Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento de pesquisas em Redes de Computadores

Chapter · November 2014

CITATIONS
14

READS
9,772

5 authors, including:

Dorgival Olavo Guedes
Federal University of Minas Gerais
157 PUBLICATIONS **1,562** CITATIONS
SEE PROFILE

Luiz Vieira
Federal University of Minas Gerais
153 PUBLICATIONS **4,393** CITATIONS
SEE PROFILE

Marcos Augusto M. Vieira
Federal University of Minas Gerais
181 PUBLICATIONS **2,681** CITATIONS
SEE PROFILE

Rafaela Natalia da Silva Nunes
61 PUBLICATIONS **607** CITATIONS
SEE PROFILE

Capítulo 4

Redes Definidas por Software: uma abordagem sistêmica para o desenvolvimento de pesquisas em Redes de Computadores

Dorgival Guedes, Luiz Filipe Menezes Vieira, Marcos Menezes Vieira, Henrique Rodrigues e Rogério Vinhal Nunes

Abstract

Software Defined Networks (SDN) are a new paradigm for the development of research in computer networks, which has been getting the attention of the academic community and the network industry. A lot of the attention so far has focused on the OpenFlow standard, one of the elements which make this approach possible. However, Software Defined Networks go way beyond OpenFlow, creating new perspectives in terms of abstractions, control environments and network applications which can be developed easily and without the limitations of the current network technologies. This short course take a systemic approach to the topic, with theoretical and practical aspects. Considering the theory, we discuss the various components of a software-defined network system, including issues such as network element virtualization, the structure of the network operating system and its applications, as well as the challenges this new approach still has to face, and the ongoing research efforts in Brazil and around the world. To illustrate the new development possibilities offered by the new paradigm, the practical part of the course focuses on the POX network operating system, created with research and teaching in mind.

Resumo

Redes Definidas por Software (Software Defined Networks, ou SDN) constituem um novo paradigma para o desenvolvimento de pesquisas em redes de computadores que vem ganhando a atenção de grande parte da comunidade acadêmica e da indústria da área. Muita da atenção até o momento tem sido voltada para o padrão OpenFlow, um dos elementos que tornaram possível esse enfoque. Entretanto, Redes Definidas por Software vão além de OpenFlow, abrindo novas perspectivas em termos de abstrações, ambientes de controle e aplicações de rede que podem ser desenvolvidas de forma simples e livre

das limitações das tecnologias de rede atuais. Este minicurso adota uma abordagem sistêmica da área, com aspectos de teoria e prática. Na parte teórica, discutimos os diversos componentes de um sistema de rede definido por software, incluindo soluções para virtualização dos elementos de rede, sistemas operacionais de rede e novas aplicações, bem como os desafios de pesquisa que esse paradigma ainda precisa enfrentar e os diversos esforços de pesquisa em andamento no Brasil e no mundo. Para ilustrar as novas possibilidades de desenvolvimento que o paradigma oferece, a parte prática foca no sistema operacional de redes POX, desenvolvido especificamente para fins de pesquisa e ensino.

4.1. Introdução

A comunidade de redes se encontra hoje em uma situação complexa: o sucesso da área pode ser considerado estrondoso, já que hoje a tecnologia de redes de computadores permeia todos os níveis da sociedade. A maioria das atividades da sociedade hoje de alguma forma atravessa uma ou mais redes de computadores. A tecnologia está nos lares, na forma de redes domiciliares, nas rotinas de implementação de políticas públicas, na forma do governo eletrônico, na educação, onde a Web se tornou uma das fontes essenciais de informação para os estudantes nos diversos níveis. A Internet se tornou um artefato conhecido e acessado por uma fração significativa da população e iniciativas de inclusão digital são desenvolvidas em diversas esferas com o objetivo de expandir seu alcance, idealmente a toda a população mundial.

Tamanho sucesso, entretanto, traz consigo um problema para a comunidade de pesquisa. Como grande parte da sociedade depende hoje da Internet em suas atividades do dia-a-dia e tecnologias de acesso à rede se tornaram *commodities* de fácil acesso, estabilidade se tornou uma característica essencial da Internet. Isso significa que pesquisas com novos protocolos e tecnologias não são mais possíveis na Internet em geral, devido ao risco de interrupção das atividades para as quais ela já se tornou ferramenta essencial. Não só isso, mas também a economia de escala possível pelo crescimento da rede e a larga adoção das tecnologias já desenvolvidas inviabiliza a inserção de novas tecnologias que dependam de alterações do hardware a ser utilizado.

Mesmo pesquisadores trabalhando em iniciativas como *backbones* de pesquisa básica, como a Internet2, se vêm frente a um problema complexo para justificar a adoção em larga escala das tecnologias desenvolvidas nesses ambientes. O potencial de ruptura de tais avanços se torna um forte argumento contra sua adoção.

Esses problemas levaram diversos pesquisadores a afirmar que a arquitetura de redes de computadores em geral e a rede mundial (a Internet) atingiram um nível de amadurecimento que as tornaram pouco flexíveis. A expressão usada em muitos casos é que a Internet está calcificada (*ossified*, em inglês), referindo-se ao processo que substitui cartilagens (mais elásticas) por ossos ao longo do envelhecimento dos seres vivos.

Para tentar contornar esse problema, a comunidade de pesquisa em Redes de Computadores tem investido em iniciativas que levem à implantação de redes com maiores recursos de programação, de forma que novas tecnologias possam ser inseridas na rede de forma gradual. Exemplos de iniciativas desse tipo são as propostas de redes ativas (*active networks*) [Tennenhouse and Wetherall 2007], de *testbeds* como

o PlanetLab [Peterson and Roscoe 2006] e, mais recentemente, do GENI [Turner 2006, Elliott and Falk 2009]. Redes ativas, apesar do seu potencial, tiveram pouca aceitação pela necessidade de alteração dos elementos de rede para permitir que se tornassem programáveis. Iniciativas mais recentes, como PlanetLab e GENI, apostam na adoção de recursos de virtualização para facilitar a transição para novas tecnologias. Apesar de serem consideradas de grande potencial no longo prazo, tais iniciativas ainda enfrentam desafios em questões como garantir o desempenho exigido pelas aplicações largamente utilizadas hoje utilizando-se tais elementos de rede virtualizados.

Uma outra forma de abordar o problema, a fim de oferecer um caminho de menor impacto e que possa ser implementado em prazos mais curtos e com bom desempenho, consiste em estender o hardware de encaminhamento de pacotes de forma mais restrita. Considerando-se que a operação que precisa de alto desempenho nos elementos de comutação atual é o encaminhamento de pacotes, algumas iniciativa propõem manter essa operação pouco alterada, para manter a viabilidade de desenvolvimento de hardware de alto desempenho, mas com uma possibilidade de maior controle por parte do administrador da rede. Essa proposta se inspira em uma tecnologia já largamente adotada atualmente, o chaveamento (encaminhamento) baseado em rótulos programáveis, popularizado pelo MPLS (*Multi-protocol Label Switching*) [Davie and Farrel 2008, Kempf et al. 2011].

Com MPLS, o controle fino sobre o tráfego de rede se torna possível ao se atribuir a cada pacote um rótulo (*label*) que determina como o mesmo será tratado pelos elementos de rede. Explorando esse recurso, administradores de rede podem exercer controle diferenciado sobre cada tipo de tráfego de rede, assumindo que os mesmos possam ser identificados para receberem rótulos apropriados. Com base nessa observação, uma ideia trabalhada por diversos pesquisadores é a manutenção de um hardware de encaminhamento de alto desempenho, com a possibilidade de permitir que o administrador de rede (ou o desenvolvedor de aplicações para a rede) determine como fluxos sejam rotulados e encaminhados.

4.1.1. Origens

A iniciativa mais bem sucedida nesse sentido foi, sem dúvida, a definição da interface e do protocolo OpenFlow [McKeown et al. 2008]. Com OpenFlow, os elementos de encaminhamento oferecem uma interface de programação simples que lhes permite estender o acesso e controle da tabela de consulta utilizada pelo hardware para determinar o próximo passo de cada pacote recebido. Dessa forma, o encaminhamento continua sendo eficiente, pois a consulta à tabela de encaminhamento continua sendo tarefa do hardware, mas a decisão sobre como cada pacote deve ser processado pode ser transferida para um nível superior, onde diferentes funcionalidades podem ser implementadas. Essa estrutura permite que a rede seja controlada de forma estensível através de aplicações, expressas em software. A esse novo paradigma, deu-se o nome de Redes Definidas por Software, ou *Software Defined Networks* (SDN).

Do ponto de vista histórico, SDNs têm sua origem na definição da arquitetura de redes *Ethane*, que definia uma forma de se implementar políticas de controle de acesso de forma distribuída, a partir de um mecanismo de supervisão centrali-

zado [Casado et al. 2009]. Naquela arquitetura, cada elemento de rede deveria consultar o elemento supervisor ao identificar um novo fluxo. O supervisor consultaria um grupo de políticas globais para decidir, com base nas características de cada fluxo, como o elemento de encaminhamento deveria tratá-lo. Essa decisão seria comunicada ao comutador na forma da programação de uma entrada em sua tabela de encaminhamento com uma regra adequada para o novo fluxo (que poderia, inclusive, ser seu descarte). Esse modelo foi posteriormente formalizado por alguns dos autores na forma da arquitetura OpenFlow.

4.1.2. Motivação

Desde seu surgimento, diversos pesquisadores da área de Redes de Computadores e empresas voltadas para o desenvolvimento de equipamentos e sistemas de gerência de redes têm voltado sua atenção para o paradigma de Redes Definidas por Software (SDN). O *Open Networking Summit*, um encontro de pesquisadores e profissionais da indústria, organizado recentemente pela Universidade de Stanford e a empresa Nicira, teve um número de participantes muito acima do esperado pelos organizadores, resultando em 250 inscritos e uma lista de espera de mais de 250 nomes. Outro indicador desse interesse é o número de publicações em conferências da área que abordam algum elemento de Redes Definidas por Software. No SBRC 2011, por exemplo, mais de 10 trabalhos focaram aspectos da área.

Esse interesse se deve às diversas possibilidades de aplicação do paradigma, que se apresenta como uma forma potencial de implantação do que se convencionou denominar Internet do Futuro. Com a proposta de uma interface de programação para os elementos de comutação da rede, como o padrão OpenFlow, fabricantes e pesquisadores abraçaram a ideia de uma estrutura de redes onde a comutação de pacotes não precisa mais ser definida pelo princípio de roteamento de redes Ethernet ou IP, mas que pode ser controlado por aplicações (software) desenvolvido independentemente do hardware de rede. Além de um modelo de referência de um comutador OpenFlow implementado como um processo de usuário e de uma implementação de um comutador no espaço do kernel para ambientes virtualizados, o Open vSwitch, diversos fabricantes já oferecem no mercado produtos que implementa a interface OpenFlow.

Um aspecto importante é que Redes de Definidas por Software constituem um universo muito maior que aquele definido pelo padrão OpenFlow. OpenFlow oferece, por exemplo, uma solução simples para a criação de diversas redes virtuais sobre uma infraestrutura física, onde cada rede virtual é composta por *switches* nível 2, roteadores e *gateways* de aplicação tradicionais. Mas o paradigma de Redes Definidas por Software também abre a possibilidade de se desenvolver novas aplicações que controlem os elementos de comutação de uma rede física de formas impensadas no passado.

Para se desenvolver essas novas aplicações, é importante se compreender a funcionalidade de controladores de rede (também denominados Sistemas Operacionais de Rede ou *Network Hypervisors*). Esses elementos oferecem um ambiente de programação onde o desenvolvedor pode ter acesso aos eventos gerados por uma interface de rede que siga um padrão como OpenFlow e pode também gerar comandos para controlar a infraestrutura de chaveamento. Com esse ambiente, torna-se mais simples implementar políticas de segurança baseadas em níveis de abstração maiores que endereços Ethernet ou IP, que

cubram todos os pontos de rede, por exemplo. Da mesma forma, com SDNs é possível implementar controladores de rede que implementem lógicas de monitoração e acompanhamento de tráfego mais sofisticadas, ou soluções que ofereçam novas abstrações para os usuários da rede, como cada usuário de um *datacenter* ter a visão de que todas as suas máquinas estão ligadas a um *switch* único e privado, independente dos demais.

Este minicurso foi concebido tendo essas possibilidades em mente, com o objetivo de oferecer aos alunos, pesquisadores e profissionais da área uma introdução ao paradigma e às oportunidades que ele oferece. Para isso apresentar o POX, um controlador desenvolvido especificamente para o ambiente de pesquisa e ensino, com a proposta de ser facilmente extensível e de fácil entendimento.

4.1.3. Organização do texto

Com a motivação apresentada na seção anterior em mente, este material está organizado nas seguintes seções:

1. Introdução — esta seção, cujo objetivo foi introduzir o tema de forma abstrada;

2. Componentes de um sistema baseado em SDNs — apresenta uma definição mais refinada do conceito de SDN, abordando os elementos normalmente identificados com o paradigma de forma a dar ao leitor uma visão geral do papel de cada componente e das relações entre eles;

3. Programação dos elementos de rede — discute os requisitos mínimos da interface de programação, com foco principal na definição do modelo OpenFlow;

4. Particionamento de recursos (virtualização) — detalha o conceito de *slicing* da rede e seu uso para permitir a implementação de redes virtuais;

5. Controladores de rede — aborda os principais projetos de controladores disponíveis para a implementação de SDNs;

6. Aplicações — discute como o conceito de SDN pode ser aplicado a diferentes contextos e lista iniciativas de pesquisa relacionadas no Brasil e no mundo;

7. Desenvolvimento de sistemas utilizando POX — apresenta os detalhes da arquitetura do controlador POX e os elementos principais para a programação do mesmo;

8. Desafios de pesquisa — discute desafios para a arquitetura SDN que devem ser abordados para viabilizar sua aplicação nos diversos contextos;

9. Considerações finais — oferece um fechamento da discussão, amarrando os diversos conceitos e pontos de vista apresentados.

Ao final deste trabalho, esperamos que os participantes tenham uma visão das possibilidades da área e um entendimento de como um controlador como POX pode ser utilizado para colocar em prática diferentes ideias.

4.2. Componentes de um sistema baseado em SDNs

Com base na discussão anterior, de forma abstrata, podemos dizer que uma Rede Definida por Software (SDN) é caracterizada pela existência de um sistema de controle (software) que pode controlar o mecanismo de encaminhamento dos elementos de comutação da rede por uma interface de programação bem definida. De forma mais específica, os elementos de comutação exportam uma interface de programação que permite ao software inspecionar, definir e alterar entradas da tabela de roteamento do comutador como ocorre, por exemplo, com comutadores OpenFlow. O software envolvido, apesar de potencialmente poder ser uma aplicação monolítica especialmente desenvolvida, na prática tende a ser organizado com base em um controlador de aplicação geral, em torno do qual se contróem aplicações específicas para o fim de cada rede. É possível, ainda, utilizar-se um divisor de visões para permitir que as aplicações sejam divididas entre diferentes controladores. A figura 4.1 ilustra essa organização, para uma rede com elementos com e sem fio.

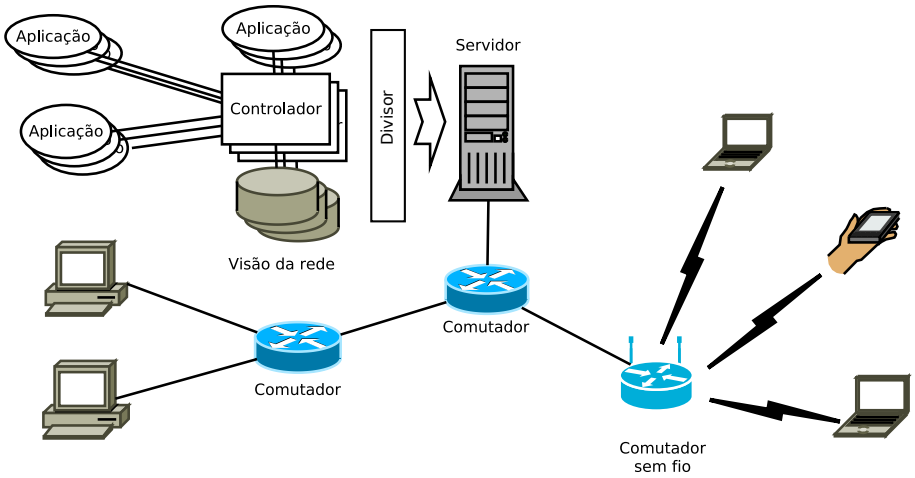

Figura 4.1. Estrutura geral de uma SDN

4.2.1. Elementos de comutação programáveis

Obviamente, o princípio básico de redes definidas por software é possibilidade de programação dos elementos de rede. Como mencionado anteriormente, essa programação, ao contrário de propostas anteriores, como Redes Ativas, se restringe a uma manipulação simples de pacotes baseado no conceito de fluxos, ou seja, sequências de pacotes que compartilham atributos com valores bem definidos. A definição exata do que constitui um fluxo é função exatamente dos recursos oferecidos pela interface de programação.

