UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

# HAVOX: UMA ARQUITETURA PARA ORQUESTRAÇÃO DE TRÁFEGO EM REDES OPENFLOW

Rodrigo Soares e Silva

**Orientador**

Sidney Cunha de Lucena

Rio de Janeiro, RJ - Brasil
Outubro de 2017

HAVOX: UMA ARQUITETURA PARA ORQUESTRAÇÃO DE TRÁFEGO EM REDES OPENFLOW

Rodrigo Soares e Silva

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

---

Sidney Cunha de Lucena, D.Sc. - UNIRIO

---

Carlos Alberto Vieira Campos, D.Sc. - UNIRIO

---

Igor Monteiro Moraes, D.Sc. - UFF

---

José Ferreira de Rezende, Dr. - UFRJ

Rio de Janeiro, RJ - Brasil
Outubro de 2017

Silva, Rodrigo Soares e.
S586 Havox: Uma Arquitetura para Orquestração de Tráfego em Redes OpenFlow / Rodrigo Soares e Silva. – Rio de Janeiro, 2017.
118 f.

Orientador: Lucena, Sidney Cunha de.
Dissertação (Mestrado) - Universidade Federal do Estado do Rio de Janeiro, Programa de Pós-Graduação em Informática, 2017.

1. SDN. 2. OpenFlow. 3. DSL. 4. Orquestração. 5. RouteFlow
I. Lucena, Sidney Cunha de, orient. II. Universidade Federal do Estado do Rio de Janeiro. Centro de Ciências Exatas e Tecnologia. Curso de Mestrado em Informática. III. Título.

À minha mãe Eliana, a mulher mais guerreira que eu conheço, cuja crença no potencial dos filhos é inabalável e sempre moveu montanhas para nos dar todo o suporte que precisávamos. Todas as vitórias que já obtive e ainda obterei, atribuo a ela.

# Agradecimentos

Primeiramente, agradeço a toda a minha família, em especial à minha mãe Eliana e ao meu irmão Vinícius, além da minha avó Nilvete, de todos os meus tios, tias e primos. Muito obrigado por todo o apoio que ganhei na condução deste trabalho, pela paciência e pela compreensão nos vários momentos em que eu estava muito focado e por isso acabava me ausentando dos momentos familiares. Nossa família é pequena, mas nela só há pessoas ímpares!

Dos membros da família, meu agradecimento póstumo e saudades eternas ao meu pai, Márcio. De onde ele estiver, tenho certeza que tem acompanhado os meus passos e está orgulhoso de mais esta conquista minha.

Agradeço à minha amada Alessandra, namorada, amiga e cúmplice, por todas as vezes em que me colocava para cima com suas palavras reconfortantes e votos de que tudo vai dar certo no final. De fato deu! Desde que ela entrou na minha vida, venho me tornando um homem mais confiante e disciplinado, seguindo o seu exemplo como pessoa mais estudiosa que eu conheço.

Um agradecimento aos meus amigos de infância, Ricardo, Costão e Charles. Aliás, são meus irmãos! Obrigado por todo o apoio, pelas conversas motivacionais e pela paciência nas vezes que eu recusava convites para sair e para visitá-los por conta dos estudos.

Ao falar das amizades, não posso deixar de citar também aqueles que tive o imenso prazer de conhecer durante a minha trajetória como mestrando, em especial o Bruno, o Diego, a Lilian e o João Felipe, que entraram no programa comigo, bem como os meus colegas do Laboratório de Distribuição e Redes com quem tive a oportunidade de conviver e de aprender novos conhecimentos.

Gratidão também pela orientação singular do professor e amigo Sidney, com

quem tive uma parceria de mais de dois anos de pesquisas. Quem sabe serão mais! Muito obrigado por me nortear nesta árdua caminhada, pelas conversas e pela paciência até o momento de eu finalmente encontrar o meu tema.

Meus agradecimentos aos membros da banca, professores Beto, Igor e Rezende. Estou feliz por ter o meu trabalho avaliado por pessoas de referência na área. Em especial, ao professor Igor, que foi o meu orientador da graduação na UFF e agora se faz presente na avaliação do meu segundo grande passo acadêmico.

Sou grato à UNIRIO como instituição, bem como aos demais membros do corpo docente e técnicos administrativos. As aulas que tive dos professores e o serviço exemplar dos técnicos-administrativos foram de grande valia para a concretização deste trabalho.

Às equipes que desenvolveram os projetos RouteFlow e Merlin, dois dos principais componentes deste trabalho, meus agradecimentos pelas contribuições de grande relevância e que permitiram que este trabalho fosse idealizado.

Um sincero agradecimento ao caro leitor que demonstra interesse por esta pesquisa. Desde já me coloco à disposição para esclarecer qualquer ponto, tenha passado o tempo que for, e faço votos de que, caso use os frutos deste trabalho, que estes sejam de grande valia e que as derivações sejam para contribuir ainda mais com a ciência.

Por último, porém mais importante que tudo, agradeço a Deus, Aquele que está nos maiores eventos cósmicos e ao mesmo tempo nas menores ligações da matéria. Durante a minha trajetória no curso, várias vezes pensei em desistir, mas Ele me dava sinais claros para que continuasse seguindo em frente, sinais esses na forma de situações e acontecimentos que se desdobravam tão incrivelmente que me faziam refletir e me manter na caminhada. Me considero um homem cético para quase tudo nesta vida, mas da presença e atuação Dele eu não tenho dúvidas.

*"Todos nós temos nossas máquinas do tempo.*
*Algumas nos levam de volta, elas são chamadas recordações.*
*Algumas nos levam adiante, elas são chamadas sonhos."*
Jeremy Irons, em *A Máquina do Tempo*

Silva, Rodrigo Soares e. **Havox: Uma arquitetura para orquestração de tráfego em redes OpenFlow**. UNIRIO, 2017. 99 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

## RESUMO

As redes definidas por *software* já vêm conferindo maior poder de controle aos administradores de rede, mas gerar muitas regras OpenFlow simultaneamente ainda é um trabalho árduo. Faz-se necessário antever cada tipo de tráfego, ou no mínimo implementar o tratamento a ser executado na camada de controle para pacotes ainda desconhecidos. Já existem projetos de pesquisa que geram um conjunto de regras OpenFlow para operações básicas de rede, bem como ferramentas com gramática própria que interpretam descrições de políticas de rede em alto nível e as traduzem para regras a serem instaladas nos *switches*. Observou-se a oportunidade de combinar o funcionamento dessas ferramentas em um novo recurso que também forneça ao administrador uma linguagem de configuração amigável para definir como o tráfego deve ser encaminhado no domínio. Este trabalho apresenta o Havox, uma arquitetura que visa auxiliar a orquestração do tráfego passante num domínio através de uma descrição simples e baseada nos campos OpenFlow. Como prova de conceito, a arquitetura Havox é aplicada a situações em que há mais de uma rota conhecida para quaisquer prefixos de rede, o que torna possível o uso do protocolo OpenFlow e seus campos, como por exemplo a porta da aplicação, para selecionar diferentes tipos de tráfego e encaminhá-los por saídas distintas. Sua sintaxe é interpretada e gera um conjunto de regras que possuem prioridade superior àquelas criadas pela plataforma de roteamento IP RouteFlow, que também foi integrada à pilha de aplicações de rede. Dessa forma, pacotes que correspondam às regras criadas pelo administrador com o uso do Havox terão tratamento diferenciado e podem ser balanceados conforme planejado, enquanto os demais pacotes serão tratados normalmente pelas regras criadas pelo RouteFlow.

**Palavras-chave:** SDN, OpenFlow, DSL, Orquestração, RouteFlow.

## ABSTRACT

*Software-defined networking has been providing more control to network administrators, but generating several OpenFlow rules simultaneously is yet a tough job. It is necessary to predict every single kind of traffic, or at least implement the treatment to be executed in the control layer for yet unknown packets. There are already research projects which generate a set of OpenFlow rules for basic network operations, as well as tools with their own grammars which parse high level network policy descriptions and translate them to rules to be installed in the switches. An opportunity was observed to combine some of those tools in a new resource capable of giving to the network administrator a friendly configuration language to help him define how the data traffic should be forwarding inside the domain. This work presents Havox, an architecture to help data traffic orchestration inside a domain using a simple, OpenFlow fields-based description. As a proof of concept, Havox architecture is meant to be used in situations where there are multiple known routes for any network prefix, which makes using OpenFlow protocol fields like the application port possible, for instance, in order to select different types of traffic and forward them through distinct exits. Its syntax is interpreted and generates a set of rules that have more priority than those created by the RouteFlow IP routing platform, which was also integrated to the network application stack. Therefore, packets that match rules created by the administrator using Havox will have a distinct treatment and can be balanced as planned, while other packets will be handled normally by RouteFlow-created rules.*

**Keywords:** SDN, OpenFlow, DSL, Orchestration, RouteFlow.

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos

# Lista de Nomenclaturas

| | |
|---|---|
| API | *Application Programming Interface* |
| AS | *Autonomous System* |
| BGP | *Border Gateway Protocol* |
| DSL | *Domain-Specific Language* |
| EGP | *Exterior Gateway Protocol* |
| FIB | *Forwarding Information Base* |
| FTP | *File Transfer Protocol* |
| HTTP | *Hypertext Transfer Protocol* |
| HTTPS | *Hypertext Transfer Protocol Secure* |
| IANA | *Internet Assigned Number Authority* |
| ID | *Identifier* |
| IGP | *Interior Gateway Protocol* |
| IP | *Internet Protocol* |
| IPC | *Inter-Process Communication* |
| ISP | *Internet Service Provider* |
| IXP | *Internet Exchange Point* |
| JSON | *JavaScript Object Notation* |
| LACNIC | *Latin America and Caribbean Network Information Centre* |
| LXC | *Linux Containers* |
| MAC | *Media Access Control* |
| NLRI | *Network Layer Reachability Information* |
| ONF | *Open Networking Foundation* |
| OSPF | *Open Shortest Path First* |
| P4 | *Programming Protocol-independent Packet Processors* |
| PID | *Process Identifier* |
| QoS | *Quality of Service* |

| | |
|---|---|
| RIB | *Routing Information Base* |
| RIP | *Routing Information Protocol* |
| RIR | *Regional Internet Registry* |
| RVM | *Ruby Version Manager* |
| SDN | *Software-Defined Networking* |
| SLA | *Service Level Agreement* |
| SQL | *Structured Query Language* |
| SSH | *Secure Shell* |
| TCP | *Transmission Control Protocol* |
| TLS | *Transport Layer Security* |
| VLAN | *Virtual Local Area Network* |

# 1. Introdução

A *Internet* é uma rede de redes, as quais cada uma possui uma administração diferente com políticas e acordos diferentes [1].

Esses domínios administrativos, os quais são referenciados como sistemas autônomos (AS, do inglês *Autonomous System*), têm suas próprias estratégias de negócios no que tange ao tráfego de pacotes que passam por seus roteadores e enlaces. Independentemente das decisões internas, na teoria um AS não exerce controle sobre atributos específicos de camadas superiores da pilha de protocolos da *Internet*, como as portas de transporte ou o endereço de origem. Em uma rede tradicional, o controle é possível somente sobre o endereço de destino. Há, porém, a intenção de se obter esse controle baseado em camadas superiores, mas ainda é muito difícil para os ASes alcançar isso.

Roteadores que executam protocolos de roteamento específicos tomam decisões baseadas no conhecimento individual que cada um tem das imediações [2]. Para tanto, rotas são propagadas interna e externamente, o que permite que os roteadores mantenham cada um uma tabela de roteamento atualizada, a partir da qual decisões sobre o melhor caminho para um pacote podem ser adotadas. Alguns protocolos de roteamento podem manter uma visão global da rede [3, 4]. Esses podem ser usados para roteamento interno dentro de um domínio de dispositivos conhecidos, mas o conhecimento global necessário da rede os torna inviáveis para o roteamento externo que compreende à *Internet*.

Com o avanço das pesquisas em redes definidas por *software*, uma nova perspectiva de controle sobre o tráfego passante se abre para os domínios administrativos. A manutenção da lógica de decisão em um elemento controlador centralizado executando em *hardware* genérico em vez de em cada roteador de forma distribuída permitiu que um controle mais granular sobre os pacotes pudesse ser

exercido. Se configura, assim, a visão de uma pilha de rede definida por *software* contendo as camadas de dados (onde ocorre o encaminhamento), de controle (onde é orquestrado o encaminhamento) e de aplicação (onde residem aplicações que adicionam lógicas ainda mais sofisticadas), sobrepostas nessa ordem.

Esforços de pesquisa têm sido feitos em todas as camadas dessa pilha para a solução de questões em aberto envolvendo redes de computadores. Um dos principais esforços, certamente, é na busca por soluções para o problema de gerenciamento de tráfego dentro dos ASes [5, 6].

No escopo de redes definidas por *software* e com o problema supracitado em evidência, soluções como plataformas de roteamento IP e linguagens específicas de domínio (DSL, do inglês *Domain Specific Language*) [7] que facilitam a configuração da rede ganham destaque porque tentam lidar não apenas com a complexidade intrínseca de configuração e manutenção da rede, mas também com a viabilização do emprego de redes definidas por *software* em cenários reais e já maduros de roteamento na *Internet*.

Centrado nesses pontos, foi idealizado este trabalho, que propõe uma arquitetura flexível que une duas soluções acadêmicas relevantes no tema, a saber, o RouteFlow [8, 9] e o Merlin [10, 11, 12]. Seus papéis na arquitetura serão descritos mais adiante. Essa integração é garantida com uma camada adicional de abstração sobre ambas as soluções, que fornece uma DSL legível e de fácil utilização, de forma que uma linha de configuração na DSL é capaz de gerar múltiplas regras de encaminhamento. Essa DSL define a forma como o encaminhamento do tráfego na camada de dados se dará. Além disso, a DSL pode ser aprimorada com novas funcionalidades conforme os casos de uso.

Como prova de conceito, um cenário de testes com oito sistemas autônomos é montado, sobre o qual um dos ASes executa a arquitetura proposta e os demais ASes anunciam rotas para prefixos de rede. Os resultados mostram que a arquitetura, alimentada por diretivas de orquestração na DSL implementada e munida do conhecimento das rotas anunciadas, é capaz de orquestrar com êxito o tráfego conforme as configurações definidas. Testes realizados com um gerador de tráfego também mostram que os pacotes são encaminhados internamente de acordo com o que foi configurado.

## 1.1 Objetivos

Este trabalho tem como objetivo principal fornecer uma arquitetura para orquestração de tráfego em redes definidas por *software*, tomando como componentes da sua pilha de aplicações uma plataforma de roteamento IP como o RouteFlow [8, 9] e um *framework* de gerenciamento de caminhos de rede que forneça uma DSL capaz de ser usada como código intermediário, como o Merlin [10, 11, 12].

A hipótese que espera-se elucidar é se dois componentes como os citados, que têm propósitos diferentes dentro de um mesmo campo, podem ser complementares ao ponto de juntos proverem maior abstração e facilidade de configuração do comportamento da rede mediante o uso de uma linguagem próxima à humana.

A escolha dos componentes RouteFlow e Merlin é em razão de ambos já implementarem funcionalidades que são necessárias para a operação da arquitetura. Como foi citado anteriormente, as soluções se propõem a resolver problemas distintos de redes definidas por *software*, mas podem ser empregados de forma sinérgica sob o controle de um terceiro componente que atua como interposto entre ambos. Esse terceiro componente, o cerne da arquitetura, é uma biblioteca que mantém comunicação com o RouteFlow e com o Merlin por meio de interfaces *web* e SSH (*Secure Shell*).

Nesta prova de conceito, o RouteFlow e o Merlin são requisitos para a operação da arquitetura Havox, visto que são feitas consultas às tabelas de roteamento das instâncias mantidas pelo RouteFlow e utiliza-se expressões regulares específicas para a saída padrão do Merlin. No entanto, é possível a adaptação da arquitetura para outras soluções similares, especialmente após a realização dos trabalhos futuros que já foram identificados.

## 1.2 Metodologia de pesquisa

A metodologia de pesquisa que norteou este trabalho é a *Design Science*, devido à intenção desta pesquisa de resolver um problema de ordem prática ao implementar um artefato de *software* enquanto se preocupa em expandir o conhecimento científico acerca do campo de atuação [13, 14]. O artefato de *software* referido é a arquitetura Havox.

O trabalho é validado com um estudo de caso, no qual um cenário emulando um ambiente real é usado e são observadas as alterações teorizadas nas tabelas de fluxo dos *switches* avaliados.

O artefato implementado neste trabalho pode ser utilizado como base para novas implementações de expansão, seja para melhorar uma funcionalidade, seja para resolver um novo problema. O artefato, isto é, a arquitetura, pode ser dividida em partes que também podem ser usadas de forma independente para futuros aprimoramentos.

## 1.3 Resumo das contribuições

Uma das contribuições deste trabalho, a principal, é a orquestração de tráfego por meio de uma arquitetura que une a plataforma de roteamento IP RouteFlow e o *framework* de gerenciamento de caminhos de rede Merlin com uma camada de abstração sobre ambos, representada por uma biblioteca que atua como componente central da arquitetura e mantém comunicação com os demais componentes.

A DSL criada para configurar a arquitetura através de diretivas é a segunda contribuição, provendo fácil configuração e possibilidade para expansões futuras conforme a demanda. A DSL é desenvolvida sobre a linguagem de propósito geral Ruby[1] e, portanto, utiliza os recursos e vantagens da mesma.

Outra contribuição implícita é a possibilidade de implantação do núcleo da arquitetura, onde está centralizada toda a lógica de criação de regras, como um serviço remoto na nuvem [15], seguindo a tendência da decomposição de sistemas em microsserviços com funcionalidades específicas e bem definidas [16].

Há também as contribuições diretas no código-fonte dos componentes utilizados, o qual um módulo especial foi desenvolvido na plataforma de roteamento IP usada para que esta possa fazer requisições de regras à interface *web* da biblioteca central. Ainda com relação a código, outra contribuição é o suporte a mais atributos de correspondência dos pacotes além do que a plataforma de roteamento IP já suporta, como o endereço IP de origem, necessários para que a arquitetura opere conforme o esperado.

[1] Disponível em `https://www.ruby-lang.org/en/`.

## 1.4 Organização do trabalho

Passado este capítulo introdutório e motivacional, este trabalho procede com a seguinte organização textual em uma abordagem *top-down* de detalhes:

- O capitulo 2 é dedicado para a fundamentação teórica, fornecendo todo o arcabouço teórico necessário para o entendimento da arquitetura, da sua execução e da avaliação dos resultados.
- O capítulo 3 aborda a proposta da arquitetura introduzida em um nível mais alto, explicando como os conceitos teóricos se encaixam e como a solução é projetada. Nesse capítulo também são citados outros trabalhos que guardam similaridade com este.
- O capítulo 4 se preocupa em elucidar os detalhes da implantação da arquitetura em um aspecto técnico, abordando desafios encontrados, decisões de implementação adotadas, os recursos computacionais e a configuração de todos os componentes da pilha de aplicações da arquitetura.
- O capítulo 5 valida a proposta com a descrição de um experimento usando um cenário de teste que executa a arquitetura, instala as configurações definidas pelo usuário e é legitimada com testes usando um gerador de tráfego. São discutidos os resultados observados, as vantagens do uso da arquitetura, os fatores de desempenho e escalabilidade e as limitações identificadas.
- O capítulo 6 encerra este trabalho com uma retrospectiva do que foi feito e das contribuições realizadas. Aponta também trabalhos futuros, possibilidades de expansão da arquitetura e finaliza com considerações acerca do campo de atuação.

# 2. Fundamentação teórica

Este capítulo explica os recursos teóricos nos quais este trabalho se consolida, bem como discorre sobre as soluções RouteFlow e Merlin, que são trabalhos acadêmicos que integram a arquitetura Havox.

## 2.1 Roteamento na *Internet*

A *Internet* é uma rede global formada por inúmeras redes menores, cada qual com suas características e políticas internas [1]. A rede dessas entidades menores que compõe a *Internet* é conhecida como sistema autônomo, ou AS. Não é necessário que um AS saiba como seus vizinhos operam internamente, e em geral cada um emprega um conjunto de tecnologias e protocolos internos mais conveniente para seus negócios, mas eles devem compartilhar de um protocolo externo padronizado a fim de que possam trocar dados. Esse protocolo é o BGP (*Border Gateway Protocol*) e como é um tema central neste trabalho, será detalhado em uma subseção própria mais adiante neste capítulo.

Nem todas as redes podem ser referidas como sistemas autônomos. Para uma rede ser um AS, ela deve poder divulgar a outros as rotas que levam aos seus computadores e sub-redes, deve ter um número de AS único que o identifica e também deve possuir seu próprio conjunto de endereços de rede públicos, alocados por um Registro Regional de Internet (RIR, do inglês *Regional Internet Registry*), que por sua vez tem prefixos de rede atribuídos pela IANA (*Internet Assigned Number Authority*). O RIR responsável pela alocação de prefixos de rede no Brasil e na América Latina é o LACNIC (*Latin America and Caribbean Network Information Centre*). Os sistemas autônomos são categorizados em *stub*, *multi-homed* e de trânsito [17].

Um AS *stub*, ou *single-homed*, possui apenas um enlace de saída para a *Internet*, isto é, possui apenas um vizinho. Sistemas autônomos nessa categoria em geral são grupos que preferem não depender de um provedor de acesso para obterem endereços de rede que os permita acessar a *Internet*, e por terem apenas um enlace de saída, não podem dar trânsito para outros sistemas autônomos.

Um AS *multi-homed* possui múltiplas conexões de saída ou múltiplos vizinhos. Também usa das mesmas intenções que justifiquem ser um AS, tal qual um AS *stub*, mas com redundância em caso de falha de um dos enlaces de saída para a *Internet*, assegurando assim a sua interoperação. Apesar de terem múltiplos vizinhos, um AS nessa categoria não fornece trânsito para os demais.

Um AS de trânsito é um AS *multi-homed* que fornece trânsito de pacotes para seus vizinhos por meio de acordos de nível de serviço (SLA, do inglês *Service Level Agreement*). Esses acordos preveem questões políticas como que tipo de tráfego será suportado e o sentido do fluxo de dados.

Ainda há também uma classificação de sistemas autônomos em *tiers* 3, 2 e 1. Um AS *tier* 3 compra acordos de trânsito de dados de todos os seus vizinhos e geralmente são ISPs (*Internet Service Provider*) menores que interligam redes privadas à *Internet*. Um AS *tier* 2 também compra acordos de trânsito de alguns vizinhos, mas possui acordos de *peering* com outros, o que na prática são acordos de trânsito gratuito e mútuo, beneficiando ambos os grupos. Um AS *tier* 1 possui acordos de *peering* com todos os seus vizinhos e não compra acordos de trânsito com nenhum, além de possuírem acesso a todas as demais redes da *Internet*, tornando-os os principais sistemas centrais que compõem o seu núcleo.

No experimento de validação conduzido neste trabalho, o AS em foco que usa a arquitetura Havox possui quatro vizinhos, o que o enquadra a princípio na categoria *multi-homed*, abstraindo-se de SLAs relacionados a trânsito de pacotes. Porém, a arquitetura criada pode ser usada para qualquer tipo de AS em ambas as classificações.

### 2.1.1 A infraestrutura da *Internet*

De um ponto de vista lógico de infraestrutura, um AS é formado por todos os computadores, tecnologias e sub-redes que estão sob a sua administração, abstraindo assim de como é a infraestrutura física da *Internet*. Essa visão física pode ser dividida nos níveis de usuário, de acesso e de núcleo.

A infraestrutura no nível de usuário são os equipamentos, como *modems*, que conectam um usuário ou uma rede privada a uma rede de acesso. Os equipamentos de uma rede de usuário são mais simples pois não têm a necessidade de armazenar e nem de processar rotas para todas as redes da *Internet*, e também obtêm seus endereços de rede de um provedor de acesso, o que os tornam muito baratos.

A infraestrutura no nível de acesso compreende as tecnologias que conectam usuários às redes de núcleo da *Internet*. Como os equipamentos de nível de usuário atuam com velocidades muito inferiores às de nível de núcleo, no nível de acesso geralmente são empregadas técnicas para que múltiplos usuários compartilhem de um enlace de mais alta capacidade, como a multiplexação de conexões. Mas nem sempre isso é regra, visto que o usuário pode ser uma empresa que requer uma conexão de alta velocidade.

A infraestrutura no nível de núcleo são os equipamentos de alta capacidade e poder de processamento que mantêm o tráfego de pacotes da *Internet* funcionando. Esse nível é mantido por operadoras de telecomunicações, órgãos governamentais, redes de pesquisa e grupos que gerenciam enlaces de alta velocidade e oferecem trânsito de dados a outros grupos. É nesse nível que ocorrem os anúncios de rotas e são tomadas as decisões de roteamento interdomínio com base nas políticas de gerenciamento e de negócios de cada AS.

Dentre os equipamentos citados, destacam-se os roteadores. Em uma rede tradicional, os roteadores são os elementos de rede que tomam as decisões de como os fluxos de pacotes serão roteados. Os roteadores internos em geral mantêm em memória a topologia da rede do domínio e fazem o repasse com base no estado dos enlaces e nós. Já os roteadores de borda têm visão apenas dos vizinhos do AS e se baseiam em anúncios de rotas para efetuar decisões. Em ambos os tipos, os roteadores devem ser configurados de modo que operem com coerência entre si e em compatibilidade com as políticas de negócios do AS, uma vez que todos possuem camadas de controle e de dados próprias e operam de forma independente uns dos outros. Um novo conceito de rede que desacopla essas camadas de controle e de dados dos roteadores tem sido pesquisado e cada vez mais empregado. Tal conceito, de redes definidas por *software*, será abordado na Seção 2.2.

### 2.1.2 Roteamento interno e externo

Os roteadores internos interligam outros roteadores internos, computadores e sub-redes dentro do domínio do AS, rodam protocolos de roteamento IGP (*Interior Gateway Protocol*) e processam rotas apenas pertinentes às sub-redes internas. Já os roteadores de borda mantêm vínculos com roteadores de borda de outros sistemas autônomos e processam rotas para qualquer rede da *Internet* rodando protocolos de roteamento EGP (*Exterior Gateway Protocol*). Os sistemas autônomos têm liberdade para definirem o protocolo IGP que melhor atenda aos seus negócios, porém é fundamental que executem um protocolo EGP padrão em comum com os demais sistemas autônomos para que seja possível a interoperabilidade.

Alguns protocolos de roteamento interno, como o OSPF (*Open Shortest Path First*) [3], manipulam duas estruturas de dados especiais dos roteadores que armazenam as rotas conhecidas: a RIB (*Routing Information Base*) e a FIB (*Forwarding Information Base*). Na RIB são mantidas todas as rotas aprendidas pelo roteador, mesmo quando se trata de múltiplas rotas para um mesmo prefixo de rede. Dessas múltiplas rotas, uma sempre é adotada como sendo a melhor rota para um determinado destino, de acordo com o seu custo menor em relação às restantes, custo esse calculado com base em critérios como número de saltos, condições do enlace e configurações no âmbito do SLA do AS. A RIB geralmente tem muitas entradas, o que pode tornar o encaminhamento de pacotes ineficiente à medida que a mesma cresce, pois o roteador teria que ficar consultando a melhor rota dentre várias para um mesmo destino. Como a condição de melhor rota costuma perdurar por algum tempo, essa rota é também incluída na FIB, que pode já conter inclusive o endereço MAC (*Media Access Control*) do próximo salto para aquela rota. A FIB, ao contrário da RIB, mantém apenas as melhores rotas para os destinos conhecidos, permitindo que o roteador encontre mais rapidamente a entrada necessária para encaminhar o pacote. Em caso de uma rota na FIB deixar de ser a melhor ou de existir, a próxima melhor rota da RIB é instalada na FIB.

Nem todos os protocolos de roteamento interno mantêm uma RIB e uma FIB nos roteadores simultaneamente. O protocolo RIP (*Routing Information Protocol*) [4] mantém apenas FIBs e é um protocolo de vetor de distâncias, ao contrário do OSPF, que é um protocolo de estado de enlace. Os caminhos entre os nós são medidos com base no número de saltos entre eles, onde tabelas são difundidas entre os roteadores para que os mesmos possam atualizar as suas rotas conhecidas. Com isso, cada roteador atualiza a sua FIB com a melhor rota para um determinado

endereço. O protocolo RIP é adequado para domínios de maior tamanho, posto que se torna mais difícil para cada roteador rodando um protocolo de estado de enlace manter informações do estado corrente de uma rede grande. Apesar disso, o protocolo RIP tem a desvantagem de vir com um tempo considerável de convergência interna das rotas. Alguns ASes combinam o RIP e o OSPF de forma que grupos de roteadores se comuniquem externamente usando o RIP e internamente usando o OSPF.

A prova de conceito produzida neste trabalho faz consultas às tabelas de roteamento de cada roteador virtual, obtém todas as rotas para cada destino conhecido e as mescla em uma única tabela de rotas conhecidas. Com isso, ela conhecerá todas as rotas existentes para um determinado endereço e se torna capaz de criar políticas que envolvam o encaminhamento de tipos distintos de tráfego por saídas do domínio para destinos conhecidos. Mais detalhes sobre essas políticas são descritos na Seção 2.5 e no Capítulo 3.

### 2.1.3 O protocolo BGP

O protocolo BGP (*Border Gateway Protocol*) [2] é um protocolo de roteamento do tipo EGP que opera sobre o protocolo TCP na porta 179. No presente, o BGP é o único protocolo padronizado e comumente aceito para interoperabilidade de sistemas autônomos. Sua operação se dá pela propagação de vetores de caminho, diferenciando-se de outros algoritmos de vetor de distância porque propaga uma lista completa de saltos da origem até o destino, o que evita que *loops* de roteamento ocorram. No caso, cada salto corresponde a um AS e essa lista é composta pelos números de AS, que são identificadores numéricos únicos atribuídos a cada AS pela RIR correspondente.

#### 2.1.3.1 Tipos de mensagem

Uma mensagem BGP pode ser de quatro tipos cujos identificadores são indicados no campo Tipo do cabeçalho.

Uma mensagem *Open* é transmitida entre roteadores de sistemas autônomos distintos, quando um deles pretende estabelecer uma sessão BGP com o outro. Ao contrário de outros protocolos de roteamento, nós BGP não são descobertos, mas previamente configurados pelo administrador da rede, em virtude da existência ou não de SLAs entre os sistemas autônomos participantes da sessão.

