

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Lucas Borges Fernandes

**Roteamento Multicaminhos em Redes
Definidas por Software Usando P4**

Uberlândia, Brasil

2017

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Lucas Borges Fernandes

**Roteamento Multicaminhos em Redes Definidas por
Software Usando P4**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau
de Bacharel em Ciência da Computação.

Orientador: Lásaro Jonas Camargos

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2017

Lista de abreviaturas e siglas

API	Application Programming Interface
SDN	Software Defined Networks
QoS	Quality of Service
PSA	Portable Switch Architecture
P4	Programming Protocol-Independent Packet Processors
OSS	Open-Source Software
TCP	Transmission Control Protocol
WDM	Wavelength Division Multiplexing
ECMP	Equal-Cost Multi-Path Algorithm
JSON	Javascript Object Notation
MAC	Media Access Control
IP	Internet Protocol

Sumário

1	INTRODUÇÃO	5
1.1	Objetivos	6
1.1.1	Objetivo Geral	6
1.1.2	Objetivos específicos	6
1.2	Justificativas	6
1.3	Método	7
1.4	Etapas	7
1.4.1	Análise da Literatura	7
1.4.2	Proposição da nova arquitetura	7
1.4.3	Desenvolvimento da nova arquitetura	7
1.4.4	Validação da Proposta	8
2	FUNDAMENTAÇÃO TEÓRICA	9
2.1	Referencial Teórico	9
2.1.1	Redes Definidas por Software - SDN	9
2.1.2	OpenFlow	10
2.1.3	Linguagem P4	10
2.1.4	Mininet	11
2.2	Trabalhos Correlatos	11
2.2.1	Soluções de segurança em redes utilizando a linguagem P4	11
2.2.2	Roteamento multicaminho para provisão eficiente de recursos interdomínio com qualidade de serviço em redes ópticas WDM	11
2.2.3	ECMP	12
3	DESENVOLVIMENTO	13
3.1	Arquitetura P4	13
3.1.1	Software Interpretador	13
3.1.2	Compilador	13
3.1.3	Linguagem	15
3.1.3.1	Parser	16
3.1.3.2	Table Match	17
3.1.3.3	Table Action	17
3.2	Ambiente de desenvolvimento	18
3.2.1	Instalação	18
3.2.2	Geração do projeto	19
3.2.3	Adaptação do projeto	20

3.2.4	Execução do projeto	20
3.2.5	Considerações	22
3.3	Projetos Desenvolvidos	22
3.3.1	Contador de pacotes por destino	22
3.3.2	Contador de pacotes por porta física	24
3.3.3	Contador de pacotes por drop programado	25
3.3.4	Multicaminhos utilizando abordagem Probabilística	27
3.3.5	Multicaminhos utilizando abordagem Round-Robin	33
3.3.6	Multicaminhos utilizando abordagem Mista	36
4	VALIDAÇÃO DA PROPOSTA	41
4.1	Multicaminhos utilizando abordagem Probabilística	41
4.1.1	Comandos	41
4.1.2	Resultados	43
4.1.3	Conclusão	44
4.2	Multicaminhos utilizando abordagem Round-Robin	45
4.2.1	Comandos	45
4.2.2	Resultados	46
4.2.3	Conclusão	46
4.3	Multicaminhos utilizando abordagem Mista	46
4.3.1	Comandos	47
4.3.2	Resultados	48
4.3.3	Conclusão	49
5	TRABALHOS FUTUROS	50
6	CONCLUSÃO	51
	Referências	52

1 Introdução

O aumento da popularidade das redes de computadores como por exemplo a *Internet*, acarretou um aumento considerável no número de serviços oferecidos que as utilizam, tornando necessária uma maior preocupação com relação a por exemplo performance e disponibilidade. Para que ações sejam tomadas para mitigar tais problemas, é necessário primeiro que se tenham métricas da rede e posteriormente maneiras de configurá-la para atingir determinado fim.

Tradicionalmente, as redes de computadores possuem o Plano de Dados acoplado com o Plano de Controle, ou seja, todo o mecanismo responsável pelas decisões de roteamento estão fisicamente presentes no mesmo lugar que os mecanismos responsáveis por exemplo pela leitura de pacotes. Esta configuração apresenta limitações significativas para o gerenciador da rede, que deve configurar e obter métricas de cada comutador individualmente [5].

Uma alternativa que vêm ganhando muito espaço na indústria são as Redes Definidas por Software, ou *SDN* em inglês. Estas são redes cujo Plano de Controle está fisicamente desacoplado do Plano de Dados, de forma que é possível configurar a partir de um *software* central os comutadores da rede, e também obter métricas gerais de forma fácil e dinâmica [5].

Neste trabalho apresenta-se uma alternativa ao protocolo *SDN* mais utilizado atualmente, cujo nome é OpenFlow. Para isto, foi tomado como base o projeto desenvolvido por [7], que possuiu como objetivo implementar uma plataforma *SDN* com módulos de monitoramento de rede e decisões de rotas multicaminhos.