A operação de encaminhamento dos elementos de comutação em redes baseadas em pacotes segue um princípio simples: cada pacote recebido em uma das interfaces do comutador é inspecionado e gera uma consulta à tabela de encaminhamento do comutador. No caso de *switches* Ethernet, essa consulta é baseada no endereço de hardware (MAC) de destino do pacote; em roteadores IP, em um prefixo do endereço IP de destino; em redes ATM, nos valores dos campos VCI/VPI da célula ATM, etc. Caso a consulta não tenha sucesso, o pacote é descartado. Usualmente, há ainda a possibilidade de se definir um comportamento padrão (*default*) nos casos de insucesso. Uma vez identificado o

destino do pacote, o mesmo atravessa as interconexões internas do comutador para atingir a porta de destino, onde ela é enfileirada para transmissão.

Ao longo da evolução das redes de computadores, esse processo de consulta (*lookup*) e chaveamento (*switching*) já foi amplamente estudado, resultando hoje em soluções baseadas usualmente em hardware com desempenho suficiente para acompanhar as taxas de transmissão do meio (*wire speed switching*).

4.2.2. Divisores de recursos/visões

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. Tornou-se viável, por exemplo, manter um comportamento tradicional para fluxos denominados "de operação" e dar tratamento diferente para fluxos "de pesquisa". Uma nova possibilidade, imediatamente identificada por pesquisadores, é a de estender essa divisão para permitir que diferentes tipos de pesquisa sejam colocados em operação em paralelo, na mesma rede física. Para isso, seria necessário estender o modelo com a noção de visões diferentes da rede, dividindo os recursos entre diferentes controladores.

Estendendo o modelo de comportamento dos elementos de chaveamento, baseados em consultas usando campos do pacote, o processo de divisão de recursos se baseia exatamente em definir partições do espaço de consulta disponível. Isso pode ser visto como uma extensão do princípio de particionamento baseado em padrões parciais: alguns dos bits usados no processo de consulta são utilizados para definir espaços que devem ser tratados de forma isolada. Essa prática é basicamente uma extensão do princípio de particionamento do tráfego Internet baseado em número de porto, por exemplo.

4.2.3. Controlador

Uma vez definida uma interface de programação para os elementos de chaveamento, seria possível desenvolver uma aplicação que utilizasse diretamente as chamadas dessa interface para controlar cada *switch* separadamente. Entretanto, esse enfoque traz limitações semelhantes àquelas associadas ao desenvolvimento de software diretamente ligado a um elemento de hardware: o desenvolvimento exige que o programador lide com tarefas de baixo nível na interface com o dispositivo e o sofware resultante é normalmente muito dependente do hardware adotado. Novos desenvolvimentos exigem que todas as funcionalidades de baixo nível sejam reimplementadas.

Essas limitações levaram à identificação da necessidade de um novo nível na arquitetura, que concentre as tarefas de manipulação direta dos dispositivos de rede e ofereça uma abstração de mais alto nível para o desenvolvedor. Esse elemento, talvez mais que a arquitetura OpenFlow, define a natureza de uma rede definida por software. Em uma analogia direta das funcionalidades de um sistema operacional, esse elemento age como um sistema operacional para a rede: provendo o controle direto dos dispositivos de rede, oferecendo uma interface mais eficiente para os desenvolvedores, isolando os detalhes de cada componente.

O controlador de rede, ou sistema operacional de rede, ou, ainda, *hypervisor* da rede (em alusão ao conceito derivado da area de sistemas virtualizados) pode, dessa

forma, concentrar a comunicação com todos os elementos programáveis que compõem a rede e oferecer uma visão unificada do estado da rede [Casado et al. 2010b]. Sobre essa visão, comandos podem ser executados e outros programas podem ser desenvolvidos que implementem novas funcionalidades. Um dos pontos fortes de SDNs é exatamente a formação dessa visão centralizada das condições da rede, sobre a qual é possível desenvolver análises detalhadas e chegar a decisões operacionais sobre como o sistema como um todo deve operar. A existência dessa visão global simplifica a representação dos problemas e a tomada de decisão, bem como a expressão das ações a serem tomadas.

É importante observar que essa noção de visão única e centralizada da rede como expressa pelo sistema operacional da rede é uma visão lógica. Não necessariamente exige que o controlador opere como um elemento concentrador, fisicamente localizado em um ponto único do sistema. A abstração de visão global pode muito bem ser implementada de forma distribuída, seja pela divisão dos elementos da visão entre domínios diferentes, seja pela implementação de um controlador realmente distribuído, que se valha de algoritmos de consenso, por exemplo, para manter a visão consistente entre suas partes.

Diferentes tipos de controladores de rede já foram desenvolvidos dentro do contexto de redes definidas por software. Muitos deles se apresentam como ambientes de tempo de execução (*run-time systems*) que oferecem uma interface imperativa para programação da rede. A linguagem utilizada, nesse caso, determina em grande parte o estilo de desenvolvimento e as funcionalidades oferecidas. Outros controladores, entretanto, ultrapassam a noção de uma interface de programação imperativa e oferecem novas abstrações, como um ambiente de programação funcional ou declarativa. Nesses casos, os recursos da linguagem utilizada se prestam à implementação de novas funcionalidades, como detecção de conflitos ou depuração automatizada da rede.

Muitos controladores já implementados, desenvolvidos sem a preocupação básica de escalabilidade frente a grandes demandas, optam pela estrutura centralizada por simplicidade. Por outro lado, alguns dos esforços de desenvolvimento voltados para a implantação em grandes sistemas, como os *datacenters* de grandes corporações, utilizam diferentes formas de distribuição para garantir a escalabilidade e disponibilidade do sistema.

4.2.4. Aplicações de rede

Nesse contexto, considerando os controladores de SDN como o sistema operacional da rede, o software desenvolvido para criar novas funcionalidades pode ser visto com a aplicação que executa sobre a rede física. Valendo-se da visão unificada dos recursos de rede que o controlador oferece, pode-se, por exemplo, implementar soluções de roteamento especialmente desenhadas para um ambiente particular, controlar a interação entre os diversos comutadores, oferecendo para os computadores a eles ligados a impressão de estarem ligados a um único *switch*, ou a um único roteador IP.

4.3. Programação dos elementos de rede

O princípio por trás de Redes Definidas por Software é a capacidade de se controlar o plano de encaminhamento de pacotes através de uma interface bem definida. Sem dúvida, a interface associada ao paradigma desde seu início (de fato, um dos elementos motivado-

res da sua criação) é OpenFlow, apesar de não ser a única forma de se implementar uma SDN.

4.3.1. O padrão OpenFlow

O OpenFlow é um padrão aberto para Redes Definidas por Software que tem como principal objetivo permitir que se utilize equipamentos de rede comerciais para pesquisa e experimentação de novos protocolos de rede, em paralelo com a operação normal das redes. Isso é conseguido com a definição de uma interface de programação que permite ao desenvolvedor controlar diretamente os elementos de encaminhamento de pacotes presentes no dispositivo. Com OpenFlow, pesquisadores podem utilizar equipamentos de rede comerciais, que normalmente possuem maior poder de processamento que os comutadores utilizados em laboratórios de pesquisa, para realizar experimentos em redes "de produção". Isso facilita a transferência dos resultados de pesquisa para a indústria.

Uma característica básica do modelo OpenFLow é uma separação clara entre os planos de dados e controle em elementos de chaveamento. O plano de dados cuida do encaminhamento de pacotes com base em regras simples (chamadas de ações na terminologia OpenFlow) associadas a cada entrada da tabela de encaminhamento do comutador de pacotes (um *switch* ou roteador). Essas regras, definidas pelo padrão, incluem (i) encaminhar o pacote para uma porta especifica do dispositivo, (ii) alterar parte de seus cabeçalhos, (iii) descartá-lo, ou (iv) encaminhá-lo para inspeção por um controlador da rede. Em dispositivos dedicados, o plano de dados pode ser implementado em hardware utilizando os elementos comuns a roteadores e *switches* atuais. Já o módulo de controle permite ao controlador da rede programar as entradas dessa tabela de encaminhamento com padrões que identifiquem fluxos de interesse e as regras associadas a eles. O elemento controlador pode ser um módulo de software implementado de forma independente em algum ponto da rede.

4.3.1.1. Estrutura e Possibilidades

Um grande trunfo da arquitetura OpenFlow é a flexibilidade que ela oferece para se programar de forma independente o tratamento de cada fluxo observado, do ponto de vista de como o mesmo deve (ou não) ser encaminhado pela rede. Basicamente, o padrão OpenFlow determina como um fluxo pode ser definido, as ações que podem ser realizadas para cada pacote pertencente a um fluxo e o protocolo de comunicação entre comutador e controlador, utilizado para realizar alterações dessas definições e ações. A união de uma definição de fluxo e um conjunto de ações forma uma entrada da tabela de fluxos OpenFlow [McKeown et al. 2008].

Em um *switch* OpenFlow, cada entrada na tabela de fluxos pode ser implementada como um padrão de bits representado em uma memória TCAM (Ternary Content-Addressable Memory). Nesse tipo de memória, bits podem ser representados como zero, um ou "não importa" (*don't care*), indicando que ambos os valores são aceitáveis naquela posição. Como o padrão é programado a partir do plano de controle, fluxos podem ser definidos da forma escolhida pelo controlador (p.ex., todos os pacotes enviados a partir do endereço físico A para o endereço físico B, ou todos os pacotes TCP enviados da

máquina com endereço IP X para o porto 80 da máquina com endereço IP Y). A figura 4.2 apresenta uma visão geral de uma entrada da tabela OpenFlow. Cada pacote que chega a um comutador OpenFlow é comparado com cada entrada dessa tabela; caso um casamento seja encontrado, considera-se que o pacote pertence àquele fluxo e aplica-se as ações relacionadas à esse fluxo. Caso um casamento não seja encontrado, o pacote é encaminhado para o controlador para ser processado — o que pode resultar na criação de uma nova entrada para aquele fluxo. Além das ações, a arquitetura prevê a manutenção de três contadores por fluxo: pacotes, bytes trafegados e duração do fluxo. Esses contadores são implementados para cada entrada da tabela de fluxos e podem ser acessados pelo controlador através do protocolo.

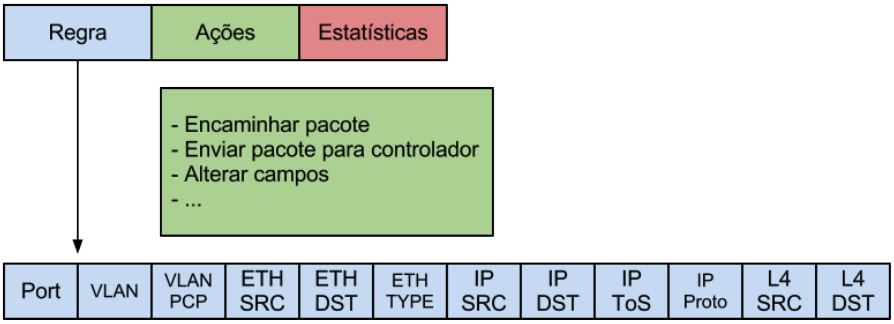

Figura 4.2. Exemplo de uma entrada na tabela de fluxos OpenFlow.

Esse pequeno conjunto de regras cria diversas possibilidades, pois muitas das funcionalidades que são implementadas separadamente podem ser agrupadas em um único controlador OpenFlow, utilizando um pequeno conjunto de regras. Alguns exemplos das possibilidades são apresentadas na figura 4.3. As entradas representam o uso do *switch* OpenFlow para realizar encaminhamento de pacotes na camada de enlace, implementar um *firewall* e realizar encaminhamento de pacotes na camada de enlace utilizando VLANs, respectivamente.

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

Figura 4.3. Exemplos de uso de um Switch OpenFlow.

Apesar de possuir um conjunto pequeno de ações simples, alguns descrevem o OpenFlow com uma analogia ao conjunto de instruções de um microprocessador x86 que, apesar de pequeno e simples, provê uma vasta gama de possibilidades para o desenvolvimento de aplicações. O OpenFlow cria possibilidades semelhantes para o desenvolvimento de aplicações no contexto de redes de computadores.

4.3.1.2. Limitações e futuras versões

A versão atual do OpenFlow é a 1.1, que ainda possui 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. No entanto, a versão 2.0 está sendo formulada e um dos objetivos é eliminar algumas dessas limitações.

Alguns *switches* comerciais já suportam o padrão OpenFlow, como é o caso do HP Procurve 5400zl, do NEC IP880 e do Pronto 3240 e 3290. Espera-se que à medida que a especificação OpenFlow evolua, mais fabricantes de equipamentos de rede incorporem o padrão às suas soluções comerciais.

4.3.2. Implementações em software

A implementação de referência do modelo OpenFlow é um comutador em software, executando no espaço de usuário em uma máquina Linux. Esse modelo tem sido utilizado como base em diversos experimentos, mas carece de um melhor desempenho. Desenvolvido para suprir essa lacuna, o Open vSwitch (OvS) é um switch virtual que segue a arquitetura OpenFlow, implementado em software, com o plano de dados dentro do kernel do sistema operacional Linux, enquanto o plano de controle é acessado a partir do espaço de usuário [Pfaff et al. 2009]. Em particular, essa implementação foi desenvolvida especificamente para controlar o tráfego de rede da camada de virtualização em ambientes virtualizados [Pettit et al. 2010].

Em sua implementação atual (figura 4.4), o Open vSwitch é composto por dois componentes principais: um módulo presente no núcleo do sistema operacional, denominado "Fast Path", e um componente no nível de usuário, o "Slow Path". O *fast path* interage ativamente com o tráfego de rede, pois é responsável por procurar rotas na tabela de fluxos e encaminhar pacotes de rede. Já o *slow path* é o componente do Open vSwitch onde são implementadas as demais funcionalidades associadas ao plano de controle, como as interfaces de configuração do *switch*, a lógica de uma ponte com aprendizado (*learning bridge*) e as funcionalidades de gerência remota, etc. A interação entre os dois módulos se dá prioritariamente através da manipulação da tabela de fluxos.

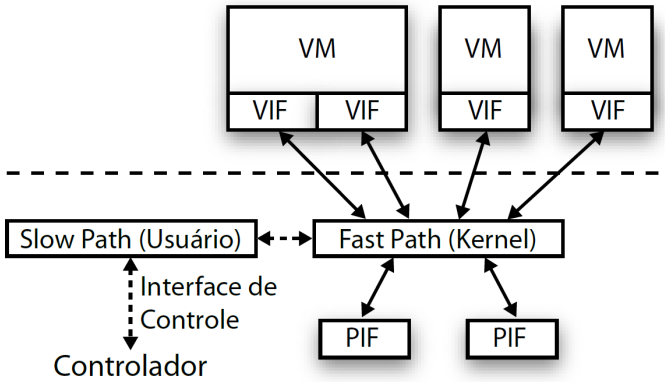

Figura 4.4. Arquitetura do Open vSwitch.

A implementação do *fast path* é simples e seu código é composto por poucas linhas

(em torno de 3000 — um número de linhas bem pequeno quando comparado às 30.000 do *slow path*). Existem dois motivos para essa decisão de projeto, sendo que o primeiro deles é o desempenho. O *fast path* é a parte crítica do sistema quando consideramos o tempo de processamento e, portanto, quanto menor o processamento realizado para cada pacote, maior a capacidade de processamento desse componente. Essa característica torna indispensável a sua implementação como uma parte do sistema operacional, posicionando-o mais próximo às NICs. Além disso, como a lógica implementada pelo *fast path* é dependente das APIs do sistema operacional, manter a sua complexidade pequena facilita a tarefa de portar o Open vSwitch a um outro sistema operacional. O segundo motivo é o esforço reduzido em adaptar as funcionalidades do *fast path* à aceleração de hardware (*hardware acceleration/offoading*), que eventualmente pode ser oferecida pela máquina física.

Além de oferecer as funcionalidades da arquitetura OpenFlow, o OvS, na sua configuração padrão, opera como um *switch* Ethernet normal. Para simplificar a sua integração com o *kernel* Linux, o OvS emula as interfaces de rede daquele sistema e utiliza partes do código fonte de seu módulo *bridge*, que implementa o *switch* de rede padrão do Linux. Como resultado dessas decisões de projeto, o Open vSwitch pode ser utilizado como um substituto imediato para os *switches* virtuais adotados pelos VMMs baseadas em Linux mais utilizados atualmente, como Xen, XenServer e KVM e vem sendo adotado como o *switch* padrão em diversas distribuições, mesmo quando as funcionalidades da arquitetura OpenFlow não são utilizadas.

