to implement SDN applications. The MiniNet simulator is also presented as a tool to evaluate SDN scenarios by implementing the OpenFlow Interface. Finaly, some examples are implemented with the MiniNet in order to compile a set of network scenarios, which might serve as basis for new research in the SDN field, or as a powerful didactic tool for teaching complex network concepts.

Keywords: Software Defined Network (SDN), OpenFlow, POX, MiniNet

Sumário

1 Introdução **1**
1.1 Motivação 1
1.2 Objetivos 2
1.3 Metodologia 2
1.4 Organização do Projeto 3

2 Revisão Teórica **4**
2.1 História das Redes de Computadores 4
2.2 Arquiteturas Atuais 7
2.3 Evolução das Redes de Computadores 11
2.4 Arquiteturas do Futuro 14
2.5 Redes Virtualizadas 16
2.6 Redes Definidas por *Software* 18
2.7 Sistema OpenFlow 19

3 Redes Definidas por Software **23**
3.1 Sugimento das Redes Definidas por Software 24
3.2 Motivaçao 25
3.3 Arquitetura 25
3.4 Elementos Programáveis das redes SDN 26
3.5 Divisores de Recursos 28
3.6 Controlador SDN 31
3.6.1 NOX 32
3.6.2 POX 33
3.6.3 Maestro 33
3.6.4 Beacon 34
3.6.5 Floodlight 34
3.6.6 Frenetic 34
3.6.7 Onix 35
3.6.8 SNAC 35
3.6.9 Trema 36
3.7 Aplicações 36
3.7.1 Controle de Acesso 36
3.7.2 Gerenciamento de Redes 37
3.7.3 Gerenciamento de Energia 37
3.7.4 Comutador Virtual Distribuído 38

3.7.5 Redes Domésticas . . . 38
3.7.6 Roteador Expansível de Alta Capacidade . . . 39
3.7.7 Redes de Grande Porte . . . 40

4 OpenFlow **41**
4.1 Pradrão OpenFlow . . . 42
4.2 Componentes de uma Rede OpenFlow . . . 43
4.3 Protocolo OpenFlow . . . 44
4.4 Controlador OpenFlow . . . 46
4.5 Funcionamento . . . 46
4.6 Aplicações do OpenFlow . . . 48
4.7 O OpenFlow Atualmente . . . 49

5 Controlador POX **51**
5.1 Capacidades do POX . . . 51
5.2 Instalação do POX . . . 52
5.3 Executando o POX . . . 52
5.4 Componentes do POX . . . 53
5.5 Compreendendo a Função launch() . . . 54
5.6 Núcleo do POX . . . 55
5.7 Eventos no POX . . . 56
5.8 Pacotes do POX . . . 56
5.9 Threads no POX . . . 58
5.10 OpenFlow no POX . . . 59
5.11 Mensagens OpenFlow no POX . . . 61
5.11.1 of.ofp_packet_out . . . 61
5.11.2 of.ofp_flow_mod . . . 61
5.11.3 of.ofp_match . . . 62
5.11.4 ofp_action . . . 62
5.12 Desenvolvendo Componentes Próprios . . . 62
5.12.1 Como criar um componente básico no POX . . . 63

6 MiniNet **65**
6.1 Funcionamento . . . 66
6.2 MiniNet na Prática . . . 67
6.2.1 Pré-Requisitos . . . 67
6.2.2 Download VM MiniNet . . . 67
6.2.3 Configuração da VM MiniNet . . . 67
6.2.4 Notas sobre o Prompt de Comando . . . 68
6.2.5 Acessando a VM MiniNet via SSH . . . 69
6.2.6 Instalando os Editores de Texto . . . 70
6.2.7 Aprendendo as Ferramentas de Desenvolvimento . . . 70
6.2.8 Principais Comandos do MiniNet . . . 71
6.2.9 Comandos Utilizados para a Inicialização do CLI do MiniNet . . . 75
6.3 Outros Comandos e Ferramentas da VM MiniNet . . . 79
6.4 Teste o Simulador . . . 85

7 Implementações Desenvolvidas **86**
7.1 Instalando e Configurando o POX 86
7.2 Executando um Componente do POX no MiniNet 87
7.3 Cenários de Simulação 88
7.3.1 Componente HUB 89
7.3.2 Componente SWITCH 91
7.3.3 Componente SWITCHES 92
7.3.4 Componente FIREWALL 92
7.3.5 Componente ROUTER 94
7.3.6 Componente ESPECÍFICO 94
7.4 Resultados Esperados 95

8 Discussão e Conclusão **97**
8.1 Desafios de Pesquisa 98
8.2 Considerações Finais 99
8.3 Trabalhos Futuros 100

Referências **101**

A Descrição dos Elementos do POX **105**

B Códigos das Implementações Desenvolvidas **116**
B.1 Código do Terceiro Cenário 116
B.2 Código do HUB 117
B.3 Código do SWITCH 119
B.4 Código do SWITCHES 121
B.5 Código do FIREWALL 122
B.6 Código do ROUTER 125
B.7 Código do ESPECÍFICO 130

C API MiniNet (MiniGUI) **136**

Lista de Figuras

2.1 Diferenças entre as topologias do sistema telefônico e da ARPANET. . . 5
2.2 Pilha TCP/IP. 8
2.3 Protocolo TCP/IP [31]. 9
2.4 Representação dos tipos de arquitetura da Internet do Futuro [17]. . . 15
2.5 Exemplo de uma rede virtualizada, com três redes compartilhando o mesmo substrato físico [33]. 17
2.6 Arquitetura do comutador OpenFlow [42]. 21

3.1 Arquiteturas de roteadores: modelo atual e modelo programável [40]. 26
3.2 Identificação dos fluxos OpenFlow pelo FlowVisor [20]. 29
3.3 Fluxo de comandos do FlowVisor [14]. 30
3.4 Principais controladores SDN. 32

4.1 Componentes de uma rede OpenFlow [33]. 44
4.2 Definição de um fluxo na arquitetura OpenFlow [19]. 45
4.3 Exemplo de uma tabela de fluxos de um comutador OpenFlow. O campo representado por um ”*” indica que qualquer valor é aceito naquela posição, ou seja é um campo que não importa no reconhecimento do fluxo [19]. 48

5.1 Comparações de desempenho entre o POX e o NOX [34]. 52

6.1 MiniNet API miniedit.py. 83
6.2 MiniNet API consoles.py. 83

7.1 Primeiro Cenário de Testes. 89
7.2 Segundo Cenário de Testes. 89
7.3 Terceiro Cenário de Testes. 90

C.1 MiniGUI API minigui.py (API de inicialização do CLI do MiniNet). 137

Lista de Quadros

5.1 Opções de inicialização do POX. 52
5.2 Atributos gerais dos Eventos OpenFlow no POX. 60

6.1 Tipos de mensagens OpenFlow apresentadas ao iniciar o controlador. . 80
6.2 Tipos de mensagens OpenFlow apresentadas ao realizar um ping. . . . 81

A.1 Descrição dos Componentes nativos do POX. 105
A.2 Continuação do Quadro A.1. 106
A.3 Continuação do Quadro A.2. 107
A.4 Análise da classe Ethernet do POX. 108
A.5 Análise da classe IP do POX. 108
A.6 Análise da classe TCP do POX. 109
A.7 Argumentos de construção da classe Timer. 109
A.8 Continuação do Quadro A.7. 110
A.9 Métodos da classe Timer. 110
A.10 Eventos fornecidos do módulo OpenFlow no POX. 110
A.11 Continuação do Quadro A.10. 111
A.12 Definição dos atributos da classe ”of.ofp_packet_out” do POX. 112
A.13 Continuação do Quadro A.12. 112
A.14 Definição dos atributos da classe ”of.ofp_flow_mod” do POX. 112
A.15 Continuação do Quadro A.14. 114
A.16 Definição dos atributos da classe ”of.ofp_match” do POX. 114
A.17 Definição de algumas das classes ”ofp_action” do POX. 115

Capítulo 1

Introdução

A infraestrutura mundial das redes de computadores constitui hoje um dos serviços críticos da sociedade. Esta infraestrutura, chamada de Internet, provê uma série de serviços que atendem a todos os setores da sociedade, sendo considerada um sucesso. Todavia, as mesmas razões que permitiram o nascimento e o crescimento da Internet, são vistas hoje como barreiras para seu desenvolvimento e, portanto, ameaçam sua capacidade de suprir as necessidades futuras da sociedade.

Essas barreiras levaram a comunidade científica a pensar em novas propostas para atender essas necessidades. Tais propostas contribuem para a definição da Internet do Futuro. Embora tais propostas sejam capazes de atender as necessidades, sua implementação é difícil de ser realizada, pois, como as redes de computadores são parte da infraestrutura crítica da sociedade, mudanças tornam-se obstáculos quase intransponíveis, e acabam sendo descartadas pelos administradores das redes.

Essa inflexibilidade na arquitetura da Internet também traz um desafio para os pesquisadores da área, pois seus experimentos acabam não sendo avaliados em redes reais. Dessa forma, em geral, testes de novas tecnologias são realizados em ambientes virtuais que simplificam a realidade não fornecendo, assim, o nível de fidelidade necessário para uma implementação em redes reais.

O paradigma de Redes Definidas por Software (*Software Defined Networks* - SDN) e o protocolo OpenFlow oferecem um caminho para vencer esse desafio, por meio de uma solução seja implantada de forma gradativa em redes de produção.

1.1 Motivação

O sucesso das redes de computadores evidenciou o trabalho de pesquisadores da área, mas sua chance de gerar impacto tornou-se cada vez mais remota. Isto ocorre devido a dificuldade para a implantação de novos protocolos e tecnologias, visto que novas propostas, em geral, requerem mudanças em todos os milhares de equipamentos de rede instalados.

As redes de computadores são parte da infraestrutura crítica do nosso dia-a-dia, e portanto mudanças tornam-se obstáculos e acabam sendo descartadas pelos admi-

nistradores de rede. Todas essas barreiras dificultam a implantação de novas ideias que acabam não sendo testadas em ambientes reais.

O paradigma de Redes Definidas por Software e o protocolo OpenFlow foram desenvolvido com esse propósito, viabilizar o teste de novas propostas em redes reais sem afetar as redes de produção. Dessa forma, além de atender os problemas da inflexibilidade das redes de computadores, o paradigma SDN nos permite atender as necessidades das novas aplicações de rede que deverão surgir em um futuro próximo, resolvendo problemas associados a requisitos de escalabilidade, gerenciamento, mobilidade e segurança por meio de uma arquitetura de rede programável.

1.2 Objetivos

Este trabalho tem como objetivos gerais:

- Apresentar o paradigma de redes SDN identificando suas características e capacidades;
- Apresentar o protocolo OpenFlow e suas características;
- Apresentar os detalhes da arquitetura do controlador de redes SDN POX e os elementos principais para a implementação de aplicações no mesmo;
- Por fim, apresentar o simulador de redes SDN MiniNet e todas suas características.

Este trabalho tem como objetivos específicos:

- Mostrar as possibilidades da arquitetura de rede SDN;
- Mostrar como utilizar um controlador de rede SDN, como o POX, para a implementação de diferentes soluções na área de redes;
- Implementar componentes no POX que sirvam como ponto de partida para a sua utilização prática.

1.3 Metodologia

O presente trabalho faz uma revisão bibliográfica dos principais conceitos das redes de computadores, apresenta seu contexto histórico, sua evolução, as tecnologias atuais e do futuro, além das implicações deste contexto nos dias de hoje. É apresentado o estado da arte, em redes de computadores, para resolver as imperfeições da arquitetura atual. Dentre elas encaixa-se como a mais promissora as redes SDN e o uso do protocolo OpenFlow em comutadores de produção. São apresentadas as características deste paradigma de redes, as características do protocolo OpenFlow e todos os elementos que compõe o modelo de arquitetura deste paradigma.

São propostas implementações de referência que servem como base para o desenvolvimento de aplicações na área de Redes Definidas por Software. As implementações propostas são validadas por meio de simulação e sua corretude é atestada.

1.4 Organização do Projeto

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 traz o contexto histórico das redes de computadores, apresentando suas limitações e as inflexibilidades de sua arquitetura atual. O Capítulo 3 aborda as Redes Definidas por Software, apresentando suas características. O Capítulo 4 apresenta uma ferramenta de virtualização de redes, o protocolo OpenFlow. No Capitulo 5 é apresentado o controlador POX, um controlador de Redes Definidas por Software especialmente desenvolvido para o ambiente de pesquisa e ensino. No Capitulo 6 é apresentado o simulador de redes SDN MiniNet, suas características, seu funcionamento além de ser posto em prática com um exemplo simples e de fácil implementação. O Capítulo 7 apresenta cenários de teste que servem como ponto de partida para as primeiras aplicações de redes SDN. O Capítulo 8 apresenta os resultados do trabalho, levantando desafios de pesquisa para o paradigma SDN e ressaltando a importância da arquitetura de Redes Definidas por Software nas redes de computadores do futuro.

Os Apêndices estão organizados da seguinte forma. O Apêndice A descreve alguns elementos do controlador POX. O Apêndice B contém os códigos das implementações propostas neste trabalho. Por fim, o Apêndice C contém o código de uma API desenvolvida neste trabalho para o simulador MiniNet.

Capítulo 2

Revisão Teórica

As redes de computadores se tornaram parte da infraestrutura crítica de nossas empresas, casas e escolas, sendo um sucesso do cotidiano de bilhões de pessoas. Desde sua origem, as redes de computadores tem crescido bastante e seu uso ficou cada vez mais diversificado. A massificação das tecnologias de rede foi obtida, em grande parte, pela sua boa adoção sistemática no meio comercial, ao mesmo tempo em que programas multiusuários começaram a ser desenvolvidos. Nesse contexto as redes de computadores cresceram agregando milhões de equipamentos.

O sucesso das redes de computadores deve, em grande medida, a simplicidade de seu núcleo. Na arquitetura atual, a inteligencia da rede está localizada nos sistemas de borda, enquanto o núcleo é simples e transparente. Embora essa simplicidade tenha tido sucesso, viabilizando a Internet, também é a razão para seu engessamento. Todas essas limitações apresentam problemas estruturais que são difíceis de serem resolvidos, tais como escalabilidade, mobilidades e gerenciamento de serviço [9].

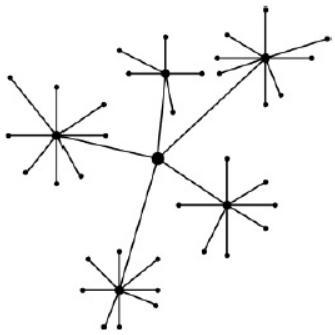
Este capítulo apresenta o contexto histórico das redes de computadores, sua evolução durante os anos, as tecnologias atuais e do futuro, além das implicações deste contexto nos dias de hoje. É apresentado o estado da arte, em redes de computadores, para resolver as implicações da arquitetura atual. Dentre elas encaixa-se como a mais promissora o conceito de Redes Definidas por Software e o uso do protocolo OpenFlow.

2.1 História das Redes de Computadores

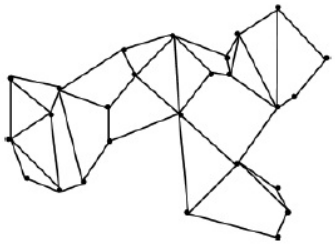
As Redes de Computadores surgiram a partir de pesquisas militares no período da Guerra Fria na década de 1960 quando dois blocos ideológicos e politicamente antagônicos exerciam enorme poder sobre o mundo. O líder de um desses blocos, os Estados Unidos, temia um ataque do outro bloco, liderado pela União Soviética, a uma de suas bases militares. Um ataque poderia revelar todas as informações sigilosas dos Estados Unidos. Temendo tal ataque, o Departamento de Defesa dos Estados Unidos desenvolveu uma rede de comunicação que tinha o objetivo de descentralizar as informações até então concentrada em pontos específicos.

Como exemplo, pode-se citar as redes telefônicas, que eram organizadas seguindo uma topologia em estrela que continha pontos centrais (Figura 2.1a), dessa maneira, se algum desses pontos sofresse um ataque soviético a rede poderia se desestruturar, causando um grande prejuízo e impacto na guerra.

Para descentralizar as informações foi criado uma novo modelo de troca e compartilhamento das informações entre as bases militares, constituindo a primeira rede de computadores, a ARPANET (*Advanced Research Projects Agency Network* [39]). A ARPANET foi projetada e constituída para ser uma rede tolerante a falhas e altamente distribuída. Sua estrutura era organizada seguindo uma topologia em malha (Figura 2.1b) que utilizava comutação por pacotes entre seus nós.



(a) Exemplo de Topologia Estrela



(b) Exemplo de Topologia Malha

Figura 2.1: Diferenças entre as topologias do sistema telefônico e da ARPANET.

A ARPANET era formada por minicomputadores chamados IMP (*Interface Message Processors*) conectados por linhas de transmissão de 56 kbps. Para garantir a confiabilidade, cada IMP era conectado a pelo menos dois outros IMPs criando assim caminhos alternativos em caso de falhas. Caso acontecesse uma falha a mensagem seria encaminhada automaticamente para rotas alternativas, problema que não era possível resolver utilizando uma topologia estrela 2.1a. Assim, cada nó era constituído por um IMP e um *host* conectados por um fio curto, no qual o *host* enviaria uma mensagem de 8063 bits para seu IMP que, por sua vez, dividiria essa mensagem em pacotes de 1008 bits e os encaminhava de forma independente até o seu destino [44].

Depois que a tensão entre URSS e EUA diminuiu, a ARPANET cresceu, pois os EUA permitiram que pesquisadores desenvolvessem estudos para a ARPANET trazendo mais usuários a rede. Com isso a dificuldade em administrar o sistema cresceu e uma nova ARPANET surgiu com usuários que não tinham relações militares.

O desenvolvimento na rede aconteceu de tal modo que todos os usuários se ajudavam e contribuíam para a formação de uma rede de computadores global. Esse crescimento trouxe dificuldades para a interconexão de novas redes demonstrando que os protocolos da ARPANET não eram adequados e não serviam para o cenário atual. Nesse contexto pesquisadores começaram a procurar novas maneiras que

resolvessem este problema, esse esforço resultou na criação dos protocolos e do modelo TCP/IP [7].

O TCP/IP foi criado com o objetivo de manipular a comunicação entre redes permitindo que o tráfego de informações fosse encaminhado de um lugar para outro com confiabilidade, com pacotes de diferentes tamanhos, tratando e recuperando erros de transmissão e falhas de sequenciamento de pacotes, além de realizar controle de fluxo, congestionamento e verificação de erros fim-a-fim.

Por meio da *National Science Foundation* (NSF) [36], o governo dos EUA e empresas particulares investiram na criação de *backbones*, que consistia em poderosos, porém simples, computadores conectados entre si com a capacidade de dar vazão a grandes fluxos de dados. Os *backbones* são ligações centrais de um sistema de rede mais amplo, ou mais elevado na hierarquia das redes de computadores.

O *backbone* pode ser comparado a uma grande estrada. Durante toda sua extensão há entradas e saídas para diversas cidades, as quais compõem redes de menor porte. Todas essas vias, ou pequenas redes, estão conectadas à estrada principal. É possível exemplificar *backbones* de ligações intercontinentais, que por sua vez derivam de *backbones* nacionais até *backbones* regionais. Neste ultimo encontram-se as empresas que exploram o acesso as redes de computadores e que nos fornece a conectividade com tais redes.

A união de todas as redes define uma interligação global chamada Internet. É importante destacar que a Internet não possui um único dono. Isso acontece pelo fato de que as conexões entre essas redes é feita considerando as políticas locais de cada instituição. Nesse sentido não há uma administração centralizada, embora existam organizações que se dedicam a definir padrões, normas e regras para sua utilização e disponibilização, o que garante o seu funcionamento. Podemos exemplificar as seguintes organizações:

- O *World Wide Web Consortium* (W3C) [46] que concentra-se na padronização de tecnologias da Web.
- A OASIS (*Organization for the Advancement of Structured Information Standards*) [37] é um consórcio internacional especializado no desenvolvimento de padrões para segurança da Web.
- A *Internet Engineering Task Force* (IETF) [28] que concentra-se no desenvolvimento da arquitetura das redes de computadores e a operação uniforme da mesma por meio de protocolos. Cada padrão da IETF é publicado como uma RFC (*Request for Comments*) e está disponível gratuitamente.
- O IEEE (*Institute of Electrical and Electronics Engineers*) [27] é uma organização que cria padrões nas áreas de tecnologia da informação, telecomunicações, medicina e saúde, transporte e outros.
- O comitê ISO (*International Organization for Standards*) [29] é o maior desenvolvedor de padrões do mundo e mantém uma rede de institutos nacionais de padronização em mais de 140 países.

2.2 Arquiteturas Atuais

Ao passar dos anos as redes de computadores foram crescendo tornando-se um aglomerado de redes em escala mundial. A interconexão de milhares de computadores por meio de protocolos de comunicação padronizados que permite o acesso a informações e a todo tipo de serviço de dados define a Internet, que é proveniente da expressão *internetwork* (comunicação entre redes).

Uma maneira simples de visualizar a Internet é considerar uma nuvem com computadores conectados a ela. Essa nuvem é considerada o núcleo das redes de computadores. É uma nuvem dinâmica e cresce a medida que crescem e nascem novas redes.

A arquitetura atual das redes de computadores atende praticamente os mesmos requisitos especificados na arquitetura da ARPANET, esses requisitos consistem em:

- conectividade - permite que qualquer estação de qualquer rede possa enviar dados para qualquer outra estação de qualquer outra rede;
- generalidade - dar suporte a diferentes tipos de serviços e aplicações;
- heterogeneidade - permitir a interconexão de diferentes dispositivos e tecnologias de rede;
- robustez - efetuar a comunicação desde que exista algum caminho entre origem e destino;
- acessibilidade - facilitar a conexão de novas estações ou redes.

Para atender tais requisitos a arquitetura atual das redes de computadores é fundamentada em uma arquitetura monolítica baseada em um modelo em camadas conhecido como modelo TCP/IP. Baseado no modelo mais geral proposto pelo OSI/ISO[1], o modelo TCP/IP é utilizado atualmente na Internet devido a sua abrangência e abordagem geral de características descritas em cada camada [44].

O modelo TCP/IP, ou a pilha de protocolos TCP/IP, é um padrão industrial de protocolos destinados as redes de computadores sendo as principais peças da arquitetura das redes de computadores atuais [31]. A arquitetura de redes tem como objetivo realizar a interligação de computadores e redes, não importando o tipo, entre si. A arquitetura TCP/IP está organizada em quatro camadas: a camada de aplicação; a camada de transporte; a camada de rede; e a camada de enlace/físico. (Figura 2.2).

Camada de Aplicação - A camada de aplicação localiza-se acima da camada de transporte, tem a função de prover serviços para as aplicações criando a comunicação, por meio de uma rede, com outras aplicações. A camada de aplicação é responsável por identificar e estabelecer a disponibilidade da aplicação nas máquinas disponibilizando os recursos para que tal comunicação aconteça. Ela contém todos os protocolos de nível mais alto tais como o SMTP(*Simple Mail*

[1] *Open Systems Interconnection* - OSI é o modelo lançado em 1984 pela *International Organization for Standardization* - ISO para definir formalmente uma arquitetura padrão para a comunicação entre máquinas e redes heterogêneas

Figura 2.2: Pilha TCP/IP.

Transfer Protocol), o FTP(*File Transfer Protocol*), o HTTP(*Hypertext Transfer Protocol*), DNS (*Domain Name Service*) e outros protocolos que foram incluídos no decorrer dos anos com novas aplicações e modificações na rede.

Camada de Transporte - A camada de transporte situada entre as camadas de aplicação e de rede, tem como função promover um canal de comunicação lógico fim-a-fim entre as camadas de aplicação rodando em diferentes computadores não se preocupando com os detalhes das camadas inferiores. Os protocolos de transporte são implementados, em geral, nos sistemas de borda da rede, já que por sua vez tem implementações mais complexas e oferecem um canal lógico fia-a-fim para a camada de aplicação.

Camada de Rede - A camada de rede é responsável pelo roteamento dos pacotes entre fonte e destino. A camada de rede permite que os *hosts* envie pacotes em qualquer rede garantindo que eles trafegarão independentemente até o destino, muitas vezes utilizando rotas diferentes chegando até mesmo em ordem diferente daquela em que foram enviado. A camada de rede é formada basicamente pelo protocolo IP que tem como característica o ”melhor esforço” (*best effort*), isto é, faz o melhor esforço para envio de um pacote entres os *hosts* não dando nenhuma garantia de entrega, obrigando então as camadas superiores tratarem suas limitações.

Camada de Enlace/Física - A camada de enlace/física é responsável pela transmissão e recepção de quadros da camada de rede de um dispositivo para a camada de rede de outro dispositivo fisicamente adjacente, estabelecendo um controle de fluxo por meio de um protocolo de comunicação entre sistemas.

O modelo TCP/IP tem o objetivo de reduzir a complexidade da arquitetura de rede por meio da divisão e do isolamento das funcionalidades da rede, permitindo que cada camada preste serviço para a camada superior. Nesse contexto o modelo TCP/IP faz com que a cada passagem entre as camadas haja um encapsulamento da camada superior em um novo tipo de *pacote* em que são adicionados cabeçalhos dos devidos protocolos das camadas correntes, o desencapsulamento ocorre no receptor, e também nos roteadores ao longo de sua rota, conforme mostra a Figura 2.3.

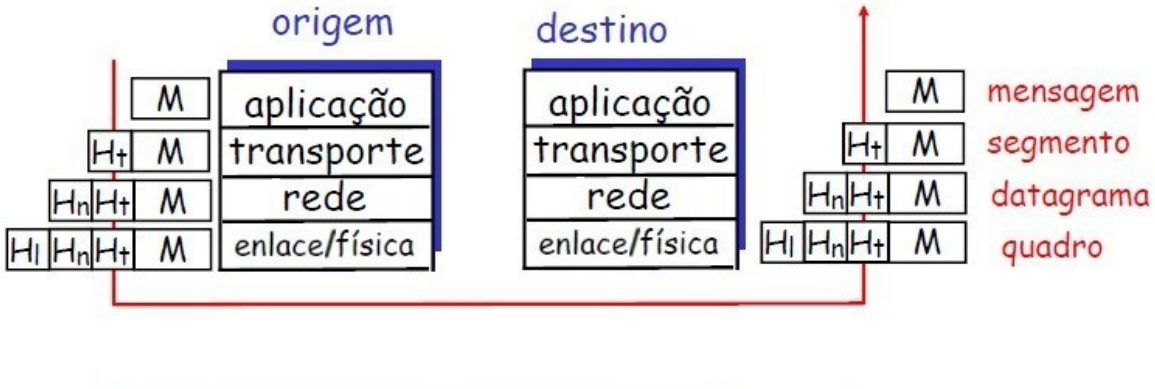


Figura 2.3: Protocolo TCP/IP [31].

Além disso a arquitetura atual das redes de computadores baseia-se na utilização da comutação em "*pacotes*", que divide uma informação ou mensagem em diversas unidades de tamanhos variáveis chamados de pacotes, em geral menor que o tamanho da mensagem original, e os envia por caminhos alternativos da origem até o destino [44].

Essa característica somada a uma topologia em malha garante a robustez e a conectividade da rede, pois com caminhos alternativos há uma garantia que se um nó da rede falhe o fluxo de dados não será interrompido, pois os pacotes podem procurar um novo caminho na rede caso haja uma infraestrutura existente, ou seja, um outro caminho da estação origem ao destino.

A utilização da comutação em pacotes também garante a eficiência da rede pois os nós evitam o desperdício de recursos devido ao compartilhamento da banda disponível [31]. Assim, os pacotes compartilham os recursos da rede usando totalmente a banda disponível do enlace não desperdiçando seus recursos como acontece no caso do paradigma de comutação por circuitos, onde os recursos são dedicados ao fluxo de dados não havendo o compartilhamento dos recursos do enlace.

Cada pacote é enviado utilizando o serviço de melhor esforço. O serviço de melhor esforço não requer que os nós sejam complexos garantindo a heterogeneidade da rede, permitindo assim a interconectividade de diferentes dispositivos e tecnologias. Essa característica resulta em computadores simples e de baixo custo no núcleo das redes. Por esse motivo, o núcleo das redes de computadores não garantem nenhum controle de fluxo nem tempo de envio, nem mesmo a entrega dos pacotes, deixando toda a responsabilidade de controle para as extremidades, ou seja, os computadores na borda da rede.

A rede também baseia-se na ideia de ser transparente, ou seja, o pacote é encaminhado da origem até o destino sem que a rede modifique seus dados, garantindo a generalidade da rede e, portanto fornecendo suporte a diferentes tipos de serviços e aplicações [35].

Outra ideia é a do transporte fia-a-fim que consiste em esconder do usuário final a complexidade do transporte dos dados. Dessa forma o núcleo da rede tem apenas a função de encaminhar os pacotes, com a qualidade do serviço de melhor esforço.

O transporte fim-a-fim provê a comunicação lógica entre os processos de aplicação executados nas estações de origem e destino.

Outra solução para o atendimento dos requisitos das redes de computadores é o uso do endereçamento global que garante a acessibilidade da rede, facilitando a conexão de novas estações e novas redes na Internet. O endereçamento global garante a unicidade de cada nó, é com base nessas informações que as decisões de encaminhamento dos pacotes são tomadas. Esse endereçamento é conhecido como endereço IP (*Internet Protocol*) que além de identificar os nós, provê uma forma de localização dos nós na rede global.

Em seguida novos requisitos foram incluídos na Internet tais como o controle distribuído, o cálculo global do roteamento e a divisão em regiões [35]. Todos estes requisitos e soluções são os princípios que configuram os protocolos e a estrutura das redes de computadores atuais.

Controle Distribuído - permite a divisão do sistema de controle em módulos interconectados por meio da rede de comunicação, proporcionando a divisão do processamento, a redução de custo, além de facilitar o diagnóstico e manutenção do sistema e de aumentar a sua flexibilidade e agilidade [43].

Calculo global de roteamento - define os caminhos que os pacotes devem seguir de uma estação de origem a uma estação de destino. Se a rede utiliza o paradigma de comutação por pacotes a decisão de roteamento é tomada para cada pacote. Se a rede utiliza o paradigma de comutação por circuito, o caminho a ser seguido é definido no estabelecimento do circuito virtual de maneira que todos os pacotes da mensagem devem seguir este caminho.

Divisão em regiões - a Internet é formada por um conjunto de redes de computadores interconectadas entre si [31]. Cada rede pode ser independente uma da outra e por sua vez pode utilizar regras e políticas de encaminhamento diferentes. Cada rede ou conjunto de redes, que operam sob um mesmo conjunto de regras administrativas define um Sistema Autônomo - SA (*Autonomous System*- AS). Cada SA se conecta a Internet por meio de um provedor de serviço de Internet (*Internet Service Provider* - ISP) que opera o *backbone* conectando o cliente a outros provedores de serviço. A comunicação entre cada SA é realizada pelo protocolo BGP (*Border Gateway Protocol*) que troca informações entre SA's vizinhos.

A colaboração entre os diferentes SA's garante que a rede seja totalmente distribuída. Esse tipo de estrutura atribuí robustez a Internet, pois caso algum SA falhe ou restrinja a comunicação, a rede constrói rotas alternativas que não utilizarão tal SA.

Na Internet, as redes de acesso são interconectadas segundo uma hierarquia de níveis de ISP's. No topo dessa hierarquia existe um número relativamente pequeno de ISP's denominados *Tier 1* ou ISP's de nível 1. Esses ISP's são especiais com capacidades de transmitirem pacotes a taxas extremamente altas, tem uma cobertura internacional e conectam-se diretamente a cada um dos outros ISP's de nível 1 [31]. Esse ISP's também são conhecidos como redes de *backbone* da Internet e por sua vez prestam servições para os ISP's de nível 2 (*Tier 2*).

Um ISP de nível 2 normalmente tem alcance regional ou nacional e conectam-se apenas a alguns poucos ISP's de nível 1. O ISP de nível 2 é denominado um *cliente* do ISP de nível 1, que, por sua vez, é denominado *provedor* do seu cliente. Abaixo do *Tier 2* estão os IPS de níveis mais baixos que se conectam a Internet por meio de um ou mais IPS's de nível 2 e, na parte mais baixa dessa hierarquia, estão os IPS's de acesso.

2.3 Evolução das Redes de Computadores

A utilização do modelo em camadas, a transparência e o principio fim-a-fim permitiram o crescimento da Internet. No entanto o fato da sua popularidade e do seu tamanho terem crescido absurdamente durante os últimos anos levou a incapacidade de sua estrutura suportar o grande número de usuários. Hoje as mesmas causas que levaram ao crescimento da Internet são as causas do seu engessamento e restrição do seu crescimento. Nesse sentido a estrutura das redes de computadores vem se modificando aos poucos para atender seus requisitos inicialmente propostos.

A Internet foi criada dando destaque à generalidade e heterogeneidade na camada de rede [35], ou seja, a Internet é feita de diversos e distintos dispositivos que podem se interconectar de maneiras diferentes. Sua estrutura, ou seja, seu *backbone* consiste de simples, porém poderosos, computadores que formam o núcleo da rede. Nesse sentido o núcleo da Internet é simples e transparente com inteligência nos sistemas de borda, que por sua vez são ricos em funcionalidades. Além disso sua administração é descentralizada, pois consiste na interligação de diversos SA's. Todo esse contexto leva ao famoso engessamento das redes de computadores.

Um núcleo simples, que se baseia no melhor esforço, não é capaz de fornecer informações sobre o funcionamento interno da rede. Isso implica que o usuário fique frustrado quando algo não funciona, pois ele não obtêm informação sobre tal erro. Outra consequência é a grande sobrecarga de configurações manuais, depurações de erros e complexidade no projeto de novas aplicações.

Para resolver esses problemas a estrutura das redes de computadores vem se modificando, ao longo do tempo, por meio de "adaptações" [35]. Todas essas "adaptações" foram introduzidas durante o tempo para atender as novas necessidades e requisitos, além do aumento da demanda que não estavam previstas no projeto original. Assim as redes de computadores vem sendo modificadas de diversas maneiras para fornecer uma maior escalabilidade, mobilidade, gerenciamento e segurança [9].

O serviço de nome de domínios (DNS) é uma dessas modificações. O DNS é um sistema de gerenciamento de nomes hierárquico e distribuído operando para resolver nomes de domínios em endereços de rede IP [31]. Outra dessas modificações foi a introdução do CIDR (*Classless Inter-Domain Routing*) que permite as organizações obterem uma faixa identificadora de rede de tamanho variavél [31]. Além dessas modificações podemos destacar a criação de sub-redes, de sistemas autônomos e modificações no protocolo TCP (*Transmission Control Protocol*).

O protocolo IP também sofreu modificações, entre elas podemos destacar o IP *multicasting*, que permite que um *host* envie pacotes de dados para um grupo de

hosts; o IPv6, que aumenta o número de endereços disponíveis, simplifica o cabeçalho antigo, permitir a identificação de fluxos entre outras coisas; O NAT (*Networking Address Translation*), que faz com que um conjunto de *hosts* passe a ser endereçado por um único endereço de IP; e muitas outras modificações.

Todas essas modificações foram necessárias para atender as demandas que vinham crescendo para as redes de computadores. As "adaptações" na arquitetura atual da Internet demonstram que seu projeto inicial não atende mais aos requisitos atuais, e também não podemos ajustá-los para as necessidades futuras. Nesse sentido a arquitetura atual das redes de computadores apresenta inúmeros problemas ainda não solucionados tais como: o problema de endereçamento IP, a mobilidade da rede, a segurança, o gerenciamento, a escalabilidade, além da disponibilidade de serviço e sua qualidade.

Endereçamento - O endereçamento contém problemas associados a escassez de endereços e ao número de informações que um endereço IP carrega, tais como localização e identificação [30]. Temos também problemas da falta de autenticidade do endereço, visto que uma estação pode-se passar por outra usurpando seu endereço IP. Mesmo com as modificações propostas para o endereçamento IP tais como NAT, DNS e CIDR a Internet tem crescido tanto que essas modificações já não conseguem suprir todas as necessidades atual tais como problemas de segurança, autenticidade, replicação de dados e serviços de rede.

Mobilidade - Cada vez mais cresce o número de dispositivos móveis que utilizam a Internet sem fio. A questão da mobilidade consiste na transição entre os nós móveis em diferentes pontos de acesso sem que haja a perda da conexão. Essa questão fere alguns princípios do projeto original da Internet e mesmo com as modificações propostas para resolver este problema, hoje a arquitetura atual das rede de computadores já não atendem mais satisfatoriamente os requisitos atuais.

Segurança - Problemas como a disseminação de vírus, a negação de serviço e *spams* não foram previstas no projeto inicial da Internet. Hoje esses problemas estão cada vez maiores atingindo cada vez mais usuários na rede. Dessa maneira fez-se necessário que a segurança fosse tratada pela borda da rede, ou seja, os computadores finais. Mesmo com essas modificações ataques distribuídos de negação de serviço, por exemplo, ainda consistem em um problema na arquitetura atual.

Gerenciamento - Na arquitetura atual das redes de computadores o gerenciamento da rede é feito de forma distribuída com inteligência nas bordas. O crescimento da Internet tem prejudicado esse gerenciamento devido a dificuldade de se gerenciar distribuidamente tantos fluxos de dados. Além disso a arquitetura atual não dispõe de ferramentas de diagnóstico que possam identificar a origem de problemas de funcionamento tornando o gerenciamento ainda pior. Uma das modificações para se resolver esse problema foi o protocolo SNMP (*Simple Network Management Protocol*) [10][2]. No entanto o SNMP restringe-

[2] *Simple Network Management Protocol* - SNMP é um protocolo de gerenciamento de rede que realiza o intercâmbio de informação entre os dispositivos de rede.

se basicamente a monitoração dos dispositivos da rede fornecendo apenas um diagnóstico simples dos dispositivos de rede, deixando ainda em aberto o problema de gerenciamento de aplicações e servidores. Dessa maneira o SNMP é insuficiente para atender as necessidades atuais.

Escalabilidade - A escalabilidade é um dos maiores problemas da Internet atual. O aumento do número de estações nas redes leva a um aumento exponencial nas tabelas de roteamentos dos dispositivos do núcleo da rede. Além disso, com o aumento do número de estações, aplicações que demandam muita banda começaram a ser mais utilizadas, como é o caso de aplicações multimídia. Outro problema da escalabilidade está relacionado as redes móveis, pois em grande parte dos seus protocolos há restrições quanto ao número máximo de nós por rede.

Disponibilidade - A disponibilidade da rede consiste em oferecer um serviço de rede confiável, robusto e que fique sempre disponível. No entanto, devido ao tamanho crescimento nas redes de computadores, a infraestrutura atual da Internet não é capaz de oferecer tal serviço. Embora a Internet tenha sido projetada para obter uma melhor disponibilidade de serviço que uma rede telefônica, esta por sua vez oferece uma maior confiabilidade de entrega de dados. Isso deve-se ao fato que a Internet utiliza a comutação por pacote e as redes telefônicas utilizam a comutação por circuito. Embora a segunda tenha sua confiabilidade de entrega dos dados garantida não contem a robustez que a primeira tem. Todavia, os serviços atuais de disponibilidade já não são mais suficientes no contexto atual devido a inúmeras falhas dos provedores de serviço e dos usuários. Estudos mostram que apenas 35% das rotas ficam disponíveis por 99,9% do tempo [32], o tempo de indisponibilidade de serviços na rede devido a erros de *software* e de usuários chega a 40% do tempo de conexão [8]. Disponibilidade do acesso a rede sem fio chega a 43% abaixo do divulgado [3] devido a infraestrutura da rede. Isso demonstra a necessidade de uma nova arquitetura que seja capaz de lidar de maneira mais eficiente com erros e que simplifique as tarefas dos usuários, uma vez que o perfil das pessoas que acessam a rede se modificou.

