Havox: Um serviço para orquestração de tráfego em alto nível em redes OpenFlow

Rodrigo Soares e Silva[1], **Sidney Cunha de Lucena**[1]

[1]Programa de Pós-Graduação em Informática Aplicada
Universidade Federal do Estado do Rio de Janeiro (UNIRIO)
Rio de Janeiro, RJ – Brasil

{rodrigo.soares, sidney}@uniriotec.br

Resumo. *As redes definidas por software já vêm conferindo maior poder de controle aos operadores de rede, mas gerar muitas regras OpenFlow simultaneamente ainda é um trabalho árduo. Existem propostas que geram um conjunto de regras para operações básicas de rede, bem como ferramentas com gramática própria que interpretam políticas de rede e as traduzem para regras. Observou-se que a combinação dessas soluções resulta em um novo recurso que fornece uma linguagem de configuração amigável para definir como o tráfego deve ser encaminhado. Este trabalho apresenta o Havox, um serviço que auxilia a orquestração de tráfego através de uma linguagem de configuração simples, de alto nível e baseada nos campos OpenFlow. Os experimentos mostraram uma considerável diminuição do esforço de programação da rede de forma escalável, reduzindo o número de parâmetros e de linhas de código.*

Abstract. *Software-defined networking has been providing more control to network operators, but generating several OpenFlow rules simultaneously is yet a tough job. There are already research projects that generate a set of 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. It was observed that a combination of these solutions can generate a new resource capable of giving to network administrators a friendly configuration language to help them define how data traffic should be forwarded. This work presents Havox, a service meant to help with traffic orchestration through a simple, high level configuration language based on OpenFlow fields. Experiments show a considerable reduction of network programming effort in a scalable way, reducing both the number of parameters and the number of lines of code.*

1. Introdução

Um sistema autônomo (AS, do inglês *Autonomous System*) tem suas próprias estratégias de negócios no que tange ao tráfego de pacotes que passa por sua rede. Entretanto, em uma rede tradicional, ASes não são capazes de encaminhar tráfego com base em atributos de camadas superiores da pilha de protocolos, como portas de transporte, ou pelo endereço de origem. Com o avanço das pesquisas em redes definidas por *software* (SDN, do inglês *Software-Defined Networking*) [ONF 2012], uma nova perspectiva de controle sobre o tráfego se abre para os ASes. A manutenção da lógica de decisão em um elemento controlador centralizado, em vez de em cada roteador de forma distribuída, permitiu que

um controle mais granular sobre os pacotes pudesse ser exercido. Soluções como plataformas de roteamento IP e linguagens específicas de domínio (DSL, do inglês *Domain Specific Language*) [Mernik et al. 2005], que facilitam a configuração da rede, ganham destaque neste novo cenário porque lidam com a complexidade intrínseca de configuração e manutenção da rede, assim como viabilizam o emprego de SDN em cenários reais e já maduros de roteamento [Xia et al. 2015, Wickboldt et al. 2015].

Centrado nesses pontos, propõe-se neste trabalho o Havox[1], um serviço que integra e flexibiliza duas soluções conhecidas: o RouteFlow [Nascimento et al. 2011, Rothenberg et al. 2012] e o Merlin [Soulé et al. 2014]. O primeiro é uma plataforma de roteamento IP e o segundo é um *framework* de gerenciamento de fluxos de pacote, ambos sobre redes OpenFlow [McKeown et al. 2008]. Essa integração é garantida com uma camada adicional de abstração sobre ambas as soluções, que fornece uma linguagem de configuração legível e de fácil utilização implementada sobre a linguagem de propósito geral Ruby, de forma que uma linha de configuração é capaz de gerar múltiplas regras de encaminhamento. Através dessas configurações, define-se a forma como o encaminhamento do tráfego na camada de dados se dará, e a linguagem ainda pode ser aprimorada com novas funções de configuração, conforme os casos de uso. Além disso, a integração das ferramentas exigiu também contribuições no código-fonte[2] do RouteFlow, tanto no desenvolvimento de um módulo para requisições de regras ao Havox como na implementação de suporte a mais atributos de correspondência de pacotes, como o IP de origem.

Os experimentos realizados mostram que uma diretiva de orquestração (uma instrução de encaminhamento de tráfego) do Havox é capaz de gerar múltiplas regras OpenFlow, com número crescente de regras conforme o número de *switches* aumenta. Observou-se que o número de regras criadas escala enquanto o número de diretivas permanece pequeno, o que indica que o esforço para se configurar o comportamento da rede permanece baixo, mesmo que a rede aumente em número de *switches*.