4.3.3. Outras propostas

Apesar do foco principal dos ambientes de Redes Definidas por Software hoje ser o modelo/protocolo OpenFlow e a forma como ele expõe os recursos do *switch*, há outras possibilidades de implementação de uma interface de programação que atenda os objetivos do paradigma. O paradigma SDN não se limita ao OpenFlow, nem o exige como elemento essencial.

A forma como comutadores MPLS podem ser programados para tratar cada fluxo com base no rótulo a ele associado mostram que essa tecnologia pode ser facilmente estendida para a implantação de uma SDN [Davie and Farrel 2008]. Uma primeira iniciativa nesse sentido utiliza o princípio da interface definida por OpenFlow sobre o comportamento usual de roteadores MPLS [Kempf et al. 2011]. Nesse caso, os autores estendem o modelo da tabela de encaminhamento do plano de dados de OpenFlow para incluir a noção de porto virtual. Esse tipo de porto virtual armazena internamente as noções de encapsulamento e desencapsulamento necessários ao processamento de pacotes com MPLS.

Uma segunda proposta é o conceito de interface de rede sNICh [Ram et al. 2010]. Originalmente, sNICh foi apresentado como uma proposta para uma nova arquitetura de rede para ambientes virtualizados, onde a divisão do plano de dados é feita de forma a dividir as tarefas de encaminhamento entre o *host* e a interface de rede. Essa divisão visa garantir a eficiência do encaminhamento entre máquinas virtuais no mesmo hospedeiro e a rede. Por se apresentar como uma opção para o uso de *switches* de software como o Open vSwitch, mantendo uma estrutura programável, é possível se imaginar que a interface definida para essa arquitetura também possa ser usada para o controle de roteamento a

partir de um módulo de software como um controlador SDN.

Finalmente, há também propostas para novas implementações que alteram a divisão de tarefas entre controlador e os *switches*, como no caso da arquitetura DevoFlow [Mogul et al. 2010]. O argumento dos autores é que no padrão OpenFlow tradicional, a necessidade de que todos os fluxos sejam acessíveis para o controlador SDN impõe demandas sobre o hardware e limitações de desempenho que podem não ser aceitáveis em alguns casos, em particular para fluxos de alto desempenho e que podem ser definidos por regras simples. Dessa forma, DevoFlow reduz o número de casos em que o controlador precisa ser acionado, aumentando a eficiência. Muitas das soluções propostas para essa arquitetura, inclusive, podem ser aplicadas diretamente em implementações OpenFlow tradicionais. Um exemplo disso é a adoção de regras específicas (sem o uso de coringas, como todos os elementos de identificação do fluxo instanciados para o fluxo particular a ser tratado) para cada fluxo que seja identificado pelo controlador.

4.4. Particionamento de recursos (virtualização)

Redes definidas por software, ao definirem uma forma flexível de se alterar a funcionalidade da rede, abriram espaço para que pesquisadores e desenvolvedores usassem a rede como ambiente de testes. Entretanto, ao se conectar os elementos de comutação de uma rede a um controlador único, a capacidade de se desenvolver e instalar novas aplicações na rede fica restrita ao responsável por aquele controlador. Mesmo que o acesso a esse elemento seja compartilhado entre diversos pesquisadores, ainda há a questão da garantia de não-interferência entre as diversas aplicações e a restrição de que todos devem utilizar a mesma interface, sem opções.

Para se resolver esse problema, surgiu a demanda pela capacidade de se poder dividir a rede em fatias (*slices*) e atribuir cada fatia a um controlador diferente. Uma forma de se fazer esse tipo de divisão de recursos de rede em contextos usuais é através do uso de VLANs (redes locais virtuais), onde um cabeçalho especial é usado para definir que cada pacote pertença a uma rede virtual particular. O uso de VLANs, entretanto, tem suas próprias limitações e está amarrado em sua definição à tecnologia Ethernet (ou IEEE 802.x). Essa amarração torna complexo sua aplicação em contextos onde as fatias devem se estender por mais de uma tecnologia de rede.

Considerando a analogia de que o controlador de SDN se assemelha a um sistema operacional da rede, uma forma de se implementar fatias na rede seria lançando mão do princípio de virtualização. Dessa forma, os recursos da rede seriam virtualizados e apresentados de forma isolada para cada desenvolvedor que desejasse ter seu próprio controlador sobre a rede. Como no caso da virtualização de máquinas físicas, a virtualização também pode ser implementada sobre a rede física em diversos níveis, por exemplo, abaixo do controlador ou acima do mesmo, como nos casos de virtualização do hardware ou virtualização do sistema operacional.

A primeira solução a ganhar larga aplicação em SDNs foi a divisão direta dos recursos OpenFlow da rede física (virtualização do hardware), através do FlowVisor [Sherwood et al. 2009, Sherwood et al. 2010]. Nessa solução, o FlowVisor opera como um controlador de rede responsável apenas pela divisão do espaço de endereçamento disponível na rede OpenFlow. Fatias da rede são definidas pela união,

interseção e/ou diferença de valores dos diversos campos disponíveis para identificação de fluxos OpenFlow, conforme ilustrado na figura 4.5. A definição dessas fatias contituem regras, as quais definem as políticas de processamento.

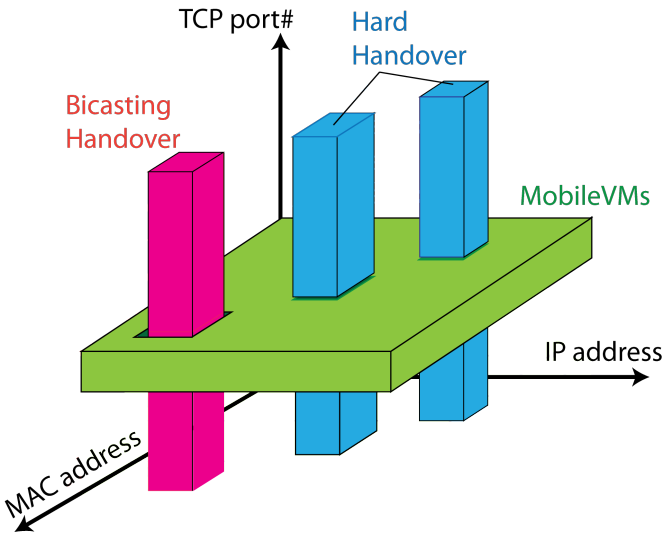

Figura 4.5. FlowVisor: definição de fatias com base em três atributos de fluxos. Na figura, o domínio *bicasting handover* corresponde a um conjunto definido por uma faixa de endereços MAC e uma faixa de endereços IP; *mobile VMs* engloba todos os pacotes com uma certa faixa de números de porto, exceto se contidos na faixa de *bicasting handover*; finalmente, *hard handover* engloba dois conjuntos com endereços IP e MAC em faixas, exceto nos casos em que os portos correspondem à faixa de *moble VMs*.

Uma instância de FlowVisor se coloca entre os diversos controladores e os dispositivos físicos. Comandos dos controladores são observados para se certificar que as regras por eles geradas não excedam a definição do domínio daquele controlador na rede. Se necessário, elas são re-escritas para estender os padrões utilizados com a definição do domínio. Por outro lado, mensagens enviadas pelos *switches* OpenFlow são inspecionadas e direcionadas para o controlador OpenFlow apropriado em função do seu domínio de origem. A figura 4.6 ilustra esse processo. Como um controlador, é possível inclusive a construção de hierarquias de FlowVisors, cada um dividindo uma seção de um domínio definido por uma instância de um nível anterior.

Flowvisor, entretanto, é apenas uma das formas possíveis de se dividir os recurso físicos de uma SDN. Também é possível dotar o controlador de recurso de particionamento de recursos, apesar dessa solução ser menos adotada até o momento. Além disso, o conceito de virtualização também pode ser aplicado aos elementos de rede visíveis para as aplicações. Por exemplo, um controlador pode organizar certas portas de um conjunto de portas espalhadas por diversos comutadores da rede física subjacente e oferecer a visão para a aplicação SDN de um único switch, onde as diversas portas estejam "diretamente" interligadas. Esse tipo de abstração pode ser útil, por exemplo, para oferecer domínios de isolamento de tráfego (apenas as portas pertencentes a tal *switch* virtual estariam no mesmo domínio de broadcast Ethernet, por exemplo. Esse tipo de aplicação de virtualização é previsto, por exemplo, para o controlador POX, apesar de ainda não estar

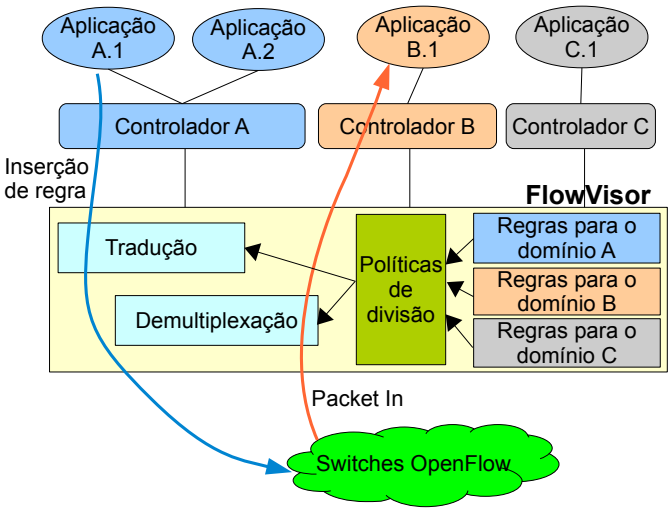

Figura 4.6. FlowVisor: organização do sistema e fluxo de comandos. Um comando de inserção de regra originado em uma aplicação do domínio A passa pelo FlowVisor, que traduz a regra para garantir que ela se aplique apenas ao domínio de A. Um pacote encaminhado por um *switch* OpenFlow para o controlador é processado pelo demultiplexador para identificar qual o controlador do nível acima que é reponsável por aquela faixa dos atributos de entrada.

completamente implementado.

4.5. Controladores de rede

Nesta seção são descritos os principais controladores SDN que existem atualmente. Apresentamos uma breve motivação por trás de cada um deles, suas características principais e, em alguns casos, exemplos simples de programação ou uma descrição dos elementos principais da API.

4.5.1. NOX

NOX é o controlador original OpenFlow e tem como principal função hoje o desenvolvimento de controladores eficientes em C++. Ele opera sobre o conceito de fluxos de dados. Ele checa o primeiro pacote de cada fluxo e procura na tabela de fluxos para determinar a política a ser aplicada. O controlador gerencia o conjunto de regras instaladas nos *switches* da rede reagindo a eventos de rede. Atualmente a maioria dos projetos de pesquisa na área são baseados no NOX, que é um sistema operacional simples para redes e que provê primitivas para o gerenciamento de eventos bem como as funções para a comunicação com os switches [Gude et al. 2008].

NOX define uma série de eventos:

- *packet_in(switch; port; packet),* acionado quando o *switch* envia um pacote recebido por uma porta física para o controlador.
- *stats_in(switch; xid; pattern; packets; bytes)* acionado quando o *switch* retorna os contadores de pacotes e bytes em resposta a um pedido pelas estatísticas associadas

às regras contidas no padrão *pattern*. O parâmetro *xid* representa um identificador para o pedido.

- *flow_removed(switch; pattern; packets; bytes)* acionado quando uma regra com padrão *pattern* supera o seu tempo limite e é removido da tabela de fluxo do *switch*. Os parâmetros *packets* e *bytes* contêm os valores dos contadores para a regra.
- *switch_join(switch)* acionado quando o *switch* entra na rede.
- *switch_exit(switch)* acionado quando o *switch* sai da rede.
- *port_change(switch; port; up),* acionado quando o enlace ligado a uma dada porta física é ligado ou desligado. O parâmetro *up* representa o novo status do enlace.

NOX também provê funcionalidades para enviar mensagens aos *switches*:

- *install (switch; pattern; priority; timeout; actions)* insere uma regra com o dado padrão, prioridade, tempo limite e ações na tabela de fluxos do *switch*.
- *uninstall (switch; pattern)* remove a regra contida em padrão da tabela de fluxos do *switch*.
- *send(switch; packet; action)* envia o dado pacote para o *switch* e aplica a ação lá.
- *query_stats(switch; pattern)* gera um pedido para estatísticas de uma regra contida no padrão no *switch* e retorna um identificador de requisição *xid* que pode ser usado para mapear uma resposta assíncrona do *swtich*.

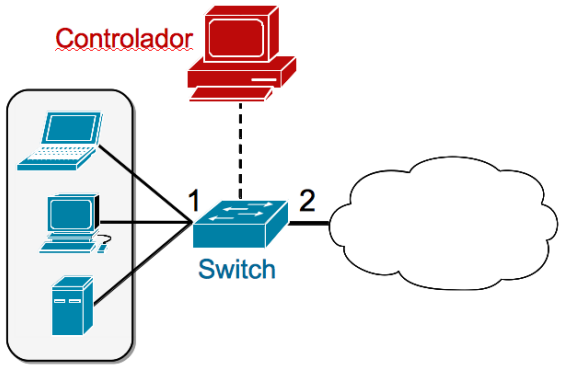

Figura 4.7. Topologia simples

O programa em execução no controlador define um manipulador para cada um dos eventos construídos dentro do NOX, mas pode ser estruturado como um programa arbitrário. Exemplo: para ilustrar o uso de OpenFlow, considere um programa controlador escrito em Python que implementa um *hub* repetidor simples. Suponha que a rede tem um único *switch* ligado a um conjunto de *hosts* internos no porto 1 e uma rede ampla no porto 2, como mostrado na figura 4.7. O manipulador *switch_join* abaixo invoca o

repetidor quando o *switch* se junta à rede. A função repetidor instala regras no *switch* que o instruem a enviar pacotes da porta 1 para a porta 2 e vice-versa. As regras são permanentes (prioridade DEFAULT e tempo limite *None*).

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

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

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

NOX obteve uma grande aceitação entre os pesquisadores da área de SDN. A existência de duas interfaces, C++ e Python, permite que o mesmo ambiente seja utilizado em situações que exigem alto desempenho e em casos onde a capacidade de expressão de Python agilizam o desenvolvimento e simplificam o código resultante. Como será discutido mais tarde, POX foi desenvolvido com base no modelo de NOX, mas com a premissa de ser completamente escrito em Python, resultando em uma interface mais elegante e simples dentro daquela linguagem. Os desenvolvedores de POX acreditam que este seja adequado para substituir NOX nos casos em que Python é utilizado, enquanto NOX ainda permanece como um ambiente adequado para implementações que tenham demandas mais elevadas em termos de desempenho.

4.5.2. Trema

Trema [tre 2012] é uma implementação OpenFlow para desenvolvimento de controladores usando as linguagens de programação Ruby e C. O escopo de Trema é ajudar os desenvolvedores a criar facilmente seus próprios controladores OpenFlow. Não tem como objetivo fornecer uma implementação específica de um controlador OpenFlow.

Controladores Trema OpenFlow são simples *scripts* escritos em Ruby. É possível escrever seu próprio controlador adicionando manipuladores de mensagem para sua classe *controller*. Com um arquivo de configuração é possível descrever a topologia da rede na qual o controlador é executado e passá-lo para execução pelo trema. Um exemplo de arquivo de configuração é apresentado a seguir.

```
class MyController < Controller
   # packet_in handler
   def packet_in dpid, msg
      send_flow_mod_add(
         dpid,
         :match => ExactMatch.from(msg),
         :buffer_id => msg.buffer_id,
         :actions => ActionOutput.new(msg.in_port+1)
      )
   end
end
```

Trema tem um emulador de rede OpenFlow integrado. Não é preciso preparar *switches* OpenFlow e *hosts* para testar aplicações de controlador. O código a seguir mostra um exemplo de arquivo de configuração de rede que cria dois *switches* virtuais.

```
# network.conf
vswitch { dpid ``0xabc'' }
vhost ``host1''
vhost ``host2''
link ``0xabc'', ``host1''
link ``0xabc'', ``host2''
```

A execução é feita com a seguinte linha de comando:

```
./trema run mycontroller.rb -c network.conf
```

4.5.3. Maestro

Maestro é uma plataforma escalável de controladores para *switches* OpenFlow em Java [Cai et al. 2010].

Maestro é um sistema operacional de rede desenvolvido para orquestrar aplicações de controle no modelo SDN. O sistema fornece interfaces para implementação modulares de aplicações de controladores de rede para acessar e modificar o estado da rede e coordenar suas interações. O modelo de programação do Maestro fornece interfaces para:

- Introdução de novas funções de controle personalizadas, adicionando componentes de controle modular.
- Manutenção do estado da rede em nome dos componentes de controle.
- Composição de componentes de controle, especificando a sequência de execução e o estado dos componentes da rede compartilhada.

Em particular, Maestro já inclui módulos de descoberta de recursos físicos da rede, utilizado para contruir sua visão global da estrutura da rede, e módulos para implementar protocolos de roteamento usuais sobre essa estrutura.