Este trabalho se limita a implementar uma alternativa ao módulo de decisão de rotas multicaminhos. A decisão de substituir o protocolo OpenFlow surgiu da necessidade de se encontrar uma maneira mais dinâmica para definir uma Rede Definida por Software, uma vez que o protocolo OpenFlow está cada vez mais complexo devido a sua natureza generalista, que faz com que seja necessário que o protocolo se ajuste ao longo do tempo para suportar novos cabeçalhos e comutadores de fabricantes diferentes [13].

A alternativa escolhida para este trabalho foi uma arquitetura que utiliza a linguagem P4. Esta foi criada com o intuito de se programar o Plano de Dados de um comutador utilizando uma linguagem de programação [3], de forma também a prover uma API única para a inserção de regras de roteamento, esta que pode ser acessada remotamente por um *software* controlador.

Uma das vantagens da utilização de um ecossistema P4 se dá na ausência da

necessidade de se adequar a comutadores de fabricantes diferentes, uma vez que os comutadores utilizados devem obrigatoriamente comportar programas P4. Outra vantagem está no poder de configuração que o gerenciador da rede P4 possui com a linguagem.

Portanto, neste trabalho serão apresentadas implementações de algoritmos de decisão de rotas multicaminho utilizando a linguagem P4, de forma a validar experimentalmente que é possível substituir a implementação mostrada em [7], que utiliza OpenFlow, por uma que utiliza inteiramente um ecossistema P4.

1.1 Objetivos

1.1.1 Objetivo Geral

Implementar infraestrutura para roteamento multicaminhos em comutadores baseados em P4, de forma simples, eficiente, dinâmica e configurável. Este trabalho pode ser visto como uma implementação parcial do trabalho de [7], mas em uma arquitetura de Rede Definida por Software usando a linguagem P4, em vez de OpenFlow.

1.1.2 Objetivos específicos

- Implementar algoritmos de seleção de rotas multicaminhos utilizando apenas a linguagem P4 e uma rede de comutadores P4;
- Validar experimentalmente a implementação proposta;
- Disponibilizar à comunidade *Open-Source Software (OSS)* as implementações deste trabalho para que possam ser usadas tanto como base para melhoramentos como para aprendizado das tecnologias desenvolvidas.

1.2 Justificativas

O protocolo OpenFlow é historicamente a alternativa mais utilizada para a construção de Redes Definidas por Software. No entanto, com o advento de novos protocolos de rede, novos tipos de comutadores e necessidades cada vez mais específicas, o protocolo precisou se adaptar de tal forma que hoje ele vêm se tornando cada vez mais complexo.

Esse problema despertou na indústria a necessidade de se construir uma alternativa mais flexível que independe de fabricante, cabeçalhos disponíveis e comutadores pré-programados. Dessa necessidade foi concebida a linguagem P4, que da mesma forma que hoje linguagens de programação conseguem ser executadas em diferentes máquinas, ela consegue o mesmo feito com diferentes comutadores, desde que os mesmos sejam construídos utilizando a arquitetura *Portable Switch Architecture (PSA)* [6].

Essa liberdade permite que os programadores de Redes Definidas por Software consigam dedicar mais tempo na implementação de suas regras de negócio do que possibilitando a integração dos diferentes comutadores que compõem sua rede. Além disso, com a utilização da linguagem P4, os programadores possuem maior poder de configuração, uma vez que não existem rotinas pré-configuradas por fabricantes e que a linguagem dá acesso total às rotinas internas do comutador.

Portanto, este trabalho visa provar que é possível substituir o módulo multicaminhos definido utilizando o protocolo OpenFlow no trabalho de [7] por um módulo escrito apenas na linguagem P4.

1.3 Método

Neste trabalho seguiremos um método experimental, com desenvolvimento de uma prova de conceito e validação da mesma por meio de uma rede P4 definida utilizando a tecnologia *Mininet* e topologias simples.

1.4 Etapas

1.4.1 Análise da Literatura

Análise da literatura focando nas seguintes áreas: Redes definidas por Software, Linguagem P4, OpenFlow e algoritmos multicaminhos.

Será necessário entender todas as tecnologias e processos que serão utilizados durante o desenvolvimento do trabalho. Uma pesquisa extensiva na área de Redes Definidas por Software deverá ser feita, tendo como foco as diferenças e vantagens da utilização do protocolo OpenFlow e da Linguagem P4.

1.4.2 Proposição da nova arquitetura

Proposição da nova arquitetura que utilizará a linguagem P4 ao oposto da atual desenvolvida em [7], que utiliza somente o protocolo OpenFlow.

Após o entendimento das tecnologias envolvidas no processo, será necessário desenvolver provas de conceito simples utilizando a linguagem P4, e após o seu domínio, a nova arquitetura multicaminhos deverá ser esquematizada para o seu desenvolvimento.

1.4.3 Desenvolvimento da nova arquitetura

Após a definição da nova arquitetura e do domínio das tecnologias envolvidas, a mesma deverá ser desenvolvida.