As demais seções deste trabalho são organizadas da seguinte forma: a Seção 2 descreve com maiores detalhes o Havox, a Seção 3 apresenta a validação experimental da proposta, a Seção 4 referencia os trabalhos relacionados e a Seção 5 traz as conclusões e os trabalhos futuros.

2. Proposta

Esta seção inicia pela descrição macroscópica do Havox e, em seguida, aprofunda-se no funcionamento da suas diretivas de orquestração e na integração entre os componentes utilizados.

2.1. A arquitetura do Havox

A arquitetura do Havox possui o RouteFlow e o Merlin como componentes, com ambos operando em sinergia dentro de suas funções.

O RouteFlow convencional mantém, em uma camada virtual, instâncias de roteamento Quagga[3] em contêineres, dispostas em uma topologia idêntica à dos *switches*

[1]Disponível em `https://github.com/rodrigosoares/havox`.

[2]*Fork* do código original disponível em `https://github.com/rodrigosoares/RouteFlow`.

[3]Disponível em `https://www.nongnu.org/quagga/`.

OpenFlow na camada de dados espelhada, de forma que cada instância de roteamento esteja associada a um *switch*. Quando eventos nessa última camada ocorrem, eles são tratados pelo correspondente na camada virtual, que toma decisões de roteamento. Essas decisões são convertidas em instruções OpenFlow e instaladas nos respectivos *switches*. O módulo RFServer do RouteFlow é o responsável por gerenciar como os *switches* da camada de dados, as instâncias de roteamento da camada virtual e o controlador se comunicam. Para tanto, ele recebe mensagens dos módulos RFClient e RFProxy. O primeiro executa em cada contêiner de instância de roteamento e o último na camada de controle. As mensagens internas da plataforma são comutadas através de um canal de comunicação interprocesso (IPC) implementado na forma de um banco de dados não relacional MongoDB.

O operador da rede define dois arquivos que são processados pelo sistema: um arquivo de diretivas de orquestração e um arquivo de descrição da topologia. Esses arquivos, bem como parâmetros opcionais de configuração, são enviados ao núcleo da arquitetura pelo módulo RFHavox, elemento adicionado ao RouteFlow que integra a plataforma à arquitetura. Na versão atual da arquitetura, esse módulo obtém os arquivos para submissão de um diretório específico no sistema de arquivos do RouteFlow, o que requer que o operador crie os arquivos nesse diretório e os especifique na inicialização do módulo. O RFHavox é executado depois que os demais módulos já entraram em operação, quando ele fará uma requisição contendo os arquivos citados anteriormente e receberá novas regras OpenFlow geradas com base nesses arquivos. Na sequência, o módulo injeta as mensagens contendo essas regras no IPC do RouteFlow para eventual instalação.

O Merlin é originalmente usado para calcular os caminhos entre *hosts* de uma rede, na forma de regras OpenFlow em formato textual. Os caminhos são identificados por uma ID de VLAN, que atua como um rótulo interno que distingue cada fluxo. Cada fluxo possui uma regra no *switch* de entrada da rede, que avalia a correspondência dos campos OpenFlow dos pacotes vindos de um *host* de origem e define uma ID de VLAN interna para este fluxo. Nos *switches* intermediários, podem haver zero ou mais regras que avaliam somente a ID de VLAN para determinar o encaminhamento do fluxo, e no *switch* de saída há regras que removem a ID de VLAN e encaminham o pacote para o *host* de destino. O Merlin é projetado como uma solução para redes institucionais, daí o uso do termo *host* e a necessidade de se especificar *hosts* de origem e de destino. Este trabalho expande seu escopo de uso para o cenário de um AS, onde estes *hosts* passam a representar roteadores de ASes vizinhos ou sub-redes.

A arquitetura é ilustrada na Figura 1. Seu cerne é a biblioteca homônima Havox, que é invocada através de uma API *web* que recebe requisições HTTP com parâmetros bem definidos e responde com mensagens em formato padronizado JSON, fazendo assim 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 escritas na linguagem de configuração do Havox através do RFHavox e é respondido com as regras OpenFlow que devem ser instaladas nos *switches*, com prioridade superior às das demais regras já instaladas pelo RouteFlow. O Merlin, por sua vez, é uma dependência que é usada para a geração dos caminhos de fluxo dos pacotes.