Além disso, Maestro tenta explorar o paralelismo dentro de uma única máquina para melhorar o desempenho do sistema de transferência. A característica fundamental de uma rede OpenFlow é que o controlador é responsável pela criação inicial de cada fluxo entrando em contato com switches relacionados. O desempenho do controlador pode ser o gargalo. No projeto do Maestro tentou-se exigir o menor esforço possível dos programadores para conseguir gerenciar a paralelização. Maestro lida com a maior parte do trabalho tedioso e complicado de gerir distribuição de carga de trabalho e agendamento de *threads*.

No Maestro, os programadores podem alterar a funcionalidade do plano de controle escrevendo programas *single-threaded*. Maestro incorpora um conjunto de modelos e técnicas que abordam a especificação de requisitos do OpenFlow e explora paralelismo em nome dos programadores. Para maximizar o rendimento do sistema, a carga

de trabalho tem de ser uniformemente distribuída entre as *threads* em Maestro, de modo que nenhum núcleo do processamento fique ocioso, enquanto há trabalho a fazer. Isto é alcançado fazendo com que todas as *threads* compartilhem a fila de tarefas de pacotes. O projeto

4.5.4. Beacon

Beacon é outro controlador baseado em Java que suporta tanto a operação baseada em eventos quanto em *threads* [bea 2012].

O projeto vem sendo desenvolvido já há um ano e meio, sendo considerado estável. Seu registro de operação inclui a gerência de uma rede com 100 *switches* virtuais, 20 *switches* físicos em um *datacenter* experimental. O sistema tem uma estrutura modular que permite que o controlador seja atualizadom em tempo de execução sem interromper outras atividades de encaminhamento de pacotes. O pacote opcionalmente incorpora o servidor web Jetty e um arcabouço extensível para o desenvolvimento de interfaces de usuário personalizadas.

4.5.5. FML

A linguagem FML (Flow Management Language) é uma linguagem declarativa de alto nível que permite especificar políticas de gerência e segurança em redes OpenFlow. FML foi desenvolvida para substituir os mecanismos tradicionais de configuração de redes para reforçar as políticas dentro das redes de empresas [Hinrichs et al. 2009]. A linguagem é simples e pode ser usada para expressar muitas configurações comuns usadas em redes atualmente. FML foi projetado para admitir uma implementação eficiente, adequada para grandes redes corporativas.

FML opera sobre fluxos unidirecionais. Isso significa que a política resultante á aplicada igualmente em todos os pacotes dentro do mesmo fluxo, e políticas podem ser construídas que tratam cada fluxo diferentemente.

O sistema resultante permite a especificação de alto nível de várias tarefas de gerenciamento de forma resumida e estruturada, liberando os administradores de rede do trabalho penoso de configurar regras de controle de acesso em roteadores, *firewalls*, servidores de NAT e VLANs para alcançar políticas de uso de rede simples. A natureza declarativa da FML permite que os administradores se concentrem sobre a política de decisões, em vez de detalhes de implementação.

Uma política FML é um conjunto de regras não-recursivas. Por exemplo, as três afirmações seguintes dizem que Todd e Michelle são superusuários e um superusuário não tem restrições de comunicação.

```
allow(Us;Hs;As;Ut;Ht;At;Prot;Req) <= superuser(Us)
superuser(Todd)
superuser(Michelle)
```

Os argumentos para *allow* são símbolos que correspondem a oito campos de um fluxo dentro do sistema. São eles: usuário, host e ponto de acesso fonte (U_s;H_s;A_s),

usuário, host e ponto de acesso alvo (U_t;H_t;A_t;), protocolo (Prot) e se o fluxo é uma requisição (Req).

Além de *allow* , há outras palavras chaves para o controle de acesso, como *deny, waypoint, avoid* e *ratelimit*.

Para qualidade de serviços, FML possui três palavras chaves *latency, jitter, band* que podem ser usadas para configurar diferentes requisitos para diferentes fluxos.

FML também permite a inclusão de referência a fontes externas, como consultas SQL em uma base de dados ou procedimentos remotos .

4.5.6. Frenetic

Frenetic é um sistema baseado em linguagem funcional desenvolvido para programar redes OpenFlow [Foster et al. 2010]. Frenetic permite que o operador da rede, ao invés de manualmente configurar cada equipamento da rede, programe a rede como um todo. Frenetic é implementado sobre NOX, em Python, e foi projetada para resolver problemas de programação com o OpenFlow/NOX. Frenetic introduz abstrações funcionais para permitir programas modulares e a composição desses programas.

Frenetic é composto de duas sublinguagens integradas: uma linguagem declarativa de consultas para classificar e agregar tráfego da rede e uma biblioteca reativa funcional para descrever políticas de envio de pacotes em alto nível.

$$
\begin{array}{lll}
\textit{Queries} & q ::= & \texttt{Select}(a)\ * \\
 & & \texttt{Where}(fp)\ * \\
 & & \texttt{GroupBy}([qh_1, \ldots, qh_n])\ * \\
 & & \texttt{SplitWhen}([qh_1, \ldots, qh_n])\ * \\
 & & \texttt{Every}(n)\ * \\
 & & \texttt{Limit}(n) \\
\textit{Aggregates} & a ::= & \texttt{packets} \mid \texttt{sizes} \mid \texttt{counts} \\
\textit{Headers} & qh ::= & \texttt{inport} \mid \texttt{srcmac} \mid \texttt{dstmac} \mid \texttt{ethtype} \mid \\
 & & \texttt{vlan} \mid \texttt{srcip} \mid \texttt{dstip} \mid \texttt{protocol} \mid \\
 & & \texttt{srcport} \mid \texttt{dstport} \mid \texttt{switch} \\
\textit{Patterns} & fp ::= & \texttt{true\_fp}() \mid qh\texttt{\_fp}(n) \mid \\
 & & \texttt{and\_fp}([fp_1, \ldots, fp_n]) \mid \\
 & & \texttt{or\_fp}([fp_1, \ldots, fp_n]) \mid \\
 & & \texttt{diff\_fp}(fp_1, fp_2) \mid \texttt{not\_fp}(fp)
\end{array}
$$

Figura 4.8. Sintaxe de Consulta do Frenetic

A figura 4.8 mostra a sintaxe das consultas em Frenetic. Cada cláusula é opcional, exceto *Select*, no qual identifica qual o tipo de evento retornado pela consulta: evento contendo pacotes, contador de bytes ou contador de pacotes. O operador $*$ é usado para combinar cláusulas.

A cláusula *Select(a)* agrega os resultados retornados pela consulta a. A cláusula *where(fp)* filtra os resultados, mantendo apenas aqueles pacotes que satisfazem o filtro de padrão fp. Os padrões de consultam podem usar os campo de cabeçalho

como porta (*inport*), endereço MAC de origem (*srcmac*), endereço MAC de destino (*dstmac*), entre outros. Filtro de padrões mais complicados podem ser construídos usando operadores de conjunto como interseção (*and_fp*), união (*or_fp*),diferença (*diff_fp*) e complemento(*not_fp*). Cláusulas *GroupBy([qh1,...,qhn])* subdividem o conjunto de pacotes consultados em subconjuntos baseado nos campos de cabeçalhos *qh*. Cláusulas *Every(n)* particionam os pacotes por tempo, agrupando pacotes que estejam na mesma janela de tempo. Finalmente, cláusulas *Limit(n)* limitam o número de pacotes em cada subconjunto. Um exemplo de consulta utilizando Frenetic é apresentado a seguir.

```
def web_query():
      return \
          (Select(sizes) *
           Where(inport_fp(2) & srcport_fp(80))) *
           Every(30))
```

A consulta seleciona os pacotes que chegam na porta física 2 e da porta de origem TCP 80. Ela soma o tamanho de todos esses pacotes a cada 30 segundos e retorna um evento com o resultado.

4.5.7. Onix

Onix é um controlador desenvolvido em parceria pela Nicira, Google e NEC, com o objetivo de permitir o controle de redes de grandes dimensões de forma confiável [Koponen et al. 2010]. Para garantir a escalabilidade e confiabilidade do sistema, Onix provê abstrações para particionar e distribuir o estado da rede em múltiplos controladores distribuídos, endereçando as questões de escalabilidade e tolerância a falha que surgem quando um controlador centralizado é utilizado.

A visão global da rede é mantida através de uma estrutura de dados denominada NIB (*Network Information Base*), que representa um grafo com todas as entidades presentes na rede física. A NIB é o centro do sistema e a base para o modelo de distribuição de Onix. Aplicações de controle da rede são implementadas através de operações de leitura e atualização do estado da NIB; escalabilidade resiliência são obtidos particionando-se e replicando-se a NIB entre os servidores do sistema. A consistência das atualizacões sobre a base distribuída é controlada pela aplicação e utiliza dois repositórios com compromissos diferentes entre os as garantias de resiliência e desempenho (uma base de dados transacional replicada e uma DHT baseada em memória). Para garantir a escalabilidade, Onix prevê a organização da NIB em estruturas hierárquicas, com concentradores e diferentes níveis de detalhes para os recursos do sistema.

Os resultados apresentados atestam o bom desempenho do sistema e o artigo discute em algum detalhe os compromissos adotados com relação à escalabilidade e resiliência do sistema. O sistema resultante, conforme descrito, sugere um grande cuidado com a funcionalidade e uma estrutura complexa que atende aos requisitos de projeto apresentados. Entretanto, Onix é, pelo menos até o momento, um produto fechado, proprietário.

4.5.8. Click

O roteador modular Click [Kohler et al. 2000] é um projeto que expressou o objetivo de permitir que equipamentos de rede fossem programáveis bem antes do advento de redes definidas por software. Sendo considerado um dos projetos que influenciaram a definição de OpenFlow. O roteador Click enfatiza a modularidade, permitindo que pesquisadores criem módulos de processamento de pacotes customizados. Entretanto, Click foca exclusivamente em *switches* de software (implementado como módulo do kernel do Linux). Já existe uma interface do OpenFlow para o Click, o elemento OpenFlow-Click [Mundada et al. 2009]. Esse elemento permite o contralador instalar regras para fazer pacotes atravessarem diferentes elementos. O OpenFlowClick permite um controlador central controlar vários roteadores Click ao mesmo tempo. Esta interface tem o potencial para novas possibilidades de aplicações de processamento de tráfego, como eliminar pacotes duplicados ou detecção de anomalias via inspeção periódica de pacotes.

4.5.9. Floodlight

Floodlight [flo 2012] é um controlador OpenFlow para redes corporativas baseado totalmente na linguagem Java e distribuído segundo a licença Apache. O projeto se originou do controlador Beacon e agora é apoiado por uma comunidade de desenvolvedores e também pela Big Switch Networks, *start-up* que produz controladores comerciais que suportam o Floodlight. O núcleo e módulos principais são escritos em Java. Recentemente adicionaram Jython, o que permite desenvolvimento na linguagem Python. Em sua arquitetura, todos os elementos são módulos e módulos exportam serviços. Toda a comunicação entre módulos é feita através de serviços. A interface ItopologyService permite descobrir a topologia da rede automaticamente. Permite integrar com redes não openflow e é compatível com a ferramenta de simulação Mininet [Lantz et al. 2010].

4.5.10. SNAC

Simple Network Access Control(SNAC) [sna 2012] é um controlador OpenFlow no qual se utiliza uma interface web para monitorar a rede. Ele incorpora uma linguagem de definição de políticas flexível que permite realizar buscas utilizando padrões de alto nível para especificar políticas de controle de acesso. SNAC também provê uma ferramenta de monitamento gráfico, mas não é um ambiente de programação genérica, como outros controladores discutidos aqui.

4.5.11. NEC Programmable Flow

Na parte de produtos comerciais, temos a família de produtos do NEC Programmable Flow [nec 2012]. A família provê tanto componentes de software como também de hardware. Em software, temos o ProgrammableFlow Management Console, que é uma ferramenta para monitoramento que provê um ponto de controle centralizado. Em hardware, temos o controlador ProgrammableFlowController que permite virtualização em nível de rede e é útil, por exemplo, para *data center* pois permite monitorar e controlar redes com vários níveis de gerência. Dentre os *switches*, podemos citar o PF5420 e o PF5820 que são compatíveis com OpenFlow e reduzem o custo de operação da rede já que não precisam executar algoritmos dsitribuídos tradicionais como arvoré geradora, já que funcionam com um controlador centralizado.

4.5.12. Mininet

Apesar de não se tratar de um controlador propriamente dito, na parte de emuladores e simuladores, é importante mencionar o Mininet [Lantz et al. 2010], uma ferramenta de simulação de Redes Definidas por Software. É um sistema que permite a rápida prototipação de grandes redes utilizando apenas um computador. Mininet cria Redes Definidas por Software escaláveis utilizando primitivas de virtualização do Sistema Operacional, como processos e namespaces de rede. Com essa primitivas, ele permite rapidamente criar, interagir e customizar protótipos de Redes Definidas por Software.

4.6. Aplicações

Pela flexibilidade que Redes Definidas por Software oferecem para a estruturação de sistemas em rede, o princípio estruturador oferecido por esse paradigma pode ser útil em praticamente todas as áreas de aplicação de Redes de Computadores. A estruturação da rede com comutadores programáveis e uma visão (lógica) centralizada da rede podem ser de grande valia no desenvolvimento de novas funcionalidades em um grande número de ambientes. Nesta seção, discutimos primeiramente os principais contextos já identificados que podem se beneficiar da aplicação de SDN. Em seguida, identificamos alguns dos principais projetos de pesquisa em andamento que se valem do paradigma para avançar a fronteira do conhecimento em suas áreas, no Brasil e no mundo.

4.6.1. Contextos de aplicação

Se considerarmos que SDN define uma nova forma de estrutura um sistema em rede, podemos considerar que ela seja aplicável a qualquer tipo de aplicação de Redes de Computadores que possa se beneficar de um maior grau de organização das funcionalidades oferecidas em torno de uma visão completa da rede. Entretanto, alguns domínios de aplicação já foram identificados e vêm sendo tratados por diferentes projetos de pesquisa.

Controle de acesso

Pela natureza da interface de programação de OpenFlow, onde o tratamento é definido pelos padrões que identificam fluxos, a sua aplicação ao problema de controle de acesso é bastante direta. A visão global da rede oferecida pelo controlador SDN permite que regras de acesso sejam desenvolvidas com base em informações abrangentes e não apenas o que seria possível com o uso de um *firewall* em um enlace específico da rede. De fato, como mencionado anteriormente, um dos projetos precursores de OpenFlow foi exatamente Ethane, um sistema de controle de acesso distribuído, baseado no conceito de uma visão global das políticas de controle e do estado da rede [Casado et al. 2009].

O poder de expressão disponível ao se adotar uma visão global permite a implementação de regras que levem em conta não apenas tipos de protocolo e pontos de origem/destino, mas a relação entre estes e a identidade do usuário de forma simples. Além disso, a facilidade de se definir qual rota adotar para cada fluxo viabiliza inclusive a utilização de filtros especiais de tráfego (*middle-boxes*), como dispositivos de inspeção de pacotes (DPI) e *proxies*. Um exemplo do que pode ser feito nesse caso é o sistema SpSb, um *honeypot* de alta interatividade que está sendo desenvolvido no DCC/UFMG para

o estudo do comportamento de *bots*, especialmente aqueles usados para a distribuição de *spam* [Silva et al. 2011]. Uma máquina é interposta entre o computador infectado e o restante da Internet com um comutador Open vSwitch e um controlador POX. Cada fluxo iniciado pela máquina infectada é inspecionado pelo controlador, que pode decidir entre bloqueá-lo, permitir sua passagem para o restante da rede ou redirecioná-lo para um conjunto se servidores desenvolvidos para emular serviços importantes que permitam ao sistema controlar o comportamento do *bot*. Por exemplo, um servidor de DNS especialmente configurado pode ser programado para registrar as consultas efetuadas e redirecionar acessos quando necessário. Um tipo de acesso que seria sempre redirecionado seria qualquer tentativa de contato a um servidor de correio eletrônico por SMTP, que seria enviado para um servidor especialmente configurado para emular qualquer servidor de correio e armazenar as mensagens recebidas, sem entretanto repassá-las para o restante da rede.

Gerência de Redes

Um dos argumentos sempre utilizados com relação a SDNs é que a visão global da rede simplifica as ações de configuração e gerência de rede, enquanto os contadores associados aos fluxos permitem uma monitoração clara dos fluxos de interesse. Entretanto, ainda não está claro como essas funcionalidades podem ser melhor integradas a práticas de gerência habituais. Uma forma de integração interessante é certamente a integração direta de *switches* OpenFlow a sistemas de gerência tradicionais, como os que utilizam o protocolos SNMP [Farias et al. 2011].