1.4.4 Validação da Proposta

Com a nova arquitetura pronta, deverão ser realizados diversos testes com a finalidade de provar que é possível substituir a atual desenvolvida por [7] pela desenvolvida neste trabalho.

2 Fundamentação Teórica

Este capítulo explica os conceitos que dão base a este projeto, assim como todas as tecnologias que serão utilizadas no seu desenvolvimento. Também serão apresentados os trabalhos correlatos.

2.1 Referencial Teórico

Nesta seção, serão explicados os conceitos fundamentais para o entendimento deste projeto, como por exemplo as Redes Definidas por Software, o protocolo OpenFlow, a linguagem P4 e suas dependências e o simulador de redes Mininet.

2.1.1 Redes Definidas por Software - SDN

As Redes Definidas por Software, ou *Software Defined Networks (SDN)* [9] representam hoje uma nova abordagem para o gerenciamento de redes. Tradicionalmente o Plano de Dados e o Plano de Controle de um comutador estão unidos dentro de um mesmo hardware, ou seja, toda a regra de encaminhamento está contida no mesmo dispositivo que contém os mecanismos de roteamento.

Tal arquitetura dificulta o gerenciamento da rede no sentido de que para configurá-la para determinado fim, é provável que tenha-se que alterar a configuração de diversos pontos do sistema manualmente, em contraste com o encontrado em uma *SDN*, que possibilita alterar todos os pontos da rede a partir de um único software controlador.

As Redes Definidas por Software visam separar o Plano de Controle do Plano de Dados, de forma a isolar a lógica de encaminhamento do hardware, possibilitando assim que o gerenciador da rede possa configurá-la de forma muito mais dinâmica, prática, barata e adaptável. [2]

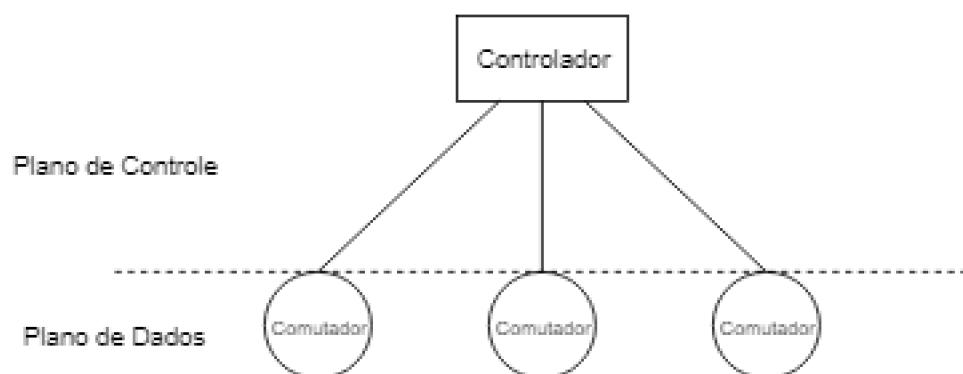


Figura 1 – Ilustração da separação do Plano de Controle do Plano de Dados.

Algumas aplicações das *SDN* são o provisionamento de qualidade de serviço (*QoS*) e obtenção de métricas da rede de forma mais eficiente.

2.1.2 OpenFlow

Atualmente as Redes Definidas por Software [9] são implementadas em sua maioria esmagadora utilizando o protocolo OpenFlow. Este protocolo é tido como um dos primeiros padrões adotados para as *SDN* e é indiscutivelmente o mais aceito na indústria. Ele provê acesso ao Plano de Dados ao mesmo tempo em que dispõe uma API simples para o Plano de Controle, que neste caso é representado por um *software* controlador da rede.

Um comutador OpenFlow [11] possui uma tabela de regras de encaminhamento de pacotes, onde o *match* é feito nos bits do cabeçalho do pacote e as ações são especificadas a partir deste. Ao receber um pacote, um comutador OpenFlow identifica o *match* associado e executa a regra. Esta tabela de regras é populada a partir do controlador que por sua vez é controlado pelo gerenciador da rede. [13]

O protocolo OpenFlow obteve grande aceitação da indústria, mas sua natureza generalista, que propõe o funcionamento da *SDN* independentemente dos comutadores utilizados, acabou por tornar o protocolo cada vez mais complexo, uma vez que novos protocolos, cabeçalhos e comutadores de diferentes fabricantes estão surgindo. [10]

2.1.3 Linguagem P4

A natureza generalista do protocolo OpenFlow, conforme mencionado anteriormente, acabou por tornar o protocolo cada vez mais complexo, de tal forma que viu-se necessária a criação de uma nova arquitetura de comutadores.

Esta arquitetura possibilita que os comutadores sejam totalmente programáveis por uma linguagem cuja finalidade é definir o Plano de Dados, ao mesmo tempo em que uma API é disponibilizada para o Plano de Controle, que por sua vez irá popular as regras de encaminhamento. Isso elimina a necessidade de preocupação com os cabeçalhos disponíveis e com o fabricante, uma vez que o Plano de Controle e o Plano de Dados foram criados com o propósito de conversarem entre si, mesmo desacoplados.