Uma mensagem *Update* indica se rotas devem ser incluídas ou removidas dos vizinhos cujas sessões BGP já foram estabelecidas. A mensagem tem as seções *Withdrawn Routes*, *Path Attributes* e NLRI (*Network Layer Reachability Information*). No caso de uma inclusão, é transmitida uma mensagem com os campos *AS Path* e *Next Hop* da seção *Path Attributes* preenchidos, respectivamente, com a lista de números de AS que compõem o caminho até o prefixo e com o endereço de rede do próximo salto, bem como a seção NLRI também é preenchida com todos os prefixos acessíveis com as informações da seção *Path Attributes*. No caso de uma remoção, a seção *Withdrawn Routes* é preenchida com os prefixos de rede cujas rotas não são mais válidas. Uma mesma mensagem *Update* pode conter informações de inclusão e de remoção de rotas, bastando que os campos citados estejam preenchidos. Na literatura, mensagens *Update* contendo inclusões de rotas também são chamadas de anúncios.

Uma mensagem *Notification* é transmitida quando um erro ocorre, culminando no encerramento da sessão BGP vigente. A mensagem é preenchida com o código do erro, que pode ser relacionado a *loop* de rotas detectado, próximo salto (campo *Next Hop*) inválido, versão do protocolo BGP não suportada, entre outros.

Mensagens *Keep Alive* são enviadas periodicamente durante sessões estabelecidas. O ideal é que essas mensagens sejam configuradas para serem enviadas dentro de um intervalo de tempo negociado por ambos os sistemas autônomos vizinhos. Quando nem mensagens *Keep Alive* e nem mensagens *Update* são recebidas dentro desse intervalo em um número de vezes também configurado, considera-se que o AS que deveria ter enviado a mensagem não está mais respondendo e, portanto, a sessão é encerrada e as rotas que passam por ele são removidas. É possível que o intervalo de tempo seja configurado como zero, desta forma não sendo necessário o envio de mensagens *Keep Alive* entre os vizinhos. Isso, porém, não é recomendado porque pode causar o descarte de fluxos inteiros de pacotes, se enviados a um vizinho que não se tem mais conhecimento de que está funcional.

#### 2.1.3.2 Implantação

Conforme citado anteriormente, o BGP é o único protocolo interdomínio utilizado a nível global pelos sistemas autônomos. Em uma visão macroscópica, cada AS corresponde a uma unidade de roteamento em um conjunto executando o BGP como protocolo padrão, onde cada um abstrai totalmente de como os vizinhos implementam o BGP e os protocolos de roteamento interno. Isso confere

liberdade ao AS de determinar como os protocolos devem ser implementados, contanto que recebam e respondam mensagens compatíveis com o formato esperado externamente.

O AS pode executar o BGP em roteadores convencionais, que mantém a lógica de operação do protocolo em suas respectivas camadas de controle. O BGP não precisa nem mesmo ser executado em roteadores, podendo também ser executado em computadores genéricos que se comportem como roteadores, que sejam capazes de processar e anunciar rotas e de manter sessões BGP com roteadores de domínios vizinhos, desde que existam enlaces entre os pares. No contexto das redes definidas por *software*, conceito descrito na seção seguinte deste capitulo, o BGP pode executar em um plano acima da camada de controle, com o controlador fazendo o intermédio entre a máquina executando o protocolo e o dispositivo da camada de dados que possui um enlace com um roteador de um AS vizinho. É dessa forma que o RouteFlow trabalha, conforme explicado mais adiante na Seção 2.6.

## 2.2 Redes definidas por *software*

Recapitulando o que foi abordado na Subseção 2.1.1, uma arquitetura de rede tradicional é composta, em princípio, por dispositivos de encaminhamento pré-configurados e interligados fisicamente por meio de enlaces de comunicação. Esses dispositivos operam de acordo com as configurações definidas pelo administrador da rede, o que faz com que tenham autonomia na hora de realizar o encaminhamento de um fluxo de pacotes ou executem algoritmos específicos que determinam como o fluxo será tratado. Com isso, cada dispositivo opera de maneira independente em relação aos demais e, na maioria das vezes, requerem configuração manual e individualizada. Cada dispositivo possui as suas próprias camadas de controle e de dados, fortemente acopladas, sendo na primeira onde as decisões de encaminhamento são tomadas e na última onde o ato do repasse dos pacotes de fato ocorre.

Do momento da popularização da *Internet* até meados dos anos 2000, esse tipo de arquitetura com dispositivos independentes entre si era suficiente para o modo de navegar do usuário, em uma época onde o uso da rede era centrado na localização dos recursos. No entanto, esse tipo de arquitetura tornou-se insuficiente para suprir as demandas de um modo de uso orientado a conteúdo,

abstraindo-se de sua localização. Mais ainda, a configuração manual e distribuída dos dispositivos de encaminhamento não é escalável, sendo extremamente difícil e suscetível a erros realizar a configuração individual dos dispositivos de forma que sejam independentes, mas operem em harmonia com os SLAs da rede. O acoplamento entre as camadas de controle e de dados dos dispositivos acaba se tornando um obstáculo para a obtenção de um comportamento flexível e mais trivialmente programável da rede.

O conceito de rede definida por *software*, comumente referenciada pela sigla SDN (*Software-Defined Networking*) [18] e também conhecida como rede programável, separa as camadas de controle e de dados dos dispositivos da rede, submetendo a regência do repasse de pacotes da camada de dados a um dispositivo controlador logicamente centralizado. Este implementa *handlers* para eventos da rede, como quando um dispositivo se conecta ou quando um pacote novo é recebido e requer tratamento. Esse controlador orquestra o comportamento da rede como um todo e atua como uma ponte entre a rede física subjacente e as aplicações que fazem uso dela. Para tanto, o controlador fornece uma API (*Application Programming Interface*) norte para comunicação com as aplicações de alto nível e uma API sul para orquestração da camada de dados [19]. As camadas de aplicação, de controle e de dados têm uma organização lógica em pilha, conforme mostra a Figura 2.1.

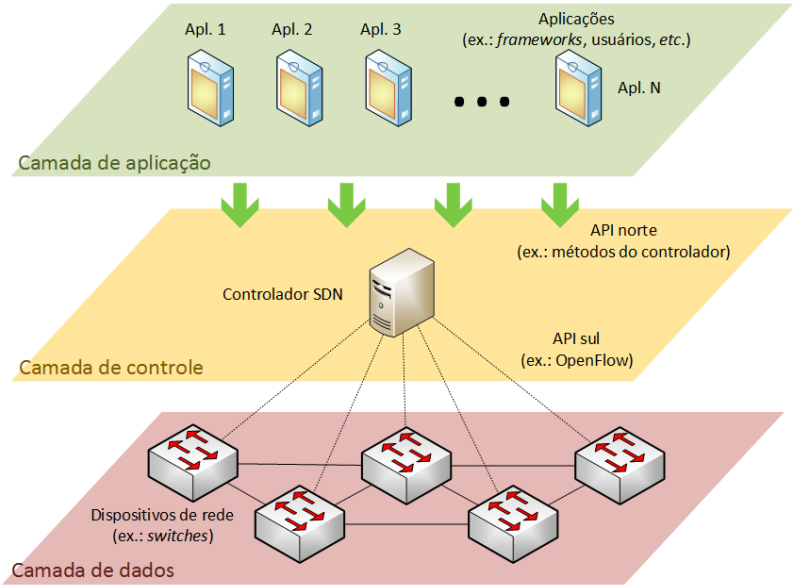



Figura 2.1: Pilha de camadas em SDN.

A API norte é um conjunto de métodos, funções ou *endpoints* que o controlador implementa e expõe às aplicações de rede. Estas, por sua vez, executam conforme

suas implementações e consomem serviços do controlador, seja fazendo leitura do estado da rede subjacente ou enviando instruções para modificar o comportamento dos dispositivos.

Por seu turno, a API sul é um conjunto de instruções legíveis pelos dispositivos da camada de dados. Geralmente, a comunicação entre as camadas de controle e de dados se dá pelo protocolo OpenFlow [20], descrito na seção seguinte, pois é o protocolo padrão aceito pela ONF (*Open Networking Foundation*) e é também o mais utilizado nas pesquisas em SDN, incluindo este trabalho. No entanto, existem outras propostas de API sul na literatura, como o NETCONF [21], OVSDB [22], LISP [23], POF [24] e OpFlex [25].

A desvinculação de *hardware* e *software* nas redes definidas por *software* trazem várias vantagens no gerenciamento da rede. O controlador pode ser implementado em uma máquina convencional com *hardware* de propósito geral, em uma máquina virtual ou até mesmo em uma máquina remota, não necessariamente sendo um dispositivo específico de rede, podendo inclusive ser escalada com mais recursos computacionais conforme a demanda. Os dispositivos da camada de dados também necessitam apenas implementar o protocolo de comunicação com o controlador, não necessitando mais que sejam utilizados sistemas fechados e proprietários. Com isso, a rede como um todo se torna escalável, flexível e mais fácil de passar por manutenções por conta da lógica de controle centralizado, sendo, por fim, também menos custosa economicamente.

As vantagens de redes definidas por *software* reduzem a propensão a erros decorrentes da configuração manual e individual dos dispositivos. Essa configuração passa a ser realizada pelo dispositivo controlador, que executa um algoritmo programado pelo operador e que deve prever como lidar com os pacotes passantes. Isso acaba por incorporar a propensão a outros tipos de falhas, como o encaminhamento ou o descarte errôneo de pacotes e o próprio controlador se tornar um ponto de falha. Por isso, há esforços de pesquisas que visam elevar o nível de abstração da rede com o intuito de facilitar a programação do seu comportamento [7, 10, 26, 27]. Este trabalho está inserido nesse escopo, pois fornece como uma de suas contribuições uma DSL para a orquestração do comportamento da rede sob a regência da arquitetura proposta. O conceito de DSL será abordado mais adiante neste capítulo.

Na literatura, um dispositivo da camada de dados é também chamado de comutador OpenFlow, *switch* ou *datapath*. Para fins de padronização, deste ponto

em diante este trabalho tratará esses dispositivos pela denominação "*switch*".

## 2.3 O protocolo OpenFlow

OpenFlow [20] é um protocolo de comunicação aberto que opera entre as camadas de controle e de dados de uma rede definida por *software* e permite que um administrador de rede possa programar como os pacotes de dados irão trafegar por entre os *switches* do domínio. O protocolo OpenFlow opera sobre o protocolo da camada de transporte TCP (*Transmission Control Protocol*), geralmente com o suporte do protocolo criptográfico TLS (*Transport Layer Security*), utilizando as portas lógicas 6633 ou 6653, sendo esta última em versões mais recentes desde 18 de julho de 2013.

Além de atuar como uma API de comunicação entre controladores e *switches*, o protocolo OpenFlow permite que fabricantes de dispositivos possam oferecer uma interface aberta, bem definida e padronizada de programação do comportamento do dispositivo sem que tenham que explicitar detalhes da implementação interna e proprietária do mesmo. Para tanto, os fabricantes devem seguir um conjunto de especificações definido pela ONF [28, 29, 30, 31, 32, 33]. Essas especificações descrevem os métodos que os dispositivos devem implementar, quais parâmetros devem ser capazes de receber e como deverão responder, através de *endpoints* também padronizados. Descrevem também quais campos de cabeçalhos de outros protocolos de rede a versão em questão do OpenFlow é capaz de ler e operar sobre. A Tabela 2.1 associa as versões do OpenFlow desde 2009 às respectivas quantidades de campos de cabeçalhos legíveis. Foram omitidas as versões beta anteriores à 1.0 e as versões de correção, de acordo com a convenção do versionamento semântico [34].

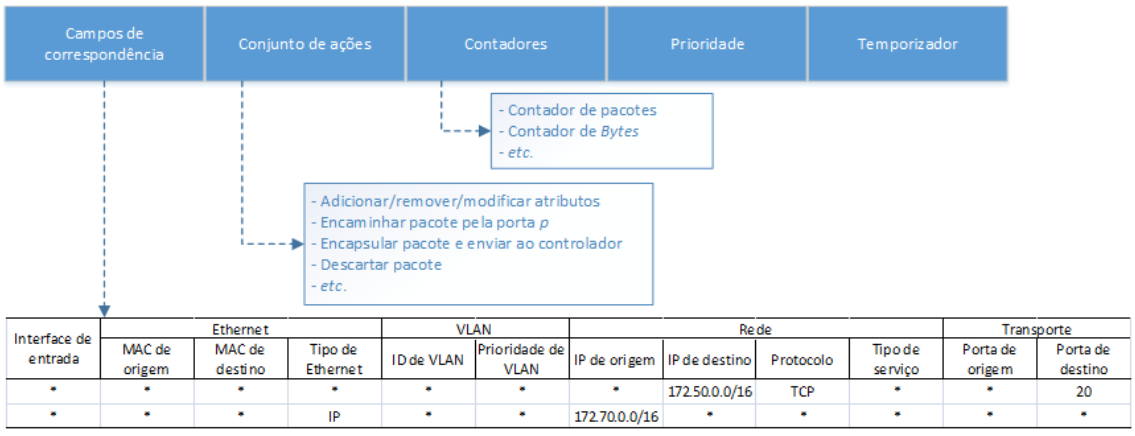
| Versão | Lançamento | # de campos |
|---|---|---|
| 1.0 | Dez 2009 [28] | 12 |
| 1.1 | Fev 2011 [29] | 15 |
| 1.2 | Dez 2011 [30] | 36 |
| 1.3 | Jun 2012 [31] | 40 |
| 1.4 | Out 2013 [32] | 41 |
| 1.5 | Dez 2014 [33] | 44 |

Tabela 2.1: Números de campos legíveis por versão do protocolo OpenFlow.

### 2.3.1 A tabela de fluxo

Um fluxo da camada de dados é uma sequência de pacotes que compartilham de um ou mais atributos de cabeçalho cujos valores são passíveis de serem correspondidos contra um valor-referência ou tomados como iguais dentro de um determinado domínio preestabelecido. Um exemplo de fluxo é o conjunto de todos os pacotes cuja porta lógica de destino é a 80, padrão do protocolo HTTP (*Hypertext Transfer Protocol*). No caso, 80 é o valor-referência e todos os pacotes direcionados para esse valor de porta, no escopo citado, configuram um mesmo fluxo. Como muitos fluxos distintos passam pelos *switches* da camada de dados e os mesmos devem ser capazes de repassar esses fluxos conforme a camada de controle determina, infere-se que é necessária a presença de uma estrutura especial que os identifique: a tabela de fluxo.

Cada *switch* mantém pelo menos uma tabela de fluxo própria nos moldes da Figura 2.2. Cada linha dessa tabela contempla uma entrada de fluxo de pacotes previsto, uma ação a ser tomada caso haja uma correspondência dessa entrada de fluxo contra os atributos do cabeçalho de um pacote sendo avaliado, o tempo de validade da entrada, o valor de prioridade da entrada em relação às demais e estatísticas sobre as circunstâncias em que ocorreram correspondências, como o número de pacotes correspondidos pela entrada. Para fins de padronização dos termos utilizados neste trabalho e na prova de conceito implementada, as entradas de fluxo supracitadas são chamadas "regras".




| Interface de entrada | Ethernet | | | VLAN | | Rede | | | | Transporte | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | MAC de origem | MAC de destino | Tipo de Ethernet | ID de VLAN | Prioridade de VLAN | IP de origem | IP de destino | Protocolo | Tipo de serviço | Porta de origem | Porta de destino |
| * | * | * | * | * | * | * | 172.50.0.0/16 | TCP | * | * | 20 |
| * | * | * | IP | * | * | 172.70.0.0/16 | * | * | * | * | * |

Figura 2.2: A tabela de fluxo do OpenFlow 1.0 e seus campos.

Quando um fluxo de pacotes chega a um *switch*, o mesmo avalia os atributos do cabeçalho dos pacotes contra a tabela de fluxo e (1) aplica ao fluxo uma ou mais ações previstas pela regra correspondida ou (2) encaminha o primeiro pacote do fluxo para o controlador para casos em que não há correspondência. Por exemplo,

dado que uma tabela de fluxo possua duas regras, uma regra *R1* para repassar pela porta de saída *A* do *switch* quaisquer pacotes com destino à porta TCP *X* e outra regra *R2* para descartar quaisquer pacotes com destino à porta TCP *Y*, todo fluxo com destino à porta TCP *X* terá uma correspondência com a regra *R1* e será repassado pela porta *A*, ao passo em que todo fluxo com destino à porta TCP *Y* corresponderá à regra *R2* e será descartado. Em ambos os casos, a aplicação das ações se dá imediatamente e sem a intervenção da camada de controle. Ainda no escopo deste exemplo, se um fluxo destinado à porta TCP *Z* chegar ao *switch*, sua avaliação não resultará em quaisquer correspondências. Consequentemente, o primeiro pacote do fluxo será encapsulado em uma mensagem OpenFlow *packet-in* e então será enviado ao controlador para tomada de decisão. O controlador pode então instruir ao *switch* para que futuros pacotes com destino à porta TCP *Z* sejam repassados pela porta de saída *B* ou que simplesmente sejam descartados. O *switch* terá uma nova regra *R3* prevendo um fluxo para a porta de destino TCP *Z* que repassa os pacotes pela porta de saída *B* caso correspondam à regra. Dependendo da lógica implementada, o controlador pode até mesmo instalar regras referentes a esse fluxo em outros *switches* do caminho, o que faria com que esse pacote não precisasse mais subir à camada de controle.

### 2.3.2 Abordagens reativa e proativa

A rede pode ser implementada com uma abordagem reativa, com uma abordagem proativa ou com um equilíbrio de ambas. Na abordagem reativa, a rede se comportará conforme o exemplo dado, com tabelas de fluxo que aumentam em quantidade de regras conforme novos fluxos não previstos chegam e são tratados pelo controlador. Esta abordagem tem a vantagem de não exigir que se tenha um conhecimento *a priori* do tráfego de pacotes, mas requer um algoritmo de tomada de decisão mais elaborado no controlador e implica em um atraso sempre que um novo fluxo é avaliado, uma vez que o primeiro pacote precisa ser tratado pelo controlador antes de uma nova regra ser criada. Na abordagem proativa, as tabelas de fluxo dos *switches* já são inicializadas com todas as regras para os tipos previstos de fluxo que passarão pela rede. Nesta abordagem, dificilmente o controlador será compelido a tratar um fluxo novo e geralmente há a intenção de descarte dos pacotes de fluxos não previstos. Esta abordagem tem a vantagem de minimizar a latência causada pela submissão de pacotes ao controlador, mas requer um conhecimento *a priori* do tipo de tráfego passante e criar regras de antemão pode se tornar muito difícil, dependendo do tamanho e do propósito da

rede. Há a possibilidade de um equilíbrio de ambas as abordagens, com determinados tipos de fluxo sendo previstos e novos tipos de fluxo sendo avaliados pelo controlador. É nessa abordagem mais equilibrada que este trabalho se alicerça, com regras de fluxos de controle da rede e regras derivadas de políticas do usuário sendo previamente instaladas nas tabelas de fluxo e regras oriundas da obtenção de anúncios de rotas sendo instaladas conforme a demanda.

### 2.3.3 Evolução do protocolo

A partir da versão 1.1, foram introduzidas novas funcionalidades no protocolo OpenFlow, como o processamento em *pipeline* dos pacotes com suporte a múltiplas tabelas de fluxo [29], uso de regra especial para tratamento de *table miss* [31] e a separação das tabelas do *pipeline* em tabelas de entrada e de saída [33].

Pacotes ingressantes em um *switch* OpenFlow de versão mais recente têm seus cabeçalhos confrontados contra regras em tabelas de fluxo de entrada que, conforme as correspondências, encaminham o fluxo para outras tabelas de entrada específicas de forma análoga a um comando "*go to*"ou para ser processado por uma tabela de grupos de ações, além dos comportamentos já citados de envio ao controlador e de descarte. A tabela de grupos de ações mantém um conjunto, identificado por um ID (*identifier* ou código de identificação), de ações comuns a serem aplicadas simultaneamente em um fluxo. Se uma tabela de entrada não tem correspondências com o fluxo de pacotes, é verificado se existe uma regra que trate casos de *miss*, ou ausência de correspondência na tabela. Caso exista, as ações previstas pela regra são adotadas, senão os pacotes do fluxo são descartados.

O fluxo processado pelas tabelas de entrada e pela tabela de grupos de ações é novamente processado, desta vez pelas tabelas de saída, caso o *switch* as tenha definidas. O processamento pelas tabelas de saída segue a mesma lógica das tabelas de entrada, exceto pela presença de uma tabela de grupos de ações. Ao final do processamento, o fluxo é encaminhado conforme a regra correspondida ou é descartado.

Nas versões mais recentes do protocolo, a maior quantidade de tabelas que cada *switch* mantém auxilia na organização das regras de fluxo e das ações a serem adotadas em grupo, mas pode aumentar a complexidade de programação do comportamento da rede e impactar na eficiência do encaminhamento ao buscar por correspondências em múltiplas tabelas.

O protocolo OpenFlow está em contínuo aprimoramento, com novas versões menores e de correção sendo lançadas anualmente. Levando em conta o número crescente de campos legíveis pelo OpenFlow conforme mostra a Tabela 2.1 e a maior complexidade de definição de regras em múltiplas tabelas por *switch*, há uma proposta de grandes mudanças estruturais do protocolo para a próxima versão maior, 2.0, a fim de delegar ao programador a implementação de algoritmos de *parsing* dos campos, tornando o protocolo mais escalável e flexível. A proposta é discutida na Subseção 3.4.6.

A ferramenta criada neste trabalho opera sobre a versão 1.0 do protocolo OpenFlow devido a limitações de alguns componentes que formam a pilha de aplicações. Na verdade, a versão do OpenFlow é ortogonal ao funcionamento da ferramenta, isto é, a ferramenta já está preparada para operar sobre versões mais recentes do protocolo, requerendo apenas que os componentes utilizados sejam atualizados por seus desenvolvedores para que sejam compatíveis.

## 2.4 Linguagens específicas de domínio

Uma linguagem específica de domínio, ou DSL, consiste em uma gramática de programação com escopo limitado a um problema ou domínio de atuação em particular [7]. O exemplo mais conhecido de DSL é o SQL (*Structured Query Language*), linguagem declarativa de consulta e manipulação de dados armazenados em bancos de dados [35]. DSLs são opostas às linguagens de propósito geral no que tange ao escopo de uso, pois as últimas podem ser utilizadas para a resolução de problemas de múltiplos domínios.

DSLs podem ser implementadas com um compilador próprio ou sobre uma linguagem de propósito geral, a qual um compilador intermediário, tradutor ou métodos específicos farão a conversão do código escrito na DSL para um código correspondente na linguagem de propósito geral.

Pode se tornar interessante o desenvolvimento de uma DSL quando uma determinada classe de problemas ocorre com frequência e a sua resolução pode ser implementada de maneira mais clara, automatizada e objetiva ao usuário do que seria por linguagens de propósito geral [7]. O foco do usuário é mantido na descrição em alto nível da solução do problema e não em como essa solução é implementada em detalhes de baixo nível, fornecendo assim uma camada de abstração.

Em relação ao domínio em que este trabalho está inserido, existem DSLs que fornecem abstração dos detalhes de funcionamento e configuração da rede física subjacente. Algumas dessas DSLs são tratadas nas Seções 2.5 e 3.4. Uma das contribuições deste trabalho é uma DSL que é implementada sobre o Ruby, linguagem de propósito geral flexível e que facilita a metaprogramação com a passagem de blocos de código como parâmetro dos métodos definidos. O código escrito usando a DSL proposta neste trabalho é avaliado pelo interpretador do Ruby nas etapas de execução da arquitetura.

## 2.5 O *framework* Merlin

Merlin [10, 12, 11] é um *framework* de gerenciamento de redes de código aberto[1], implementado na linguagem OCaml[2], que permite que um administrador possa expressar as políticas internas da rede em uma linguagem declarativa e de alto nível de abstração baseada em predicados lógicos e expressões regulares. Essa linguagem é uma DSL com uma gramática própria do Merlin. O administrador pode abstrair de como serão geradas as regras OpenFlow a serem instaladas nas tabelas de fluxos e quais serão as ações a serem adotadas para correspondências, delegando ao Merlin a responsabilidade de gerar um conjunto de regras aplicáveis ao cenário de uso da rede. Para tanto, o Merlin opera com um componente chamado Gurobi[3], um *software* de resolução de problemas de otimização matemática. Apesar de ser proprietário, o Gurobi possui licença acadêmica gratuita e o Merlin opera usando as funcionalidades contempladas por essa licença.

Conforme supracitado, a gramática própria do Merlin permite a configuração do comportamento da rede através de políticas descritas na linguagem declarativa do *framework*, basicamente um conjunto de predicados lógicos associados a expressões regulares de caminho de fluxo. Os predicados lógicos são conjuntos de pares chave-valor cujas chaves são atributos de cabeçalho dos pacotes compatíveis com o protocolo OpenFlow e os valores estão nos domínios dos respectivos atributos. As expressões regulares denotam o caminho que o fluxo de pacotes correspondido pelo predicado lógico deverá seguir na topologia da rede. O exemplo do Código 2.1 demonstra como se apresenta uma linha de política de rede descrita na linguagem declarativa do Merlin.

[1]Disponível em `https://github.com/merlin-lang/merlin`.
[2]Disponível em `https://ocaml.org/`.
[3]Disponível em `http://www.gurobi.com/`.

```
1 ipProto = 0x06 and tcpDstPort = 80 -> .* s3 .*;
```

Código 2.1: Linha de código de política do Merlin.

No exemplo, o predicado lógico `ipProto = 0x06 and tcpDstPort = 80` corresponde a um fluxo de pacotes do protocolo TCP (código 6 em hexadecimal) com destino à porta padrão do protocolo HTTP (porta 80). Todo fluxo com essas características deve ser encaminhado pela rede subjacente de forma que sempre passe pelo *switch s3*, não importando por quais *switches* o fluxo de pacotes trafegue antes ou depois, conforme denota o trecho `.* s3 .*` da linha de exemplo. Esse trecho de código, quando compilado pelo Merlin, dará origem a várias regras OpenFlow a serem instaladas nas tabelas de fluxos dos *switches*, todas com predicados lógicos que refletem o trecho de código e com ações de repasse de pacotes pelas saídas que levem ao *switch s3* direta ou indiretamente.

O administrador da rede submete ao *framework* um arquivo descrevendo as políticas de encaminhamento interno na linguagem declarativa do Merlin e um segundo arquivo contendo a topologia da rede na linguagem DOT[4], usada para a descrição de grafos em texto puro. O Merlin processa esses arquivos, validando se as políticas descritas são compatíveis com a topologia fornecida e compilando o código na linguagem própria em regras OpenFlow separadas por *switch*. O resultado é impresso em formato textual no terminal que invocou o *framework* ou submetido para o Frenetic, projeto relacionado descrito na Seção 3.4.

O Merlin foi projetado inicialmente como uma solução aplicável a redes institucionais, onde se tem conhecimento não apenas dos *switches*, mas também dos *hosts*, isto é, dos computadores e dispositivos com endereços IP definidos na rede. Este trabalho expande o escopo de utilização do Merlin para o cenário interdomínio da *Internet*, tratando tanto as políticas antes de serem submetidas ao *framework* quanto as regras geradas depois da compilação.

A adaptação do Merlin neste trabalho é uma manipulação dos seus dados de entrada e de saída, sem intervenção em seu código-fonte. Alterações no código-fonte poderiam ter sido feitas, porém envolveriam uma grande mudança lógica da solução, uma vez que a mesma é projetada para redes institucionais. Além disso, a linguagem OCaml possui uma curva de aprendizado inerente. Ambos os fatores culminariam em um tempo de implementação substancial e portanto

[4]Disponível em `http://www.graphviz.org/content/dot-language`.

inviável, já que a manipulação da entrada e da saída de dados do Merlin já é suficiente para o escopo almejado.

A prova de conceito deste trabalho invoca o Merlin em outra máquina virtual através de uma sessão SSH, submetendo os arquivos de descrição de política e de topologia e fazendo o *parsing* das regras OpenFlow textuais geradas e impressas no terminal. Posteriormente, as regras são ajustadas conforme as opções passadas pelo usuário e estruturadas para serem exportadas e instaladas pelo RouteFlow, descrito na seção seguinte.

## 2.6 A plataforma RouteFlow

RouteFlow [8, 9] é uma plataforma de roteamento IP (*Internet Protocol*) de código aberto[5] com arquitetura implementada sobre uma rede de *switches* OpenFlow subjacente e uma camada virtual que executa instâncias de *softwares* de roteamento com protocolos livres. O controlador faz o intermédio entre as camadas virtual e de dados, lendo as rotas geradas nas tabelas de roteamento Linux pelas instâncias de roteamento e traduzindo-as para regras OpenFlow nas tabelas de fluxo dos *switches*. Isso é possível porque as instâncias de roteamento na camada virtual espelham a mesma topologia que os *switches* na camada de dados, estando cada instância logicamente associada a um *switch* OpenFlow. A Figura 2.3 ilustra a arquitetura do RouteFlow. Uma observação é que a camada virtual referida no trabalho do RouteFlow é na realidade a camada de aplicação da pilha de camadas das redes definidas por *software*.

Assim como a camada de controle, a camada virtual do RouteFlow também é executada em *hardware* genérico, não proprietário e de propósito geral. Essa camada é formada por contêineres LXC (*Linux Containers*)[6] que compartilham de um mesmo *kernel* Linux mesmo sendo interfaces de espaço de usuário distintas. Cada contêiner roda uma instância de *software* de roteamento com suporte aos protocolos BGP, OSPF, RIP, entre outros, e as instâncias mantêm as tabelas de roteamento dos respectivos contêineres atualizadas. Em uma visão mais atômica, os contêineres se comportam como roteadores virtuais interligados conforme a topologia dos *switches* da camada de dados e são capazes de se comunicar com roteadores legados e instâncias de roteamento com enlaces para os *switches* associados.

[5] Disponível em `https://github.com/routeflow/RouteFlow`.
[6] Disponível em `https://linuxcontainers.org/`.