Outros projetos, como OMNI [Mattos et al. 2011], oferecem opções para simplificar o processo de administração de redes OpenFlow pela inclusão de uma interface de controle baseada em uma arquitetura orientada serviços. Além disso, OMNI inclui uma plataforma orientada a agentes para simplificar a expressão de comportamentos que devem ser monitorados e sobre os quais o sistema deve atuar.

Também utilizando uma arquitetura multi-agentes, pesquisadores da UFAM têm estudado a integração de técnicas de Inteligência Artificial a redes definidas por software. A visão global da rede provida pelos controladores oferece um ponto focal sobre o qual técnicas de detecção de anomalias ou padrões frequentes podem ser aplicadas para se detectar diversos tipos de comportamento, como ataque de negação de serviço [Braga et al. 2010].

Redes domiciliares

Redes domésticas têm sido apontadas como um dos grandes desafios para a indústria de redes, especialmente no que se refere a dotar o usuário dessas redes de recursos eficientes para sua gerência e configuração [Dixon et al. 2010]. Vencer esse desafio exigirá tanto o investimento em soluções para os problemas de interação dos usuários com os dispositivos de rede, quanto em novas soluções para automação de configuração e implementação de políticas de segurança para o lar.

Uma forma de aplicar os princípios de SDN nesse contexto é utilizar um roteador doméstico com recursos OpenFlow e transferir para o provedor de acesso a responsabilidade de controlar a rede através de um controlador SDN. Dessa forma, esse sistema operacional de rede seria capaz de programar cada roteador doméstico com as regras de acesso apropriadas para cada usuário e ter uma visão global de todo o tráfego que atravessa o enlace de acesso a cada residência, construindo dessa forma uma visão agregada do que pode ser considerado tráfego lícito e quais padrões de tráfego podem indicar a ação de atacantes ou código malicioso. Esse enfoque tem sido adotado por pesquisadores da Georgia Tech [Feamster 2010, Calvert et al. 2011].

Por outro lado, outra solução interessante pode ser a implementação de um controlador de rede interno à rede doméstica. Essa solução se torna atraente especialmente frente ao aumento da complexidade dessas redes, que contam hoje com um número crescente de dispositivos ativos, como *smartphones* e consoles de jogos, além dos computadores usuais, além de poderem contar hoje com mais de um elemento de comutação, tanto para enlaces cabeados quanto para enlaces sem fio. Um controlador doméstico poderia acompanhar todo o tráfego interno da rede, potencialmente tornando mais simples a detecção de máquinas comprometidas por *malware* (cujo tráfego de disseminação seria mais facilmente identificdo contra o padrão usual do tráfego do lar) e a configuração de serviços da rede, como impressoras e novos dispositivos de acesso.

Gerência de energia

O interesse em soluções que conservem energia em sistemas de computação vem crescendo. Em ambientes corporativos e grandes *datacenters*, onde os recursos de rede assumem dimensões significativas, reduzir o consumo de energia consumida pelo meio de comunicação se torna uma possibilidade interessante. Ações como a redução da taxa de transmissão de enlaces sub-utilizados ou seu desligamento completo (e de dispositivos de rede, da mesma forma) se tornam viáveis a partir do momento em que uma rede definida por software oferece uma visão global do estado da rede, que simplifica a identificação desses elementos ociosos e mesmo a redefinição de rotas a fim de desviar tráfego de elementos passíveis de desligamento. Além disso, o controle das rotas e decisões de encaminhamento permite a implantação de pontos de controle (*proxies*) que interceptem tráfego "ruidoso" na rede, evitando que pacotes de menor importância atinjam máquinas que podem operar em um modelo de *wake-on-lan*, evitando que as mesmas sejam ativadas desnecessariamente.

Comutador virtual distribuído

Ambientes virtualizados, comuns hoje em redes corporativas e grandes *datacenters*, usualmente implicam no uso de *switches* implementadas por software no hypervisor instalado em cada máquina física a fim de prover a conectividade exigida por cada máquina virtual (VM). Um exemplo de tal *switch* é o Open vSwitch, discutido anteriormente. A facilidade de se mover as máquinas virtuais entre os hospedeiros físicos, um recurso bastante valorizado nesses ambiente, torna o processo de configuração e monitoração dessas

máquinas um desafio que complica a gerência dos *switches* virtuais. Uma forma de reduzir essa complexidade seria oferecer a visão de um *switch* único, virtual, ao qual todas as VMs estariam conectadas. Dessa forma, migrações de VMs teriam impacto mínimo nas configurações da rede virtual, uma vez que elas continuariam ligadas a uma porta do único *switch* visível para a rede. Essa abstração pode ser facilmente implementada através de um controlador de rede, já que ele pode inserir automaticamente as regras de encaminhamento de tráfego entre as portas dos diversos *switches* de software para criar o efeito desejado. Com o uso de *switches* físicos que também implementem o protocolo OpenFlow, mesmo máquinas físicas não virtualizadas podem fazer parte dessa rede.

Roteador expansível de alta capacidade

Grandes *datacenters* e as bordas das redes de acesso de grandes provedores, bem como das suas redes de longa distância, possuem demandas de conectividade elevadas. Nesses casos, usualmente um roteador de grande porte é usualmente a única solução viável, a um custo bastante elevado. Utilizando-se uma rede definida por software, pode ser possível substituir esse roteador de grande porte por um banco de *switches* de porte médio, como as utilizadas internamente em um *cluster*, desde que dotadas de interfaces OpenFlow. De 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 fazendo parte do mesmo roteador. Uma malha de interconexão do tipo *fat-tree* garantiria uma banda de interconexão suficiente entre os comutadores, de forma que o controlador não teria que lidar com gargalos inesperados entre portas do roteador. As informações de roteamento seriam distribuídas entre os *switches* em função das demandas identificadas pelo controlador. Os algoritmos de roteamento necessários ao funcionamento do roteador poderiam ser executados também pelo controlador, que se encarregaria de manter a interface com os demais roteadores com os quais o roteador virtual tenha que se comunicar.

Datacenters multi-usuários

Em ambientes onde aplicaçõs de vários usuários necessitem coexistir, como em *datacenters* públicos (*multi-tenant*), uma demanda sempre presente é o isolamento de tráfego entre os diversos usuários. Uma forma de se obter esse isolamento com dispositivos atuais é através da configuração de VLANs individuais para cada cliente. Assim, tráfego de cada usuário é transportado apenas dentro da sua VLAN, garantindo o isolamento necessário para fins de segurança e privacidade. Entretanto, o recurso de VLANs é limitado pelo tamanho dos campos usados para identificá-las, tornando inviável sua utilização quando o número de usuários da rede cresce. Além disso, as interfaces de configuração desse tipo de solução tendem a ser limitadas, tornando a gerência de configuração desse tipo de rede uma tarefa bastante complexa. A solução para esse problema pode ser construída diretamente sobre a abstração do comutador virtual distribuído mencionado anteriormente: a cada usuário é fornecido um *switch* virtual que interliga suas máquinas, independente da localização das mesmas na rede física. O controlador de rede é responsável por definir as rotas entre as portas dos diversos comutadores que compõem o comutador virtual e por

garantir a alocação de recursos suficientes em cada caso. Como cada cliente é conectado a um *switch* diferente, o isolamento de tráfego é obtido diretamente pela implementação.

Além de garantir o isolamento de tráfego de forma eficaz, essa solução tem a vantagem de permitir que se ofereça a cada usuário o controle direto de sua rede. Quaisquer configurações que estejam disponíveis para o *switch* virtual podem ser controladas pelo usuário que controla a rede, uma vez que a mesma está isolada das demais. Soluções como balanceamento de carga, estabelecimento de níveis de qualidade de serviço e redundância de rotas podem ser colocadas sob o controle do usuário, desde que o controlador de rede garanta sua implementação no nível do *switch* virtual distribuído.

4.6.2. Alguns projetos de pesquisa

Pela característica recente do tema, diversos projetos de pesquisa ainda focam o desenvolvimento de controladores de rede que preencham as características de um sistema operacional de rede, na definição de SDN. Esses projetos já foram discutidos na seção 4.5. Aqui discutimos projetos principalmente focados em aplicações do paradigma em alguns dos contextos mencionados anteriormente. Diversos são os que abordam problemas que afetam um ou mais domínios mencionados anteriormente. A lista a seguir não tem a intenção de ser exaustiva, dado o número de iniciativas, mas pretende ilustrar alguns desses projetos.

Ripcord

O *datacenter* foi identificado rapidamente como um dos principais contextos onde SDN poderia ser aplicado com sucesso. Isso pode ser observado na identificação de aplicações como o comutador virtual distribuído e o ambiente multi-usuário. Um ponto importante dos *datacenters* hoje é a demanda por novas topologias de rede que permitam a construção de infraestruturas de comunicação eficiente e robustas. Nesse contexto, Ripcord foi concebido como um sistema que aplicaria os conceitos de SDN na construção de um sistema de rede que pudesse ser configurado para diferentes topologias e diferentes soluções de roteamento, a fim de permitir a escolha da solução mais adequada para diferentes condições de contorno [Heller et al. 2010a, Casado et al. 2010a].

Além das características de modularidade, escalabilidade e operação com múltiplos usuários, Ripcord também tinha como metas definir um mecanismo de migração de VMs de forma transparente e com recursos de balanceamento de carga para o tráfego de rede. O protótipo desenvolvido foi implementado sobre o NOX, com diferentes módulos para implementar cada função identificada como essencial à sua operação. Entre os módulos (*engines*) principais, destaca-se o de topologia, que mantém a visão da rede física, o de aplicação, que controla a visão de cada usuário (*tenant*) sobre a rede, e o de roteamento, que implementa o mecanismo de roteamento com balanceamento de carga escolhida. O sistema foi testado com três políticas de roteamento com diferentes graus de complexidade e apresentou bom desempenho.

RouteFlow

No contexto de redes de longa distância, a implementação de protocolos de roteamento é uma preocupação frequente. Uma rede definida por software que pretenda operar de maneira eficiente com outras redes a seu redor deve implementar políticas de roteamento bem definidas e, mais que isso, deve ser capaz de trocar informações sobre suas rotas com elementos vizinhos. RouteFlow oferece uma solução distribuída para que se possa aplicar protocolos de roteamento bem definidos sobre uma rede definida por software, permitindo a implantação imediata de protocolos de roteamento já definidos e oferecendo um arcabouço extensível sobre o qual novos protocolos podem ser implementados [Nascimento et al. 2011].

RouteFlow executa como uma aplicação sobre NOX que controla a comunicação com os comutadores OpenFlow da rede e o servidor RouteFlow, que mantém a visão das rotas na rede. Os protocolos de roteamento propriamente ditos são executados por máquinas virtuais que executam versões do(s) protocolo(s) de roteamento escolhido(s) em um ambiente virtualizado. Essas versões dos protocolos são obtidos normalmente do Quagga *routing engine*. Cada roteador definido na visão de rede definida executa como uma máquina virtual, trocando mensagens dentro do ambiente virtualizado com os demais roteadores da rede, de acordo com o protocolo escolhido. Se a especificação da rede prevê a existência de roteadores adicionais que não fazem parte da rede definida por software, as mensagens dos protocolos de roteamento que devem ser trocadas entre os roteadores virtualizados e esse roteadores reais são injetadas e retiradas da rede através dos comutadores OpenFlow.

Elastic Tree

Com foco na questão de gerência (e economia) de energia, o objetivo de Elastic Tree é controlar a topologia da rede física de um *datacenter* para reduzir o consumo devido a canais sub-utilizados [Heller et al. 2010b]. O princípio de operação nesse caso é que, em situações de carga reduzida, o nível de replicação de recursos encontrado em uma rede de *datacenter* bem provisionada pode levar a um consumo de energia desnecessário. Em uma topologia *fat-tree*, por exemplo, há diversos níveis de *switches* interligando os servidores da rede; o número de enlaces e *switches* a cada nível é definido de forma que haja diversas rotas entre cada par de servidores, a fim de evitar gargalos na rede. Em condições de carga baixa, entretanto, muitos desses caminhos redundantes se tornam desnecessários e poderiam ser removidos para reduzir o consumo de energia.

Elastic Tree monitora continuamente o tráfego da rede e decide, com base nos objetivos de economia de energia e disponibilidade expressos pelo administrador, quais enlaces e switches precisam permanecer ativos e quais podem ser desligados. A visão global do tráfego e da topologia são providos pelo controlador SDN adotado (NOX, no protótipo implementado). Diversos algoritmos de otimização foram implementados para oferecer diferents políticas de controle, refletindo diferentes compromissos entre economia e disponibilidade. Em casos onde o padrão de tráfego se limitava em sua maioria a servidores próximos na topologia, a energia consumida chegava a ser apenas 40% da

energia consumida com a topologia original, sem controle. Os ganhos se reduzem caso o padrão de tráfego exija a comunicação entre servidores distantes na topologia (exigindo maior conectividade nos níveis superiores da hiearquia de *switches*), mas são ainda significativos para uma ampla gama de valores.

Gatekeeper-ng

Ainda no contexto de redes para *datacenters*, um outro aspecto que tem recebido grande atenção é a questão de se estabelecer garantias de alocação de tráfego para diferents usuários com suas aplicações. Gatekeeper é uma solução proposta para esse fim que se baseia em uma visão abstrata onde cada usuário receberia garantias equivalentes às providas caso sua aplicação executasse em um conjunto de máquinas conectadas por um *switch* privativo, sem compartilhamento com outras aplicações. As garantias de desempenho seriam determinadas pela banda alocada para o enlace de conexão de cada máquina, que poderia ser escolhida pelo usuário em um modelo com custos associados [Rodrigues et al. 2011b, Rodrigues et al. 2011a].

A abstração oferecida por Gatekeeper se encaixa precisamente ao modelo de comutador virtual distribuído discutido anteriormente. Considerando-se que o ambiente alvo do sistema são *datacenters* virtualizados, onde a conexão de cada máquina virtual à rede se dá através de *switches* de software (Open vSwitch), a extensão do modelo para utilizar uma rede definida por software que realmente implemente a abstração de um *switch* virtual distribuído pode ser feita de forma simples [Pettit et al. 2010]. Além disso, ao se integrar esse modelo ao ambiente Open Stack, que também se baseia na noção de uma rede privada assinalada para cada usuário [ope 2012], o processo de configuração, disparo de máquinas virtuais e reconfiguração no caso de migração de VMs pode ser totalmente integrado, dando origem ao ambiente Gatekeeper-ng, atualmente em desenvolvimento.

POX at Home

Esse projeto aborda o desenvolvimento de uma aplicação SDN para gerência de redes domésticas. Inicialmente denominado *NOX at home*, foi recentemente adaptado para utilizar o controlador POX. O princípio por trás do projeto é que, dadas as velocidades de acesso disponíveis para usuários residenciais, a capacidade de processamento disponível em roteadores domésticos é suficiente para inspecionar cada fluxo que atravessa o enlace de acesso.

A aplicação POX nesse caso implementa um analisador de protocolos que acompanha cada novo fluxo observado na rede local e reconstrói o fluxo de aplicação para protocolos identificados como relevantes, como SMTP, HTTP e DNS. Essa reconstrução continua até que o controlador tenha informação suficiente para se avaliar a conformidade do tráfego em relação a um conjunto de regras de acesso e aplicar um algoritmo de detecção de anomalias para determinar a natureza do tráfego [Mehdi et al. 2011]. Nesse momento, uma regra é emitida para incluir uma rota direta para o restante do fluxo, ou para descartá-lo, caso seja julgado inadequado. Há também a possibilidade de se configurar a travessia de um *middle-box* caso algum tipo de pós-processamento do fluxo seja

necessário

4.7. Desenvolvimento de sistemas utilizando POX

POX vem sendo desenvolvido como um sucessor natural de NOX para iniciativas de pesquisa e ensido. O objetivo dos desenvolvedores é que ele venha a substituir NOX nos casos onde desempenho não seja um requisito crítico — essencialmente, nos casos onde a interface Python é utilizada. Nesse aspecto, POX traz uma interface mais moderna e uma pilha SDN mais elegante, além de oferecer melhor desempenho que NOX com Python, como pode ser visto na figura 4.9.

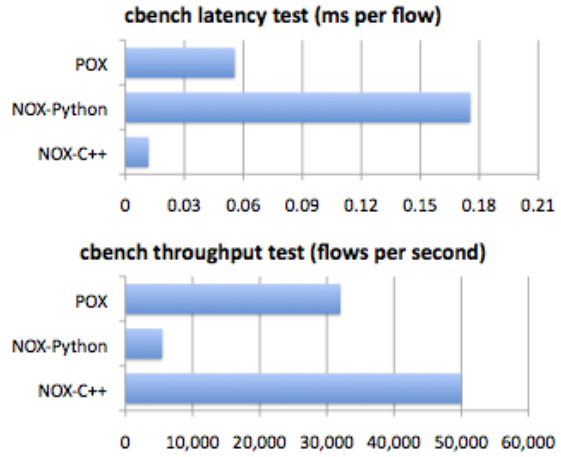

Figura 4.9. Comparação de desempenho entre POX e NOX, com suas duas interfaces (C++ e Python). Figura publicada por Murphy McCauley no site `http://www.noxrepo.org/2012/03/introducing-pox/`.