Na Figura 1, a sequência de passos se inicia com (1) o processo do RouteFlow invocando o módulo RFHavox. Este (2) envia uma requisição à API *web*, contendo os

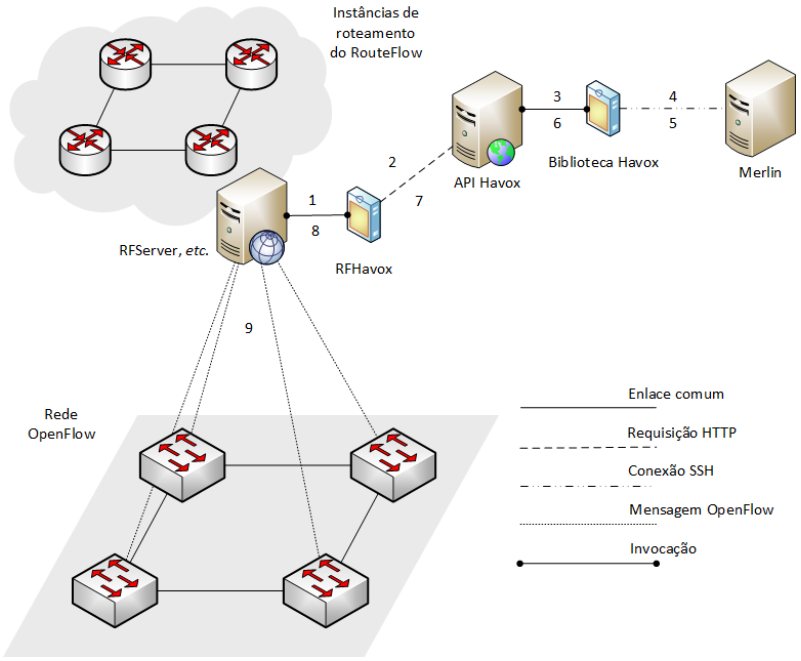


Figura 1. A arquitetura Havox.

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 geradas 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 (6) são devolvidas à API, que (7) encapsula as regras em mensagem JSON e a envia ao módulo RFHavox como resposta. O módulo RFHavox extrai as regras especiais da mensagem JSON e (8) as enfileira para processamento pelos demais módulos do RouteFlow. Oportunamente, (9) as regras são instaladas nos *switches* OpenFlow da camada de dados.

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*. Como o Merlin, enquanto dependência, também consome esse arquivo, os *hosts* representam os roteadores vizinhos de outros ASes, enquanto os *switches* representam os roteadores do domínio. A Figura 2 ilustra essa associação entre as camadas virtual e de dados. O arquivo de diretivas na linguagem de configuração do Havox é um conjunto de instruções de encaminhamento de tráfego do AS. 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 o submete ao mesmo. O Merlin processa essas políticas e responde ao Havox 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,

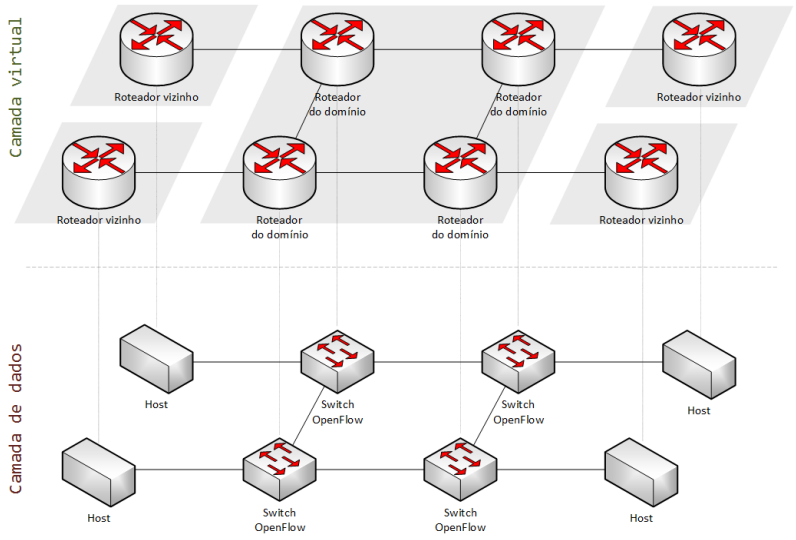


Figura 2. Associação entre a camada de dados e a camada virtual.

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.

Como prova de conceito, foi implementada para o Havox uma política onde, para uma dada classe de tráfego, é especificado qual o *switch* de saída do AS a ser usado, dentre aqueles que conduzem ao destino em questão. Assim sendo, as diretivas do Havox contêm o conjunto de campos de cabeçalho OpenFlow que correspondem à classe de tráfego e o *switch* de borda que dá saída para o tráfego correspondente.

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.

Em suma, as nomenclaturas adotadas para caracterizar as regras OpenFlow neste trabalho são:

- **Regra primitiva:** Regra gerada pelo Merlin, textual, não estruturada e obtida da leitura em saída padrão (*stdout*).
- **Regra básica:** Regra estruturada, gerada pelo processo de execução convencional do RouteFlow e derivada das decisões de roteamento da camada virtual.
- **Regra especial:** Regra gerada pela biblioteca Havox, estruturada e tratada a partir das regras primitivas e que tem prioridade sobre as regras básicas por serem originalmente definidas pelo usuário.