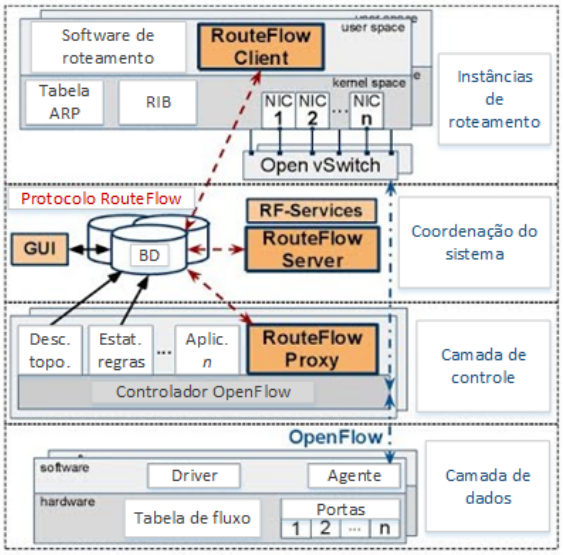



Figura 2.3: Arquitetura do RouteFlow.

A nível experimental, a camada de dados do RouteFlow é gerenciada pelo Mininet[7], um emulador de redes virtuais que fornece suporte para a criação de *testbeds* locais contendo *hosts*, *switches*, enlaces e controladores, todos elementos virtuais. No caso do RouteFlow, a rede de *switches* do Mininet é inicializada com um arquivo de configuração da topologia refletindo a mesma organização dos roteadores da camada virtual e com o controlador definido como remoto, apontando para o endereço IP da máquina que executa o RouteFlow. Quando em produção, o RouteFlow é implantado com *switches* OpenFlow reais em vez de elementos virtualizados pelo Mininet.

A arquitetura do RouteFlow possui três componentes principais: *RFClient*, *RFServer* e *RFProxy*. O RFClient é um código de segundo plano, ou *daemon*, que executa de forma distribuída em cada um dos contêineres e sua função é notificar o RFServer de quaisquer eventos de alteração nas tabelas de roteamento do contêiner. O RFServer é uma aplicação que gerencia os diversos *daemons* RFClient, mapeando os contêineres e interfaces de rede aos respectivos *switches* e portas, repassando à camada virtual os pacotes de protocolos de roteamento oriundos da camada de dados e instruindo o RFProxy a instalar regras OpenFlow baseadas nas decisões tomadas na camada virtual. O RFProxy é uma extensão acoplada ao controlador que recebe as instruções do RFServer e notifica o mesmo sobre quaisquer eventos da camada de dados. Essas instruções são mensagens *RouteMod*, objetos que carregam a instrução de adição, de modificação ou de remoção de regras OpenFlow das tabelas de fluxo dos *switches*.

[7]Disponível em `http://mininet.org/`.

Como cada módulo do RouteFlow é um processo diferente, a plataforma implementa uma IPC (comunicação inter-processo, do inglês *Inter-Process Communication*) usando um banco de dados não relacional MongoDB[8]. Essa IPC emulada pelo banco é separada por canais de comunicação entre o RFServer e as instâncias de RFClient e entre o RFServer e o RFProxy. Em âmbito de banco, os canais são coleções (análogo a tabelas em bancos relacionais) e as mensagens enfileiradas são documentos (análogo a tuplas). As mensagens enfileiradas são eventualmente consumidas pelos processos-alvo.

A plataforma RouteFlow oferece a vantagem de tornar o encaminhamento dos fluxos de pacotes mais eficiente, uma vez que apenas os primeiros pacotes de cada fluxo serão submetidos à camada de controle e posteriormente à camada virtual para tomada de decisão pelos roteadores virtuais. Dessa forma, todos os demais pacotes de um mesmo fluxo serão imediatamente repassados pelos *switches* com o mínimo de latência média possível, enquanto as respectivas regras nas tabelas de fluxo forem válidas. Numa rede tradicional com camadas de controle e de dados acopladas nos roteadores, cada pacote de um mesmo fluxo teria que passar por uma avaliação pelos algoritmos de roteamento a cada salto, impactando em uma latência média de repasse maior.

O RFServer é o núcleo do RouteFlow, onde ocorre de fato toda a orquestração da rede. Para a realização deste trabalho, foi adicionado um novo módulo, *RFHavox*, que faz uma requisição HTTP a uma API do Havox após a inicialização do RouteFlow. A API responde com uma mensagem JSON (*JavaScript Object Notation*) contendo informações de novas regras OpenFlow a serem instaladas na camada de dados. As regras vindas do Havox possuem prioridade maior do que as já administradas pelo RouteFlow, visto que são geradas pelo Merlin após o processamento das políticas de rede descritas pelo administrador.

Além do RFHavox, o código do RouteFlow foi atualizado com a lógica de manipulação de VLAN (*Virtual Local Area Network*), de endereço IP de origem e de prefixos de rede com máscaras variadas, atributos necessários para que o operador de rede tenha ainda mais controle sobre o tráfego passante através de políticas de encaminhamento. Mais detalhes sobre essas atualizações e sobre a integração com o RFHavox serão tratados nos capítulos 3 e 4.

[8]Disponível em `https://www.mongodb.com/`.

## 2.7 Síntese

Este capítulo abordou todo o conhecimento teórico necessário para o entendimento deste trabalho, desde conceitos até protocolos, discutindo brevemente como cada item se conecta com o que foi realizado na prática.

Foram também abordados o RouteFlow e o Merlin, duas soluções de relevância acadêmica que, além de serem trabalhos relacionados, são também componentes da arquitetura proposta neste trabalho. No caso do primeiro, foi realizado um avanço no seu código-fonte, de forma que o permita se comunicar com o núcleo da arquitetura e gerenciar mais campos OpenFlow. No caso do último, as manipulações foram sobre seus dados de entrada e de saída.

# 3. Proposta

Este capítulo explica a proposta em uma abordagem *top down*, iniciando pela descrição macroscópica da arquitetura Havox e da sua pilha de aplicações e aprofundando em detalhes de funcionamento da DSL proposta e da integração entre os componentes utilizados. Ao fim, elenca alguns trabalhos que guardam relação com a proposta deste.

## 3.1 Visão geral

Consoante com o que foi mencionado no Capítulo 1, o objetivo deste trabalho é fornecer um meio de orquestrar o tráfego de pacotes dentro de um domínio administrativo usando uma linguagem de configuração simples, com foco na distribuição dos fluxos passantes por múltiplos roteadores de borda que dão saída para prefixos IP conhecidos, baseado nos campos de protocolos de rede legíveis pelo protocolo OpenFlow.

Para alcançar o objetivo, este trabalho apresenta o Havox, uma arquitetura para orquestração de tráfego em redes OpenFlow cujo núcleo é uma biblioteca escrita na linguagem Ruby e que engloba uma API *web*, o *framework* Merlin e a plataforma RouteFlow modificada para se comunicar remotamente com a API. O processo de orquestração se dá com a escrita de diretivas, que são instruções associadas, cada qual, a um *switch* de saída do domínio e a um conjunto de campos OpenFlow para correspondência com atributos dos cabeçalhos dos pacotes. As diretivas são especificadas com uma DSL própria e que opera sobre a linguagem Ruby.

A arquitetura é capaz de moldar a forma como o tráfego de pacotes é encaminhado dentro de um domínio administrativo sob a ótica de redes definidas por

*software*. O tráfego é balanceado entre diferentes saídas do domínio de acordo com os atributos de cabeçalho dos pacotes passantes. Isso é possível porque as rotas BGP geradas por instâncias de roteamento Quagga vinculadas com outros ASes são lidas e estruturadas de forma que seja feita uma mesclagem em memória das rotas obtidas de todas as instâncias Quagga. Essas rotas são mantidas em uma lista de redes alcançáveis. Uma vez que se tenha conhecimento centralizado de que uma saída do domínio leva a um determinado prefixo de rede e que se tenha regras OpenFlow definidas para um fluxo de pacotes, encaminha-se o fluxo correspondido pela saída desejada. Os pacotes que não correspondem a nenhum fluxo previsto seguem a lógica de encaminhamento convencional mantida pelo RouteFlow.

Este trabalho discorre usando alguns termos diferentes que se referem à mesma posição na arquitetura proposta, mas em camadas distintas. Um *switch* na camada de dados está associado a uma instância de roteamento na camada virtual dentro do domínio administrativo. De igual maneira, um *host* na camada de dados está associado a uma instância de roteamento fora do domínio administrativo. Em suma, *switches* representam roteadores dentro do AS e *hosts* representam roteadores de ASes vizinhos ou remotos. A Figura 3.1 sintetiza essa associação, que será mais aprofundada no Capítulo 5.

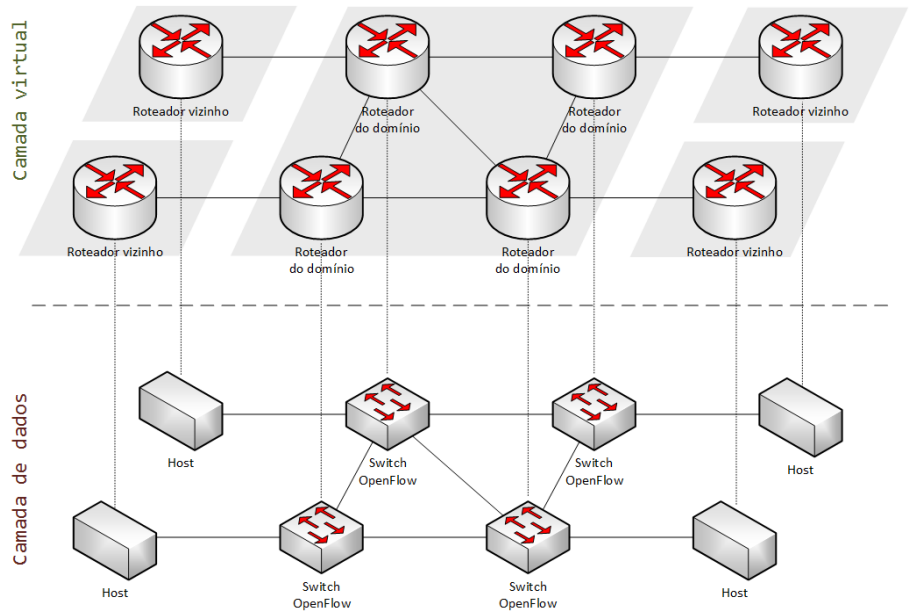



Figura 3.1: Associação entre a camada de dados e a camada virtual.

O termo *host* foi usado neste trabalho para designar um roteador de borda de um AS vizinho ou remoto para fins de rápida associação quando o foco explicativo estiver na camada de dados, onde opera o Mininet, usado no cenário de testes

do Capítulo 5. *Host* também é o termo usado pelo Merlin para se referir aos nós da rede que não são *switches* OpenFlow e que, portanto, somente enviam e recebem pacotes, sendo análogos aos computadores e demais dispositivos de rede. A arquitetura proposta abstrai de como são implementados os roteadores de borda dos ASes vizinhos e remotos, o que justifica a preferência por tratar como *switch* apenas os dispositivos internos que mantêm regras OpenFlow e são análogos aos roteadores do domínio e tratar como *host* quaisquer outros dispositivos de vizinhos análogos a roteadores, quando a camada de dados estiver sendo abordada.

## 3.2 Arquitetura

A arquitetura é desenhada como mostra a Figura 3.2. O cerne da arquitetura é a biblioteca homônima Havox, que é invocada pela aplicação *web* que recebe requisições HTTP com parâmetros bem definidos e responde com mensagens em formato padronizado JSON. Essa aplicação *web* é a API que faz o intermédio entre a biblioteca e seus consumidores. No caso deste trabalho, o consumidor da API é o RouteFlow, que envia os arquivos de topologia da rede e de diretivas na DSL do Havox e é respondido com as regras OpenFlow que devem ser instaladas nos seus *switches* administrados com prioridade superior às demais regras.

Na Figura 3.2, a sequência de passos se inicia com (1) o módulo RFServer do RouteFlow invocando o módulo RFHavox. Este (2) envia uma requisição POST à API do Havox, contendo os arquivos de diretivas e de topologia, bem como parâmetros de configuração. A API, ao receber a requisição, (3) invoca a biblioteca Havox e repassa a esta os parâmetros. A biblioteca realiza a transcompilação das diretivas em políticas Merlin. Feito isso, (4) abre uma conexão SSH com a máquina que executa o Merlin, enviando o arquivo de políticas transcompilado e o arquivo de topologias. Nessa mesma conexão, (5) são coletadas as regras primitivas da saída padrão gerada pelo processo de compilação do Merlin. As regras primitivas são estruturadas e tratadas pela biblioteca, com base nos parâmetros de configuração especificados, e em seguida são (6) devolvidas à API, que (7) encapsula as regras em mensagem JSON e responde ao módulo RFHavox com a mensagem. O módulo RFHavox extrai as regras especiais da mensagem JSON e as enfileira para processamento pelos demais módulos do RouteFlow. Eventualmente, as regras (9) são instaladas nos *switches* OpenFlow da camada de dados.

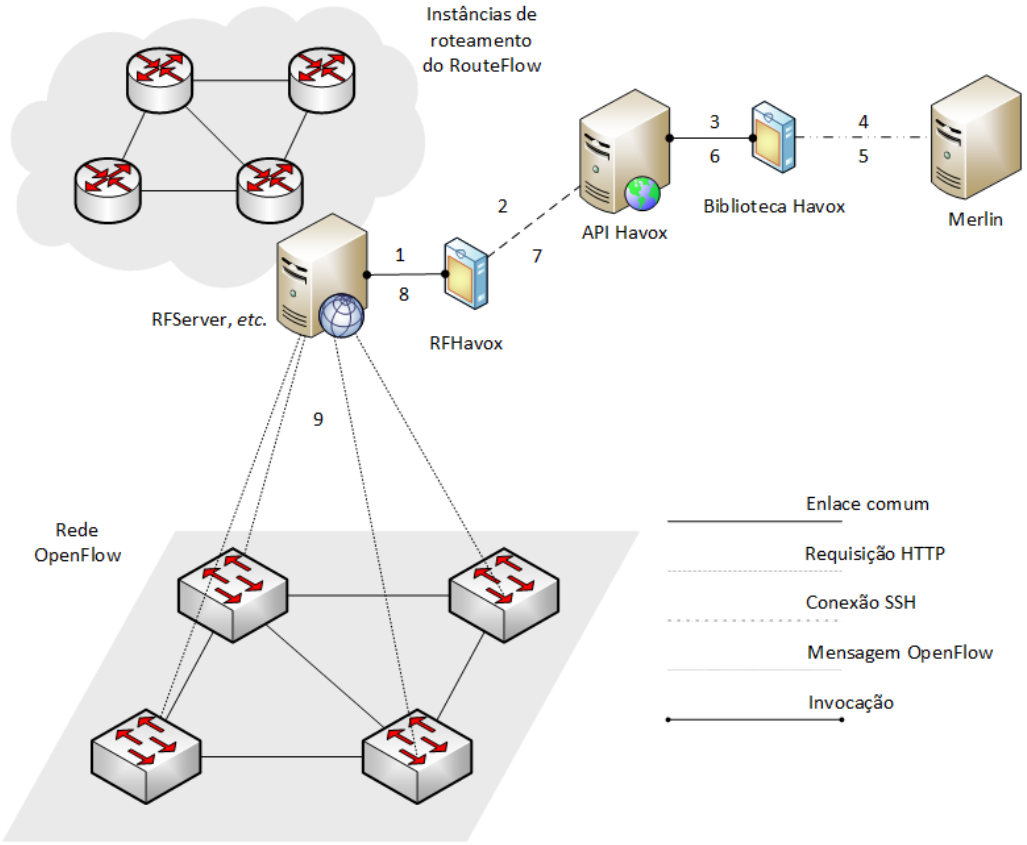



Figura 3.2: A arquitetura Havox.

A topologia de rede enviada pela requisição do módulo RFHavox segue o padrão de descrição de grafos na linguagem DOT e marca quais nós são *switches* e quais são *hosts*. Já o arquivo de diretivas na DSL do Havox é um conjunto de instruções de saída de tráfego do AS, cada qual com o seu conjunto de campos de cabeçalho OpenFlow que correspondem a uma determinada classe de tráfego e um *switch* de borda que dá saída para o tráfego correspondente. Ambos os arquivos são criados pelo usuário e enviados pelo RFHavox à API a partir do sistema de arquivos do RouteFlow.

A partir dos arquivos supracitados, o Havox renderiza um arquivo de políticas na DSL do Merlin contendo as correspondências de cada classe de tráfego e submete ao mesmo. O Merlin processa essas políticas e responde com regras de fluxo textuais impressas em saída padrão. Essas regras primitivas, como serão chamadas deste ponto em diante, são lidas, processadas e estruturadas pelo Havox. Uma vez estruturadas, as regras são traduzidas para a sintaxe do RouteFlow e encapsuladas em uma mensagem JSON de resposta da API. Nesse estágio, as regras são consideradas especiais, pois são oriundas das diretivas definidas pelo usuário e por isso terão prioridade superior às demais regras

básicas geradas pelo RouteFlow.

Uma vez que as regras OpenFlow geradas pelo Havox são instaladas nas tabelas de fluxo pelo RouteFlow com prioridade superior, todo tráfego que corresponda a essas regras especiais seguirá a lógica de encaminhamento interno estabelecido e será guiado até o *switch* de saída definido pelo operador, de acordo com as rotas conhecidas. O restante das classes de tráfego não correspondidas respeitará a lógica de encaminhamento convencional das regras básicas do RouteFlow. A Tabela 3.1 define a nomenclatura que será adotada neste trabalho na referência das regras OpenFlow em questão.

| Tipo | Origem | Características da regra |
|---|---|---|
| Primitiva | Merlin | Textual, não estruturada e obtida da leitura em saída padrão (*stdout*). |
| Básica | RouteFlow | Estruturada e derivada das decisões de roteamento da camada virtual. |
| Especial | Havox | Estruturada, tratada e que tem prioridade sobre as básicas por vir do usuário. |

Tabela 3.1: Nomenclatura adotada para os tipos das regras OpenFlow.

### 3.3 A biblioteca

A biblioteca Havox fornece uma DSL de configuração do encaminhamento dentro de um domínio administrativo, focando especialmente em definir como será a saída dos fluxos de pacotes passantes. O objetivo da proposta que deu origem à biblioteca, como prova de conceito, é mostrar que é possível encaminhar pacotes de forma mais flexível, granular e alinhada com os SLAs estabelecidos pelo AS.

Além do objetivo central, a proposta também tem o intuito de mostrar que é possível expandir continuamente a biblioteca com novas funcionalidades de acordo com as necessidades da rede, bem como é possível desacoplar e substituir componentes da pilha de aplicações até então necessários, como o Merlin e o RouteFlow, à medida em que surjam outros com mais vantagens de uso ou caso a própria biblioteca ou a arquitetura Havox como um todo se torne autossuficiente ao ponto de desempenhar as funções desses componentes. A arquitetura foi projetada para ser modular e facilmente adaptável conforme cada realidade de uso [36].

```
1 topology 'example.dot'
2
3 exit(:s5) { source_port 20 }
4 exit(:s6) { source_ip '200.20.207.31' }
5 exit(:s7) {
6   destination_port 80
7   destination_ip '200.156.151.09'
8 }
```

Código 3.1: Exemplo de arquivo de diretivas do Havox.

### 3.3.1 A linguagem

Um arquivo escrito na DSL da biblioteca contém um conjunto de diretivas, cada qual contendo um *switch* de saída e um bloco de palavras-chave correspondentes a campos OpenFlow conhecidos, com respectivos valores que serão processados para gerar as regras de encaminhamento que serão instaladas nos *switches*. Há também a diretiva de referência ao arquivo de topologia que descreve a disposição dos dispositivos na camada de dados. O Código 3.1 contém exemplos de definições de diretivas e seus campos.

No Código 3.1, a diretiva `topology` referencia o nome ou caminho do arquivo de topologia que, se submetido junto com o arquivo de diretivas, estará no mesmo diretório. Em seguida, são definidas três diretivas de saída de tráfego do AS (diretiva `exit`) baseado nos campos OpenFlow especificados dentro de cada bloco. A primeira diretiva corresponde a todos os pacotes cuja porta TCP de origem seja a 20, de transmissão de dados do FTP (*File Transfer Protocol*), e instrui a saída desses pacotes pelo *switch* de borda *s5*. A segunda diretiva corresponde aos pacotes oriundos do endereço IP 200.20.207.31, com saída pelo *switch s6*. Ambas possuem blocos contendo apenas um campo. A terceira e última diretiva possui um bloco de dois campos e corresponde aos pacotes de tráfego *web* com destino ao endereço IP 200.156.151.09, instruindo saída pelo *switch s7*.

A DSL de configuração da arquitetura Havox não possui um compilador próprio, como no caso do Merlin. Em vez disso, o código escrito na DSL é avaliado pelo interpretador da linguagem de propósito geral Ruby. As diretivas de configuração são métodos Ruby invocados tendo o nome da topologia, os *switches* e os blocos de campos OpenFlow como argumentos, quando aplicáveis. Por exemplo, no Código 3.1, a diretiva `topology` invoca o método de mesmo nome e passa como argumento uma *string* correspondente ao caminho do arquivo de topologia. Por sua vez, a diretiva `exit` invoca o método homônimo e passa a

```
1 foreach (s, d): cross({ h2; h3; h4 }, { h1 })
2   tcpSrcPort = 20 -> .* s5;
3
4 foreach (s, d): cross({ h1; h3; h4 }, { h2 })
5   ipSrc = 200.20.207.31 -> .* s6;
6
7 foreach (s, d): cross({ h1; h3; h4 }, { h2 })
8   tcpSrcPort = 80 and ipDst = 200.156.151.09 -> .* s7;
```

Código 3.2: Código Merlin transcompilado a partir do código Havox.

este um argumento de tipo símbolo[1] ou de tipo *string* com o nome do *switch* e outro argumento contendo um bloco de código, que corresponde aos campos OpenFlow.

O argumento de bloco de código com campos OpenFlow e valores também segue uma lógica similar àquela citada anteriormente, com a diferença apenas na chamada aos respectivos métodos. Quando uma linha do bloco contendo um campo e seu valor é avaliada, a invocação do método de mesmo nome resulta em exceção porque o mesmo não está de fato implementado. Em vez disso, no tratamento da exceção o nome do método é capturado junto com o seu argumento (o valor do campo) e ambos são armazenados como par chave-valor em um conjunto vinculado à diretiva para processamento futuro. Essa técnica é possível em razão do suporte à *metaprogramação*[2] que a linguagem Ruby dispõe.

O arquivo contendo as diretivas e seus respectivos blocos de campos e valores é lido pela biblioteca Havox, a qual transformará cada diretiva de saída de tráfego em trechos de políticas na DSL do Merlin. Essa etapa de transformar o código na DSL do Havox em código na DSL do Merlin é chamada de *transcompilação*[3]. No exemplo do Código 3.1, a biblioteca transcompila as diretivas para o Código 3.2, legível pelo compilador do Merlin.

Confrontar os Códigos 3.1 e 3.2 permite observar que automaticamente foram inferidos, durante o processo de transcompilação, os vizinhos por onde o tráfego entrará na rede (*h2*, *h3* e *h4*) e o vizinho por onde ele deverá sair (*h1*), junto

[1] Em Ruby, o tipo símbolo é uma *string* especial, não mutável e prefixada com um caractere de dois pontos. Usualmente um símbolo é usado para definir um identificador interno do programa. Múltiplas instâncias de *strings* idênticas ocupam múltiplas posições de memória, enquanto múltiplas instâncias idênticas de símbolos ocupam apenas uma posição.

[2] Metaprogramação [37] é uma técnica de programação que permite que programas sejam usados como parâmetro de entrada para outros programas, de forma que os mesmos possam modificar a si mesmos durante a execução.

[3] Transcompilação, ou compilação fonte-a-fonte [38], é o processo de transformar o código-fonte escrito em uma linguagem *A* para código-fonte em uma linguagem *B*.

com o *switch* de acesso deste (*s5*), no caso da primeira diretiva transcompilada. As palavras reservadas `foreach` e `cross` nas linhas do iterador indicam que para cada tráfego com origem *s* e destino *d*, *s* podendo ser *h2*, *h3* ou *h4* e *d* sendo *h1*, deve ser aplicada a regra descrita pelo código aninhado. O processo é similar quanto as demais diretivas. Por uma limitação da versão atual do Merlin, é necessária a repetição das linhas do iterador para cada regra individual.

O Código 3.2 transcompilado é submetido ao Merlin, que compilará esse código e gerará as regras OpenFlow primitivas que serão lidas, estruturadas e exportadas para o RouteFlow pelo Havox. O processo de inferência dos vizinhos citado anteriormente, bem como o tratamento das regras primitivas em regras estruturadas, serão explicados mais adiante neste capítulo.

A justificativa para se executar essa etapa de transcompilação de código é prevenir o operador de ter que definir os *hosts* de entrada e de saída durante a criação das regras OpenFlow, bem como não obrigá-lo a conhecer a sintaxe do Merlin. A nomenclatura dos *hosts*, suas conexões com os *switches* e as palavras reservadas de iteração do Merlin são detalhes de mais baixo nível que podem ser abstraídos em prol do foco no que de fato importa, que é a orquestração do tráfego. Além disso, caso o Merlin sofra atualizações ou até mesmo seja substituído por outro componente de função similar, as mudanças serão transparentes para o operador.

Para as versões utilizadas do Merlin e do OpenFlow, a versão de prova de conceito da biblioteca Havox suporta as sintaxes de campos previstos na Tabela 3.2. Conforme a biblioteca for sendo expandida, novos campos serão adicionados, bem como novas diretivas além de `exit` e de `topology`. Os planos de expansão serão abordados na seção de trabalhos futuros no capítulo final deste trabalho.

Os valores dos campos da Tabela 3.2 também podem ser definidos de maneira dinâmica. Como a DSL do Havox é implementada sobre a linguagem Ruby, que tem suporte a metaprogramação, é possível especificar os valores como sendo funções anônimas que serão avaliadas em tempo de execução durante a transcompilação de código. O Código 3.3 mostra um exemplo de uso de função anônima como valor para um dos campos de diretiva.

O campo `destination_port` do Código 3.3 recebe como valor uma função anônima, ou *lambda*, cujo código é o trecho "`-> { [80, 443].sample }.call`". Quando a função anônima for avaliada e executada com a chamada do método

| Campo | Exemplo | Descrição |
|---|---|---|
| `destination_ip` (*str*) | '200.156.151.09' | Endereço IP de destino. |
| `destination_mac` (*str*) | 'cc:c3:af:08:05:16' | Endereço MAC de destino. |
| `destination_port` (*int*) | 443 | Porta de destino. |
| `ethernet_type` (*int*) | 2048 ou 0x0800 | Código do protocolo de rede encapsulado. |
| `in_port` (*int*) | 1 | Interface de entrada do pacote no *switch*. |
| `ip_protocol` (*int/str*) | 17, 0x11 ou 'udp' | Código do protocolo de transporte encapsulado. |
| `source_ip` (*str*) | '200.20.207.31' | Endereço IP de origem. |
| `source_mac` (*str*) | 'cc:c3:af:11:05:88' | Endereço MAC de origem. |
| `source_port` (*int*) | 80 | Porta de origem. |
| `vlan_id` (*int*) | 29 | ID da VLAN. |
| `vlan_priority` (*int*) | 8 | Valor de prioridade da VLAN. |

Tabela 3.2: Campos suportados pela biblioteca Havox.

```
1 exit(:s8) {
2   destination_port -> { [80, 443].sample }.call
3   destination_ip '207.31.96.6'
4 }
```

Código 3.3: Uso de função anônima como valor de campo em diretiva.

`#call`, ela avaliará o vetor de inteiros composto pelos valores 80 e 443 e retornará um deles aleatoriamente, com o método `#sample`. O valor retornado pela função anônima será usado para o campo `destination_port`.

O exemplo mostra que é possível definir valores dinâmicos para quaisquer dos campos suportados por meio do uso de funções anônimas. Esse recurso permite que o programador defina dinamicamente o valor dos campos, obtendo os mesmos a partir de cálculos mais complexos, de outros serviços locais ou remotos ou de variáveis de ambiente, não importando o código da função anônima, desde que a mesma seja implementada na linguagem Ruby e que seu retorno seja um valor compatível com o campo em questão.

### 3.3.2 Dependências de configuração

Além das dependências de componentes da pilha de aplicações que a biblioteca e a arquitetura como um todo possuem, é necessário que algumas configurações estejam definidas e funcionais.

Uma das dependências é a necessidade de que o protocolo OSPF esteja em

operação pela plataforma RouteFlow. Apesar de não consumir rotas OSPF para estabelecer o encaminhamento, a biblioteca depende dessas rotas para inferir as associações entre os *switches* da camada de dados e as instâncias de roteamento da camada virtual. Essa decisão de projeto foi tomada para esta prova de conceito com o intuito de minimizar a escrita de código de configuração, de forma que o usuário não tenha que fornecer manualmente as associações entre *switches* e instâncias de roteamento.

Para inferir as associações supracitadas, em primeiro lugar é feito um *parsing* do arquivo de topologia, que é escrito na linguagem DOT de descrição de grafos e descreve a organização da camada de dados, conforme o Código 3.4. O *parsing* permite obter quais nós são *switches*, quais são *hosts* e as arestas entre eles. São lidos também os endereços IP de cada *switch*, que devem obrigatoriamente constar na descrição do grafo de conectividade, bem como quaisquer outros atributos específicos de cada nó. O endereço IP deve ser o da interface de enlace com o *host*. Após o *parsing*, o sistema já terá toda a disposição da rede em memória.

```
1 digraph g1 {
2   h1 [type = host, mac = "00:00:00:00:00:01", ip = "10.0.0.100"];
3   h2 [type = host, mac = "00:00:00:00:00:02", ip = "10.0.0.200"];
4
5   s1 [type = switch, ip = "10.0.0.1", id = 1];
6
7   s1 -> h1 [src_port = 1, dst_port = 1, cost = 1];
8   h1 -> s1 [src_port = 1, dst_port = 1, cost = 1];
9
10  s1 -> h2 [src_port = 2, dst_port = 1, cost = 1];
11  h2 -> s1 [src_port = 1, dst_port = 2, cost = 1];
12 }
```

Código 3.4: Exemplo de grafo de conectividade na linguagem DOT.