NOX e outros controladores de primeira geração são construídos ao redor do conceito de mensagens OpenFlow como primitivas básicas. Assim sendo, as aplicações ser organizam ao redor de ciclos de recebimento/tratamento/geração de mensagens. POX está sendo organizado ao redor da noção de visão global da rede: aplicações não necessitarão necessariamente receber e enviar mensagens diretamente, mas poderão consultar a visão corrente da rede e atuar diretamente sobre ela para obter seus objetivos (os recursos para esse fim ainda estão em desenvolvimento). Outros elementos que fazem parte do contexto de projeto de POX incluem a previsão de recursos para a distribuição dessa visão global e para a depuração de aplicações desenvolvidas sobre essa visão.

4.7.1. Instalação e execução

POX[1] exige Python 2.7 (ou superior) para executar. Ele deve ser compatível com Python 3.x, mas nenhuma verificação cuidadosa foi feita a esse respeito. Não há registro de elementos que sejam específicos de uma plataforma particular e ele pode ser executado no Linux, Mac OS ou Windows. Para um melhor desempenho, recomenda-se o uso da implementação no ambiente PyPy [2].

[1] O conteúdo dessa seção é diretamente derivado de notas de utilização disponíveis no site `http://www.noxathome.org/wiki/POXConcepts`.

[2] `http://pypy.org/`

Obtenção do código

Originalmente, POX era distribuído o controle de versão Mercurial (hg), mas no momento da escrita deste material ele estava sendo movido para o controlador Git e deve ser em breve disponibilizado no GitHub [3].

Início da execução e o mecanismo `launch()`

POX é iniciado ao se executar o módulo `pox.py`. Ele recebe como argumentos a lista de nomes de módulos que devem ser instanciados durante a execução. Cada módulo identificado é localizado e a função `lauch()` de cada módulo é executada, se existir. Em seguida, POX abra a interface de comandos de Python para uso interativo. Módulos são procurados na hierarquia de diretórios usual de Python e nos diretórios `pox` e `ext` do pacote. Assim sendo, para se disparar o módulo que implementa *switches* Ethernet com aprendizado, utiliza-se o comando
`./pox.py dumb_l2_switch`

Cada módulo pode receber argmentos através de opções de execução indicadas após o nome do módulo. Essas opções são passadas como parâmetros para a função `launch()` do módulo.

Em particular, as opções de configuração de OpenFlow são úteis ao se iniciar a execução de POX associada a um ambiente de rede criado com o arcabouço Mininet. Como mencionado na seção 4.5.12, Mininet cria uma rede virtual de comutadores OpenFlow executando em uma máquina virtual disparada na máquina local. Os comutadores e *hosts* na rede virtual podem ser configurados para se comunicar com uma instância de POX executando no computador hospedeiro. Para isso, POX deve ser iniciado da seguinte forma:
`./pox.py openflow.of_01 --address=10.1.1.1 --port=6634`
Em seguida, no terminal Mininet, dispara-se a rede virtual com o comando:
`sudo mn --controller=remote --ip=10.1.1.1 --port=6634`

Para que isso seja possível, a classe `of_01` foi definida com o método `launch()` na forma:

```
    def launch (port=6633, address = ''0.0.0.0''):
        '''Operações a serem executadas ao se iniciar o
módulo'''
```

O mesmo recurso pode se utilizado por qualquer módulo desenvolvido no sistema.

Além disso, `pox.py` reconhece algumas opções próprias, que devem ser listadas primeiro:

```
    --verbose exibe stack traces para exceções durante a
inicialização
    --no-cli não inicia a interface interativa de comandos
    --no-openflow não inicia o módulo openflow automaticamente
```

[3] `https://github.com/`

Testes e depuração

No momento, não há ainda uma rotina de testes completamente definida, mas há um primeiro esboço de como se construir um arcabouço para testes de unidade e integração no sub-diretório `test/`.

Um dos objetivos maiores de POX é dotar o ambiente SDN de mecanismos de depuração sofisticados que integrem as diversas camadas de uma rede definida por software. Em sua versão atual, entretanto, há apenas um sistema de depuração que aumenta os níveis de avisos e de geração de *logs*, habilita a detecção de *deadlocks* e outras medidas úteis. Esse sistema é iniciado ao se executar `./pox.debu.py` ao invés do comando original. Além disso, pode ser útil se incluir nos módulos desenvolvidos recursos de execução passo-a-passo, o que pode ser feito ao se executar a linha `import pdb; pdb.set_trace()`, que iniciará a execução passo-a-passo.

Registro no `pox.core`

Ao executar, o objeto `pox.core` é criado e oferece a funcionalidade de se registrar objetos arbitrários, que passam a ser acessíveis para os módulos do sistema. Por exemplo, você pode escrever o código de um tipo de comutador de rede, que pode então ser registrado como `pox.core.switch` em tempo de execução. Uma vantagem disso é que outros podem posteriormente implementar um comutador totalmente diferente que implemente a mesma interface de programação e registrá-lo como `pox.core.switch`, por sua vez. Outras aplicações que utilizem a interface do comutador podem continuar a funcionar, sem modificações. Uma vantagem desse mecanismo é que ele depende menos do uso de comandos `import` do Python, reduzindo o código de inicialização. Como exemplo, o código a seguir, do módulo `of_01`, cria um objeto responsável pelas tarefas de controle do protocolo OpenFlow e registra-o para que o mesmo seja facilmente referenciado por outros objetos:

```
def launch (port = 6633, address = ''0.0.0.0''):
  if core.hasComponent('of_01'):
      return None
  l = OpenFlow_01_Task(port = int(port), address = address)
  core.register(''of_01'', l)
  return l
```

Orientação a eventos

POX usa eventos frequentemente. O sistema de controle de eventos é implementado na biblioteca `revent`. O princípio geral da implementação é que objetos se tornam eventos, não apenas uma coleção de campos e métodos. Um objeto pode disparar eventos e outros podem esperar por eventos gerados por aquele objeto registrando um tratador (*handler*) com ele. Tratadores de eventos executam de forma não preemptiva, até o seu fim.

Um objeto se torna uma fonte potencial de eventos ao herdar de `EventMixin` (apesar dessa classe ter sido projetada com o objetivo de ser adequada para a combinação

em tempo de execução de classes e objetos, não há suporte a isso no momento e a única forma de uso apoiada no momento é herdando-se dela). Tal objeto normalmente especificará quais tipos de eventos ele pode disparar, apesar disso não ser obrigatório. Os eventos propriamente ditos são normalmente classes que estendem `Event` e passam a possuir certas características, mas na prática podem ser qualquer coisa. Qualquer objeto interessado em um evento pode registrar um tratador para ele.

É possível também registrar o interesse em todos os eventos disparados por um objeto de forma implícita com métodos como `EventMixin.listenTo()`. Nesse caso, se `Foo` e `Bar` herdam ambos de `EventMixin` e `Bar` dispara eventos `timeout`, `newMsg` e `linkFailure` e `Foo` define métodos `handle_timeout`, `handle_newMsg` e `handle_linkFailure`, `Foo` pode associar todos esses tratadores diretamente aos eventos apropriados de `Bar` simplesmente executando o método `self.listenTo(someBarObject)`. Essa forma é usualmente mais simples que registrar cada tratador para um número de eventos.

4.7.2. Modelo de *threads*

Na maioria das vezes o modelo de eventos não levanta a necessidade da preocupação do uso de *threads*, porém, caso um componente precise utilizar alguma forma de concorrência pode ser interessante entender o modelo de *threads* usado no POX.

O modelo de *threads* do POX é programado como a biblioteca de nome `recoco` para poder criar um ambiente de *threads* cooperativas, que em um modelo no qual as próprias *threads* delegam o processamento entre si, ao contrário do modelo preemptivo no qual o sistema operacional ativamente interrompe o processamento e o delega a outra *thread*.

As *threads* cooperativas precisam ser criadas criando uma classe que estenda `recoco.Task` e implemente um método `run`. Após a criação a *thread* precisa ser escalonada e quando for escolhida para executar, ela pode considerar que executará atomicamente até terminar ou até entregar o processamento (através de funções como o `Sleep`, por exemplo).

Portanto é importante ficar atento que qualquer operação blocante irá bloquear o funcionamento do sistema por completo.

Nada impede que sejam escritos componentes que usam o modelo preemptivo padrão do python, porém é importante atentar que o modelo preemptivo irá funcionar em paralelo com o modelo cooperativo, o que pode causar problemas no desenho de alguns componentes e dados inconsistentes.

Alguns dos problemas causados por esse tipo de interação podem ser corrigidos com as seguintes ações:

1. Se somente é desejado que uma *thread* normal se comunique com uma task cooperativa do recoco, isso pode ser feito utilizando *locks* normais e estruturas *threadsafe* como o `dequeue` do Python. (com cuidado para que os *locks* inseridos não travem o processamento das outras tasks por muito tempo)

2. Se uma *thread* só precisa se comunicar com eventos usando a biblioteca

revent, ela pode usar a função POX.deferedRaise() no lugar de self.raiseEvent(). A primeira função irá levantar um evento de dentro de uma task, fazendo isso de modo seguro.

3. Uma *thread* normal pode executar código dentro de um bloco com recoco.scheduler.synchronized():. Ao executar dentro de um bloco com essa marcação, a *thread* nativa irá esperar por uma oportunidade de parar as *threads* cooperativas. E assim, executará o código dentro do bloco antes de permitir que as outras *threads* cooperativas executem novamente.

Outra funcionalidade que a biblioteca recoco implementa é um Timer simples para funcionar com esse modelo de *threads* cooperativas. Ele funciona de forma similar ao Timer comum do Python, como no exemplo a seguir:

```
from pox.lib.recoco import Timer
self._timer = Timer(30.0, self._timerHandler, recurring=True)
```

Esse trecho de código chama o timer a cada 30 segundos para executar a função _timerHandler definida no próprio módulo de forma recorrente. Os demais parâmetros definidos para o construtor da classe Timer são os seguintes:

```
timeToWake     tempo a esperar antes de chamar o callback (segundos)
callback       funcao a ser chamada quando o timer expirar
absoluteTime   uma hora especifica para executar (usa time.time())
recurring      chama o timer repetidamente ou somente uma vez
args, kw       argumentos para a funcao de callback
scheduler      escalonador recoco a usar (None escolhe o default)
started        se False, precisa chamar .start() para iniciar o timer
selfStoppable  se True, callback pode retornar False para parar o timer
def __init__ (self, timeToWake, callback, absoluteTime = False,
              recurring = False, args = (), kw = {}, scheduler = None,
              started = True, selfStoppable = True):
```

4.7.3. Organização do código

Em sua versão atual, o código fonte do POX está dividido em um conjunto de diretórios que compreende tanto módulos essenciais para seu funcionamento básico quanto componentes adicionais que oferecem funcionalidades adicionais úteis.

4.7.4. Componentes

Componente no POX é o nome dado às aplicações que são programadas para executar sobre ele, utilizando a sua interface para realizar uma determinada tarefa sobre a rede controlada. Essas aplicações podem ser incluídas pelo desenvolvedor para aproveitar as funcionalidades já implementadas.

Os seguintes diretórios contêm um ou mais componentes com suas determinadas funções:

- **gui_backend:** Tem a finalidade de oferecer informações para o plugin de interface gráfica do POX. Ele monitora as informações que passam pelo controlador para reunir dados necessários para a interface gráfica.
- **host_tracker:** Mantém informações sobre as máquinas conectadas à rede, em termos de em qual porta de qual *switch* OpenFlow elas estão conectadas e como estão configuradas (endereço MAC e IP, quando dispoonível). Essa informação pode ser usada, por exemplo, para implementar um serviço de ARP diretamente no núcleo da rede. Futuramente essa informação pode ser integrada com os componentes de topologia.
- **messenger:** Este componente possibilita a comunicação entre o POX e aplicações externas através de mensagens codificadas em JSON. Dessa forma é possível reunir informações no controlador e se comunicar com o mundo exterior de diversas formas independentes do protocolo de comunicação.
- **misc:** Componentes diversos. No momento possui dois componentes, `dnsspy` e `recocospy`, que apresentam apenas informações para depuração do protocolo DNS e da biblioteca de *threads* adotada.
- **samples:** Pacote de componentes simples que servem como exemplo de programação do POX. No momento possui 4 exemplos simples de diferentes tipos de componentes que utilizam diferentes partes do sistema a fim de servir como base para a construção de componentes mais complexos.
- **topology:** Reune informações da topologia da rede populada por demais componentes.
- **web:** Pacote de componentes que permitem a criação de um servidor web para interagir com o mundo exterior. No momento possui um componente de *framework* web e uma interface para exibir mensagens do componente `messenger` em um servidor web.
- **forwarding:** Conjunto de componentes que trata do encaminhamento de pacotes. Possui diversas implementações que simulam *switches* (níveis 2 e 3) com comportamento diferenciado para o ambiente openflow.

4.7.5. Módulos de funcionamento

Os módulos de funcionamento incluem o núcleo do POX e bibliotecas auxiliares.

- **log:** Controla o nível e a formatação dos logs do POX.
- **lib:** Bibliotecas auxiliares, como implementações de políticas de *threads*, manipulação de endereços, eventos, pacotes e outras bibliotecas que são utilizadas pelos componentes.
- **openflow:** Implementação do protocolo Openflow. Possui não só uma interface para a criação e tratamento de pacotes OpenFlow como alguns componentes de exemplo que utilizam essa interface.

- **core.py:** Núcleo do POX, possui funções base para criação de componentes, como funções de registrar componentes, eventos, etc.
- **__init__.py:** Construtor do POX.
- **license.py:** Listagem da licença GPL.

De interesse particular, são o módulo `openflow`, o módulo `packet`, e a biblioteca `revent`, que é a responsável por gerar os eventos do POX, já discutidos anteriormente.

Biblioteca `openflow`

A biblioteca `openflow` implementa facilidades para a comunicação com comutadores OpenFlow e para a recepção e geração de pacotes do protocolo OpenFlow. Cada vez que um comutador OpenFlow se torna ativo no sistema, ele estabelece uma conexão com o controlador. Essa conexão é representada por um objeto no sistema que é também um evento e pode ser observado por elementos que se interessem pelo surgimento de novos comutadores, como o módulo que mantém a visão da topologia da rede.

Por exemplo, pacotes oriundos dos comutadores OpenFlow são recebidos pelo módulo openflow e geram um evento `PacketIn`, que pode ser observado por qualquer objeto que esteja interessado na chegada de um pacote. Esse evento pode ser observado ao se registrar um tratador com o controlador propriamente dito, ou pode ser observado diretamente no objeto representando a conexão de um comutador em particular.

Para criar uma ação de `output`, existe a função `ofp_action_output` que ao receber uma porta do switch já cria uma estrutura de dados de ação para ser concatenada a uma mensagem criada com a função `ofp_packet_out` através da função `append` presente no objeto `actions` e depois ser enviada para o switch. Todo esse processo pode ser ilustrado pelos seguintes comandos:

```
msg = of.ofp_packet_out()
msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
self.connection.send(msg)
```

Biblioteca `packet`

A biblioteca `packet` possibilita que se acesse detalhes do pacote sendo manipulado. Por exemplo, um pacote recebido por um evento `PacketIn`, após ser traduzido com a função `parse` possui os campos `src` e `dst` que correspondem aos endereços Ethernet do pacote identificado. Também é interessante considerar o campo `next` que possui informações sobre o próximo cabeçalho do pacote, como qual protocolo ele segue (IP, ARP, etc). Por exemplo, um pacote IP possui informações do IP de origem e destino através dos campos `next.protosrc` e `next.protodst`. A utilização desses campos ficará mais clara no exemplo de componente que será comentado posteriormente.

Mais detalhes sobre as funções presentes duas bibliotecas podem ser encontrados nos links `http://www.noxathome.org/doc/pox/pox.openflow.libopenflow_01.html` e `http://www.noxathome.org/doc/pox/pox.lib.revent.revent.html`.

4.7.6. Exemplo de programação de um componente POX

Um componente no POX é uma implementação de uma aplicação que usa o Framework para escutar e levantar eventos. No POX é bem simples criar e levantar um componente. Primeiro será tratada a parte de criação de um componente e depois como executá-lo e testá-lo.

4.7.7. Criação de um componente mínimo

O POX facilita muito a criação de componentes. Primeiro é necessário criar um diretório dentro do diretório `pox` ou `ext`, que são os diretórios comuns para a pesquisa de componentes. Por exemplo, será criado o diretório `ext/foo`.