2.2. A linguagem de configuração

Um arquivo escrito na linguagem de configuração do serviço contém um conjunto de diretivas de orquestração, cada qual contendo um *switch* de saída e um bloco de palavras-

```
1 topology 'example.dot'
2 associate :r1, :s1
3
4 exit(:s0) { destination_ip '200.156.0.0/16' }
5 exit(:s1) { destination_port 20 }
6 exit(:s2) {
7   source_port 80
8   source_ip '200.20.0.0/16'
9 }
```

Código 1. Exemplo de arquivo de diretivas da arquitetura.

chave correspondentes a campos OpenFlow conhecidos, com os 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 e a diretiva de associação manual de instâncias de roteamento da camada virtual do RouteFlow a *switches*. O Código 1 contém exemplos de definições de diretivas e seus campos.

No Código 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. Depois, a diretiva `associate` associa o roteador *r1* ao *switch s1* manualmente. Em seguida, são definidas três diretivas de orquestração 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 com destino ao endereço de rede 200.156/16, e instrui a saída desses pacotes pelo *switch* de borda *s0*. A segunda diretiva corresponde aos pacotes cuja porta TCP de destino seja a 20, de transmissão de dados do FTP (*File Transfer Protocol*), com saída pelo *switch s1*. 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* de resposta com origem no endereço de rede 200.20/16, instruindo saída pelo *switch s2*. As tabelas 1 e 2 listam as diretivas implementadas e os campos OpenFlow suportados, respectivamente.

Diretiva	Argumento 1	Argumento 2
`topology`	caminho do arquivo (*str*)	–
`associate`	roteador (*str/sym*)	*switch* (*str/sym*)
`exit`	*switch* (*str/sym*)	lista de campos (*block*)

Tabela 1. Diretivas Havox e seus argumentos.

A linguagem de configuração do Havox não possui um compilador próprio, como no caso do Merlin. Em vez disso, o código escrito em sua sintaxe 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 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 este um argumento de tipo símbolo ou de tipo *string* com o nome do *switch* e outro argumento contendo um bloco de código, que corresponde aos campos

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	Cód. do protocolo de rede.
`in_port` (*int*)	1	Interface de entrada.
`ip_protocol` (*int*/*str*)	17, 0x11 ou “udp”	Cód. do protocolo de transporte.
`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 2. Campos OpenFlow suportados pelas diretivas de orquestração.

```
1 foreach (s, d): cross({ h1; h2; h3 }, { h0 })
2   ipDst = 200.156.0.0 -> .* s0;
3
4 foreach (s, d): cross({ h0; h2; h3 }, { h1 })
5   tcpDstPort = 20 -> .* s1;
6
7 foreach (s, d): cross({ h0; h1; h3 }, { h2 })
8   tcpSrcPort = 80 and ipSrc = 200.20.0.0 -> .* s2;
```

Código 2. Código Merlin transcompilado a partir do código Havox.

OpenFlow. Essa linguagem de configuração pode ser expandida adicionando-se métodos para novas funcionalidades. Toda o código é coberto por testes unitários que garantem que novas adições e eventuais alterações não prejudiquem o funcionamento esperado do que já existe. Maiores detalhes sobre a implementação em Ruby do Havox podem ser obtidos em seu repositório.

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. 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.

Cada diretiva de orquestração é transformada em trechos de políticas na DSL do Merlin. Essa etapa de transformar o código na linguagem do Havox em código na linguagem do Merlin é chamada de *transcompilação*[4]. No exemplo do Código 1, o Havox transcompila as diretivas para o Código 2, legível pelo compilador do Merlin.

Confrontar os Códigos 1 e 2 permite observar que automaticamente foram inferidos, durante o processo de transcompilação, os vizinhos por onde o tráfego entrará na rede (*h1*, *h2* e *h3*) e o vizinho por onde ele deverá sair (*h0*), junto com o *switch* de acesso deste (*s0*), 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*,

[4]Transcompilação, ou compilação fonte-a-fonte [Schordan and Quinlan 2003], é o processo de transformação do código-fonte escrito em uma linguagem *A* para código-fonte em uma linguagem *B*.

s podendo ser *h1*, *h2* ou *h3* e *d* sendo *h0*, deve ser aplicada a regra descrita pelo código aninhado. O processo é similar quanto às demais diretivas de orquestração. Por uma limitação da versão atual do Merlin, é necessária a repetição das linhas do iterador para cada regra individual. O Código 2 transcompilado é então submetido ao Merlin, que compilará esse código e gerará as regras OpenFlow primitivas que serão lidas, estruturadas e exportadas para o RouteFlow pela API *web* do Havox.