Qualidade de Serviço - A Qualidade de Serviço (*Quality of Service* - QoS) tem sido amplamente estudada pela comunidade cientifica, no entanto prover QoS na arquitetura atual da Internet é uma tarefa complicada devido às várias restrições que a infraestrutura da rede atual impõe. Implementações para dar suporte ao QoS vão contra a estrutura atual da rede, pois qualquer mecanismo de reserva de banda ou mudança de prioridade nos pacotes irão afetar o principio de melhor esforço. É interessante que os provedores de serviços ofereçam QoS para se destacar na concorrência de mercado, no entanto é bastante complicado garantir o QoS devido ao fluxo de dados da origem ao destino, que muitas das vezes atravessarem diversos SA's durante a comunicação. Assim, para se garantir o QoS na arquitetura atual é necessário que haja um acordo entre os diversos SA's de origem ao destino do pacote. Dessa forma torna-se necessário projetar uma nova arquitetura de rede para lidar com este problema.

Como foi apresentado, as modificações realizadas na arquitetura da rede atual já não são suficientes para atender alguns problemas recentes, além disso algumas modificações ferem os requisitos da Internet. Sendo assim, se faz necessário a introdução de mudanças no núcleo da rede, porém essas mudanças não são bem vistas entre os administradores de rede, devido ao fato destes não arriscarem a implementação de novos serviços que possam indisponibilizar, mesmo que por pouco tempo, o uso da rede, ou que não sejam efetivamente melhores.

A falta de confiabilidade nestes novos serviços deve-se ao fato que pesquisadores ao criarem novos protocolos não conseguem, em geral, testar estes em redes reais, somente em redes simuladas por meio de simuladores de rede. Dessa maneira esses novos serviços não garantem que são melhores que os serviços atuais e por muitas vezes não são postos em prática, pois os administradores não querem correr o risco de indisponibilizar seu serviço.

Enfim, acaba se tornando um problema sem solução, pois novos protocolos não são testados em redes reais, devido a necessidade de mudanças em milhares de equipamentos legados, e por esse motivo provedores de serviço não adquirem confiança necessária para a ampla implementação em ambientes reais. Dessa forma, muitos pesquisadores consideram que a infraestrutura de rede está ”ossificada”, não podendo ser modificada [19].

2.4 Arquiteturas do Futuro

As seções anteriores mostraram que a simplicidade da arquitetura atual das redes de computadores juntamente com os requisitos preestabelecidos da pilha de protocolos TCP/IP permitiu o seu crescimento fácil e rápido já que as novas aplicações que surgiam não necessitavam realizar modificações no núcleo da rede, apenas nos computadores de borda. No entanto esses mesmos motivos que fizeram a Internet crescer foram os mesmos motivos que pararam seu crescimento.

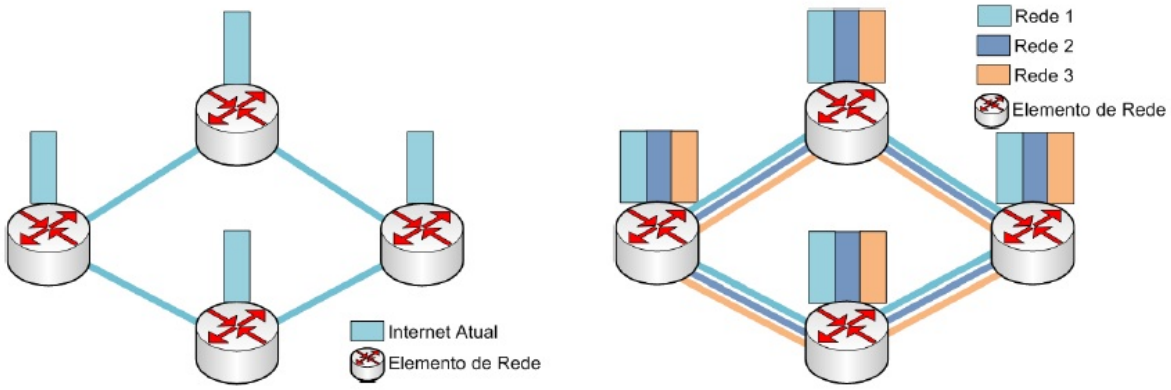
A Internet se tornou ossificada tornando suas modificações de difícil implementação [19]. O grande crescimento da Internet trouxe diversos problemas estruturais que já não podem ser resolvidos apenas com modificações da arquitetura atual. Dessa maneira sua estrutura simples que levou a Internet a crescer, agora está limitando seu crescimento. Algumas das necessidades atuais só podem ser solucionadas com mudanças estruturais [2], portanto é necessário uma nova arquitetura para atender os requisitos atuais e os requisitos das futuras aplicações que serão ainda propostas.

As arquiteturas do futuro para as redes de computadores são propostas que visam promover a flexibilidade com a capacidade de mudar e evoluir gradativamente, dando suporte à inovação no núcleo da rede. Os modelos que demonstram a Redes de Computadores do Futuro podem ser divididos em duas abordagens ou paradigmas, a abordagem purista e a abordagem pluralista [11, 17, 2].

A abordagem purista, modela as redes de computadores em uma arquitetura monolítica, com uma única pilha de protocolos executando sobre a estrutura física das redes (Figura 2.4a).

Essa abordagem assemelha-se a arquitetura atual das redes de computadores com a diferença que os protocolos nela utilizados deverão ser flexíveis e adaptáveis o suficiente para garantir a interoperabilidade dos elementos de rede aos novos requisitos e demandas. Nesta abordagem a virtualização e as redes sobrepostas são apenas ferramentas que podem ser usadas para agregar novas funcionalidades não fazendo parte de um aspecto fundamental da arquitetura em si.

A segunda abordagem, pluralista, aborda uma arquitetura voltada a múltiplas pilhas de protocolos executando simultaneamente (Figura 2.4b). Essa abordagem faz com que diversas redes possam executar em paralelo para atender aos requisitos de cada nova aplicação. Além disso, sua implementação pode acontecer de forma gradual, pois esta abordagem é compatível com a atual.



(a) Arquitetura Atual e Purista

(b) Arquitetura Pluralista

Figura 2.4: Representação dos tipos de arquitetura da Internet do Futuro [17].

Todas as abordagens pluralistas baseiam-se na ideia de executar múltiplas redes virtuais sobre um substrato físico compartilhado [17]. Nessa abordagem as técnicas de virtualização de redes tornam-se fundamentais para permitir a coexistência em paralelo de múltiplas arquiteturas de protocolos. Com a virtualização de redes as redes virtuais, no entanto, irão se diferenciar no formato dos pacotes, na maneira como será tratado o endereçamento dos pacotes e nas execuções dos protocolos, portanto, cada pilha de protocolos, embora divida o mesmo substrato físico, é independente.

Comparando as duas abordagens apresentadas, a abordagem purista apresenta-se ainda mais complexa do que a pluralista. A abordagem purista torna-se mais complexa pois a nova arquitetura e novos protocolos de rede terão de ser capazes de resolver todos os problemas atuais e o problemas que ainda estão por vir.

Já a abordagem pluralista é mais simples no sentido de que sua implementação se dará de forma gradual de maneira que cada rede virtualizada possa atender a necessidade de cada nova aplicação, visto que toda sua ideia baseia-se em executar redes virtualizadas sobre um substrato físico compartilhado.

Existem outras abordagens além das abordagens purista e pluralista na literatura. Sendo essas arquiteturas que resolvem problemas específicos, como roteamento, segurança, qualidade de serviço entre outros. Além dessas abordagens não resolverem o problema, como um todo, não está claro como integrar essas diversas arquitetu-

ras em um único arcabouço consistente, visto que essas arquiteturas em geral são incompatíveis umas com as outras, não podendo ser usadas simultaneamente.

A abordagem purista defende a ideia que essas soluções especificas poderiam ser integradas a uma única arquitetura devido a sua flexibilidade. Por outro lado, os pluralistas defendem a ideia que tais redes especificas poderiam atuar simultaneamente por meio de técnicas de virtualização de redes. Ambas as ideias são muito discutidas no meio acadêmico [2].

2.5 Redes Virtualizadas

Como já apresentado, a injeção de novas ideias e implementações no núcleo da rede sofre discriminação pelos administradores de rede devido a falta de confiança no bom funcionamento na rede, devido aos riscos de indisponibilizar a rede, além do custo benefício a eles envolvidos. Uma das propostas vistas como alternativas para o desenvolvimento de inovações juntamente ao tráfego de produção dar-se com a virtualização de redes.

Para explicar a virtualização de redes, vamos primeiro explicar a virtualização de sistemas. A virtualização de sistemas é uma técnica que permite que um nó computacional, ou seja, uma máquina, execute múltiplos processos oferecendo a cada um deles a ilusão de estar executando sobre recursos dedicados. Dessa forma cada máquina virtual acessa interfaces similares a máquina real. Para que as máquinas virtuais tenham a impressão de estarem sendo executadas sobre uma maquina real, os ambientes virtuais devem ser isolados, ou seja, a execução de uma máquina virtual não deve interferir na outra, assim uma máquina virtual deve ter acesso a máquina real como se fosse a única.

Nesse contexto a virtualização de redes também faz uma abstração de um recurso. A virtualização de redes passou a permitir que os componentes de uma rede física compartilhassem sua capacidade de maneira que realize simultaneamente múltiplas funções, estabelecendo infraestruturas lógicas distintas e mutuamente isoladas provendo um método para que múltiplas arquiteturas de rede distintas compartilhem o mesmo substrato físico. Em outras palavras uma rede virtual é uma rede composta pela interconexão de um conjunto de roteadores virtuais que representam uma ”fatia” de roteadores físicos compartilhados.

Como as redes virtuais são isoladas, o tráfego experimental não afetará o trafego de produção (Figura 2.5). Esse modelo consiste em uma das principais abordagens para as redes de computadores do futuro, em que cada rede virtual é isolada e possui sua própria pilha de protocolos e seu gerenciamento individual.

A topologia de uma rede virtual não precisa ser idêntica à topologia de uma rede física, embora seja necessário uma rede física para transportar os dados. Uma rede virtual é uma rede composta por roteadores lógicos conectados em uma determinada topologia. Dessa maneira os enlaces virtuais são criados pelo particionamento do enlaces físicos, tal particionamento corresponde a uma ”fatia” da banda disponível no enlace físico, portanto, a banda do enlace físico é dividida entre os enlaces virtuais.

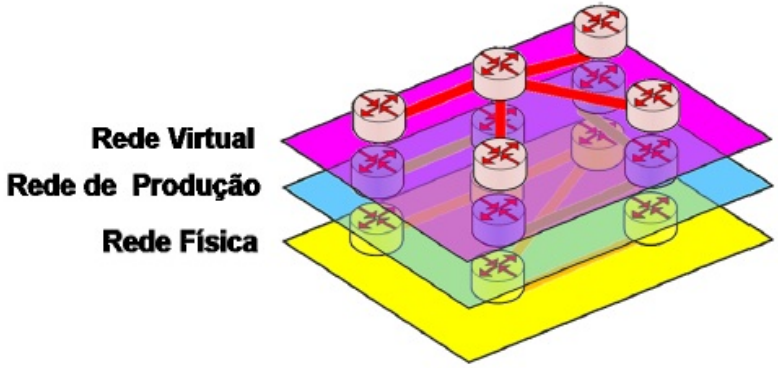


Figura 2.5: Exemplo de uma rede virtualizada, com três redes compartilhando o mesmo substrato físico [33].

Como a representação da topologia das redes virtuais não é necessariamente igual a topologia da rede real, para representar roteadores virtuais adjacentes em roteadores físicos não adjacentes, utiliza-se o tunelamento, que cria túneis entre os roteadores físicos para tornar a rota de transmissão transparente para a topologia virtual. Essa característica de poder mapear a rede virtual em uma rede física permite que as redes virtuais sejam bastante flexíveis. Para que haja essa flexibilidade se faz necessário uma função de migração de redes.

Uma função de controle fundamental nas redes virtualizadas é a migração de redes virtuais. A migração faz a movimentação dos nós das redes virtuais sobre os nós da rede física [47]. Podemos aplicar a migração de diversas maneiras como por exemplo a manutenção de nós na rede, que muitas das vezes causa uma quebra na conexão devido ao seu desligamento causando um atraso para que as rotas sejam reorganizadas entre os nós. Assim com a migração poderíamos contornar este cenário com uma simples migração do nó virtual para um nó físico que esteja em funcionamento. Dessa maneira as redes virtuais garantem que a topologia lógica não seja alterada e assim as rotas continuem válidas.

Além dos casos de manutenção podemos destacar os casos de ataques de negação de serviço, bastando trocar todos os nós do substrato em ataque para outros nós físicos fora da região do ataque, casos para economia de energia, quando há nós físicos subutilizados a migração pode os detectar na rede física e assim poderem ser desligados entre outros.

A migração de redes virtuais ainda sofre alguns desafios tais para remapear os enlaces virtuais sobre um ou mais enlaces físicos e como reduzir o tempo de migração, visto que os nós virtuais ficam indisponíveis durante o processo possibilitando a perda de pacotes no núcleo da rede. As redes virtuais trazem novos horizontes para o núcleo da rede, com essa grande flexibilidade as redes virtuais permitem a instanciação, a remoção e a configuração de recursos de redes virtuais sob demanda permitindo também que as redes sejam monitoradas enquanto estão ativas. Dessa maneira a virtualização tem sido amplamente usada para o desenvolvimento de propostas para a Internet do Futuro [35], e para o desenvolvimento de redes experimentais. Dessa forma, a comunidade cientifica começou a investir mais no tema, pois as

redes virtuais são capazes de oferecer ambientes reais para testes de novas ideias.

Nesse contexto, os pesquisadores de rede estão trabalhando em algumas redes virtuais programáveis como a GENI [26], um centro de investigação para a experimentação de novas arquiteturas de rede e sistemas multiusuários. Esse tipo de rede tem como objetivo possibilitar a experimentação de protocolos em larga escala por meio de Switches e Roteadores programáveis utilizando virtualização. Assim, essa rede deve possuir Switches e Roteadores espalhados por todo o mundo que podem ser modificados para executar protocolos experimentais. A proposta da GENI em geral é oferecer ao pesquisador uma ”fatia” dos recursos da rede que consiste em enlaces, roteadores, switches e terminais para que ele possa executar seus experimentos, por meio da virtualização, podendo configurar seus recursos como desejar.

As redes virtualizadas, embora forneçam uma grande flexibilidade para controlar o núcleo da rede, carregam consigo algumas desvantagens. O gerenciamento de uma rede virtual se parece muito com o gerenciamento de uma rede física, além disso temos uma banda muito limitada, visto que a banda de enlace virtual é uma fatia da banda física real. Outro problema é o mapeamento dos nós virtuais entre os reais, que exige grande atenção, pois se um dos nós físicos parar de funcionar todos os nós virtuais a ele referenciados deixaram de funcionar até que haja uma migração, causando prejuízos e atrasos na transmissão da rede.

Nesse contexto, o paradigma de redes virtuais é parte de uma das abordagens e propostas para a Internet do Futuro. A abordagem pluralista se baseia nessas redes como parte de sua arquitetura, garantindo maior flexibilidade no núcleo da rede, permitindo que diferentes requisitos de aplicações sejam atendidos. Esse paradigma possibilita que inúmeros elementos de redes virtuais possam co-existir em um único equipamento físico, mas possui ainda enormes desafios a serem vencidos para que esta arquitetura seja uma realidade.

2.6 Redes Definidas por *Software*

Mesmo com o grande crescimento e evolução da Internet, observamos que sua arquitetura não evoluiu suficientemente nos últimos anos. Embora todas as modificações realizadas, a arquitetura de Internet já não está mais atendendo as demandas das novas aplicações.

Lembramos também que com toda essa evolução das redes de computadores, a mesma tornou-se comercial e nesse sentindo os equipamentos de rede tornaram-se ”caixas pretas”, ou seja implementações integradas baseadas em *software* e *hardware* proprietário. O resultado de toda essa evolução causou o já comentado engessamento da redes de computadores [19]. Em contraste a essa realidade os pesquisadores de redes e a comunidade científica começaram a desenvolver novas propostas para a criação da redes de computadores do futuro, ou seja, novas arquiteturas de implementação do núcleo da rede.

Uma das principais propostas para essa nova arquitetura das redes de computadores baseia-se em redes capazes de ser programadas sob demanda, ou seja, redes que sejam programáveis ou redes que sejam flexíveis para lidar com requisitos e

problemas atuais e futuros. Uma das formas de se prover programabilidade às redes é por meio da implementação de Redes Definidas por Software.

Redes Definidas por Software, ou redes programáticas, são redes cujo substrato físico é composto por equipamentos de propósito geral e a função de cada equipamento, ou conjunto de equipamentos, é realizada por um *software* especializado [33]. As Redes Definidas por Software são baseadas na separação entre o plano de controle e o plano de dados.

O plano de controle é responsável pelos protocolos e pelas tomadas de decisão que resultam na confecção das tabelas de encaminhamento. O plano de dados, também chamado plano de encaminhamento, cuida da comutação e repasse dos pacotes de rede. Como já mencionado, nos equipamentos atuais ambos, o plano de controle e de dados, são executados e implementados no próprio equipamento, impedindo qualquer tomada de decisão que não tenha sido prevista nestes protocolos.

Para que haja uma tomada de decisão, é necessário quebrar essa restrição permitindo que o equipamento de rede encaminhe os pacotes por meio de protocolos e implementações externas sendo abrigadas em uma máquina física ou virtual. Nesse sentido o plano de controle será independente daquele pré-configurado e não se limitará aos protocolos implementados pelo proprietário ou fabricante.

Para que se estabeleça uma interface de comunicação entre o plano de controle e o plano de encaminhamento é necessário que os desenvolvedores criem APIs para que se possa realizar tal comunicação. No entanto, isso acarretaria na criação de múltiplas APIs diferentes para cada fabricante. Para resolver esse problema pode-se optar pela padronização destas APIs. Mesmo com a padronização, ainda assim haveria uma limitação nas possíveis ações a serem desempenhadas pelo plano de controle, já que estas mais uma vez deveriam ser previstas pelo *software* do equipamento.

O OpenFlow [42] é uma proposta que permite a implementação das Redes Definidas por Software, uma vez que estabelece uma interface de comunicação entre o plano de encaminhamento e o plano de controle com um padrão de interface bastante flexível para que seja possível a instalação de regras de encaminhamento baseadas em diversos parâmetros de protocolos de camadas distintas.

Sendo assim, existem duas principais correntes para prover o conceito de Redes Definidas por Software em redes na arquitetura do Futuro. A primeira é representada pelo OpenFlow, uma tecnologia promissora, que provê alto desempenho e controle da rede, a outra corrente é representada pelas propostas de arquiteturas de redes baseadas em redes virtualizadas, a qual fornece uma grande flexibilidade para controlar o núcleo da rede. Ambas fazem parte da abordagem pluralista de arquiteturas da Internet do Futuro.

2.7 Sistema OpenFlow

As redes virtualizadas como a GENI são interessantes para diminuir as barreiras para o surgimento de novas ideias e testes dessas mesmas, porem a implementação dessas redes podem ser custosas devido sua grande escala [19]. Para permitir a ex-

perimentação de novas propostas para redes de computadores em escalas menores, foi proposto o Comutador OpenFlow [42], cujo maior objetivo é possibilitar que um pesquisador execute seus experimentos em uma rede real. Nesse sentido a plataforma OpenFlow tem como objetivo criar um ambiente de rede de teste programável, unindo as qualidades da virtualização de redes com o conceito de Redes Definidas por Software.

A virtualização de redes, utilizando uma rede IP sobre IP, consegue de maneira simples criar uma rede de testes com uma escala considerável [35]. Os elementos desse tipo de rede virtual são roteadores que executam sobre uma plataforma de virtualização, ou seja, são roteadores virtualizados. Nesse cenário, os roteadores virtuais são máquinas virtuais que executam funções de roteamento.

Roteadores virtuais agem de maneira similar aos roteadores físicos convencionais, mas o desempenho dos roteadores virtuais é inferior ao dos roteadores físicos convencionais, pois esse tipo de virtualização tem a inconveniência de utilizar implementações baseadas na arquitetura das redes atuais apresentando problemas de isolamento nas operações de entrada e saída do roteador físico. Desse maneira qualquer nova proposta realizada em redes virtualizadas, IP sobre IP, tem a desvantagem de carregar consigo as características e limitações da arquitetura atual da rede.

O OpenFlow é uma plataforma de virtualização de redes baseada na comutação de fluxos. A diferença é que o comutador OpenFlow utiliza o conceito de redes definidas por *software*, cujo o substrato físico é composto pela parte que cuida do tráfego de produção da rede e outra parte que cuida do tráfego experimental das novas propostas de rede. Assim, a plataforma OpenFlow procura oferecer uma opção controlável e programável, 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 2.6.

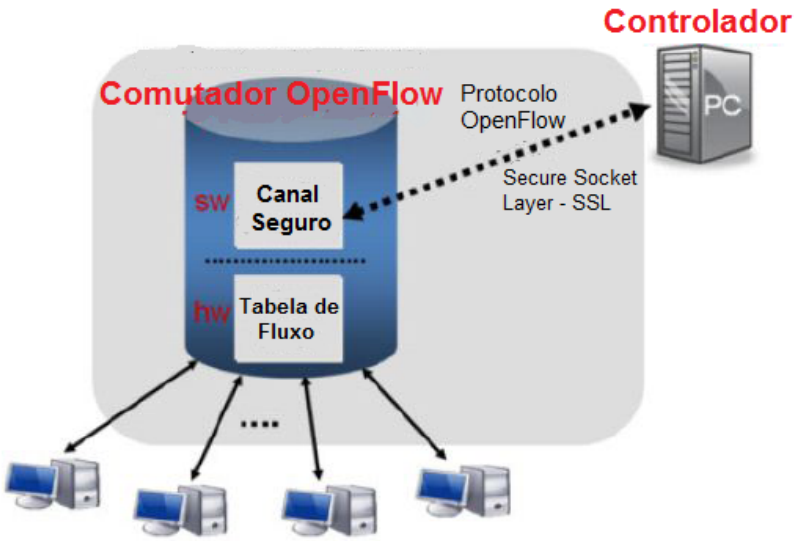


Figura 2.6: Arquitetura do comutador OpenFlow [42].

A vantagem do protocolo OpenFlow é que com sua utilização, pesquisadores podem testar suas experiências sem que interfira no tráfego real da rede de produção e sem a necessidade que os fabricantes de comutadores exponham os projetos de *software* e hardware dos seus equipamentos.

O OpenFlow possibilita que os fabricantes dos roteadores atuais possam adicionar as funcionalidades do OpenFlow aos seus comutadores sem necessitarem expor o projeto de *software* e hardware do seus equipamentos. É necessário deixar claro que esses equipamentos devem possuir baixo custo e desempenho semelhante aos já utilizados, de forma que os administradores da rede aceitem a substituição dos equipamentos já existentes.

O comutador OpenFlow tem como característica a implementação em baixo custo, pois não se faz necessário que haja grandes mudanças no núcleo da rede. As mudanças podem ser realizada de forma gradual e não influenciarão o fluxo de produção da rede. Assim, todas a políticas de roteamento e de tráfego não sofrerão mudanças e caso sofram, essas não deverão ser significativas. Dessa maneira sua implementação na infraestrutura será realizada com um baixo custo e com um alto desempenho, pois os pesquisadores não irão influenciar no tráfego de produção, deixando o fluxo de dados com desempenho semelhante ao de um comutador normal.

A arquitetura OpenFlow permite que vários pesquisadores utilizem o comutador para realizar várias pesquisas ao mesmo tempo. Isso acontece, pois comutador OpenFlow virtualiza o tráfego experimental da rede para que as várias pesquisas possam compartilhar o mesmo fluxo experimental. Outro ponto importante no paradigma OpenFlow é a generalização do plano de dados, ou seja, qualquer modelo de encaminhamento de dados que se baseie na tomada de decisão fundamentada em algum valor do campo de cabeçalho dos pacotes pode ser suportada. Dessa maneira a arquitetura OpenFlow pode suportar uma ampla gama de pesquisas científicas.

A garantia do isolamento do tráfego experimental do tráfego de produção é uma das melhores características da arquitetura OpenFlow, pois permite que a infraestrutura se modifique sem que interfira na produção do SA. Isso acontece pois a arquitetura do comutador OpenFlow é subdividida em dois planos, o plano de controle de produção, implementado em *hardware*, e o plano de controle experimental, implementado no controlador OpenFlow e executado no *hardware*.

Os comutadores OpenFlow implementam módulos que permitem a comunicação com o controlador de forma independente e segura, separada do fluxo real da rede. Dessa maneira as entradas na tabela do fluxo experimental dos comutadores são interpretadas como decisões em *cache* realizadas no *hardware* e são planejadas no plano de controle no *software*. Em outras palavras isso quer dizer que as decisões do fluxo experimental são planejadas no *software* e executadas no *hardware*.

O OpenFlow apresenta várias características que ajudam na difícil transição da arquitetura atual para as arquiteturas do futuro. Visto que essa transição não pode ser feita instantaneamente, o paradigma OpenFlow irá realizar essa transição de maneira gradativa, pois as novas propostas dos pesquisadores poderão agora ser validadas em rede real não atrapalhando o fluxo de produção da rede. Dessa maneira os administradores de rede terão maior confiabilidade nas novas propostas de roteamento, pois estas não foram validadas em redes simuladas, e assim aceitaram implementar as novas ideias em suas redes.

Capítulo 3

Redes Definidas por Software

Como podemos observar, apesar de toda essa evolução formidável das redes de computadores, em termos de penetração e aplicações, sua arquitetura, que fez com que sua evolução fosse possível, não evoluiu da mesma forma e por esse motivo já não é capaz de atender os requisitos atuais da Internet. Além do mais, a Internet se tornou comercial e por sua vez seus equipamentos de rede se tornaram caixas pretas, isto é, implementações baseadas em *software* fechado sobre um *hardware* próprio.

Dessa forma as redes de computadores se tornaram enormes e ao mesmo tempo estão restringidas pelas suas próprias limitações, sua arquitetura distribuída e fechada. Contrapondo a abordagem da arquitetura atual das redes de computadores, surgiram ideias e pesquisas sobre a Internet do Futuro, pesquisas que visam resolver os problemas até então enfrentados pela sua arquitetura monolítica distribuidamente fechada, tais como problemas associados ao endereçamento, a mobilidade, a escalabilidade, o gerenciamento e a qualidade de serviço.

As abordagens da Internet do Futuro visam resolver os problemas atuais e atender as novas demandas de requisitos estabelecidos pela Internet. Em geral essas abordagens se caracterizam por por um plano de controle concentrado que permitem mover grande parte da lógica de tomada de decisões dos dispositivos de redes para controladores externos, que podem ser implementados com o uso de tecnologia de servidores comerciais, um recurso abundante, escalável e barato [40].

Nos equipamentos de rede tradicionais as tomadas de decisão ocorrem no seu próprio interior, ou seja, o roteamento dos pacotes de rede é definido por algorítimos previamente calculados geralmente fechados, de difícil ou impossível modificação, uma vez implementado em uma rede o equipamento toma suas próprias decisões para enviar o pacote.

Se o controle das tomadas de decisão fosse logicamente centralizado haveria a possibilidade da definição do comportamento da rede em software, não apenas pelos próprios fabricantes do equipamento, mas também por fornecedores ou pelos próprios usuários, como, por exemplo, operadores de rede.

As Redes Definidas por Software (*Software Defined Networks, (SDN)*) constituem esse novo paradigma para o desenvolvimento das redes de computadores. As redes SDN abrem novas perspectivas em ambientes de controle lógico da rede, em novas

aplicações de rede podendo ser desenvolvidas de forma simples e livre dos limites da arquitetura atual.

3.1 Sugimento das Redes Definidas por Software

Como vimos no capítulo anterior, as redes de computadores fazem parte da infraestrutura crítica do dia-a-dia da sociedade, logo novas propostas para novas soluções de rede fazem-se necessárias mas ao mesmo tempo são difíceis de serem implementadas.

Isso ocorre devido ao risco de interrupção das atividades que pode ocorrer ao implementar essas novas propostas. Há também problemas econômicos, pois a adoção de novas ideias, em geral, requer de alterações do *hardware* utilizado na rede, fazendo com que os administradores de rede inviabilizem tais reformas estruturais.

Esses problemas levam a ossificação das redes de computadores. A Internet atingiu um nível de amadurecimento que a tornou pouco flexível. Como visto anteriormente, a comunidade cientifica visa contornar esse problema com a implementação de redes com maiores recursos de programação. Exemplos como a GENI [26] que apostam em recursos de virtualização de rede para permitir que se tornem programáveis, apesar de ter um grande potencial, tiveram pouca aceitação pelos administradores de rede pela tamanha necessidade de alteração dos elementos de rede.

Uma maneira de contornar esse problema reduzindo o impacto de interferência na rede de produção, consiste em estender o *hardware* de encaminhamento de pacote de forma mais restrita sobre um controle fino de tráfego. Basta atribuir um rótulo ao pacote que determinará como o mesmo sera tratado pelos outros elementos da rede.

Dessa forma administradores de rede podem exercer controle diferenciado sobre determinados tráfegos na rede. Com base nessa característica esse *hardware* permite que um administrador de rede desenvolva aplicações de rede que determinam como os fluxos serão tratados na rede de produção.

Foi com essa ideia que o desenvolvimento do protocolo de rede OpenFlow [42] partiu. O protocolo OpenFlow permite que os elementos de encaminhamento da rede, os comutadores, possam conter uma interface de programação simples que permita o acesso ao controle de fluxo, ou a tabela de comutação, utilizado pelo *hardware*. Assim o roteamento dos pacotes poderão ser programáveis de maneira que os pacotes recebidos sejam encaminhados pelas portas programadas pelo administrador.

O encaminhamento dos pacotes ainda será eficiente, visto que ocorrerá ainda no *hardware*, porém as decisões de cada pacote encaminhado poderá ser processado por uma camada superior em que diferente funcionalidades podem ser implementadas. Essa estrutura permite que a rede seja controlada por meio de aplicações desenvolvidas em *software*. A essa nova abordagem da arquitetura de redes dá-se o nome de Redes Definidas por Software, ou *Software Defined Networks (SDN)*.

3.2 Motivaçao

A abordagem de redes SDN vem sendo muito pesquisada no meio acadêmico. Isso ocorre devido as diversas possibilidades de aplicação que esse paradigma traz para as redes de computadores. As redes SDN se apresentam como uma forma potencial de implantação da Internet do Futuro.

Propostas como o OpenFlow abordam a ideia de que não é necessário que a comutação de pacotes seja definida pelo principio de roteamento de redes IP. Os elementos da rede, roteadores e *switches*, não irão mais controlar a parte lógica do processo de comutação de pacotes. O processamento de comutação de pacotes poderá ser controlado por aplicações em *software* desenvolvidos independentemente do *hardware*.

No entanto o universo SDN é muito maior que do que aquele definido pelo controle do processo lógico de comutação, definido pelo padrão OpenFlow. O OpenFlow é apenas uma das aplicações abordadas pelas Redes Definidas por Software, o novo paradigma abre também a possibilidade de se desenvolver outras aplicações que controlam os elementos de comutação de uma rede física de maneiras totalmente distintas das que é possíveis com a arquitetura atual das redes de computadores.

O desenvolvimento dessas aplicações torna-se possível por meio de controladores de rede, chamados de Sistemas Operacionais de Rede. Esses elementos oferecem um ambiente de programação favorável para o desenvolvimento dessas aplicações, como por exemplo, desenvolver técnicas de roteamento para determinados tipos de fluxos de dados na rede.

Os Sistemas Operacionais de Rede formam um ambiente de programação onde o desenvolvedor pode ter acesso aos eventos gerados por uma interface de rede, por exemplo o OpenFlow, gerando os comandos para controlar a estrutura de chaveamento e comutação. Com esse tipo de aplicação, torna-se mais simples implementar políticas de segurança baseados em níveis de abstrações maiores que os atuais endereços IP's cobrindo todos os pontos de rede, por exemplo. [14]

As redes SDN possibilitam a implementação de aplicações de rede que realizam lógicas de monitoração e acompanhamento de tráfego mais sofisticado e mais completo que os atuais.

3.3 Arquitetura

A arquitetura atual do roteadores de rede (Figura 3.1) é formada basicamente por duas camadas distintas: O *software* de controle; e o *hardware* dedicado ao encaminhamento de pacotes.

O primeiro é encarregado de tomar as decisões de roteamento, definindo a tabela de comutação para os roteadores. Este por sua vez transfere essas decisões, a tabela de comutação, por meio de uma API proprietária para o *hardware* de encaminhamento, que realiza a comutação dos pacotes ao nível do *hardware*. A única interação de gerência do usuário, no caso o administrador de rede, com o dispositivo,

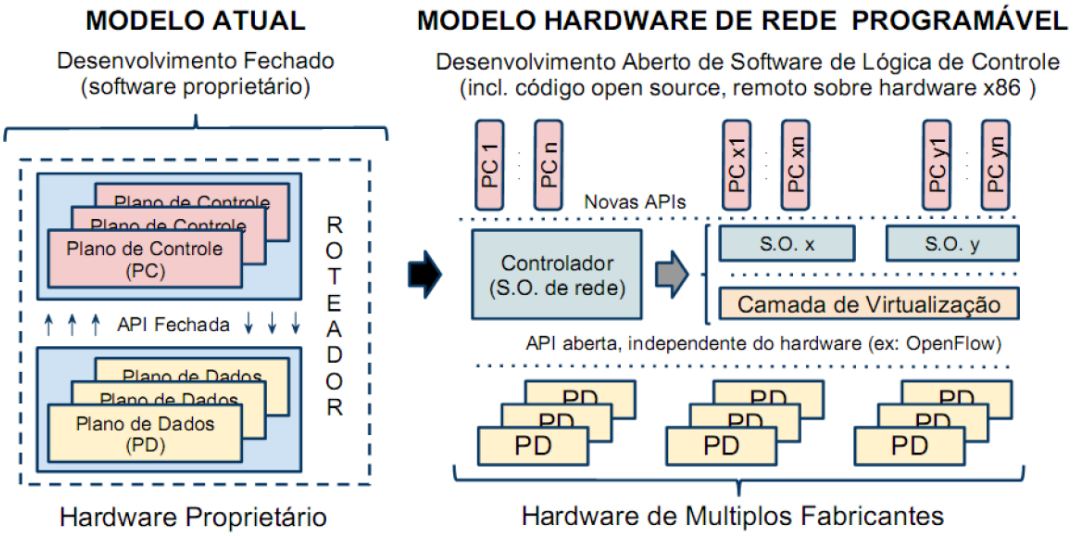


Figura 3.1: Arquiteturas de roteadores: modelo atual e modelo programável [40].

ocorre por meio de interfaces de configuração Web ou SNMP limitando-se ao uso de funcionalidades básicas programadas pelo fabricante.

Sendo a arquitetura atual definida por duas camadas autocontidas, não é necessário que elas sejam fechadas em um mesmo equipamento. A arquitetura das redes SDN (Figura 3.1) subdivide essas camadas de forma que seja possível programar remotamente o dispositivo de rede, permitindo que a camada de controle possa ser movida para um servidor dedicado e com alta capacidade de processamento.

Desse modo, mantém-se o alto desempenho no encaminhamento dos pacotes em hardware aliado à flexibilidade de se inserir, remover fluxo de dados por meio de aplicações em *software* por meio de um protocolo aberto, API, para programação da lógica do equipamento.

3.4 Elementos Programáveis das redes SDN

De maneira mais específica, os elementos de comutação, *switches* e roteadores, exportam uma interface de programação que permite *software* de rede inspecionar, definir e alterar a tabela de roteamento do comutador, isso acontece por exemplo nos comutadores OpenFlow.

O *software* em questão, tende a ser na prática organizado com base em controladores de aplicação geral, os Sistemas Operacionais de Rede, que controlam aplicações especificas para a finalidade de cada rede. Outra possibilidade, e talvez uma das mais interessantes desse paradigma, é a capacidade de utilizar um divisor de visões, como por exemplo o FlowVisor (discutido na Seção 3.5), que permite que as aplicações de rede sejam divididas entre diferentes controladores.

O principio básico das Redes Definidas por Software é a capacidade de programar os elementos de uma rede de computadores. Essa programação é a simples mani-

pulação dos pacotes em um fluxo. Esse fluxo geralmente é definido em função dos recursos oferecidos pela interface de programação.

A maneira como os elementos de rede realizam a operação de encaminhamento dos pacotes é simples. Cada pacote recebido em uma das interfaces do comutador de rede é inspecionado e dele é gerado uma consulta a uma tabela de encaminhamento do comutador. Atualmente nos *switches* Ethernet, essa consulta é baseado no endereço MAC de destino do pacote, em roteadores IP, em um prefixo do endereço IP de destino.

Caso o comutador não encontre esse endereço na tabela de comutação, o pacote é descartado ou segue um comportamento padrão, como por exemplo enviar ele a todas as portas saídas do comutador (*broadcast*). Uma vez identificado o destino do pacote, seja ele encontrado na consulta da tabela de comutação ou definido por um comportamento padrão, o mesmo atravessa as interconexões do comutador para atingir a porta de destino, onde ela entra em uma fila para a transmissão.

Ao passar do tempo, com a evolução das redes de computadores, o processo de consulta (*lookup*) e chaveamento (*switching*) foi amplamente estudado no meio acadêmico, resultando hoje em soluções baseadas usualmente em *hardware* com desempenho suficiente para acompanhar as taxas de transmissão do meio [14].

Nesse sentindo as redes SDN tem a capacidade de controlar o plano de encaminhamento de pacotes por meio de uma interface bem definida. Sem dúvida uma das interfaces mais conhecidas deste paradigma, desde o inicio é o OpenFlow. O principal objetivo do OpenFlow é permitir que se utilize equipamentos de rede comerciais para pesquisas de novos protocolos de rede em paralelo ao tráfego real de produção da rede.

Isso ocorre com os elementos de divisão de recursos, mais explicitados adiante, no qual se define uma interface de programação que permite o desenvolvedor controlar diretamente a tabela de encaminhamento dos comutadores de pacotes presentes na rede. Esse tipo de proposta dá mais credibilidade a pesquisa para a indústria, pois as novas ideias serão validadas com o uso de comutadores de rede reais, de alto processamento e utilizadas em redes reais que em geral contém grande fluxo de dados.

Apesar do OpenFlow ser o foco principal do paradigma SDN, o paradigma não se limita apenas ao OpenFlow e a forma como ele expõe os recursos dos comutadores de rede, nem o exige como elemento essencial. Há diversas outras possibilidades de implementação de uma interface de programação que atenda os objetivos do paradigma.

Outra possibilidade de implementação é o conceito de interface de rede sNICh [16]. O sNICh é uma implementação para ambientes em redes virtualizadas em que a divisão do plano de dados é feita de forma em que se possa dividir as tarefas de encaminhamento entre *host* e interface de rede. Com essa divisão é garantido uma eficiência no encaminhamento entre as máquinas virtuais no mesmo hospedeiro e a rede.

Dessa forma é possível se imaginar uma interface definida para essa arquitetura que possa ser usada para o controle de roteamento por um *software* implementado em um controlador de rede, que se apresente como uma opção para o uso de comutadores de *software* como os *switches* OpenFlow.

Ainda é possível implementar outras propostas para o paradigma SDN que alteram a divisão de tarefas entre o controlador e os *switches*. Isso é apresentado na proposta de arquitetura do DevoFlow [15]. O DevoFlow aborda o argumento que o OpenFlow é muito dependente do controlador de rede SDN.

No OpenFlow existe a necessidade de que todos os fluxos sejam acessíveis para o controlador, isso por sua vez impõe demandas sobre o *hardware* e limitações de desempenho que podem não ser aceitáveis em casos particulares nos quais podem ser definidos por regras simples.

