Orquestrando tráfego em redes OpenFlow com o Havox

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. Inserido nesse escopo da geração múltipla de regras, 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 para se definir o comportamento da rede.*

Abstract. *Software-defined networking has been providing more control to network operators, but generating several OpenFlow rules simultaneously is yet a tough job. In this scope of multiple rules generation, this work presents Havox, a service meant to help with traffic orchestration through a simple, high level configuration language based on OpenFlow fields. Tests 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 needed to orchestrate the network behavior.*

1. Introdução

Os sistemas autônomos (ASes) têm suas próprias estratégias de negócios no que tange ao tráfego de pacotes que passa por sua rede. Entretanto, um AS tradicional não é capaz 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) uma nova perspectiva de controle sobre o tráfego se abre a partir da manutenção da lógica de decisão em um elemento controlador centralizado, permitindo que um controle mais granular sobre os pacotes pudesse ser exercido. Soluções como plataformas de roteamento IP e linguagens específicas de domínio (DSL) [Mernik et al. 2005], que facilitam a configuração da rede, ganham destaque neste novo cenário ao lidar com a complexidade intrínseca de configuração da rede, viabilizando o emprego de SDN em cenários reais e já maduros de roteamento.

Centrado nesses pontos, propõe-se neste trabalho o Havox, 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 Ruby. Através dessa linguagem, define-se a forma como o encaminhamento do tráfego se dará, e ela ainda pode ser aprimorada com novas funcionalidades, conforme os casos de uso.

Os experimentos realizados mostram que uma diretiva de orquestração do Havox, com reduzido número de parâmetros, é capaz de gerar múltiplas regras OpenFlow, com número crescente de regras conforme o número de *switches* aumenta.

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 e a Seção 4 traz as conclusões e os trabalhos futuros.

2. Proposta

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[1] em contêineres, dispostas em uma topologia idêntica à dos *switches* 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*. As mensagens internas da plataforma são comutadas através de um canal de comunicação interprocesso (IPC) implementado em um banco de dados não relacional MongoDB.

No Havox, 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 serviço pelo módulo RFHavox, elemento adicionado ao RouteFlow que integra a plataforma à arquitetura. O RFHavox é executado depois que os demais módulos já entraram em operação, quando ele fará uma requisição contendo os arquivos citados anteriormente ao Havox, e receberá novas regras OpenFlow geradas com base neles. Na sequência, o módulo injeta as mensagens contendo essas regras no IPC do RouteFlow para 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. No *switch* de saída há regras que removem a ID de VLAN e encaminham o pacote para o *host* de destino. O Havox expande o escopo de uso do Merlin para o cenário de um AS, onde estes *hosts* passam a representar roteadores de ASes vizinhos ou sub-redes.

A arquitetura usada é ilustrada na Figura 1. Seu cerne é o serviço Havox, que possui uma API *web* que recebe requisições HTTP com parâmetros bem definidos e res-

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

ponde com mensagens em formato padronizado JSON, fazendo assim o intermédio entre o serviço e o RouteFlow.