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 já são definidos no arquivo de topologia e 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.

2.3. 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*.

2.3.1. Etapa de transcompilação

O processo de transcompilação se inicia a partir do momento em que o RouteFlow envia uma requisição à API do Havox por meio do módulo RFHavox. A requisição é um método POST enviado para a URL da aplicação *web* da API e contém os arquivos de diretivas e de topologia. Os arquivos são repassados à biblioteca quando são recebidos pela API, iniciando a transcompilação descrita na Subseção 2.2.

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 adjacentes. 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.

Em seguida, 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 em uma lista de redes alcançáveis conhecidas, gerada a partir das rotas BGP obtidas do RouteFlow.

Feitas as inferências necessárias e o tratamento dos valores de endereçamento IP, um bloco de código Merlin é gerado para cada diretiva de orquestração da arquitetura. O arquivo contendo o código resultante é submetido ao Merlin, que o compilará e gerará as regras OpenFlow primitivas em saída padrão.

2.3.2. 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 dessas regras.

As regras primitivas são obtidas pelo Havox através do *parsing* da saída padrão gerada pelo Merlin, 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. Depois, o tratamento seguirá conforme os parâmetros passados na requisição. Os nomes dos campos são traduzidos para os nomes análogos usado pelo RouteFlow. Além disso, como o RouteFlow não implementa o enfileiramento de mensagens para qualidade de serviço, as ocorrências da ação `enqueue`, rotineiramente usadas pelo Merlin, são substituídas pela ação `output`.

Seguindo esta etapa, é iniciada a recuperação das máscaras de sub-rede que foram removidas na etapa de transcompilação. Todas as regras são iteradas e cada uma tem os valores dos seus atributos de endereço IP de origem e de destino avaliados, quando presentes. Para tanto, o sistema se baseia na lista de redes alcançáveis obtida anteriormente. Ao final da etapa, as regras são encapsuladas em uma mensagem JSON que formará a resposta da API *web*.

2.3.3. Etapa de instalação

Na etapa de instalação, as regras OpenFlow especiais encapsuladas na mensagem JSON são extraídas pelo processo RFHavox do RouteFlow.

As regras são enfileiradas no IPC do RouteFlow como mensagens RouteMod. O IPC do RouteFlow é uma coleção do MongoDB (análoga a uma tabela) e as mensagens RouteMod são documentos (análogos a registros) que possuem o ID do *switch*, as correspondências, as ações a serem adotadas, o valor de prioridade e o tipo de modificação na tabela de fluxo (adição, remoção ou alteração de regra). As regras vindas do serviço Havox têm valores de prioridade superior às das demais regras básicas, pois provêm do operador da rede e não dos processos de roteamento do RouteFlow. Oportunamente, as regras serão consumidas do IPC e instaladas nos devidos *switches*.

3. Validação experimental

Como prova de conceito, foram realizados experimentos processando uma das diretivas de orquestração do Código 1, além da diretiva de definição da topologia. A diretiva escolhida é a primeira do código apresentado, que determina que todo tráfego que entrar no domínio com destino à rede 200.156/16 deve deixar o mesmo pelo *switch s0*. Esta diretiva única foi testada para uma topologia de quatro *switches*, 2 por 2, dispostos em anel e uma topologia de nove *switches*, 3 por 3, dispostos em grade. Em ambos os casos, todos os *switches* são vinculados a *hosts*, que representam roteadores de ASes vizinhos ou sub-redes, como exposto na Figura 2, e os *switches* da coluna à direita possuem enlace com o *switch s0* de saída. Para cada AS vizinho foi definida uma faixa de sub-rede diferente que foi devidamente anunciada via BGP para o AS de trânsito que interliga os demais ASes, sobre o qual será aplicada a política de orquestração a ser testada.

As Figuras 3 e 4 contêm as disposições das topologias, que foram implementadas através do MiniNExT[5]. Diferente do Mininet[6], o MiniNExT permite o isolamento das instâncias virtuais de roteamento do RouteFlow, para que cada uma possua sua própria FIB. Com o VirtualBox, criou-se uma VM para o MiniNExT, uma para o RouteFlow e uma para o Merlin, com o Havox na máquina hospedeira. A correta ação das diretivas aplicadas foi comprovada gerando-se tráfego com iPerf[7] e observando o trajeto dos fluxos pelos contadores de regras OpenFlow.