A linguagem foi criada com o nome P4, que significa *Programming Protocol-Independent Packet Processors*. A linguagem hoje está em constante desenvolvimento, tendo como versão mais recente a 16 e como mais utilizada e escolhida para este projeto a 14. [3]

Existem vários compiladores P4 e vários *softwares* para os comutadores, sendo os últimos os responsáveis por disponibilizar a API para o *software* controlador, cuja implementação ainda não foi padronizada.

2.1.4 Mininet

O Mininet se trata de um emulador de redes, desenvolvido por pesquisadores da Universidade de Stanford nos Estados Unidos, com o objetivo auxiliar o desenvolvimento de Redes Definidas por Software, para que qualquer pessoa possa as executar, avaliar e experimentar. Este se tornou uma ferramenta essencial para a o desenvolvimento de protótipos de soluções *SDN*, pois caso contrário seria necessário realizar a experimentação em um ambiente com comutadores reais. [12]

O emulador possibilita a criação de *links*, *hosts*, comutadores e controladores. Também é possível personalizá-los utilizando uma API escrita em Python.

2.2 Trabalhos Correlatos

Nesta seção, é apresentada uma visão geral sobre trabalhos relacionados ao escopo proposto por este projeto. São eles:

2.2.1 Soluções de segurança em redes utilizando a linguagem P4

Um exemplo de trabalho que utiliza a linguagem P4 para programar dispositivos de rede é o desenvolvimento de soluções de segurança, como por exemplo em [1]. Neste trabalho, foram adicionadas duas funções de criptografia na linguagem, de forma que os dispositivos encaminhadores de pacotes eram encarregados de criptografar os dados, tendo suporte aos protocolos Ethernet, IPv4 e TCP. Como resultado, as funções de criptografia obtiveram bons resultados, não acarretando nenhum atraso significativo no encaminhamento dos pacotes.

Como nosso trabalho, este adicionou ao *software* embutido no comutador métodos responsáveis por receber informações do programa P4 e executar algoritmos pré-configurados. Portanto, através deste foi possível verificar a possibilidade de criação de vários métodos multicaminhos.

2.2.2 Roteamento multicaminho para provisão eficiente de recursos interdomínio com qualidade de serviço em redes ópticas WDM

Roteamentos tradicionais de caminho único podem apresentar problemas de congestionamento e limitações relacionadas a banda passante, e para apresentar alternativas, várias pesquisas com foco em algoritmos multicaminhos foram desenvolvidas, como por exemplo em [4]. Nesta pesquisa, foram apresentados dois algoritmos multicaminhos: MRRTM e MRDT. Ambos apresentaram desempenho superior quando comparados com abordagens tradicionais de caminho único.

Diferentemente do trabalho proposto, este apresenta dois algoritmos de nome MR-RTM e MRDT, cujas implementações são baseadas em heurísticas que levam em conta a largura de banda e uma série de outras métricas. Para o nosso trabalho foram propostos algoritmos simples cuja única finalidade é validar a possibilidade de criação de variadas regras para escolha de multicaminhos utilizando a linguagem P4.

Contudo, o trabalho correlato possibilitou um maior conhecimento na área de Redes Definidas por Software com foco em aplicações de roteamento multicaminhos.

2.2.3 ECMP

Um algoritmo apresentado para a escolha de rotas em uma rede com suporte a multicaminhos foi apresentado em [8]. Com o nome de *Equal-Cost Multi-Path Algorithm*, ou ECMP, o algoritmo possui o objetivo de distribuir a carga entre os caminhos de forma igualitária, de forma a não sobrecarregar um único caminho.

Uma abordagem semelhante a apresentada pelo algoritmo ECMP pode ser alcançada por meio dos experimentos propostos no nosso trabalho. Duas delas são as abordagens Probabilística e a Mista, que caso configuradas para possuir a mesma probabilidade para cada rota, irão enviar virtualmente a mesma quantidade de pacotes para cada uma. Outra abordagem é a Round-Robin, que possui uma natureza sequencial e cíclica, de forma que a diferença entre o número de pacotes enviados a cada caminho para cada fluxo diferente não passará de 1.

3 Desenvolvimento

Nesse capítulo são apresentadas informações relacionadas ao processo de desenvolvimento da arquitetura P4 proposta, tais como os experimentos realizados, topologias utilizadas, ambiente de desenvolvimento e tecnologias envolvidas.

3.1 Arquitetura P4

A linguagem P4, juntamente com o *software* que a interpreta dentro do comutador e o seu compilador, são peças essenciais para o desenvolvimento da *SDN* proposta por este trabalho. Nesta seção cada parte da arquitetura será apresentada em detalhes.