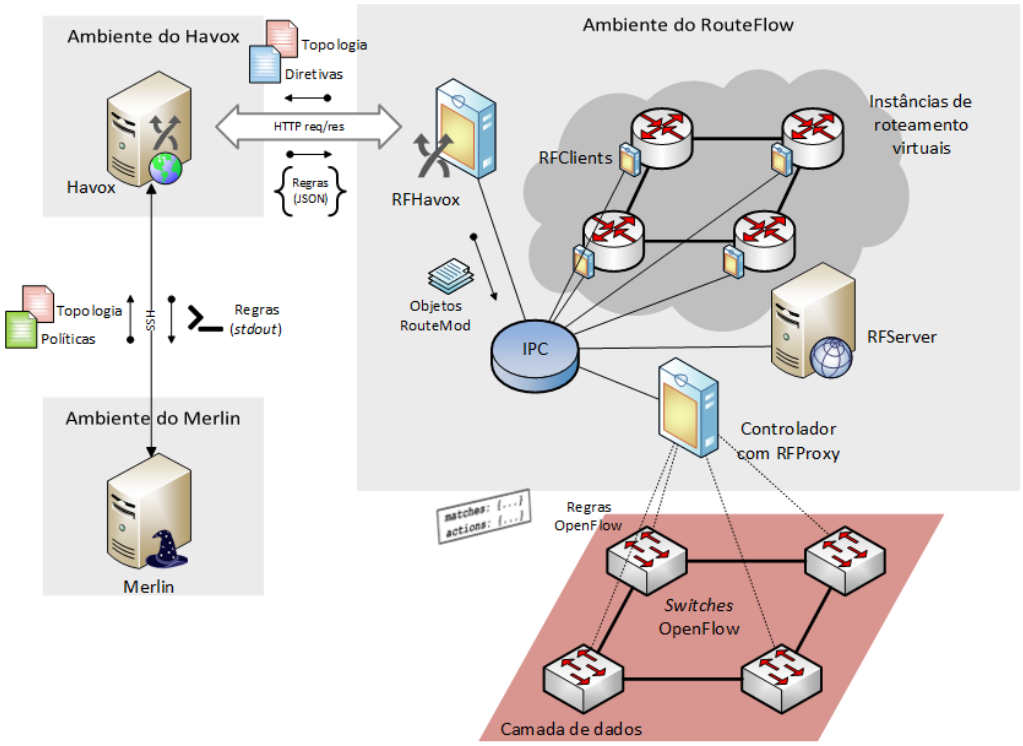


Figura 1. A arquitetura Havox.

Na Figura 1, a sequência de passos se inicia quando o processo do RouteFlow invoca o módulo RFHavox. Este envia uma requisição à API *web* do Havox, contendo os arquivos de diretivas e de topologia criados pelo operador. O Havox realiza a transcompilação das diretivas em políticas Merlin e abre uma conexão SSH com a máquina que executa o Merlin para enviar o arquivo de políticas transcompilado e o arquivo de topologias. Nessa mesma conexão, o serviço coleta as regras primitivas da saída padrão gerada pelo processo de compilação do Merlin. As regras primitivas são estruturadas e tratadas pelo Havox e em seguida são entregues ao RFHavox em uma resposta HTTP pela API em formato JSON. O módulo RFHavox extrai as regras da mensagem JSON, as transforma em objetos RouteMod e as enfileira para processamento pelos demais módulos do RouteFlow. Oportunamente, as regras são instaladas nos *switches* OpenFlow da camada de dados. Essas regras especiais, por serem derivadas de diretivas definidas pelo operador, possuirão valores de prioridade maior do que os das regras convencionais criadas pelo RouteFlow.

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 e os *switches*, os roteadores do domínio. Já 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.

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

Código 1. Exemplo de arquivo de diretivas da arquitetura.

2.2. A linguagem de configuração

Um arquivo escrito na linguagem da arquitetura contém um conjunto de diretivas de orquestração, 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*. 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. 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*. 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*. A tabela 1 lista as diretivas.

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.

Os campos suportados no argumento de bloco da diretiva `exit` são: `destination_ip` (*str*), `destination_mac` (*str*), `destination_port` (*int*), `ethernet_type` (*int*), `in_port` (*int*), `ip_protocol` (*int*/*str*), `source_ip` (*str*), `source_mac` (*str*), `source_port` (*int*), `vlan_id` (*int*) e `vlan_priority` (*int*).