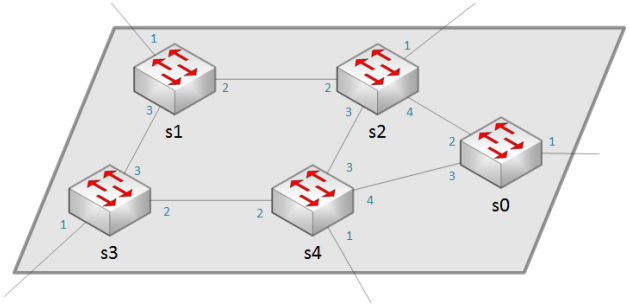


Figura 3. Topologia 2 por 2 com *s0* à direita.

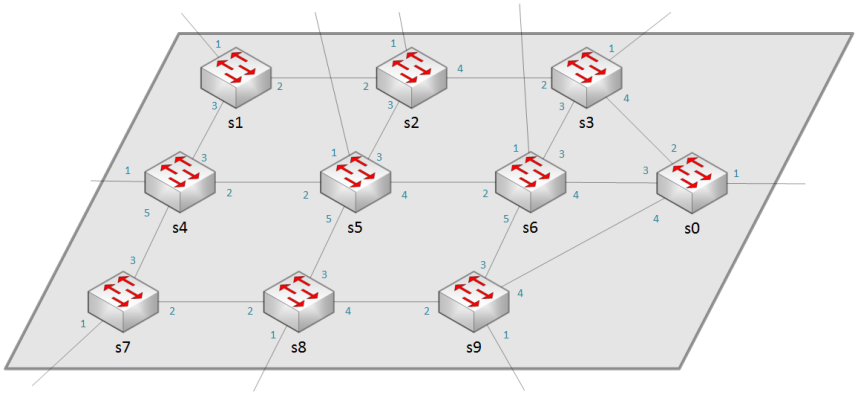


Figura 4. Topologia 3 por 3 com *s0* à direita.

No caso da topologia 2 por 2, a diretiva de orquestração derivou 10 regras OpenFlow distribuídas entre os *switches*. Nos *switches* *s1*, *s2*, *s3* e *s4* são instaladas as regras que leem o campo de endereço IP de destino e, em caso de correspondência, atribuem ao fluxo uma ID de VLAN interna e repassam o pacote para a interface de saída devida. Especificamente no caso dos *switches* de entrada *s1* e *s3*, que não possuem enlace com o *switch* de saída *s0*, é feito o repasse para *s2* e para *s4*, respectivamente. Esses últimos atuam também como *switches* intermediários, que verificarão a correspondência da ID de VLAN definida nos *switches* anteriores e farão o repasse para *s0*, caso correspondam. É importante relembrar que, nos *switches* intermediários, somente a ID de VLAN é avaliada, pois esse atributo passa a atuar como um rótulo do tráfego. Em todos os casos, quando os pacotes chegarem ao *switch* *s0*, as IDs de VLAN são removidas, caso correspondam, e os pacotes são encaminhados para o *host* (AS vizinho) associado a ele. Em suma, foram

[5]Disponível em `https://github.com/USC-NSL/miniNExT`.
[6]Disponível em `https://github.com/mininet/mininet`.
[7]Disponível em `https://iperf.fr/`.

geradas 4 regras de entrada, 2 regras intermediárias e 4 regras de saída, totalizando as 10 regras. A Figura 5 mostra a derivação da diretiva `exit` nas regras.

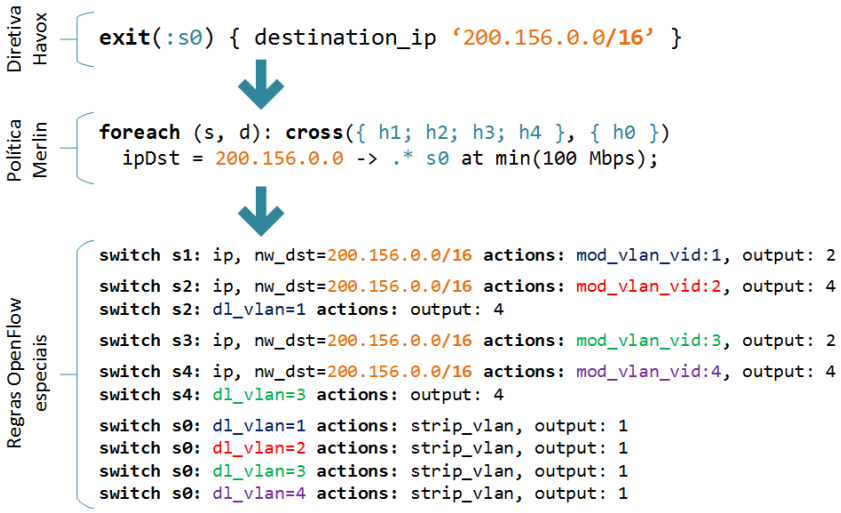


Figura 5. Transformação da diretiva em regras na topologia 2 por 2.