No exemplo de descrição de topologia do Código 3.4, dois *hosts h1* e *h2* possuem enlaces com o *switch s1*. Os atributos de cada nó são listados dentro dos pares de colchetes, incluindo seus endereços IP, e são armazenados em memória. No momento da inferência de associação entre *switches* e instâncias de roteamento, apenas o endereço IP do *switch* é importante.

A biblioteca mantém também uma estrutura análoga à RIB, chamada neste trabalho de RIB global, com todas as rotas obtidas a partir do RouteFlow. Uma vez obtida a lista de *switches* e seus endereços IP das interfaces com os *hosts*, o

processo de inferência da associação entre *switches* e instâncias de roteamento segue como exibido pela Figura 3.3.

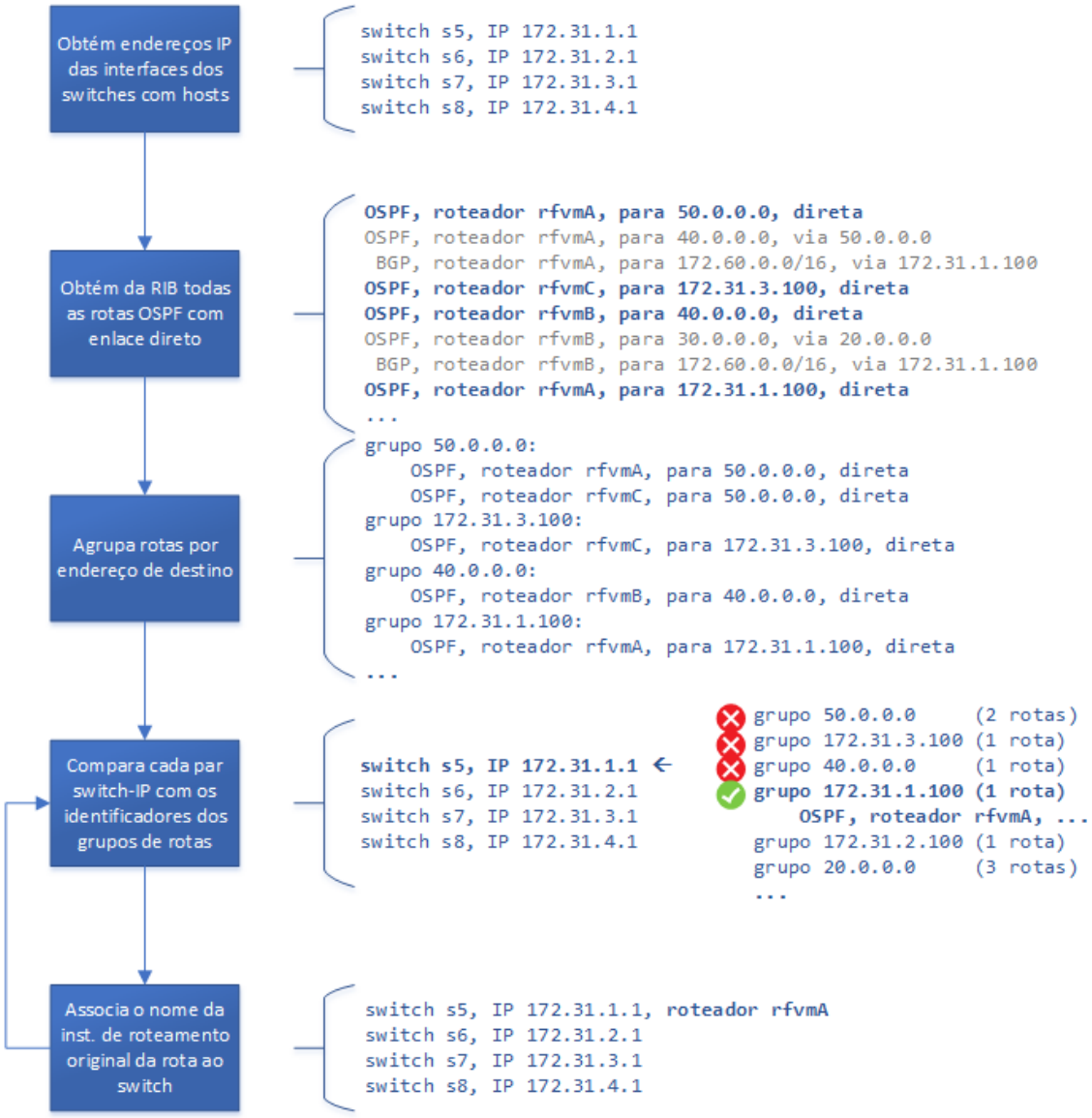



Figura 3.3: Inferência das associações entre *switches* e instâncias de roteamento.

É importante salientar que, no quarto passo, a comparação do endereço IP do *switch* com o endereço IP que identifica o grupo em avaliação é feita verificando se ambos pertencem à mesma rede. No exemplo da Figura 3.3, os endereços IP 172.31.1.1 do *switch s5* e 172.31.1.100 que identifica um dos grupos pertencem à mesma rede 172.31.1.0/24. Como o grupo contém uma única rota originada da instância de roteamento *rfvmA*, infere-se que *s5* é representado por *rfvmA* na camada virtual do RouteFlow.

Outra dependência de configuração da arquitetura é que as instâncias de roteamento devem ser capazes de operar o protocolo BGP. As rotas obtidas de

anúncios BGP são essenciais para que as diretivas que contenham atributos de correspondência por endereços IP sejam tratadas adequadamente antes de serem transcompiladas para a sintaxe do Merlin. Caso não existam rotas BGP conhecidas, ou o protocolo não esteja em operação, a biblioteca não irá tratar os atributos de endereços IP de origem e de destino das regras OpenFlow.

Os prefixos de rede obtidos das rotas BGP coletadas da RIB são mantidos em uma lista de redes alcançáveis. Essa lista contém todos os prefixos de rede conhecidos para os quais o AS é capaz de enviar tráfego. Ela é iterada toda vez que a biblioteca trata os atributos relacionados a endereçamento IP, a fim de restabelecer as máscaras de sub-rede que são removidas durante a transcompilação para a sintaxe do Merlin. Mais detalhes na subseção seguinte.

### 3.3.3 O módulo de políticas e a integração com o Merlin

Neste trabalho, uma política é um conjunto de regras, em blocos de código ou já compiladas, que define um comportamento que a rede subjacente deve respeitar. A biblioteca possui um módulo de políticas que abrange a integração com o Merlin.

Durante a etapa de transcompilação das diretivas Havox em políticas Merlin, é necessário remover as máscaras de sub-rede dos prefixos de rede nos atributos de correspondência por endereço IP de origem e de destino. A razão disso é porque o Merlin não opera com endereços contendo máscaras. Ao final da transcompilação, o código Merlin gerado terá todos os endereços IP dos atributos, se houver, sem as suas máscaras. Estas serão recuperadas depois, na etapa de tratamento.

Passada a etapa de transcompilação das diretivas da DSL do Havox, o código Merlin gerado é gravado em um arquivo no sistema de arquivos local. Esse arquivo é submetido para o compilador do Merlin, o qual irá gerar regras OpenFlow primitivas impressas em *stdout*. Como essas regras são textuais, é realizado um *parsing* das mesmas e posterior estruturação em objetos[4].

Depois da estruturação, as regras passam por um processo de tratamento cujas etapas podem ou não ser executadas, de acordo com os parâmetros que o usuário fornece na requisição à API do Havox. São elas:

[4]No escopo da programação orientada a objetos, um objeto é uma instância de uma classe, que por sua vez representa um conjunto de objetos com características similares. Objetos guardam estados próprios e possuem atributos e métodos de acordo com a implementação de suas classes.

**Sobrescrita de atributos:** O próprio *framework* Merlin ainda não é plenamente estável em sua versão corrente. Dependendo das políticas que serão compiladas, são geradas regras com atributos conflitantes no predicado, o que requer uma correção da saída. Esta etapa identifica e permite que os atributos sejam reescritos, caso a opção para tal seja passada. Caso contrário, o primeiro valor atribuído é usado. Um exemplo de caso em que isso acontece é quando uma política contendo um atributo de endereço IP de destino é compilada. As regras geradas virão tanto com o endereço IP do *host* de destino quanto com o endereço IP de destino passado pela diretiva. Porém, em todos os testes realizados, o endereço IP do *host* sempre vem antes do endereço vindo da diretiva. Se este parâmetro estiver definido como falso, o endereço IP do *host* será considerado. Se definido como verdadeiro, o valor do atributo é sobrescrito após a leitura do endereço vindo da diretiva.

**Substituição de ações de encaminhamento:** O padrão do Merlin é gerar regras cujas ações de repasse sejam para enfileiramento dos pacotes nas saídas especificadas dos *switches* (ação `enqueue(<porta>, <fila>)`), o que dá suporte a QoS (qualidade de serviço, do inglês *Quality of Service*) do protocolo OpenFlow quando há filas configuradas. Como este trabalho ainda é uma prova de conceito, o recurso de filas ainda não é utilizado, portanto esta etapa substitui as ocorrências de ação de enfileiramento por ação de encaminhamento simples (`output(<porta>)`).

**Expansão de predicados:** Subconjuntos de regras geradas pelo Merlin que correspondem a um determinado fluxo de pacotes são identificados internamente por IDs de VLAN, que atuam como rótulos únicos para cada tipo de tráfego. Nos *switches* de entrada, as primeiras regras são formadas por predicados completos com todos os atributos a serem avaliados do pacote. Quando há uma correspondência, além da ação de encaminhamento, o Merlin também aplica ao pacote uma ação de definição de ID de VLAN (`set_vlan_id(<número>)`) com um valor único e arbitrário. Ao sair do *switch* de entrada, os pacotes daquele fluxo passam a ter o atributo `vlan_id` definidos e os *switches* seguintes do caminho avaliarão apenas esse atributo para identificar o fluxo. No *switch* de saída, o atributo `vlan_id` é apagado (ação `set_vlan_id(null)`) e os pacotes do fluxo são encaminhados para o *host* devido. Esta etapa expande todas as regras para que usem o predicado completo nas correspondências, em vez do ID de VLAN. As Tabelas 3.3 e 3.4 exemplificam um caso de uso sem e com expansão das regras, com três

*switches s1*, *s2* e *s3* interligados e *hosts* conectados a *s1* e *s3*, de entrada e saída do fluxo, respectivamente. Para fins de entendimento, foram usados atributos genéricos *a*, *b* e *c* e nomes em vez de números como parâmetro das ações `output`. Esta etapa foi projetada visando trabalhos futuros, quando se pretende usar a biblioteca Havox para exportar regras OpenFlow para outras plataformas diferentes do RouteFlow e que podem não implementar o suporte a IDs de VLAN. O RouteFlow mesmo não implementa VLANs em sua versão original. Porém, para este trabalho, o código-fonte do RouteFlow foi aprimorado para que contemplasse esse suporte.

**Tradução de sintaxe das regras:** Conforme mencionado, um dos intuitos deste trabalho é que a ferramenta cerne da arquitetura, a biblioteca Havox, seja modular e tenha capacidade para ser integrada a diferentes arquiteturas e casos de uso. As regras OpenFlow estruturadas podem ser traduzidas para quaisquer sintaxes conhecidas, desde que as lógicas de tradução sejam implementadas. Nesta prova de conceito, tendo em vista a arquitetura como um todo, o padrão estabelecido é a sintaxe compatível com o RouteFlow.

| *Switch* | Regra não expandida |
|---|---|
| *s1* | `a = 1 and b = 2 and c = 3 -> set_vlan_id(10) and output(s2)` |
| *s2* | `vlan_id = 10 -> output(s3)` |
| *s3* | `vlan_id = 10 -> set_vlan_id(null) and output(host)` |

Tabela 3.3: Regras não expandidas usando IDs de VLAN.

| *Switch* | Regra expandida |
|---|---|
| *s1* | `a = 1 and b = 2 and c = 3 -> output(s2)` |
| *s2* | `a = 1 and b = 2 and c = 3 -> output(s3)` |
| *s3* | `a = 1 and b = 2 and c = 3 -> output(host)` |

Tabela 3.4: Regras expandidas sem o uso de IDs de VLAN.

Todas as etapas de tratamento citadas são opcionais e dependem dos argumentos fornecidos à API do Havox. Independentemente dos argumentos, é executado um tratamento adicional para a recuperação das máscaras de sub-rede dos prefixos de rede dos atributos de endereçamento IP das regras.

A lista de redes alcançáveis, se populada, é iterada para cada regra que contenha atributos de endereçamento IP. O valor do atributo é avaliado para verificar se está contido dentro da faixa de endereços de cada prefixo da lista. O prefixo que contiver será usado no lugar do valor do atributo de endereço IP de origem

ou de destino, o que fará com que a informação de máscara seja recuperada para a regra.

O Merlin automaticamente inclui atributos de endereço IP de origem e de destino em todas as regras, visto que o *framework* foi idealizado para operar em redes institucionais, mesmo que tal atributo não tenha vindo por diretiva. Isso ocorre porque o Merlin usa os endereços IP do *host* de origem e do *host* de destino para classificar o tráfego, supondo que os pacotes sejam criados e entregues nos *hosts*. Numa rede institucional funciona dessa maneira, mas esse não é o *modus operandi* da *Internet*, cujos pacotes podem ter origens e destinos diversos. Para este trabalho, esses atributos de endereçamento IP adicionados pelo Merlin são considerados inválidos.

O tratamento da máscara de sub-rede também elimina os atributos de correspondência por endereços IP inválidos. Os valores são identificados como inválidos quando não constam na lista de redes alcançáveis. O resultado final do tratamento são regras que só possuirão correspondência por endereços IP de origem ou destino se de fato esses atributos vieram das diretivas.

Findados os passos de tratamento, o conjunto final de regras especiais é exportado pela API em formato de mensagem JSON.

### 3.3.4 O módulo de rotas e a integração com o RouteFlow

O módulo de rotas é responsável pela obtenção e tratamento das rotas conhecidas do sistema e pela integração com o RouteFlow.

Durante a inicialização, o RouteFlow envia uma requisição POST [39] à API do Havox para obter as regras especiais que serão instaladas. A razão de se optar pelo método POST do HTTP em lugar do método GET é que o RouteFlow envia à API os arquivos de topologia e de diretivas Havox, além dos parâmetros opcionais descritos na seção anterior.

Para que o RouteFlow pudesse fazer essa requisição, foi desenvolvido o módulo RFHavox, que implementa a requisição contendo os dois arquivos e os parâmetros de configuração necessários. O módulo RFHavox é invocado após a execução dos demais módulos do RouteFlow, quando as rotas BGP já terão convergido. A Tabela 3.5 descreve os parâmetros da requisição que são reconhecidos pela API.

| Nome | Exemplo | Req.? | Descrição |
|---|---|---|---|
| `dot_file` | `'routeflow.dot'` | Sim | Arquivo de topologia. |
| `hvx_file` | `'routeflow.hvx'` | Sim | Arquivo de diretivas. |
| `qos` | `'min(100 Mbps)'` | Não | *String* de QoS para as políticas. |
| `force` | `'true'` | Não | Sobrescrever atributos? |
| `output` | `'true'` | Não | Substituir `enqueue` por `output`? |
| `expand` | `'false'` | Não | Expandir predicados? |
| `syntax` | `'routeflow'` | Não | Sintaxe das regras geradas. |

Tabela 3.5: Parâmetros da requisição legíveis pela API.

A requisição recebida pela API do Havox é repassada ao módulo de rotas da biblioteca, que nesta prova de conceito precisa se conectar novamente ao RouteFlow para obter as rotas conhecidas. Por intermédio do RouteFlow, a biblioteca envia comandos "`show ip route`" a todas as instâncias de roteamento Quagga ativas, instruindo-as a imprimir as suas tabelas de roteamento completas em *stdout*. As rotas são lidas por *parsing* e estruturadas em objetos tal qual as regras no módulo de políticas.

Quando estruturadas, as rotas são mantidas em uma outra estrutura, a RIB global, que centraliza todas as rotas das instâncias de roteamento. Cada rota guarda o nome da sua instância de roteamento de origem, seu protocolo, o endereço de rede de destino, o endereço IP do próximo salto, se está na FIB e se é a melhor rota da sua instância de roteamento àquele endereço de rede. Essa RIB global será consultada durante o processo de transcompilação de diretivas em políticas do Merlin.

A resposta da requisição é devolvida ao módulo RFHavox, que irá extrair as regras especiais da mensagem JSON, encapsulá-las em mensagens RouteMod e enfileirá-las no canal de comunicação existente entre o RFServer e o RFProxy.

## 3.4 Trabalhos relacionados

No Capítulo 2, foram introduzidos o Merlin e o RouteFlow, que são, além de trabalhos relacionados, componentes que integram a arquitetura proposta neste trabalho. Neste capítulo, foi explicado também como esses componentes integram a arquitetura Havox. Entretanto, existem várias outras iniciativas de linguagens de programação de redes OpenFlow, seguindo diversos paradigmas, com o intuito principal de elevar a abstração das camadas inferiores na pilha SDN [19, 40]. Foi feita uma extensa pesquisa na literatura atual sobre SDN, as linguagens pro-

postas e assuntos relacionados até o momento deste trabalho ser idealizado, então pode-se afirmar que cada um dos trabalhos revisados tem uma relação com este, mesmo que minimamente. Serão elencados nesta seção, porém, aqueles que não apenas guardam maior similaridade com a proposta deste trabalho, mas também possuem código-fonte disponibilizado em repositório público ou que ainda são mantidos pelos seus desenvolvedores.

### 3.4.1 O projeto Frenetic

O projeto Frenetic[5] [26] é uma plataforma de programação de redes definidas por *software* que fornece um conjunto de métodos para a abstração da camada de dados e sintaxes de alto nível para a definição de políticas e de consultas ao estado da rede. Inclusive, o *framework* Merlin possui o Frenetic na sua pilha de aplicações, uma vez que a saída gerada pelo *framework* tem o propósito de ser usada como configuração a ser traduzida pela segunda etapa de compilação de uma rede controlada pelo Frenetic, conforme será explicado adiante.

O Frenetic permite que o administrador da rede possa programar aplicações que irão operar sobre o controlador, fazendo uso da API e dos métodos fornecidos para definir como se dará o comportamento da rede, bem como para levantar estatísticas relacionadas às correspondências entre pacotes e regras, como por exemplo o número de pacotes e de Bytes que corresponderam a uma determinada regra OpenFlow. Essa abstração fornecida pela plataforma é implementada por um processo de compilação de duas etapas que traduz o código em alto nível do administrador para políticas de encaminhamento intermediárias, que por sua vez são traduzidas por um sistema de tempo real do Frenetic em regras OpenFlow para instalação nas tabelas de fluxo dos *switches*. No caso da abstração do levantamento de estatísticas, são usados algoritmos de tratamento de pacotes que não tiveram correspondência na camada de dados. Esses pacotes chegam ao controlador, são avaliados segundo seus cabeçalhos e posteriormente são encaminhados pelas saídas devidas, sem a instalação de novas regras de fluxo. O motivo de não se instalar regras de fluxo nesses casos é que se houver uma regra no *switch* que corresponda a um determinado fluxo de pacotes visado para a coleta de estatísticas mais complexas, os mesmos não subirão ao controlador para análise e, portanto, serão desconsiderados.

A vantagem da plataforma do Frenetic sobre o uso de outros controladores

[5]Disponível em `https://github.com/frenetic-lang/frenetic`.

```
1 def repeater():
2   rules = [
3     Rule(inport: 1, [fwd(2)]),
4     Rule(inport: 2, [fwd(1)])
5   ]
6   register(rules)
7
8 def web_monitor():
9   q = (Select(bytes) *
10        Where(inport = 2 & srcport = 80) *
11        Every(30))
12  q >> Print()
13
14 def main():
15  repeater()
16  web_monitor()
```

Código 3.5: Exemplo de código do Frenetic.

convencionais é que estes em geral fornecem apenas métodos de mais baixo nível cujas implementações consistem de orquestrar diretamente a camada de dados. O Frenetic adiciona uma sofisticação à camada de controle ao usar os conceitos de modularização e de composição, em que métodos invocados durante a programação da rede adicionam, de forma isolada, múltiplas chamadas a métodos de mais baixo nível durante a primeira etapa da compilação. O exemplo do Código 3.5, escrito em Python[6] para o Frenetic, ilustra esses conceitos.

O código define dois métodos, `repeater` e `web_monitor`. O método `repeater` cria um vetor contendo duas instâncias de regra, sendo a primeira para encaminhar pela porta 2 os pacotes que venham pela porta 1 e a segunda, vice-versa. Em seguida, invoca o registro dessas regras na tabela de fluxos do *switch*. O método `web_monitor` usa a sintaxe de consulta própria do Frenetic para levantar estatísticas de número de Bytes de pacotes *web* que vêm da porta 2 do *switch* a cada 30 segundos e imprimir conforme a implementação do método `Print`. Por fim, o método `main`, como o nome já diz, é o método principal do código e invoca ambos os métodos supracitados para os fluxos de pacotes que chegam ao controlador. Esse trecho de código tomado como exemplo ilustra como a sintaxe do Frenetic permite que o administrador da rede abstraia de como as regras serão criadas e instaladas em níveis mais baixos da pilha de aplicações.

De início foi cogitado utilizar o Frenetic como parte da arquitetura deste trabalho. No entanto, isso adicionaria uma intensa etapa de implementação de código

[6]Disponível em `https://www.python.org/`.

```
1 (tpDst = 22 => [*.IDS.*])
2 & (tpDst = 22 => [*] with 1)
3 & (any => [GW.*.A])
```

Código 3.6: Exemplo de configuração do FatTire (caracteres adaptados).

do Frenetic para lidar com os protocolos básicos da rede, como o ICMP e o próprio BGP, e tal implementação não representaria contribuições de fato para este trabalho. Optou-se então por utilizar o RouteFlow, pois apesar de não contar com uma sintaxe sofisticada como a do Frenetic, já fornecia um ambiente pronto e que lida com os pacotes dos protocolos básicos citados.

### 3.4.2 O projeto FatTire

FatTire[7] [27] é um projeto integrante da plataforma Frenetic que também provê uma gramática de configuração simplificada do comportamento da camada de dados de redes OpenFlow, porém voltada para a definição de regras de tolerância a falhas em caso de queda de enlaces e de *switches*. Tal qual o Merlin, o FatTire também usa expressões regulares para a definição de caminhos que os pacotes correspondidos devem seguir. Entretanto, políticas de encaminhamento descritas para o compilador do FatTire especificam também expressões regulares para caminhos alternativos, caso o caminho primário seja inviabilizado. Um exemplo de configuração na sintaxe do FatTire é mostrado no Código 3.6.

Supondo que `IDS`, `GW` e `A` sejam nós da rede que implementam a detecção de intrusão, o *gateway* com a *Internet* e o acesso à rede interna, respectivamente, que o asterisco especifique quaisquer outros nós, que `tpDst` referencie o campo OpenFlow de porta lógica da aplicação e que `any` especifique quaisquer outros campos OpenFlow, o Código 3.6 instrui o sistema a criar regras que façam com que o tráfego SSH obrigatoriamente passe pelo nó que realiza detecção de intrusão (linha 1), com pelo menos um caminho possível de redundância em caso de falhas de enlace (linha 2). Independentemente das configurações anteriores, qualquer tipo de tráfego pode seguir por quaisquer nós intermediários da rede, desde que se originem no *gateway* e terminem no nó de acesso à rede interna (linha 3).

O processo de compilação do FatTire é similar ao do Merlin quanto à avaliação dos caminhos definidos pelas expressões regulares. O compilador instancia um grafo de encaminhamento em memória que é usado para verificar se os caminhos

[7] Disponível em `https://github.com/frenetic-lang/fattire`.

especificados pelo programador são condizentes com a topologia da rede. Caso seja especificado um caminho impossível, como o encaminhamento entre dois nós não conectados, ou não seja possível cumprir os requisitos de redundância especificados, o processo de compilação é interrompido com um erro.

Na época da submissão do projeto FatTire, o resultado da compilação era um novo arquivo de políticas para o NetCore, outro projeto da plataforma Frenetic. O NetCore[8] também era uma linguagem intermediária para a criação de regras OpenFlow. Detalhes sobre o NetCore foram propositalmente omitidos deste trabalho, visto que o mesmo foi descontinuado e incorporado ao compilador do Frenetic, já descrito anteriormente. Segundo o histórico de submissões de código no repositório, o FatTire agora compila diretamente para a sintaxe do Frenetic.

O FatTire, caso ainda fosse um projeto separado do Frenetic, talvez pudesse ser usado no lugar no Merlin como componente que calcula caminhos na forma de regras OpenFlow, por ter uma sintaxe ainda mais simplificada e visto que para este trabalho o que importa de fato é a abstração na escrita dos predicados das políticas e a definição de caminhos a serem percorridos internamente usando expressões regulares, algo que ambos implementam.

### 3.4.3 O projeto Propane

O projeto Propane[9] [41] é firmado nas mesmas motivações das pesquisas em redes definidas por *software*, apesar de não empregar propriamente esse conceito. O Propane é um compilador que traduz as políticas descritas com alto nível de abstração em configurações BGP distribuídas nos roteadores do domínio administrativo, se propondo assim a minimizar os problemas conhecidos e recorrentes da configuração do protocolo nos sistemas autônomos, como por exemplo a alta complexidade e a coerência com os SLAs vigentes, o que tornam a tarefa árdua, cara e muito propensa a erros [42, 43].

O usuário do Propane especifica quesitos como condições, preferências e restrições de tráfego em uma gramática própria cujo conteúdo passa por análises léxica, sintática e semântica, naturais de um processo de compilação de código, que avaliarão se o código é válido dentro do escopo do domínio e se fere as restrições lógicas e físicas do mesmo, como a topologia local e a vizinhança. O resultado da compilação são as configurações individuais e textuais das instâncias

[8] Disponível em `https://github.com/frenetic-lang/netcore-1.0`.
[9] Disponível em `https://github.com/rabeckett/propane`.

```
1 define Prefs = exit(R1 >> R2 >> Peer >> Prov)
2 define transit(x, y) = enter(x | y) & exit(x | y)
3 define cust_transit(x, y) = later(x) & later(y)
4
5 define Routing = {
6   PCust => Prefs & end(Cust)
7   true => Prefs
8 }
9
10 define NoTrans = {
11   true => !transit(Peer, Prov) & !cust_transit(Cust, Prov)
12 }
13
14 Routing & NoTrans
```

Código 3.7: Exemplo de política descrita na DSL do Propane.

BGP que operam no AS. O Código 3.7 é um exemplo da sintaxe simplificada do Propane.

No exemplo, supondo que `R1` e `R2` sejam roteadores de borda que vinculam o AS a outros dois, `Cust` (cliente que paga por trãnsito) e `Prov` (provedor com trânsito cobrado), e que `R4` e `R5` ligam a outro AS `Peer` (par com trânsito gratuito), é declarada a variável `Prefs` contendo a ordem de preferência de saída de tráfego por `R1`, `R2`, `Peer` e `Prov`, na devida ordem. São definidas em seguida as funções `transit` e `cust_transit`, as quais a primeira estabelece tráfego de trânsito com entrada e saída entre vizinhos `x` e `y` e a última estabelece trânsito para trâfegos que passem tanto por `x` quanto por `y` depois que deixarem o AS. `Routing` define que todo tráfego direcionado aos prefixos de `Cust` contidos em `PCust` deve seguir a preferência expressa em `Prefs` e deve seguir caminhos que terminem no próprio AS `Cust`. Qualquer outro tipo de tráfego que não confere com os prefixos de `Cust` deve apenas respeitar as preferências em `Prefs`. `NoTrans` usa as funções `transit` e `cust_transit` definidas anteriormente para não permitir trânsito entre `Peer` e `Prov` e para prevenir contra tráfego que deixe o AS e mais tarde passe tanto por `Cust` quanto por `Prov`. Na última linha, as regras de `Routing` e `NoTrans` são invocadas.

Ainda sobre o cenário de exemplo, o Código 3.7 implementa as seguintes políticas: (1) priorizar saída para o cliente `Cust`, depois para o par `Peer` e por último para o provedor `Prov`, (2) não permitir tráfego entre `Prov` e `Peer`, (3) para `Cust`, preferir saída pelo roteador `R1` a `R2`, (4) garantir que `Cust` esteja no caminho para seus prefixos e (5) garantir que `Cust` não seja trânsito para `Prov`. As políticas 1, 3 e 4 são implementadas pelas regras contidas em `Routing` e as políticas 2 e 5,

pelas em `NoTrans`.

O Propane guarda similaridades com o conceito de SDN no que tange à definição centralizada de políticas da rede, mas difere-se essencialmente porque não há um controlador separado da camada de dados dos dispositivos. Deveras, os dispositivos de rede ainda operam de forma independente uns dos outros, com apenas a definição de suas configurações BGP compiladas por um elemento central executando o Propane. Os motivos citados no projeto são que SDN ainda é mais empregado em situações intradomínio, quando definições de rotas e afins estão em um escopo interdomínio, e que a dependência das redes programáveis de um controlador logicamente centralizado, mesmo que fisicamente escalável, ainda pode representar um ponto de falha da rede caso os enlaces entre as camadas de controle e de dados fiquem inoperantes, o que aumentaria demais o tempo de reação a problemas relacionados às rotas. O problema da resiliência do controlador poderia ser mitigado com o uso de soluções já conhecidas de garantia de alta disponibilidade, como o balanceador de cargas HAProxy[10].

O uso do Propane como componente da arquitetura Havox é idealizado como trabalho futuro no Capítulo 6, na configuração das instâncias de roteamento BGP rodando na camada virtual do RouteFlow. Atualmente, essas instâncias executam com as configurações definidas caso a caso.

### 3.4.4 Um ponto de troca de tráfego definido por *software*

Uma outra iniciativa de se utilizar redes definidas por *software* em questões envolvendo o protocolo BGP é o projeto SDX[11] [44], que implementa um ponto de troca de tráfego (IXP, do inglês *Internet Exchange Point*) com um *switch* OpenFlow interligando roteadores de borda de sistemas autônomos participantes.

O SDX oferece uma abstração na qual cada AS tem a visão de um *switch* OpenFlow virtual próprio, onde podem ser compiladas regras a partir de políticas escritas na DSL Pyretic[12] [45]. Como os *switches* virtuais não existem de fato, essas políticas pertinentes a cada AS são avaliadas e transformadas em uma política global, que é posteriormente compilada para regras OpenFlow que contemplam tudo o que foi definido pelos participantes.

[10] Disponível em `http://www.haproxy.org/`.