3.1.1 Software Interpretador

A linguagem P4 por si só é responsável pela definição do Plano de Dados de um comutador. Ou seja, a linguagem detém o poder de definir as tabelas de roteamento, as ações possíveis a serem realizadas após determinado *match*, os contadores, cabeçalhos suportados, entre outros. Para que o programa P4 seja traduzido de um texto de extensão *.p4* para definições reais em um comutador, é necessário um compilador e um *software* dedicado para traduzir a saída gerada pelo primeiro em instruções para o comutador.

O *software* responsável por receber o *input* do compilador, ou melhor dizendo, o *software* interpretador, ainda não está em sua versão final, assim como todo o resto do ecossistema P4. No entanto, existe uma versão de certa forma consolidada, de nome *behavioral_model*¹. Esta versão foi a utilizada neste trabalho devido ao seu grau de maturidade e pelo fato de executar programas P4 na sua versão 14.

Durante o trabalho, foi necessário realizar um *fork* deste projeto. Isto aconteceu devido a necessidade de acrescentar alguns algoritmos necessários para o cálculo dos multicaminhos. Seu código está em *C++* e será exibido ao longo do trabalho.

3.1.2 Compilador

Conforme mencionado anteriormente, a linguagem P4 necessita de um compilador que gera o *input* necessário para o projeto *behavioral_model*. Este *input* é gerado em um formato *JSON* e carrega todas as informações necessárias para a geração das instruções do comutador.

¹ <https://github.com/p4lang/behavioral-model>

O formato *JSON* é apenas um requisito do projeto *behavioral_model*, e conforme falado anteriormente, não é o padrão do ecossistema P4 devido a atual volatilidade do mesmo.

O compilador utilizado neste trabalho possui o nome de *p4c-bm*² e um trecho do *JSON* gerado pelo mesmo pode ser visto na Listagem 3.1.

Listagem 3.1 – Trecho do arquivo JSON gerado pelo compilador p4c-bm.

```

1  "actions": [
2    {
3      "name": "set_destination",
4      "id": 0,
5      "runtime_data": [
6        {
7          "name": "dmac",
8          "bitwidth": 48
9        },
10       {
11         "name": "port",
12         "bitwidth": 9
13       }
14     ],
15     "primitives": [
16       {
17         "op": "modify_field",
18         "parameters": [
19           {
20             "type": "field",
21             "value": [
22               "ethernet",
23               "dstAddr"
24             ]
25           },
26           {
27             "type": "runtime_data",
28             "value": 0
29           }
30         ]
31       },
32       {
33         "op": "modify_field",
34         "parameters": [
35           {
36             "type": "field",
37             "value": [
38               "standard_metadata",
39               "egress_spec"
40             ]
41           },
42           {
43             "type": "runtime_data",
44             "value": 1
45           }
46         ]
47       }
48     ]

```

² <https://github.com/p4lang/p4c-bm>

49 }
50 1,

No trecho acima é possível observar uma lista de ações definidas pelo programa P4 compilado. Todos os nomes, parâmetros, tipos são especificados de forma a possibilitar a execução do programa no projeto *behavioral_model*.

3.1.3 Linguagem

Um programa escrito na linguagem P4 segue um fluxo pré-definido cujo entendimento é de vital importância para a definição do Plano de Dados de um comutador.

Primeiro, todo pacote recebido é tratado pelo *parser* do programa. Este possui como objetivo ler os cabeçalhos e detalhar os mesmos para que no próximo estágio seja possível utilizá-los para o *match* nas tabelas de roteamento.

Feito o *match* baseado nas regras inseridas a partir do Plano de Controle, o pacote se torna alvo da ação cadastrada. Esta pode realizar qualquer alteração nos cabeçalhos dos pacotes, desde alterar o *IP* destino até alterar o *IP* de origem. Após a execução da ação, o pacote passa para o fluxo de saída do comutador e é encaminhado para o seu destino.

Este fluxo está ilustrado na Figura 2.

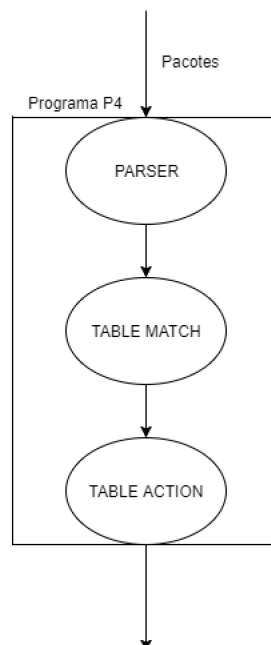


Figura 2 – Ilustração dos estágios de processamento de um pacote recebido por um comutador que está executando um código P4.

A seguir cada passo será descrito de forma mais detalhada a partir de exemplos:

3.1.3.1 Parser

Assim que um pacote qualquer entra no comutador P4, o primeiro passo a ser realizado é a leitura do cabeçalho presente. Existem duas construções responsáveis por este processo. A primeira é a construção *header*, que define os cabeçalhos esperados e a quantidade de bits associada aos mesmos. Um trecho de código exemplo pode ser visto na Listagem 3.2.