Dessa forma, o DevoFlow reduz o número de casos em que o controlador precisa ser acionado, aumentando a eficiência. Outras propostas de solução podem ser aplicadas ao OpenFlow, basta que se façam regras especificas para cada fluxo que seja identificado pelo controlador.

3.5 Divisores de Recursos

Uma das principais vantagens das Redes Definidas por Software são as diversas formas de se dividir os recursos das redes. A possibilidade de se associar todo um processamento complexo, definido por *software*, a pacotes que se encaixem em um determinado padrão abriu a possibilidade de se associar diversos comportamentos em uma mesma rede [14].

Como dito no capítulo anterior, essa possibilidade de divisão dos comportamentos fez com que se tornasse viável manter um comportamento tradicional para fluxos reais de produção e um outro comportamento diferente para fluxos de pesquisa. O primeiro trata-se do tráfego real de produção de uma rede, o comportamento pode seguir as orientações do fabricante do *hardware*, ou mesmo dos roteadores legados da rede. O segundo trata-se do tráfego de pesquisa de novas soluções de rede, a possibilidade de realizar pesquisas em redes reais, não simuladas. É de grande valia para pesquisas, para novas soluções em rede. A proposta do OpenFlow [42] parte deste principio.

A capacidade de dividir a rede em fatias já ocorre na arquitetura atual da Internet com a virtualização de redes por meio do uso de VLANs (redes locais virtuais), em que existe uma cabeçalho especial no pacote que é usado para definir a qual rede virtual ele pertence. O uso de VLANs no entanto tem suas limitações definidas em sua tecnologia de rede 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.

Considerando essa analogia do uso de VLANs com o contexto atual das redes SDN é possível estender essa divisão de uma 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.

Assim com a extensão dessa divisão, é permitido que diferentes tipos de pesquisas possam ser colocadas em operação de forma paralela, na mesma rede física, junto com o tráfego de produção, bastando que sejam colocados elementos de divisão de visão na rede, efetuando a divisão dos recursos entre os diferente controladores de rede.

Isso evita que a rede fique restrita a um único controlador abrindo espaço para que pesquisadores diferentes desenvolvam novos protocolos de rede usando tecnologias diferentes em um mesmo ambiente de testes. Além disso garante a não interferência entre as diversas aplicações e permite a utilização de diferentes interfaces SDN na rede. A virtualização pode ser implementada sobre uma rede física em diversos níveis da mesma maneira que ocorre na virtualização de máquinas físicas.

A primeira solução para a divisão de recursos nas redes SDN foi a divisão direta dos recursos OpenFlow da rede física pelo FlowVisor [21]. O FlowVisor age como um controlador de rede que tem a responsabilidade de dividir o espaço de endereçamento disponível em uma rede OpenFlow. A diferenciação dessas fatias só é possível graças as rotulações dos pacotes definidas nos cabeçalhos do pacote. O comportamento dos elementos de comutação no paradigma SDN se baseia na consulta desse campos.

Essa prática nada mais é do que uma extensão do principio de encaminhamento do tráfego da Internet atual, em que campos do endereço determinam direções no grafo das redes de computadores. No FlowVisor esses diversos campos identificam os fluxos OpenFlow(Figura 3.2) em que a definição dessas fatias constituem regras, as quais definem as políticas de processamento.

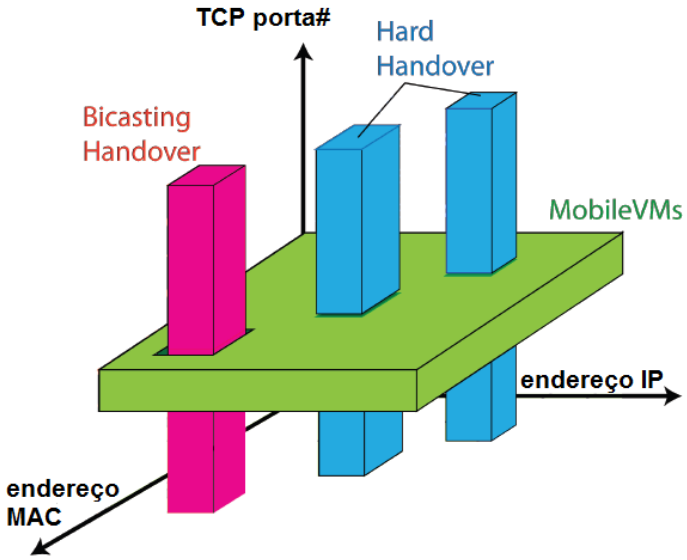


Figura 3.2: Identificação dos fluxos OpenFlow pelo FlowVisor [20].

A Figura 3.2 mostra a definição das fatias com base nos três atributos de fluxo, o endereço IP, MAC e a porta TCP. Na figura o domínio *bicasting handover* define o conjunto que corresponde a faixas do eixo de endereços IP e MAC. O domínio do *mobileVMs* engloba todos os pacotes de todos os endereços IP e MAC mas com uma faixa especifica de número de portas. Já o *hard handover* corresponde a faixas do conjunto de endereços IP e MAC com faixas do número de portas. Em outras palavras o *bicasting handover* observa os eixos que correspondem os endereços IP e MAC. O *mobileVMs* observa apenas o eixo das portas TCP. E finalmente o *hard handover* observa os três eixos do fluxo do gráfico.

O FlowVisor se coloca entre os diversos elementos da rede. O comandos dos controladores são analisados pelo FlowVisor para se certificar que as regras geradas não excedam a definição do domínio do devido controlador. Isso serve para que um pesquisador não se intrometa nas regras definidas por outros pesquisadores da rede em outros fluxos da rede. Caso aconteça uma exceção o FlowVisor reescreve os padrões utilizados com a definição do domínio.

Mensagens enviadas pelos comutadores OpenFlow são analisadas e direcionadas para o controlador apropriado em função do seu domínio. A Figura 3.3 apresenta esse processo pelas duas vias de ida e volta. O comando de inserção de uma regra (seta continua) originado em uma aplicação do domínio ”A” atravessa o FlowVisor, que por sua vez, traduz a regra para garantir que ela se aplique apenas ao domínio de ”A”. Um pacote enviado por um *switch* OpenFlow para a sua aplicação (seta pontilhada) é processado pelo FlowVisor para a identificação do controlador responsável daquela faixa de domínio.

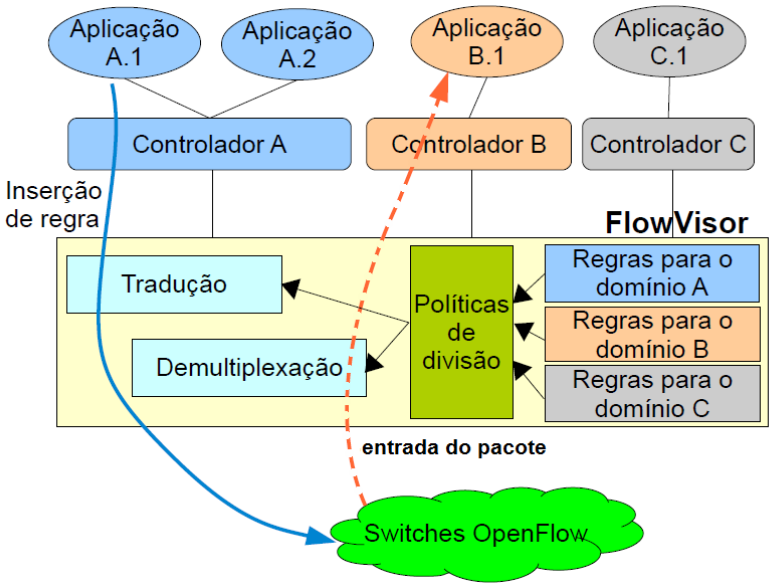


Figura 3.3: Fluxo de comandos do FlowVisor [14].

Ainda é possível criar uma topologia em que seja feita uma construção de hierarquias de FlowVisors cada um dividindo uma seção de um domínio definido por uma instância de um nível anterior.

O FlowVisor é apenas uma das maneiras possíveis de se dividir os recursos em uma Rede Definida por Software. Também é possível adotar um controlador para dividir os recursos para outros controladores na rede. O conceito de virtualização também pode ser aplicado aos elementos de rede visíveis para as aplicações.

3.6 Controlador SDN

Sendo definida uma interface de programação dos comutadores de rede é necessário desenvolver uma aplicação que utilize essa interface para controlar cada *switch* separadamente. Esse desenvolvimento trás consigo limitações associadas ao *hardware* de cada equipamento. Este desenvolvimento exige que o programador lide com tarefas de baixo nível no desenvolvimento de um *software* diretamente ligado a um elemento de *hardware* ocasionando erros e aumentando a complexidade. Além disso novos desenvolvimentos exigem que todas as funcionalidades de baixo nível sejam reimplementadas.

Assim, faz-se necessário um novo nível na arquitetura das redes SDN, um nível que concentre as tarefas de manipulação dos elementos de rede oferecendo uma abstração de mais alto nível para o desenvolvedor, fazendo uma analogia clara aos sistemas operacionais de computadores pessoais (PCs). Essa é a definição da natureza do paradigma de SDN.

Esse componente, chamado de ”sistema operacional de rede”, ou melhor chamado de Controlador SDN, pode concentrar a comunicação com todos os elementos programáveis da rede oferecendo uma visão unificada da rede. Dessa forma é possível desenvolver programas que além de ter uma visão centralizada da rede, com análises detalhadas, é possível também implementar novas funcionalidades para chegar a decisões operacionais de como o sistema deve operar [6], obtendo uma melhor gerência da rede.

Essa visão unificada da rede não necessariamente precisa ser centralizada. A analogia que fazemos a sistemas operacionais distribuídos podem ser implementado também nos controladores. A implementação pode der desenvolvida de forma distribuída, seja pela divisão dos elementos de diferentes SA’s ou por um controlador realmente desenvolvido de forma distribuída com algoritmos que sejam capazes de manter uma visão consistente entre suas partes.

Foram desenvolvidos diversos controladores para o paradigma SDN. Muitos dos quais apresentam ambientes de tempo real de execução que oferecem uma interface imperativa para programação da rede. O que determina em grande parte o estilo de desenvolvimento e as funcionalidades que o controlador oferece é a linguagem de programação em que ele foi desenvolvido seja ele em C, Java, Python, Ruby entre outros.

Há, no entanto, outros controladores que ultrapassam a noção de uma interface de programação propriamente dito, e utilizam abstrações com um ambiente de programação funcional ou declarativo. Nesse caso as implementações do controlador se prestam as funcionalidades de detecção do conflitos ou depuração da rede.

Muitos dos controladores já desenvolvidos não se preocupam com requisitos de escalabilidade e disponibilidade da rede optando assim por estruturas centralizadas por pura simplicidade. No entanto, há desenvolvimentos voltados para a implementação de grandes sistema de redes nos quais utilizam-se diferentes formas de distribuição de controle para garantir requisitos como escalabilidade e disponibilidade do sistema. Essa seção apresenta alguns controladores SDNs desenvolvidos, sendo apresentado suas características principais. A Figura 3.4 apresenta uma tabela comparativa dos principais controladores SDN.

Nome	Linguagem	Plataforma	Característica
NOX	C++, Python	Linux	Controlador de referência OpenFlow
POX	C++, Python	Windows, Mac, Linux	Evolução do NOX
Maestro	Java	Windows, Mac, Linux	Explora o paralelismo
Beacon	Java	Windows, Mac, Linux, Android	Multiplataforma, Multithreaded
Floodlight	Java	Windows, Mac, Linux	Pode-se integrar a redes não OpenFlow
Frenetic	Funcional/Declarativa Própria	Linux	Programa a rede como um todo
Onix	C++, Python, Java	Windows, Mac, Linux	Gerência redes de grande porte
SNAC	C++	Linux	Monitoramento de redes OpenFlow; Interface Web
Trema	C, Ruby	Linux	Script; Emulador

Figura 3.4: Principais controladores SDN.

3.6.1 NOX

O NOX [18] é o controlador de referência que acompanha o OpenFlow, contém APIs desenvolvidas em C++ e Python. O NOX serve como uma camada de abstração criando as aplicações e serviços que gerenciam as entradas de fluxo nos *switches* OpenFlow. Seu funcionamento dar-se da seguinte maneira: se o pacote não possui nenhuma entrada da tabela de fluxos o mesmo é encaminhado ao NOX, normalmente o primeiro pacote de cada fluxo é enviado ao controlador, este por sua vez checa e procura uma regra determinando a política a ser aplicada.

Dependendo da aplicação pode-se determinar uma regra padrão como por exemplo: optar por enviar todos os pacotes de determinados fluxos ao controlador. Uma aplicação de DHCP seria um bom exemplo já que o fluxo desta aplicação não precisa ser adicionado na tabela.

O NOX oferece uma interface de programação na qual para se desenvolver ou implementar uma nova solução de rede dois conceitos são utilizados, o **componente** e o **evento**. O **componente** é um encapsulamento de funcionalidades que são carregadas com o NOX, um *firewall* por exemplo pode ser implementado como um componente e agregado ao NOX. Os componentes podem ser escritos em C++ ou Python. Um **evento** é uma ação realizada sobre um determinado fluxo. As aplicações implementadas no NOX utilizam um conjunto de manipuladores que são registados para serem executados quando um evento específico acontece [18].

Os eventos são gerenciados por mensagens OpenFlow por meio de pacotes recebidos de um *switch* na rede. Quando um *switch* envia um pacote para o controlador NOX por meio do protocolo OpenFlow, este é analisado pelo controlador e dispara um evento de acordo com a política determinada pela aplicação.

A grande maioria dos projetos de pesquisa na área de Redes Definias por Software são baseados no controlador NOX, que é um controlador simples para redes que provê primitivas para o gerenciamento de eventos bem como as funções para a comunicação com os *switches* [18]. Entretanto seu desenvolvimento foi descontinuado, dando lugar ao POX.

3.6.2 POX

O POX é o irmão mais novo do NOX. Sua essência é uma plataforma para o desenvolvimento e a prototipagem rápida de aplicações de *software* para redes SDN usando Python.

O POX está em constante desenvolvimento e têm o objetivo de substituir o NOX. Sua arquitetura, baseada no NOX, é mais estável e sua interface é mais elegante resultando em um controlador mais moderno e simples.

Embora o POX seja mais estável, o NOX ainda permanece como um ambiente adequado para implementações que tenha demandas mais elevadas em termos de desempenho. No entanto os desenvolvedores do POX acreditam que este seja mais adequado para substituir o NOX nos casos em que Python é utilizado. Este controlador será discutido em destaque no capítulo 5.

3.6.3 Maestro

O Maestro [4] é outro controlador de rede do paradigma SDN para *switches* OpenFlow. O Maestro foi desenvolvido em JAVA e tem como objetivo orquestrar aplicações de controle por meio de interfaces que acessam e modificam o estado da rede coordenando suas interações.

O modelo de programação do Maestro oferece: interfaces para a introdução de novas funções de controle personalizadas; interfaces para a manutenção do estado da rede; e componentes de controle para a especificação da sequência de execução dos componentes da rede.

O Maestro explora o paralelismo de uma máquina ao máximo para obter um maior desempenho do sistema de transferência entre o controlador e o *switch*, visto que o controlador pode ser um gargalo devido a responsabilidade da criação de cada fluxo de dados entre os *switches*. O Maestro alivia o esforço que os programadores têm para realizar a paralelização do sistema operacional de rede, trazendo assim um maior desempenho no controle da rede e diminuindo esse gargalo.

3.6.4 Beacon

O Beacon [12] é mais um controlador baseado em JAVA que suporta tanto operações de controle baseadas em eventos quanto operações baseadas em *threads*.

O projeto vem sendo desenvolvido desde agosto de 2011 na Universidade de Stanford. Suas principais características são:

- tem uma implementação estável, consegue gerenciar 100 *switches* virtuais e 20 físicos em redes experimentais por meses sem inatividade;

- é multi-plataforma, por ser desenvolvido em JAVA;
- é *software* livre baseado em licença GPL v.2;
- tem um rápido desempenho por ser *multithreaded*;

O Beacon contém uma estrutura que permite que o controlador seja atualizado em tempo de execução sem interromper outras atividades de encaminhamento de pacotes.

3.6.5 Floodlight

O Floodlight [24] é um projeto que se originou do controlador Beacon e agora é apoiado pela *Open Networking Foundation (ONF)* e também pela *Big Switch Networks*[1], que produz controladores comerciais que suportam o Floodlight.

O Floodlight é totalmente baseado na linguagem JAVA com seu núcleo e módulos principais escritos em JAVA, recentemente foi adicionando o Jython, o que permite o desenvolvimento na linguagem Python. Ele é distribuído segundo a licença Apache permitindo que o administrador de rede possa utilizá-lo para praticamente qualquer finalidade. Além disso o projeto está sendo desenvolvido por uma comunidade aberta na qual todos podem participar. Sua documentação, status sobre o projeto, roteiro e erros estão todos disponíveis.

Sua arquitetura é formada por módulos que exportam serviços. Dessa forma toda a comunicação entre os módulos é feita por meio de serviços. Sua interface permite identificar e descobrir o status e a topologia de uma rede automaticamente. Outra característica importante do Floodlight é que com ele pode se integrar redes não OpenFlow e também é compatível com a ferramenta de simulação MiniNet.

3.6.6 Frenetic

O Frenetic [25] é um sistema de rede baseado em linguagem funcional desenvolvido para operar em redes OpenFlow. Em geral os controladores especificam a política de encaminhamento nos *switches* OpenFlow uma de cada vez. O Ele permite que o operador da rede, ao invés de configurar manualmente cada *switch* da rede, programe a rede como um todo.

O Frenetic é implementado sobre o NOX, foi projetado para resolver os problemas de programação com o OpenFlow utilizando o controlador NOX. Além disso introduz abstrações funcionais para permitir aplicações modulares e a composição dessas aplicações.

O Frenetic é composto de duas sublinguagens integradas, sendo uma linguagem do tipo declarativa para consultas de classificação e agregação de tráfego na rede, e outra linguagem funcional para descrever as políticas de encaminhamento de pacotes em alto nível para toda a rede.

[1] `http://www.bigswitch.com`

3.6.7 Onix

O Onix [22] é um controlador que tem o objetivo de gerenciar redes de grande porte de maneira distribuída. O Onix é um sistema operacional distribuído de rede que provê abstrações para particionar e distribuir o estado da rede em múltiplos controladores distribuídos garantindo problemas de escalabilidade e disponibilidade que podem surgir quando um controlador centralizado é utilizado.

A rede é mantida por meio de uma estrutura de dados que representa um grafo com todos os elementos da rede física. É por meio desta estrutura que a visão global do sistema é constituída sendo a base do modelo de distribuição do Onix. As interfaces de controle da rede são implementadas por meio de operações de leitura e atualização do estado dessa estrutura de dados garantindo assim a escalabilidade e disponibilidade da rede por meio do particionamento e da replicação da estrutura entre diversos os servidores do sistema.

O resultado do sistema mostra um bom desempenho em relação a escalabilidade da rede apresentada atendendo os requisitos do projeto em questão. O Onix é um produto fechado e portanto é um controlador proprietário.

3.6.8 SNAC

O SNAC [41], ou *Simple Network Access Control* é um controlador de monitoramento de redes OpenFlow. Sua interface é baseada em uma ferramenta web que mostra os status e as características da rede. Ele incorpora uma linguagem de definição de política flexível com uma interface de fácil utilização para configurar dispositivos e eventos da rede.

Todas as suas funcionalidades são agrupadas em categorias disponíveis na interface web na qual é possível se monitorar as tarefas do dia-a-dia das políticas de controle da rede. É possível ver os status do sistema e gerenciar as políticas de segurança da rede permitindo um acesso rápido a qualquer página dentro do sistema. A maioria das páginas são atualizadas automaticamente para manter as informações exibidas de acordo com o estado interno das políticas de controle da rede.

Apesar do SNAC ter um bom controle de gerenciamento e monitoramento gráfico de uma rede OpenFlow ele não é um ambiente de programação genérico como os outros controladores apresentados.

3.6.9 Trema

O Trema [45] é uma aplicação OpenFlow para o desenvolvimento de controladores. Ele utiliza as linguagens de programação Ruby e C para que pesquisadores desenvolvam e criem de maneira fácil seus próprios controladores OpenFlow. O Trema não tem como objetivo fornecer uma implementação específica como controlador, seu objetivo é ajudar desenvolvedores a criarem implementações por meio de *scripts* simples escritos em Ruby ou C.

O Trema ainda possui um emulador de rede OpenFlow próprio para a execução de testes. Não é necessário *switches* e *hosts* físicos, nem ambientes virtuais como o MiniNet, para testar aplicações para o controlador desenvolvido.

3.7 Aplicações

Seguindo o presente contexto, considerando agora os controladores do paradigma SDN como sistemas operacionais de rede, o *software* desenvolvido para criar novas funcionalidade pode ser visto como uma aplicação que é executada sobre uma rede física.

Dessa forma, observando agora a rede como um sistema unificado e gerenciado por um sistema operacional, pode-se, por exemplo, implementar soluções de roteamento e de encaminhamento especialmente desenhadas para ambientes particulares, controlando as interações entre os diversos comutadores da rede.

Essa tamanha flexibilidade que este paradigma oferece para se estruturar sistemas de rede é útil em praticamente todas as áreas de aplicação das redes de computadores. Sua estrutura lógica centralizada permite o desenvolvimento de novas funcionalidades de maneira fácil e bem diversificada, abrangendo assim uma grande diversidade de ambientes de rede. Considerando que as redes SDN definem essa nova forma de estrutura, podemos considerar que ela seja aplicável a qualquer tipo de ambiente no cenário de redes de computadores nos beneficiando de uma melhor organização das funcionalidades oferecidas em torno de uma visão lógica completa da rede.

Essa seção aborda os principais contextos em que podemos nos beneficiar com essa estrutura de rede, apresentando exemplos e maneiras de como podemos utilizar essas aplicações em diversos casos.

3.7.1 Controle de Acesso

Uma das principais características do paradigma SDN consiste no tratamento dos pacotes por meio de padrões que identificam seus fluxos. Nesse sentido o desenvolvimento de aplicações que gerenciam o controle de acesso é bastante trivial nesse paradigma.

A visão lógica global que o paradigma SDN nos trás permite a configuração de políticas de controle de acesso que sejam desenvolvidas com base em informações abrangentes, diferentemente do que ocorre nas redes atuais, em que o controle de acesso é definido, por exemplo, seguindo um política de *firewall* em um enlace especifico da rede.

Além disso essa visão lógica global da rede nos permite implementar regras que levem em conta não apenas tipos de protocolo e pontos de origem e destino dos pacotes, mas a relação entre dois pontos distintos da rede e a identidade do usuário de forma simples. A facilidade de se definir qual rota adotar em cada fluxo nos permite criar filtros especiais para o controle de acesso de determinados tipos de tráfego em determinados elementos da rede.

3.7.2 Gerenciamento de Redes

Como o estado da rede pode agora ser centralizada em um controlador, as decisões sobre a forma de fluxos de determinadas rotas podem ser mudadas dinami-

camente com base na preferência do operador e gerenciada centralmente por meio deste controlador. A visão global da rede simplificam as ações de configuração e gerência da rede.

Nas redes de computadores atuais o gerenciamento é realizado por configurações de baixo nível que depende do conhecimento da rede, semelhante ao desenvolvimento de aplicações sem ajuda do sistema operacional. Podemos exemplificar o controle de acesso aos usuários de uma rede utilizando uma ACL (*Access Control List*) que consiste em uma lista de controle de acesso que necessita do conhecimento do endereço IP do usuário, que é um parâmetro de baixo nível independentemente da rede.

Nesse sentido, analogamente ao desenvolvimento de aplicações utilizando um sistema operacional, com o paradigma SDN e a utilização de um sistema operacional de rede torna-se mais fácil o gerenciamento das redes de computadores por meio de interfaces de alto nível para controlar e observar a rede, semelhante às interfaces de leitura e escrita em diversos recursos oferecidos por um sistema de computador.

Dessa forma os sistemas operacionais de rede fornecem interfaces genéricas de programação que permitem o desenvolvimento de aplicações de gerenciamento das redes, centralizando o gerenciamento da rede e resolvendo os problemas advindos da escalabilidade, diferentemente do que ocorre nas redes de computadores atuais.

3.7.3 Gerenciamento de Energia

O gerenciamento de energia também é um dos pontos fortes que podemos obter de maneira fácil com o paradigma SDN. Atualmente as soluções que conservam e gerenciam energia em sistemas da computação vem crescendo bastante. Em redes de grande porte, onde os recursos de rede consomem uma parcela significativa do gasto de energia, reduzir o consumo de energia consumida pelo meio de comunicação se torna uma possibilidade interessante.

Em determinados períodos do dia alguns elementos da rede ficam subutilizados devido ao baixo fluxo de dados na rede. O desligamento desses dispositivos de rede tornam-se viáveis devido a baixa taxa de transmissão dos mesmos provocando a redução do consumo de energia nessa rede.

Isso é possível devido a característica da visão global da rede que é provida pelo paradigma SDN. Uma vez tendo essa visão global da rede torna-se fácil a identificação desses elementos ociosos e mesmo a redefinição de rotas a fim de desviar o tráfego de elementos passíveis de desligamento. Além disso o uso do controle de rotas de decisão permite a implementação de pontos de controle que podem interceptar tráfego ”ruidoso” na rede, evitando que pacotes de menor importância atinjam o elementos subutilizados passíveis de deligamento, evitando que os mesmo sejam ativados desnecessariamente.

3.7.4 Comutador Virtual Distribuído

Virtualização de redes hoje é bastante utilizada em redes corporativas e em redes de grande porte. Usualmente são utilizados *switches* implementados por *software*

instalado em cada máquina física a fim de prover a conectividade exigida por cada máquina virtual.

A facilidade de mover as máquinas virtuais entre os hospedeiros físicos torna o processo de configuração e monitoração dessas máquinas um desafio que complica a gerência dos *switches* virtuais [14]. Uma maneira possível para reduzir essa complexidade é o uso do paradigma SDN. A ideia é oferecer uma visão de um *switch* único, virtual a todas as máquinas virtuais da rede de maneira que ofereça a elas a impressão de estarem ligadas a um único *switch*. Dessa forma a migração das máquinas virtuais entre os hospedeiros físicos seria simples e teria um impacto minimo nas configurações da rede virtual pois elas continuariam ligadas a mesma porta do "único" *switch* visível para toda a rede.

Essa abstração pode ser implementada de maneira muito simples com um controlador de rede, por exemplo o NOX, já que ele poderia inserir automaticamente as regras de encaminhamento de pacotes entre as portas dos diversos *switches* criando o efeito desejado. Com o uso de *switches* físicos OpenFlow também é possível fazer com que máquinas físicas, não virtualizadas, possam fazer parte desta rede.

3.7.5 Redes Domésticas

As redes domiciliares vem sendo consideradas um dos grandes desafios para a indústria de redes de computadores, de modo especial no que se refere ao uso de ferramentas eficientes para sua gerência e configuração [13]. Esse investimento exige tanto soluções que resolvam a interação com o usuário quanto soluções eficiente para automação de configuração e implementação de políticas de segurança para o lar.

Uma maneira de utilizar aplicações SDNs nesse contexto seria o uso de roteadores domésticos OpenFlow com o provedor de acesso tendo a responsabilidade de poder controlar a rede por meio de um controlador SDN. Dessa maneira o controlador SDN seria capaz de programar, alterando suas políticas e regras de controle em cada roteador doméstico. Com essa possibilidade o provedor de acesso poderia gerir as regras de acesso apropriadas para cada tipo de usuário, além é claro de ter uma visão global da rede como um todo, gerenciando elementos subutilizados e priorizando elementos com mais tráfego na rede.

Outra possibilidade que aborda esse contexto são as pesquisas que gerenciam o tráfego licito e ilícito nas redes de computadores [5, 23]. Se os provedores de acesso puderem obter a visão global de todos os seus usuários na rede ele é capaz de indicar quais padrões de tráfego são as ações de atacantes (*crackers*) ou código malicioso melhorando assim a segurança nas redes de computadores.

Claro que essas possibilidades exigem um forte investimento na infraestrutura da rede, visto que o protocolo OpenFlow exige a substituição dos equipamentos de rede, além, é claro, de ser necessário um canal seguro (SSL) entre os roteadores OpenFlow e o controlador SDN, aumentando ainda mais o gasto em infraestrutura. No entanto essas possibilidades, embora não sejam viáveis (ainda), podem ser consideradas o futuro das redes de computadores.

Ainda há outros tipos de soluções domiciliares que podemos usar com o paradigma SDN, podemos considerar a implementação de um controlador de rede interno

à rede domiciliar. Esse tipo de solução se torna interessante quando observamos o aumento da complexidade das rede domésticas nos dias atuais. As redes domiciliares contam hoje com diversos equipamentos que funcionam com a Internet, desde celulares, notebooks a eletrodomésticos, sejam eles conectados por enlaces com fio ou sem fio.

Com um controlador doméstico um usuário poderia observar todo o tráfego interno da rede doméstica, podendo gerenciar o uso de banda em cada equipamento de rede, configurando os serviços da rede tais como impressoras e novos dispositivos de acesso, além é claro de observar e detectar potenciais máquinas infectadas por *malware*.

3.7.6 Roteador Expansível de Alta Capacidade

As redes de computadores mais robustas possuem em suas bordas elevadas demandas de conectividade. Com isso faz-se necessário o uso de roteadores de grande porte para que suportem essas demandas, entretanto tais roteadores possuem um custo bastante elevado.

Com o paradigma SDN, é possível substituir esse roteador de grande porte por uma banco de *switches* de médio porte desde que estes sejam dotados de interfaces OpenFlow. Dessa forma, semelhante ao que ocorre no caso do comutador virtual distribuído, um controlador SDN pode preencher as tabelas de roteamento de cada *switch* com as rotas necessárias para interligar o tráfego entre duas portas de *switches* diferentes que sejam vistas como parte de um mesmo roteador [14].

Todas as informações de roteamento seriam distribuídas entre os *switches* pelo controlador em função das demandas identificadas. Os algoritmos de roteamento do roteador poderiam também ser executados pelos controladores, que se encarregaria de manter a interface com os demais roteadores externos de outras redes.

3.7.7 Redes de Grande Porte

Nas redes de grande porte em que existem aplicações de vários usuários, um problema sempre recorrente é o isolamento do tráfego entre essas diversas aplicações. O problema hoje é resolvido com a utilização de VLANs individuais para cada cliente. Dessa forma o tráfego gerado pelas aplicações de cada usuário é transportado apenas dentro da sua VLAN, garantindo o isolamento necessário para fins de segurança e privacidade.

Todavia, os recursos de VLANs são limitadas pelo tamanho dos campos utilizados para sua identificação, tornando inviável sua utilização em redes que possuem um grande numero de usuários. Outro problema é que as interfaces de configuração e gerência desses tipos de rede são bastante complexas dificultando bastante o controle da rede.

Uma maneira de solucionar esse problema é a utilização de uma abstração de um comutador virtual distribuído por meio de um controlador SDN. Dessa forma é fornecido a cada usuário um *switch* virtual que irá interligar suas máquinas independentemente da localização física na rede. O controlador fica então responsável

por definir as rotas entre as portas dos diversos elementos que compõe o comutador virtual garantindo a alocação de recursos suficientes em cada caso. Pelo fato de cada usuário estar conectado a um *switch* virtual diferente, o isolamento do tráfego é garantido diretamente pela implementação da rede SDN.

Outra consequência desse tipo de isolamento de tráfego além de eficaz e ilimitado é que existe a possibilidade de se oferecer a cada usuário o controle direto de sua rede. As configurações que estejam disponíveis para o *switch* virtual podem ser controladas pelo o usuário já que elas não irão influenciar as outras redes por estarem isoladas das demais. Soluções como balanceamento de carga, qualidade de serviço e redundância de rotas podem ser colocadas sob o controle do usuário desde que o controlador de rede garanta esse tipo de implementação no *switch* virtual.

Capítulo 4

OpenFlow

Observamos nos capítulos anteriores que as redes de computadores atuais já não atendem requisitos atuais da Internet e para que esses requisitos sejam atendidos foram apresentadas algumas propostas para a Internet do Futuro. Uma das propostas mais promissoras que vêm ganhando destaque no meio acadêmico são as Redes Definidas por Software (*Software Defined Networks*, (SDN)).

As redes SDN constituem um novo paradigma para o desenvolvimento das redes de computadores abrindo novas perspectivas em ambientes de controle lógico da rede de maneira que possam ser desenvolvidas de forma simples e livres dos limites da arquitetura atual da rede.

Nesse contexto as redes SDN trazem consigo a capacidade de controlar o plano de encaminhamento dos pacotes da rede, no entanto para que isso ocorra é necessário uma interface que seja simples para realizar a comunicação entre os elementos de comutação da rede e o controlador da rede. De fato a interface que foi associada desde o inicio ao paradigma SDN, sendo um dos elementos mais motivadores para sua criação, foi o **OpenFlow**.

O OpenFlow [19] foi proposto pela Universidade de Stanford, seu objetivo inicial é atender à demanda de validação de novas propostas de arquitetura e protocolos de rede sobre equipamentos comerciais. Dessa forma é possível implementar uma tecnologia capaz de promover a inovação no núcleo da rede, por meio da execução de redes de teste em paralelo com as redes de produção.

A proposta do OpenFlow promove a criação de redes SDN, usando elementos comuns de rede, tais como *switches*, roteadores, pontos de acesso ou, até mesmo, computadores pessoais[1] [19].

Este capítulo apresenta as características do padrão OpenFlow, bem como os elementos necessários para sua configuração e aplicação em uma Rede Definida por Software. Também é apresentado o comutador OpenFlow, suas características, seu funcionamento e como é utilizado entre os comutadores OpenFlow e um controlador SDN.

[1]Uma das maneiras de se implementar um comutador OpenFlow em computadores pessoais é com o Open vSwitch [38], seu funcionamento se baseia em um módulo do Linux que implementa o encaminhamento programável de pacotes diretamente no *kernel* do sistema, seu plano de controle é acessado a partir do espaço de usuário.

4.1 Pradrão OpenFlow

O OpenFlow é um padrão aberto para o paradigma de redes SDN, ele consiste de uma interface de programação que permite ao desenvolvedor controlar diretamente os elementos de encaminhamento de pacotes presentes no dispositivo.

A pesquisa na área de redes de computadores possui diversos desafios em relação a implementação e a experimentação de novas propostas de rede em ambientes reais. O pesquisador de redes em geral não possui uma rede de teste que apresente desempenho próximo de uma rede real. Para isso foi proposto o OpenFlow que permite que um pesquisador possa experimentar suas propostas em redes reais, sem atrapalhar o fluxo de produção, obtendo uma melhor validação na sua pesquisa para a indústria.

O OpenFlow consiste de um mecanismo que é executado em todos os comutadores da rede para que haja o isolamento do tráfego de produção e o tráfego experimental. Dessa forma os pesquisadores podem reprogramar os comutadores de maneira que não haja interferência no tráfego de produção, ou seja, a rede continuará funcionando da mesma forma.

Outro objetivo da proposta do OpenFlow é permitir que os fabricantes de *hardware* possam inserir as funcionalidades do OpenFlow nos seus comutadores sem a necessidade de expor o projeto desses equipamentos. Um requisito importante nessa abordagem, já que a infraestrutura será modificada, é que tais equipamentos devem possuir um custo baixo e um desempenho semelhante aos já utilizados na rede de produção, de maneira que se torne viável para os administradores de rede a troca desses equipamentos pelos já utilizados.

O projeto do OpenFlow tem por objetivo ser flexível e atender o seguinte requisitos estipulados:

- capacitar o uso em implementação de baixo custo e alto desempenho;
- possibilitar o suporte de uma vasta gama de pesquisas científicas;
- garantir o isolamento do tráfego de produção e o tráfego experimental;
- garantir que não seja necessário a exposição do projeto dos fabricantes em suas plataformas.

Uma das características básicas do padrão OpenFlow é a separação clara entre os planos de **dados** e de **controle** nos elementos de comutação. O plano de **dados** é responsável pelo encaminhamento dos pacotes associando entradas na tabela de encaminhamento de cada comutador, essas regras são definidas pelo padrão e incluem os seguintes itens como tomada de ação:

- encaminhar o pacote para uma porta especifica do dispositivo;
- alterar parte de seus cabeçalhos;
- descartar o pacote;
- encaminhar o pacote para a análise de um controlador de rede.

É necessário deixar claro que o plano de dados pode ser implementado em hardware utilizando elementos comuns a roteadores e *switches* atuais. Em geral essa estrutura se configura em dispositivos dedicados que tenha a interface de configuração OpenFlow, por exemplo, comutadores de uma rede de produção que se queira fazer testes experimentais de pesquisas de rede.

Por outro lado o plano de **controle** fica responsável pelo controle da rede permitindo ao controlador SDN da rede programar as entradas dessa tabela de encaminhamento com padrões que identificam fluxos de interesse e as regras associadas a eles. O elemento responsável pelo plano de controle pode ser um módulo em *software* implementado de forma independente em algum ponto da rede. Esse elemento, denominado controlador SDN, fica responsável por instalar regras e ações no *hardware* de rede.

4.2 Componentes de uma Rede OpenFlow

Para que uma rede programável com OpenFlow exista é necessário que sua arquitetura seja composta por quatro componentes a ela associados. É necessário equipamentos habilitados que tenham a capacidade de alterar suas tabelas de encaminhamento conforme as decisões de um controlador em *software* a eles conectados por de um canal seguro.

Tabela de Fluxos - Cada entrada na tabela de fluxos do *hardware* de rede é composta por regras, ações e controles de estatística. A regra é formada pela definição dos valores de um ou mais campos do cabeçalho do pacote, é por meio dela que é determinado o fluxo. As ações então ficam associadas ao fluxo e vão determinar como os pacotes devem ser processados, para onde vão ser encaminhados ou se serão descartados. Os controles de estatística consistem de contadores utilizados para manter estatísticas de utilização e para remover fluxos inativos ou que não existam mais. Logo, as entradas da tabela de fluxos são interpretadas pelo *hardware* como decisões em *cache* do plano de controle em *software*, sendo, portanto, a mínima unidade de informação no plano de dados da rede [40].

Protocolo OpenFlow - É o protocolo para a comunicação entre o comutador OpenFlow e o controlador de rede para que haja a troca de mensagens por um canal seguro. Essas mensagens podem ser simétricas, assíncronas ou ainda iniciadas pelo controlador [42].

Controlador - O controlador, fica responsável por tomar as decisões adicionando e removendo entradas na tabela de fluxos de acordo com uma política de encaminhamento. O controlador é uma camada de abstração da infraestrutura física que tem como objetivo facilitar o desenvolvimento de aplicações e serviços que gerenciam as entradas de fluxo na rede. Como já dito, esse modelo de abstração se assemelha ao outros sistemas de software que provêm a abstração do

hardware como, por exemplo, os sistemas operacionais de computadores pessoais. Esse modelo permite a evolução em paralelo das tecnologias no plano de dados e as inovações na lógica das aplicações de controle.

Canal Seguro - Este elemento não só garante que uma rede SDN não sofra ataques de elementos mal intencionados como garante também que a troca de informações entre os comutadores e controladores da rede sejam confiáveis com baixa taxa de erros. A interface de acesso que o projeto do OpenFlow recomenda é o SSL (*Secure Socket Layer*), no entanto existem outras alternativas como o TCP e o PCAP (*packet capture*) sendo muito úteis em ambiente virtuais e de experimentação por sua simplicidade de utilização.

A Figura 4.1 mostra a arquitetura de uma rede OpenFlow. Os comutadores OpenFlow se comunicam com o controlador por meio do protocolo OpenFlow em um canal seguro. O controlador adiciona ou remove entradas nas tabelas de fluxos dos comutadores de acordo com as aplicações de controle de cada rede virtual.