[11] Disponível em `https://noise-lab.net/projects/software-defined-networking/sdx/`.

[12] O Pyretic também é um trabalho relacionado a este, mas foi propositalmente omitido porque foi descontinuado e absorvido pelo projeto Frenetic.

```
1 (match(dstport = 80) >> fwd(B)) +
2 (match(dstport = 443) >> fwd(C))
```

Código 3.8: Política de saída para o *switch* virtual do AS *A* no SDX.

```
1 (match(srcip = {0.0.0.0/1}) >> fwd(B1)) +
2 (match(srcip = {128.0.0.0/1}) >> fwd(B2))
```

Código 3.9: Política de entrada para o *switch* virtual do AS *B* no SDX.

A Figura 3.4 ilustra a abstração provida pelo SDX. Nela, três *switches* virtuais estão associados aos sistemas autônomos *A*, *B* e *C*, respectivamente. O conjunto dos três *switches* virtuais é na realidade um único *switch* físico com quatro portas físicas, *A*, *B1*, *B2* e *C*, que levam aos participantes correspondentes.

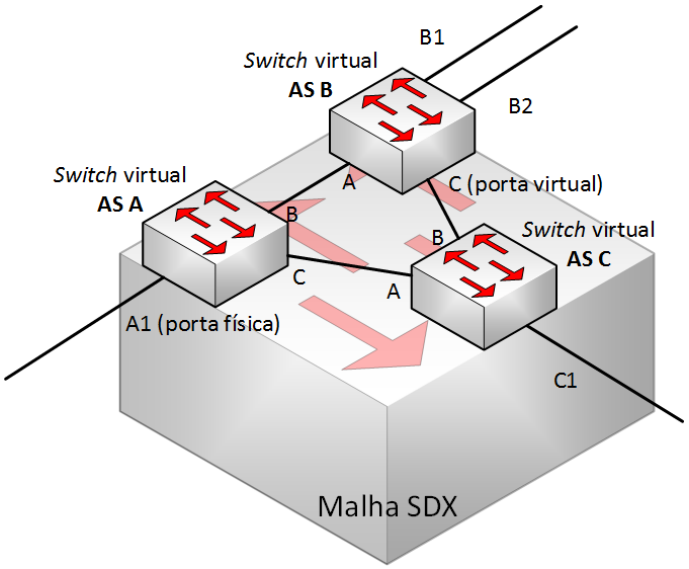



Figura 3.4: Exemplo de abstração de *switches* virtuais provida pelo SDX.

O AS *A* pode escrever, por exemplo, uma política de saída conforme o Código 3.8, determinando que tráfego HTTP vindo de si próprio deve ser encaminhado para o AS *B* e tráfego HTTPS (*Hypertext Transfer Protocol Secure*) para *C*. De igual maneira, o AS *B* pode escrever uma política de entrada nos moldes do Código 3.9, determinando que tráfego de fora com origem na rede 0.0.0.0/1 deve entrar em *B* pelo acesso 1 e tráfego com origem na rede 128.0.0.0/1, pelo acesso 2. Na definição do trabalho que deu origem ao SDX, uma política é de entrada quando estabelece regras para tráfego que vai do SDX para o AS e de saída quando define regras para tráfego que sai do AS para o SDX.

O sistema do SDX compila essas políticas individuais de cada AS participante

```
1 (match(port = A1, dstport = 80, srcip = {0.0.0.0/1}) >> fwd(B1))
2 (match(port = A1, dstport = 80, srcip = {128.0.0.0/1}) >> fwd(B2))
3 (match(port = A1, dstport = 443) >> fwd(C1))
4 (match(port = C1, srcip = {0.0.0.0/1}) >> fwd(B1))
5 (match(port = C1, srcip = {128.0.0.0/1}) >> fwd(B2))
```

Código 3.10: Política global resultante do SDX gerada a partir das políticas do AS *A* e do AS *B*.

em uma única política global similar ao Código 3.10. Posteriormente, essa política global é traduzida em regras OpenFlow pelo processo de compilação padrão do Pyretic [45, 26].

A solução de IXP usando o sistema SDX se baseia na ideia de que também é possível aplicar o conceito de SDN em cenários interdomínio envolvendo BGP, mesmo que seja mais complexo do que em cenários intradomínio, como redes institucionais. De fato, há uma dificuldade inerente em se manter a operação em SDN quando a rede se estende a outros domínios administrativos. Questões operacionais e políticas surgem, como por exemplo a compatibilidade entre os dispositivos da camada de dados de sistemas autônomos distintos, a localização do controlador e quem tem autoridade sobre o mesmo. O projeto SDX atua nos IXPs, que são zonas neutras entre os sistemas autônomos mas ainda assim são domínios e estão sob a responsabilidade de alguém, então pode-se dizer que é uma solução intradomínio para um problema interdomínio. Este trabalho segue a mesma linha, fornecendo uma possível solução intradomínio que deriva da obtenção das rotas BGP interdomínio.

### 3.4.5 Roteamento como serviço com o RouteFlow Aggregator

Conforme já citado, a arquitetura RouteFlow mantém, em contêineres na camada virtual, instâncias de *software* de roteamento operando com conectividade idêntica à topologia de *switches* OpenFlow na camada física subjacente, em uma relação de um contêiner para um *switch*.

Como uma expansão para a arquitetura RouteFlow, foi implementado o RouteFlow Aggregator [46, 47, 48], que opera sobre a mesma e fornece uma plataforma de roteamento como serviço capaz de prover abstração da conectividade na camada física de *switches*. A expansão extingue a relação um para um entre contêineres com instâncias de roteamento e *switches* OpenFlow e cria uma máquina virtual agregadora que roda uma única instância de roteamento, como

se de um ponto de vista externo toda a rede contasse com apenas um roteador fazendo vizinhança com sistemas autônomos adjacentes. Isso é possível porque no lugar da associação de instâncias de roteamento a *switches*, é feita a associação das interfaces de acesso ao AS dos *switches* com as interfaces da máquina virtual agregadora. Com isso, tomando o BGP como exemplo de protocolo, é necessário apenas um arquivo de configuração que prevê a conectividade do AS em questão com os vizinhos.

O RouteFlow Aggregator extrai as informações geradas pelas decisões de roteamento da camada virtual e dissemina atualizações para as tabelas de fluxo dos *switches* OpenFlow. Isso acaba minimizando, senão eliminando, o uso de protocolos IGP para roteamento interno, como o RouteFlow original usa entre seus contêineres. A limitação do RouteFlow Aggregator, porém, reside no fato de que todos os *switches* devem estar sempre conectados em *full-mesh*, de forma que um fluxo de pacotes dê no máximo um salto dentro do domínio administrativo, de uma borda a outra.

O modo de operação do RouteFlow Aggregator se assemelha ao do projeto SDX, visto que em ambos os casos há a visão macroscópica de um único dispositivo operando no domínio. A diferença, além das propostas, é que no caso do SDX de fato se trata de um *switch* OpenFlow real operando sob a abstração de *switches* virtuais alocados para cada AS participante, enquanto que no caso do RouteFlow Aggregator se trata de uma instância de roteamento virtual que abstrai os múltiplos *switches* OpenFlow na camada de dados subjacente.

O RouteFlow Aggregator foi cogitado para integrar a arquitetura deste trabalho. Contudo, devido a dificuldades de acesso ao seu código-fonte, não foi possível realizar essa integração nesta etapa da prova de conceito.

### 3.4.6 Uma proposta para a base do OpenFlow 2.0

O protocolo OpenFlow, em suas versões menores lançadas até então, requer que sejam especificados os cabeçalhos de protocolos de rede sobre os quais opera [28, 29, 30, 31, 32, 33]. Conforme foi mostrado na Tabela 2.1, a cada incremento de versão menor do protocolo OpenFlow foram adicionados novos campos de cabeçalho legíveis. A maioria desses campos se refere a protocolos de rede que vão sendo adicionados à compatibilidade com o OpenFlow conforme a demanda ou a QoS. Essa adição recorrente de suporte a protocolos de rede pelo OpenFlow claramente não é escalável, visto que a todo momento surgem novos protocolos

```
1 header ethernet {
2   fields {
3     dst_addr:  48; // Largura em bits do MAC de destino.
4     src_addr:  48; // Largura em bits do MAC de origem.
5     ethertype: 16; // Largura em bits do ID de protocolo.
6   }
7 }
```

Código 3.11: Exemplo de cabeçalho em código P4.

que aumentam a complexidade das especificações, mas não flexibiliza a definição de novos campos de cabeçalhos. Surgiu então a iniciativa de pesquisa do P4 (*Programming Protocol-independent Packet Processors*) [49].

O P4[13] é um candidato a novo padrão para o protocolo OpenFlow na sua próxima versão maior, 2.0, cujo objetivo é permitir que os programadores de rede possam definir como deve ser realizada a leitura e a correspondência dos cabeçalhos dos pacotes dos vários protocolos de rede sem estarem restritos ao que determina a especificação do OpenFlow. Para tanto, o programador da rede implementa e submete aos *switches* o código dos *parsers* dos cabeçalhos de pacotes e as regras de correspondência e ações para processamento desses cabeçalhos. O objetivo é que o programador da rede abstraia dos detalhes de funcionamento da camada de dados subjacente e delegue a um compilador a responsabilidade de traduzir as definições agnósticas de *parsers* para um programa dependente do dispositivo.

Um programa em P4 tem (1) definições de cabeçalhos com a especificação da organização em sequência e da largura em bits dos campos, (2) definições de *parsers* com a especificação de como identificar cabeçalhos dos pacotes, (3) definições de tabelas com a especificação de campos que podem ter correspondência e as respectivas ações a serem tomadas, (4) definições de ações complexas a partir de ações básicas e (5) programas de controle que definem a ordem em que as tabelas de correspondências e ações são avaliadas por pacote. Os Códigos 3.11, 3.12, 3.13, 3.14 e 3.15 exemplificam um programa P4, na devida ordem, para definir um cabeçalho, seu *parser*, uma tabela de correspondência, uma ação e o programa de controle principal.

O programa de controle descrito no Código 3.15 aplica aos pacotes as regras de correspondência definidas na tabela `ethernet_table`, descrita no Código 3.13. A tabela verifica se o valor de endereço MAC de destino confere exatamente com

[13] Disponível em `http://p4.org/`.

```
1 parser ethernet {
2   switch(ethertype) {
3     case 0x0800: ipv4; // Quadro encapsula um datagrama IPv4.
4     case 0x86dd: ipv6; // Quadro encapsula um datagrama IPv6.
5     case 0x0806: arp;  // Quadro tem uma mensagem ARP.
6   }
7 }
```

Código 3.12: Exemplo de *parser* em código P4.

```
1 table ethernet_table {
2   reads {
3     ethernet.dst_addr: exact;
4   }
5   actions {
6     set_src_addr;
7   }
8   size: 1024; // Total de entradas suportadas pela tabela.
9 }
```

Código 3.13: Exemplo de tabela em código P4.

```
1 action set_src_addr(new_addr) {
2   // Modifica o MAC de origem com o valor fornecido.
3   set_field(ethernet.src_addr, new_addr);
4 }
```

Código 3.14: Exemplo de ação em código P4.

```
1 control main {
2   // Avalia o pacote com a tabela ethernet_table.
3   apply(ethernet_table);
4 }
```

Código 3.15: Exemplo de programa de controle em código P4.

o fornecido e, em caso positivo, aplica a ação que redefine o endereço MAC de origem, conforme o Código 3.14. Para fazer o *parsing* dos endereços MAC, a tabela usa o *parser* do Código 3.12 e os campos previstos no Código 3.11 do cabeçalho `ethernet`.

Quando ou se vier a ser oficializado, o P4 trará mudanças substanciais na forma como se dá atualmente a programação de redes definidas por *software*. As regras OpenFlow passarão a ser geradas por códigos de *parsing* executando dentro dos *switches*, instalados através de controladores. A proposta dá uma primeira impressão de que está devolvendo a lógica de controle para a camada de dados, o que iria na contra-mão dos preceitos de SDN. Porém, um olhar mais apurado revela que a lógica de orquestração permanece na camada de controle e o que é delegado de fato à camada de dados é só a execução dos *parsers* implementados. O benefício direto disso é a possibilidade de que as correspondências e ações implementadas sejam agnósticas à versão do protocolo OpenFlow nos *switches*.

## 3.5 Síntese

Este capítulo elucidou a proposta deste trabalho e seus pormenores em um aspecto conceitual, sem entrar em minúcias de como se sucedeu a instalação e implantação da ferramenta, dos componentes usados, das máquinas virtuais e conexões entre elas e dos recursos computacionais utilizados, deixando esse nível de detalhamento técnico para o capítulo seguinte.

Foi explicado que o cerne da arquitetura Havox é a biblioteca homônima, que tem o seu funcionamento exposto por meio de uma API que fornece um *endpoint* de regras para requisições POST oriundas do RouteFlow. Os arquivos de topologia e de diretivas Havox, bem como os parâmetros passados, definem como será a transcompilação das diretivas em políticas Merlin. Essas políticas, no formato de um arquivo contendo blocos de código da DSL do Merlin, são submetidas ao mesmo, que irá compilar e retornar regras OpenFlow primitivas, posteriormente tratadas pelo Havox e entregues ao RouteFlow via mensagens JSON. Por fim, este último irá instalar as regras especiais nas tabelas de fluxo dos *switches* com prioridade superior às suas próprias regras básicas, permitindo que as definições de tráfego criadas pelo usuário prevaleçam sobre as da plataforma.

Foram também elencados trabalhos relacionados que possuem relação com a proposta deste. Cada um deles possui um ponto de interseção com este, de

forma que serviram de base para o projeto ou servirão para planejar futuros aprimoramentos ou caminhos que deverão ser seguidos.

Alguns dos trabalhos apresentados, apesar de guardarem bastante similaridade com este, diferem quanto ao escopo de utilização. O caso do ponto de troca de tráfego SDX, por exemplo, é similar no que tange à compilação de políticas de usuário em regras de encaminhamento, mas difere-se porque este trabalho tem como objetivo a orquestração de encaminhamento dentro de um AS com vários *switches*, enquanto o SDX objetiva a configuração de um único *switch* através da visão de *switches* virtuais para cada AS participante.

# 4. Implantação

Este capítulo explana os aspectos técnicos da instalação e execução da arquitetura Havox e dos seus componentes. Serão abordados os motivos para a escolha dos recursos utilizados, os desafios encontrados e as razões para as decisões que foram tomadas, algumas puramente técnicas e outras subjetivas.

## 4.1 Recursos computacionais

Como prova de conceito, a arquitetura foi implementada e testada usando máquinas virtuais gerenciadas pelo *software* virtualizador VirtualBox, versão 5.0.20, em uma máquina hospedeira. Três máquinas virtuais de configurações parecidas executavam isoladamente o Merlin, o RouteFlow e o Mininet. Esse último é usado para emular uma rede física com *switches* OpenFlow virtualizados pelo Open vSwitch[1].

A máquina hospedeira executava o sistema operacional Ubuntu 16.04 x64 sobre *hardware* formado por um processador Intel Core i5-4210U de 2.40 GHz com dois núcleos e memória DDR3L de 8 GB. A máquina hospedeira era responsável pela execução da biblioteca Havox e de sua API.

Cada uma das máquinas virtuais possuía 1 GB de memória. A máquina virtual responsável pelo RouteFlow usava um sistema operacional Ubuntu Server 12.04. As máquinas responsáveis pelo Merlin e pelo Mininet usavam sistemas operacionais Ubuntu 14.04. Essas máquinas virtuais foram obtidas dos respectivos repositórios.

As conexões físicas entre as máquinas virtuais e a máquina hospedeira, no

[1]Disponível em `http://openvswitch.org/`.

escopo da configuração do VirtualBox, eram por meio de uma conexão exclusiva de hospedeiro e uma conexão NAT[2]. A conexão exclusiva de hospedeiro mantinha uma rede acessível apenas pelas máquinas com esse modo de conexão e sem acesso à *Internet*. A conexão NAT era o modo que permitia que as máquinas virtuais acessassem a *Internet* através da máquina hospedeira. O acesso à *Internet* não era necessário para a condução do experimento, mas sim para a atualização ou instalação de pacotes e comunicação com os repositórios que mantêm os códigos-fonte dos componentes.

As conexões lógicas entre os processos executando nas máquinas se dava via SSH usando o endereço de rede criado pelas conexões exclusivas de hospedeiro. Mesmo que os componentes todos executem na mesma máquina hospedeira, o intuito era mostrar que a arquitetura está projetada desde a sua concepção para que os componentes possam estar localizados em quaisquer endereços remotos.

## 4.2 Configuração dos componentes

Cada componente da arquitetura Havox opera em um ambiente computacional próprio e isolado dos demais, o que requer sequências de configuração distintas para cada caso. Mesmo com alguns dos componentes tendo máquinas virtuais disponíveis em seus repositórios de origem, passos adicionais de configuração são necessários, a fim de que toda a integração seja realizada.

As versões das linguagens dos componentes são descritas incluindo suas versões de correção, isto é, o terceiro número, tomando a convenção do versionamento semântico [34] como referência. A instalação de versões que diferem apenas em correção geralmente não impactam no funcionamento de um sistema, visto que contemplam apenas refatorações ou correções de *bugs*. Portanto, mesmo que este trabalho sugira um componente na versão 2.4.0, por exemplo, não deve haver incompatibilidades caso seja instalada a versão 2.4.1 do mesmo.

### 4.2.1 Configuração da biblioteca Havox e sua API

A biblioteca Havox e a API que expõe suas funcionalidades são tecnicamente dois projetos de aplicação distintos, `havox` e `havox_api`, ambos implementados sobre a linguagem de propósito geral Ruby, versão 2.4.0. Ruby é uma linguagem

[2]Mais detalhes das opções de conexão em `https://www.virtualbox.org/manual/ch06.html`.

interpretada, multiparadigma, reflexiva, dinamicamente tipada e com gerenciamento automático de memória.

A biblioteca tem a estrutura de uma *gem* (ou gema) do Ruby, que é o termo pelo qual as bibliotecas são oficialmente conhecidas pela comunidade da linguagem. Uma *gem* pode ser importada como dependência por qualquer outra *gem* ou aplicação escritas em Ruby. Além disso, as convenções acerca das *gems* estabelecem que seja usado o versionamento semântico [34] para cada atualização de código, minimizando incompatibilidades com outros recursos por meio do controle de versão. Apesar da nomenclatura estabelecida pela comunidade Ruby ser "*gem*", este trabalho continuará referenciando a sua contribuição principal pelo termo agnóstico de linguagem "biblioteca".

Como qualquer biblioteca da linguagem, ela é invocada para execução pela aplicação da API, que é implementada usando o Sinatra[3], um *framework* leve em Ruby projetado para a rápida disponibilização de microsserviços da *web*. A versão utilizada do Sinatra é a 2.0.0.

Para a confecção deste trabalho e para a prova de conceito realizada, tanto a biblioteca Havox quanto a sua API estão na versão 0.9 em seus repositórios públicos. Portanto, os códigos-fonte de ambas exatamente conforme este trabalho descreve podem ser obtidos nessa versão.

Optou-se pela implementação em Ruby por alguns motivos. Um deles é a experiência prévia que já se tinha com a linguagem, o que possibilitou que funcionalidades complexas fossem implementadas com maior destreza do que seriam em outras linguagens. Outro motivo é o fato de a linguagem prover facilidade para se aplicar a metaprogramação, conceito fundamental para a implementação de uma DSL que requer que blocos inteiros de código sejam usados como argumentos do próprio programa. Um terceiro motivo, a ser explicado na seção de trabalhos futuros no Capítulo 6, é a integração da biblioteca com a arquitetura Trema[4], controlador de SDN também implementado em Ruby capaz de inicializar e gerenciar *switches* virtuais usando o Open vSwitch.

A configuração dos projetos da biblioteca e da API requer que a linguagem Ruby na versão 2.4.0 esteja instalada. Essa instalação pode ser realizada pelo gerenciador de pacotes padrão do sistema operacional, por pacote de instalação próprio ou por um gerenciador de versões do Ruby. Esta última forma é a

[3]Disponível em `http://www.sinatrarb.com/`.
[4]Disponível em `https://trema.github.io/trema/`.

```
1 Havox.configure do |config|
2   config.rf_host          = '192.168.56.101'
3   config.rf_user          = 'routeflow'
4   config.rf_password      = 'routeflow'
5   config.rf_lxc_names     = ['rfvmA', 'rfvmB', 'rfvmC', 'rfvmD']
6   config.merlin_host      = '192.168.56.102'
7   config.merlin_user      = 'frenetic'
8   config.merlin_password  = 'frenetic'
9   config.merlin_path      = '/home/frenetic/merlin'
10  config.gurobi_path      = '/opt/gurobi701/linux64'
11 end
```

Código 4.1: Arquivo `config/havox_setup.rb` da API.

recomendada, visto que facilita a instalação e manutenção de versões distintas no mesmo sistema. Dois gerenciadores de versões do Ruby conhecidos são o RVM[5] (*Ruby Version Manager*) e o rbenv[6]. Neste trabalho, foi usado o RVM.

Após instalada a linguagem Ruby e obtidos os códigos-fonte dos projetos da biblioteca e da API, deve-se configurar o arquivo `config/havox_setup.rb` no projeto da API. Esse arquivo de configuração concentra as definições necessárias para que a arquitetura possa operar. Nele devem constar os endereços IP das máquinas do RouteFlow e do Merlin, os respectivos usuários e senhas para conexão SSH e os caminhos dos projetos do Merlin e do Gurobi, componente do Merlin, no sistema de arquivos da máquina virtual. O Código 4.1 mostra um bloco de configurações da biblioteca Havox dentro da API e a Tabela 4.1, as descrições dos campos.

| Campo | Descrição |
|---|---|
| `rf_host` (*str*) | Endereço IP da máquina do RouteFlow. |
| `rf_user` (*str*) | Nome de usuário da máquina do RouteFlow. |
| `rf_password` (*str*) | Senha da máquina do RouteFlow. |
| `rf_lxc_names` (vetor *str*) | Nome das instâncias de roteamento do RouteFlow. |
| `merlin_host` (*str*) | Endereço IP da máquina do Merlin. |
| `merlin_user` (*str*) | Nome de usuário da máquina do Merlin. |
| `merlin_password` (*str*) | Senha da máquina do Merlin. |
| `merlin_path` (*str*) | Caminho do Merlin na máquina que o executa. |
| `gurobi_path` (*str*) | Caminho do Gurobi na máquina do Merlin. |

Tabela 4.1: Descrição das configurações do Código 4.1.

O Código 4.1 configura o Havox para se conectar via SSH à máquina do RouteFlow no endereço 192.168.56.101 usando o nome de usuário e senha "`routeflow`"e, uma vez conectado, acessar via nova conexão SSH os contêineres do Route-

[5]Disponível em `https://rvm.io/`.
[6]Disponível em `https://github.com/rbenv/rbenv`.

Flow que mantêm as instâncias de roteamento *rfvmA*, *rfvmB*, *rfvmC* e *rfvmD* para coleta de rotas. De igual maneira, o Havox se conectará ao Merlin no endereço remoto 192.168.56.102, com nome de usuário e senha "`frenetic`", e invocará o *framework* no caminho "`/home/frenetic/merlin`"do sistema de arquivos remoto, referenciando a licença do Gurobi, dependência do Merlin, em "`/opt/gurobi701/linux64`", também no sistema de arquivos remoto.

O RouteFlow, o Merlin e devidas dependências também podem ser configurados na mesma máquina que a biblioteca Havox e sua API. Nesse caso, os endereços IP devem ser *localhost* e os nomes de usuário e senhas devem ser os da máquina local.

Para o funcionamento da biblioteca nesta prova de conceito, é necessário que o processo do RouteFlow esteja em execução, pois é ele que mantém o funcionamento dos contêineres das instâncias de roteamento. Por sua vez, o processo do Merlin é invocado somente quando há novas submissões de políticas e se encerra após imprimir as regras em *stdout*.

### 4.2.2 Configuração do *framework* Merlin

O projeto do Merlin é implementado sobre a linguagem OCaml, versão 4.03.0. OCaml é uma linguagem de propósito geral compilada, multiparadigma, estaticamente tipada e com gerenciamento automático de memória. O código-fonte do Merlin usado neste trabalho foi recuperado do seu repositório público, com estado apontado pelo *commit* "`adbeeefaaf`".

O Merlin é uma aplicação que tem o propósito de executar sobre o Frenetic, conforme explicado no Capítulo 2, e por isso a sua saída padrão são regras primitivas e textuais que devem ser legíveis por este. Como ainda não há a intenção de se integrar o Frenetic à pilha de aplicações da arquitetura Havox, os resultados gerados pelo Merlin são coletados e processados em vez de serem utilizados pelo Frenetic.

A execução do Merlin requer que a linguagem OCaml esteja instalada no sistema. A instalação pode ser feita por gerenciador de pacotes do sistema operacional, por instalador próprio ou por um gerenciador de versões e pacotes do OCaml. Assim como existe o RVM para o Ruby, existe e é recomendado o OPAM[7] para gerenciar as versões instaladas do OCaml.

[7]Disponível em `https://opam.ocaml.org/`.

O Merlin possui outras dependências que requerem instalação manual, especialmente o otimizador Gurobi. Esta dependência é responsável por gerar um grafo de conectividade a partir do arquivo de topologia DOT submetido e calcular o melhor caminho de *switches* entre dois *hosts*. O resultado desse processamento é entregue ao Merlin, que irá gerar as regras primitivas sobre. O Gurobi é a única dependência proprietária do Merlin e, logo, da arquitetura Havox, requerendo que seja obtida uma licença acadêmica gratuita no *site* oficial.

Após a instalação do Merlin e do Gurobi, as informações de acesso e de caminho de ambos no sistema de arquivos da máquina devem ser informadas nas configurações da biblioteca Havox dentro do projeto da API, conforme mostra o Código 4.1.

### 4.2.3 Configuração da plataforma RouteFlow

Os módulos RFServer, RFProxy e RFHavox do projeto da plataforma RouteFlow são implementados sobre a linguagem Python, versão 2.7.3, enquanto o módulo RFClient é implementado sobre a linguagem C++ com compilador em versão 4.6.3. Python e C++ são linguagens de propósito geral multiparadigma, sendo a primeira interpretada, reflexiva, dinamicamente tipada e com gerenciamento automático de memória e a segunda compilada, estaticamente tipada e sem gerenciamento automático de memória.

Como o RouteFlow não possui atualizações recentes, a versão vigente é compatível apenas com o Ubuntu 12.04, pois algumas dependências não são compatíveis com versões mais recentes do sistema operacional. A arquitetura Havox usa uma máquina virtual distribuída pela equipe do RouteFlow através do seu repositório oficial. Essa máquina executa o Ubuntu na versão citada e contém todas as dependências necessárias para o funcionamento básico da plataforma, incluindo a linguagem Python na versão 2.7.3. A partir do *commit* "`a066b410b0`"do repositório público do RouteFlow, foi realizado um *fork* para outro repositório público próprio, onde as contribuições de código para integração com este trabalho foram armazenadas.

O módulo RFHavox, contribuição deste trabalho ao código-fonte da plataforma, requer a instalação de duas bibliotecas adicionais do Python chamadas *Requests* e *Ipaddress*, que não constam no conjunto de bibliotecas padrão. *Requests* é necessária porque fornece métodos que auxiliam no envio e recebimento de arquivos em requisições HTTP, com abstração dos detalhes de nível mais baixo,

como codificação textual e tamanho em *Bytes*. *Ipaddress* auxilia na manipulação de endereços de rede fornecidos por *string*, permitindo que cálculos de sub-redes e testes de inclusão em uma faixa de endereços sejam realizados de forma trivial.

Antes da execução do RouteFlow com o suporte do módulo RFHavox, é necessário configurar os parâmetros da requisição com a URL do *endpoint* da API que receberá a requisição e com o caminho local dos arquivos de diretivas Havox e de topologia da rede.

Do lado da API do Havox, o bloco de configurações do Código 4.1 deve ter as informações e credenciais de acesso pertinentes ao RouteFlow definidas.

Por fim, o RouteFlow deve ser executado somente se a rede virtual de *switches* gerenciada pelo Mininet já estiver ativa, conforme será abordado na próxima seção.

### 4.2.4 Configuração do emulador Mininet

O Mininet é um emulador de redes de *switches* OpenFlow virtuais implementado em Python, versão 2.7.6. A versão utilizada do Mininet é a 2.1.0.

A rede virtual criada pelo Mininet é configurada para operar com o controlador remoto mantido pelo RouteFlow e, por isso, o comando de execução do Mininet deve referenciar o endereço IP da máquina executando o RouteFlow como sendo o do controlador.

Um arquivo de topologia próprio do Mininet, escrito em Python, deve ser referenciado no ato da execução. Esse arquivo descreve a configuração topológica dos *switches* e dos *hosts* da mesma forma que o arquivo de topologia que é consumido pela biblioteca Havox e pelo Merlin. A diferença é que o arquivo de topologia do Mininet é exclusivo para o emulador e contém informações de outras bibliotecas importadas e lógica de programação, ao contrário do outro, que é uma mera descrição da conectividade do grafo topológico na linguagem DOT.

Os *hosts* criados pelo Mininet são capazes de disparar comandos de teste entre si e até mesmo simular conexões cliente-servidor usando a biblioteca SimpleHTTPServer[8] do Python. Porém, esses *hosts* compartilham de um mesmo espaço de PID (*Process Identifier*) e de usuário e, portanto, não são capazes de manter aplicações maiores e *softwares* de roteamento isolados e com configurações

[8]Documentação em `https://docs.python.org/2/library/simplehttpserver.html`.

distintas.

Para resolver essa questão do isolamento de processos, configurou-se a extensão MiniNExT[9] sobre o Mininet convencional. Essa extensão adiciona o suporte a espaços de PID e de usuários separados para cada *host*, permitindo que aplicações iguais executem com configurações diferentes em cada *host*. O uso do MiniNExT possibilitou que cada *host* da rede virtual fosse levantado com uma instância de roteamento Quagga devidamente configurada de forma a atuar, em cada um, como um AS distinto. Em contrapartida, a extensão não sofre atualizações há mais de um ano até a confecção deste trabalho e requereu que a versão menor do Mininet fosse reduzida da 2.3.0 para a 2.1.0, o que não causou impacto nos testes.