Listagem 3.2 – Trecho de um programa P4 responsável pela definição dos cabeçalhos esperados.

```
1 header_type ethernet_t {
2     fields {
3         dstAddr : 48;
4         srcAddr : 48;
5         etherType : 16;
6     }
7 }
8
9 header_type ipv4_t {
10     fields {
11         version : 4;
12         ihl : 4;
13         diffserv : 8;
14         totalLen : 16;
15         identification : 16;
16         flags : 3;
17         fragOffset : 13;
18         ttl : 8;
19         protocol : 8;
20         hdrChecksum : 16;
21         srcAddr : 32;
22         dstAddr : 32;
23     }
24 }
25
26 header ethernet_t ethernet;
27
28 header ipv4_t ipv4;
```

Na Listagem 3.2, são definidos dois tipos de cabeçalho, um para a parte *ethernet* e outro para a parte *ipv4*. Feito isso, o próximo passo é a leitura do pacote tendo como base esses cabeçalhos. A construção responsável por essa etapa possui o nome de *parser*, e pode ser vista na Listagem 3.3.

Listagem 3.3 – Trecho de um programa P4 responsável pela leitura do pacote tendo como base os cabeçalhos definidos.

```
1 #define ETHERTYPE_IPV4 0x0800
2
3 parser start {
4     return parse_ethernet;
5 }
6
7 parser parse_ethernet {
```

```
8     extract(ethernet);
9     return select(latest.etherType) {
10         ETHERTYPE_IPV4 : parse_ipv4;
11         default: ingress;
12     }
13 }
14
15 parser parse_ipv4 {
16     extract(ipv4);
17     return ingress;
18 }
```

Após a leitura do pacote, o mesmo é levado para a próxima etapa, cuja função é verificar se alguma regra definida no controlador especifica que o pacote em questão deve ser manipulado de alguma forma.

3.1.3.2 Table Match

Após a leitura detalhada dos pacotes, serão verificadas as entradas cadastradas das tabelas de roteamento. A definição das tabelas e as ações possíveis em caso de *match* são descritas no programa P4. No entanto, todas as entradas da tabela devem ser cadastradas via controlador, que neste trabalho utiliza uma API com comandos de manipulação pré-definidos pelo projeto *behavioral_model*. É possível observar o trecho de código responsável pela definição de uma tabela de roteamento na Listagem 3.4.

Listagem 3.4 – Trecho de um programa P4 responsável pela definição de uma tabela de roteamento.

```
1 table forward {
2     reads {
3         ipv4.dstAddr : exact;
4     }
5     actions {
6         set_destination;
7     }
8 }
```

O trecho de código acima define uma tabela de roteamento de nome *forward* cujo *match* é realizado comparando o cabeçalho de *IP* de destino. Caso o *match* ocorra, é possível realizar apenas uma ação, cujo nome é *set_destination*. A definição da ação é feita por meio de uma construção específica da linguagem P4 que será detalhada a seguir.

3.1.3.3 Table Action

Uma ação, ou *action*, está diretamente atrelada a uma tabela e possui o poder de alterar um pacote tendo como base parâmetros enviados pelo controlador ou até mesmo valores pré-definidos que nunca serão alterados em tempo de execução. Para a tabela *forward* de roteamento definida na seção anterior, foi especificada uma ação de nome *set_destination*, cuja implementação pode ser vista na Listagem 3.5.

Listagem 3.5 – Trecho de um programa P4 responsável pela definição de uma ação associada a uma tabela de roteamento.

```

1 action set_destination(dmac, port) {
2     modify_field(ethernet.dstAddr, dmac);
3     modify_field(standard_metadata.egress_spec, port);
4 }
```

Esta ação possui como objetivo alterar a porta física de saída do pacote e também o endereço *MAC* de destino. Ou seja, esta ação possui a única finalidade de encaminhar o pacote para o seu destino, baseando-se em uma regra definida pelo controlador. Esta que possui o seguinte formato:

```
table_add forward set_destination 10.0.0.10 => 00:04:00:00:00:00 1
```

Traduzindo, esta regra especifica que para todo pacote cujo *IP* de destino é 10.0.0.10, deve-se aplicar a ação *set_destination* passando como parâmetros o endereço *MAC* destino 00:04:00:00:00:00 e a porta física de saída 1.

3.2 Ambiente de desenvolvimento

Devido ao fato de ser uma tecnologia recente, a linguagem P4 ainda não possui um ambiente de desenvolvimento robusto e estável. Os existentes na comunidade ainda deixam a desejar quando o assunto é legibilidade e curva de aprendizado.

Portanto, foi concebida uma plataforma para desenvolvimento de programas P4, onde a integração com Mininet é levada em conta e toda a estrutura necessária para execução da rede, comandos e leitura de métricas é gerada automaticamente.

A plataforma foi nomeada p4-dev e possui sua base de código escrita em Python. Para a execução da rede composta por comutadores P4, foram utilizados os projetos *behavioral-model*, software executado pelo comutador P4, e *p4c-bm*, responsável pela geração da representação JSON dos programas P4.

A utilização da plataforma é dada em quatro passos, explicados a seguir.