Dentro deste diretório será criado o construtor `__init__.py` e o módulo do componente `foo.py`. O construtor é necessário somente se o componente for chamado pelo nome do diretório, caso contrário o componente pode ser chamado pelo nome do diretório seguido por um ponto seguido pelo nome do arquivo (por exemplo, foo.foo). Caso seja interessante chamar o componente pelo nome do diretório, o método `launch` deve estar listado no construtor, caso contrário no arquivo que será chamado (no nosso exemplo, o foo.py).

O método `launch` é executado assim que o POX chama o componente. Ele tem como finalidade registrar um módulo como componente no núcleo do POX. Em nosso exemplo o método `launch` localizado no construtor que chama a classe classe `Foo` no arquivo `foo.py` seria o seguinte:

```
def launch (number, food = "spam"):
  from foo import Foo
  from pox.core import core
  core.registerNew(foo.Foo)
```

Note que dois argumentos são necessários, `number` e `food`, sendo que `food` é automaticamente colocado com o valor “spam” caso nenhum argumento seja passado. A chamada do POX para esse novo componente criado com os parâmetros utilizados deve ser da seguinte forma:

```
./pox.py foo --food=eggs --number=42
```

No arquivo `foo.py` é necessário que a classe `Foo` herde a classe `EventMixIn` para que seja possibilitada de levantar eventos e que utilize a função herdada `listenTo` para poder escutar os eventos levantados. Por exemplo:

```
class Foo (EventMixin):
  def __init__ (self):
    self.listenTo(core.openflow)
```

Agora o componente Foo já escuta a eventos, mas também é necessário criar funções para tratar certos eventos. Para tratar o evento `ConnectionUp`, levantado toda vez que o POX detecta uma nova conexão, é necessário que a classe Foo possua um método `_handle_ConnectionUp` (em outros eventos bastaria mudar o nome de ConnectionUp para o evento desejado). Esse método recebe como parâmetro o próprio evento e pode realizar qualquer computação com ele.

```
  def _handle_ConnectionUp (self, event):
    log.debug("Connection %s" % (event.connection,))
    ...
```

4.7.8. Um componente mais realista: Switch L2

Para exemplificar melhor o funcionamento de um componente, vamos usar o exemplo de um componente que simula um *switch* L2 com aprendizado a partir de um comutador OpenFlow. O comportamento desse *switch* é simplificado e segue o padrão a seguir:

```
Para cada novo fluxo PacketIn:
1) Use endereço/porta de origem p/ atualizar tabela endereço/porta
2) O tipo do pacote Ethernet é LLDP?
   Sim:
      2a) Descarte o pacote -- LLDP nunca é encaminhado
          FIM
3) O destino é multicast?
      3a) Envie o pacote por todas as portas
          FIM
4) O endereço de destino NÃO está na tabela endereço/porta?
      4a) Envie o pacote por todas as portes
          FIM
5) A porta de saída é a mesma de entrada?
      5a) Descarte o pacote e seus similares por um tempo
6) Instale uma entrada na tabela de fluxos do switch
    para que este fluxo siga para a porta apropriada
   6a) Envia os dados do pacote para a porta apropriada
FIM
```

Os passos do processamento descrito devem ser facilmente entendidos em face do comportamento usual de *switches* Ethernet com aprendizado. O passo 2, entretanto, trata um aspecto particular de funcionamento de redes OpenFlow. Cada comutador OpenFlow pode executar o protocolo de descoberta de enlaces, LLDP. Esse protocolo permite, por exemplo, que cada comutador identifique seus vizinhos; dessa forma, ele poderia ser utilizado para a construção de um mecanismo de composição da visão global da rede mencionada anteriormente. Neste exemplo, esses pacotes são apenas descartados.

O código completo é apresentado nas figuras 4.10 e 4.11. Elementos importantes do código são discutidos a seguir. Comando de *log* são utilizados de forma simplificada para reduzir o tamanho do código. Posteriormente discutimos o uso de mensagens de *log* mais elaboradas usando recursos de Python.

```
1  class l2_factory (EventMixin):
2    """
3    Espera por conexoes de switches OpenFlow e cria o objeto para
4    transformá-los em switches L2 com aprendizado.
5    """
6    def __init__ (self):
7      self.listenTo(core.openflow)
8
9    def _handle_ConnectionUp (self, event):
10     log.debug("Connection %s" % (event.connection,))
11     LearningSwitch(event.connection)
12
13 def launch ():
14   core.registerNew(l2_factory)
```

Figura 4.10. Código geral para registro de *switches* L2 com aprendizado no POX

O código da figura 4.11 define a classe `LearningSwitch`, que implementa o *switch* propriamente dito. Antes de analisarmos esse código, entretanto, devemos observar que o código da figura 4.10 define outra classe derivada de `EventMixin`, `l2_factory`. Essa última classe age como uma fábrica de *switches*: seu construtor (linhas 6–7) se registra com o componente `openflow` do *core*, de forma que o tratador de eventos para novas conexões (novos comutadores) é instalado (linhas 9–11). Ao detectar uma conexão de um novo comutador um objeto da classe `LearningSwitch` é criado e associado à conexão. As linhas 13–14 definem a função `launch()` para esse módulo: ela registra um objeto da classe construtora junto ao *core* do POX.

Já na figura 4.11, encontramos a definição do comportamento dos *switches* com aprendizado, como discutido anteriormente. A classe `LearningSwitch` também herda de `EventMixin` para ser capaz de tratar eventos de chegada de pacotes. Objetos dessa classe começam (construtor, linhas 2–5) identificando a conexão de controle do comutador OpenFlow, criando um dicionário para registrar os endereços MAC conhecidos e registrando-se para receber eventos gerados pelo comutador — nesse caso, eventos de `PacketIn`. O restante do código corresponde à implementação do algoritmo discutido anteriormente, onde os passos são identificados pela numeração apresentada nos comentários. As funções `drop()` e `flood()` foram ocultadas assim como as importações de pacotes para simplificar o código.

Alguns detalhes de implementação que merecem destaque são o processamento do pacote para identificação dos cabeçalhos dos diversos protocolos que o compõem (linha 12), o preenchimento da tabela de endereços (linha 13) e o acesso a diversas informações derivadas do processamento do pacote (linhas 15, 19, 23), o uso de funções de `log` para registrar a evolução do código. Finalmente, as linhas de 35 a 41 compõem a mensagem OpenFlow usada para registrar o fluxo no comutador, com a regra para encaminhar o pacote para a porta de saída identificada pela tabela de endereços MAC.

```
class LearningSwitch (EventMixin):
  def __init__ (self, connection):
    self.connection = connection # Switch que será controlada
    self.macToPort = {}          # Tabela interna
    self.listenTo(connection)    # Para ouvir eventos PacketIn

  def _handle_PacketIn (self, event):
    """
    Trata os pacotes das mensagens do switch para o algoritmo.
    Metodos auxiliares flood() e drop( ) seriam definidas aqui.
    """
    packet = event.parse()
    self.macToPort[packet.src] = event.port                    # 1

    if packet.type == packet.LLDP_TYPE:                        # 2
      drop()
      return

    if packet.dst.isMulticast():                               # 3
      flood()                                                  # 3a
      return

    if packet.dst not in self.macToPort:                       # 4
      log.debug("Port for %s unknown -- flooding" % (packet.dst,))
      flood()                                                  # 4a
      return

    port = self.macToPort[packet.dst]
    if port == event.port:                                     # 5
      log.warning("Same port. Drop.")
      drop(10)                                                 # 5a
      return

    log.debug("installing flow")
    msg = of.ofp_flow_mod()                                    # 6
    msg.match = of.ofp_match.from_packet(packet)
    msg.idle_timeout = 10
    msg.hard_timeout = 30
    msg.actions.append(of.ofp_action_output(port = port))
    msg.buffer_id = event.ofp.buffer_id                        # 6a
    self.connection.send(msg)
```

Figura 4.11. Código para implementação de um *switch* L2 com aprendizado no POX

4.7.9. Exemplo de levantamento de evento: componente de topologia

Além de poder escutar e reagir a eventos, pode ser interessante levantar um evento para ser tratado por outros componentes. Esse tipo de interface simplifica a amarração entre

diferentes módulos.

Primeiro, é necessário identificar o tipo de eventos que o componente pode levantar. No componente de topologia isso é feito da seguinte forma:

```
class Topology (EventMixin):
  _eventMixin_events = [
    SwitchJoin,
    SwitchLeave,
    HostJoin,
    HostLeave,
    EntityJoin,
    EntityLeave,
    Update
  ]
```

Além de herdar a classe `EventMixin` para poder levantar eventos, como descrito anteriormente, a lista `_eventMixin_events` é alimentada com a lista dos eventos que o módulo pode levantar. Cada um desses eventos é uma classe definida anteriormente que herda alguma classe de eventos e pode ser levantado utilizando a função `raiseEvent` da classe `EventMixin`. No caso do componente de topologia, essa função é reescrita para adicionar um tratamento especial ao evento de `Update`, mas logo em seguida a função da superclasse é chamada normalmente:

```
def raiseEvent (self, event, *args, **kw):
  """
  Sempre que um evento for levantado, um evento de Update tambem
  sera levantado, por isso reescrevemos o metodo do EventMixin
  """
  rv = EventMixin.raiseEvent(self, event, *args, **kw)
  if type(event) is not Update:
    EventMixin.raiseEvent(self, Update(event))
  return rv
```

No caso, a função da superclasse recebe, além do objeto do próprio componente, um objeto do tipo do evento a ser levantado, como é mostrado com o evento `Update`. Após essa ação de levantamento de eventos, todos os componentes que possuírem uma função de tratamento para esses eventos (como por exemplo, `_handle_HostJoin`) poderão tratá-los normalmente como qualquer outro evento do POX.

4.7.10. Utilização do componente de log

O POX tem uma funcionalidade de *logging* facilmente utilizada e configurável em diversos níveis. Para ter acesso a essa ferramenta, basta utilizar a função `getLogger` da biblioteca `core`, passando o nome do componente registrado que você deseja acessar (caso nenhum nome seja passado, o log retornado utilizará o nome do componente corrente).

Portanto, basta recuperar o log do componente corrente com a linha:

```
log = core.getLogger()
```

Essa operação foi ocultada no código da figura 4.11 por questões de espaço.

A classe de log do POX utiliza o logging padrão do python, portanto utiliza o mesmo formato de métodos e *string* para poder exibir as mensagens, como os seguintes métodos:

```
Logger.info(msg, *args, **kwargs)
Logger.warning(msg, *args, **kwargs)
Logger.error(msg, *args, **kwargs)
Logger.critical(msg, *args, **kwargs)
Logger.log(lvl, msg, *args, **kwargs)
Logger.exception(msg, *args)
```

Graças aos recursos de manipulação de *strings* de Python, é possível construir-se mensagens bastante informativas sem que as funções de *logging* precisem manipular um grande número de argumentos. Por exemplo, uma mensagem de registro mais completa para registrar a instalação de um novo fluxo no comutador (linha 34) na figura 4.11 poderia ser re-escrita como:

```
   log.debug("installing flow for %s.%i -> %s.%i" %
             (packet.src, event.port, packet.dst, port))
```

Mais detalhes sobre a classe logging do python podem ser encontrados em `http://docs.python.org/library/logging.html`.

Além disso, o POX permite que sejam declarados diferentes níveis para o log de cada componente durante a inicialização através de parâmetros para o `launch` do componente de log. Por exemplo, para carregar o componente `web.webcore` com um nível de log somente de `INFO` enquanto os outros componentes mantêm o nível de log padrão, o POX deverá ser chamado da seguinte forma:

```
./pox.py web.webcore log.level --web.webcore=INFO
```

Esse comando chama o componente `web.webcore` e o componente `log.level`, passando para o último o parâmetro `web.webcore=INFO`, o que definirá o nível de `INFO` para o log do componente `web.webcore`.

4.7.11. Testando seu componente com o Mininet

O Mininet é um simulador de rede OpenFlow que consiste em criar um conjunto de máquinas virtuais conectadas por switches OpenFlow utilizando uma interface simples para configurar, monitorar e modificar esta rede. Com o Mininet é possível simular uma rede com vários nós para testar um componente em um ambiente mais simplificado e de fácil manipulação.

No momento o Mininet oferece uma imagem de máquina virtual já instalada e configurada com a aplicação compilada e funcional. Para utilizá-la basta utilizar um

monitor de máquinas virtuais (a imagem disponibilizada é do VirtualBox em `http://www.openflow.org/downloads/OpenFlowTutorial-101311.zip`).

Ao levantar a imagem disponibilizada em um monitor de máquinas virtuais, basta logar na máquina (usuário: openflow senha: openflow) e configurar o Mininet para usar o POX como controlador.

Por exemplo, em uma máquina que o IP da máquina executando o POX seja 10.1.1.1 e é interessante que ele seja levantado no porto 6634 (o padrão é 6633), na máquina controladora deve ser levantado o POX:

```
./pox.py <componente> --address=10.1.1.1 --port=6634
```

Já na máquina virtual do Mininet é necessário apontar para o controlador levantado:

```
sudo mn --controller=remote --ip=10.1.1.1 --port=6634
```

Mais informações sobre como configurar e utilizar o Mininet podem ser encontradas nos links `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/MininetGettingStarted`, `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/MininetVMSetupNotes` e `http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/MininetWalkthrough`.

4.8. Desafios de pesquisa

O paradigma de Redes Definidas por Software abre uma grande gama de novas possibilidades para a pesquisa em Redes de Computadores. A seção 4.6 discutiu diversos contextos de aplicação em que a aplicação de SDN pode oferecer novas possibilidades de evolução e novos enfoques. Além deles, certamente outras formas de aplicação do paradigma ainda estão por ser identificados e desenvolvidos em modelos de pesquisa consolidados.

Um ponto importante a considerar é que o conceito de Redes Definidas por Software em si ainda é algo novo e muito há para se fazer no sentido de consolidar o paradigma. Nesse sentido, diversos desafios de pesquisa ainda se apresentam para aqueles que desejem avaliar os princípios norteadores dessa nova área. Questões como qual a abstração de programação mais adequada, como implementar mecanismos de depuração, como lidar com questões como distribuição do controlador para lidar com aspectos de desempenho e confiabilidade, se apresentam como áreas abertas para a pesquisa.

Visão da rede

Um dos elementos chave do conceito de SDN é a noção da visão geral da rede disponível através do controlador da rede. Os controladores atuais ainda não apresentam essa visão como um elemento estrutural básico, apesar dela poder ser construída com base nas informações derivadas das mensagens recebidas. Há uma expectativa de que

POX ofereça essa visão como um elemento arquitetônico básico, através da noção do "Modelo de Objetos da Rede", o *Network Object Model* (NOM). Essa abstração permitirá ao desenvolvedor de aplicações SDN exprimir suas demandas e as ações esperadas em função do grafo de rede e não mais como código reativo organizado ao redor dos eventos de chegada de mensagens OpenFlow.

Em um primeiro nível, o NOM deve refletir uma visão bastante clara da infraestrutura que constitui a rede física e principais elementos de software relacionados, como *switches* Open vSwitch. Os objetos dessa representação devem exportar interfaces que permitam ao desenvolvedor representar ações como a chamada de métodos implementados por esses objetos. Esses métodos, por sua vez, se responsabilizariam pela emissão de mensagens OpenFlow ou de qualquer padrão de controle dos elementos de rede que esteja disponível.

Sobre essa visão da rede física, pode-se construir novas visões que representem a visão do usuário/desenvolvedor para uma aplicação específica. Essa visão pode não conter todos os nós da rede física, incluir enlaces que não existem na rede subjacente, mas que podem ser implementados como túneis sobre a mesma, ou ainda elementos de rede virtualizados, como comutadores virtuais distribuídos.

Sistema operacional da rede

Em diversos pontos deste curso nos referimos à noção de Sistema Operacional da Rede, como o ambiente de execução que estabelece a ligação entre as aplicações SDN e o substrato da rede física. O Professor Scott Shenker, em suas palestras, tem chamado a atenção para o fato de que esse conceito pode abrir o caminho para uma nova forma de se ver e pensar Redes de Computadores, não mais em termos de artefatos tecnológicos mas de princípios organizacionais abstratos, como ocorre com a área de Sistemas Operacionais propriamente dita: pesquisadores podem utilizar noções com escalonamento de processos, algoritmos de sincronizacão e estruturas de dados para memória virtual para pensar sobre os princípios básicos da área. Com esse enfoque, um desafio importante é a identificação de quais seriam os blocos arquitetônicos essenciais para a área de Redes de Computadores que seriam adicionados ao ambiente de tempo de execução de um controlador SDN, para compor soluções como as possíveis em outras áreas da Ciência da Computação, como Sistemas Operacionais ou Bancos de Dados, que já se organizaram em torno desses conceitos básicos.