## 4.3 Desafios acerca dos componentes

Os componentes utilizados na arquitetura desempenham, cada um, funções fundamentais para o correto funcionamento do sistema como um todo. A ausência deles implicaria em implementar suas funções dentro do cerne da arquitetura, que é a biblioteca Havox. Almeja-se, porém, a absorção de algumas funções dos componentes em trabalhos futuros, conforme explicado no capítulo final, a fim de tornar a arquitetura Havox robusta e autossuficiente caso os componentes se tornem obsoletos e parem de ser atualizados. Não necessariamente essa absorção seria na forma de implementação de novas funcionalidades da biblioteca, o que a inflaria e iria contra os preceitos da Engenharia de *Software* [50], mas na forma de novas bibliotecas e serviços complementares.

Um dos desafios deste trabalho foi aprender sobre o funcionamento do RouteFlow e do Merlin, visto que essas ferramentas também são experimentais e ainda não são devidamente documentadas. Foram realizadas múltiplas releituras minuciosas dos artigos publicados referentes às ferramentas e um longo processo de engenharia reversa dos códigos-fonte. Especialmente no caso do Merlin, outra etapa longa foi a instalação e configuração do ambiente de execução, visto que no início da pesquisa as instruções de instalação não eram claras. Uma das medidas adotadas foi o contato direto com os autores da ferramenta, formando um canal que possibilitou a resolução dos problemas de configuração encontrados.

Outro desafio também relacionado ao Merlin foi quanto ao aprendizado da sua

[9] Disponível em `http://mininext.uscnsl.net/`.

DSL. Como não há ainda documentação sobre a sintaxe, o processo de engenharia reversa acabou sendo muito longo e revelou inclusive a existência de muitos *bugs* associados a palavras-chave existentes nos analisadores léxico e sintático do compilador, que geram exceções mesmo sendo previstas. Os próprios exemplos de código obtidos a partir dos artigos publicados não são plenamente funcionais, o que indica que o código-fonte do compilador sofreu grandes mudanças estruturais desde a sua publicação, mas que não foram devidamente documentadas.

O Merlin vem recebendo novas atualizações de código de um ano até a confecção deste trabalho, então espera-se que a instabilidade do projeto seja amenizada com o tempo. O RouteFlow, porém, não recebe novas atualizações dos seus autores há mais de três anos. Por isso, um dos objetivos a ser atingido em prol da arquitetura Havox como trabalho futuro é contribuir para o código do RouteFlow ou se tornar independente deste.

Na verdade, a contribuição para o código-fonte do RouteFlow já está sendo realizada, com a implementação do suporte ao atributo de ID de VLAN (`vlan_id`) e às ações de definição e de remoção de ID de VLAN (`set_vlan_id` e `strip_vlan`). Foi implementado também o suporte a correspondência pelo atributo de endereço de IP de origem. Essas contribuições de código demandaram um certo tempo de engenharia reversa, análise de impacto e implementação, a fim de minimizar instabilidades decorrentes da intervenção. No entanto, foram essenciais para que a biblioteca Havox pudesse entregar ao usuário o controle também sobre esses atributos, permitindo que seja feita a orquestração de tráfego baseado na origem.

## 4.4 Síntese

Este capítulo tratou dos detalhes de implementação e implantação dos diversos componentes da pilha de aplicações da arquitetura Havox, passando pelos desafios técnicos que foram suplantados, pelas versões dos recursos de *software*, pela descrição dos recursos de *hardware* e pelas decisões tomadas ao longo do processo. O exposto neste capítulo fornece o embasamento técnico da proposta e dos conceitos abordados no capítulo anterior e prepara para o entendimento do experimento de validação, que será tratado no próximo capítulo.

Em suma, neste capítulo foi explicado que a principal contribuição deste trabalho consiste de dois projetos separados, uma aplicação *web* que atua como API e invoca a biblioteca Havox, que guarda toda a lógica implementada e se comu-

nica com o Merlin e com o RouteFlow. Toda a arquitetura opera sobre a camada de dados controlada pelo RouteFlow, mas mantida fisicamente pelo emulador Mininet. O Merlin, o RouteFlow e o Mininet estão dispostos cada qual em uma máquina virtual própria e gerenciada pela máquina hospedeira, que executa o Havox como um serviço remoto. A comunicação entre o Havox e seus componentes se dá por conexões SSH, cujas credenciais de acesso são fornecidas via bloco de configurações.

# 5. Validação experimental

Embasado pela proposta e pelos detalhes de implementação descritos nos capítulos anteriores, este capítulo aborda o cenário de teste e os resultados obtidos, bem como discute os resultados, elenca as limitações da arquitetura proposta e indica possíveis soluções para suplantar as limitações.

## 5.1 Cenário de teste

A arquitetura foi validada em um cenário não *full-mesh* composto por quatro *switches* OpenFlow e sete *hosts*, todos elementos virtualizados no ambiente de emulação Mininet. A nível físico subjacente, os quatro *switches*, *s5*, *s6*, *s7* e *s8*, estão interligados cada um a um *host* adjacente, *h1*, *h2*, *h3* e *h4*, respectivamente. Há ainda outros *hosts*, *h5*, *h6* e *h7*, os quais o primeiro está conectado a *h2* e *h4* e os dois últimos, a *h1*. Vide Figura 5.1.

A nível lógico, os quatro *switches* são representados na camada virtual por suas instâncias de roteamento Quagga e juntos integram um único AS, considerado neste cenário o principal. Cada um dos *hosts* também executa uma instância de roteamento. Os *hosts* são análogos a roteadores de borda, os quais cada um representa um AS distinto. No cenário ilustrado pela Figura 5.1, como os *hosts h1*, *h2*, *h3* e *h4* são adjacentes aos *switches* na camada de dados, na abstração superior os ASes 1000, 2000, 3000 e 4000 são vizinhos ao AS 1, com os ASes 5000 (*h5*), 6000 (*h6*) e 7000 (*h7*) sendo remotos.

Cada AS do cenário de testes divulga rotas próprias, que eventualmente chegam ao AS 1 por anúncios BGP e são instaladas nas RIBs das instâncias de roteamento da camada virtual do RouteFlow. Os prefixos anunciados pelos diversos *daemons* BGP do cenário de teste e as interfaces das conexões são mostrados na

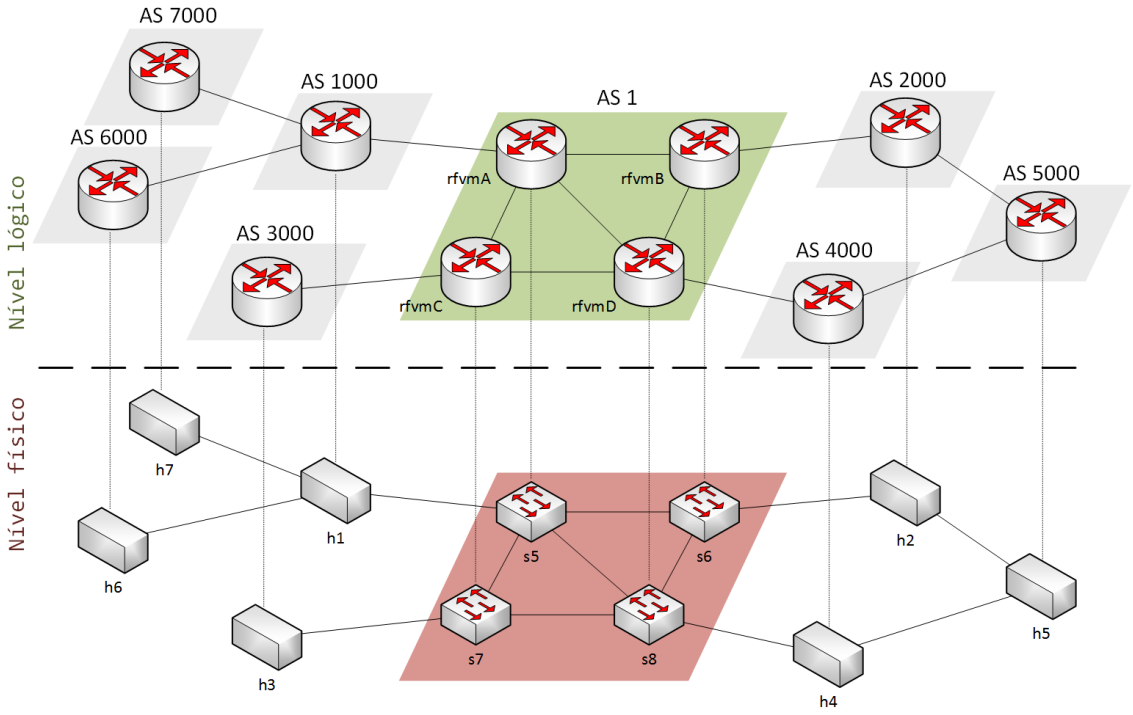



Figura 5.1: Disposição dos elementos de rede no cenário de testes.

Figura 5.2. Os *daemons* estão operando com configuração BGP simples, isto é, apenas obtendo rotas e repassando-as para os vizinhos.

Uma vez instaladas nas RIBs individuais, as rotas são coletadas e estruturadas pelo Havox. Elas são mantidas em uma estrutura de RIB global, a partir da qual é possível obter uma lista de redes alcançáveis pelo domínio e listar todas as rotas conhecidas para um determinado endereço IP. A Tabela 5.1 resume as informações acerca das rotas conhecidas no AS 1 do cenário de teste.

| Rede | AS | Via AS | Inst. rot. | *Switch* |
|---|---|---|---|---|
| 172.10/16 | 1000 | 1000 | *rfvmA* | *s5* |
| 172.60/16 | 6000 | 1000 | *rfvmA* | *s5* |
| 172.70/16 | 7000 | 1000 | *rfvmA* | *s5* |
| 172.20/16 | 2000 | 2000 | *rfvmB* | *s6* |
| 172.40/16 | 4000 | 2000 | *rfvmB* | *s6* |
| 172.50/16 | 5000 | 2000 | *rfvmB* | *s6* |
| 172.30/16 | 3000 | 3000 | *rfvmC* | *s7* |
| 172.20/16 | 2000 | 4000 | *rfvmD* | *s8* |
| 172.40/16 | 4000 | 4000 | *rfvmD* | *s8* |
| 172.50/16 | 5000 | 4000 | *rfvmD* | *s8* |

Tabela 5.1: Rotas BGP conhecidas pelo AS 1 do cenário.

A conectividade entre todos os ASes do cenário para o anúncio de rotas e o *parsing* e estruturação dessas rotas conhecidas pelo AS 1 são condições necessárias

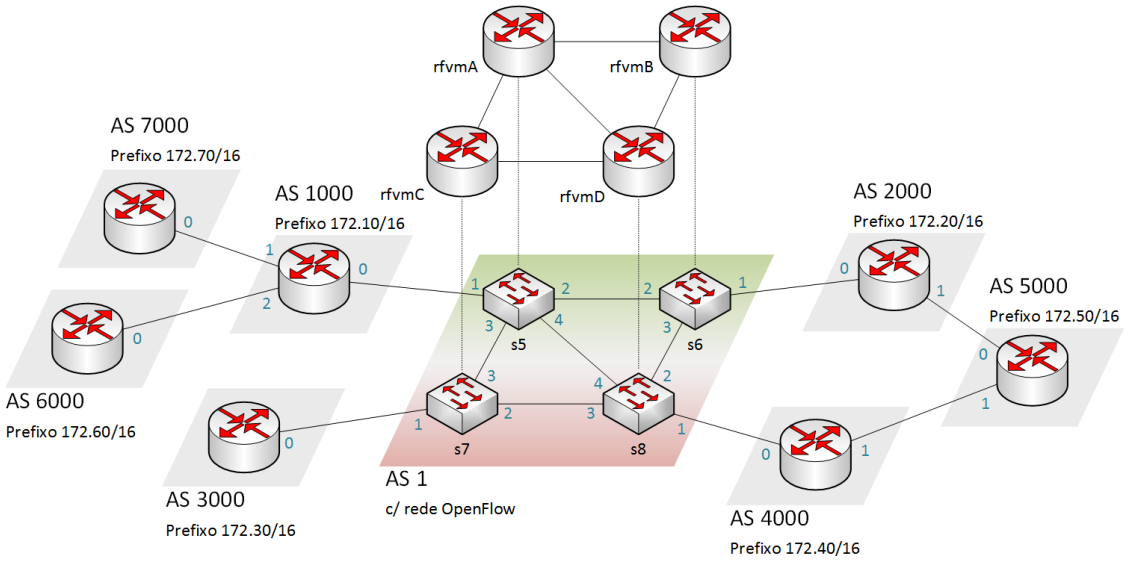



Figura 5.2: Prefixos de rede e interfaces dos ASes do cenário.

para a execução do processo de transcompilação das diretivas Havox em políticas Merlin e, posteriormente, em regras OpenFlow.

## 5.2 Execução

Todo o processo desde o envio da requisição à API do Havox até a instalação das regras OpenFlow especiais nas tabelas de fluxo dos *switches* é dividido em três etapas: *transcompilação*, *tratamento* e *instalação*.

### 5.2.1 Etapa de transcompilação

O processo de transcompilação se inicia a partir do momento em que o RouteFlow modificado envia uma requisição à API do Havox por meio do módulo RFHavox. A requisição método POST é enviada para a URL da aplicação *web* da API e contém os arquivos de diretivas e de topologia, conforme os Códigos 5.1 e 5.2, bem como os parâmetros e valores listados na Tabela 5.2. Todo o processo é resumido pela Figura 5.3.

As diretivas do arquivo "`routeflow.hvx`", definidas como prova de conceito, instruem o sistema a direcionar (1) para a saída via *switch s5* todos os pacotes para aplicações *web*, (2) para a saída via *switch s6* todos os pacotes com destino à rede 172.50/16, (3) para a saída via *switch s7* todos os pacotes originados na rede 172.70/16 e (4) para a saída via *switch s8* todos os pacotes FTP de dados com

```
topology 'routeflow.dot'

exit(:s5) { destination_port 80 }
exit(:s6) { destination_ip '172.50.0.0/16' }
exit(:s7) { source_ip '172.70.0.0/16' }
exit(:s8) {
  destination_port 20
  destination_ip '172.50.0.0/16'
}
```

Código 5.1: Diretivas Havox no arquivo "routeflow.hvx".

```
digraph g1 {
  h1 [type = host, ip = "172.31.1.100"];
  h2 [type = host, ip = "172.31.2.100"];
  h3 [type = host, ip = "172.31.3.100"];
  h4 [type = host, ip = "172.31.4.100"];

  s5 [type = switch, ip = "172.31.1.1", id = 5];
  s6 [type = switch, ip = "172.31.2.1", id = 6];
  s7 [type = switch, ip = "172.31.3.1", id = 7];
  s8 [type = switch, ip = "172.31.4.1", id = 8];

  s5 -> h1 [src_port = 1, dst_port = 1, cost = 1];
  h1 -> s5 [src_port = 1, dst_port = 1, cost = 1];

  s6 -> h2 [src_port = 1, dst_port = 1, cost = 1];
  h2 -> s6 [src_port = 1, dst_port = 1, cost = 1];

  s7 -> h3 [src_port = 1, dst_port = 1, cost = 1];
  h3 -> s7 [src_port = 1, dst_port = 1, cost = 1];

  s8 -> h4 [src_port = 1, dst_port = 1, cost = 1];
  h4 -> s8 [src_port = 1, dst_port = 1, cost = 1];

  s5 -> s6 [src_port = 2, dst_port = 2, cost = 1];
  s5 -> s7 [src_port = 3, dst_port = 3, cost = 1];
  s5 -> s8 [src_port = 4, dst_port = 4, cost = 1];

  s6 -> s5 [src_port = 2, dst_port = 2, cost = 1];
  s6 -> s8 [src_port = 3, dst_port = 2, cost = 1];

  s7 -> s5 [src_port = 3, dst_port = 3, cost = 1];
  s7 -> s8 [src_port = 2, dst_port = 3, cost = 1];

  s8 -> s5 [src_port = 4, dst_port = 4, cost = 1];
  s8 -> s6 [src_port = 2, dst_port = 3, cost = 1];
  s8 -> s7 [src_port = 3, dst_port = 2, cost = 1];
}
```

Código 5.2: Descrição da topologia subjacente no arquivo "routeflow.dot".

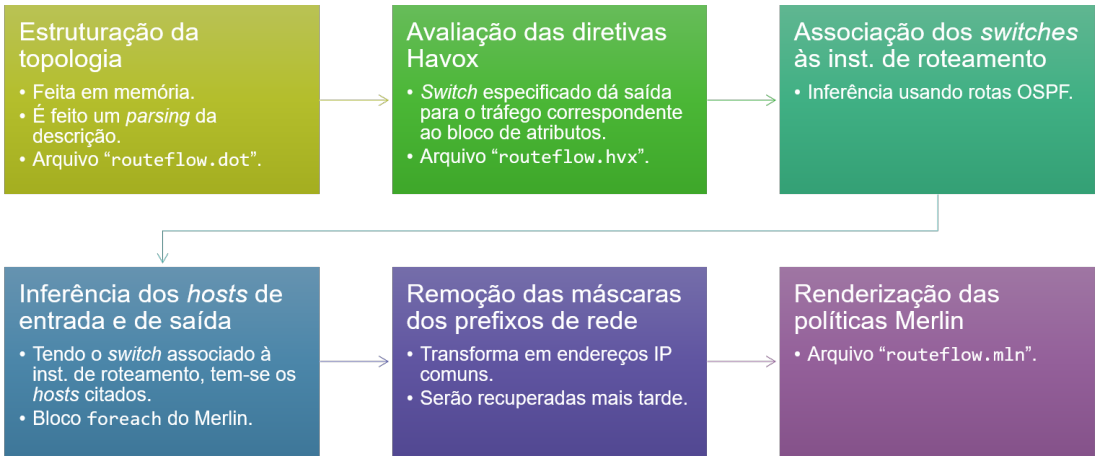



Figura 5.3: Processos da etapa de transcompilação das regras.

destino à rede 172.50/16.

| Parâmetro | Valor | Razão |
|---|---|---|
| `force` | `'true'` | Nos casos de múltiplas definições de um mesmo atributo pelo Merlin, o último valor é o válido. |
| `expand` | `'false'` | Não é necessária a expansão das regras porque o RouteFlow foi aprimorado para suportar VLAN. |
| `output` | `'true'` | O RouteFlow não suporta enfileiramento e QoS, portanto deve-se trocar a ação `enqueue` por `output`. |
| `syntax` | `'routeflow'` | A API deve responder com regras traduzidas para a sintaxe do RouteFlow. |

Tabela 5.2: Parâmetros textuais da requisição para a API do Havox.

O arquivo de topologia "`routeflow.dot`"propositalmente não descreve *hosts* associados a ASes remotos. É suficiente indicar apenas os *hosts* vizinhos porque é através da leitura das arestas entre *switches* e *hosts* que o Havox infere os *switches* de borda do AS 1. Isso também mostra que abstrai-se da conectividade entre ASes vizinhos e remotos, importando apenas indicar aqueles que se conectam com o domínio. Extrapolando o cenário de teste, se um AS vizinho se conectasse ao AS 1 por múltiplas conexões e roteadores de borda, o mesmo número de arestas e de *hosts* deveria ser descrito no arquivo.

Os arquivos são repassados à biblioteca Havox quando são recebidos pela API. Inicia-se então o processo de avaliação dos arquivos, a começar pelas diretivas. Por serem blocos de código Ruby disfarçados por metaprogramação, cada diretiva é estruturada em objetos *Directive*, que guardam o tipo de diretiva, o *switch* de saída definido e os atributos que devem compor o predicado de correspondência das regras OpenFlow. Como prova de conceito, o único tipo de diretiva até então

implementado que lida com pacotes no domínio é o tipo `exit`, que direciona o tráfego correspondente para a saída especificada.

A topologia é estruturada em memória após a avaliação do arquivo de topologia, permitindo que o sistema saiba quais são os *switches* de saída e os *hosts* representantes dos ASes vizinhos. Uma vez que se tem essas informações, torna-se trivial separar os *hosts* de origem e os *hosts* de destino baseados no *switch* de saída, conforme a sintaxe do Merlin exige para realizar a compilação. A exemplo da diretiva número 1 (sair por *s5* todo tráfego para aplicações *web*), sabe-se que *s5* tem conexão com o *host h1*, isto é, com o AS 1000. Logo, a política Merlin derivada dessa diretiva terá os *hosts* de origem sendo *h2*, *h3* e *h4*, e o de destino, o *h1*.

Munido dos objetos de diretiva e das conexões entre *switches* de saída e *hosts*, o sistema inicia a etapa de transcompilação em políticas Merlin. Cada diretiva Havox derivará em uma política Merlin. Os nomes dos atributos das diretivas são convertidos para os nomes análogos na sintaxe do Merlin, conforme a Tabela 5.3.

| Nome Havox | Nome Merlin |
|---|---|
| `destination_ip` | `ipDst` |
| `destination_mac` | `ethDst` |
| `destination_port` | `tcpDstPort` |
| `ethernet_type` | `ethTyp` |
| `in_port` | `port` |
| `ip_protocol` | `ipProto` |
| `source_ip` | `ipSrc` |
| `source_mac` | `ethSrc` |
| `source_port` | `tcpSrcPort` |
| `vlan_id` | `vlanId` |
| `vlan_priority` | `vlanPcp` |

Tabela 5.3: Nome dos atributos nas diretivas Havox e os análogos Merlin.

Em seguida na transcompilação, se o atributo em avaliação for referente ao IP de origem ou ao IP de destino, a máscara de sub-rede do endereço é removida, deixando apenas o IP estático. Isso é necessário porque o Merlin não é atualmente projetado para lidar com endereços completos de rede. Mais adiante no processo, a máscara será restabelecida com base na lista de redes alcançáveis conhecidas.

Feitos a tradução dos nomes dos atributos e o tratamento dos valores de endereçamento IP, os conjuntos de pares atributo-valor são concatenados com um conectivo conjuntivo "`and`", formando assim um predicado de correspondência do Merlin. O *switch* de saída definido para a diretiva é concatenado após o trecho

```
1 foreach (s, d): cross({ h2; h3; h4 }, { h1 })
2   tcpDstPort = 80 -> .* s5 at min(100 Mbps);
3
4 foreach (s, d): cross({ h1; h3; h4 }, { h2 })
5   ipDst = 172.50.0.0 -> .* s6 at min(100 Mbps);
6
7 foreach (s, d): cross({ h1; h2; h4 }, { h3 })
8   ipSrc = 172.70.0.0 -> .* s7 at min(100 Mbps);
9
10 foreach (s, d): cross({ h1; h2; h3 }, { h4 })
11   tcpDstPort = 20 and ipDst = 172.50.0.0 -> .* s8 at min(100 Mbps);
```

Código 5.3: Arquivo de políticas "`routeflow.mln`"resultante da transcompilação.

de caracteres "`.*`", formando um caminho em expressão regular que termina no *switch*. Esse caminho em expressão regular é concatenado com uma expressão de QoS própria do Merlin, caso presente, que pode ser expresso na requisição. Por limitações atuais do Merlin, não colocar uma expressão de QoS faz com que o compilador do *framework* levante um erro. Portanto, uma expressão arbitrária "`min(100 Mbps)`"é usada, mas a mesma não modifica o comportamento final porque as ações de enfileiramento mais adiante serão convertidas em ações de saída convencional. Se a arquitetura implementasse QoS, essa expressão serviria para garantir um mínimo de 100 Mbps de vazão no encaminhamento.

O predicado é concatenado com o resultado da concatenação entre o caminho em expressão regular e a expressão de QoS, usando um conectivo "`->`" e formando uma expressão Merlin. No exemplo da diretiva número 1, a expressão Merlin resultante é "`tcpDstPort = 80 -> .* s5 at min(100 Mbps);`". Essa expressão é aninhada sozinha em um bloco "`foreach`"do Merlin, que define que para cada par de *hosts* origem-destino especificado, aquela expressão será compilada. Os *hosts* de origem e de destino são inferidos conforme citado anteriormente, a partir do *switch* de saída. O bloco de código resultante é uma política Merlin.

Esses passos da transcompilação são executados para cada diretiva Havox. Ao final do processo, um conjunto de políticas Merlin será gerado, uma para cada diretiva. A partir dos Códigos 5.1 e 5.2, e dos parâmetros da requisição especificados na Tabela 5.2, será gerado um arquivo de políticas Merlin "`routeflow.mln`" contendo o Código 5.3.

Observa-se no código resultante que os cabeçalhos dos blocos `foreach` contêm ambos os conjuntos de *hosts* de origem (o primeiro) e de destino (o segundo) inferidos a partir dos *switches* de saída. Nota-se também que as máscaras de sub-

rede foram removidas dos valores de endereço IP, o que permitirá que o processo de compilação interno do Merlin execute sem problemas.

O arquivo de políticas "`routeflow.mln`" gerado é submetido ao compilador do Merlin, junto com o arquivo de topologia "`routeflow.dot`". Este último, diga-se de passagem, é um parâmetro necessário no processo de compilação do Merlin. O Havox apenas realiza um *parsing* do seu conteúdo para executar a sua lógica de transcompilação, mas não faz nenhuma alteração no mesmo.

A vantagem obtida até este momento final da etapa de transcompilação com o uso das diretivas Havox em relação à escrita direto em sintaxe Merlin é que o operador não precisa manualmente definir os pontos de entrada e de saída de pacotes na rede para cada expressão, haja vista que o arquivo de topologia já contém informações que permitem essa inferência de *hosts*. Ademais, o uso de diretivas Havox permite que o operador expresse os valores dos campos dinamicamente com o uso de funções anônimas, conceito explicado e exemplificado no Capítulo 3. Justifica-se, então, o uso da diretivas Havox devido à abstração dos detalhes citados, permitindo o foco do operador na orquestração do tráfego.

### 5.2.2 Etapa de tratamento

Depois que o Merlin compila as políticas em regras OpenFlow primitivas e as imprime em saída padrão, tem início a etapa de tratamento e estruturação dessas regras, resumida pelas Figuras 5.4 e 5.5. A Figura 5.6 mostra um terminal contendo parte das regras OpenFlow primitivas impressas. Neste cenário de testes, 26 regras são geradas.


Figura 5.4: Processos da etapa de tratamento durante o *parsing* das regras.

É possível observar que todas as regras primitivas ilustradas pela Figura

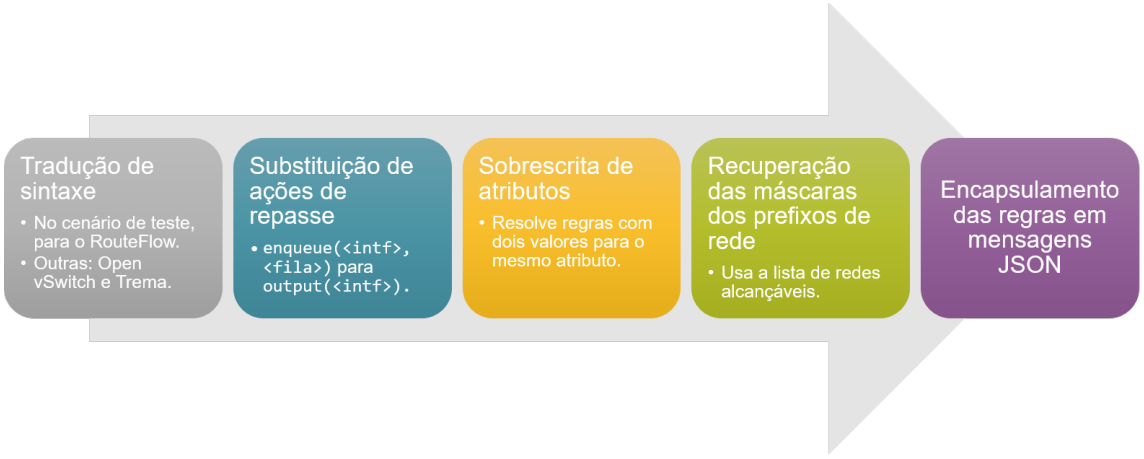



Figura 5.5: Processos da etapa de tratamento durante a estruturação das regras.

```
OpenFlow rules (26):

On switch 6     (switch = 6
and (((ipSrc = -1407253916 and ipDst = -1407254172))
    and (nwProto = 6 and tcpDstPort = 80))) -> SetField(vlan, 12)       Enqueue(2,2)
On switch 5     (switch = 5 and vlanId = 12) -> SetField(vlan, <none>)  Enqueue(1,3)
On switch 8     (switch = 8 and vlanId = 11) -> SetField(vlan, <none>)  Enqueue(1,3)
On switch 5     (switch = 5
and (((ipSrc = -1407254172 and ipDst = -1407253404))
    and (((nwProto = 6 and tcpDstPort = 20))
        and (ethTyp = 2048 and ipDst = -1406009344)))) -> SetField(vlan, 11)    Enqueue(4,1)
On switch 6     (switch = 6
and (((ipSrc = -1407253916 and ipDst = -1407253404))
    and (((nwProto = 6 and tcpDstPort = 20))
        and (ethTyp = 2048 and ipDst = -1406009344)))) -> SetField(vlan, 10)    Enqueue(3,1)
On switch 8     (switch = 8 and vlanId = 10) -> SetField(vlan, <none>)  Enqueue(1,2)
```

Figura 5.6: Regras OpenFlow primitivas impressas em saída padrão.

5.6 contêm ações de enfileiramento (`enqueue`) para as correspondências. Mais ainda, algumas das regras têm atributos que sofrem múltiplas definições de valores, como o atributo `ipDst`. Tomando esse atributo como exemplo, em algumas ocorrências observa-se que a ele são definidos os valores -1407253404 e -1406009344, que são os números inteiros correspondentes aos endereços IP 172.31.4.100 e 172.50.0.0, na devida ordem, em complemento a dois. O endereço IP 172.31.4.100 é o definido no arquivo de topologia para o *host h4* e o endereço IP 172.50.0.0 é o definido como correspondência nas políticas.

Esse fenômeno da múltipla definição de atributos ocorre nas regras compiladas de políticas onde há o atributo de correspondência por endereço IP de origem ou, como neste caso, de destino. Isso acontece porque o Merlin sempre adiciona os atributos de endereço IP de origem e de destino referentes aos *hosts* de origem e de destino às regras compiladas de blocos `foreach`. Quando o usuário adiciona uma correspondência por um desses atributos, o compilador acaba duplicando a correspondência, uma com o valor do endereço IP do *host* e outra com o valor definido pelo usuário, em vez de dar preferência ao do usuário.