3.2.1 Instalação

O primeiro passo é clonar o projeto com a ferramenta Git³ e, na raiz do projeto, executar o script de instalação:

```
$ python core/tools/build.py
```

³ <https://git-scm.com/>

Com isso, todas as dependências necessárias para a geração de projetos e execução das redes P4 Mininet terão sido instaladas.

3.2.2 Geração do projeto

Cada rede P4 é representada como um projeto na plataforma p4-dev. Ou seja, cada rede é isolada de outra de tal forma que o desenvolvedor possui total liberdade para modificar as topologias e regras de negócio sem ter que se preocupar com efeitos colaterais. Para gerar um novo projeto, basta executar o seguinte comando:

```
$ python core/tools/gen_project.py <project-name>
```

Feito isso, será criada uma pasta com o nome do novo projeto dentro de p4-dev/projects. Essa pasta possui a seguinte árvore de arquivos:

```
project-name
├── build/
├── images/
├── mininet/
│   └── project-name.py
├── p4/
│   └── project-name.p4
├── tools/
│   ├── apply_commands.py
│   ├── build_project.py
│   ├── run_cli.py
│   └── run_project.py
├── util/
│   └── commands.txt
└── DOC.md
```

Alguns arquivos são criados durante a geração do projeto. Estes são necessários para a execução do projeto, conforme será descrito posteriormente neste trabalho.

3.2.3 Adaptação do projeto

Conforme dito anteriormente neste trabalho, cada rede é representada por um projeto no ambiente de desenvolvimento p4-dev. Portanto, o desenvolvedor possui total liberdade para alterar a organização dos arquivos gerados, a topologia da rede e também a forma com que a rede é instanciada.

A organização dos arquivos não deve ser interpretada como a única correta. A intenção é somente agilizar o desenvolvimento por meio de uma estrutura que já se demonstrou eficiente.

Caso o desenvolvedor opte por continuar com a arquitetura sugerida, os passos para instanciamento da rede, aplicação de comandos e leitura de métricas será descrita no próximo tópico deste trabalho.

3.2.4 Execução do projeto

Para a execução do projeto, ambos os arquivos `project-name.py` e `project-name.p4` deverão estar populadas com o código Python de instanciamento da rede Mininet e código P4 que será executado no comutador, respectivamente.

Feito isso, o próximo passo é gerar a representação JSON do código P4 desenvolvido utilizando o compilador `p4c-bm`. Para isso, deve-se executar o seguinte comando de dentro da pasta raiz do projeto:

```
$ python tools/build_project.py
```

O programa `build_project.py` irá procurar pelo programa P4 de nome `project-name.p4` dentro da pasta `p4/` do projeto e alimentar o compilador com ele.

Após a compilação, deve-se executar a rede Mininet e alimentar os comutadores P4 com o JSON gerado. Vale lembrar que cada comutador estará executando o código do *behavioral-model* de forma distribuída, e o mesmo necessita dos arquivos JSON que representam o comutador em que estão contidos. Para executar a rede Mininet, deve-se executar o seguinte comando de dentro da pasta raiz do projeto:

```
$ python tools/run_project.py
```

Uma interface para comandos Mininet será criada, e a partir dela será possível realizar testes na rede, como por exemplo o envio de pacotes de um *host* para outro. No entanto, é necessário inserir entradas nas tabelas de roteamento. Caso contrário, o comutador em questão não saberá o que fazer com os pacotes que chegam até ele.

Para isso, é disponibilizado um programa Python que realiza a inserção de comandos por meio de uma interface pré-definida pelo projeto *behavioral-model*. No contexto da plataforma p4-dev, por padrão o comutador será alimentado com os comandos contidos dentro do arquivo `commands.txt`, presente na pasta `util/` do projeto. Para isso, basta executar o seguinte comando de dentro da raiz do projeto:

```
$ python tools/apply_commands.py
```

Vale ressaltar que não existe uma documentação oficial que descreve todos os comandos possíveis que um comutador P4 aceita. Para este trabalho, foi realizada a análise do código Python responsável pela aplicação dos comandos, que está presente no projeto *behavioral-model*. Alguns exemplos de comandos são:

```
table_add port_forward set_port 10.0.0.10 => 1
table_add port_forward set_port 10.0.1.10 => 2
```

Onde na primeira entrada `table_add` representa o comando de adição de uma entrada em determinada tabela, `port_forward` representa o nome da tabela, `set_port` representa a ação que deve ser tomada pelo comutador, `10.0.0.10` representa o *IP* destino em que a ação será executada e `1` representa o número da porta física para o qual o pacote será roteado.

Este trabalho utiliza parte dos comandos mais importantes para um comutador, mas ainda assim é necessário analisar o código Python responsável pela execução dos mesmos. Uma documentação completa de todos os comandos seria de extrema ajuda e, portanto, uma excelente idéia para trabalho futuro.