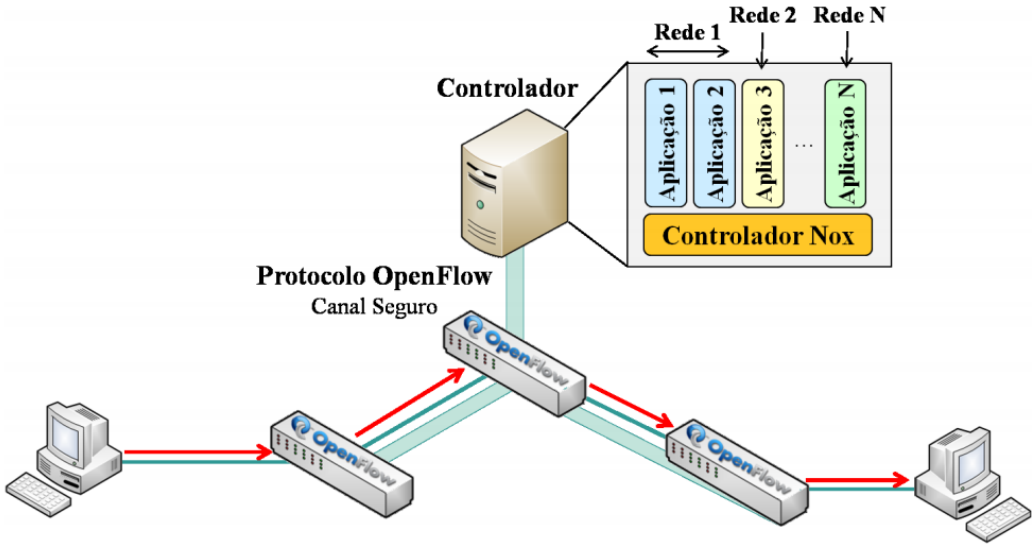


Figura 4.1: Componentes de uma rede OpenFlow [33].

4.3 Protocolo OpenFlow

Uma das vantagens em se utilizar uma arquitetura OpenFlow é a flexibilidade que ela oferece para se programar de forma independente o tratamento de cada fluxo da rede e como ele deve ou não ser encaminhado pela rede. De uma maneira mais prática o OpenFlow determina como um fluxo pode ser definido, quais serão as ações que podem ser realizadas para cada pacote pertencente a este fluxo e qual é o protocolo de comunicação entre o controlador e os comutadores utilizados para realizar as definições de fluxo e ações.

Dessa forma uma entrada na tabela de fluxos de um comutador OpenFlow é formada pela união da definição do fluxo e um conjunto de ações a ele determinadas. Um fluxo é constituído pela definição dos valores de um ou mais campos do cabeçalho do pacote a ser processado pelo dispositivo OpenFlow. Dessa forma os campos do cabeçalho descrevem o fluxo que por sua vez descrevem quais pacotes combinam com aquele fluxo. Esses campos formam uma tupla de doze elementos que reúne características dos protocolos da camada de enlace, de rede, e de transporte segundo o modelo TCP/IP. Cada fluxo contém regras, para a definição dos pacotes, ações e contadores estatísticos a ele associados, como pode ser visto na Figura4.2.

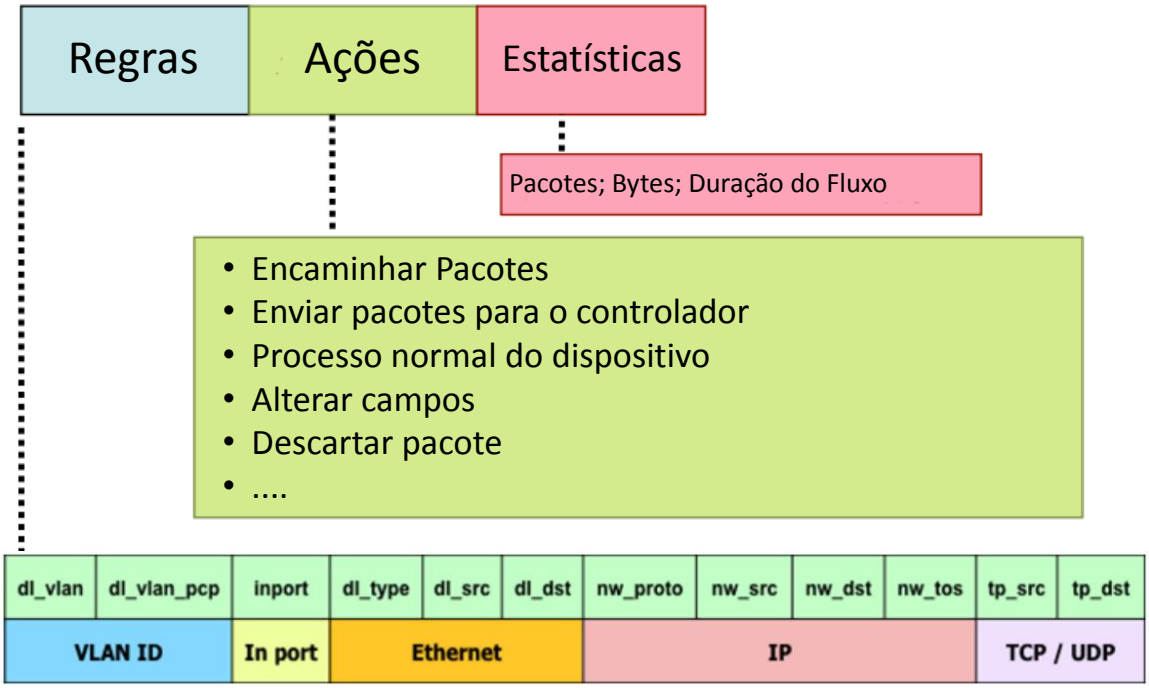


Figura 4.2: Definição de um fluxo na arquitetura OpenFlow [19].

Nesse contexto com a definição de fluxo, é possível observar que as regras de encaminhamento de um pacote não se restringem ao endereço IP ou endereço MAC dos pacotes. O protocolo OpenFlow é mais abrangente de maneira que os elementos formados pelos campos do pacote, as tuplas, podem conter valores exatos ou valores arbitrários, que combinam com qualquer valor que o pacote comparado apresente para o campo. O encaminhamento pode se dar por outras características do pacote, como por exemplo, as portas de origem e destino do protocolo de transporte. Essas características descrevem o funcionamento do protocolo OpenFlow explicitados adiante.

4.4 Controlador OpenFlow

O controlador OpenFlow segue praticamente as mesmas características dos controladores SDN apresentados no capitulo anterior. Ele consiste de um controle centralizado, seja fisicamente ou logicamente, que executa aplicações de controle sobre a rede OpenFlow configurando e gerenciando as tabelas de fluxo dos elementos encaminhadores.

É obviamente necessário que o controlador implemente o protocolo OpenFlow para se comunicar, por meio de mensagens, com os elementos encaminhadores (comutadores com OpenFlow habilitado), enviando os comando para a rede. O controlador age como um sistema operacional de rede agindo sobre o plano de controle como uma interface entre as aplicações de controle da rede e a própria rede.

As mensagens enviadas pelo controlador podem ou não necessitar de resposta do elemento encaminhador e são classificadas de acordo com os seguintes tipos:

Mensagens de características - Geralmente ocorrida após o estabelecimento do canal entre o comutador e o controlador OpenFlow, o controlador solicita ao comutador o pedido de conexão, bem como as caracteristicas especificas e as capacidades do mesmo para que o controlador possa gerenciar melhor a rede.

Mensagens de configurações - O controlador é capaz de definir e consultar parâmetros de configuração no comutador. O comutador responde a solicitação de consulta do controlador.

Mensagens de modificações do estado - São mensagens enviadas pelo controlador para gerenciar os estados dos comutadores. Seu principal objetivo é adicionar ou remover entradas nas tabelas de fluxo dos comutadores, ou seja, adicionar e remover regras.

Mensagens de leitura do estado - São mensagens enviadas pelo controlador para ler os status e coletar as estatísticas (contadores) do controlador.

Mensagens de *packet-out* - Essas mensagens são usadas para enviar pacotes completos para o comutador. Quando não há uma correspondência entre o pacote e alguma entrada na tabela de fluxos do comutador, o pacote então é encaminhado, por completo, para o controlador (*Parcket-In*), que este por sua vez reenvia, utilizando o *Packet-out*, o mesmo pacote com seu conjunto de ações a ele associado.

Mensagens de barreira - São mensagens utilizadas pelo controlador para garantir se a dependências de mensagens foram cumpridas ou para receber notificações de operações concluídas.

4.5 Funcionamento

Definido o protocolo e o controlador OpenFlow o funcionamento de uma rede OpenFlow é programado a partir do plano de controle. Os fluxos podem ser definidos da forma escolhida pelo controlador, como por exemplo, todos os pacotes enviados

pela máquina com IP ”X” para a maquina com IP ”Y” ou, ainda, todos os pacotes pertencentes a VLAN ”Z”. As ações são definidas a partir destes fluxos e incluem:

(a) encaminhar os pacotes pertencentes àquele fluxo para determinada(s) porta(s);

(b) enviar os pacotes para o controlador;

(c) descartar o pacote como medida de segurança;

(d) modificar os campos dos cabeçalhos dos pacotes;

(e) encaminhar o pacote para o processamento normal do dispositivo.

As ações do último tipo **(e)** garantem que o tráfego experimental não interfira no tráfego de produção da rede. Outra maneira de garantir isso é definindo um conjunto de VLANs para fins experimentais. Com essa separação dos tráfegos é possível ter equipamentos híbridos que processam tráfego legado de acordo com os protocolos e as funcionalidades embarcadas do equipamento e, ao mesmo tempo, de maneira isolada, obtém o tráfego baseado nas regras do OpenFlow. As ações associadas aos fluxos formam as entradas nas tabelas de fluxo do comutador OpenFlow.

Além das ações, a arquitetura OpenFlow também possui contadores para o controle de cada fluxo. Cada fluxo contém contadores para o controle de: número de pacotes; números de *bytes* trafegados no fluxo; e duração do fluxo. Esses contadores são implementados para cada entrada da tabela de fluxos e podem ser controlados e acessados pelo controlador por meio do protocolo OpenFlow.

Após definida a entrada a tabela de fluxo, cada pacote que chega a um comutador OpenFlow é comparado com cada entrada nessa tabela e caso um casamento seja encontrado, considera-se que o pacote pertence àquele fluxo e por sua vez irá agir de acordo com uma ação preestabelecida, exemplo: ações do tipo **(a)** ou **(d)**. Caso o casamento não seja encontrado, o pacote é encaminhado por completo para o controlador para ser processado, ação do tipo **(b)**, ou pode ser apagado de acordo com alguma política de segurança preestabelecida, ação do tipo **(c)**. Alternativamente, apenas o cabeçalho é encaminhado ao controlador mantendo o pacote armazenado no *buffer* do *hardware*. Caso o pacote seja encaminhado ao controlador, o controlador analisará o pacote e dependendo da análise da aplicação poderá criar uma nova entrada para aquele fluxo.

Em geral, os pacotes que chegam ao controlador sempre correspondem ao primeiro pacote de um novo fluxo ou, dependendo da aplicação, o controlador pode optar por instalar uma regra no comutador para que todos os pacotes de determinado fluxo sejam enviados para o controlador para serem tratados de maneira individual. Esse último caso corresponde normalmente a pacotes de controle(ICMP, DNS, DHCP) ou protocolos de roteamento (OSPF, BGP) [40].

Essas tabelas de fluxo criam um conjunto de regras que podem servir em diversas possibilidades, muitas das funcionalidades que são implementadas separadamente podem ser agrupadas em um único controlador OpenFlow, como por exemplo, funcionalidades que implementam serviços como NAT, *firewall* e VLANs. Alguns exemplos de possibilidades são apresentadas na Figura 4.3, em que cada linha representa um fluxo com suas respectivas ações a serem tomadas.

Port	VLAN	ETH SRC	ETH DST	IP SRC	IP DST	IP Proto	L4 SRC	L4 DST	Ações
*	*	*	00:4F:...	*	*	*	*	*	Port 4
*	*	*	*	*	*	TCP	*	22	DROP
*	1	*	00:4F:...	*	*	*	*	*	Port 4, 6, 9

Figura 4.3: Exemplo de uma tabela de fluxos de um comutador OpenFlow. O campo representado por um "*" indica que qualquer valor é aceito naquela posição, ou seja é um campo que não importa no reconhecimento do fluxo [19].

Apesar do protocolo OpenFlow possuir um pequeno conjunto de ações simples ele ainda é capaz de prover uma vasta gama de possibilidades para o desenvolvimento de aplicações de rede, isso se deve ao fato que esses conjuntos de ações se bem combinados e desenvolvidos em uma aplicação podem atender aos requisitos específicos de um determinado fluxo. Somando-se isso ao fato de uma rede OpenFlow executar múltiplas aplicações em paralelo atendendo os requisitos específicos de diversos fluxos com um controle lógico central o OpenFlow torna-se uma ótima alternativa para a nova arquitetura da Internet do Futuro.

4.6 Aplicações do OpenFlow

O OpenFlow permite o experimento de novas propostas na área de redes de computadores podendo até mesmo utilizar uma infraestrutura de rede já existente. Dentre as possíveis aplicações que se pode realizar com uma rede OpenFlow, podem ser classificadas as seguintes: [19]

Novos protocolos de roteamento - Como vimos, aplicações diversas podem tratar de fluxos específicos de uma rede OpenFlow. Assim quando um pacote de um novo fluxo chegar a um comutador, ele é encaminhado para o controlador que é responsável por escolher a melhor rota para o pacote seguir na rede a partir de uma política adotada pela aplicação (um novo protocolo de roteamento) correspondente. Após isso, o controlador adiciona as entradas na tabela de fluxos de cada comutador pertencente a rota escolhida e os próximos pacotes desse fluxo que forem encaminhados na rede não necessitarão ser enviados para o controlador. Dessa forma novos protocolos de roteamento podem ser implementados para diversos fluxos específicos na rede de forma que tais protocolos cuidem separadamente de um conjunto de fluxos.

Mobilidade - Uma rede OpenFlow que possui pontos de acesso sem fio permite que clientes móveis utilizem sua infraestrutura para se conectarem a Internet.

Dessa forma mecanismos de *handoff*[2] podem ser executados no controlador com uma simples migração dinâmica das tabelas de fluxo dos comutadores de acordo com a movimentação do cliente, permitindo a redefinição da rota utilizada.

Redes não IP - O comutador OpenFlow analisa arbitrariamente os campos do pacote para definir a qual fluxo ele pertence de acordo com a tabela de fluxos do comutador, permitindo a flexibilidade na definição do fluxo. Dessa forma podem ser testadas, por exemplo, novas formadas de endereçamento e encaminhamento para as redes de computadores sob a arquitetura OpenFlow.

Redes com processamento de pacotes - Existem aplicações OpenFlow que realizam processamento de cada pacote de um fluxo, nesse caso a aplicação implementada no controlador OpenFlow cria uma regra nos comutadores da rede forçando que cada pacote recebido seja enviado ao controlador para ser analisado e processado pela aplicação. Como já mencionado esses casos correspondem, em geral, a pacotes de controle (ICMP, DNS, DHCP) ou protocolos de roteamento (OSPF, BGP).

4.7 O OpenFlow Atualmente

O OpenFlow pode ser visto como uma tecnologia promissora e inovadora que abre uma série de oportunidades de desenvolvimento tecnológico na áreas das redes de computadores. Esse fator potencial da tecnologia OpenFlow têm atraído a atenção da indústria no desenvolvimento de protótipos com suporte ao OpenFlow (HP, NEC, *Extreme*, *Arista*, *Ciena*, *Juniper*, *Cisco*); no suporte dos fornecedores de processadores em silício (*Broadcom*, *Marven*); na criação de empresas como a *Nicira* e a *Big Switch Networks*; e no interesse de operadoras e provedores de serviço, tais como *Deutsche Telekom*, *Docomo*, *Google*, *Facebook* e *Amazon*. Espera-se que à medida que a especificação OpenFlow evolua, mais fabricantes de equipamentos incorporem o padrão em suas soluções comerciais.

O projeto, inicialmente desenvolvido na Universidade de Stanford, continua sendo desenvolvido pela *Open Networking Foundation* (ONF). A versão 1.1 do protocolo OpenFlow foi lançada em 28 de fevereiro de 2011 possuindo ainda algumas limitações em termos do uso do padrão em circuitos ópticos e uma definição de fluxos que englobe protocolos que não fazem parte do modelo TCP/IP. Em fevereiro de 2012 a ONF publicou a versão 1.2 do OpenFlow, melhor especificado mais flexível e com menos restrições.

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 paras as redes SDN com a tecnologia OpenFlow. Em fevereiro de 2012 a *Big Switch Networks* lançou o controlador Floodlight já mencionado anteriormente. Nesta mesma data a HP

[2] *Handoff* é o procedimento empregado em redes sem fio para tratar de uma unidade móvel de uma célula para outra de forma transparente ao utilizador.

anunciou que estava tomando providências para lançar o seus primeiro equipamentos de rede com OpenFlow habilitado. E em abril de 2012 a Google descreveu como uma rede interna da empresa tinha sido reprojetada para uma arquitetura OpenFlow com melhoria de desempenho e eficiência relevantes.

Tudo isso demonstra que a consolidação das tecnologias de equipamentos programáveis em *software* no padrão OpenFlow, ampliou o conceito de redes definidas por *software* envolvendo-o a novos paradigmas de gerência integrada, inovação em protocolos e serviços baseados em recursos de redes virtualizadas ou definidas por *software*.

Capítulo 5

Controlador POX

O principal componente de uma Rede Definida por Software, como já apresentado, é o controlador SDN, também chamado de sistema operacional de rede. O controlador é o que define a natureza do paradigma SDN. É o componente responsável por concentrar a comunicação com todos os elementos programáveis da rede, oferecendo uma visão unificada da rede.

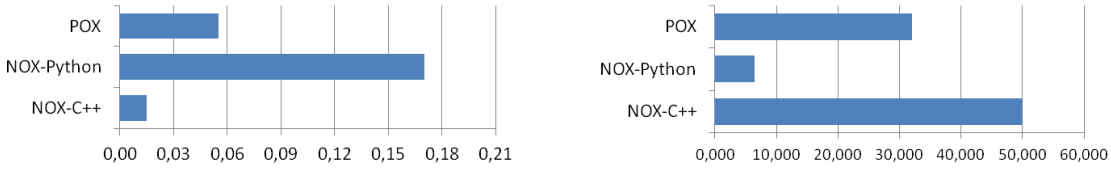
Como vimos, existem diversos controladores para o paradigma SDN. A maneira de como desenvolver aplicações SDN depende muito da linguagem de programação em que o controlador foi desenvolvido bem como sua arquitetura e complexidade. Entre todos os controladores destacados é apresentado o POX, um controlador SDN desenvolvido especificamente para fins de pesquisa e ensino.

Este capítulo apresenta as principais características do POX, os detalhes da sua arquitetura e os elementos principais para a programação de componentes do mesmo de acordo com as especificações encontradas em POX Wiki [1].

5.1 Capacidades do POX

O POX vêm sendo desenvolvido com o objetivo de substituir o NOX em iniciativas de pesquisa e ensino. Ele é considerado o irmão mais novo do NOX, sua essência é uma plataforma simples para o desenvolvimento e prototipagem rápida de software de controle de rede usando o Python.

O objetivo dos desenvolvedores é que ele a venha substituir o NOX nos casos em que o desempenho não seja um requisito crítico. Como pode-se ver na Figura 5.1, observamos que o desempenho do NOX sendo executado em C++ é superior ao POX, porém nos casos em que o NOX é executado em Python o POX é mais eficiente oferecendo um melhor desempenho. Além disso o POX traz consigo uma interface mais moderna e uma pilha SDN mais elegante. Por esses motivos é que o POX é considerado para fins de pesquisa e ensino no desenvolvimento de aplicações SDN.



(a) Teste de latência(ms/fluxo)

(b) Teste de vazão(fluxo/seg)

Figura 5.1: Comparações de desempenho entre o POX e o NOX [34].

5.2 Instalação do POX

Para se executar o POX é necessário ter o Python 2.7 ou superior instalado no seu sistema operacional. O POX é suportado pelos sistemas Windows, Linux e Mac OS, sendo este último o mais recomendado.

O POX é distribuído e hospedado no github[1] em `http://github.com/noxrepo/pox`.

5.3 Executando o POX

Para se iniciar o POX é necessário executar o módulo `pox.py`, para isso fazemos: "`./pox.py`" em um terminal. O POX tem algumas opções de inicialização que podem ser usadas na linha de comando, são elas:

Quadro 5.1: Opções de inicialização do POX.

Opção	Descrição
`--verbose`	Para exibir informações extras, muito útil para a depuração de problemas de inicialização.
`--no-cli`	Serve para que não inicialize a interface interativa de comandos (não é muito útil).
`--no-openflow`	Não inicializará o módulo openflow automaticamente.

Executar o POX sem nenhum módulo não faz muito sentido. O POX funciona a partir de módulos e componentes, no qual o público-alvo são os pesquisadores que queiram desenvolver seus próprios componentes. Dessa forma os pesquisadores tem apenas o trabalho de desenvolver seus componentes e executá-los no POX.

Para executar os componentes basta especificá-los na linha de comando após qualquer uma das opções acima ou apenas após a chamada do módulo `pox.py` dessa forma:

[1]Site de colaboração, análise e gerenciamento de códigos abertos para projetos em desenvolvimento `https://github.com/`

```
./pox.py --no-cli [meu módulo]
```

É possível especificar vários componentes na linha de comando. Em geral alguns componentes não funcionam bem em conjunto e alguns componentes dependem de outros componentes, isso dependerá muito do ambiente de trabalho do pesquisador. Dessa forma é possível especificar vários componentes da seguinte maneira:

```
./pox.py --no-cli [módulo1] [meu módulo2]
```

Os componentes podem ter argumentos próprios, para executá-los com essas opções basta colocar o nome do componente seguido da opção com dois traços, dessa forma:

```
./pox.py --no-cli [módulo1] --[opção] [meu módulo2] --[opção]
```

No caso em que o pesquisador precise iniciar muitos módulos com muitas opções a linha de comando pode ficar um pouco confusa, para isso basta escrever um componente simples que lança outros componentes.

5.4 Componentes do POX

Os componentes do POX são programas que são executados no controlador, em outras palavras os componentes fornecem as funções para que o controlador trabalhe em uma rede SDN de acordo com a política do(s) componente(s). Dessa forma quando se fala em componentes para o POX quer-se dizer o que é possível colocar na linha de comando do POX em sua inicialização.

O POX já traz consigo alguns componentes básicos entre eles há alguns convenientes para testes e há outros que são apenas exemplos, mas a intenção é que o pesquisador possa criar seu próprio componente. Nos Quadros A.1, A.2 e A.3 (Apêndice A) são apresentados alguns destes componentes, alguns estão somente na versão *beta*.

Dentre os apresentados podemos destacar o componente "`py`", que inicializa um interpretador Python para depuração interativa, o componente "`forwarding.l2_learning`", que faz com que os *switches* OpenFlow funcionem como *switches* de auto aprendizagem e o componente "`misc.of_tutorial`", que trata-se do tutorial OpenFlow[2] para iniciantes.

[2]OpenFlow Tutorial `http://www.openflow.org/wk/index.php/OpenFlow_Tutorial`

5.5 Compreendendo a Função launch()

Em geral todos os componentes devem ter uma função chamada `launch`. Quando executamos o POX e chamamos algum componente, o POX executa a função `launch()` deste componente. Esta função deve realizar todas as ações do componente instanciando classes ou chamando outras funções e métodos. Por exemplo:

```
1 def launch():
2   print "Meu componente"
3   " " "
4   Outras operacoes a serem executadas
5   " " "
```

Caso a função `launch` receba parâmetros, é possível atribuí-los na linha de comando quando inicializar o POX.

```
1 def launch(a, b = "dois", c = True ):
2   print "Meu componente imprime:" , a, b, c
3   " " "
4   Outras operacoes a serem executadas
5   " " "
```

Podemos inicializar os parâmetros dessa forma no terminal:

```
./pox.py meu_coponente --a=3 --b --c=disabled
```

Note que neste exemplo o parâmetro "`a`" não recebe nenhum valor padrão. Caso tente iniciar o componente sem atribuir um valor nas opções, o POX retornará um erro. É possível atribuir, nas opções, qualquer tipo à variável, neste caso foi atribuído um inteiro, mas poderia ser uma string.

Note que ao parâmetro "`a`", não foi atribuído nenhum valor, neste caso o Python atribuiu o valor de "`a`" igual a "`True`", caso "`b`" não estivesse nas opções de inicialização ele teria atribuído o valor "`dois`".

Já o parâmetro "`c`", que recebe por padrão o valor "`True`", caso se queira enviar o valor "`False`" não será possível atribuir diretamente pelas opções de inicialização, pois o Python entenderá o valor como uma *String*. O que se pode fazer é criar algum código que converta o valor para "`False`" (booleano), ou utilizar-se da função `pox.lib.util's str_to_bool()` que aceita valores como "`on`", "`true`" ou "`enabled`" como "`True`" (booleano) e o restante como "`False`" (booleano).

5.6 Núcleo do POX

O POX tem um objeto chamado "`pox.core`" que serve como repositório para grande parte de suas API's. Sua finalidade principal é proporcionar um ponto de encontro entre os componentes. Para isso os componentes precisam registrar objetos arbitrários que irão passar a ser acessíveis por todos os módulos do sistema. Por exemplo, é possível criar um componente e registrar suas funcionalidades no núcleo para que outros componentes utilizem essas funções sem depender do comando "`import`" do Python, reduzindo o código de inicialização.

Uma das principais vantagens dessa abordagem é que as dependências entre os componentes não são codificadas, diferentes componentes que exponham a mesma interface podem trocá-las facilmente entre si. Muitos módulos no POX irão querer ter acesso ao núcleo, para isso basta importar o objeto:

```
1 from pox.core import core
```

Para registrar um objeto no núcleo podemos utilizar o "`core.register()`" que leva dois argumentos, o primeiro é o nome que se deseja usar quando for requisitado objetos desse componente, o segundo é o objeto que queremos registrar no núcleo. Segue o exemplo de um componente simples com uma função `launch()` que registra o componente `coisa`.

```
1  from pox.core import core
2  class MeuComponente (object):
3    def __init__ (self, an_arg):
4      self.arg = an_arg # Atribuicao da istancia a variavel 'arg'
5      print "\n\nMeuComponente TEM que imprime a variavel arg:", self.arg
6
7    def funcao (self):
8      print "\n\nMeuComponente ESTA imprimindo a variavel arg:", self.arg, "\
         n\n"
9
10 def launch ():
11   componente = MeuComponente("spam")
12   core.register("coisa", componente)
13   core.coisa.funcao() #  ira imprimir: "MeuComponente esta imprimindo a
       variavel arg: spam"
```

Existem outras formas de se registrar objetos no núcleo, para maiores informações consulte a POX Wiki [1].

5.7 Eventos no POX

O POX é orientado a eventos, seu sistema de controle é implementado pela biblioteca `pox.lib.revent`. O princípio desta implementação é que os objetos se tornam

eventos e não apenas uma coleção de campos e métodos. Um objeto pode disparar eventos e outros podem esperar por esses eventos registrando manipuladores (*handlers*).

Eventos no POX são todos os objetos instanciados de subclasses de `revent.Event`. Um objeto que gera eventos deve herdar de `revent.EventMixin`. Tais objetos normalmente especificam quais tipos de eventos ele pode disparar. Dessa forma esses eventos são normalmente classes que estendem de `Event` passando a possuir certas características, mas na prática podem ser qualquer coisa. O objeto que esteja interessado em um evento pode registrar um manipulador (*handler*) para ele.

Alternativamente é possível escutar eventos diretamente, utilizando o núcleo do POX. A classe que cria os eventos, que herdam de `revent.EventMixin`, pode registrar objetos no núcleo do POX de maneira que, para as classe que queiram ouvir esses eventos, basta executar o método registrado no núcleo do POX. É o que acontece com a classe pox.openflow.of_01 que cria os eventos openflow e registra eles no método `core.openflow.addListeners(self)` do núcleo do POX.

Caso um objeto tenha interesse em todos os eventos disparados por outro objeto não é necessário registrar um manipulador para cada evento, pode-se utilizar o método `EventMixin.listenTo()`. Por exemplo, se existem duas classes `Abc` e `Def` que herdam de `revent.EventMixin` e se `Abc` disparasse os eventos `x` e `y`, para que `Def` possa registrar manipuladores para esses eventos normalmente seria necessário definir os seguintes métodos `handle_x` e `handle_y`. Porém a classe `Def` pode registrar todos os eventos de `Abc` de uma só vez simplesmente executando o método `EventMixin.listenTo()` da seguinte maneira: `self.listenTo(someAbcObject)`. Essa forma é usualmente mais simples que registrar um manipulador para cada evento. Maiores informações sobre os eventos do POX podem ser encontrados em POX Wiki [1]

5.8 Pacotes do POX

Por ser um controlador de redes SDN, o POX tem que interagir com diversos tipos de pacotes. Para isso o POX tem uma biblioteca exclusiva para a análise e construção de pacotes, a biblioteca `pox.lib.packet`.

Esta biblioteca tem suporte a uma série de tipos de pacotes, a maioria desses pacotes tem algum tipo de cabeçalho e uma espécie de carga útil (*payload*) que muitas vezes é outro tipo de pacote. Por exemplo, o POX frequentemente trabalha com pacotes `ethernet`, que por sua vez contém pacotes `ipv4` e que por sua vez contém pacotes `tcp`. São alguns tipos de pacotes suportados pelo POX:

- ethernet
- IPv4
- TCP
- DHCP
- LLDP
- ARP
- ICMP
- UDP
- DNS
- VLAN

As classes de pacotes no POX podem ser encontradas no diretório `pox/lib/packet`. Para importar a biblioteca de pacotes basta fazer:

```
1 import pox.lib.packet as pkt
```

Pode-se navegar pelos pacotes encapsulados de duas maneiras, usando o método `payload` de cada pacote para obter o pacote superior da pilha de protocolos TCP/IP ou usando o método `find()`. Um exemplo simples de como é possível analisar pacotes ICMP usando o `payload`.

```
1 def parseICMP(packet):
2     if eth_packet.type == ethernet.IP_TYPE: # verifica se o pacote ethernet
          contem um pacote ip
3         ip_packet = eth_packet.payload # ip_packet recebe o payload de
             eth_packet
4         if ip_packet.protocol = ipv4.ICMP_PROTOCOL # verifica se o pacote
             ip utiliza o protocolo icmp
5             icmp_packet = ip_packet.payload # icmp_packet recebe o payload
                 de ip_packet
6 ...
```

Um maneira mais simples de obter os pacotes é utilizando o método `find()`. Ele localiza o pacote encapsulado pelo nome, por exemplo, '`ipv4`'. Caso o método nao encontre nenhum pacotes deste tipo ele retornará `None` (nada). Por exemplo:

```
1 def handle_IP_packet (packet):
2   ip = packet.find('ipv4') # utilizacao do metodo find()
3   if ip is None:
4     # Este pacote nao tem pacote ip
5     return
6   print "Fonte do pacote IP:", ip.srcip # imprime a fonte do pacote ip
```

Os Quadros A.4, A.5 e A.6 (Apêndice A) mostram uma análise de algumas das principais classes da biblioteca `pox.lib.packet`, maiores detalhes sobre as outras classes podem ser encontradas em suas classes especificas no diretório `pox/lib/packet`.

5.9 Threads no POX

Geralmente o modelo de eventos não levanta a necessidade do uso de *threads*, porém caso algum componente queira executar alguma ação ao longo do tempo,

como por exemplo, examinar os *bytes* transferidos ao longo de um fluxo específico a cada dez segundos, é interessante entender o modelo de *threads* utilizado no POX.

O modelo de *threads* do POX é modelado pela biblioteca `pox.lib.recoco`. Esta biblioteca cria um modelo de *threads* cooperativas em que as próprias *threads* delegam o processamento entre si. Diferente do que ocorre no modelo preemptivo no qual o sistema ativamente interrompe o processamento de um *thread* e o delega a outra.

Dessa forma todas as *threads* cooperativas precisam ser criadas por uma classe que estenda `pox.lib.recoco.Task` e implemente o método `run`. Após a sua criação a *thread* precisa ser escalonada e quando for escolhida para ser executada ela pode se considerar um processo atômico, que irá ser executada até o final ou até bloquear a si própria com o método `Sleep()`.

Ao trabalhar com *threads* deve-se ter cautela para evitar executar uma operação que pode bloquear o funcionamento do sistema. Por esses motivos é interessante trabalhar com a classe `pox.lib.recoco.Timer` que por sua vez já estende da classe `Task` facilitando muito o trabalho do modelo de *threads*.

A classe `Timer` foi projetada para lidar com a execução de pedaços de códigos de uma única vez ou de forma recorrente. Esta classe já lida com modelo de *threads* automaticamente, bastando apenas definir seus argumentos de entrada quando chamada. No apêndice A podem ser encontrados os Quadros A.7 e A.8, que mostram os argumentos de construção da classe, e o Quadro A.9, que mostra seus métodos.

Outra maneira possivel de se trabalhar com *thread* é utilizar-se do núcleo do POX com o objeto `callDelayed()` que age exatamente da mesma maneira do `Timer` sem que se precise importar a biblioteca `recoco`.

Exemplo de como se trabalhar com *threads* (Disparo de uma única vez):

```
1  from pox.lib.recoco import Timer
2
3  def handle_timer_elapse (mensagem):
4    print "Me mandaram falar:", mensagem
5
6  Timer(10, handle_timer_elapse, args = ["Ola"])
7
8  # Ira imprimir "Me mandaram falar: Ola" em 10 segundos
9
10 # Uma outra maneira de se fazer o Timer utilizando o nucleo do POX:
11 from pox.core import core # Muitos componente fazem isso!
12 core.callDelayed(10, handler_timer_elapse, "Ola")
```

Exemplo de como se trabalhar com *threads* (Disparo recorrente):

```
1  # Simula uma longa viagem
2
3  from pox.lib.recoco import Timer
4
5  nos_estamos_la = False
6
```

```
7  def nos_ja_estamos_la():
8    if nos_estamos_la:
9      return False # Cancela o Timer (veja o selfStoppable)
10   print "Nos ja estamos la?"
11
12 Timer(30, nos_ja_estamos_la, recurring = True) # ficara imprimindo a
     pergunta: "Nos ja estamos la?" a cada 30 segundos ate nos_estamos_la =
     True, e cancela o Timer.
```

5.10 OpenFlow no POX

O propósito principal do POX é desenvolver aplicações de controle OpenFlow. Esta seção descreve algumas características das interfaces de controle OpenFlow do POX. A biblioteca OpenFlow implementa facilidades para a comunicação com comutadores OpenFlow e para a recepção e criação de pacotes.

O POX tem um componente que se comunica diretamente com os comutadores OpenFlow, este componente é o `openflow.of_01` (01 refere-se a versão do protocolo OpenFlow 1.0). Como já foi dito este componente inicia automaticamente ao se executar o POX (a menos que a opção `--no-openflow` seja declarada) e quando é iniciado, por padrão, ele registra um objeto no núcleo do POX (`pox.core`) com o nome "openflow". Essa é a principal interface para se trabalhar com o OpenFlow no POX. É possível utilizar este objeto para enviar e receber mensagens de controle para os comutadores com OpenFlow habilitado.

O POX trabalha com o OpenFlow por meio de eventos. Os eventos são registrados pelo componente `of_01` e podem ser observados por qualquer outro componente ao registrar um tratador (*handle*) para esses eventos. Por exemplo, pacotes oriundos de comutadores OpenFlow são recebidos pelo objeto `openflow`, que por sua vez gera um evento `PacketIn` que pode ser observado por qualquer outro componente que esteja interessado na chegada do pacote.

A maioria dos eventos relacionados ao OpenFlow são criados em resposta direta a uma mensagem recebida pelo comutador OpenFlow. Em geral os eventos OpenFlow têm os seguintes atributos (Quadro 5.2):

Quadro 5.2: Atributos gerais dos Eventos OpenFlow no POX.

Atributo	**Tipo**	**Descrição**
`connection`	Connection	É a conexão com o comutador que causou esse evento
`dpid`	long	Corresponde ao ID do comutador que causou o evento
`ofp`	ofp_header subclass	É a mensagem OpenFlow que o evento trás, a seção 5.11 explica os tipos de mensagens

O módulo OpenFlow fornece vários eventos para controle de mensagens OpenFlow, esses eventos são descritos nos Quadros A.10 e A.11 (Apêndice A). Todos os eventos são registrados pelo módulo OpenFlow do POX e suas classes podem ser encontradas em `pox.openflow.__init__.py`, maiores detalhes de seus atributos e métodos podem ser encontrados em POX Wiki [1].

A seguir é apresentado um componente que escuta os eventos `ConnectionUp`, `ConnectionDown`, `PortStatus`, `PacketIn` e `FlowRemoved` respectivamente.

```
1  from pox.core import core
2  from pox.lib.util import dpidToStr # dpidToStr e um metodo que converte o
       id de um comutador para string
3
4  log = core.getLogger() # para exibir logs no sistema, neste exemplo e
       utilizado o log.debug()
5
6  class MyComponent (object):
7    def __init__ (self):
8      core.openflow.addListeners(self) # para ouvir todos os eventos OpenFlow
           do core
9
10   def _handle_ConnectionUp (self, event): # tratador de eventos
       ConnectionUp, todas as vezes que este evento ocorrer, este metodo e
       executado
11     log.debug("O switch %s se conectou ao controlador.", dpidToStr(event.
         dpid))
12
13   def _handle_ConnectionDown (self, event): # tratador de eventos
       ConnectionDown
14     log.debug("O switch %s perdeu a conexao com o controlador.", dpidToStr(
         event.dpid))
15
16   def _handle_PortStatus (self, event): # tratador de eventos PortStatus
17     if event.added:
18       action = "adicionada"
19     elif event.deleted:
20       action = "retirada"
21     else:
22       action = "modificada"
23     log.debug("A porta %s no switch %s foi %s.",(event.port, dpidToStr(
         event.dpid), action)
24
25   def _handle_PacketIn (self, event): # tratador de eventos PacketIn
26     log.debug("Chegou um pacote no switch %s" pela porta %d, dpidToStr(
         event.connection.dpid), event.port)
27     packet = event.parsed # para que se possa trabalhar com o pacote
28
29   def _handle_FlowRemoved (self, event): # tratador de eventos FlowRemoved
30     log.debug("Fluxo foi removido no switch %s", dpidToStr(event.dpid))
31
32 def launch ():
33   core.registerNew(MyComponent) # registrando o componente no nucleo (core)
```

Para que se possa compreender melhor o que esse componente faz, basta executá-lo juntamente com um componente do tipo `forwarding` como o `forwarding.l2_learning` e observar sua saída.

5.11 Mensagens OpenFlow no POX

São com as mensagens OpenFlow que os comutadores se comunicam com os controladores. Todas essas mensagens são definidas na Especificação OpenFlow [42]. Existem várias versões da especificação, o POX atualmente trabalha com o OpenFlow versão 1.0.

O POX contém classes e constantes que correspondem a todos os elementos do protocolo OpenFlow, eles estão definidos em `pox.openflow.libopenflow_01.py`. Grande parte dos nomes das classes são os mesmos da especificação, outros no entanto se diferem para se adequar ao sistema, além disso o POX define algumas estruturas que não correspondem a nenhuma estrutura da especificação. Adiante é apresentado algumas dessas classes e suas descrições básicas.