As regras primitivas são obtidas pelo Havox através do *parsing* da saída padrão com o uso de expressões regulares. Para cada regra primitiva, é identificado o ID do *switch* ao qual ela é destinada e são avaliados suas correspondências com valores e suas ações com argumentos.

Após o *parsing*, o tratamento das regras seguirá conforme os parâmetros passados na requisição. Como foi especificado o parâmetro para forçar a redefinição de atributos referente ao fenômeno supracitado, o Havox irá considerar o último valor definido como válido, ou seja, o endereço IP 172.50.0.0 como destino. Como foi especificado também o parâmetro para a troca da ação de enfileiramento pela ação de saída convencional, a ação `enqueue` será substituída pela ação `output`. O primeiro argumento da ação `enqueue` é a interface de saída do *switch* e será usado como argumento único da ação `output`. O segundo argumento da ação `enqueue` é o ID da fila e, como não há filas e nem QoS em uso, será descartado.

Os nomes dos atributos de correspondência e das ações das regras passam em seguida por uma tradução para uma sintaxe conhecida. Neste cenário de teste, é usada a sintaxe do RouteFlow, que irá traduzir os termos do Merlin para termos legíveis pelo RouteFlow, conforme a Tabela 5.4. Alguns atributos não são ainda implementados pelo RouteFlow, portanto seus análogos na tabela foram omitidos.

| Nome Merlin | Nome RouteFlow |
|---|---|
| `ipDst` *(atr)* | `ipv4` |
| `ethDst` *(atr)* | `ethernet` |
| `tcpDstPort` *(atr)* | `tp_dst` |
| `ethTyp` *(atr)* | `ethertype` |
| `port` *(atr)* | – |
| `ipProto` *(atr)* | `nw_proto` |
| `ipSrc` *(atr)* | `ipv4_src` |
| `ethSrc` *(atr)* | – |
| `tcpSrcPort` *(atr)* | `tp_src` |
| `vlanId` *(atr)* | `vlan_id` |
| `vlanPcp` *(atr)* | – |
| `Output` *(aç)* | `output` |
| `SetField(vlan, id)` *(aç)* | `set_vlan_id` |
| `SetField(vlan, <none>)` *(aç)* | `strip_vlan` |

Tabela 5.4: Nome dos atributos e ações Merlin e os análogos no RouteFlow.

Depois que os atributos e as ações são traduzidos, é iniciado o tratamento dos atributos de endereço IP de origem e de destino, com o objetivo de devolver as máscaras de sub-rede que foram removidas na etapa de transcompilação e de

eliminar as correspondências de endereço IP que foram adicionadas pelo Merlin por padrão a todas as regras.

Todas as regras são iteradas e cada uma tem os valores dos seus atributos de endereço IP de origem e de destino avaliados. Para tanto, o sistema conta com a lista de prefixos de rede alcançáveis obtida das RIBs das instâncias de roteamento.

Quando um endereço IP de origem ou de destino é avaliado, os prefixos de rede da lista de redes alcançáveis são iterados a fim de se encontrar aquele cuja faixa abrange o endereço IP corrente. Por exemplo, a faixa de endereçamento do prefixo de rede 172.50.0.0/16, constante na lista como anunciado pelo AS 5000, abrange o endereço IP 172.50.0.0. O Havox irá então substituir o valor do atributo de endereço IP de destino de 172.50.0.0 para 172.50.0.0/16 na regra corrente.

O mesmo algoritmo também elimina correspondências de endereço IP inseridas por padrão pelo Merlin. Ao confrontar um endereço IP de origem 172.31.4.100 contra a lista de redes alcançáveis, não será encontrada nenhuma rede cuja faixa abranja esse endereço. Para o Havox, esse endereço é então considerado inválido. O valor do atributo de endereço IP de origem inválido é marcado como nulo e posteriormente será eliminado. A Figura 5.7 ilustra o funcionamento dessa lógica.

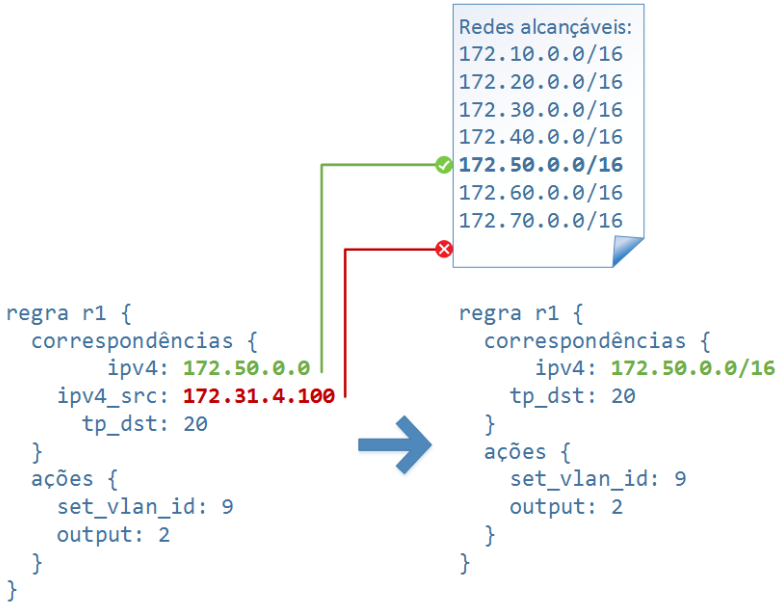



Figura 5.7: Recuperação das máscaras de sub-rede dos endereços IP e eliminação dos endereços inválidos.

A eliminação de atributos de endereços IP de origem e de destino inválidos das regras é de suma importância. A razão é que os pacotes que trafegarão pelo domínio do AS não são apenas originados nos *hosts* vizinhos, mas de qualquer

lugar da *Internet* e para qualquer lugar da *Internet*. Se esses atributos não fossem devidamente tratados ou eliminados, não seria possível orquestrar o encaminhamento porque todas as regras OpenFlow corresponderiam a endereços IP de origem e de destino de vizinhos, e os pacotes que vêm e vão de e para a *Internet* possuem endereços de origem e de destino mais variados possíveis.

Terminados a estruturação e o tratamento das 26 regras primitivas geradas pelo Merlin para o cenário de teste, o conjunto de regras, agora especiais, é entregue da biblioteca para a API, onde será encapsulado em uma mensagem JSON e entregue ao RouteFlow como resposta da requisição.

### 5.2.3 Etapa de instalação

A etapa de instalação das regras se inicia quando o módulo RFHavox do RouteFlow recebe a resposta da requisição enviada à API do Havox. A mensagem JSON vinda na resposta é desencapsulada e as 26 regras OpenFlow especiais para este cenário de teste são obtidas. A Figura 5.8 resume os processos desta etapa.

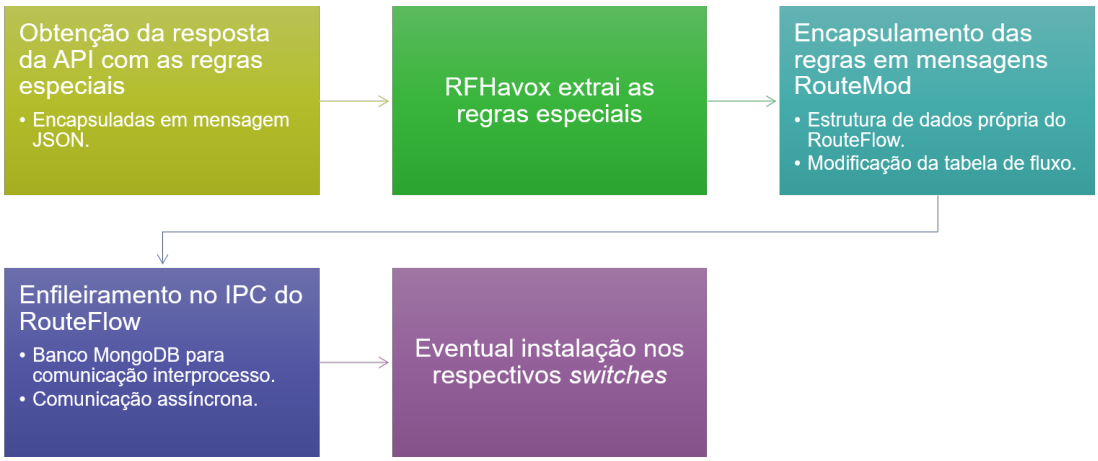



Figura 5.8: Processos da etapa de instalação das regras.

O módulo RFHavox irá iterar as 26 regras especiais e criar, para cada uma, objetos *RouteMod*. Esse objeto é uma mensagem de modificação de tabelas de fluxo do RouteFlow e encapsula as regras OpenFlow a serem instaladas. Como prova de conceito, apenas objetos RouteMod do tipo *adição* são usados, porém existem também os tipos *remoção* e *modificação*.

Os objetos RouteMod contêm, além dos atributos com valores e das ações a serem tomadas, opções de instalação como ID do controlador e a prioridade da regra. Como neste trabalho é usado apenas um controlador OpenFlow, o ID usado é o padrão do RouteFlow. Já a prioridade estabelecida para as regras especiais é a

máxima possível da plataforma. O intuito é que as regras OpenFlow criadas pelo operador através do Havox tenham precedência sobre quaisquer outras regras básicas geradas pelo processo do RouteFlow. A razão é que as regras especiais refletem as decisões de âmbito político do AS, como os SLAs e estratégias de negócio.

As mensagens RouteMod que encapsulam cada regra são posteriormente enviadas ao serviço IPC do RouteFlow, isto é, ao banco MongoDB da plataforma usado para a comunicação assíncrona entre todos os módulos. As mensagens enfileiradas no serviço IPC serão eventualmente consumidas pelo RFProxy, que extrairá as regras e as instalará nas tabelas de fluxo dos respectivos *switches* identificados pelos seus IDs no ambiente do Mininet.

## 5.3 Resultados

O resultado de todas as etapas citadas na seção anterior são as 26 regras OpenFlow especiais instaladas juntamente com as demais regras do RouteFlow. Uma consulta às tabelas de fluxo dos *switches* usando o comando "`ovs-ofctl dump-flows <switch>`" revela a presença das regras especiais. As Tabelas 5.5, 5.6, 5.7 e 5.8 listam as regras transcritas da saída padrão do comando para cada *switch*. A primeira coluna de cada tabela é uma informação auxiliar e referencia a diretiva Havox do Código 5.1 que serviu de base para a regra.

| D | Predicado | Ações |
|---|---|---|
| 4 | `tcp, nw_dst=172.50.0.0/16, tp_dst=20` | `mod_vlan_vid:11, output:4` |
| 3 | `ip, nw_src=172.70.0.0/16` | `mod_vlan_vid:3, output:3` |
| 2 | `ip, nw_dst=172.50.0.0/16` | `mod_vlan_vid:6, output:2` |
| 2 | `dl_vlan=5` | `output:2` |
| 3 | `dl_vlan=2` | `output:3` |
| 1 | `dl_vlan=7` | `strip_vlan, output:1` |
| 1 | `dl_vlan=12` | `strip_vlan, output:1` |
| 1 | `dl_vlan=8` | `strip_vlan, output:1` |

Tabela 5.5: Regras especiais do *switch s5* listadas no ambiente do Mininet.

Como é possível observar, cada fluxo é rotulado com um ID de VLAN usado apenas no encaminhamento interno. Recapitulando o que foi explicado em capítulos anteriores, quando um pacote ingressante no AS 1 por um *switch* corresponde a uma regra, ele é rotulado com um ID de VLAN único e encaminhado pela saída devida. No próximo *switch*, apenas o seu ID de VLAN será avaliado, o que levará o *switch* a remover o ID e encaminhar pelo enlace de saída para outro

| D | Predicado | Ações |
|---|---|---|
| 4 | tcp, nw_dst=172.50.0.0/16, tp_dst=20 | mod_vlan_vid:10, output:3 |
| 1 | tcp, tp_dst=80 | mod_vlan_vid:12, output:2 |
| 3 | ip, nw_src=172.70.0.0/16 | mod_vlan_vid:2, output:2 |
| 2 | dl_vlan=5 | strip_vlan, output:1 |
| 2 | dl_vlan=4 | strip_vlan, output:1 |
| 2 | dl_vlan=6 | strip_vlan, output:1 |

Tabela 5.6: Regras especiais do *switch s6* listadas no ambiente do Mininet.

| D | Predicado | Ações |
|---|---|---|
| 4 | tcp, nw_dst=172.50.0.0/16, tp_dst=20 | mod_vlan_vid:9, output:2 |
| 1 | tcp, tp_dst=80 | mod_vlan_vid:8, output:3 |
| 2 | ip, nw_dst=172.50.0.0/16 | mod_vlan_vid:5, output:3 |
| 3 | dl_vlan=3 | strip_vlan, output:1 |
| 3 | dl_vlan=2 | strip_vlan, output:1 |
| 3 | dl_vlan=1 | strip_vlan, output:1 |

Tabela 5.7: Regras especiais do *switch s7* listadas no ambiente do Mininet.

AS vizinho ou a repassar o pacote para outro *switch* do AS 1. No cenário de teste foram criados então 11 fluxos de pacotes distintos, visto que foram usados 11 IDs de VLAN. A Figura 5.9 ilustra quatro desses fluxos, cada um para um *switch* de borda.

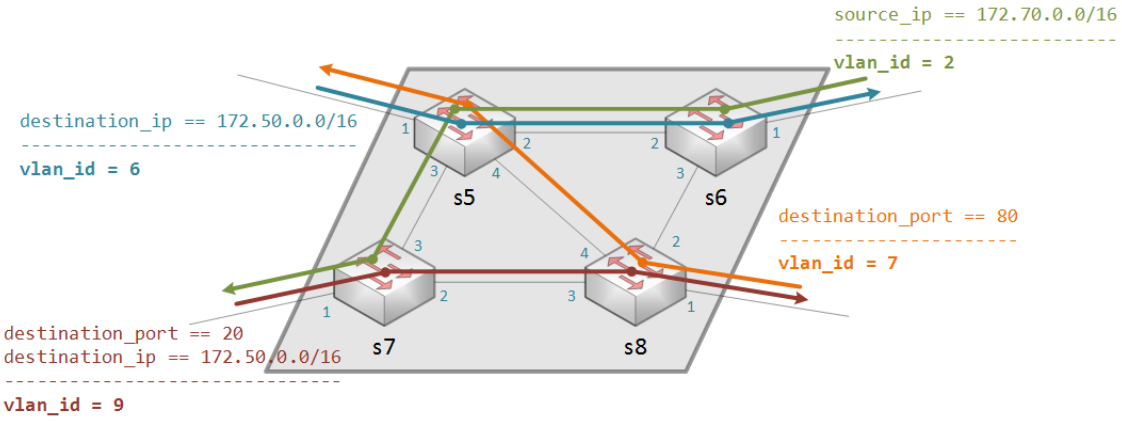



Figura 5.9: Fluxos de pacotes desde o ingresso no AS 1 até a saída do mesmo.

É importante destacar que a validade das regras é mantida durante as etapas de execução. A maior contribuição do Merlin para a arquitetura Havox é a criação de conjuntos de regras para cada fluxo, identificados pelo atributo de ID de VLAN. Essa característica faz com que os demais atributos do pacote sejam avaliados apenas quando ingressam na rede, que é quando o Havox realiza os tratamentos necessários. Como há manipulação apenas nas primeiras regras que correspondem aos fluxos, não há o risco de se gerar inconsistências, uma vez que

| D | Predicado | Ações |
|---|---|---|
| 1 | `tcp, tp_dst=80` | `mod_vlan_vid:7, output:4` |
| 3 | `ip, nw_src=172.70.0.0/16` | `mod_vlan_vid:1, output:3` |
| 2 | `ip, nw_dst=172.50.0.0/16` | `mod_vlan_vid:4, output:2` |
| 4 | `dl_vlan=10` | `strip_vlan, output:1` |
| 4 | `dl_vlan=9` | `strip_vlan, output:1` |
| 4 | `dl_vlan=11` | `strip_vlan, output:1` |

Tabela 5.8: Regras especiais do *switch s8* listadas no ambiente do Mininet.

as regras seguintes dos fluxos avaliam apenas o ID de VLAN.

As correspondências dos fluxos de pacotes com as regras OpenFlow instaladas puderam ser comprovadas com testes de geração de tráfego usando a ferramenta iPerf[1]. A Figura 5.10 mostra uma captura de tela que exibe a saída padrão do comando "`ovs-ofctl dump-flows <switch>`" para cada *switch* do cenário.

```
$ sudo ovs-ofctl dump-flows s5 | grep 49200
n=6702.59s, table=0, n_packets=7, n_bytes=518, idle_age=6126, priority=49200,ip,nw_src=172.70.0.0/16 actions=mod_vlan_vid:3,output:3
n=6702.594s, table=0, n_packets=7, n_bytes=518, idle_age=6422, priority=49200,ip,nw_dst=172.50.0.0/16 actions=mod_vlan_vid:6,output:2
n=6702.593s, table=0, n_packets=7, n_bytes=534, idle_age=145, priority=49200,dl_vlan=5 actions=output:2
n=6702.603s, table=0, n_packets=7, n_bytes=534, idle_age=4935, priority=49200,dl_vlan=7 actions=strip_vlan,output:1
n=6702.589s, table=0, n_packets=0, n_bytes=0, idle_age=6702, priority=49200,dl_vlan=2 actions=output:3
n=6702.63s, table=0, n_packets=7, n_bytes=534, idle_age=493, priority=49200,dl_vlan=12 actions=strip_vlan,output:1
n=6702.625s, table=0, n_packets=7, n_bytes=534, idle_age=1916, priority=49200,dl_vlan=8 actions=strip_vlan,output:1
n=6702.629s, table=0, n_packets=7, n_bytes=518, idle_age=5259, priority=49200,tcp,nw_dst=172.50.0.0/16,tp_dst=20 actions=mod_vlan_vid:11,output:4
$
$
$ sudo ovs-ofctl dump-flows s6 | grep 49200
n=6708.383s, table=0, n_packets=7, n_bytes=518, idle_age=499, priority=49200,tcp,tp_dst=80 actions=mod_vlan_vid:12,output:2
n=6708.274s, table=0, n_packets=0, n_bytes=0, idle_age=6708, priority=49200,ip,nw_src=172.70.0.0/16 actions=mod_vlan_vid:2,output:2
n=6708.274s, table=0, n_packets=7, n_bytes=534, idle_age=150, priority=49200,dl_vlan=5 actions=strip_vlan,output:1
n=6708.274s, table=0, n_packets=0, n_bytes=0, idle_age=6708, priority=49200,dl_vlan=4 actions=strip_vlan,output:1
n=6708.275s, table=0, n_packets=7, n_bytes=534, idle_age=6428, priority=49200,dl_vlan=6 actions=strip_vlan,output:1
n=6708.28s, table=0, n_packets=0, n_bytes=0, idle_age=6708, priority=49200,tcp,nw_dst=172.50.0.0/16,tp_dst=20 actions=mod_vlan_vid:10,output:3
$
$
$ sudo ovs-ofctl dump-flows s7 | grep 49200
n=6711.488s, table=0, n_packets=7, n_bytes=518, idle_age=1925, priority=49200,tcp,tp_dst=80 actions=mod_vlan_vid:8,output:3
n=6711.487s, table=0, n_packets=7, n_bytes=518, idle_age=154, priority=49200,ip,nw_dst=172.50.0.0/16 actions=mod_vlan_vid:5,output:3
n=6711.486s, table=0, n_packets=7, n_bytes=534, idle_age=6135, priority=49200,dl_vlan=3 actions=strip_vlan,output:1
n=6711.486s, table=0, n_packets=0, n_bytes=0, idle_age=6711, priority=49200,dl_vlan=2 actions=strip_vlan,output:1
n=6711.482s, table=0, n_packets=0, n_bytes=0, idle_age=6711, priority=49200,dl_vlan=1 actions=strip_vlan,output:1
n=6711.489s, table=0, n_packets=7, n_bytes=518, idle_age=2178, priority=49200,tcp,nw_dst=172.50.0.0/16,tp_dst=20 actions=mod_vlan_vid:9,output:2
$
$
$ sudo ovs-ofctl dump-flows s8 | grep 49200
n=6715.158s, table=0, n_packets=7, n_bytes=518, idle_age=4947, priority=49200,tcp,tp_dst=80 actions=mod_vlan_vid:7,output:4
n=6715.144s, table=0, n_packets=0, n_bytes=0, idle_age=6715, priority=49200,ip,nw_src=172.70.0.0/16 actions=mod_vlan_vid:1,output:3
n=6715.148s, table=0, n_packets=0, n_bytes=0, idle_age=6715, priority=49200,ip,nw_dst=172.50.0.0/16 actions=mod_vlan_vid:4,output:2
n=6715.17s, table=0, n_packets=0, n_bytes=0, idle_age=6715, priority=49200,dl_vlan=10 actions=strip_vlan,output:1
n=6715.162s, table=0, n_packets=7, n_bytes=534, idle_age=2182, priority=49200,dl_vlan=9 actions=strip_vlan,output:1
n=6715.176s, table=0, n_packets=7, n_bytes=534, idle_age=5272, priority=49200,dl_vlan=11 actions=strip_vlan,output:1
```

Figura 5.10: Saída padrão exibindo correspondências de fluxos de pacotes com algumas das regras.

Na Figura 5.10, as regras estão agrupadas por *switch* e cada regra correspondida com os testes de geração de tráfego teve as métricas `n_packets` e `n_bytes` incrementadas com o número de pacotes e com o número de Bytes correspondidos, respectivamente. Os comandos da ferramenta iPerf executados estão listados na Tabela 5.9, juntamente com os *hosts* a partir do qual foram executados e com as diretivas do Código 5.1 que cada um testou.

Nos comandos listados, foram usados endereços IP dentro das faixas das redes divulgadas pelos ASes, o que mostra que as correspondências de endereços IP

[1]Disponível em `https://iperf.fr/`.

| D | *Host* | Comando iPerf |
|---|---|---|
| 1 | *h2* | `iperf -c 172.30.1.10 -p 80` |
| 1 | *h3* | `iperf -c 172.20.1.10 -p 80` |
| 1 | *h4* | `iperf -c 172.30.1.10 -p 80` |
| 2 | *h1* | `iperf -c 172.50.1.10 -p 81` |
| 2 | *h3* | `iperf -c 172.50.1.10 -p 81` |
| 3 | *h7** | `iperf -c 172.40.1.10 -p 81` |
| 4 | *h1* | `iperf -c 172.50.1.10 -p 20` |
| 4 | *h3* | `iperf -c 172.50.1.10 -p 20` |

Tabela 5.9: Lista de variações do comando iPerf usadas para testar as correspondências.

dentro de faixas funcionam conforme o previsto. Em alguns casos, foi usada uma porta de destino arbitrária para que não houvesse correspondência não intencionada com regras que preveem as portas 20 e 80.

O comando que testa a diretiva número 3 do Código 5.1 e é executado a partir do *host h7*, marcado com asterisco na Tabela 5.9, na verdade parte de um outro *host* levantado dentro da rede do AS 7000 no cenário, de forma que esse *host* esteja na mesma rede 172.70.0.0/16 que o *host h7*. Dessa forma, o tráfego originado desse *host* tem um endereço IP de origem dentro da faixa compreendida pela rede citada.

Algumas regras listadas na saída padrão exibida pela Figura 5.10 não tiveram pacotes correspondidos (`n_packets` igual a zero) porque os comandos que poderiam gerar pacotes que correspondessem a essas regras não foram executados, visto que o tráfego não passaria pelo domínio do AS 1. Um exemplo é a geração de tráfego originado dos *hosts h2* e *h4* para testar as diretivas número 2 e 4. O tráfego não passaria pelos *switches* do AS 1 porque a rede 172.50.0.0/16 é anunciada pelo AS 5000, vizinho tanto ao AS 2000 quanto ao 4000, o que faria com que o tráfego seguisse pelos enlaces diretos entre os referidos ASes.

## 5.4 Discussão

Juntamente com o cerne deste trabalho, que é a biblioteca Havox e a sua API, o tratamento das regras primitivas tão logo compiladas pelo Merlin e a adaptação do RouteFlow para o suporte a IDs de VLAN, a endereços IP de origem e a máscaras de sub-rede variadas permitem que a arquitetura proposta forneça um poder de controle de mais alto nível sobre a orquestração do tráfego dentro do

domínio de um AS, bem como aumenta a abstração acerca dos pormenores de conectividade e de operação da rede subjacente.

No campo teórico, a arquitetura proposta contempla o operador da rede com os benefícios aludidos, posto que as diretivas Havox foram processadas e transformadas em regras OpenFlow com sucesso no ambiente de experimentação do Mininet com *switches* virtuais. No entanto, no campo prático, em ambientes reais dentro de sistemas autônomos, há fatores que são apenas aproximados durante a experimentação e outros que acabam não sendo levados em conta.

### 5.4.1 Configuração do ambiente

Em um ambiente real, *switches* físicos com suporte ao protocolo OpenFlow seriam usados e máquinas dedicadas para cada componente da arquitetura se fariam necessárias, com possibilidade para aumento dos recursos computacionais de cada um conforme a demanda de processamento. Os enlaces entre esses equipamentos deveriam ser redundantes e os sistemas protegidos por soluções de alta disponibilidade, como o HAProxy, uma vez que a falha de comunicação com um dos processos pode comprometer todo o sistema. Uma configuração em ambiente real deve então levar em conta não apenas o custo-benefício da solução, mas também os planos de contenção e de contingência de crise, com avaliação de riscos envolvidos e melhores abordagens de solução, como o campo do gerenciamento de projetos sugere [50, 51].

Em âmbito de *software*, o uso da arquitetura Havox não isenta que os *daemons* BGP nas instâncias de roteamento Quagga sejam configurados, mesmo que minimamente. O Havox precisa ter acesso às rotas BGP conhecidas pelas instâncias de roteamento, e essas rotas são obtidas porque as sessões BGP com roteadores de vizinhos devem estar ativas e estabelecidas. O pareamento entre os ASes depende de uma série de fatores não apenas técnicos, mas comerciais, e envolve SLAs de troca de tráfego. A arquitetura Havox facilita a orquestração do tráfego interno ao adicionar o controle sobre outros atributos dos pacotes além de apenas o endereço de destino, mas ainda requer que as sessões BGP e os acordos entre os domínios estejam vigorando.

### 5.4.2 Desempenho e escalabilidade

Alguns pontos da arquitetura merecem atenção no que tange ao desempenho geral e à escalabilidade do sistema.

Um desses pontos se refere a forma como algumas funcionalidades da biblioteca Havox foram implementadas. A lista de redes alcançáveis, por exemplo, não é um problema em um cenário de teste como o que valida este trabalho. Porém, em um cenário real e contemporâneo, uma tabela BGP pode ter mais de 700 mil rotas [52]. Além do problema intrínseco ao crescimento exponencial das tabelas de roteamento [53], a manutenção de uma lista de redes alcançáveis pode requerer muito espaço de armazenamento e a iteração sequencial de uma lista como essa para verificar se um endereço IP está dentro da faixa de uma das redes conhecidas consumiria um tempo substancial, fatores que certamente inviabilizariam o uso da arquitetura proposta. Felizmente, há formas de mitigar os impactos ocasionados por esses fatores.

Com relação a essas questões, em vez da manutenção de uma lista de redes alcançáveis em memória, poderia-se utilizar um banco de dados não relacional como o MongoDB ou o Cassandra[2]. Bancos não relacionais já são soluções de mercado para lidar com quantidades massivas de dados no campo de *Big Data*. O problema do espaço e da iteração seria reduzido à manutenção da comunicação entre o Havox e o banco e ao tempo de processamento da lógica de verificação de pertinência de um endereço IP a uma faixa de rede.

Uma outra forma de melhorar essa verificação seria o armazenamento em banco dos prefixos de rede definidos nas diretivas antes da remoção de suas máscaras de sub-rede. Esse armazenamento seria acompanhado de um índice para rápida recuperação posterior. O próprio índice poderia ser um endereço IP arbitrário, único e sequencial, o que permitiria que o Merlin o processasse como um endereço convencional e depois o mesmo seria usado na recuperação do prefixo de rede original na etapa de tratamento.

Outro ponto que merece atenção é o que diz respeito à quantidade de regras OpenFlow a serem instaladas nos *switches*. No cenário testado, com quatro *switches*, foram criadas 26 regras especiais que, em média, significam 6 ou 7 regras instaladas nos devidos *switches*, os quais já tinham instaladas algumas regras básicas oriundas do próprio RouteFlow. Em um ambiente real, dezenas de

[2]Disponível em `http://cassandra.apache.org/`.

*switches* podem integrar o AS, cada qual com dezenas de regras básicas do RouteFlow e ainda terem instaladas as regras especiais do Havox, que pode processar também dezenas de diretivas específicas. A grande quantidade de regras a serem instaladas pode causar um estouro das tabelas de fluxo dos *switches* ou aumentar o tempo que leva para a verificação de correspondência entre um pacote e as regras. Esse fator pode ser mitigado com o uso de *switches* de maior capacidade computacional.

## 5.5 Análise comparativa e vantagens

A arquitetura Havox une duas soluções acadêmicas relevantes no âmbito de SDN, o RouteFlow e o Merlin, de forma que as funcionalidades de ambos possam cooperar para que o controle administrativo de um AS possa ser realizado de maneira mais trivial e compatível com o modelo de uso da *Internet* contemporânea. Além disso, a arquitetura oferece também uma DSL de orquestração do tráfego da rede simplificada e intuitiva, provendo a abstração de elementos de conectividade da rede subjacente e isentando o operador de ter conhecimento das combinações de pares de *hosts* de origem e de destino possíveis ao escolher um *switch* de saída no ato da definição das diretivas.

Além desse benefício de uso da DSL, a arquitetura também trata automaticamente as regras geradas pelo Merlin e, por meio de sua API, confere flexibilidade de configuração de como as regras devem ser tratadas, permitindo que até mesmo a sintaxe das regras seja alterada de acordo com o destino delas, bastando que o sistema conheça a lógica de tradução. O objetivo almejado para a API do sistema é que ela proveja um serviço remoto [16], inclusive não necessitando que a mesma esteja dentro do domínio administrativo.