De maneira análoga, na topologia 3 por 3 a diretiva `exit` derivou 27 regras OpenFlow distribuídas entre os nove *switches* de entrada e intermediários, mais o *switch* de saída. As 27 regras se desdobram em 9 de entrada, 9 intermediárias e 9 de saída.

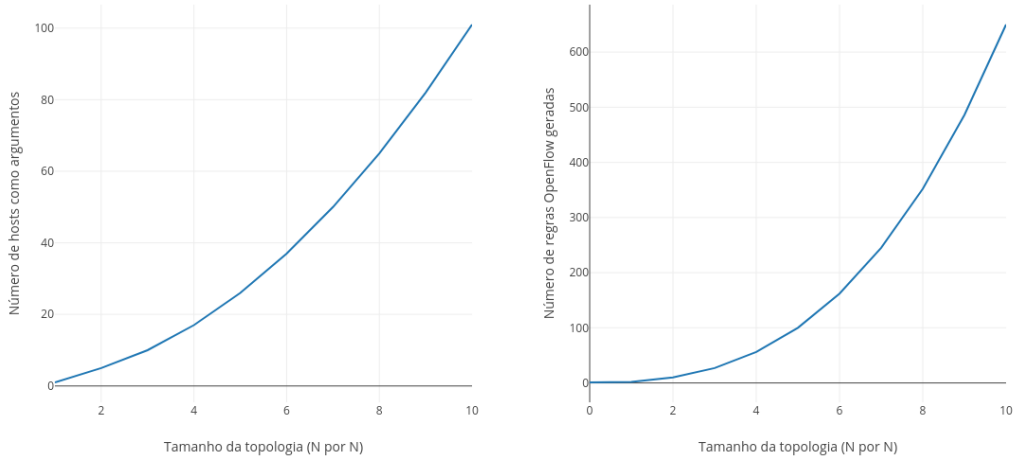
A Figura 5 ainda permite observar que, no estágio da transcompilação da política Merlin, o número de *hosts* de entrada está diretamente associado ao tamanho da topologia. As diretivas de orquestração do Havox permitem que o operador se abstraia da definição desses *hosts*, usando apenas, para qualquer tamanho de topologia, dois parâmetros para definir a política de saída da rede: o *switch* de saída e os campos de correspondência. Caso esta descrição fosse realizada diretamente no Merlin, o número de argumentos de *hosts* cresceria exponencialmente com o número de *switches* da topologia, conforme mostra a Figura 6(a). Além disso, a DSL do Merlin está limitada a operar com *hosts* de origem e destino definidos por IPs estáticos, reduzindo bastante seus cenários de aplicação. O Havox expande o uso do Merlin para cenários mais abrangentes contemplando múltiplos ASes, sem a necessidade de se explicitar IPs de origem e destino. As redes podem ser remotas, conhecidas apenas por seus prefixos.

Observou-se que, para topologias em grade de tamanhos n por n, como 4 por 4 em diante, contendo ainda um *switch* de saída mais à direita, a quantidade de regras gerada por diretiva de orquestração respeita a seguinte equação:

$$n_r = \sum_{i=1}^{t} t(2 + i - 1) \tag{1}$$

onde t é o "grau" da topologia em grade (por exemplo, numa topologia 4 por 4, $t = 4$).

Com base nessa equação, para topologias em grade com tamanhos laterais maiores, o número de regras OpenFlow criadas cresce exponencialmente como mostra a Figura 6(b). Com o Havox, neste caso, haverá apenas uma única diretiva de orquestração para qualquer topologia, o que reduz substancialmente o esforço de programação da rede.



(a) (b)

Figura 6. Transcompilação de uma diretiva "exit" em função da topologia: (a) número de argumentos do tipo "host" gerados para o Merlin; e (b) número de regras OpenFlow geradas para o conjunto de *switches*.

4. Trabalhos relacionados

Há outras iniciativas para redes OpenFlow com o intuito de elevar a abstração das camadas inferiores e diminuir o esforço de programação da rede [Xia et al. 2015, Trois et al. 2016].

O Frenetic [Foster et al. 2013] é uma plataforma de programação que fornece um conjunto de métodos para abstração da camada de dados e sintaxes de alto nível para a definição de políticas e de consultas ao estado da rede. A solução permite ao operador programar aplicações operando sobre o controlador, através da API fornecida. Essa abstração é implementada por um processo de compilação de duas etapas que traduz o código do operador para políticas de encaminhamento intermediárias, que são então traduzidas por um sistema de tempo real em regras OpenFlow. O Merlin possui similaridades com o Frenetic, porém é mais genérico quanto ao gerenciamento da rede, com funcionalidades adicionais. Mesmo assim, tanto o Frenetic quanto o Merlin não contemplam as funcionalidades de orquestração do Havox.