5.11.1 of.ofp_packet_out

O principal objetivo desta mensagem é instruir um comutador a enviar um pacote em uma determinada porta. Outra utilidade desta mensagem é instruir um comutador a descartar um pacote que esteja no seu *buffer*. Esta classe pode ser encontrada na linha 2886 do arquivo `pox/openflow/libopenflow_01.py`. Os Quadros A.12 e A.13 (Apêndice A) mostram os atributos e as descrições da classe.

5.11.2 of.ofp_flow_mod

O principal objetivo desta mensagem é modificar as tabelas de fluxo de um comutador. O controlador pode adicionar, modificar e deletar entradas na tabela de fluxo de um comutador. Esta classe pode ser encontrada na linha 1816 do arquivo `pox/openflow/libopenflow_01.py`. Exemplos de como realizar uma entrada na tabela de fluxo de um comutador podem ser vistos em POX Wiki [1]. Os Quadros A.14 e A.15 (Apêndice A) mostram os atributos e as descrições da classe.

5.11.3 of.ofp_match

Esta classe define a estrutura de um fluxo. É possível construir estruturas de fluxo a partir do nada para que possam ser utilizadas em outras classes OpenFlow do POX. Esta classe pode ser encontrada na linha 405 do arquivo `pox/openflow/libopenflow_01.py`. O Quadro A.16 (Apêndice A) mostra os atributos e suas descrições. Exemplo de como criar uma estrutura de um fluxo:

```
1 # criar uma estrutura de fluxo em uma linha
2 my_match = of.ofp_match(in_port = 5, dl_dst = EthAddr("01:02:03:04:05:06"))
3
```

```
4 #ou criar uma estrutura de fluxo em varias linhas
5 my_match = of.ofp_match()
6 my_match.in_port = 5
7 my_match.dl_dst = EthAddr("01:02:03:04:05:06")
```

5.11.4 ofp_action

É um conjunto de classes que definem ações possíveis de uma regra. Quando definida uma ação (uma classe) o comutador realizará esta ação a partir do fluxo registrado na tabela de fluxos. Essas classes podem ser encontradas no arquivo `pox/openflow/ libopenflow_01.py`. O Quadro A.17 (Apêndice A) mostra as classes de ação do POX e suas descrições. Maiores detalhes sobre o funcionamentos dessas classe podem ser encontrados em POX Wiki [1].

5.12 Desenvolvendo Componentes Próprios

Esta seção mostra como construir um componente próprio para o POX. Em alguns casos o pesquisador pode reutilizar trechos de códigos de alguns componentes já existentes nos seus próprios componentes. Nesses casos é possível fazer uma copia de algum componente e trabalhar em cima deles. O Capitulo 7 mostrará como criar componentes mais concretos e como testá-los.

Como podemos observar os componentes do POX são apenas módulos Python. É possível colocar esses módulos em qualquer lugar. O usual é colocar esse módulos no diretório `ext`. Este diretório é o local mais conveniente para que os pesquisadores possam colocar seus componentes próprios, pois o POX pesquisa automaticamente esse caminho quando é executado.

Dessa forma uma maneira simples de começar a construir um componente próprio é copiar um componente já existente (`forwarding.l2_learning.py`) para o diretório `ext`, renomeá-lo (`meu_componente.py`) e começar a modificar o arquivo. Para executar ele no POX basta dar o comando: `./pox.py meu_coponente`.

5.12.1 Como criar um componente básico no POX

Esta seção mostra como criar um componente para o POX. Primeiramente é necessário criar um arquivo em branco no diretório ext (de preferência). Por exemplo o arquivo `teste.py`. Abra o arquivo (vide código a seguir) e crie o método `launch()` (linha 22) e se quiser alguns parâmetros (Seção 5.5). Este método é executado assim que o POX chama o componente. É interessante registrar o componente no núcleo do POX, seção 5.6 (linha 23 ou 25). Agora é necessário criar a classe do componente `teste` (linha 6), essa classe deve escutar os eventos openflow, para isso é necessário executar o método `addListeners` do núcleo do POX (Seção 5.7) (linha 9). Agora que o componente `teste` pode escutar eventos é necessário criar funções para tratar certos eventos. Para tratar o evento `ConnectionUp`, que é lançado toda

vez que o POX detecta uma nova conexão, é necessário criar um tratador para ele, o método `_handle_ConnectionUp` (linha 14), esse método recebe como parâmetro o próprio evento e pode realizar qualquer tarefa com ele (Seção 5.10). O componente `teste.py` ficará assim:

```
1  from pox.core import core
2  from pox.lib.util import dpidToStr # dpid_to_str e um metodo que converte o
       id de um comutador para string
3
4  log = core.getLogger() # para exibir logs no sistema, neste exemplo e
       utilizado o log.debug()
5
6  class Teste (object):
7    def __init__ (self, an_arg):
8      self.arg = an_arg
9      core.openflow.addListeners(self) # para ouvir todos os eventos OpenFlow
           do core
10
11   def meu_nome (self):
12     print "Sou o componente:", self.arg
13
14   def _handle_ConnectionUp (self, event): # tratador de eventos
       ConnectionUp, todas as vezes que este evento ocorrer, este metodo e
       executado
15     log.debug("O switch %s se conectou ao controlador.", dpidToStr(event.
         dpid))
16   self.meu_nome()
17
18   def _handle_PacketIn (self, event):
19   # executando o metodo registardo no nucleo do pox
20   core.teste.meu_nome() # imprimira "Sou o componente: TesteOpenFlow" toda
       vez que um pacote chegar
21
22 def launch ():
23   component = Teste("TesteOpenFlow")
24   core.register("teste", component) # registrando o componente teste
25   # outra maneira de registrar o componente: core.registerNew(Teste)
```

Para executá-lo faça:

```
./pox.py teste.py
```

É necessário deixar claro que o código gerado aqui funcionará tanto em redes SDN reais como no simulador MiniNet, que será explicado no próximo capítulo. No Capítulo 7 será mostrado como testar os componentes aqui apresentados e como criar componentes mais realistas e mais complexos.

Capítulo 6

MiniNet

Para a criação de uma arquitetura SDN é necessário múltiplos componentes em sua infraestrutura. Inicialmente precisamos de uma máquina controladora operando algum sistema operacional de rede, como por exemplo, o NOX ou o POX. Posteriormente se fazem necessários elementos encaminhadores que tenham o plano de controle separado do plano de dados com alguma interface de programação, como por exemplo o OpenFlow. Além disso, ainda é necessário toda uma infraestrutura física que interligue esses elementos encaminhadores e o controlador por um canal seguro para a criação de uma Rede Definida por Software.

Ocorre que nem sempre um pesquisador dispõe de uma infraestrutura como essa para realizar seus experimentos, pois essa estrutura possui um custo elevado. Para lidar com esse problema é possível usar um simulador de Redes Definidas por *Software*.

O MiniNet é uma ferramenta para a simulação de Redes Definidas por Software que permite a rápida prototipação de uma grande infraestrutura virtual de rede com a utilização de apenas um computador. O Mininet também possibilita a criação de protótipos de redes virtuais escaláveis baseados em *software* como OpenFlow utilizando primitivas de virtualização do Sistema Operacional. Com essas primitivas, ele permite criar, interagir e customizar protótipos de Redes Definidas por Software de forma rápida.

São algumas características do MiniNet:

- Fornecer uma maneira simples e barata para a realização de testes em redes para desenvolvimento de aplicações OpenFlow;
- Permite que múltiplos pesquisadores possam trabalhar de forma independente na mesma topologia de rede;
- Permite o teste de uma topologia grande e complexa, sem mesmo a necessidade de uma rede física;
- Inclui ferramentas para depurar e executar testes em toda a rede;
- Dá suporte a inúmeras topologias, e inclui um conjunto básicos de topologias;
- Oferece API's simples em Python para criação e experimentação de redes.

Com o MiniNet é possível realizar testes de depuração e de resoluções de problemas podendo se beneficiar de ter uma rede completa experimental em um laptop ou PC. Fornece, de maneira intuitiva, ferramentas para o desenvolvimento de aprendizagem na área de redes. Sua interface permite sua utilização em pesquisas e em aulas para o uso prático de técnicas e soluções de redes.

O MiniNet é melhor executado em máquina virtual (VM MiniNet) sendo executada em VMware ou VirtualBox para os sistemas operacionais Windows, Mac e Linux com ferramentas OpenFlow já instaladas. Alternativamente é possível instala-lo no Ubuntu 11.10, embora os desenvolvedores não recomendem.

Mesmo com todas essas qualidades o MiniNet ainda não fornece desempenho e qualidade fieis de uma rede real, embora o código utilizado nele sirva para uma rede real baseada em *switches* de *software* NetFPGAs[1], ou *switches* e, *hardware* comerciais. Isso ocorre devido aos recursos que são multiplexados em tempo real pelo kernel da máquina simuladora, e uma vez que a largura de banda total é limitada por restrições de CPU e memória da mesma.

Este capítulo irá apresentar um passo-a-passo dos principais comandos do MiniNet, assumindo uma utilização com máquinas virtuais. Mostrará também o passo-a-passo de como estabelecer um ambiente virtual de testes para a criação e o desenvolvimento de uma arquitetura baseada em Redes Definidas por Software. Com as explicações deste capítulo é possível compreender melhor o MiniNet e, assim, poder executar os componentes do POX em uma rede virtual, como será apresentado no Capítulo 7.

6.1 Funcionamento

Quase todos os sistemas operacionais virtualizam os recursos de computação usando uma abstração de processos. O MiniNet usa processos baseados na virtualização de vários *hosts* e *switches* executados em um único *kernel* do sistema operacional.

O MiniNet pode criar *switches* OpenFlow, controladores e *hosts* para a comunicação da rede simulada. O MiniNet conecta *switches* e *hosts* usando enlaces *ethernet* virtuais entre os pares.

O MiniNet ainda depende do *kernel* do Linux para sua execução, no entanto seus desenvolvedores estão desenvolvendo outras técnicas para sua execução em outros sistemas operacionais com virtualização baseada em processos, como por exemplo o Solaris.

[1] O projeto NetFPGA refere-se ao esforço para o desenvolvimento de *hardwares* de código aberto e plataforma de *software* para permitir a prototipagem rápida de dispositivos de rede. O projeto visa principalmente os pesquisadores, usuários da indústria e também estudantes em sala de aula. `netfpga.org`

6.2 MiniNet na Prática

São apresentados os procedimentos de instalação e configuração do VM MiniNet, que além de recomendado pelos desenvolvedores é mais simples e pode ser executado em Windows, Mac e Linux por meio de uma VMware, VirtualBox, QEMU ou KVM.

6.2.1 Pré-Requisitos

É necessário um computador com 2GB de RAM e pelo menos 6GB de espaço livre no disco rígido. Um processador mais rápido pode acelerar o tempo de *boot* da máquina virtual, e uma tela maior pode ajudar a gerenciar as múltiplas janelas de terminal.

6.2.2 Download VM MiniNet

O download do VM Mininet pode ser feito em `https://github.com/downloads/mininet/mininet/mininet-vm-ubuntu11.10-052312.vmware.zip`. Nele está incluído uma imagem de disco VMware no formato `.vmdk` (disco de máquina virtual) no qual pode ser usado na maioria dos sistemas de virtualização.

É necessário também um terminal capaz de se conectar via SSH[2] e um servidor X[3]. Com todos esses requisitos devidamente instalados basta inicializar a máquina virtual com o *software* de virtualização de preferência e extrair o arquivo baixado.

6.2.3 Configuração da VM MiniNet

Antes de executar o VM MiniNet algumas configurações são necessárias em seu software de virtualização:

- A escolha do sistema operacional deve ser o Linux Ubuntu;
- A alocação deve ser de pelo menos 512Mb de memória para a máquina virtual;
- Utilizar duas placas de rede na máquina virtual, sendo uma destinada a sua própria rede interna (NAT) e outra para uma interface entre a máquina virtual e máquina física para o uso do SSH (*host-only network*)

Para realizar essas configurações consulte a ajuda do *software* de virtualização utilizado. Neste ponto o sistema está pronto para a execução do VM MiniNet.

Quando estiver executando a máquina virtual use os seguinte parâmetros:

[2] **SSH** (**Secure Shell**) é, ao mesmo tempo, um programa de computador e um protocolo de rede que permitem a conexão com outro computador na rede de forma a permitir execução de comandos de uma unidade remota.

[3] **X Window System**, **X-Window**, **X11** ou simplesmente **X** é um software de sistema e um protocolo que fornece uma base para interfaces gráficas de usuário (com o conceito de janelas) e funcionalidade rica de dispositivos de entrada para redes de computadores.

- Username: **openflow**
- Password: **openflow**

A conta `root` não está ativada para login, é possível utilizar o comando '`sudo`' para realizar comandos como superusuário.

Primeiro encontre os endereços IPs da máquina virtual, no console escreva:

```
$ ifconfig
```

Será possível ver três interfaces (Ex.: `eth0, eth1, lo`). As duas primeiras são as configurações das duas placas de rede já configuradas antes de executar a máquina virtual. Caso seja possível ver, basta executar o comando:

```
$ sudo dhclient eth0
$ sudo dhclient eth1
```

Execute novamente o comando '`ifconfig`' para habilitar as interfaces, que agora poderão ser vistas.

Em alguns casos essas interfaces estão com outros nomes, no entanto o padrão é '`ethX`', onde '`X`' é algum número, execute os comandos com esses nomes e verifique se as interfaces são habilitadas.

Com as interfaces habilitadas anote o endereço IP da segunda interface (*host-only network*) que provavelmente seguirá o padrão de IP da sua máquina física. Após isso não é necessário utilizar mais o *shell* da máquina virtual, pois com este IP podemos conectar a máquina física na máquina virtual via SSH, tornando mais fácil a execução de comandos e o uso de múltiplas janelas de terminal.

6.2.4 Notas sobre o Prompt de Comando

Os comando apresentados neste documento são mostrados com um prompt de comando para indicar o subsistema que se destina. Por exemplo:

```
$ ls
```

Os comandos seguidos de `$` indica que eles deverão ser digitados em um prompt de comando do sistema Unix (Linux ou OS X), seja na máquina virtual ou na própria máquina física.

```
mininet> h2 ping -c3 h3
```

Os comandos seguidos `mininet>` indica que eles deverão ser digitados no ambiente simulado do MiniNet, somente na máquina virtual, assumindo que se utilizará a VM MiniNet.

```
# ping -c3 10.0.0.3
```

Os comandos seguidos `#` indica que eles deverão ser digitados em uma janela de servidor X. Geralmente são usados quando se acessa algum nó na rede virtual. O símbolo '`#`' no Unix também indica que você está com acesso de super usuário.

6.2.5 Acessando a VM MiniNet via SSH

Com todos os requisitos instalados e as configurações feitas siga:

Para usuários de Mac OS X e Linux: Abra um terminal e digite:

```
$ ssh -X openflow@[Endereço IP anotado(Guest)]
```

Após isso será pedido o *password*, digite `openflow`. A opção '`-X`' é para habilitar o servidor X na máquina virtual. Execute:

```
$ xterm
```

Uma nova janela de terminal deve aparecer. Se tiver sucesso então está feita a configuração básica. Feche o *xterm*. Caso não tenha sucesso verifique a instalação do servidor X na plataforma. Caso queira outras janelas de SSH repita o processo.

Para usuários de Windows: é necessário executar o servidor X na máquina hospedeira (física) e configura-lo para ser executado no SSH instalado. Como servidor X é recomendado o Xming[4], como SSH é recomendado o PuTTY[5]. Para ativar o Xming no PuTTY abra o PuTTY e vá na aba Connection→SSH→X11, e

[4] **Xming** é uma implementação do *X Windows System* para sistemas operacionais do Microsoft Windows.

[5] O **PuTTY** é um cliente SSH destinado a promover o acesso remoto a servidores via Shell Seguro e a construção de túneis cifrados entre servidores.

em seguida marque a caixa '*Enable X11 Forwarding*'. Após essa configuração vá na aba *Session* e em '*Host Name (or IP address)*' na caixa de texto digite o IP anotado anteriormente da sua máquina virtual. Em '*Port*' escreva 22. Não esqueça de deixar marcado o *radio button SSH* em '*Connection Type*'. Se tudo der certo será pedido o *login* e o *password*, digite '`openflow`' nos dois. Execute:

```
$ xterm
```

Uma nova janela de terminal deve aparecer. Se tiver sucesso então está feita a configuração básica no Windows. Feche o *xterm*. Caso queira outras janelas de SSH repita o processo.

6.2.6 Instalando os Editores de Texto

Com todo o processo concluído será possível ter total acesso ao VM Mininet pelo SSH. Instale o seu editor de texto preferido, por exemplo, o vim:

```
$ sudo apt-get install vim
```

Outro editor recomendado é o Gedit, que por sua vez é gráfico e de fácil utilização. Também é possível instalar o Eclipse, que é uma ótima IDE para se trabalhar.

6.2.7 Aprendendo as Ferramentas de Desenvolvimento

Esta seção apresenta o ambiente de desenvolvimento para a fácil familiarização das ferramentas utilizadas. São cobertas tanto ferramentas de depuração quanto ferramentas específicas de OpenFlow.

Definição dos termos, começando com os tipos de terminais:

terminal da máquina virtual: Este terminal é o console criado quando você iniciou a MV. Não é possível copiar e colar neste terminal, nenhuma das ações com o mouse terá efeito neste terminal, inclusive barras de rolagem. Não será necessário o uso deste terminal, uma vez que você consiga configurar a rede.

terminal SSH: Ele é criado a partir do software de SSH, conforme apresentado na seção anterior. Este terminal é utilizado para se conectar ao *shell* da máquina virtual. É possível copiar e colar por este terminal além de ter barra de rolagem de pelo menos 500 linhas.

terminal xterm: Ele é utilizado para se conectar a um *host* da rede virtual. Ele é utilizado dentro do CLI[6] do MiniNet para que se possa ter múltiplas janelas para o desenvolvimento. Ele se conectará a uma elemento da rede virtual.

A VM MiniNet contém uma série de ferramentas de rede pré-instaladas, sejam gerais ou específicas do OpenFlow. Algumas delas são:

Controlador OpenFlow: Situa-se acima da interface OpenFlow. A MV contém um controlador de referência pré-instalado no ambiente que age como um *self-learning Switch Ethernet*.

***Switch* OpenFlow:** Situa-se abaixo da interface OpenFlow. A MV contém três opções de *switches* para a rede virtual, *switches* que utilizem o *user-space*, *switches* que utilizem o *kernel-space* e *switches* que simulam o Open vSwitch já apresentado na pagina 41.

dpctl: Utilitário de linha de comando que envia mensagens rápidas do protocolo OpenFlow, útil para visualizar portas de *switch* e estatísticas de fluxo, além de manualmente inserir entradas de fluxo.

Wireshark: Ferramenta gráfica geral para a visualização de pacotes (não é específico de OpenFlow). A distribuição de referência OpenFlow inclui um Wireshark dissector, que disseca e analisa as mensagens OpenFlow enviado pela porta padrão do OpenFlow (6633).

iperf: Ferramenta geral de linha de comando para testar a velocidade de uma única conexão TCP.

MiniNet A plataforma de simulação de rede que é utilizada.

6.2.8 Principais Comandos do MiniNet

Antes de mostrar as principais configurações para a execução do CLI do MiniNet, será mostrado os principais comando utilizados dentro do CLI do MiniNet. Como já dito assumimos que esteja utilizando a VM MiniNet.

Para iniciar o CLI do MiniNet no terminal basta executar o comando:

```
$ sudo mn
```

Por padrão uma topologia minima é criada. Essa topologia inclui dois *hosts* ligados a um *switch* OpenFlow por um enlace *ethernet* e um controlador ligado ao *switch*.

Em seguinte o CLI do MiniNet estará sendo executado apresentando no terminal: 'mininet>'

[6] *Command-line Interface* (CLI) é um meio de interação com um programa de computador onde o usuário (cliente) emite comando para um programa na forma de sucessivas linhas de texto (linhas de comando).

Para exibir os comandos do MiniNet, digite no terminal:

```
mininet> help
```

Exibir nós:

```
mininet> nodes
```

Exibir links

```
mininet> net
```

Mostrar informações sobre todos os nós:

```
mininet> dump
```

Se a primeira palavra digitada no CLI do MiniNet for o nome de um nó (*switch*, *host* ou controlador), o comando seguinte será executado por este nó. Exemplo:

```
mininet> h2 ifconfig
```

Irá apresentar as configurações de IP do *host* '`h2`'. Outra opção é abrir uma janela de terminal separada para o nó desejado, para isso utiliza-se o servidor X. Exemplo:

```
mininet> xterm h2 h3 s1
```

Isso irá abrir três janelas cada uma com o nome correspondente ao seu nó. Vá na janela com nome '*switch: s1 (root)*' e execute:

```
# dpctl dump-flows tcp:  127.0.0.1:6634
```

Nada irá acontecer, pois o *switch* ainda não possui entradas de fluxo adicionadas. Na janela com nome *'host: h2 (root)'* e execute:

```
# ping 10.0.0.3
```

Voltando a janela com nome *'switch: s1 (root)'* execute: '`# dpctl dump-flows tcp: 127.0.0.1:6634`'. Agora sim é possível ver as entradas de fluxo do *switch*.

O `dpctl` é um ferramenta que vem com a distribuição de referência OpenFlow permitindo a visibilidade e o controle sobre uma tabela de fluxos de um único *switch* OpenFlow. Ela é muito útil para a depuração de redes, pois permite exibição do estado dos fluxos e dos contadores de fluxo. A maioria dos *switches* OpenFlow começa com uma porta de escuta passiva (em sua configuração atual essa porta é 6634), na qual é possível interagir com o *switch* pela interface OpenFlow, sem ter a necessidade de adicionar código de *debugging* no controlador.

Para exibir o estado das portas e capacidades do *switch* entre no seu terminal '`xterm`' e execute:

```
# dpctl show tcp:127.0.0.1:6634
```

Para exibir a tabela de fluxo de um *switch* entre no seu terminal '`xterm`' e execute:

```
# dpctl dump-flows tcp:127.0.0.1:6634
```

Para instalar fluxos manualmente em um *switch* entre no seu terminal '`xterm`' e execute:

```
# dpctl add-flow tcp:127.0.0.1:6634 in_port=1,actions=output:2
```

Este comando faz uma entrada na tabela de fluxo do *switch* criando uma regra. Informa ao *switch* que os tráfegos vindos da porta (entrada) 1 serão encaminhado para a porta (saída) 2. Note que por padrão este regra tem um tempo de validade (60 segundos), caso não haja tráfego neste fluxo. Alternativamente podemos modificar o tempo de validade. Exemplo:

```
# dpctl add-flow tcp:127.0.0.1:6634 in_port=1,idle_timeout=120,actions=
output:2
```

Agora o tempo de validade é de 120 segundos. Para maiores detalhes sobre a ferramenta `dpctl` execute o comando '`# dpctl -h`'.

Outro comando utilizado dentro do CLI do MiniNet é o:

```
mininet> iperf
ou
mininet> iperf [fonte] [destino]
```

O `iperf` analisa a largura de banda entre dois *hosts* da rede.

Um comando para teste de alcance entre todos os nós da rede é o:

```
mininet> pingall
```

O comando `pingall` faz com que todos os *hosts* da rede enviem *pings*[7] entre si.

Um comando últil para teste é o comando:

```
mininet> link [nó1] [nó2] [up/down]
```

Este comando desativa ou reativa links entre dois nós. Útil para testes de depuração.

Para sair do CLI do MiniNet:

```
mininet> exit
```

Para maiores detalhes de comando utilizados dentro do CLI do MiniNet visite site `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet` e procure a documentação do MiniNet.

[7] PING: é um utilitário que usa o protocolo ICMP para testar a conectividade entre equipamentos. Seu funcionamento consiste no envio de pacotes para o equipamento de destino e na "escuta" das respostas.

6.2.9 Comandos Utilizados para a Inicialização do CLI do MiniNet

Já conhecendo os principais comandos utilizados dentro do CLI do MiniNet, é apresentado então os comandos utilizados para inicializar o MiniNet (inicializar o CLI), apresentando as principais opções de inicialização.

Exibir as opções de inicialização (ainda fora do CLI) digite:

```
$ sudo mn -h
```

Este passo a passo abrangem a maioria dos comandos e opções listadas no uso das entradas do MiniNet.

Se o MiniNet não estiver funcionando corretamente, apresentando erros tente limpar todo o estado residual ou processos por meio do comando:

```
$ sudo mn -c
```

Para realizar o teste de *ping*, afim de testar seu controlador, sem necessitar de entrar no CLI execute:

```
$ sudo mn --test pingpair
```

Este comando criará a topologia básica e fará um teste entre todos os pares de *ping*, depois do teste ele sai do CLI. Outro comando pra teste é:

```
$ sudo mn --test iperf
```

Este comando também cria a topologia básica e executa o `iperf` em dois *hosts* analisando a largura de banda alcançada.

Como já vimos a topologia padrão é um único *swtch* ligado a dois *hosts*, no entanto você pode criar topologias diferentes.

```
$ sudo mn --topo single,X
ou
$ sudo mn --topo linear,X
ou
$ sudo mn --topo tree,X,Y
```

A primeira cria uma topologia com um único *switch* ligados a 'X' *hosts*, onde 'X' é um número. A segunda cria uma topologia linear com 'X' *swiches* conectados entre si, ligados cada um a 'X' *hosts*. A terceira cria uma topologia em árvore de altura 'X' e com o número 'Y' de filhos por nó.

Ainda assim é possível criar topologias totalmente diferentes e personalizadas por meio de uma simples API em Python. Um exemplo é fornecido em 'custom/topo-2sw-2host.py'. Este exemplo cria dois *switches*, ligados entre si, conectados a um *host* cada.

```
"""Custom topology example

author: Brandon Heller (brandonh@stanford.edu)

Two directly connected switches plus a host for each switch:

host --- switch --- switch --- host

Adding the 'topos' dict with a key/value pair to generate our newly defined
    topology enables one to pass in '--topo=mytopo' from the command line.
   """

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        leftHost = 1
        leftSwitch = 2
        rightSwitch = 3
        rightHost = 4

        # Add nodes
        self.add_node( leftSwitch, Node( is_switch=True ) )
        self.add_node( rightSwitch, Node( is_switch=True ) )
        self.add_node( leftHost, Node( is_switch=False ) )
        self.add_node( rightHost, Node( is_switch=False ) )
```

```
33
34          # Add edges
35          self.add_edge( leftHost, leftSwitch )
36          self.add_edge( leftSwitch, rightSwitch )
37          self.add_edge( rightSwitch, rightHost )
38
39          # Consider all switches and hosts 'on'
40          self.enable_all()
41
42  topos = { 'mytopo': ( lambda: MyTopo() ) }
```

codigos/topo-2sw-2host.py

Veja que é muito simples a criação de topologias customizadas, nas linha 23 a 26 temos o exemplo de como podemos criar os nós da rede, podemos colocar qualquer nome, no entanto temos que identificar um número pra cada nó. Nas linha 29 a 32 temos os exemplo de como adicionar nós a topologia da rede. Utiliza-se o método '`add_node`' identificando em sua segunda entrada se o nó é um *switch* ou não (*true/false*). Nas linhas 35 a 37 temos o exemplo de como podemos criar as arestas entre os nós da rede. Utiliza-se o método '`add_edge`' identificando os pares de nós que estão interligados. Dessa maneira podemos criar topologias totalmente personalizadas de acordo com pesquisa.

Para executar o MiniNet com a topologia customizada execute:

```
$ sudo mn --custom ~/mininet/custom/minha_topologia.py --topo mytopo
```

Neste exemplo o arquivo '`minha_topologia.py`' está na pasta '`/mininet/custom/`', mas pode ficar em qualquer pasta do sistema. A opção '`mytopo`' é o nome da sua topologia (veja a linha 42).

Por padrão os *hosts* e *switches* começam com endereços MAC aleatórios. Isso pode tornar díficil a depuração de suas aplicações de rede, uma vez que cada nova execução do MiniNet gera endereçõs MAC distintos entre seus *hosts* e *switches*. Para resolver este problema podemos utilizar a opção `--mac` em que cada nó terá um endereço simples e de fácil letura. Exemplo:

```
$ sudo mn --mac
```

O que antes com o comando `h2 ifconfig` apresentaria um endereço MAC = 7e:7e:63:3a:6d:c0, agora apresenta um endereço MAC = 00:00:00:00:00:02.

Para abrir um terminal de servidor X para cada nó da rede faça:

```
$ sudo mn -x
```

Isso pode ser interessante para realizar teste de monitoramento de pacotes como por exemplo o `tcpdump` ou outros testes.

Para tipos de *switch* por padrão é[8] utilizado o *switch* em espaço de kernel, no entanto podemos utilizar *switch* em espaço de usuário. Exemplo:

```
$ sudo mn --switch user
```

Note, no entanto, que a largura de banda nesta opção é muito menor, faça o teste com o '`iperf`'. Isso acontece pois os pacote agora precisam atravessar o espaço de usuário para o espaço de *kernel* e voltar a cada salto ao invés de ficar no *kernel* enquanto atravessam o *switch*. O *switch* no espaço de usuário é mais fácil de modificar (não é necessário lidar com nenhum *kernel*), no entanto é mais lento na simulação.

Esta opção pode ser interessante como ponto de partida para a implementação de novas funcionalidades e novas propostas de rede. Outra opção é o uso do Open vSwitch (OVS) que já vêm pre-instalado na VM MiniNet. Os testes relatam uma banda muito semelhante ao *switch* em espaço de *kernel*. Exemplo:

```
$ sudo mn --switch ovsk
```

Por padrão o controlador executado na simulação é o controlador de referência '`--controller=ref`', isso quando não há nenhum controlador sendo executado na máquina virtual. Quando há um controlador sendo executado na máquina virtual o controlador padrão é o controlador remoto '`--controller=remote`'. Geralmente quando se tem um controlador sendo executado na máquina virtual, este foi iniciado pelo o usuário em outra janela SSH, isso será especificado melhor no próximo capítulo.

Quando especificamos a opção '`--controller=remote`' sem nenhum controlador sendo executado, a rede se inicia com um controlador que não realiza nenhuma ação. Isso tem sentido quando criamos regras manuais para testes específicos com o comando '`dpctl add-flow`' já apresentado anteriormente.

Alternativamente podemos iniciar um controlador remoto localizado fora da VM MiniNet, pode ser na própria máquina física ou em qualquer outra máquina conectada na internet. Esta opção pode ser conveniente quando se têm alguma versão personalizada de algum módulo de um controlador específico. Exemplo:

[8]Na última versão estável do MiniNet, não se utiliza mais os *switches* em espaço de *kernel*, por motivos de segurança e incompatibilidades com o sistema. O padrão agora utilizado é o *switch* OVS.

```
$ sudo mn --controller=remote --ip=[ip do controlador] --port[porta
de escuta do controlador (padrão = 6634)]
```

No próximo capítulo será apresentado melhor um exemplo de uso dessa opção.

6.3 Outros Comandos e Ferramentas da VM Mini-Net

A VM MiniNet possui o OpenFlow Wireshark dissector pré-instalado. O Wireshark é extremamente útil para a análise de mensagens do protocolo OpenFlow bem como depuração de um modo geral. Para iniciar o Wireshark crie uma nova janela SSH e certifique-se que ela executará o servidor X, explicado anteriormente.

O comando para abrir o Wireshark é:

```
$ sudo wireshark &
```

Você provavelmente vai receber uma mensagem de aviso por usar o Wireshark com acesso `root`. Ignore o aviso e pressione `OK`. Clique em `Capture`→`Interfaces` no menu. Abrirá uma nova janela e clique no botão `Start` ao lado da interface `lo`.

Neste momento será apresentado uma lista com diversos pacotes em constante incremento. Crie um filtro para controle do tráfego OpenFlow. Na caixa de filtro coloque '`of`' e pressione '`apply`' para filtrar o tráfego OpenFlow.

Primeiro execute o MiniNet com um controlador remoto:

```
$ sudo mn --controller=remote
```

Agora inicie o controlador de referência em outra janela SSH:

```
$ controller ptcp:
```

Isso inicia um controlador simples, que atua como um *learning switch* sem instalar nenhuma entrada na tabela de fluxo.

Será exibida diversas mensagens Wireshark, começando com '`Hello`'. Clique em '`Features Reply`'. Abrirá uma nova janela, clique no triangulo ao lado de '`OpenFlow Protocol`' para exibir os campos de mensagens. Clique em no triangulo ao lado de

'`Switch Features`' e veja as capacidades do *datapath*. Sinta-se a vontade, explore os recursos do Wireshark.

Quadro 6.1: Tipos de mensagens OpenFlow apresentadas ao iniciar o controlador.

Mensagem	**Tipo**	**Descrição**
Hello	Controlador→Switch	após o TCP *handshake*, o controlador envia seu número de versão para o switch.
Hello	Switch→Controlador	o *switch* responde com seu número de versão suportado.
Features Request	Controlador→Switch	o controlador pede ao *switch* para ver quais portas estão disponíveis.
Set Config	Controlador→Switch	O controlador pede ao *switch* para enviar expirações de fluxo
Features Reply	Switch→Controlador	as respostas do *switch* com uma lista de suas portas e outras características.
Port Status	Switch→Controlador	permite ao *switch* informar ao controlador mudanças de velocidades de porta ou de conectividade. Ignorar este, pois parece que está com problemas nesta versão do OpenFlow.

Quando há muitos *switches*, a análise do Wireshark pode ficar um pouco confusa, pois varias mensagens repetidas vão aparecer. Mas isso não acontece quando criamos a topologia com um único *switch*. Note que as mensagens de `Echo Request/Echo Reply` são mensagens para manter a conexão entre os *switches* e o controlador ativas.

Agora vamos ver as mensagens geradas em *pings* na rede. Modifique seu filtro no Wireshark:

```
of && (of.type != 3) && (of.type != 2)
```

Agora na janela do MiniNet execute um *ping*:

```
mininet> h2 ping -c1 h3
```

Na janela Wireshark, é possível ver uma série de novos tipos de mensagens:

Primeiro vemos a mensagem `Packet-In` com uma requisição ARP. Logo depois vemos a mensagem `Packet-Out` que corresponde a ação tomada pelo controlado no *switch*, esta ação no caso é o envio em *broadcast* pelas portas do *switch* com exce-

Quadro 6.2: Tipos de mensagens OpenFlow apresentadas ao realizar um ping.

Mensagem	**Tipo**	**Descrição**
Packet-In	Switch→Controlador	o pacote foi enviado ao controlado pois não possui uma entrada correspondente a esse pacote na tabela de fluxo do *switch*.
Packet-Out	Controlador→Switch	controlador envia um pacote de uma ou mais portas do *switch*.
Flow-Mod	Controlador→Switch	adiciona uma entrada na tabela de fluxo do *switch*.
Flow-Expired	Switch→Controlador	um fluxo expirou após um período de inatividade.

ção a porta que chegou o pacote. Quando o pacote de resposta chega ao *switch* este já conhece o caminho para 'h2' e neste momento ele envia a mensagem `Flow-Mod` criando uma entrada na tabela de fluxo do *switch*. Nos próximos *pings*, não há envolvimento do controlador, a exibição das mensagens no Wireshark devem parar por ai até a validade do fluxo vencer e apresentar a mensagem `Flow-Expired`.

Caso tenha qualquer outra dúvida ou dificuldade em de utilizar o Wireshark procure pela documentação em `http://www.wireshark.org/`.

Outra possibilidade interessante no uso do MiniNet é a possibilidade de se executar *scripts* em um *host*, dentro do CLI ou em sua inicialização. Você pode executar em um *host*. Exemplo:

```
mininet> h2 config_script
```

Ou mesmo ser utilizada no próprio CLI:

```
mininet> source my_cli_script
```

Ou utilizá-la na inicialização do CLI:

```
$ sudo mn --pre my_cli_script
```

Todas esses comandos evita a digitação repetitiva no CLI em seus testes e depurações de sua aplicação de rede. É possível criar um arquivo com múltiplos comandos

do CLI para determinado fim específico. Note que nos exemplos acima o arquivo deve ficar na pasta raiz do MiniNet, no entanto pode-se especificar o local no comando.

Um exemplo de um *script* é apresentado a seguir:

```
py "Configuring network"
h3 ifconfig h3-eth0 10.0.1.2/24
h4 ifconfig h4-eth0 10.0.1.3/24
h5 ifconfig h5-eth0 10.0.2.2/24
h3 route add default gw 10.0.1.1
h4 route add default gw 10.0.1.1
h5 route add default gw 10.0.2.1
py "Current network:"
net
dump
```

O MiniNet também fornece APIs para a criação e a configuração de Redes Definidas por Software. Alguns exemplos de uso pode ser encontradas em `mininet/examples`, suas descrições encontram-se em `mininet/examples/README`. Destaca-se duas API's gráficas, não muito utilizáveis mais que consistem em ideias interessantes. Exemplo:

```
$ sudo ./ mininet/examples/miniedit.py
```

Abrirá uma janela gráfica com a qual é possível desenhar uma topologia de maneira simples e fácil, no entanto essa API não é funcional e não possível se fazer absolutamente nada com ela, feche-a e ela encerrará o CLI do MiniNet (Figura 6.1).

Outro exemplo é:

```
$ sudo ./ mininet/examples/consoles.py
```

Abrirá uma janela gráfica com uma grade de janelas de console, uma para cada nó, e permite a interação com o acompanhamento de cada console, incluindo monitoração com um gráfico e comandos como o *iperf*, *pings* (Figura 6.2).

Também é possível criar API's próprias para MiniNet. Uma API simples basicamente utiliza-se das bibliotecas `mininet.net`, `mininet.node`, `mininet.cli`. A primeira contém os módulos necessários para se montar a rede virtual propriamente dita, a segunda por sua vez contém os módulos para se criar todos os nós virtuais dessa rede e a terceira serve para que se possa executar o CLI do MiniNet. A seguir é apresentada uma API simples com comentários.

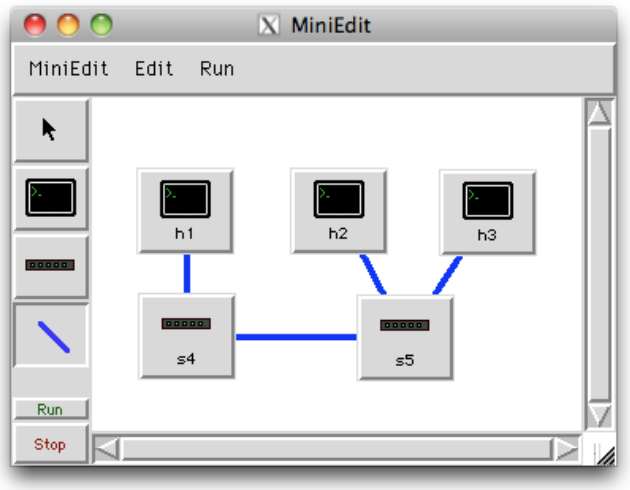


Figura 6.1: MiniNet API miniedit.py.