No que tange à comparação com outras soluções, a primeira comparação natural é com o próprio Merlin puro, que integra a pilha de aplicações da arquitetura Havox. Conforme explicado no Capítulo 2, o Merlin provê uma linguagem de orquestração de redes OpenFlow baseada na definição de caminhos por expressões regulares.

O Merlin é desenvolvido com um foco em redes institucionais, onde os *hosts* representam computadores e dispositivos de rede com endereçamento estático ou que são conhecidas as suas localizações exatas dentro do domínio. Isso pode ser comprovado pelo fato de o *framework* operar apenas com endereços IP de origem

e de destino estáticos, não operando com prefixos de rede contendo máscaras. Outro fator que comprova isso é a inserção automática que o Merlin faz dos endereços IP dos pares de *hosts* nas regras. Deveras, em uma rede OpenFlow institucional onde o tráfego além dos *hosts* não precisa ser levado em conta, faz sentido que os endereços de origem e de destino dos fluxos sejam inseridos para diferenciação do tráfego.

O Havox expande a utilização do Merlin para o escopo da *Internet*, onde os vizinhos dos *switches* não são de fato *hosts*, mas roteadores de borda de outros sistemas autônomos, e onde o tráfego passante decerto não se inicia e nem se finaliza apenas em vizinhos. Para tanto, as diretivas na linguagem do Havox são transformadas em políticas Merlin respeitando as limitações do *framework*, com inferência correta dos pares de *hosts* origem-destino baseados nos *switches* de saída e com a remoção e posterior resgate das máscaras dos prefixos de rede, sem que o operador tenha que intervir no tratamento. Um confronto entre os Códigos 5.1 de diretivas Havox e 5.3 de políticas Merlin transcompiladas, junto com os resultados das Tabelas 5.5 a 5.8, mostra as vantagens do tratamento que a arquitetura proposta faz, em relação a como seria se o operador tivesse que escrever manualmente o Código 5.3 e tratar por conta própria os resultados.

A transformação da segunda diretiva do Código 5.1 em política Merlin e na sequência em regras especiais é mostrada na Figura 5.11 para fins de comparação. Nota-se que o operador só precisa saber do *switch* e dos campos de correspondência ao definir uma diretiva. A política Merlin transcompilada já tem o bloco de iteração e os *hosts* definidos, além da máscara de sub-rede removida. As sete regras OpenFlow especiais resultantes são distribuídas entre os quatro *switches* do domínio. As regras iniciais dos fluxos inclusive já têm as máscaras de sub-rede recuperadas.

Uma outra comparação possível é com o *framework* FatTire, que também segue a mesma linha do Merlin quanto à definição de caminhos por expressões regulares e geração de regras OpenFlow que satisfaçam as políticas estipuladas. O FatTire tem a vantagem de oferecer redundância ao criar caminhos alternativos que os pacotes podem trafegar em caso de falhas no caminho original. Porém, o *framework* não trabalha com pares de *hosts* origem-destino, o que é necessário para que se saiba de e para quais vizinhos um fluxo deve ser encaminhado. Além do mais, o FatTire também é projetado com foco em redes institucionais e teria de ter a sua utilização generalizada para sistemas autônomos da mesma forma que o Havox faz com o Merlin.

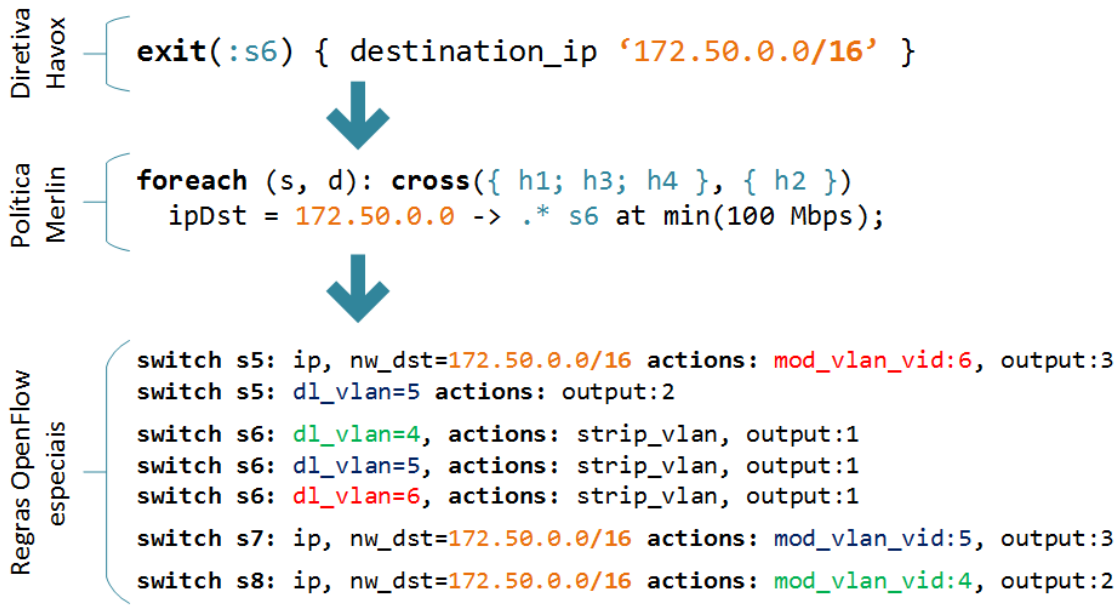



Figura 5.11: Transformação de uma das diretivas em um conjunto de regras especiais.

## 5.6 Limitações

Como numa curva de aprendizado, o tempo que levou para o entendimento apropriado das ferramentas RouteFlow e Merlin foi necessário para se poder construir este trabalho. Nesse processo, foi observado também que ambas possuem limitações que não impactam seriamente para esta prova de conceito proposta, mas que precisam ser suplantadas para a concretização de trabalhos futuros e aprimoramentos planejados.

A primeira e mais simples limitação da arquitetura Havox é ainda não suportar endereçamento IPv6. Essa é na realidade uma limitação do Merlin, que por consequência acaba sendo também da arquitetura proposta. Uma vez que o Merlin passe a suportar endereços IPv6, será trivial aprimorar o Havox para tal, bastando implementar a tradução para a sintaxe do Merlin e adaptar o tratamento de máscaras para contemplar também endereços IPv6.

Outra limitação do Havox é a não definição de prioridades entre as diretivas. As regras OpenFlow especiais do Havox são todas instaladas com a mesma prioridade máxima no RouteFlow. Isso garante que as regras especiais do operador prevaleçam sobre as da plataforma, mas não resolve a precedência entre as regras especiais. Por exemplo, se as regras de uma diretiva de saída de tráfego com porta de destino 80 forem instaladas juntamente com as regras de uma diretiva

por porta de origem e destino 80, todas com a mesma prioridade, não há como ter certeza sobre qual corresponderá [28, 31]. O fornecimento da prioridade, ou de uma categoria de prioridade, é visado como trabalho futuro no Capítulo 6.

A terceira limitação identificada diz respeito às variações de topologia entre os *switches* OpenFlow da rede. Conforme os resultados mostram, a arquitetura transforma diretivas em regras OpenFlow com sucesso em uma organização de *switches* que não precisam estar em *full-mesh*, mas que requer que todos tenham enlaces com um *host*. Isso se dá por causa da inferência de associação entre *switches* da camada de dados e as instâncias de roteamento da camada virtual, que usa as rotas OSPF internas e os endereços IP das interfaces de conexão dos *switches* com os *hosts*, conforme explanado no Capítulo 3. Em um cenário onde existem *switches* que são apenas de trânsito e não se vinculam a nenhum *host*, não há um endereço IP de interface com *hosts*, o que prejudica a inferência. Também listado como trabalho futuro, uma solução seria fornecer direto pela requisição à API as associações entre a camada de dados e a virtual, o que tornaria essa inferência desnecessária.

Uma quarta limitação é acerca do tratamento de remoção e recuperação das máscaras de sub-redes usando a lista de redes alcançáveis conhecidas. O tratamento requer que essa lista esteja populada para validar um endereço IP e substituí-lo pelo seu prefixo de rede com máscara. Uma condicional da biblioteca garante que só ocorra o tratamento se a lista estiver populada. Se não estiver, as regras mantêm os atributos como estão. De qualquer forma, mesmo que a lista esteja populada, o uso dos atributos de endereço IP estão condicionados à existência dos respectivos prefixos de rede na lista para serem efetivamente usados. Essa limitação pode ser suplantada se for usada uma estrutura auxiliar para guardar os prefixos completos antes de terem suas máscaras removidas na transcompilação. Essa estrutura pode ser um banco não relacional, conforme sugerido na Seção 5.4.2.

A própria decisão de se utilizar o RouteFlow e o Merlin também pode ser vista como uma limitação, uma vez que a arquitetura Havox passa a depender da manutenção dessas soluções para que possa ser continuamente aprimorada. Na verdade, a arquitetura requer uma plataforma de roteamento IP apenas para consultar a tabela de rotas das instâncias de roteamento, já que as regras geradas são expostas através da API. Se essas rotas forem obtidas de outra forma, como por um arquivo a parte ou banco, então não haverá mais a necessidade de se usar especificamente o RouteFlow como parte da arquitetura. De igual maneira, o

Merlin também poderia ser substituído por outra solução que provesse o cálculo de caminhos e regras primitivas se a comunicação entre tal componente hipotético e a arquitetura Havox fosse via requisição ou banco, em vez de por conexão SSH.

Como prova de conceito, a implementação atual da arquitetura Havox no cenário testado não sofre impactos decorrentes das limitações apontadas. No entanto, com o crescimento e amadurecimento da arquitetura, as limitações eventualmente serão tratadas seguindo os preceitos estabelecidos e levando em consideração a realidade dos sistemas autônomos e da *Internet*.

## 5.7 Síntese

Este capítulo tratou da execução da arquitetura Havox e dos seus componentes sobre um cenário de testes que simula um AS principal usando a arquitetura e recebendo anúncios BGP de outros AS vizinhos e remotos. Através do fornecimento de diretivas de configuração em uma DSL própria, os resultados no formato das regras OpenFlow instaladas com êxito e dos testes de geração de pacotes que corresponderam com essas regras mostram que a arquitetura teve sucesso em orquestrar a rede OpenFlow como um todo.

O processo todo se dá em três etapas, de transcompilação, de tratamento e de instalação. Na primeira, as diretivas Havox fornecidas são transcompiladas para políticas Merlin com base na descrição da topologia física subjacente. Na segunda, após a compilação das políticas, as regras primitivas geradas são tratadas e estruturadas, de forma que possibilitem que o operador possa determinar como elas devem ser exportadas. Na última etapa, após a exportação da API, as regras já tratadas são instaladas nos *switches* gerenciados pelo RouteFlow através do módulo RFHavox criado neste trabalho. Essas regras especiais possuem precedência sobre as regras básicas da plataforma, priorizando assim as decisões do operador da rede.

# 6. Conclusão

Este capítulo final faz um resumo de todo o trabalho, desde a sua concepção até os resultados, indo além destes com a idealização dos próximos caminhos que devem ser seguidos. Neste capítulo também são salientadas as contribuições e como elas podem ser empregadas.

## 6.1 Retrospectiva

Foi realizado um extenso levantamento na literatura acerca do estado da arte em redes definidas por *software*, protocolo OpenFlow e linguagens específicas de domínio para resolver problemas de orquestração em aberto. Durante esse levantamento, este trabalho foi idealizado no momento em que foi identificada a possibilidade de integrar dois outros trabalhos de grande relevância em uma arquitetura única.

A união da plataforma de roteamento IP RouteFlow e do *framework* de gerenciamento de rede Merlin deu origem ao Havox, uma arquitetura flexível de orquestração de tráfego em redes OpenFlow.

As instâncias de roteamento que executam na camada virtual do RouteFlow são responsáveis por tomar as decisões de encaminhamento interno em um AS. Essas decisões são traduzidas e convertidas em regras OpenFlow instaladas nos *switches* correspondentes, associados em um para um com as instâncias de roteamento.

O Merlin provê uma camada de abstração no ato da descrição de políticas de encaminhamento interno, de forma que o operador possa definir predicados de correspondência de regras e os caminhos que os pacotes correspondidos devem seguir, sem que seja nacessário saber como as regras OpenFlow são ou foram

criadas.

O RouteFlow é uma solução com foco em redes de ASes, mas não fornece uma linguagem ou interface para que o usuário defina as suas próprias lógicas de encaminhamento. O Merlin oferece essa possibilidade, com uma linguagem própria e relativamente fácil de ter as políticas de encaminhamento descritas, mas é focado em redes institucionais e não executa processos de decisão de roteamento.

Foi observado que as funcionalidades de ambos os trabalhos se complementavam. Com a adição de uma lógica de transcompilação de diretivas de orquestração próprias em políticas do Merlin, tendo como base o conhecimento da rede obtido do RouteFlow, o Havox possibilitou que as aplicações desses trabalhos coexistissem e cooperassem para um gerenciamento flexível da rede de um AS, de maneira que a rede OpenFlow tenha as suas regras básicas funcionais aliadas às regras especiais definidas por um operador.

Um processo do RouteFlow implementado neste trabalho, RFHavox, fornece à API do Havox um arquivo contendo diretivas de orquestração em uma linguagem própria da arquitetura proposta, um arquivo de descrição da topologia da camada de dados e os parâmetros que definem como as regras OpenFlow criadas devem ser exportadas.

As diretivas de orquestração fornecidas contêm, cada, um conjunto de atributos de correspondência compatíveis com o protocolo OpenFlow e um *switch* de borda que deve atuar como saída para os pacotes correspondidos. O arquivo de topologia é avaliado para que o Havox saiba como é a conectividade da rede, mas não é alterado, apenas repassado para o Merlin como um dos argumentos necessários para a compilação das regras. O outro argumento é o arquivo de políticas na linguagem do Merlin que é transcompilado a partir das diretivas.

As regras compiladas são estruturadas em objetos e tratadas de acordo com os parâmetros fornecidos pelo RFHavox na requisição à API. Nesse ínterim, as rotas BGP obtidas do RouteFlow auxiliam no tratamento das regras que possuem correspondências por endereço IP de origem e de destino, validando quais atributos desses tipos nas regras devem ser validados e quais devem ser eliminados. Esse é o momento que configura o uso do Merlin para além de uma mera rede institucional.

A API exporta as regras tratadas em formato padronizado JSON. O RFHavox obtém essas regras e as enfileira no IPC do RouteFlow para eventual instalação

nos *switches* OpenFlow.

Os resultados colhidos a partir de uma execução da arquitetura Havox em um cenário de teste mostram que as regras são instaladas com sucesso, o que firma a prova de conceito como funcional, conclui o objetivo deste trabalho e abre novos horizontes para expansões futuras.

## 6.2 Contribuições

A arquitetura Havox é fruto de um trabalho multidisciplinar que empregou conceitos dos campos de redes definidas por *software*, protocolo OpenFlow, roteamento na *Internet* com o protocolo BGP, desenvolvimento de linguagens específicas de domínio e metaprogramação.

Dentro dos campos citados, este trabalho contribuiu com os seguintes artefatos e conceitos:

- Uma arquitetura de orquestração de tráfego em redes OpenFlow que une as funcionalidades de outros dois projetos de relevância acadêmica, o RouteFlow e o Merlin, com um núcleo lógico que mantém comunicação com ambos.
- Uma linguagem específica de domínio, implementada sobre a linguagem Ruby, capaz de interpretar diretivas de orquestração de tráfego contendo atributos de correspondência e um *switch* de saída para os pacotes correspondidos.
- A possibilidade de fácil expansão das funcionalidades das aplicações do Havox, que são a API e a biblioteca, com a inclusão de novos tipos de diretivas de orquestração e de novos parâmetros de requisição.
- A possibilidade de execução da API e da biblioteca em um servidor remoto na nuvem como um microsserviço, uma vez que a comunicação entre o RouteFlow e o Havox se dá por requisição HTTP, requerendo apenas a resolução da limitação relacionada à obtenção das rotas listada como trabalho futuro.
- Um módulo novo no código-fonte do RouteFlow, RFHavox, responsável por se comunicar com a API do Havox, passando parâmetros de tratamento das regras e os arquivos de diretivas e de topologia criados por um operador de

rede, bem como responsável também por desencapsular as regras geradas e processar a instalação delas nos *switches*.

- Avanços no código da plataforma RouteFlow, provendo suporte ao uso de regras OpenFlow contendo correspondências para IDs de VLAN, endereços IP de origem e máscaras de sub-rede diferentes de /32, além da implementação das ações de definição e de remoção de IDs de VLAN dos pacotes.

## 6.3 Trabalhos futuros

Durante a implementação da arquitetura proposta, houve momentos em que decisões foram tomadas em favor de uma entrega focada na prova de conceito, deixando em aberto ideias que foram pautadas como aprimoramentos futuros. Alguns desses aprimoramentos podem ser implementados em curto prazo, outros demandariam mais tempo e ainda outros exigiriam mais intervenções de código em componentes da arquitetura. São eles:

- Implementação da diretiva de descarte de pacotes correspondidos. Uma diretiva como essa é útil quando não se deseja que o AS forneça trânsito para pacotes com determinados atributos. O Código 6.1 sugere como seria usada a diretiva `drop` hipotética. Essa diretiva não foi implementada porque o Merlin ainda não tem suporte ao descarte de pacotes. Estuda-se uma intervenção no código-fonte do Merlin para a implementação do descarte ou a criação de regras de descarte por fora do *framework*.

- Implementação da definição de um caminho de *switches* nas diretivas. Pode ser interessante para o AS que um determinado fluxo de pacotes correspondidos siga por um caminho interno que passe por *switches* específicos antes do *switch* de saída. As linhas 6 e 7 do Código 6.1 conferem sugestões de como essa modificação poderia ser. Essa funcionalidade aproveitaria ainda mais o fato de que o Merlin usa expressões regulares para definir os caminhos a serem percorridos pelos pacotes.

- Definição de prioridades das diretivas. O uso do mesmo valor de prioridade máxima para todas as regras especiais, elencado como uma limitação no Capítulo 5, pode ser revisto, resultando no uso de valores inteiros ou de categorias de prioridade. As linhas 9 e 10 do Código 6.1 sugerem diretivas com prioridades definidas, dado que 49200 é o valor máximo de prioridade

suportado pelo RouteFlow e 32800 é o segundo maior valor usado pela plataforma. As prioridades das regras vindas do Havox poderiam ser qualquer valor inteiro dentro desse intervalo.

- Implementação da diretiva de associação entre *switches* e instâncias de roteamento. Como uma forma eficiente de evitar que a inferência por OSPF das associações entre a camada de dados e a virtual seja feita, o operador poderia fornecer as associações na forma de diretivas hipotéticas `associate`, conforme o Código 6.1.

- Refatoração da lógica de obtenção das rotas conhecidas. A API e a biblioteca Havox, que estão no centro da arquitetura proposta, foram inicialmente projetadas para executarem em servidores remotos, provendo um serviço de criação de regras OpenFlow em nuvem. Porém, como prova de conceito, a implementação atual fere esse princípio ao usar uma conexão SSH da máquina do Havox para a máquina executando o RouteFlow, com o objetivo de acessar as RIBs das instâncias de roteamento em cada contêiner. Um serviço na nuvem não deve se comunicar com um usuário por outros canais que não sejam por requisições HTTP [15]. As rotas poderiam ser passadas do RouteFlow para o Havox na mesma requisição das regras ou então um serviço de fornecimento de rotas poderia ser implementado como um módulo do RouteFlow, respondendo requisições de rotas oriundas do Havox. Ou ainda poderia-se usar um banco de rotas alimentado pelo RouteFlow e consumido pelo Havox.

- Atualização e remoção de regras especiais. A versão atual do módulo RFHavox no RouteFlow apenas instala as regras OpenFlow especiais vindas do Havox. Ainda há a pendência para que essas regras sejam atualizadas ou removidas conforme a demanda. O operador poderia fornecer novas diretivas periodicamente ou a comunicação com um *switch* pode ser interrompida, o que demandaria que a arquitetura fosse resiliente o suficiente para gerar novos caminhos e atualizar as tabelas de fluxo dos *switches* com novas regras especiais.

- Aprimoramento do módulo RFHavox para fornecer uma interface que permita ao usuário definir as diretivas a serem submetidas para a arquitetura Havox. Atualmente, o módulo apenas lê as diretivas do seu arquivo fonte.

- Uso de bancos não relacionais no tratamento dos endereços IP e dos prefixos de rede. Conforme apontado no Capítulo 5 como limitação, a lógica atual

```
associate :s5, :rfvmA
associate :s6, :rfvmB
associate :s7, :rfvmC
associate :s8, :rfvmD

exit([:s5, :s6]) { destination_port 22 }
exit('s8 .* s6') { source_ip 80 }

exit(:s6, 41000) { destination_port 50 }
exit(:s8, 45000) { destination_port 50 }

drop {
  destination_port 3306
  destination_ip '172.1.0.0/16'
}
```

Código 6.1: Diretivas e modificações hipotéticas a serem implementadas.

de tratamento dos atributos de endereço IP de origem e de destino antes e depois da compilação do Merlin deve ser melhorada, a fim de permitir que a arquitetura escale conforme o número de rotas conhecidas cresça. Essa lógica refatorada pode empregar o uso de bancos não relacionais.

- Uso de alguns dos trabalhos relacionados elencados no Capítulo 2 como parte da pilha de aplicações da arquitetura Havox. O Propane, por exemplo, poderia ser usado para gerar as configurações dos *daemons* BGP das instâncias de roteamento mantidas pelo RouteFlow. Talvez até pudesse ser desenvolvido um novo conjunto de diretivas Havox para lidar com o Propane, seguindo o mesmo princípio de transcompilação usado com o Merlin. Dessa forma, a configuração das instâncias de roteamento do AS, que ainda é distribuída, também passaria a ser centralizada nos mesmos moldes da definição de políticas de orquestração de tráfego.

## 6.4 Considerações finais

A arquitetura Havox proposta é alinhada com as pesquisas contemporâneas acerca do conceito de redes definidas por *software*. O foco dos estudos tem subido cada vez mais na pilha de SDN, foco esse que já foi na camada de dados, já esteve na camada de controle e agora está na camada de aplicação, com a implementação de DSLs e *frameworks* que facilitem a configuração da rede OpenFlow e permitem que o operador abstraia até mesmo de qual controlador está sendo usado.

Traçando um paralelo com outros campos da Computação, como o de desenvolvimento de aplicações *web* ou de sistemas operacionais, as implementações de soluções para SDN têm seguido a mesma lógica, onde um recurso consome serviços da API norte de outros recursos da camada de baixo da pilha e fornece uma API sul para que recursos ainda mais sofisticados da camada de cima consumam seus serviços. Nessa visão, quanto mais se está próximo do topo da pilha, mais se abstrai da lógica das camadas inferiores.

O resultado eventual do aumento da abstração citado é uma aproximação das linguagens de configuração à linguagem natural humana, a partir do ponto em que se torna possível saber o que os algoritmos fazem somente lendo as sintaxes usadas.

O fruto deste trabalho contribui com esse processo ao oferecer uma linguagem cuja leitura de código implementado nela seja compreensível até mesmo por aqueles que não atuam diretamente no campo.

# Referências Bibliográficas


[1] KUROSE, J.; ROSS, K. Computer networks and the internet. *Computer networking: A Top-down approach. 7th ed. London: Pearson*, 2016.

[2] REKHTER, Y.; LI, T.; HARES, S. *A border gateway protocol 4 (BGP-4)*. [S.l.], 2005.

[3] KATZ, D.; KOMPELLA, K.; YEUNG, D. *Traffic engineering (TE) extensions to OSPF version 2*. [S.l.], 2003.

[4] MALKIN, G. Rfc 2453: Rip version 2. *Accessed November*, 1998.

[5] XIA, W. et al. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, IEEE, v. 17, n. 1, p. 27–51, 2015.

[6] WICKBOLDT, J. A. et al. Software-defined networking: management requirements and challenges. *IEEE Communications Magazine*, IEEE, v. 53, n. 1, p. 278–285, 2015.

[7] MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM, v. 37, n. 4, p. 316–344, 2005.

[8] NASCIMENTO, M. R. et al. Routeflow: Roteamento commodity sobre redes programáveis. *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos-SBRC*, 2011.

[9] ROTHENBERG, C. E. et al. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In: ACM. *Proceedings of the first workshop on Hot topics in software defined networks*. [S.l.], 2012. p. 13–18.

[10] SOULÉ, R. et al. Managing the network with merlin. In: ACM. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. [S.l.], 2013. p. 24.

[11] SOULÉ, R. et al. Merlin: A language for provisioning network resources. In: ACM. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. [S.l.], 2014. p. 213–226.

[12] SOULÉ, R. et al. Scalable network management with merlin. In: CORNELL UNIVERSITY. *Computing and Information Science Technical Reports*. [S.l.], 2013.

[13] RECKER, J. *Scientific research in information systems: a beginner's guide*. [S.l.]: Springer Science & Business Media, 2012.

[14] DRESCH, A.; LACERDA, D. P.; JÚNIOR, J. A. V. A. *Design science research: método de pesquisa para avanço da ciência e tecnologia*. [S.l.]: Bookman Editora, 2015.

[15] MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

[16] NEWMAN, S. *Building microservices: designing fine-grained systems*. [S.l.]: "O'Reilly Media, Inc.", 2015.

[17] MAGONI, D.; PANSIOT, J. J. Analysis of the autonomous system network topology. *ACM SIGCOMM Computer Communication Review*, ACM, v. 31, n. 3, p. 26–37, 2001.

[18] ONF. Software-defined networking: The new norm for networks. *Open Networking Foundation White Paper*, v. 2, p. 2–6, 2012.

[19] TROIS, C. et al. A survey on sdn programming languages: toward a taxonomy. *IEEE Communications Surveys & Tutorials*, IEEE, v. 18, n. 4, p. 2687–2712, 2016.

[20] MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

[21] ENNS, R. et al. Netconf configuration protocol (ietf rfc 6241). *Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman.-June*, 2011.

[22] PFAFF, B.; DAVIE, B. Rfc 7047: The open vswitch database management protocol,(2013). 2013. Disponível em: <http://www.ietf.org/rfc/rfc7047.txt>.

[23] LEWIS, D. et al. Locator/id separation protocol (lisp). *RFC 6830*, 2013.

[24] YU, J. et al. Forwarding programming in protocol-oblivious instruction set. In: IEEE. *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*. [S.l.], 2014. p. 577–582.

[25] SMITH, M. et al. Opflex control protocol. *IETF*, 2014.

[26] FOSTER, N. et al. Languages for software-defined networks. *IEEE Communications Magazine*, IEEE, v. 51, n. 2, p. 128–134, 2013.

[27] REITBLATT, M. et al. Fattire: Declarative fault tolerance for software-defined networks. In: ACM. *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. [S.l.], 2013. p. 109–114.

[28] ONF. Openflow switch specification version 1.0. *Open Networking Foundation*, 2009. Disponível em: <http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf>.

[29] ONF. Openflow switch specification version 1.1. *Open Networking Foundation*, 2011. Disponível em: <http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf>.

[30] ONF. Openflow switch specification version 1.2. *Open Networking Foundation*, 2011. Disponível em: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf>.

[31] ONF. Openflow switch specification version 1.3. *Open Networking Foundation*, 2012. Disponível em: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf>.

[32] ONF. Openflow switch specification version 1.4. *Open Networking Foundation*, 2013. Disponível em: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf>.

[33] ONF. Openflow switch specification version 1.5. *Open Networking Foundation*, 2014. Disponível em: <https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.0.pdf>.

[34] PRESTON-WERNER, T. Semantic versioning 2.0.0. 2013. Disponível em: <http://semver.org>.

[35] DATE, C. J.; DARWEN, H. *A Guide to the SQL Standard*. [S.l.]: Addison-Wesley New York, 1987. v. 3.

[36] MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009.

[37] VISSER, E. Meta-programming with concrete object syntax. In: SPRINGER. *GPCE*. [S.l.], 2002. v. 2, p. 299–315.

[38] SCHORDAN, M.; QUINLAN, D. A source-to-source architecture for user-defined optimizations. In: SPRINGER. *JMLC*. [S.l.], 2003. v. 3, p. 214–223.

[39] FIELDING, R.; RESCHKE, J. Hypertext transfer protocol (http/1.1): Semantics and content (rfc 7231). 2014.

[40] SHAW, M. Abstraction techniques in modern programming languages. *IEEE software*, IEEE, v. 1, n. 4, p. 10–26, 1984.

[41] BECKETT, R. et al. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In: ACM. *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. [S.l.], 2016. p. 328–341.

[42] RAGAN, S. Bgp errors are to blame for monday's twitter outage, not ddos attacks. *CSO Online*, Nov 2016. Disponível em: <http://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html>.

[43] ANDERSON, M. Time warner cable says outages largely resolved. *The Seattle Times*, Aug 2014. Disponível em: <http://www.seattletimes.com/business/time-warner-cable-says-outages-largely-resolved/>.

[44] GUPTA, A. et al. Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, ACM, v. 44, n. 4, p. 551–562, 2015.

[45] MONSANTO, C. et al. Composing software defined networks. In: *NSDI*. [S.l.: s.n.], 2013. v. 13, p. 1–13.

[46] CORRÊA, C. *Uma solução de roteamento como serviço baseada em redes definidas por software*. Dissertação (Mestrado) — Universidade Federal do Estado do Rio de Janeiro, 2012.

[47] ROTHENBERG, C. E. et al. Revisiting ip routing control platforms with openflow-based software-defined networks. In: *Future Internet Experimental Research Workshop (WPEIF)*. [S.l.: s.n.], 2012.

[48] CORRÊA, C. et al. Uma plataforma de roteamento como serviço baseada em redes definidas por software. In: *Workshop de Gerência e Operação de Redes e Serviços (WGRS). SBRC*. [S.l.: s.n.], 2012. v. 12.

[49] BOSSHART, P. et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, ACM, v. 44, n. 3, p. 87–95, 2014.

[50] JONES, C. *Software engineering best practices*. [S.l.]: McGraw-Hill, Inc., 2009.

[51] LENNON, E. B. Contingency planning guide for information technology systems. *ITL Bulletin-June*, 2002.

[52] HUSTON, G. The growth of the bgp table-1994 to present. *http://bgp.potaroo.net*, 2005.

[53] BU, T.; GAO, L.; TOWSLEY, D. On characterizing bgp routing table growth. *Computer Networks*, Elsevier, v. 45, n. 1, p. 45–54, 2004.