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. Esta linguagem pode ser expandida adicionando-se métodos para novas funcionalidades. Toda a biblioteca é coberta 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 `https://github.com/rodrigosoares/`

```
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.

`havox`. Cada diretiva de orquestração é então transformada em trechos de políticas na DSL do Merlin. No exemplo do Código 1, a biblioteca 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. O Código 2 transcompilado é então submetido ao Merlin, que o compilará 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.

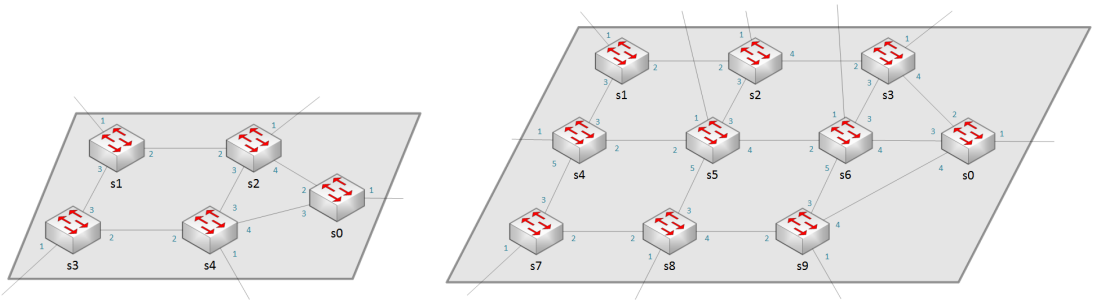
2.3. Execução

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*.

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 método POST é enviada para a URL da aplicação *web* da API e contém os arquivos de diretivas e de topologia. 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.

Depois que o Merlin compila as políticas em regras OpenFlow primitivas, tem início a etapa de tratamento e estruturação 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, é identificado o ID do *switch* ao qual ela é destinada e são avaliadas suas correspondências com valores e suas ações com argumentos. Depois, o tratamento seguirá conforme os parâmetros passados na requisição. Os nomes dos campos são então traduzidos para os nomes análogos do RouteFlow e as regras são encapsuladas em uma mensagem JSON que formará a resposta da API *web*.

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 deste como objetos RouteMod com uma prioridade superior às das demais regras



(a) (b)

Figura 2. Topologias 2 por 2 e 3 por 3, ambas com *s0* à direita.

básicas para, então, serem instaladas nos devidos *switches*.

3. Demonstração e validação experimental

A demonstração estenderá o trabalho da prova de conceito, onde foram realizados experimentos processando a primeira diretiva de orquestração do Código 1, além da diretiva de definição da topologia. Para a demonstração, serão também executadas outras diretivas deste código. Durante a prova de conceito, a diretiva única foi testada para as topologias em grade contidas nas Figuras 2(a) e 2(b). Em ambos os casos, todos os *switches* são vinculados a *hosts*, que representam roteadores de ASes vizinhos ou sub-redes, e os *switches* da coluna à direita possuem enlace com o *switch s0* de saída. Estas topologias foram implementadas através do MiniNExT[2]. 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[3] e observando o trajeto dos fluxos pelos contadores de regras OpenFlow.

No caso da topologia 2 por 2, a diretiva de orquestração derivou 10 regras OpenFlow, sendo 4 regras de entrada, 2 regras intermediárias e 4 regras de saída. Já na topologia 3 por 3, foram geradas 27 regras OpenFlow, sendo em 9 regras de entrada, 9 regras intermediárias e 9 regras de saída. Nos *switches* de entrada 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. Os *switches* que atuam como intermediários verificarão a correspondência da ID de VLAN definida nos *switches* anteriores e farão o repasse em direção a *s0*, caso correspondam. Nos *switches* intermediários, somente a ID de VLAN é avaliada, pois esse atributo passa a atuar como um rótulo do tráfego. 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.