Figura 6.2: MiniNet API consoles.py.

```
"""
Este e o exemplo de como criar uma API simples instanciando um objeto
    MiniNet
montando sua topologia e executando o CLI
"""
# importando as classes necessarias

from mininet.net import Mininet
from mininet.node import Controller
from mininet.cli import CLI
from mininet.log import setLogLevel, info # para informacoes de log

def minhaAPI():

  # Criando a rede:
```

```
15      net = Mininet( controller=Controller ) # instanciando um objeto da
            classe Mininet
16
17      info( '*** Adding controller\n' ) # imprimindo log info
18      net.addController( 'c0' ) # adicionando um controlador
19
20      info( '*** Adding hosts\n' )
21      h1 = net.addHost( 'h1', ip='10.0.0.1' ) # adicionando um no host e
            colocando seu ip
22      h2 = net.addHost( 'h2', ip='10.0.0.2' )
23
24      info( '*** Adding switch\n' )
25      s3 = net.addSwitch( 's3' ) # adicionando um switch openflow
26
27      info( '*** Creating links\n' )
28      h1.linkTo( s3 ) # criando o link de h1(host1) para s3(switch openflow)
29      h2.linkTo( s3 )
30
31      info( '*** Starting network\n')
32      net.start() # iniciando a rede
33
34      info( '*** Running CLI\n' )
35      CLI( net ) # iniciando o CLI com a rede criada
36
37    # o codigo para aqui ate que se saia do CLI
38
39      info( '*** Stopping network' )
40      net.stop() # para a rede criada
41
42  if __name__ == '__main__':
43      setLogLevel( 'info' )
44      minhaAPI()
```

Para executar esta API, crie um arquivo com o nome de `minhaAPI.py` no diretório `mininet/examples` e execute-o na máquina virtual da seguinte maneira:

```
$ sudo python -O /mininet/examples/minhaAPI.py
```

Pode-se criar API's de diversas maneiras. Este trabalho contribui com a criação de uma API's de inicialização do MiniNet com interface gráfica. Foi utilizando a biblioteca `Tkinter` do Python (Figura C.1). Seu código pode ser visto no Apêndice C.

6.4 Teste o Simulador

Agora que se sabe como executar o MiniNet, se conhece os principais comandos utilizados dentro do CLI e se conhece quais as principais opções de inicialização do

CLI, é possível utilizá-los em conjunto, misturando os diversos tipos de opções na inicialização, por exemplo:

```
$ sudo mn --custom ~/mininet/custom/topo-2sw-2host.py --topo mytopo
--mac --switch ovsk --controller remote --ip=10.0.2.2 --port=6634
--pre ~/mininet/custom/my_cli_script
```

Caso tenha alguma dúvida consulte a documentação em: `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/MininetDocumentation`.

Capítulo 7

Implementações Desenvolvidas

Entendendo a estrutura do POX e do MiniNet, agora é possível desenvolver componentes no POX e utilizá-los no MiniNet. Para se fazer isso é necessário ter instalado o MiniNet conforme apresentado no Capítulo 6 e ter instalado o POX na própria máquina virtual VM MiniNet (recomendado) ou em uma máquina que esteja hospedando a máquina virtual VM MiniNet.

Este capítulo mostra como instalar o POX e como desenvolver componentes funcionais em vários cenários de redes. Com as implementações desenvolvidas neste capítulo o pesquisador poderá entender melhor como funciona a estrutura de uma rede SDN podendo então implementar seus próprios componentes que atendam seus requisitos de pesquisa.

Os códigos das implementações desenvolvidas neste capítulo foram utilizados no simulador MiniNet, mas poderiam ser aplicados em redes SDN reais, equivalentes as simuladas, sem a necessidade de adaptações.

7.1 Instalando e Configurando o POX

Para instalar o POX abra um terminal no computador (Linux ou MAC) ou uma janela SSH conectada a máquina virtual VM MiniNet (preferencialmente se for usar Windows) e baixe o código do POX no repositório do POX no GitHub dessa forma:

```
$ git clone http://github.com/noxrepo/pox
$ cd pox
```

Caso queira utilizar a versão beta faça:

```
/pox$ git checkout betta
```

Pronto, o POX está instalado. Navegue por suas pastas para entender melhor sua estrutura, apresentada no Capítulo 5. A versão beta é a mais atual versão do POX e está em constante modificação pelos desenvolvedores.

7.2 Executando um Componente do POX no MiniNet

Agora que o MiniNet e POX estão instalados, abra dois terminais, um para o POX e outro para o MiniNet (Se o POX estiver instalado na maquina virtual VM MiniNet deverão ser abertos dois terminais SSH).

No terminal SSH da máquina virtual VM MiniNet feche todos os controladores:

```
$ sudo killall controllers
```

Depois limpe o MiniNet com o comando:

```
$ sudo mn -c
```

Se o POX estiver instalado na máquina hospedeira do VM MiniNet execute o comando a seguir para identificar o IP da máquina hospedeira, caso contrário pule esta etapa.

```
$ sudo route
```

Irá aparecer algo como:

```
Kernel IP routing table
Destination     Gateway   Genmask         Flags Metric Ref Use Iface
default         10.0.2.2  0.0.0.0         UG    0      0      0 eth1
10.0.2.0        *         255.255.255.0   U     0      0      0 eth1
192.168.56.0    *         255.255.255.0   U     0      0      0 eth2
```

Procure a linha que começa com '*default*' e anote o endereço IP da coluna '*Gateway*'.

Para iniciar o MiniNet execute o comando a seguir, se o POX estiver sido instalado na VM MiniNet (recomendado) apague a opção `--ip`.

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller=remote
--ip=[endereço ip anotado]
```

Caso o endereço anotado termine com `.1` não será possível se conectar com um controlador remoto fora da VM MiniNet, substitua o `.1` por `.2`. Pronto a rede SDN virtual está criada e o POX poderá conectar-se a ela remotamente em uma máquina hospedeira ou na própria máquina virtual VM MiniNet (recomendado).

No terminal destinado ao POX entre no diretório `pox/` ("`cd pox`") e execute algum componente do POX (seção 5.3), por exemplo o `forwarding.l2_learning` da seguinte maneira:

```
$./pox.py log.level --DEBUG forwarding.l2_learning
```

Se tudo correr bem será possível executar um `pingall` com sucesso no terminal do MiniNet. Pronto, a rede está funcionando e o componente `l2_learning` está controlando essa rede com sucesso. Já será possível executar aquele componente criado na Seção 5.12 e entender melhor seu funcionamento. A próxima seção explicará qual o cenário foi criado neste exemplo e quais os cenários e componentes que serão criados.

7.3 Cenários de Simulação

Na seção anterior foi criada a rede virtual SDN e executado um componente pronto do POX. O cenário criado na seção anterior será o mesmo cenário que será utilizado nos dois primeiros componentes (HUB e SWITCH).

Este cenário consiste de um único *switch* OpenFlow (`s1`) ligado a três *hosts* (`h2`, `h3`, `h4`) como mostra a Figura 7.1.

O segundo cenário que será apresentado consistirá de três *switches* OpenFlow (`s5`, `s6`, `s7`) ligados em árvore nos quais os dois *switches* finais (folha da árvore) terão dois *hosts* cada (`h1`, `h2`, `h3`, `h4`). Este cenário será utilizado no terceiro e quarto componente (SWITCHES, FIREWALL). Este cenário pode ser visto na Figura 7.2.

O terceiro e último cenário também terá três *switches* OpenFlow (`s0`, `s1`, `s2`) só que dessa vez interligados entre si funcionando como roteadores, nos quais cada um terá sua rede composta por dois *hosts* ([`h0`, `h1`], [`h2`, `h3`], [`h4`, `h5`]). Utilizaremos este cenário para o quinto e sexto componente (ROUTER, Especifico). Este cenário pode ser visto na Figura 7.3.

Os componentes serão especificados nas próximas seções.

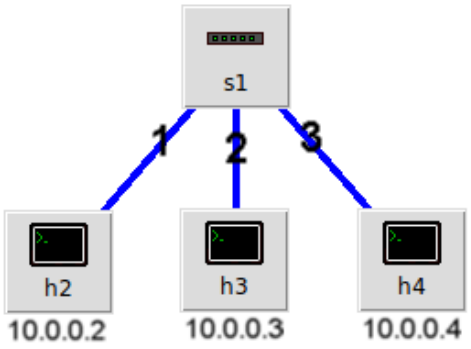


Figura 7.1: Primeiro Cenário de Testes.

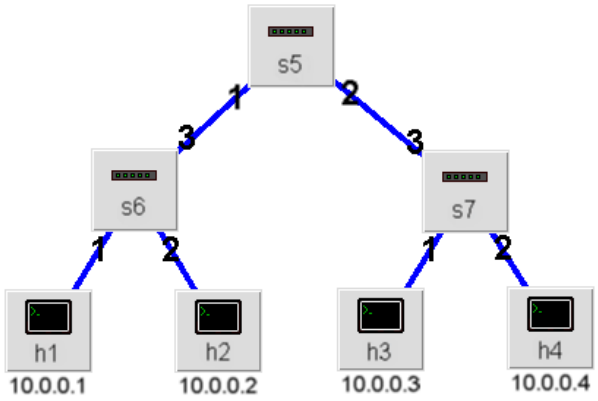


Figura 7.2: Segundo Cenário de Testes.

7.3.1 Componente HUB

Esta seção apresenta como construir um componente que age como um HUB, replicando os pacotes recebidos e enviado para todas as suas portas de saída. Para este componente será utilizado o primeiro cenário. É interessante observar como é estruturado um componente no POX e como o POX trabalha com os pacotes e as mensagens OpenFlow. No Apêndice B na seção B.2 é apresentado o código deste componente com comentários.

O componente inicia-se no método `launch` e registra um ouvinte de mensagens do núcleo do POX. Não é registrado nenhuma classe ou método no núcleo do POX, como nos exemplos anteriores, apenas utiliza-se do método `addListenerByName` já registrado no núcleo do POX pelo módulo `openflow`, executado automaticamente, para ouvir as mensagens do tipo `ConnectionUp`[1] da rede. Quando essas mensagens chegam, é executado o método `start_switch` que recebe o evento da conexão com o comutador, e a partir daí é instanciado a classe HUB.

[1]Essa mensagens são enviadas ao controlador quando um comutador entra na rede

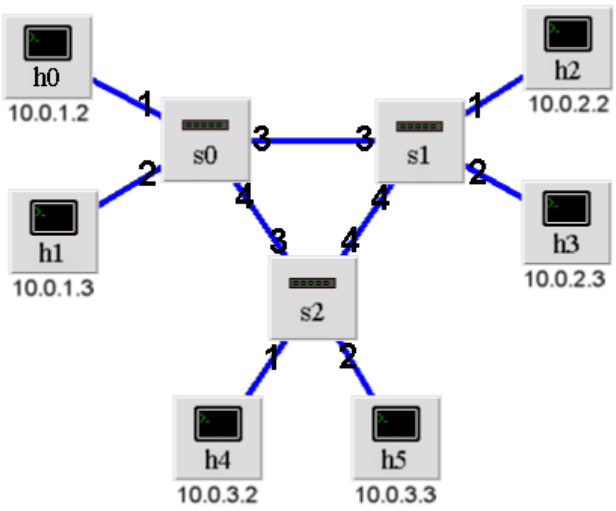


Figura 7.3: Terceiro Cenário de Testes.

A classe HUB por sua vez inicia a conexão com o comutador e registra tratadores (*handle*) de mensagens OpenFlow por meio do método `addListeners`. Dessa forma é possível ouvir as mensagens `PacketIn`[2] vindas ao controlador. Assim quando o comutador recebe pacotes, que não possuem entradas em sua tabela de fluxo, ele os envia ao controlador, que no caso deste componente faz as ações que o método `_handle_PacketIn` faz. Neste componente este método chama o método `act_like_hub` para que o comutador haja como um hub.

Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `hub.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG hub
```

Na VM MiniNet em um terminal SSH, cria-se a rede (primeiro cenário):

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller=remote
--ip=[endereço ip do host]
```

Caso o POX seja executado na própria VM MiniNet, como recomendado, basta remover a opção `--ip`. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf`, veja o comportamento dos pacotes com o WireShark (Seção 6.3).

[2]Mensagens que o controlador recebe quando o comutador recebe um pacote que não tem entradas em sua tabela de fluxo

7.3.2 Componente SWITCH

Este componente haje como um *switch* de auto aprendizagem de camada dois que aprende o caminho a medida que os pacotes chegam no comutador. Em outras palavras é um simples *switch* que aprende os caminhos de seus *hosts*. No Apêndice B na seção B.3 é apresentado o código deste componente com comentários.

O interessante deste componente é observar como se trabalha com mensagens `ofp_flow_mod` que instala entradas na tabela de fluxo de um comutador OpenFlow. Esse tipo de comportamento faz com que o *switch* execute o comutamento de pacotes muito mais rápido do que o componente anterior.

No HUB as mensagens enviadas eram do tipo `ofp_packet_out`, que são mensagens utilizadas pelo que o controlador para instruir o *switch* aonde enviar os pacotes. Esse tipo de comportamento é pior em relação ao desempenho, pois o controlador participa da decisão de encaminhamento de todos pacotes que chegam no *switch*.

Este componente é semelhante ao anterior quanto a sua estrutura, a diferença é que quando os pacotes chegam ao controlador, e não possuem entradas na tabela de fluxo referente a este pacote, o componente anota o endereço físico (MAC) da fonte do pacote em uma tabela e associa este endereço a porta de chegada do *switch* (linha 26), em seguida é verificado se o endereço de destino físico do pacote está nessa tabela, se estiver, será instalado o fluxo no comutador, se não estiver na tabela ele instrui o comutador a inundar o pacote em todas as demais portas, como o HUB faz. Sempre que um novo pacote passar pelo *switch*, a dupla formada pela porta de entrada e MAC de origem será gravada na tabela, caracterizando o mecanismo de auto-aprendizagem.

Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `switch.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG switch
```

O componente pode ser testado no cenário de teste do componente anterior. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf` veja a diferença do resultado deste último. Veja o comportamento dos pacotes com o WireShark e os tipos de mensagens OpenFlow.

7.3.3 Componente SWITCHES

O componente anterior funciona apenas em cenários de rede com um *switch*. Isso ocorre pois a tabela em que se anota os endereços físicos e suas portas de origem é uma tabela global do componente, logo quando se há mais do que um *switch* essa tabela não faz sentido, pois portas iguais de diferentes *switch* podem encaminhar pacotes para lugares diferentes na rede. No Apêndice B na seção B.4 é apresentado o código deste componente com comentários.

Faz-se necessário então anotar também os números de identificação de cada *switch* nesta tabela global, para que quando o pacote for comparado na tabela, este

deve ser comparado com o *switch* que questiona o controlador sobre o caminho de pacotes (`_handle_PacketIn`). Alternativamente pode-se colocar uma tabela exclusiva para cada *switch* quando estes instanciam a classe *Switch* (linha 75). O próximo componente faz isso.

Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `switches.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG switches
```

Na VM MiniNet em um terminal SSH, cria-se a rede (segundo cenário):

```
$ sudo mn --topo tree,2 --mac --switch ovsk --controller=remote
--ip=[endereço ip do host]
```

Caso o POX seja executado na própria VM MiniNet, como recomendado, basta remover a opção `--ip`. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf`, veja o comportamento dos pacotes com o WireShark.

7.3.4 Componente FIREWALL

Este componente mostra como é possível instalar regras especificas nos *switches* para que hajam como um firewall ou mesmo negando serviços específicos. O proposito deste componente é mostrar como é possível criar regras especificas para os comutadores da rede. No Apêndice B na seção B.5 é apresentado o código deste componente com comentários.

A estrutura do componente segue o mesmo padrão dos outros. A diferença está na tabela que guarda os valores dos endereços físicos e portas, dessa vez ela não e global, e por esse motivo não se faz mais necessário guardar o ID do *switch*, pois cada *switch* terá sua tabela quando instanciar a classe (linha 137).

Assim que um *switch* se conecta na rede é gerado uma mensagem de `ConnectionUp` que faz com que o *switch* instancie a classe `Firewall` que na sua inicialização cria a tabela `MAC/PORT` (linha 20) e chama o método `install_firewall_rules` que instala regras determinadas no *switch*.

Essa regras foram definidas como exemplo para este cenário, poderiam ser feitas outras regras ou de maneiras diferentes. A primeira regra faz com que pacotes que tenham a porta TCP 80 sejam enviados ao controlador, para que ele decida o que fazer. A segunda regra envia todos os pacotes que tenha o IP de destino igual a 10.0.0.2 para o controlador. Em ambos os casos os *switches* enviam esses pacotes por meio de uma mensagem `PacketIn` ao controlador. Este, por sua vez, decidir o caminho no método `_handle_PacketIn` do componente utilizado (linha 58), que neste exemplo retorna sem realizar ação alguma.

Dessa forma pacotes que tenham a porta TCP 80 e pacotes que tenham o endereço de destino IP 10.0.0.2 não serão comutados nessa rede. Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `firewall.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG firewall
```

O cenário de teste utilizado neste componente é o mesmo cenário do componente anterior. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf`. Para testar se os pacotes com a porta TCP 80 serão comutados, abra um `xterm` para dois nós (Ex.: `xterm h1 h3`) e faça em um deles:

```
# iperf -s -p 80
```

No outro nó faça:

```
# iperf -c [endereço ip do outro nó] -p 80
```

É possível ver que o trafego não irá fluir, fazendo o teste com outra porta é possível ver que fluirá normalmente. É necessário observar que, neste cenário especifico, pacotes com destino ao `h2` não chegarão, logo `pings` a `h2` ou oriundos de `h2` também não funcionaram. Para maiores detalhes veja o comportamento dos pacotes com o WireShark.

7.3.5 Componente ROUTER

Este componente é um pouco mais complexo que os outros. Ele utiliza o terceiro cenário. Seu cenário é composto de três sub-redes com os seguintes endereços IP: `10.0.1.1/24`, `10.0.2.1/24` e `10.0.3.1/24` com a mascara de sub-rede 255.255.255.0. Para criar esta rede, pode-se utilizar os métodos apresentados no Capítulo 6, página 77. Para configurar os IP's da rede e o *gateway* pode-se utilizar o comando `ifconfig` com as opções necessárias. Alternativamente é possível criar a topologia no MiniNet executando o código apresentado no Apêndice B na seção B.1 com comentários. O código deste componente encontra-se no Apêndice B na seção B.5.

Este componente deve implementar o protocolo de roteamento ARP[3] na rede anotando em uma tabela (linha 35), para cada *switch*, o endereço físico (MAC) da fonte

[3] **Address Resolution Protocol** ou **ARP** é um protocolo usado para encontrar um endereço da camada de enlace (MAC) a partir do endereço da camada de rede (IP).

com seu endereço rede (IP) e sua porta de chegada. Com essas informações é possível rotear os pacotes na rede.

Quando um *switch* se conecta na rede ele instancia a classe `Router` que inicia duas tabelas, uma ARP (linha 35) e outra de IP (linha 42). A primeira é a tabela usada no protocolo ARP, já a segunda serve para anotar os IP's que já passaram neste *switch*. Essa tabela serve para garantir que IP's que não existem na rede não fiquem sendo inundados pelo protocolo APR em um laço sem fim.

Quando pacotes chegam ao comutador é verificado se este pacote é do tipo ARP ou IP. Se for ARP é executado o método arp_responder (linha 45) que executa o protocolo de roteamento ARP, se for IP é executado o método `route` (linha 124) que faz o roteamento de pacotes de acordo com a tabela ARP. Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `router.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG router
```

O componente pode ser testado no terceiro cenário de teste já apresentado. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf` e veja os resultados. Veja o comportamento dos pacotes com o WireShark e os tipos de mensagens OpenFlow.

7.3.6 Componente ESPECÍFICO

Este componente é um exemplo de como controlar o funcionamento da rede de uma maneira bem específica. Pode servir como base para entender as funcionalidades do OpenFlow usando o controlador POX. Agora que já foi apresentado os componentes anteriores, este componente torna-se bem simples de entender. Seu cenário de teste é o cenário do componente anterior e só funcionará neste cenário configurado dessa maneira. No Apêndice B na seção B.7 é apresentado o código deste componente com comentários.

Cada pacote que chega ao controlador é inspecionado e atribuído a ele uma rota, dando uma entrada na tabela de fluxo do comutador. Dessa forma, cada pacote novo que chega ao *switch* é enviado ao controlador onde o método `_handle_PacketIn` os recebe e chama o método `install_rules` que instala uma regra para este pacote de acordo com uma lógica definida.

O funcionamento do fluxo de pacotes nesse componente funciona da seguinte maneira:

1 - O enlace que liga os *switches* `s0` e `s1` é exclusivo para pacotes que tenham a porta de destino e de origem `TCP 1234`, nenhum outro pacote deve passar por esse enlace.

2 - Todos os demais pacotes são comutados normalmente na rede sem passar no enlace que liga diretamente os *switches* `s0` e `s1`.

Com essas regras na rede, os pacotes que tenham portas `TCP 1234` irão ter prioridade na comutação de pacotes entre os *switches* `s0` e `s1` já que todos os outros

pacotes deverão passar por `s2`. Este caso e todos os outros servem como exemplo de como controlar uma rede SDN utilizando o POX.

Para testar este componente, basta criar o arquivo citado dentro da pasta `pox/ext` com o nome `especifico.py` e executar da seguinte maneira:

```
$ ./pox.py log.level --DEBUG especifico
```

O componente só poderá ser testado no cenário de testes do componente anterior. Execute testes no CLI do MiniNet como `ping`, `pingall` ou `iperf` e veja os resultados. Analise os resultados do `iperf` com a porta `TCP 1234` e com outras portas nos diversos nós. Veja o comportamento dos pacotes com o WireShark e os tipos de mensagens OpenFlow.

7.4 Resultados Esperados

Neste capítulo foi desenvolvido alguns componentes no POX e foram utilizados e testados na plataforma de simulação de Redes Definidas por Software, o MiniNet. Esses componentes servem como base para a criação de cenários de testes em redes reais para o paradigma de Redes Definidas por Software.

É possível agora observar na prática como funciona o protocolo OpenFlow, como funciona a troca de mensagens entre os comutadores e o controlador, além é claro de entender a estrutura de um controlador de rede SDN e poder utilizá-lo para o controle de uma rede SDN.

É necessário entender o funcionamento do POX antes de iniciar a execução destes componentes. Além disso, é necessário entender o funcionamento do MiniNet para testes em ambientes simulados. A partir de então é possível implementar estes componentes em ambientes reais com comutadores e *hosts* físicos.

Com a compreensão desses componentes é possível implementar diversos casos particulares para testes de novas pesquisas na área de redes, ou mesmo para o uso como ferramenta didática em disciplinas de redes. Apresentar os conceitos e protocolos de redes de computadores por meio de ferramentas como o MiniNet, traz para o aluno uma visão programática de como os desafios são resolvidos no núcleo da rede. Além disso, ao possibilitar que o aluno experimente um protocolo, ou ainda, permitir que ele possa estendê-lo adicionando novas funcionalidades, o educador estará preparando seus alunos para os desafios reais das redes de computadores.

Capítulo 8

Discussão e Conclusão

As redes de computadores constituem, hoje, um serviço de primeira necessidade para a sociedade. No entanto, os mesmos motivos que a mantiveram até hoje, sua arquitetura baseada no serviço de transferência fim-a-fim e a pilha de protocolos TCP/IP, também são responsáveis pelas limitações vivenciadas hoje no tangente a escalabilidade, mobilidade e gerenciamento. Contudo, para atender tais serviços, as redes de computadores foram evoluindo por meio de ”adaptações”, funcionalidades estas não previstas em sua arquitetura original. Essas ”adaptações” dificultaram e inibiram a criação de novas propostas de arquiteturas para seu núcleo, visto que essas propostas devem ser compatíveis com as ”adaptações” da rede não afetando o funcionamento da mesma. Estudos argumentam pela necessidade de se desenvolver uma nova arquitetura para a Internet, a Internet do Futuro [2, 17].

Os diversos estudos voltados para a Internet do Futuro apontam para um modelo pluralista, na qual a infraestrutura de rede deve ser capaz de dar suporte a diversas redes em paralelo, cada uma com sua pilha de protocolos e gerenciamento próprio, garantindo alta flexibilidade e permitindo a inovação no núcleo da rede. Uma tecnologia que permite o desenvolvimento desse modelo é a virtualização de redes. A virtualização de redes consiste no desenvolvimento de uma camada de abstração sobre a infraestrutura física da rede. Essa tecnologia, utilizada em projetos como a GENI [26], propõe uma nova forma de organizar as redes de computadores, utilizando máquinas virtuais para representar comutadores virtuais na rede. Entretanto, esse tipo de virtualização enfrenta o desafio de se obter uma implementação eficiente do plano de dados nos comutadores, uma vez que nas redes atuais a otimização do *hardware* de encaminhamento de pacotes atingiu patamares de desempenho significativos, impossíveis de se equiparar em *software*.

Nesse contexto, o paradigma de Redes Definidas por Software (SDN) surge como uma alternativa para o desenvolvimento do modelo de virtualização de redes. Por darem acesso às tabelas de encaminhamento de forma programável, as redes SDN oferecem uma nova forma de virtualizar a rede, não mais com máquinas virtuais independentes para cada elemento da rede virtual, mas com o particionamento dos espaços de endereçamento dessas tabelas. Dessa forma, visões abstratas da rede podem ser oferecidas a cada operador que deseje implementar um novo serviço sobre a rede física.

8.1 Desafios de Pesquisa

O paradigma SDN é considerado o modelo mais promissor de arquitetura para a Internet do Futuro [14]. Sua capacidade de implementação gradual na arquitetura de rede é um dos principais motivos que o tornam o modelo eleito pela academia e industria, além do fato de prover uma visão global e o gerenciamento centralizado da rede.

O paradigma SDN abre uma gama de novas possibilidades para a pesquisa em redes de computadores. A Seção 3.7 apresentou diversos contextos de aplicação que podem oferecer novos enfoques e possibilidades de evolução. Deve-se considerar que o paradigma SDN ainda é algo novo e, portanto, existe um longo caminho para consolidar este paradigma.

Pode-se classificar alguns dos principais desafios, conforme tratado a seguir:

A visão da rede - O paradigma SDN provê uma visão global da rede disponível pelo controlador da rede. Os controladores atuais ainda não apresentam essa visão como um elemento de sua estrutura básica, ela pode ser constituída com base nas informações derivadas das mensagens recebidas. Espera-se que os controladores ofereçam esse elemento por meio da noção do ”Modelo de Objetos da Rede”, ou Network Object Model (NOM) [14]. Essa abstração permitirá o desenvolvedor de aplicações SDN implementar o controle de ações em função do grafo de rede e não mais ao redor dos eventos de chegada de mensagens OpenFlow.

Sistema operacional de Redes - A noção do sistema operacional de rede como um ambiente de execução que estabelece a ligação entre as aplicações SDN e a rede física pode ir além das implementações de um controlador de rede SDN atual. Este conceito pode abrir caminhos para uma nova forma de se pensar em redes de computadores, não mais em termos de artefatos tecnológicos, mas de princípios organizacionais abstratos como ocorre na área de Sistema Operacionais. Pode-se utilizar questões como escalonamento de processos, algorítimos de sincronização e estruturas de dados como princípios básicos para essa nova geração de controladores de rede, fazendo com que eles realmente se comportem como sistemas operacionais.

O encaminhamento - Apesar do OpenFlow ser apresentado como a principal interface de acesso aos mecanismos de encaminhamento de cada comutador, as redes SDN não se restringem a ele. Soluções deveriam oferecer uma interface de encaminhamento pela malha da rede, independentemente da interface particular de cada elemento. Assim, aplicações poderiam se concentrar em suas características específicas, fazendo uso de primitivas de encaminhamento fim-a-fim na rede.

Depuração - A inclusão que o paradigma SND traz de ter diversos níveis de abstração entre as aplicações de rede e sua infra-estrutura física, cria a necessidade de regras que traduzam o que é expressado na linguagem e interface de aplicações e como os conceitos associados são implementados na rede física. Nesse

processo, existe a possibilidade de que a configuração da rede física não represente o que era previsto nos níveis superiores. Criar mecanismos que possam identificar quando isso ocorre, e qual é sua causa, ainda é um problema em aberto nos mecanismos de depuração do paradigma SDN.

Distribuição - A forma de controle centralizado que o paradigma SDN oferece não precisa, necessariamente, ser implementada de forma centralizada. Dessa forma, projetos de controladores distribuídos fisicamente mas centralizados logicamente ainda são possíveis linhas de pesquisa que podem ser exploradas nesse contexto. Vale lembrar que a noção de visão global da rede como algo centralizado em SDN é uma visão lógica e o paradigma não exige que essa visão deva ser, obrigatoriamente, implementada de forma centralizada. Requisitos de desempenho ou tolerância a falhas poderiam levar a decisões de distribuição dessa visão.

8.2 Considerações Finais

Foi observado que o paradigma de Redes Definidas por Software é uma grande promessa para a arquitetura da Internet do Futuro. Seu potencial apenas está começando a ser explorado, mas já é grande o interesse da área acadêmica e da industrial. O fato de existirem no mercado diversos dispositivos comerciais com o protocolo OpenFlow habilitado confirma que esse paradigma é bastante promissor.

Este trabalho apresentou o contexto histórico das redes de computadores, sua infraestrutura e seus problemas para atender todos os requisitos hoje estabelecidos pelas novas aplicações de rede. Foi observado que a arquitetura atual das redes de computadores está saturada e têm dificuldades para atender as novas demandas do mercado. Embora sejam identificadas muitas propostas de rede para atender esses requisitos, estas não são bem aceitas no mercado comercial por não serem testadas em ambientes reais. Isso deve-se ao fato das redes atuais estarem ossificadas com diversas ”adaptações” em sua arquitetura original. A comunidade acadêmica estuda formas de renovar essa arquitetura, com as propostas de Internet do Futuro. Dentre os estudos mais promissores está o paradigma de Redes Definidas por Software.

O presente trabalho buscou apresentar uma visão dos aspectos teóricos e práticos envolvidos no desenvolvimento de pesquisas em Redes Definidas por Software, apresentando suas características e elementos essenciais para o desenvolvimento nessa área, com ênfase no controlador de rede. O controlador de rede é o elemento que tem o papel essencial em qualquer iniciativa SDN, uma vez que o *software* que define as redes são desenvolvidos com base nos recursos oferecidos. Diversas opções de controladores foram descritas e algumas aplicações básicas foram apresentadas no texto. O OpenFlow, que consiste na principal interface associada ao paradigma SDN, foi apresentado e descrito. Suas características formam um dos elementos mais motivadores para o paradigma apesar de não ser a única forma de se implementar uma SDN. Foi, então, apresentado o POX, um controlador recentemente lançado e especialmente desenvolvido para o ambiente de pesquisa e ensino. Com a modelagem

e estrutura apresentadas se torna fácil desenvolver e construir aplicações SDN que utilizem a interface OpenFlow.

O cenário de testes utilizado foi feito por meio do simulador de Redes Definidas por Software MiniNet. Vale ressaltar que os testes desenvolvidos no texto podem ser utilizados em redes físicas reais. É mostrado como implementar componentes no POX e como testá-los no MiniNet. Todos as implementações desenvolvidas neste trabalho servem como base para pesquisa e modelagem de ensino na área de redes de computadores.

Considerando o sucesso do paradigma SDN e todos os desafios identificados neste texto, observa-se que existe um vasto campo para o desenvolvimento de novos projetos de pesquisa focando em SDN. Este trabalho serve como um ponta pé inicial para o desenvolvimento de novas pesquisas na área, seja como ferramenta para estudo ou como introdução didática para esse novo paradigma de redes.

8.3 Trabalhos Futuros

O paradigma SDN ainda é novo e muitos desafios de pesquisa ainda estão por vir. Como sugestão para trabalhos futuros são apresentados os seguintes itens:

- Difundir esse paradigma em laboratório e realizar testes em elementos físicos.
- Utilizar as implementações desenvolvidas no texto em uma rede real.
- Expandir futuramente esse modelo para toda a universidade.

Com a implementação deste paradigma na infraestrutura da universidade, os pesquisadores poderiam desenvolver pesquisas de novas técnicas de rede, com avaliações em redes reais sem prejuízo no tráfego de produção da universidade.

Referências

[1] A. Al-Shabibi and M. McCauley. POX-Wiki, 2013. https://openflow.stanford.edu/display/ONL/POX+Wiki.

[2] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. *IEEE Computer Society Press Los Alamitos, CA, USA*, 38(4):34–41, 2005.

[3] M. F. Caetano, A.V. Barbosa, P.S. Barreto, and J.L. Bordim. Theoretical Maximum Throughput of IEEE 802.11g Networkss. *IJCSNS International Journal of Computer Science and Network Security*, 11(4):136–146, 2011.

[4] Z. Cai, A. L. Cox, and T. S. Eugene Ng. Maestro: A System for Scalable OpenFlow Control. Technical report, Rice University, 2010.

[5] K. L. Calvert, W. K. Edwards, N. Feamster, R. E. Grinter, Y. Deng, and X. Zhou. Instrumenting Home Networks. *Instrumenting home networks. SIGCOMM Comput. Com-mun. Rev.*, 1(41):84–89, 2011.

[6] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO*, 8(10):1–8, 2010.

[7] V. G. Cerf and R. E. Kahn. A Protocol for Packet Network Intercommunication. *Institute of Electrical and Electronics Engineers (IEEE)*, 22(5):1–13, May 1974.

[8] Inc Cisco Systems. IP Telephony: The five nines story. Technical report, Cisco Systems, Inc, 2002.

[9] D. Clark, R. Braden, K. Sollins, J. Wroclawski, and D. Katabi. New Arch: Future Generation Internet Architecture. *M. I. O. T. C. L. F. C. SCIENCE*, 235(74):30–76, 2004.

[10] D. F. Contessa and E. R. Polina. Gerenciamento de Equipamentos Usando o Protocolo SNMP. Technical report, Departamento de Pesquisa e Desenvolvimento - CP Eletrônica S.A, 2010.

[11] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. *ACM SIGCOMM workshop on Future directions in network architecture*, 1(25):258–266, September 2003.

[12] D. Erickson. The Beacon Controller, 2012. https://openflow.stanford.edu/display/Beacon/Home/.

[13] C. Dixon et al. The Home Needs an Operating System (and an App Store). *In Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets*, 6(10):1–18, 2010.

[14] D. Guedes et al. 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*, 30(4):160–210, 2012.

[15] J. C. Mogul et al. DevoFlow: Cost-Effective Flow Management for High Performance Enterprise Networks. *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets*, 10(1):1–6, 2010.

[16] K. K. Ram et al. sNICh: Efficient Last Hop Networking in the Data Center. *Proceedings of the 6th ACM IEEE Symposium on Architectures for Networking and Communications Systems, ANCS*, 10(12):1–26, 2010.

[17] N. Fernandes et al. Virtual Networks: Isolation, Performance, and Trends. *Annals of Telecommunications*, 66(5-6):339–355, 2010.

[18] N. Gude et al. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun.*, 1(38):105–110, 2008.

[19] N. McKeown et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2(38):69–74, March 2008.

[20] R. Sherwood et al. Carving Research Slices Out of Your Production Networks with OpenFlow. *SIGCOMM Computer-Communication Networks*, 1(40):129–130, 2009.

[21] R. Sherwood et al. FlowVisor: A Network Virtualization Layer. Technical report, OPENFLOW-TR, 2009.

[22] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. *In Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI*, 6(10):1–14, 2010.

[23] N. Feamster. Outsourcing Home Network Security. *In Proce-edings of the 2010 ACM SIGCOMM workshop on Home networks, HomeNets*, 1(10):37–42, 2010.

[24] Floodlight. The Floodlight Controller, 2012. http://floodlight.openflowhub.org/.

[25] N. Foster, R. Harrison, and M. L. Meola. Frenetic: A High-Level Language for Openflow Networks. *In Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow, PRESTO*, 6(10):1–6, 2010.

[26] GENI. Global Environment for Network Innovations, 2012. http://www.geni.net/.

[27] IEEE. Institute of Electrical and Electronics Engineers, 2012. http://www.ieee.org/.

[28] IETF. Internet Engineering Task Force, 2012. http://www.ietf.org/.

[29] ISO. International Organization for Standards, 2012. http://www.iso.org.

[30] C. Kamienski, D. Mariz, D. Sadok, and S. Fernandes. Arquiteturas de Rede para a Próxima Geração da Internet. *Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2005*, 9(3):1–50, 2005.

[31] J. Kurose and K. Ross. *Computer Networking, A Top-Down Aproach 5ª edição*. Addison-Wesley, 2010.

[32] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. *SIGCOMM: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 8(16):175–187, 2000.

[33] D. M. F. Mattos. Xenflow: Um Sistema de Processamento de Fluxos Robusto e Eficiente para Redes Virtuais. Technical report, Deparamento de Eletrônica e de Computação. UFRJ - Universidade Federal do Rio de Janeiro, 2011.

[34] M. McCauley. NOXRepo, 2013. http://www.noxrepo.org.

[35] M. Moreira, N. Fernandes, L. Costa, and O. Duarte. Internet do Futuro: Um Novo Horizonte. *Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2009*, 8(1):1–59, 2009.

[36] NSF. National Science Foundation, 2012. http://www.nsf.gov/.

[37] OASIS. Organization for the Advancement of Structured Information Standards, 2012. http://www.oasis-open.org/.

[38] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. *In Proceedings of the Eighth ACM Workshop on Hot Topics in Networks*, 8(1):1–6, 2010.

[39] L. Roberts. Living Internet, Lawrence Roberts Manages The ARPANET Program, 2008. http://www.livinginternet.com/i/ii_roberts.htm.

[40] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, and M. F. Magalhães. OpenFlow e redes definidas por software: um novo paradigma de controle e inovação em redes de pacotes. *Cad. CPqD Tecnologia*, 7(1):1–6, July 2010.

[41] SNAC. Simple Network Access Control, 2012. http://www.openflow.org/wp/snac/.

[42] Stanford University. *OpenFlow Switch Specification*, 1.1.0 edition, February 2011.

[43] E. W. Takarabe. Sistema de Controle Distribuído em Redes de Comunicação. Master's thesis, Escola Politêcnica da Universidade de São Paulo, 2009.

[44] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks*. Pearson, Boston, 2011.

[45] Trema. Trema: full-stack OpenFlow framework for Ruby and C, 2012. http://trema.github.com/trema/.

[46] W3C. World Wide Web Consortium, 2012. http://www.w3.org/.

[47] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. *SIGCOMM Computer Communication Review*, 38(4):231–242, 2008.

Apêndice A

Descrição dos Elementos do POX

A seguir é apresentado a descrição dos componentes básicos do POX:

Quadro A.1: Descrição dos Componentes nativos do POX.

Componente	**Descrição**
py	Este componente inicializa um interpretador Python para a depuração interativa. Este componente é executado automaticamente se a opção "`--no-cli`" estiver desabilitada.
forwarding.l2_learning	Este componente faz com que os *switches* OpenFlow funcionem como *switches* de aprendizagem de camada dois. No entanto ele instala fluxos extras para conexões com o mesmo origem/destino, por exemplo, conexões com origens distintas e destinos iguais possuem dois fluxos distintos.
forwarding.l2_pairs	O mesmo que o "forwarding.l2_learning", no entanto este é mais simples e mais correto, ele instala regras baseadas puramente em endereços MAC nao levando a consideração origem e destino.
forwarding.l3_learning	Este componentes nao chega a ser um roteador, mas opera em camada três. Ele é uma espécie de *switch* de camada três, operando exatamente como um *switch* de camada dois com a diferença entre os endereçamentos de MAC para IP.
forwarding.l2_multi	Este componente haje como os outros *switches* de aprendizagem, a diferença está em ambientes com múltiplos *switches*. Neste componente quando um *switch* aprende um endereço MAC todos os outros das rede também aprendem, para isso este componente utiliza outro componente o `openflow.discovery`.

Quadro A.2: Continuação do Quadro A.1.

Componente	**Descrição**
openflow.spanning_tree	Este componente usa componentes de descoberta para construir uma visão da topologia da rede e transformar esta topologia em uma árvore. Seu objetivo é desativar inundação em portas de *switches* que não estão nesta árvore. Isso faz com que topologias que tenham laços não realize transmissões inúteis na rede. Este componente assemelha-se ao efeito produzido pelo o Protocolo *Spanning Tree*[1], mas realizando isso de uma forma diferente.
web.webcore	Este componente inicia um servidor web dentro do processo do POX. Dessa forma outros componentes podem interagir com ele fornecendo conteúdos estáticos e dinâmicos.
messenger	Este é um componente que fornece o uma interface para o POX realizar conexões por meio de mensagens bidirecionais baseadas em JSON[2].
openflow.of_01	Este é o componente que se comunica com os *switches* OpenFlow 1.0. Normalmente ele é iniciado por padrão quando não é especificada a opção `--no-openflow`. As vezes é necessário chama-lo manualmente quando se quer modificar suas opções, por exemplo, qual interface ou porta ele está ligado.
openflow.discovery	Este componente descobre qual é a topologia da rede por meio de mensagens de controle entre os *switches OpenFlow.*
openflow.debug	Este componente analisa as trocas de mensagens OpenFlow entre o controlador e os *switches* da rede. Assemelha-se as análises do tcpdump ou do WireShark (discutido melhor depois) sendo mais sintética.
openflow.keepalive	Este componente faz com que o POX envie solicitações periódicas de eco para *switches* conectados. Alguns *switches*, após um longo periodo de silêncio, podem assumir que a ligação está inativa ou que houve uma perda de conectividade com o controlador e sendo assim este irá se desligar. Dessa forma este componente evita este problema.

[1] **Spanning Tree Protocol** (**STP**) é um protocolo para equipamentos de rede que permite resolver problemas de laços em redes comutadas cuja topologia introduza anéis nas ligações.

[2] JavaScript Object Notation, é um formato leve para intercâmbio de dados computacionais que pode ou não utilizar-se de JavaScript. `http://www.json.org/`.

Quadro A.3: Continuação do Quadro A.2.

Componente	**Descrição**
misc.pong	Este componente é um exemplo simples de como se trabalhar com o monitoramento e o envio de pacotes ICMP. Sua execução faz com que todos os `pings` realizados tenha uma resposta `pong`.
misc.arp_responder	O mesmo que o anterior com a diferença que responde requisições `ARP` a partir de uma lista de entradas estáticas.
misc.packet_dump	Um simples componente que exibe informações de pacotes. É como executar um `tcpdump` em um *host*.
misc.dns_spy	Este componente monitora as respostas DNS e guarda seus resultados.
misc.of_tutorial	Este componente é um tutorial simples que mostra como transformar uma *hub* em um *switch* de aprendizagem de camada dois. Este é o tutorial utilizado no OpenFlow Tutorial[3].
misc.mac_blocker	Este componente é feito para ser usado ao lado de outros componentes de encaminhamento. Aparecerá uma interface gráfica que permite bloquear endereços MAC. Este é um exemplo que demonstra o uso de eventos de bloqueio e como criar componentes com interface gráfica.
log	O POX utiliza-se de mensagens de `log` do Python que por sua vez podem ser configuradas nas das linhas de comando, por exemplo, é possível configurar `logs` que incluam data e hora.
log.color	Este componente faz com que o log seja exibido com cores de diferenciadas no terminal, muito interessante quando se quer fazer depurações.
samples.pretty_log	Este componente personaliza as saídas do `log` para uma melhor visualização.
tk	Este componente é utilizado para ajudar na construção de interfaces gráficas baseadas no POX.

Os Quadros A.4, A.5 e A.6 mostram uma análise de algumas das principais classes da biblioteca `pox.lib.packet`, maiores detalhes sobre as outras classes podem ser encontradas nas classes especificas no diretório `pox/lib/packet`.

[3]OpenFlow Tutorial `http://www.openflow.org/wk/index.php/OpenFlow_Tutorial`

Quadro A.4: Análise da classe Ethernet do POX.

Atributos	**Descrição**
`dst`	Endereço MAC de destino
`src`	Endereço MAC da fonte
`type(int)`	Define o tamanho do campo do pacote *ethernet*.
`effective_ethertype(int)`	Delimita o tamanho do cabeçalho da VLAN
Constantes	**Descrição**
`IP_TYPE`	Define um pacote do tipo IP (Exemplo no código da pagina 57)
`ARP_TYPE`	Define um pacote do tipo ARP
`RARP_TYPE`	Define um pacote do tipo RARP
`VLAN_TYPE`	Define um pacote do tipo VLAN
`LLDP_TYPE`	Define um pacote do tipo LLDP
`JUMBO_TYPE`	Define um pacote do tipo JUMBO
`QINQ_TYPE`	Define um pacote do tipo QINQ

Quadro A.5: Análise da classe IP do POX.

Atributos	**Descrição**
`dstip`	Endereço IP de destino
`srcip`	Endereço IP da fonte
`tos(int)`	O histórico campo de tipo de serviço TOS(8 bits).
`id(int)`	Campo de identificação
`flags(int)`	Campos de flag
`ttl(int)`	Tempo de vida do pacote
`protocol(int)`	Número do protocolo IP
`csum(int)`	Checksum do pacote IP
Constantes	**Descrição**
`ICMP_PROTOCOL`	Define o protocolo do pacote como ICMP
`TCP_PROTOCOL`	Define o protocolo do pacote como TCP
`UDP_PROTOCOL`	Define o protocolo do pacote como UDP
`DF_FLAG`	Define o bit DF(*don't fragment*)
`MF_FLAG`	Define o bit MF(*more fragments*)

Quadro A.6: Análise da classe TCP do POX.

Atributos	**Descrição**
`dstport`	Define a porta TCP de destino
`srcport`	Define a porta TCP de saída da fonte
`seq(int)`	Número de sequencia
`ack(int)`	número do ACK
`off(int)`	Indica a posição à qual pertence o fragmento atual (*offset*)
`flags(int)`	flags do pacote TCP
`csum(int)`	Checksum do pacote TCP
`options`	Lista de opções da classe `tcp_opt`
`win(int)`	Tamanho da janela TCP
`urg(int)`	Flag urg de pacote TCP urgente
`FIN(bool)`	Flag FIN do pacote
`SYN(bool)`	Flag SYN do pacote
`RST(bool)`	Flag RST do pacote
`ACK(bool)`	Flag ACK do pacote
Constantes	**Descrição**
`FIN_flag`	Define os bits correspondente a flag FIN
`SYN_flag`	Define os bits correspondente a flag SYN

Os Quadros A.7 e A.8 mostram os argumentos de construção da classe `Timer`. O Quadro A.9 mostra seus métodos.

Quadro A.7: Argumentos de construção da classe Timer.

Atributos	**Descrição**
`timeToWake`	É a quantidade de tempo de espera (em segundos) para chamar a função em `callback` isso se `absoluteTime = False` ou é o tempo específico para chamar `callback` isso se `absoluteTime = True`
`callback`	A função que será chamada
`absoluteTime`	Quando `False` o `timeToWake` define o tempo de espera do `callback`, quando `True` `timeToWake` é um tempo específico no futuro. Não é possível que este valor seja `True` quando o `Timer` é recorrente
`recurring`	Quando `False` o callback dispara uma única só vez, se `True` o `callback` é recorrente pelo tempo definido em `timeToWake`

Quadro A.8: Continuação do Quadro A.7.

Atributos	Descrição
`args, kw`	Esses são os argumentos passados para a função do `callback`
`scheduler`	Qual será o agendador de tarefas utilizado (`scheduler()`). Se for `None` será executado o *scheduler* padrão
`started`	Quando `True`, o `Timer` é executado automaticamente.
`selfStoppable`	Quando `True`, se o retorno de um `Timer` recorrente for `False` ele cancela a *thread*

Quadro A.9: Métodos da classe Timer.

Métodos	Descrição
`cancel()`	Para a *thread*, não chama ele novamente
`run()`	Executa a *thread*

A seguir são apresentados os diversos eventos para o controle de mensagens OpenFlow:

Quadro A.10: Eventos fornecidos do módulo OpenFlow no POX.

Evento	Descrição
`ConnectionUp`	Evento gerado quando um comutador OpenFlow têm uma conexão estabelecida com o controlador
`ConnectionDown`	Evento gerado quando um comutador OpenFlow perde uma conexão com o controlador
`PortStatus`	Evento gerado quando há alguma mudança no status das portas dos comutadores OpenFlow
`FlowRemoved`	Evento gerado quando algum comutador remove alguma entrada em sua tabela de fluxos
`PacketIn`	Evento gerado quando algum comutador recebe um pacote que não pertença a nenhum fluxo em sua tabela de fluxos. O pacote é enviado ao controlador.

Quadro A.11: Continuação do Quadro A.10.

Evento	Descrição
`ErrorIn`	Evento gerado quando um comutador envia uma mensagem de erro OpenFlow ao controlador. É possível utilizar o método `asString` para ver a representação do erro OpenFlow
`BarrierIn`	Evento gerado em resposta a uma mensagem OpenFlow *Barrier*
`RawStatsReply`	Mostra eventos estatísticos de forma geral. Não é muito interessante pois mostram todas as mensagens OpenFlow
`StatsReply`	É a superclasse para todas as outras classes de eventos estatísticos. Os outros eventos estatísticos mostram grupos de mensagens OpenFlow separadamente. Eventos estatísticos são basicamente utilizados para depuração e gerenciamento da rede.
`SwitchDescReceived`	Classe de evento que herda `StatsReply`. Mostra os status do comutador
`FlowStatsReceived`	Classe de evento que herda `StatsReply`. Mostra os status dos fluxos do comutador
`AggregateFlowStatsReceived`	Classe de evento que herda `StatsReply`. Mostra alterações de fluxo do comutador
`TableStatsReceived`	Classe de evento que herda `StatsReply`. Mostra o status da tabela de fluxo do comutador
`PortStatsReceived`	Classe de evento que herda `StatsReply`. Mostra o status das portas do comutador
`QueueStatsReceived`	Classe de evento que herda `StatsReply`. Mostra o status das filas do comutador

Os Quadros A.12 e A.13 mostram os atributos da classe `of.ofp_packet_out`:

Quadro A.12: Definição dos atributos da classe ”of.ofp_packet_out” do POX.

Atributo	Tipo	Padrão	Descrição
`buffer_id`	`int/None`	`None`	ID do *buffer* no qual o pacote é armazenado.
`in_port`	`int`	`OFPP_NONE`	A porta no qual o pacote chegou

Quadro A.13: Continuação do Quadro A.12.

Atributo	Tipo	Padrão	Descrição
actions	list of ofp_action	[]	Lista de ações que o comutador deve realizar. Essas ações são outros tipos de mensagens OpenFlow
data	bytes/ethernet/ ofp_packet_in	(nada)	Os dados a ser enviado, ou nenhum se o pacote estiver no *buffer* do comutador (buffer_id diferente de None)

Os Quadros A.14 e A.15 mostram os atributos da classe of.ofp_flow_mod:

Quadro A.14: Definição dos atributos da classe ”of.ofp_flow_mod” do POX.

Atributo	Tipo	Padrão	Descrição
cookie	int	0	ID desta regra (entrada da tabela de fluxo)
command	int	OFPFC_ADD	Qual o comando OFPFC_XXXX. Pode-se usar os seguinte valores: ADD (adicionar uma regra),MODIFY (modificar uma regra),MODIFY_STRICT (modificar regras com valores universais),DELETE (eliminar uma regra) e DELETE_STRICT (eliminar regras com valores universais)
idle_timeout	int	OFP_FLOW_ PERMANENT	A regra irá expirar por esse período (em segundos) se não for usada. O valor padrão significa que a regra não irá expirar
hard_timeout	int	OFP_FLOW_ PERMANENT	A regra irá expirar por esse período (em segundos). O valor padrão significa que a regra não irá expirar
priority	int	OFP_DEFAULT_ PRIORITY	A prioridade da regra. Números mais altos terá prioridades maiores
buffer_id	int	None	O *buffer* do fluxo de dados
out_port	int	OFPP_NONE	Este campo é utilizado para os comandos de eliminação de regras

Quadro A.15: Continuação do Quadro A.14.

Atributo	Tipo	Padrão	Descrição
flags	int	0	Algumas flags de controle. OFPFF_SEND_FLOW_REM (envia mensagens de fluxo removido ao controlador quando uma regra expira), OFPFF_CHECK_OVERLAP (verifica se não há sobreposição de entrada na tabela, se existir envia uma mensagem ao controlador) e OFPFF_EMERG (mensagem de emergência se acontecer algo ao comutador)
actions	list of ofp_action	[]	As ações que são tomadas pelo fluxo
match	ofp_match	(nada)	A estrutura da regra, define o fluxo

O Quadro A.16 a seguir mostra os atributos da classe `of.ofp_match`:

Quadro A.16: Definição dos atributos da classe ”of.ofp_match” do POX.

Atributo	Descrição
in_port	Muda o número da porta que o pacote chegou
dl_src	MAC de origem
dl_dst	MAC de destino
dl_vlan	ID da VLAN
dl_vlan_pcp	Prioridade da VLAN
dl_type	Tamanho do pacote *Ethernet*
nw_tos	Define os bits DF e MF
nw_proto	Qual o tipo de protocolo da camada 3
nw_src	IP de origem
nw_dst	IP de destino
tp_src	TCP/UDP de origem
tp_dst	TCP/UDP de destino

O Quadro A.17 a seguir mostra as classes de ação do POX e suas descrições:

Quadro A.17: Definição de algumas das classes ”ofp_action” do POX.

Classe	**Descrição**
`of.ofp_action_output`	Encaminha pacotes para uma porta física ou virtual. Valores possíveis: OFPP_IN_PORT (envia o pacote de volta pela porta de origem), OFPP_TABLE (executa ações especificas no fluxo, só se aplica a mensagens de `ofp_packet_out`), OFPP_NORMAL (processo normal de comutação, para fluxo de produção), OFPP_FLOOD (para todas as portas, exceto a de entrada e portas com *floods* desativados), OFPP_ALL (para todas as portas, exceto a de entrada), OFPP_CONTROLLER (para o controlador), OFPP_LOCAL (para suas portas locais) ou OFPP_NONE (para nenhum lugar, descarta o pacote)
`ofp_action_enqueue`	Encaminha o pacote para uma fila designada no comutador para implementar QoS rudimentar. Note que o comportamento de filas não é de responsabilidade do OpenFlow, esta responsabilidade é somente do comutador
`ofp_action_vlan_vid`	Se o pacote não tiver um cabeçalho VLAN, essa ação adiciona uma e define seu ID com o valor especificado e sua prioridade para 0. Caso já tenha um cabeçalho VLAN ele apenas modifica sua ID
`ofp_action_vlan_pcp`	O mesmo que o anterior, só que se já tiver um cabeçalho VLAN ele apenas modifica sua prioridade
`ofp_action_dl_addr`	Usado para definir a origem ou o destino endereço MAC
`ofp_action_nw_addr`	Usado para definir a origem ou o destino endereço IP
`ofp_action_nw_tos`	Defina o campo TOS de um pacote IP
`ofp_action_tp_port`	Usado para definir a porta TCP/UDP de origem ou destino

Para maiores informações procure a POX Wiki [1].

Apêndice B

Códigos das Implementações Desenvolvidas

B.1 Código do Terceiro Cenário

A seguir é apresentado o código para a criação do terceiro cenário de testes com comentários:

```
from mininet.net import Mininet
from mininet.node import Controller, OVSKernelSwitch, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.util import createLink

def createStaticRouterNetwork():
    info( '*** Criando a Rede do Cenario 3 (Exemplo)\n' )
    # Cria uma rede vazia com o controlador remoto
    net = Mininet(controller=RemoteController, switch=OVSKernelSwitch)
    #adiciona um controlador
    net.addController('c0')
    # Cria os nos da rede.
    h0 = net.addHost('h0')
    h1 = net.addHost('h1')
    s0 = net.addSwitch('s0')
    h2 = net.addHost('h2')
    h3 = net.addHost('h3')
    s1 = net.addSwitch('s1')
    h4 = net.addHost('h4')
    h5 = net.addHost('h5')
    s2 = net.addSwitch('s2')
    # Cria os links da rede.
    h0int, s0int = createLink(h0, s0)
    h1int, s0int = createLink(h1, s0)
    h2int, s1int = createLink(h2, s1)
    h3int, s1int = createLink(h3, s1)
    h4int, s2int = createLink(h4, s2)
    h5int, s2int = createLink(h5, s2)
    s0pint, s1pint = createLink(s0, s1)
    s0pint, s2pint = createLink(s0, s2)
```

```
    s1pint, s2pint = createLink(s1, s2)
    # Configurando os ips e as interfaces
    s0.setIP(s0int, '10.0.1.1', 24)
    h0.setIP(h0int, '10.0.1.2', 24)
    h1.setIP(h1int, '10.0.1.3', 24)
    s1.setIP(s1int, '10.0.2.1', 24)
    h2.setIP(h2int, '10.0.2.2', 24)
    h3.setIP(h3int, '10.0.2.3', 24)
    s2.setIP(s2int, '10.0.3.1', 24)
    h4.setIP(h4int, '10.0.3.2', 24)
    h5.setIP(h5int, '10.0.3.3', 24)
    s0.setIP(s0pint, '10.0.1.255', 24)
    s1.setIP(s1pint, '10.0.2.255', 24)
    s2.setIP(s2pint, '10.0.3.255', 24)
    info( '*** Estado da rede:\n' )
    for node in s0, h0, h1, s1, h2, h3, s2, h4, h5:
        info( str( node ) + '\n' )
    # Inicia o CLI do mininet
    net.start()
    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    createStaticRouterNetwork()
```

codigos/cenario3.py

Para executar este código, basta criar o arquivo acima na pasta `mininet/custom` com o nome cenario3.py e compilar o código Python dentro da pasta na VM Mininet:

```
$ cd mininet/custom
sudo python -O cenario3.py
```

Após isso será criada a rede do terceiro cenário de testes.

B.2 Código do HUB

A seguir é apresentado o código do componente HUB com comentários.

```
from pox.core import core
from pox.lib.util import dpidToStr
from pox.lib.revent import *
import pox.openflow.libopenflow_01 as of

log = core.getLogger()

class Hub (object):
```

```
9     def __init__ (self, connection): #iniciador da classe
10      # a connection e necessaria para que se possa enviar as mensagens
            openflow
11      self.connection = connection
12      # faz com que se possa ouvir o evento PacketIn
13      connection.addListeners(self)
14
15    def act_like_hub (self, packet_in):
16      """
17      Implementa o HUB -- envia o pacote para todas suas portas menos a de
            entrada
18      """
19      msg = of.ofp_packet_out()# instrui em qual porta o pacote deve sair
20      msg.in_port = packet_in.in_port# atribuindo o parametro in_port da msg
21      #verifica se o pacote esta no buffer do switch
22      if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
23          # se o pacote estiver no buffer do switch utilize o pacote do
                buffer
24          msg.buffer_id = packet_in.buffer_id
25      else:
26          # o pacote nao esta no buffer do switch
27          if packet_in.data is None:#verifica se o pacote que chegou a
                controlador esta vazio
28              # o pacote esta vazio, nada a fazer
29              return
30          # o pacote que chegou ao controlador nao esta vazio, e nao esta no
                buffer do switch
31          # entao se atribui o dado do pacote que chegou ao controlador na
                msg
32          msg.data = packet_in.data
33      log.debug("Enviando para todas as portas")
34      # Adiciona a acao na msg openflow que sera mandada ao switch
35      action = of.ofp_action_output(port = of.OFPP_FLOOD)
36      msg.actions.append(action)
37      # envia a mendagem para o switch
38      self.connection.send(msg)
39
40    def _handle_PacketIn (self, event):
41      """
42      Esculta as mensagens PacketIn da rede
43      """
44      log.debug("Chegou um pacote no switch %s", dpidToStr(event.dpid))
45      packet_eth = event.parsed # Faz a analise dos dados do pacote do evento
46      #e necessario fazer o event.parsed para que se possa ter acesso aos
47      #cabecalhos do pacote
48      if not packet_eth.parsed:
49        log.warning("Pacote foi ignorado pois nao ha cabecalhos")
50        return
51
52      packet_in = event.ofp # A mensagem openflow do evento
53      self.act_like_hub(packet_in)
54
55  def launch ():#inicia-se o componente va para a linha 63
56
57      def start_switch (event): #evento obtido metodo iniciado
```

```
58          #observe que a classe hub recebe um parametro, que e a conexao do
                switch
59          #esse parametro e necessario para que se mantenha a conexao com o
                switch
60          #e possa enviar mensagens openflow para ele
61          Hub(event.connection)# enviamos a conexao do evento para a classe
62
63      # quando ouvimos o evento ConnectionUp executamos o metodo start_switch
64      # para isso, temos que registrar um ouvinte no nucleo do pox:
65      core.openflow.addListenerByName("ConnectionUp", start_switch)#
            registramos o ouvinte para obter o evento
```

codigos/hub.py

B.3 Código do SWITCH

A seguir é apresentado o código do componente SWITCH com comentários.

```
1  from pox.core import core
2  from pox.lib.util import dpidToStr
3  from pox.lib.revent import *
4  import pox.openflow.libopenflow_01 as of
5
6  log = core.getLogger()
7  """
8  e criada uma dicionario(lista) global para que o componente
9  identifique os pares ''endereco MAC'' ''porta de switch''
10 note que essa tabela e global da rede, isso nao faria sentido se
11 tivessemos mais que um switch
12 """
13 mac_to_port = {}
14 class Switch (object):
15   def __init__ (self, connection): #iniciador da classe
16     # a connection e necessaria para que se possa enviar as mensagens
           openflow
17     self.connection = connection
18     # faz com que se possa ouvir o evento PacketIn
19     connection.addListeners(self)
20
21   def act_like_switch (self, packet_in, packet):
22     """
23     Implementa o Switch de auto aprendizagem -- aprende os caminhos de
           todos os seus nos
24     instalando as entradas na tabela de fluxo do switch (desempenho muito
           melhor que o HUB).
25     """
26     mac_to_port[str(packet.src)] = packet_in.in_port
27
28     ###Enviando os pacotes atraves da instalacao de regras
29     if str(packet.dst) in mac_to_port:
30         port = mac_to_port[str(packet.dst)]
31         log.debug("Enviando pacote para a porta %d" %(port))
```

```
32          msg = of.ofp_flow_mod()
33          #criando o fluxo, esse fluxo determina apenas o endereco MAC de
                destino
34          msg.match = of.ofp_match(dl_dst = packet.dst)
35          log.debug("instalando fluxo de %s.%i" % (packet.dst, port))
36          """
37          alternativamente podemos fazer assim:
38          msg.match = of.ofp_match.from_packet(packet)
39          log.debug("instalando fluxo de %s.%i para %s.%i" % (packet.src,
                packet_in.in_port, packet.dst, port))
40          porem fluxo ficara mais especifico, instalara um fluxo
41          da fonte MAC x para o destino MAC y
42          """
43          msg.idle_timeout = 10
44          msg.hard_timeout = 30
45          if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
46              msg.buffer_id = packet_in.buffer_id
47          else:
48              if packet_in.data is None:
49                  return
50              msg.data = packet_in.data
51          msg.actions.append(of.ofp_action_output(port = port))
52          self.connection.send(msg)
53      else:
54          port = of.OFPP_FLOOD
55          log.debug("Enviando para todas as portas")
56
57          msg = of.ofp_packet_out()
58          msg.in_port = packet_in.in_port
59          if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
60              msg.buffer_id = packet_in.buffer_id
61          else:
62              if packet_in.data is None:
63                  return
64              msg.data = packet_in.data
65          # Adiciona a acao na msg openflow que sera mandada ao switch
66          action = of.ofp_action_output(port = port)
67          msg.actions.append(action)
68          # envia a mendagem para o switch
69          self.connection.send(msg)
70
71  def _handle_PacketIn (self, event):
72      """
73      Esculta as mensagens PacketIn da rede
74      """
75      log.debug("Chegou um pacote no switch %s", dpidToStr(event.dpid))
76      packet_eth = event.parsed
77      if not packet_eth.parsed:
78        log.warning("Pacote foi ignorado pois nao ha cabecalhos")
79        return
80
81      packet_in = event.ofp # A mensagem openflow do evento
82      self.act_like_switch(packet_in, packet_eth)#agora enviamos os
            cabecalhos dos pacotes
83     #para que o switch possa gravar seus enderecos em uma tabela
84
```

```
85 def launch ():
86
87     def start_switch (event):
88         Switch(event.connection)# enviamos a conexao do evento para a
               classe
89     core.openflow.addListenerByName("ConnectionUp", start_switch)
```

codigos/switch.py

B.4 Código do SWITCHES

A seguir é apresentado o código do componente SWITCHES com comentários.

```
1  from pox.core import core
2  from pox.lib.util import dpidToStr
3  from pox.lib.revent import *
4  import pox.openflow.libopenflow_01 as of
5
6  log = core.getLogger()
7  """
8  dessa vez as chaves do dicionario sao tuplas de endereco MAC e ID do switch
9  para que cada switch tenha sua propria regra de encaminhamento
10 """
11 mac_dpid_to_port = {}
12 class Switches (object):
13   def __init__ (self, connection):
14     self.connection = connection
15     connection.addListeners(self)
16
17   def act_like_switches (self, packet_in, packet, dpid):
18     """
19     Implementa varios Switches de auto aprendizagem -- e necessario que se
           tenha
20     o id de cada switch para que possa ser instalado regras especificas do
           switch.
21     """
22     mac_dpid_to_port[(str(packet.src),dpidToStr(dpid))] = packet_in.in_port
           #a chave e uma tupla
23     if (str(packet.dst),dpidToStr(dpid)) in mac_dpid_to_port:
24         port = mac_dpid_to_port[(str(packet.dst),dpidToStr(dpid))]
25         log.debug("Enviando pacote para a porta %d" %(port))
26         msg = of.ofp_flow_mod()
27         #criando o fluxo, esse fluxo determina apenas o endereco MAC de
               destino
28         msg.match = of.ofp_match(dl_dst = packet.dst)
29         log.debug("instalando fluxo de %s.%i no switch %s" % (packet.dst,
               port, dpidToStr(dpid)))
30
31         msg.idle_timeout = 10
32         msg.hard_timeout = 30
33         if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
34             msg.buffer_id = packet_in.buffer_id
```

```
35          else:
36              if packet_in.data is None:
37                  return
38              msg.data = packet_in.data
39          msg.actions.append(of.ofp_action_output(port = port))
40          self.connection.send(msg)
41      else:
42          port = of.OFPP_FLOOD
43          log.debug("Enviando para todas as portas")
44          msg = of.ofp_packet_out()
45          msg.in_port = packet_in.in_port
46          if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
47              msg.buffer_id = packet_in.buffer_id
48          else:
49              if packet_in.data is None:
50                  return
51              msg.data = packet_in.data
52          # Adiciona a acao na msg openflow que sera mandada ao switch
53          action = of.ofp_action_output(port = port)
54          msg.actions.append(action)
55          # envia a mendagem para o switch chamou o evento
56          self.connection.send(msg)
57
58    def _handle_PacketIn (self, event):
59      """
60      Esculta as mensagens PacketIn da rede
61      """
62      log.debug("Chegou um pacote no switch %s", dpidToStr(event.dpid))
63      packet_eth = event.parsed
64      if not packet_eth.parsed:
65        log.warning("Pacote foi ignorado pois nao ha cabecalhos")
66        return
67
68      packet_in = event.ofp # A mensagem openflow do evento
69      dpid = event.dpid#passa o id do switch que gerou o evento
70      self.act_like_switches(packet_in, packet_eth, dpid)#agora e enviado o
            id de cada switch
71
72  def launch ():
73
74      def start_switch (event):
75          Switches(event.connection)# enviamos a conexao do evento para a
                classe
76      core.openflow.addListenerByName("ConnectionUp", start_switch)
```

codigos/switches.py

B.5 Código do FIREWALL

A seguir é apresentado o código do componente FIREWALL com comentários.

```
1  from pox.core import core
```

```
2  from pox.lib.util import dpidToStr
3  from pox.lib.revent import *
4  import pox.openflow.libopenflow_01 as of
5  import pox.lib.packet as pkt
6
7  log = core.getLogger()
8
9  class Firewall (object):
10   def __init__ (self, connection): #iniciador da classe
11     # a connection e necessaria para que se possa enviar as mensagens
          openflow
12     self.connection = connection
13     # faz com que se possa ouvir o evento PacketIn
14     connection.addListeners(self)
15     """
16     note que agora a variavel mac_to_port nao e mais global, pertence a
          classe
17     portanto quando cada switch se conecta ao controlador sao criadas
          instancias diferentes
18     que por sua vez contem 'mac_to_port' diferentes nao necessitando do
          dpid para diferenciar
19     """
20     self.mac_to_port = {}
21     self.install_firewall_rules()#instala as regras de firewall ao se
          conectar um switch
22
23   def install_firewall_rules (self):
24     """
25     Assim que os switches se conectam ao controlador as regras de firewall
26     aqui definidas ja sao instaladas nos switches, de maneira que o pacote
          que
27     se enquadre nessas regras ja tera seu destino definido, seja ele o
          descarte,
28     enviado para uma porta ou se enviado ao controlador(PacketIn).
29     Neste ultimo caso o '_handle_PacketIn' recebera este pacote
30     que devera ser tratado la.
31     Existe varias maneiras de se implementar um firewall, esta e so uma
          maneira
32     """
33     ###Primeira Regra
34     log.debug("Primeira regra instalada no Switch %s, pacotes com destino a
           porta tcp 80 enviados para o controlador"
35               %(dpidToStr(self.connection.dpid)))
36     msg = of.ofp_flow_mod()#cria a mensagem para ser enviada ao switch
37     #e criada o fluxo que desejamos que seja pego no firewall
38     msg.match = of.ofp_match(dl_type = pkt.ethernet.IP_TYPE, nw_proto = pkt
          .ipv4.TCP_PROTOCOL, tp_dst = 80)
39     #tem que ser um pacote ethernet que tenha um pacote IP dentro que tenha
40     #um pacote TCP dentro com destino na porta 80
41     msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))#
          acao enviar ao controlador
42     self.connection.send(msg)
43     ###
44     ###Segunda Regra
45     log.debug("Segunda regra instalada no Switch %s, pacotes com destino ao
           IP 10.0.0.2 enviados para o controlador"
```

```
46                  %(dpidToStr(self.connection.dpid)))
47          msg = of.ofp_flow_mod()
48          msg.match = of.ofp_match(dl_type = pkt.ethernet.IP_TYPE, nw_dst = "
                10.0.0.2")
49          #esse fluxo e mais simples, um pacote ethernet que tenha um pacote IP
                dentro com
50          #destino ao endereco IP 10.0.0.2
51          msg.actions.append(of.ofp_action_output(port = of.OFPP_CONTROLLER))
52          self.connection.send(msg)
53          ###
54
55      def act_like_switch (self, packet_in, packet):
56          """
57          age como o switch, com diferenca no match veja abaixo
58          """
59          self.mac_to_port[str(packet.src)] = packet_in.in_port
60          if str(packet.dst) in self.mac_to_port:
61              port = self.mac_to_port[str(packet.dst)]
62              log.debug("Enviando pacote para a porta %d" %(port))
63              msg = of.ofp_flow_mod()
64              #msg.match = of.ofp_match(dl_dst = packet.dst)
65              """
66              o fluxo acima e muito simples, e preciso ser mais especifico
                    nessa regra
67              pois imagine o caso em que h1 envia pacotes para h2 todos com tcp
                     port 80
68              logo depois ele envia um pacote com tcp port 1234.
69              o switch pensara que e o mesmo fluxo e passara pela mesma porta
70              caso faca da maneira abaixo o switch percebera que o fluxo e
                    diferente
71              pois se difere a 'port tcp' e enviara o pacote para o cotrolador
                    analisar(PacketIn)
72              """
73              msg.match = of.ofp_match.from_packet(packet)
74              log.debug("instalando fluxo de %s.%i do type %d" % (packet.dst,
                    port, packet.type))
75              msg.idle_timeout = 10
76              msg.hard_timeout = 30
77              if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
78                  msg.buffer_id = packet_in.buffer_id
79              else:
80                  if packet_in.data is None:
81                      return
82                  msg.data = packet_in.data
83              msg.actions.append(of.ofp_action_output(port = port))
84              self.connection.send(msg)
85          else:
86              port = of.OFPP_FLOOD
87              log.debug("Enviando para todas as portas")
88              msg = of.ofp_packet_out()
89              msg.in_port = packet_in.in_port
90              if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
91                  msg.buffer_id = packet_in.buffer_id
92              else:
93                  if packet_in.data is None:
94                      return
```

```
95                msg.data = packet_in.data
96              action = of.ofp_action_output(port = port)
97              msg.actions.append(action)
98              self.connection.send(msg)
99
100   def _handle_PacketIn (self, event):
101     """
102     Esculta as mensagens PacketIn da rede
103     """
104     log.debug("Chegou um pacote no switch %s", dpidToStr(event.dpid))
105     packet_eth = event.parsed
106     if not packet_eth.parsed:
107         log.warning("Pacote foi ignorado pois nao ha cabecalhos")
108         return
109     """
110     Como todos os fluxos do firewall tem o destino ao controlador e gerado
            uma msg
111     do tipo PacketIn que deve ser tratada aqui, logo os pacotes que tem os
112     fluxos que o firewall enviou deve ser identificados e descartados
113     """
114     #verifica se o pack eth tem um pack ip
115     if packet_eth.type == pkt.ethernet.IP_TYPE:
116         #pega o payload dele e atribui ao pack ip
117         packet_ip = packet_eth.payload
118         #verifica a condicao do firewall
119         if packet_ip.dstip == "10.0.0.2":
120             log.warning("Pacote para o IP 10.0.0.2 e descartado")
121             return
122         #verifica se o pack ip tem um pack tcp
123         if packet_ip.protocol == pkt.ipv4.TCP_PROTOCOL:
124             #pega o payload dele e atribui ao pack tcp
125             packet_tcp = packet_ip.payload
126             #verifica a condicao do firewall
127             if packet_tcp.dstport == 80:
128                 log.warning("Pacote para a porta tcp 80 e descartado")
129                 return
130     #continua a execucao normal
131     packet_in = event.ofp
132     self.act_like_switch(packet_in, packet_eth)
133
134 def launch ():
135
136     def start_switch (event):
137         Firewall(event.connection)# enviamos a conexao do evento para a
                classe
138     core.openflow.addListenerByName("ConnectionUp", start_switch)
```

codigos/firewall.py

B.6 Código do ROUTER

A seguir é apresentado o código do componente ROUTER com maiores detalhes nos comentários.

```
from pox.core import core
from pox.lib.util import dpidToStr
from pox.lib.revent import *
import pox.openflow.libopenflow_01 as of
import pox.lib.packet as pkt
import pox
from pox.lib.packet.ethernet import ethernet
from pox.lib.packet.ipv4 import ipv4, IP_ANY
from pox.lib.packet.arp import arp
from pox.lib.revent import *
import time

# O tempo maximo parado dos fluxos
FLOW_IDLE_TIMEOUT = 10

# Tempo maximo das entradas ARP
ARP_TIMEOUT = 60 * 2

log = core.getLogger()

class Router (object):
  def __init__ (self, connection):
    self.connection = connection
    connection.addListeners(self)
    """
    e criada uma tabela ARP semelhante as anteriores, dessa vez esta tabela
    e IP para MAC e porta. Tem como chave o endereco IP e como valores a
       porta
    o MAC e o timeout.
    Exemplo de uma entrada:
    {IPAddr('10.0.1.2') : (1, EthAddr('96:95:82:9e:72:3e'),
       1358653720.522618)}
    Logo para a porta de chegada basta pegar o primeiro valor da tupla
       '[0]'
    para o endereco MAC basta pegar o segundo valor da tupla '[1]'
    para o tempo basta pegar o terceiro valor da tupla '[2]'
    """
    self.arp_table = {}
    """
    e uma tabela que guarda todos os ips que passam pelo comutador, tanto
       de destino
    quanto de fonte. Essa tabela serve para que ips que nao existem na rede
        nao fiquem sendo
    inundados em um laco sem fim quando se quer usar o protocolo ARP.
    sua chave e o IP e seus valores sao um bool e um time.
    """
    self.table_ip = {}


  def arp_responder (self, packet_in, packet, dpid):
      """
      Este metodo responde as requisicoes ARP, quando o IP nao e encontrado
          na tabela
```

```
48          ele inunda todas as portas do comutador procurando. E muito
                semelhante ao switch
49          so que dessa vez ele procura IPs e nao MACs
50          """
51          packet_arp = packet.next#pega o datagrama do quadro
52          #mostra se ele e uma requisicao ARP ou uma resposta ARP
53          log.debug("%s %i ARP %s %s => %s", dpidToStr(dpid), packet_in.in_port
                ,
54                    {arp.REQUEST:"request",arp.REPLY:"reply"}.get(packet_arp.
                        opcode,
55                    'op:%i' % (packet_arp.opcode,)), str(packet_arp.protosrc),
56                    str(packet_arp.protodst))
57          #Faz algumas conferencia basicas para ver se datagrama ARP esta
                completo:
58          if packet_arp.prototype == arp.PROTO_TYPE_IP:#primeira
59              if packet_arp.hwtype == arp.HW_TYPE_ETHERNET:#segunda
60            #terceira, verifica se o IP da fonte nao e 0.0.0.0
61                  if packet_arp.protosrc != IP_ANY:
62                      # aprende a porta/MAC atualizando o time
63                      log.debug("%s %i aprendendo %s", dpidToStr(dpid),
                            packet_in.in_port,
64                                packet_arp.protosrc)
65                      self.arp_table[packet_arp.protosrc] = (packet_in.in_port,
                            packet.src,
66                                                             time.time() +
                                                                ARP_TIMEOUT)
67                      #verifica se o pacote arp e request para fazer o reply
68                      if packet_arp.opcode == arp.REQUEST:
69                    #verifica se o ip de destino e conhecido
70                          if packet_arp.protodst in self.arp_table:
71                              #verifica se o tempo do ARP nao expirou ja
72                              if not time.time() > self.arp_table[packet_arp.
                                    protodst][2]:
73                                  #cria um pacote arp de resposta
74                                  reply = arp()
75                                  reply.hwtype = packet_arp.hwtype
76                                  reply.prototype = packet_arp.prototype
77                                  reply.hwlen = packet_arp.hwlen
78                                  reply.protolen = packet_arp.protolen
79                                  reply.opcode = arp.REPLY
80                                  reply.hwdst = packet_arp.hwsrc
81                                  reply.protodst = packet_arp.protosrc#destino=
                                        fonte do request
82                                  reply.protosrc = packet_arp.protodst#fonte=
                                        destino do request
83                      #busca na tabela o MAC
84                                  reply.hwsrc = self.arp_table[packet_arp.
                                        protodst][1]
85                                  #empacota o pacote em um quadro
86                                  ether = ethernet(type=packet.type, src=reply.
                                        hwsrc,
87                                dst=packet_arp.hwsrc)
88                                  ether.set_payload(reply)
89                                  log.debug("%i %i respondendo o ARP for %s" %
                                        (dpid,
90                                   packet_in.in_port, str(reply.protosrc)))
```

```
91                                  #manda a mensagem reply na mesma porta que
                                        chegou o request
92                                  msg = of.ofp_packet_out()
93                                  msg.data = ether.pack()
94                                  msg.actions.append(of.ofp_action_output(port
                                        = of.OFPP_IN_PORT))
95                                  msg.in_port = packet_in.in_port
96                                  self.connection.send(msg)
97                                  return
98          #para que pacotes reply nao fiquem inundando na rede no laco sem fim
99          if len(self.arp_table.keys()) != 0:#verifica se a tabela nao ta vazia
100             if packet_arp.protodst in self.arp_table:#se ja existir o ip de
                    destino na tabela
101                 #verifica se o time do APR na tabela nao ta expirado
102                 if not time.time() > self.arp_table[packet_arp.protodst][2]:
103                     #se nao tiver ele NAO inunda o pacote na rede
104                     return
105         #inunda o pacote na rede para todos os outros casos
106         log.debug("%i %i flooding ARP %s %s => %s" % (dpid, packet_in.in_port
                ,
107                   {arp.REQUEST:"request",arp.REPLY:"reply"}.get(packet_arp.
                        opcode,
108                   'op:%i' % (packet_arp.opcode,)), str(packet_arp.protosrc),
109           str(packet_arp.protodst)))
110         msg = of.ofp_packet_out()
111         msg.in_port = packet_in.in_port
112         action = of.ofp_action_output(port = of.OFPP_FLOOD)
113         msg.actions.append(action)
114         if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
115             msg.buffer_id = packet_in.buffer_id
116         else:
117             if packet_in.data is None:
118                 return
119             msg.data = packet_in.data
120         self.connection.send(msg)
121         return
122
123
124     def route (self, packet_in, packet, dpid):
125         """
126         Roteia o pacote IP ou ICMP dos nos da rede
127         Utiliza a tabela ARP para encontrar as portas de saidas do pacote
128         e instala um fluxo no comutador com essas condicoes
129         """
130         packet_ip = packet.next#pega o datagrama do quadro
131         if len(self.arp_table.keys()) == 0:
132             log.warning("Tabela ARP vazia, pacote ignorado")
133             return
134         # atualizando o time da tabela ARP
135         log.debug("%s %i atualizando %s", dpidToStr(dpid),packet_in.in_port,
                packet_ip.srcip)
136         self.arp_table[packet_ip.srcip] = (packet_in.in_port, packet.src,
                time.time() + ARP_TIMEOUT)
137         #verifica se o destino do pacote esta na tabela ARP
138         if packet_ip.dstip in self.arp_table:
139             #verifica se nao expirou
```

```
140             if not time.time() > self.arp_table[packet_ip.dstip][2]:
141                 #pega a porta relacionada com o IP de destino
142                 port = self.arp_table[packet_ip.dstip][0]
143                 log.debug("pacote IP MAC fonte %s MAC destino %s" %(packet.
                        src, packet.dst))
144                 log.debug("pacote IP vem da porta %i" %(packet_in.in_port))
145                 log.debug("IP fonte %s IP destino %s" %(packet_ip.srcip,
                        packet_ip.dstip))
146                 log.debug("Enviando pacote IP para a porta %i" %(port))
147                 #criando a mensagem openflow de instalacao de fluxo
148                 msg = of.ofp_flow_mod()
149                 msg.match = of.ofp_match.from_packet(packet, packet_in.
                        in_port)
150                 log.debug("instalando fluxo de %s para %s para a porta %i" %
                        (packet_ip.srcip, packet_ip.dstip, port))
151                 msg.idle_timeout = FLOW_IDLE_TIMEOUT
152                 msg.hard_timeout = 30
153                 if packet_in.buffer_id != -1 and packet_in.buffer_id is not
                        None:
154                     msg.buffer_id = packet_in.buffer_id
155                 else:
156                     if packet_in.data is None:
157                         return
158                     msg.data = packet_in.data
159                 msg.actions.append(of.ofp_action_output(port = port))
160                 self.connection.send(msg)
161                 return
162             else:
163                 log.warning("Time do ARP expirado!!!")
164                 #apaga a entrada da tabela ARP
165                 del self.arp_table[packet_ip.dstip]
166                 return
167         else:
168             log.warning("ARP nao encontrado na tabela!!!")
169 
170     def _handle_PacketIn (self, event):
171       """
172       Esculta as mensagens PacketIn da rede
173       """
174       packet_eth = event.parsed
175       packet_in = event.ofp
176       dpid = event.dpid
177       if not packet_eth.parsed:
178           log.warning("Pacote foi ignorado pois nao ha cabecalhos")
179           return
180 
181       if packet_eth.type == ethernet.LLDP_TYPE:
182           # Ignora pacote LLDP
183           return
184 
185       if packet_eth.type == pkt.ethernet.ARP_TYPE:#chega um pacote ARP
186           log.debug("Chegou um pacote ARP no switch %s", dpidToStr(event.dpid
                  ))
187           """
188           quando um pacote chega no comutador e anotado seu ip de fonte dado
                  a ele uma tag de
```

```
189             true com um time (ARP_TIMEOUT). Quanto ao seu destino ele e dado
                    uma tag de False e
190             um time (5), se em 5 segundos ele nao encontrar o ip, este pacote e
                     ignorado, pois ele
191             nao existe na rede.
192             """
193             #anotado o ip de fonte
194             self.table_ip[packet_eth.next.protosrc] = (True, time.time() +
                    ARP_TIMEOUT)
195             #se o ip de destino estiver na tabela
196             if packet_eth.next.protodst in self.table_ip:
197                 #se o ip de destino for false
198                 if not self.table_ip[packet_eth.next.protodst][0]:
199                     #se o time dele expirou
200                     if time.time() > self.table_ip[packet_eth.next.protodst
                            ][1]:
201                         log.warning("O host com ip %s nao pode ser encontrado
                                na rede", packet_eth.next.protodst)
202                         return
203                 else:
204                     #se o ip de destino expirou
205                     if time.time() > self.table_ip[packet_eth.next.protodst
                            ][1]:
206                         #seta ele como false
207                         self.table_ip[packet_eth.next.protodst] = (False,self.
                                table_ip[packet_eth.next.protodst][1])
208             else:#se o ip de destino nao estiver na tabela
209                 self.table_ip[packet_eth.next.protodst] = (False, time.time() +
                         5)
210             #so entao e executado o protocolo ARP
211             self.arp_responder(packet_in, packet_eth, dpid)
212         else:
213             log.debug("Chegou um pacote IP no switch %s", dpidToStr(event.dpid)
                    )
214             #executado o protocolo de roteamento
215             self.route(packet_in, packet_eth, dpid)
216         return
217
218 def launch ():
219
220     def start_switch (event):
221         Router(event.connection)# enviamos a conexao do evento para a
                classe
222     core.openflow.addListenerByName("ConnectionUp", start_switch)
```

codigos/router.py

B.7 Código do ESPECÍFICO

A seguir é apresentado o código do componente ”Especifico” com maiores detalhes nos comentários.