Virtualização de rede

Continuando a aplicação da noção de Sistema Operacional de Rede, chega-se rapidamente ao princípio de virtualização de redes. Esse conceito, bastante em voga recentemente em projetos como o GENI, propõe uma nova forma de se organizar as redes físicas, de forma a permitir a coexistência de diferentes aplicações, redes lógicas e mesmo arquiteturas de rede. Em sua origem, a partir da rede PlanetLab, a idéia de se utilizar máquinas virtuais para representar roteadores virtuais tem sido bastante discutida. Entretanto, esse tipo de virtualização enfrenta o desafio de vencer a barreira entre essas máquinas virtuais e uma

implementação eficiente do plano de dados nesse roteadores, uma vez que nas redes atuais a otimização do hardware de encaminhamento de pacotes atingiu patamares significativos que são difíceis de se equiparar em software.

O fato de Redes Definidas por Software darem acesso às tabelas de encaminhamento de forma programável oferece uma nova forma de se 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. Assim, visões abstratas da rede, como grafos virtuais, podem ser oferecidos a cada pesquisador/desenvolvedor que deseje implantar um novo serviço/arquitetura sobre a rede física. Como discutido na seção 4.4, FlowVisor oferece exatamente essa funcionalidade. Entretanto, continuando o paralelo com a área de Sistemas Operacionais, essa seria apenas uma das dimensões possíveis sobre a qual se organizar uma camada de virtualização. Assim como monitores de máquinas virtuais podem se organizar diretamente sobre o hardware, ou sobre a interface do Sistema Operacional, ou ainda como uma máquina virtual Java, também no contexto de SDN podem haver diferentes formas de se expressar a noção de redes virtuais, com diferentes compromissos em termos de desempenho, poder de expressão e facilidade de uso. A extensão do NOM em POX para incluir elementos virtualizados sobre a visão direta da rede física é um caminho possível, mas que ainda precisa ser melhor compreendido e melhor definido. Outros podem existir; por exemplo, o que seria o equivalente do recurso de para-virtualização no contexto de Sistemas Operacionais de Rede?

Abstrações de encaminhamento

Apesar de OpenFlow ter ganhado larga aceitação como a interface primordial de acesso aos mecanismos de encaminhamento de cada comutador da rede, a área de SDN não deveria se restringir a ela. Como mencionado anteriormente, outras soluções são possíveis e um controlador SDN deveria oferecer acesso a elas. Não só isso, mas uma solução para Redes Definidas por Software deveria oferecer uma interface de encaminhamento pela malha da rede, independente da interface particular de cada elemento de rede. Dessa forma, aplicações poderiam se concentrar em suas características específicas, fazendo uso de primitivas de encaminhamento fim-a-fim na malha.

Uma possibilidade, considerada em diferentes cenários, é que uma funcionalidade de programação como OpenFlow esteja disponível apenas nas bordas da rede, enquanto a estrutura interna da mesma não seja diretamente visível para o desenvolvedor. Isso daria espaço para os desenvolvedores de dispositivos de rede incluirem outras funcionalidades que não se casem facilmente com o modelo OpenFlow. Um exemplo de funcionalidade desse tipo seriam soluções para recuperação rápida de falhas de roteamento, difíceis de serem implementadas de forma eficiente no contexto de SDN.

Especificação de aplicações

Controladores de rede como Frenetic e FML já demonstram que aplicações para Redes Definidas por Software não necessariamente serão desenvolvidas utilizando-se linguagens imperativas. Sistemas como OMNI utilizam arcabouços de sistemas multi-agentes

para organizar suas aplicações e expressar as regras de controle que devem ser implementadas sobre a rede. Um dos desafios da área é certamente identificar quais paradigmas de programação e quais conjuntos de abstrações ("bibliotecas") se prestam melhor aos objetivos de diferentes cenários.

A integração de controladores SDN a outros ambientes, como sistemas orientados para problemas de inteligência artificial ou mineração de dados pode levar a novas formas de se descrever a interação entre os componentes de rede. Nesse sentido, o tipo de aplicação alvo e seu contexto de definição podem guiar o desenvolvimento das abstrações de programação da rede em direções até agora não imaginadas.

Depuração

A inclusão de diversos níveis de abstração entre a aplicação de rede e a infra-estrutura física cria a necessidade de regras de tradução entre o que é expressado na linguage/interface da aplicação e como os conceitos associados são colocados em prática na na rede física. Nesse processo, existe a possibilidade de que a configuração da rede física não represente exatamente o que era desejado no nível superior. Determinar quando isso acontece e, se possível, identificar as causas dessas inconsistências exige que sejam definidas técnicas de depuração (*debugging*) adequadas.

Como no caso do item anterior, as abstrações adotadas pelo controlador SDN, pela linguagem de programação associada determinarão o tipo de recursos de depuração que podem ser oferecidos. Qualquer solução adotada, entretanto, deve se pautar pelo acompanhamento das regras de transformação ao longo dos níveis de abstração, de forma a permitir que se estabeleçam paralelos entre cada abstração de alto nível e as regras de controle por ela geradas na rede física. Por exemplo, Frenetic se vale de sua natureza funcional para facilitar a depuração das regras de controle [Reitblatt et al. 2011].

Distribuição

É sempre importante relembrar que a noção da visão global da rede como algo centralizado em SDN é uma visão lógica. Não há nenhuma exigência no paradigma que determine que essa visão deve ser obrigatoriamente implementada de forma centralizada. Requisitos de desempenho ou de tolerância a falhas podem levar a uma decisão de distribuição dessa visão global. As decisões de projeto que levaram ao modelo do sistema Onix, por exemplo, mostram um caminho onde a distribuição em algum nível foi considerada essencial para a solução final. Outros sistemas, como Hyperflow [Tootoonchian and Ganjali 2010] apresentam outros caminhos para essa distribuição. Certamente, em função das diferentes abstrações de programação que podem vir a surgir em controladores SDN e das diferentes linguagens/interfaces de programação possíveis, muitas linhas de ação podem ser exploradas nesse contexto.

4.9. Considerações finais

O potencial do paradigma de Redes Definidas por Software apenas começa a ser explorado, mas já é grande o interesse pela área entre pesquisadores e empresas da área. O fato de já estarem no mercado os primeiros dispositivos comerciais baseados no padrão OpenFlow confirmam o momento positivo dessa iniciativa. Além disso, é claro interesse de diversas grandes empresas da área de dispositivos de rede e o surgimento de diversas novas empresas de tecnologia (*start-ups*) ao redor desses conceitos.

Nesse cenário, neste trabalho buscamos apresentar uma visão dos aspectos teóricos e práticos envolvidos no desenvolvimento de pesquisas em SDN. Em particular, apresentamos os elementos que devem ser considerados ao se iniciar um trabalho na área, com ênfase no controlador de rede. Esse elemento tem papel essencial em qualquer iniciativa sobre SDN, uma vez que o software que define esses sistemas deve ser desenvolvido com base nos recursos de expressão oferencidos pelo controlador. As diversas opções de controladores disponíveis foram descritas e os aspectos práticos de desenvolvimento de aplicações SDN foram discutidos à luz do controlador POX, recentemente lançado e especialmente desenvolvido para o ambiente de pesquisa e ensino.

Considerando-se o sucesso do paradigma e os desafios identificados neste texto, consideramos que existe um campo amplo para o desenvolvimento de novos projetos de pesquisa enfocando Redes Definidas por Software, seja como ferramenta para o desenvolvimento de novos serviços e aplicações de redes, seja como alvo de estudos sobre novas abstrações e soluções de implementação. Certamente, os trabalhos mais interessantes na área ainda estão por vir.

Além das referências apresentadas a seguir, alguns sítios concentram grande parte das discussões sobre os conceitos aqui apresentados. Os interessados na área devem acompanhar as atividades dos desenvolvedores em sítios como `noxathome.org`, `noxrepo.org`, `openflow.org`, `openwrt.org`, `openvswitch.org`, `opennetworking.org`, `opennetsummit.org`, entre outros.

Agradecimentos

Este trabalho foi parcialmente financiado pelo Instituto Nacional de Ciência e Tecnologia para a Web (InWeb), pelo CNPq, FAPEMIG, HP Brasil, pelo programa UOL Bolsa Pesquisa (www.uol.com.br) e pelo Programa de Visitas Brasil-ICSI do Movimento Brasil Colaborativo/ABDI. Murphy McCauley é o desenvolvedor principal do POX e colaborou com diversas informações utilizadas aqui. Muitos dos elementos apresentados sobre desafios de pesquisa na área de controladores SDN foram derivados de discussões com o Prof. Scott Shenker.

Referências

[bea 2012] (2012). The beacon controller. https://openflow.stanford.edu/display/Beacon/Home.

[flo 2012] (2012). Floodlight. http://floodlight.openflowhub.org/.

[nec 2012] (2012). Nec programmable flow. http://www.necam.com/pflow/.

[ope 2012] (2012). Openstack:open source software for building private and public clouds. http://openstack.org/.

[sna 2012] (2012). Simple network access control. http://www.openflow.org/wp/snac/.

[tre 2012] (2012). Trema: Full-stack openflow framework for ruby and c. http://trema.github.com/trema/.

[Braga et al. 2010] Braga, R. B., Mota, E. M., and Passito, A. P. (2010). Lightweight ddos flooding attack detection using nox/openflow. In *Proceedings of the Annual IEEE Conference on Local Computer Networks*, pages 408–415, Los Alamitos, CA, USA. IEEE Computer Society.

[Cai et al. 2010] Cai, Z., Cox, A. L., and Ng, T. S. E. (2010). Maestro: A system for scalable openflow control. Technical Report TR10-08, Rice University.

[Calvert et al. 2011] Calvert, K. L., Edwards, W. K., Feamster, N., Grinter, R. E., Deng, Y., and Zhou, X. (2011). Instrumenting home networks. *SIGCOMM Comput. Commun. Rev.*, 41(1):84–89.

[Casado et al. 2010a] Casado, M., Erickson, D., Ganichev, I. A., Griffith, R., Heller, B., Mckeown, N., Moon, D., Koponen, T., Shenker, S., and Zarifis, K. (2010a). Ripcord: A modular platform for data center networking. EECS Department Technical Report UCB/EECS-2010-93, University of California, Berkeley.

[Casado et al. 2009] Casado, M., Freedman, M. J., Pettit, J., Luo, J., Gude, N., McKeown, N., and Shenker, S. (2009). Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4):1270–1283.

[Casado et al. 2010b] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S. (2010b). Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 8:1–8:6, New York, NY.

[Davie and Farrel 2008] Davie, B. S. and Farrel, A. (2008). *MPLS: Next Steps: Next Steps*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Dixon et al. 2010] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, V. (2010). The home needs an operating system (and an app store). In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 18:1–18:6, New York, NY, USA. ACM.

[Elliott and Falk 2009] Elliott, C. and Falk, A. (2009). An update on the geni project. *SIGCOMM Comput. Commun. Rev.*, 39(3):28–34.

[Farias et al. 2011] Farias, F. N. N., Salvatti, J. J., Dias, J. M., Toda, H. S., Cerqueira, E., and Abelém, A. J. G. (2011). Implementação de um novo datapath openflow em ambientes de switches legados. In *Anais do II Workshop de Pesquisa Experimental em Internet do Futuro*, pages 15–18. SBC.

[Feamster 2010] Feamster, N. (2010). Outsourcing home network security. In *Proceedings of the 2010 ACM SIGCOMM workshop on Home networks*, HomeNets '10, pages 37–42, New York, NY, USA. ACM.

[Foster et al. 2010] Foster, N., Freedman, M. J., Harrison, R., Rexford, J., Meola, M. L., and Walker, D. (2010). Frenetic: a high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY.

[Gude et al. 2008] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008). Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38:105–110.

[Heller et al. 2010a] Heller, B., Erickson, D., McKeown, N., Griffith, R., Ganichev, I., Whyte, S., Zarifis, K., Moon, D., Shenker, S., and Stuart, S. (2010a). Ripcord: a modular platform for data center networking. *SIGCOMM Comput. Commun. Rev.*, 40:457–458.

[Heller et al. 2010b] Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., and Mckeown, N. (2010b). Elastictree: Saving energy in data center networks. In *Proceedings of the 7th Usenix Symposium on Networked Systems Design and Implementation (NSDI)*.

[Hinrichs et al. 2009] Hinrichs, T. L., Gude, N. S., Casado, M., Mitchell, J. C., and Shenker, S. (2009). Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pages 1–10, New York, NY.

[Kempf et al. 2011] Kempf, J., Whyte, S., Ellithorpe, J., Kazemian, P., Haitjema, M., Beheshti, N., Stuart, S., and Green, H. (2011). Openflow mpls and the open source label switched router. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 8–14. ITCP.

[Kohler et al. 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297.

[Koponen et al. 2010] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. (2010). Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA. USENIX Association.

[Lantz et al. 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 19:1–19:6, New York, NY, USA. ACM.

[Mattos et al. 2011] Mattos, D. M. F., Fernandes, N. C., da Costa, V. T., Cardoso, L. P., Campista, M. E. M., Costa, L. H. M. K., and Duarte, O. C. M. B. (2011). Omni:

Openflow management infrastructure. In *Proceedings of the 2nd IFIP International Conference Network of the Future - NoF'2011*, pages 1–5. IFIP.

[McKeown et al. 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74.

[Mehdi et al. 2011] Mehdi, S. A., Khalid, J., and Khayam, S. A. (2011). Revisiting traffic anomaly detection using software defined networking. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 161–180.

[Mogul et al. 2010] Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., Curtis, A. R., and Banerjee, S. (2010). Devoflow: cost-effective flow management for high performance enterprise networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 1:1–1:6, New York, NY, USA. ACM.

[Mundada et al. 2009] Mundada, Y., Sherwood, R., and Feamster, N. (2009). An openflow switch element for click. In *Proceedinds on the Symposium on Click Modular Router*.

[Nascimento et al. 2011] Nascimento, M. R., Rothenberg, C. E., Salvador, M. R., Corrêa, C. N. A., De Lucena, S. C., and Magalhães, M. F. (2011). Virtual routers as a service: The routeflow approach leveraging software-defined networks. In *Proceedings of the 6th International Conference on Future Internet Technologies (CFI)*, pages 0–3.

[Peterson and Roscoe 2006] Peterson, L. and Roscoe, T. (2006). The design principles of planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1):11–16.

[Pettit et al. 2010] Pettit, J., Gross, J., Pfaff, B., Casado, M., and Crosby, S. (2010). Virtual switching in an era of advanced edges. In *Proceedings of the 2nd Workshop on Data Center - Converged and Virtual Ethernet Switching (DC CAVES)*, DC CAVES, pages 1–7, Amsterdam, The Netherlands. ITC.

[Pfaff et al. 2009] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. (2009). Extending networking into the virtualization layer. In *Proceedings of the Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*.

[Ram et al. 2010] Ram, K. K., Mudigonda, J., Cox, A. L., Rixner, S., Ranganathan, P., and Santos, J. R. (2010). snich: efficient last hop networking in the data center. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 26:1–26:12, New York, NY, USA. ACM.

[Reitblatt et al. 2011] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. (2011). Consistent updates for software-defined networks: Change you can believe in. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, pages 1–6.

[Rodrigues et al. 2011a] Rodrigues, H., Santos, J. R., Turner, Y., Soares, P., and Guedes, D. (2011a). Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *Proceedings of the 3rd Usenix Workshop on I/O Virtualization (WIOV '11)*, Portland, OR. Usenix.

[Rodrigues et al. 2011b] Rodrigues, H., Soares, P. V., Santos, J. R., Turner, Y., and Guedes, D. (2011b). Isolamento de tráfego eficiente em ambientes virtualizados. In *Anais do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, pages 1–14. SBC.

[Sherwood et al. 2010] Sherwood, R. et al. (2010). Carving research slices out of your production networks with openflow. *SIGCOMM Comput. Commun. Rev.*, 40:129–130.

[Sherwood et al. 2009] Sherwood, R., Gibb, G., Yap, K.-K., Apenzeller, G., Casado, M., McKeown, N., and Parulkar, G. (2009). Flowvisor: A network virtualization layer. Tech. Rep. OPENFLOWTR-2009-1, OpenFlowSwitch.org.

[Silva et al. 2011] Silva, G., Arantes, A., Steding-Jessen, K., Hoepers, C., Chaves, M., Jr., W. M., and Guedes, D. (2011). SpSb: um ambiente seguro para o estudo de spambots. In *Anais do Simpósio Brasileiro de Segurança*, pages 1–5, Brasília, DF. SBC.

[Tennenhouse and Wetherall 2007] Tennenhouse, D. L. and Wetherall, D. J. (2007). Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(5):81–94.

[Tootoonchian and Ganjali 2010] Tootoonchian, A. and Ganjali, Y. (2010). Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA. USENIX Association.

[Turner 2006] Turner, J. S. (2006). A proposed architecture for the geni backbone platform. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 1–10, New York, NY, USA. ACM.