A Figura 3 mostra a derivação da diretiva nas regras para a topologia 2 por 2 e permite observar que, no estágio da transcompilação da política Merlin, o número de *hosts* de entrada está associado exponencialmente ao tamanho lateral da topologia, como mostra a Figura 4(a). Porém, as diretivas de orquestração do Havox permitem que o operador se abstraia da definição desses *hosts*, usando apenas, para qualquer tamanho de

[2]Disponível em `https://github.com/USC-NSL/miniNExT`.
[3]Disponível em `https://iperf.fr/`.

topologia, dois parâmetros para definir a política de saída da rede: o *switch* de saída e o bloco com os campos de correspondência.

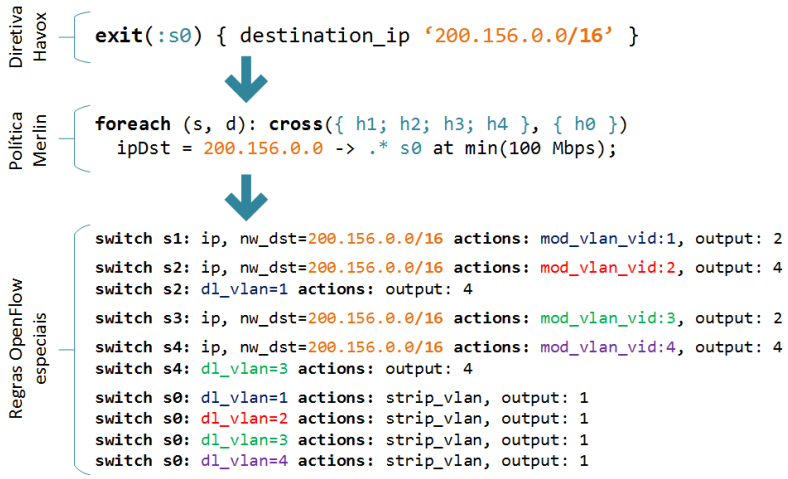


Figura 3. Transformação da diretiva em regras na topologia 2 por 2.

Observou-se ainda que, para topologias em grade de tamanhos laterais maiores, como 4 por 4 em diante, com 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 tamanho da topologia em grade e o iterador i varia conforme a coluna de *switches* "se afasta" do *switch* de saída *s0*, o que aumenta o número de *switches* intermediários no caminho. Com base nessa equação, para topologias em grade com tamanhos laterais maiores, o número de regras OpenFlow criadas também cresce exponencialmente, como ilustra a Figura 4(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.

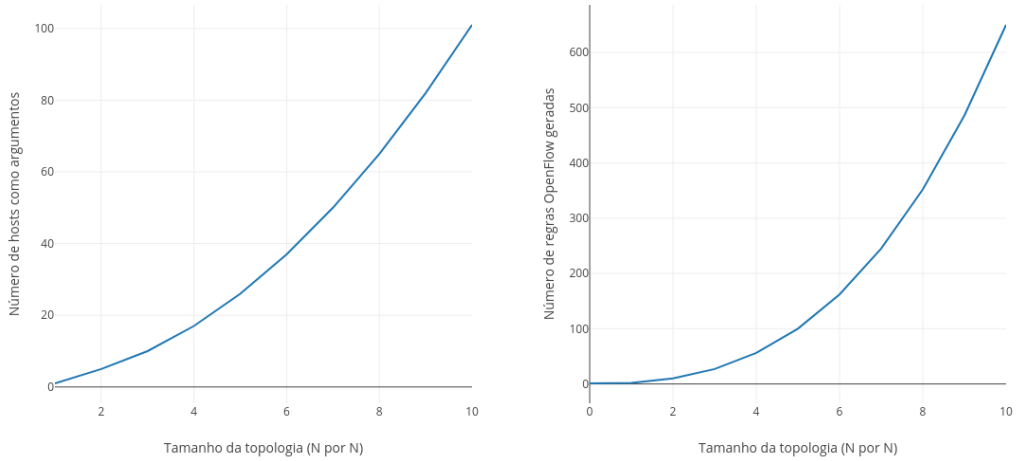
3.1. URLs e recursos necessários

As URLs referentes à documentação de instalação, ao código fonte e a demais documentos de apoio se encontram no repositório da ferramenta: `https://github.com/rodrigosoares/havox`. Lá também se encontra a URL para o vídeo demonstrativo, localizado em `https://youtu.be/Rtj7AjH5V6U`.

Para fins de demonstração, são necessárias quatro máquinas distintas, virtuais ou não, cada qual executando um componente da arquitetura. Para acomodação de todas as máquinas virtuais, a máquina hospedeira deve ter processador equivalente a i5 ou superior, com mínimos de 500 GB de espaço de armazenamento e 8 GB de memória RAM, e rodar sistema operacional Ubuntu 16.04 dotado de VirtualBox versão 5.0 ou superior.

4. Conclusão e trabalhos futuros

Este trabalho apresentou o serviço Havox, que auxilia o operador de um domínio SDN a programar o comportamento da rede a partir de uma linguagem de configuração legível e



(a) (b)

Figura 4. Gráficos com base no tamanho lateral da topologia.

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 experimentais evidenciam assim que há uma significativa redução de esforço de programação da rede, 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

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 Distribuıdos-SBRC*.

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.

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.