O projeto Propane [Beckett et al. 2016] se firma nas mesmas motivações de SDN, apesar de não empregar o mesmo conceito. O Propane é um compilador que traduz políticas descritas numa DSL própria em configurações BGP distribuídas nos roteadores do AS. 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 que avaliarão se o código é válido no escopo do domínio e se fere restrições lógicas e físicas do mesmo. O resultado da compilação são as configurações individuais e textuais das instâncias BGP que operam no AS.

Outra iniciativa envolvendo o BGP é o projeto SDX [Gupta et al. 2015], que implementa um PTT (Ponto de Troca de Tráfego) com um *switch* OpenFlow interligando

roteadores de borda dos ASes participantes. O SDX oferece uma abstração onde cada AS tem a visão de um *switch* OpenFlow virtual próprio, para compilação de regras a partir de políticas escritas em alto nível. 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.

Como uma expansão para a arquitetura RouteFlow, foi implementado o RouteFlow Aggregator [Corrêa et al. 2012], que 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 único arquivo de configuração que prevê a conectividade do AS em questão com os vizinhos.

5. Conclusão

Este trabalho apresentou o serviço Havox e como ele auxilia o operador de um domínio SDN a programar o comportamento da rede, gerando múltiplas regras OpenFlow a partir de uma linguagem de configuração legível e de fácil uso para fins de orquestração. Para tal, o Havox amplia e integra o RouteFlow e o Merlin, que são, respectivamente, uma plataforma de roteamento IP e um *framework* gerenciador de regras de fluxo, ambos sobre redes OpenFlow. Os resultados que validam a proposta foram obtidos a partir de testes sobre topologias em grade, a partir das quais foi possível obter uma equação que prevê o crescimento exponencial do número de regras OpenFlow geradas por uma única diretiva Havox, conforme o tamanho desta topologia aumenta. Evidencia-se assim que, para redes maiores, a economia de esforço de programação se torna significativa, permitindo que o operador foque na definição das políticas de orquestração em vez de na criação de inúmeras regras OpenFlow.

Como trabalhos futuros, serão implementadas outras diretivas de orquestração, como a de descarte de pacotes, e estuda-se a implementação de uma interface gráfica que permita ao operador definir as diretivas de orquestração enviadas pelo RFHavox à API, no lugar do uso de arquivos.

Referências

Beckett, R., Mahajan, R., Millstein, T., Padhye, J., and Walker, D. (2016). Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 328–341. ACM.

Corrêa, C., Lucena, S., Rothenberg, C., and Salvador, M. (2012). Uma plataforma de roteamento como serviço baseada em redes definidas por software. In *Workshop de Gerência e Operação de Redes e Serviços (WGRS). SBRC*, volume 12.

Foster, N., Guha, A., Reitblatt, M., Story, A., Freedman, M. J., Katta, N. P., Monsanto, C., Reich, J., Rexford, J., Schlesinger, C., et al. (2013). Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134.

Gupta, A., Vanbever, L., Shahbaz, M., Donovan, S. P., Schlinker, B., Feamster, N., Rexford, J., Shenker, S., Clark, R., and Katz-Bassett, E. (2015). Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344.

Nascimento, M. R., Rothenberg, C. E., Denicol, R. R., Salvador, M. R., and Magalhaes, M. F. (2011). Routeflow: Roteamento commodity sobre redes programáveis. *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuudos-SBRC*.

ONF (2012). Software-defined networking: The new norm for networks. *Open Networking Foundation White Paper*, 2:2–6.

Rothenberg, C. E., Nascimento, M. R., Salvador, M. R., Corrêa, C. N. A., Cunha de Lucena, S., and Raszuk, R. (2012). Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 13–18. ACM.

Schordan, M. and Quinlan, D. (2003). A source-to-source architecture for user-defined optimizations. In *JMLC*, volume 3, pages 214–223. Springer.

Soulé, R., Basu, S., Marandi, P. J., Pedone, F., Kleinberg, R., Sirer, E. G., and Foster, N. (2014). Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM.

Trois, C., Del Fabro, M. D., de Bona, L. C., and Martinello, M. (2016). A survey on sdn programming languages: toward a taxonomy. *IEEE Communications Surveys & Tutorials*, 18(4):2687–2712.

Wickboldt, J. A., De Jesus, W. P., Isolani, P. H., Both, C. B., Rochol, J., and Granville, L. Z. (2015). Software-defined networking: management requirements and challenges. *IEEE Communications Magazine*, 53(1):278–285.

Xia, W., Wen, Y., Foh, C. H., Niyato, D., and Xie, H. (2015). A survey on software-defined networking. *IEEE Communications Surveys & Tutorials*, 17(1):27–51.