```
from pox.core import core
from pox.lib.util import dpidToStr
import pox.openflow.libopenflow_01 as of
import pox.lib.packet as pkt
import pox
from pox.lib.packet.ethernet import ethernet
from pox.lib.packet.ipv4 import ipv4
from pox.lib.packet.arp import arp


#tempos para a instalacao de fluxos
FLOW_IDLE_TIMEOUT = of.OFP_FLOW_PERMANENT
FLOW_HARD_TIMEOUT = of.OFP_FLOW_PERMANENT

log = core.getLogger()

class Especifico (object):
  def __init__ (self, connection):
    self.connection = connection
    connection.addListeners(self)

  def install_flow (self, dl_type, nw_dst, packet_in, packet, port):
      """
      metodo que instala fluxos na tabela de fluxos do comutador
      """
      if port == 65531:
          log.debug("Instalando fluxo para todas as portas")
      else:
          log.debug("Instalando fluxo para a porta %i" % port)
      msg = of.ofp_flow_mod()
      msg.match.dl_type = dl_type
      msg.match.nw_dst = nw_dst
      msg.match.in_port = packet_in.in_port
      msg.match.dl_type = packet.type
      msg.match.dl_src = packet.src
      msg.idle_timeout = FLOW_IDLE_TIMEOUT#veja os tempo acima
      msg.hard_timeout = FLOW_HARD_TIMEOUT#veja os tempo acima
      if packet_in.buffer_id != -1 and packet_in.buffer_id is not None:
          msg.buffer_id = packet_in.buffer_id
      else:
          if packet_in.data is None:
              return
          msg.data = packet_in.data
      msg.actions.append(of.ofp_action_output(port = port))
      self.connection.send(msg)
      return

  def install_rules (self, packet_in, packet):
    """
    metodo que instala regras na rede toda
    este componente so funcionara exclusivamente para essa rede, e um
       componente bem
    especifico que ja tem determinado suas portas de entrada e saida como
       numero de hosts
    e ip dos mesmos. Serve como exemplo de como controlar o fluxo da rede.
```

```
54
55      Este componente (junto com este metodo) funciona em um cenario com tres
            switches ligados
56      entre si, cada um em sua subrede (10.0.1.x;10.0.2.x;10.0.3.x), cada um
           com dois hosts
57      ligados nas portas 1 e 2 com ips 10.0.X.2; 10.0.X.3.
58      s0 ligado a s1 pela porta 3
59      s0 ligado a s2 pela porta 4
60      s1 ligado a s0 pela porta 3
61      s1 ligado a s2 pela porta 4
62      s2 ligado a s0 pela porta 3
63      s2 ligado a s1 pela porta 4
64      Fluxo de porta destino tcp 1234 sera comutado por s0 e s1
           exclusivamente pelas portas 3 e 3
65      de cada switch, os demais pacotes nao poderao usar este caminho para
           fluxo de dados, todos os dados
66      deverao contornar a rede. (Ex. fluxo normal de s0 para s1 devera passar
            por s2)
67      """
68      #pega o proximo cabecaolho
69      p=packet.next
70      #verifica se o pacote e ipv4 ou arp e pega seus ips
71      if isinstance(p, ipv4):
72          nw_src = p.srcip
73          nw_dst = p.dstip
74          dl_type = 0x0800
75          p = p.next
76      elif isinstance(p, arp):
77          if p.opcode <= 255:
78              nw_src = p.protosrc
79              nw_dst = p.protodst
80              dl_type = 0x0806
81          else:
82              return
83      #sendo qualquer outra coisa retorna
84      else:
85          return
86      #comeca as atribuicoes de caminho
87      if (str(nw_dst).split('.'))[2] != (str(nw_src).split('.'))[2]:
88          #esse caso e quando o destino nao e local
89          if (str(nw_src).split('.'))[2] == "1":
90              #esse caso e quando a fonte e de s0
91              if packet_in.in_port == 1 or packet_in.in_port == 2:#ta no s0
                   mesmo
92                  if packet.find("tcp") != None and (packet.find("tcp").
                       dstport == 1234 or packet.find("tcp").srcport == 1234):
93                      if (str(nw_dst).split('.'))[2] == "2":#se o destino e
                           s1
94                          self.install_flow(dl_type, nw_dst, packet_in,
                               packet, 3)
95                          return
96                      else:#o destino e s2
97                          self.install_flow(dl_type, nw_dst, packet_in,
                               packet, 4)
98                          return
99                  else:
```

```
100                     self.install_flow(dl_type, nw_dst, packet_in, packet,
                            4)
101                     return
102                 elif packet_in.in_port == 4:#ta no s1 destino do s1
103                     if (str(nw_dst).split('.'))[3] == "2":
104                         self.install_flow(dl_type, nw_dst, packet_in, packet,
                                1)
105                         return
106                     else:
107                         self.install_flow(dl_type, nw_dst, packet_in, packet,
                                2)
108                         return
109                 else:#ta no s2 ou no s1 (tcp port 1234)
110                     if (str(nw_dst).split('.'))[2] == "3":#se o destino e de s2
111                         if (str(nw_dst).split('.'))[3] == "2":#se o destino e
                                de 3.2
112                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 1)
113                             return
114                         else:
115                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 2)
116                             return
117                     else:#destino nao e de s2 entao e de s1
118                         #se ele tem a port tcp 1234 e fonte s0 entao ele chegou
                                pela porta 3 e ja esta no s1
119                         if packet.find("tcp") != None and (packet.find("tcp").
                                dstport == 1234 or packet.find("tcp").srcport ==
                                1234):
120                             if (str(nw_dst).split('.'))[3] == "2":# se o
                                    destino e 2.2
121                                 self.install_flow(dl_type, nw_dst, packet_in,
                                        packet, 1)
122                                 return
123                             else:#se o destino e 2.3
124                                 self.install_flow(dl_type, nw_dst, packet_in,
                                        packet, 2)
125                                 return
126                         else:#fonte s0 passando pelo s2 com destino a s1
127                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 4)
128                             return
129             elif (str(nw_src).split('.'))[2] == "2":
130                 #esse caso e quando a fonte e de s1
131                 if packet_in.in_port == 1 or packet_in.in_port == 2:#ta no s1
                        mesmo
132                     if packet.find("tcp") != None and (packet.find("tcp").
                            dstport == 1234 or packet.find("tcp").srcport == 1234):
133                         if (str(nw_dst).split('.'))[2] == "1":#se o destino e
                                s0
134                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 3)
135                             return
136                         else:#o destino e s2
137                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 4)
```

```
138                         return
139                     else:
140                         self.install_flow(dl_type, nw_dst, packet_in, packet,
                                4)
141                         return
142                 elif packet_in.in_port == 4:#pode ser o s0 ou o s2 inunda
143                     self.install_flow(dl_type, nw_dst, packet_in, packet, of.
                            OFPP_FLOOD)
144                 else:
145                     if (str(nw_dst).split('.'))[2] == "1":#se o destino e s0
146                         if (str(nw_dst).split('.'))[3] == "2":#se o destino e
                                1.2
147                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 1)
148                             return
149                         else:
150                             self.install_flow(dl_type, nw_dst, packet_in,
                                    packet, 2)
151                             return
152                     else:
153                         return
154             elif (str(nw_src).split('.'))[2] == "3":
155                 #esse caso e quando a fonte e de s2
156                 if packet_in.in_port == 1 or packet_in.in_port == 2:#ta no s2
                        mesmo
157                     if (str(nw_dst).split('.'))[2] == "1":
158                         self.install_flow(dl_type, nw_dst, packet_in, packet,
                                3)
159                         return
160                     else:
161                         self.install_flow(dl_type, nw_dst, packet_in, packet,
                                4)
162                         return
163                 elif packet_in.in_port == 4:#pode ser o s0 ou o s1 inunda
164                     self.install_flow(dl_type, nw_dst, packet_in, packet, of.
                            OFPP_FLOOD)
165                     return
166                 else:
167                     return
168             else:
169                 return
170         #esse caso e quando o destino e local, bastando olhar seu destino(
                ultimo quarteto do ip)
171         elif (str(nw_dst).split('.'))[3] == "2":#para a porta 1
172             self.install_flow(dl_type, nw_dst, packet_in, packet, 1)
173             return
174         elif (str(nw_dst).split('.'))[3] == "3":#para a porta 2
175             self.install_flow(dl_type, nw_dst, packet_in, packet, 2)
176             return
177         #qualquer outro caso retorna
178         else:
179             return
180
181     def _handle_PacketIn (self, event):
182         """
183         Esculta as mensagens PacketIn da rede
```

```
184     """
185     packet_eth = event.parsed
186     packet_in = event.ofp
187 
188     if not packet_eth.parsed:
189         log.warning("Pacote foi ignorado pois nao ha cabecalhos")
190         return
191 
192     if packet_eth.type == ethernet.LLDP_TYPE:
193         # Ignora pacote LLDP
194         return
195 
196     #instala as regras a medida que os pacotes chegam no comutador e vao
            para o controlador
197     self.install_rules(packet_in, packet_eth)
198     return
199 
200 def launch ():
201 
202     def start_switch (event):
203         Especifico(event.connection)# enviamos a conexao do evento para a
                classe
204     core.openflow.addListenerByName("ConnectionUp", start_switch)
```

codigos/especifico.py

Apêndice C

API MiniNet (MiniGUI)

Este é o código utilizado para a criação da API MiniGUI criada neste trabalho que tem como objetivo fornecer uma interface gráfica para a inicialização do CLI do MiniNet. Caso tenha alguma dúvida sobre as opções de inicialização do CLI do MiniNet consulte o capitulo 6. A interface mostra as opções de inicialização do CLI do MiniNet e inicia o CLI no Shell. Pode ser útil para quem começa a utilizar o MiniNet e não conhece ou lembra todas as opções de inicialização.

Para executar esta API, crie um arquivo com o nome de `minigui.py` no diretório `mininet/examples` (por exemplo) e execute-o na máquina virtual da seguinte maneira:

```
$ sudo python -O /mininet/examples/minigui.py
```

Pode-se utilizar para entender melhor a estrutura do MiniNet. Esta interface gráfica foi criada com a utilização da biblioteca `Tkinter` do Python, seu código pode ser visto a seguir. A interface é auto explicativa (Figura C.1).

```
"""
THE MININET GUI
author: Lucas Costa (lucasrc.rodri@gmail.com)
"""
import os
import commands
import sys
from mininet.clean import cleanup
from mininet.cli import CLI
from mininet.log import lg, LEVELS, info, setLogLevel
from mininet.net import Mininet, init
from mininet.node import KernelSwitch, Host, Controller, ControllerParams,
    NOX
from mininet.node import RemoteController, UserSwitch, OVSKernelSwitch
from mininet.topo import SingleSwitchTopo, LinearTopo,
    SingleSwitchReversedTopo
from mininet.topolib import TreeTopo
from mininet.util import makeNumeric
```

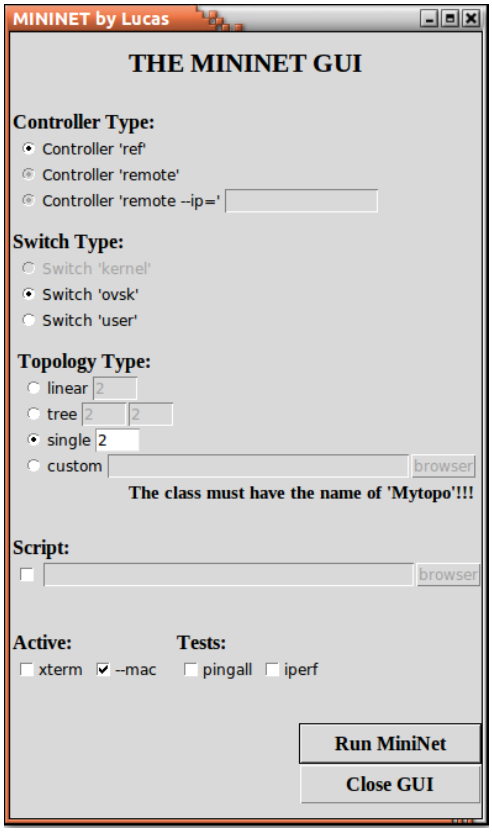


Figura C.1: MiniGUI API minigui.py (API de inicialização do CLI do MiniNet).

```
17 from Tkinter import *
18 from tkFileDialog import askopenfilename
19
20 class curry:
21     def __init__(self, fun, *args, **kwargs):
22         self.fun = fun
23         self.pending = args[:]
24         self.kwargs = kwargs.copy()
25     def __call__(self, *args, **kwargs):
26         if kwargs and self.kwargs:
27             kw = self.kwargs.copy()
28             kw.update(kwargs)
29         else:
30             kw = kwargs or self.kwargs
31         return self.fun(*(self.pending + args), **kw)
32
33 def event_lambda(f, *args, **kwds ):
34     return lambda event, f=f, args=args, kwds=kwds : f( *args, **kwds )
35
36 class MiniNetGui:
37     def __init__(self, parent):
38         self.mininet = None
39         self.controller = IntVar()
40         self.switch = IntVar()
41         self.topology = IntVar()
```

```
42          self.script = IntVar()
43          self.xterm = IntVar()
44          self.controller = 1
45          self.switch = 2
46          self.topology = 3
47          self.script = 10
48          self.xterm = 20
49          self.mac = 31
50          self.ping = 40
51          self.iperf = 50
52          self.filename_topo = None
53          self.filename_script = None
54          self.myLastButtonInvoked = None
55          parent.title("MININET by Lucas")
56          self.myParent = parent
57          ###total frame
58          self.myframe = Frame(parent)
59          self.myframe.pack()
60          ###total frame
61          ###top frame
62          self.top_frame = Frame(self.myframe,pady="3m")
63          self.top_frame.pack()
64          ###top frame
65          ###middle frame
66          self.middle_frame = Frame(self.myframe, relief=RIDGE,height=150,
                width=300)
67          self.middle_frame.pack(side=TOP, expand=NO, fill=NONE)
68          ###middle frame
69          ###bottom frame
70          self.bottom_frame = Frame(self.myframe,relief=RIDGE,height=150,
                width=300)
71          self.bottom_frame.pack(side=TOP, expand=NO, fill=NONE, anchor=W)
72          ###bottom frame
73          ###buttons frame
74          self.buttons_frame = Frame(self.myframe,relief=RIDGE,height=75,
                width=300,pady="3m")
75          self.buttons_frame.pack(side=TOP, expand=NO, fill=NONE, anchor=E)
76          ###buttons frame
77          ###controller frame
78          self.top1_middle_frame = Frame(self.middle_frame,relief=RIDGE,
                height=120, width=300)
79          self.top1_middle_frame.pack(side=TOP, expand=NO, fill=NONE, anchor=
                W,pady="3m")
80          self.top_top1_middle_frame = Frame(self.top1_middle_frame,relief=
                RIDGE,height=60, width=300)
81          self.top_top1_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
82          self.bottom_top1_middle_frame = Frame(self.top1_middle_frame,relief
                =RIDGE,height=60, width=300)
83          self.bottom_top1_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
84          ###controller frame
85          ###switch frame
86          self.top2_middle_frame = Frame(self.middle_frame,relief=RIDGE,
                height=120, width=300)
```

```
87          self.top2_middle_frame.pack(side=TOP, expand=NO, fill=NONE, anchor=
                W)
88          self.top_top2_middle_frame = Frame(self.top2_middle_frame,relief=
                RIDGE,height=60, width=300)
89          self.top_top2_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
90          ###switch frame
91          ###topology frame
92          self.middle_middle_frame = Frame(self.middle_frame, relief=RIDGE,
                height=50, width=300)
93          self.middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,pady="
                3m")
94          self.frm1_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
95          self.frm1_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
96          self.frm2_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
97          self.frm2_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
98          self.frm3_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
99          self.frm3_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
100         self.frm4_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
101         self.frm4_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
102         self.frm5_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
103         self.frm5_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
104         self.frm6_middle_middle_frame = Frame(self.middle_middle_frame,
                relief=RIDGE,height=10, width=300)
105         self.frm6_middle_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=E)
106         ###topology frame
107         ###script frame
108         self.bottom_middle_frame = Frame(self.middle_frame, relief=RIDGE,
                height=50, width=300)
109         self.bottom_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W,pady="3m")
110         self.frm1_bottom_middle_frame = Frame(self.bottom_middle_frame,
                relief=RIDGE,height=10, width=300)
111         self.frm1_bottom_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
112         self.frm2_bottom_middle_frame = Frame(self.bottom_middle_frame,
                relief=RIDGE,height=10, width=300)
113         self.frm2_bottom_middle_frame.pack(side=TOP, expand=NO, fill=NONE,
                anchor=W)
114         ###script frame
115         ###options frame
116         self.top_bottom_frame = Frame(self.bottom_frame,relief=RIDGE,height
                =75, width=300,pady="3m")
117         self.top_bottom_frame.pack(side=TOP, anchor=W)
```

```
118         self.top_bottom_left_frame = Frame(self.top_bottom_frame,relief=
                RIDGE,height=75, width=300,pady="3m")
119         self.top_bottom_left_frame.pack(side=LEFT, anchor=W)
120         self.top_bottom_right_frame = Frame(self.top_bottom_frame,relief=
                RIDGE,height=75, width=300,pady="3m",padx="3m")
121         self.top_bottom_right_frame.pack(side=LEFT, anchor=W)
122         ###options frame
123         ###title
124         self.text_title = Label(self.top_frame, text="THE MININET GUI",
                font = ('Times','16','bold'))
125         self.text_title.pack(side="top", fill="both", expand=True)
126         ###title
127         ###controller args
128         self.text_controller = Label(self.top_top1_middle_frame, text="
                Controller Type:", font = ('Times','12','bold'))
129         self.text_controller.pack(side="top", anchor=W)
130         self.radio1_controller = Radiobutton(self.top_top1_middle_frame,
                text="Controller 'ref'",variable=self.controller,
131                                              command=lambda controller = 1
                                                    : self.radioController(
                                                    controller))
132         self.radio1_controller.pack(side="top", anchor=W)
133         self.radio1_controller.select()
134         self.radio2_controller = Radiobutton(self.top_top1_middle_frame,
                text="Controller 'remote'",variable=self.controller,
135                                              value=5,command=lambda
                                                    controller = 2 : self.
                                                    radioController(controller)
                                                    )
136         self.radio2_controller.pack(side="top", anchor=W)
137         self.radio3_controller = Radiobutton(self.bottom_top1_middle_frame,
                 text="Controller 'remote --ip='",variable=self.controller,
138                                              value=6,command=lambda
                                                    controller = 3 : self.
                                                    radioController(controller)
                                                    )
139         self.radio3_controller.pack(side="left", anchor=W)
140         self.text_ip_controller = Entry(self.bottom_top1_middle_frame,
                width=15,background='gray')
141         self.text_ip_controller.pack(side="left", fill="both", expand=True,
                 anchor=E)
142         self.text_ip_controller.config(state=DISABLED)
143         ###controller args
144         ###switch args
145         self.text_switch = Label(self.top_top2_middle_frame, text="Switch
                Type:", font = ('Times','12','bold'))
146         self.text_switch.pack(side="top", anchor=W)
147         self.radio1_switch = Radiobutton(self.top_top2_middle_frame, text="
                Switch 'kernel'",variable=self.switch,
148                                              command=lambda switch = 1 :
                                                    self.radioSwitch(switch))
149         self.radio1_switch.pack(side="top", anchor=W)
150         self.radio1_switch.config(state=DISABLED)### The kernel switch this
                 off in this version###
151         self.radio2_switch = Radiobutton(self.top_top2_middle_frame, text="
                Switch 'ovsk'",variable=self.switch,
```

```
152                                             value=7,command=lambda switch
                                                    = 2 : self.radioSwitch(
                                                    switch))
153         self.radio2_switch.pack(side="top", anchor=W)
154         self.radio2_switch.select()
155         self.radio3_switch = Radiobutton(self.top_top2_middle_frame, text="
                Switch 'user'",variable=self.switch,
156                                             value=8,command=lambda switch
                                                    = 3 : self.radioSwitch(
                                                    switch))
157         self.radio3_switch.pack(side="left", anchor=W)
158         ###switch args
159         ###topology args
160         self.text_topology = Label(self.frm1_middle_middle_frame, text="
                Topology Type:",font = ('Times','12','bold'))
161         self.text_topology.pack(side="top", anchor=W)
162         ###
163         self.radio1_topology = Radiobutton(self.frm2_middle_middle_frame,
                text="linear",variable=self.topology,
164                                             command=lambda topology = 1:
                                                    self.radioTopology(topology
                                                    ))
165         self.radio1_topology.pack(side="left", anchor=W)
166         self.text1_topology = Entry(self.frm2_middle_middle_frame,width=4,
                background='gray')
167         self.text1_topology.pack(side="left", fill="both", expand=True,
                anchor=E)
168         self.text1_topology.insert(0,"2")
169         self.text1_topology.config(state=DISABLED)
170         ###
171         ###
172         self.radio2_topology = Radiobutton(self.frm3_middle_middle_frame,
                text="tree",variable=self.topology,
173                                             value=2,command=lambda
                                                    topology = 2: self.
                                                    radioTopology(topology))
174         self.radio2_topology.pack(side="left", anchor=W)
175         self.text21_topology = Entry(self.frm3_middle_middle_frame,width=4,
                background='gray')
176         self.text21_topology.pack(side="left", fill="both", expand=True,
                anchor=E)
177         self.text21_topology.insert(0,"2")
178         self.text21_topology.config(state=DISABLED)
179         self.text22_topology = Entry(self.frm3_middle_middle_frame,width=4,
                background='gray')
180         self.text22_topology.pack(side="left", fill="both", expand=True,
                anchor=E)
181         self.text22_topology.insert(0,"2")
182         self.text22_topology.config(state=DISABLED)
183         ###
184         ###
185         self.radio3_topology = Radiobutton(self.frm4_middle_middle_frame,
                text="single",variable=self.topology,
186                                             value=3,command=lambda
                                                    topology = 3: self.
                                                    radioTopology(topology))
```

```
187             self.radio3_topology.pack(side="left", anchor=W)
188             self.radio3_topology.select()
189             self.text3_topology = Entry(self.frm4_middle_middle_frame,width=4,
                    background='white')
190             self.text3_topology.pack(side="left", fill="both", expand=True,
                    anchor=E)
191             self.text3_topology.insert(0,"2")
192             ###
193             ###
194             self.radio4_topology = Radiobutton(self.frm5_middle_middle_frame,
                    text="custom",variable=self.topology,
195                                                value=4,command=lambda
                                                    topology = 4: self.
                                                    radioTopology(topology))
196             self.radio4_topology.pack(side="left", anchor=W)
197             self.text4_topology = Entry(self.frm5_middle_middle_frame,width=30,
                    background='gray')
198             self.text4_topology.pack(side="left", fill="both", expand=True,
                    anchor=E)
199             self.text4_topology.config(state=DISABLED)
200             self.button_topology = Button(self.frm5_middle_middle_frame,command
                    =self.button_topology)
201             self.button_topology.configure(text="browser", height=1, padx="0m",
                     pady="0m")
202             self.button_topology.pack(side="left")
203             self.button_topology.bind("<Return>", self.button_topology_a)
204             self.button_topology.config(state=DISABLED)
205             self.text_custom = Label(self.frm6_middle_middle_frame, text="The
                    class must have the name of 'Mytopo'!!!",font = ('Times','11','
                    bold'))
206             self.text_custom.pack(side="top", anchor=E)
207             ###
208             ###topology args
209             ###script args
210             self.text_script = Label(self.frm1_bottom_middle_frame, text="
                    Script:",font = ('Times','12','bold'))
211             self.text_script.pack(side="top", anchor=W)
212             self.check_script = Checkbutton(self.frm2_bottom_middle_frame,
                    variable=self.script, command=self.checkScript)
213             self.check_script.pack(side="left", anchor=W)
214             self.text_script = Entry(self.frm2_bottom_middle_frame,width=37,
                    background='gray')
215             self.text_script.pack(side="left", fill="both", expand=True, anchor
                    =E)
216             self.text_script.config(state=DISABLED)
217             self.button_script = Button(self.frm2_bottom_middle_frame,command=
                    self.button_script)
218             self.button_script.configure(text="browser", height=1, padx="0m",
                    pady="0m")
219             self.button_script.pack(side="left")
220             self.button_script.bind("<Return>", self.button_script_a)
221             self.button_script.config(state=DISABLED)
222             ###script args
223             ###options args
224             self.text_active = Label(self.top_bottom_left_frame, text="Active:"
                    ,font = ('Times','12','bold'))
```

```
225         self.text_active.pack(side="top", anchor=W)
226         self.check_xterm = Checkbutton(self.top_bottom_left_frame, text="
                xterm", variable=self.xterm, command=self.checkXterm)
227         self.check_xterm.pack(side="left", anchor=W)
228         self.check_mac = Checkbutton(self.top_bottom_left_frame, text="--
                mac", variable=self.mac, command=self.checkMac)
229         self.check_mac.select()
230         self.check_mac.pack(side="top", anchor=W)
231         self.text_tests = Label(self.top_bottom_right_frame, text="Tests:",
                font = ('Times','12','bold'))
232         self.text_tests.pack(side="top", anchor=W)
233         self.check_ping = Checkbutton(self.top_bottom_right_frame, text="
                pingall", variable=self.ping, command=self.checkPing)
234         self.check_ping.pack(side="left", anchor=W)
235         self.check_iperf = Checkbutton(self.top_bottom_right_frame, text="
                iperf", variable=self.iperf, command=self.checkIperf)
236         self.check_iperf.pack(side="top", anchor=W)
237         ###options args
238         ###buttons args
239         self.button1 = Button(self.buttons_frame,command=self.buttonRun,
                font = ('Times','12','bold'))
240         self.button1.configure(text="Run MiniNet", height=1, width=15)
241         self.button1.pack(side="top", anchor=SE)
242         self.button1.bind("<Return>", self.buttonRun_a)
243         self.button1.focus_force()
244         self.button2 = Button(self.buttons_frame,command=self.buttonClose,
                font = ('Times','12','bold'))
245         self.button2.configure(text="Close GUI", height=1, width=15)
246         self.button2.pack(side="top",anchor=SE)
247         self.button2.bind("<Return>", self.buttonClose_a)
248         ###buttons args
249
250     def setCustom( self, name, value ):
251         "Set custom parameters for MininetRunner."
252         if name in ( 'TOPOS', 'switches', 'hosts', 'controllers' ):
253             # Update dictionaries
254             param = name.upper()
255             globals()[ param ].update( value )
256         elif name == 'validate':
257             # Add custom validate function
258             self.validate = value
259         else:
260             # Add or modify global variable or class
261             globals()[ name ] = value
262
263     def parseCustomFile( self, fileName ):
264         "Parse custom file and add params before parsing cmd-line options."
265         custom = {}
266         if os.path.isfile( fileName ):
267             execfile( fileName, custom, custom )
268             for name in custom:
269                 self.setCustom( name, custom[ name ] )
270         else:
271             raise Exception( 'could not find custom file: %s' % fileName )
272
273     def buttonClose(self):
```

```
274             self.myParent.destroy()
275
276         def buttonClose_a(self, event):
277             self.button2Click()
278
279         def buttonRun(self):
280             print "\n
                    ************************************************************
                    "
281             print "do not forget to exit the CLI with the command '<Ctrl>+D' or
                     'exit'"
282             print "
                    ************************************************************\
                    n"
283             ipcontroller = StringVar()
284             ipcontroller = self.text_ip_controller.get()
285             if ipcontroller == "":
286                 ipcontroller = "127.0.0.1"
287             lineartopo = self.text1_topology.get()
288             if lineartopo == "":
289                 lineartopo = 2
290             treetopo_depth = self.text21_topology.get()
291             if treetopo_depth == "":
292                 treetopo_depth = 1
293             treetopo_fanout = self.text22_topology.get()
294             if treetopo_fanout == "":
295                 treetopo_fanout = 2
296             singletopo = self.text3_topology.get()
297             if singletopo == "":
298                 singletopo = 2
299             if self.xterm == 20:
300                 x = False
301             else:
302                 x = True
303             if self.mac == 30:
304                 m = False
305             else:
306                 m = True
307             #checks the topology selected
308             if self.topology == 1:
309                 topo = LinearTopo(k =int(lineartopo))
310             elif self.topology == 2:
311                 topo = TreeTopo(depth=int(treetopo_depth))
312             elif self.topology == 3:
313                 topo = SingleSwitchTopo(k=int(singletopo))
314             else:
315                 #The class must have the name of 'Mytopo'
316                 self.parseCustomFile( self.filename_topo )
317                 topo = MyTopo()
318             #checks the selected switch
319             if self.switch == 1:
320                 switch = KernelSwitch
321             elif self.switch == 2:
322                 switch = OVSKernelSwitch
323             else:
324                 switch = UserSwitch
```

```
325             #checks the selected controller
326             if self.controller == 1:
327                 controller = Controller
328             elif self.controller == 2:
329                 controller = RemoteController
330             else:
331                 controller = curry (RemoteController, defaultIP=ipcontroller)
332             #creating the network
333             net = Mininet( topo = topo ,controller = controller, switch =
                    switch, autoSetMacs=m, cleanup=True, xterms=x)
334             #start the network
335             net.start()
336             if self.ping == 41:
337                 net.pingAll()
338             if self.iperf == 51:
339                 net.iperf()
340             print "\n
                    ************************************************************
                    "
341             print "don't forget to exit the CLI with the command '<Ctrl>+D' or
                    'exit'"
342             print "
                    ************************************************************\
                    n"
343             info( '*** Running CLI\n' )
344             if self.filename_script == None:
345                 CLI( net )
346             else:
347                 CLI( net, script=self.filename_script )
348                 CLI( net )
349             info( '*** Stopping network' )
350             net.stop()
351
352         def buttonRun_a(self, event):
353             self.buttonRun()
354
355         def button_topology(self):
356             self.filename_topo = askopenfilename(filetypes=[("allfiles","*"),("
                    pythonfiles","*.py")])
357             self.text4_topology.delete(0, END)
358             self.text4_topology.insert(0,self.filename_topo)
359
360         def button_topology_a(self, event):
361             self.button_topology()
362
363         def button_script(self):
364             self.filename_script = askopenfilename(filetypes=[("allfiles","*")
                    ,("pythonfiles","*.py")])
365             self.text_script.delete(0,END)
366             self.text_script.insert(0,self.filename_script)
367
368         def button_script_a(self, event):
369             self.button_script()
370
371         def radioController(self, controller):
372             if controller == 3:
```

```
373             self.text_ip_controller.config(state=NORMAL)
374             self.text_ip_controller.configure(background='white')
375         else:
376             self.text_ip_controller.delete(0, END)
377             self.text_ip_controller.config(state=DISABLED)
378             self.text_ip_controller.configure(background='gray')
379         self.controller = controller
380 
381     def radioSwitch(self, switch):
382         self.switch = switch
383 
384     def radioTopology(self, topology):
385         if topology == 1:
386             self.text1_topology.config(state=NORMAL)
387             self.text1_topology.configure(background='white')
388             self.text21_topology.config(state=DISABLED)
389             self.text21_topology.configure(background='gray')
390             self.text22_topology.config(state=DISABLED)
391             self.text22_topology.configure(background='gray')
392             self.text3_topology.config(state=DISABLED)
393             self.text3_topology.configure(background='gray')
394             self.text4_topology.config(state=DISABLED)
395             self.text4_topology.configure(background='gray')
396             self.button_topology.config(state=DISABLED)
397         elif topology == 2:
398             self.text21_topology.config(state=NORMAL)
399             self.text21_topology.configure(background='white')
400             self.text22_topology.config(state=NORMAL)
401             self.text22_topology.configure(background='white')
402             self.text1_topology.config(state=DISABLED)
403             self.text1_topology.configure(background='gray')
404             self.text3_topology.config(state=DISABLED)
405             self.text3_topology.configure(background='gray')
406             self.text4_topology.config(state=DISABLED)
407             self.text4_topology.configure(background='gray')
408             self.button_topology.config(state=DISABLED)
409         elif topology == 3:
410             self.text3_topology.config(state=NORMAL)
411             self.text3_topology.configure(background='white')
412             self.text1_topology.config(state=DISABLED)
413             self.text1_topology.configure(background='gray')
414             self.text21_topology.config(state=DISABLED)
415             self.text21_topology.configure(background='gray')
416             self.text22_topology.config(state=DISABLED)
417             self.text22_topology.configure(background='gray')
418             self.text4_topology.config(state=DISABLED)
419             self.text4_topology.configure(background='gray')
420             self.button_topology.config(state=DISABLED)
421         else:
422 
423             self.text4_topology.config(state=NORMAL)
424             self.text4_topology.configure(background='white')
425             self.button_topology.config(state=NORMAL)
426             self.text1_topology.config(state=DISABLED)
427             self.text1_topology.configure(background='gray')
428             self.text21_topology.config(state=DISABLED)
```

```
429             self.text21_topology.configure(background='gray')
430             self.text22_topology.config(state=DISABLED)
431             self.text22_topology.configure(background='gray')
432             self.text3_topology.config(state=DISABLED)
433             self.text3_topology.configure(background='gray')
434         self.topology = topology
435
436     def checkScript(self):
437         if self.script == 10:
438             self.script = 11
439             self.text_script.config(state=NORMAL)
440             self.text_script.configure(background='white')
441             self.button_script.config(state=NORMAL)
442         else:
443             self.script = 10
444             self.text_script.delete(0, END)
445             self.text_script.config(state=DISABLED)
446             self.text_script.configure(background='gray')
447             self.button_script.config(state=DISABLED)
448             self.filename_script = None
449
450     def checkXterm(self):
451         if self.xterm == 20:
452             self.xterm = 21
453         else:
454             self.xterm = 20
455
456     def checkMac(self):
457         if self.mac == 30:
458             self.mac = 31
459         else:
460             self.mac = 30
461
462     def checkPing(self):
463         if self.ping == 40:
464             self.ping = 41
465         else:
466             self.ping = 40
467
468     def checkIperf(self):
469         if self.iperf == 50:
470             self.iperf = 51
471         else:
472             self.iperf = 50
473
474 if __name__ == '__main__':
475     setLogLevel( 'info' )
476     print "\n"*100
477     print "Starting..."
478     root = Tk()
479     myapp = MiniNetGui(root)
480     root.mainloop()
481     print "... Done!"
```

codigos/minigui.py