Após a execução dos comandos, a rede estará pronta para receber pacotes. Para isso, deve-se voltar à interface de comandos Mininet disponibilizada pela execução da rede. Supondo que existam dois hosts identificados como `h1` e `h2` na rede, um possível comando para a mesma é:

```
h1 ping -c 1000 -i 0.1 h2
```

Este comando realizará o disparo de 1000 pacotes de `h1` para `h2`, com um intervalo de 0.1 segundos entre cada.

O quarto e último passo da execução da rede é a leitura das métricas geradas. Uma possível forma de geração de métricas são os contadores. A linguagem P4 disponibiliza estruturas capazes de contabilizar certos eventos, como por exemplo o número de pacotes que passaram pelo comutador, com diferenciação de fluxo. Supondo que um contador de nome `my_counter` tenha sido criado no programa P4, para se ler o conteúdo do mesmo,

basta executar os seguintes comandos:

```
$ python tools/run_cli.py
```

Com isso, será aberta uma conexão Thrift entre o código Python e o comutador em questão. Uma interface será aberta e será possível aplicar comandos ao comutador de forma dinâmica. O comando necessário para se ler o contador `my_counter` é:

```
counter_read my_counter 0
```

Onde `counter_read` representa o comando de leitura de contadores, `my_counter` representa o nome do contador, e `0` representa o índice do contador que queremos ler. O resultado será algo semelhante a:

```
my_counter[0]=BmCounterValue(packets=31, bytes=3038)
```

Neste caso, 31 pacotes passaram pelo comutador, totalizando 3038 bytes.

3.2.5 Considerações

A plataforma `p4-dev` foi criada com o intuito de ajudar a comunidade P4, que apesar de pequena, vem crescendo bastante. Portanto, o código está disponível no GitHub e livre para qualquer contribuição.

3.3 Projetos Desenvolvidos

Ao todo, foram criados seis projetos, onde três foram utilizados para o aprendizado, e os outros três foram criados para validar a proposta de arquitetura multicaminhos utilizando P4 deste trabalho. Nessa seção, todos os projetos serão apresentados, e na próxima seção, os três últimos serão utilizados para a validação da proposta.

3.3.1 Contador de pacotes por destino

Projeto desenvolvido com o intuito de validar a hipótese de que é possível contabilizar pacotes de diversas maneiras utilizando apenas as construções nativas da linguagem P4. Neste caso, o intuito foi contabilizar pacotes com certo destino.

Para este projeto, foi construída uma topologia simples que contém apenas dois *hosts* e um comutador central, de forma a possibilitar o envio de pacotes de um *host* para outro.

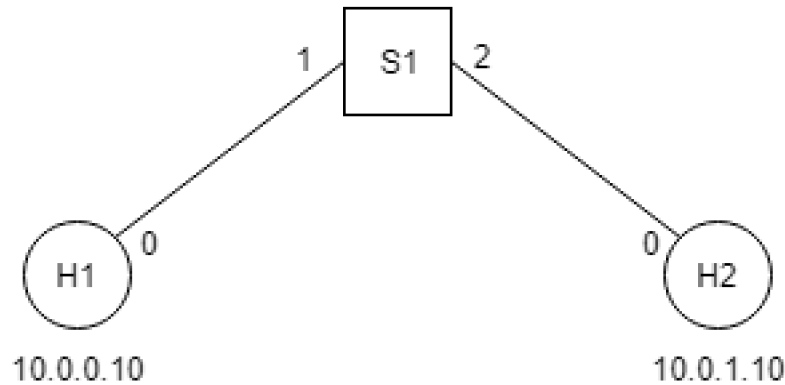


Figura 3 – Topologia utilizada para o projeto: Contador de pacotes por destino.

O código P4 exibido na Listagem 3.6 contém apenas uma tabela e um contador associado a mesma, de forma a contabilizar quantos pacotes passam por cada entrada na tabela.

Listagem 3.6 – Código fonte P4 do projeto: Contador de pacotes por destino.

```

1 #include "includes/headers.p4"
2 #include "includes/parser.p4"
3
4 counter my_direct_counter {
5     type: packets;
6     direct: forward;
7 }
8
9 action set_destination(dmac, port) {
10     modify_field(ethernet.dstAddr, dmac);
11     modify_field(standard_metadata.egress_spec, port);
12 }
13
14 table forward {
15     reads {
16         ipv4.dstAddr : exact;
17     }
18     actions {
19         set_destination;
20     }
21 }
22
23 control ingress {
24     apply(forward);
25 }

```

A tabela forward é indexada pelo *IP* de destino e possui somente uma ação possível. Como pode ser visto no código, assim que um pacote entra no comutador a primeira tabela a ser verificada é a forward. Caso um match seja encontrado, a ação `set_destination` será executada e os campos `ethernet.dstAddr`, que contém o endereço MAC destino, e `standard_metadata.egress_spec`, que contém a porta física de saída serão atualizadas com os dados definidos nos comandos a seguir:

```
table_add forward set_destination 10.0.0.10 => 00:04:00:00:00:00 1
table_add forward set_destination 10.0.1.10 => 00:04:00:00:00:01 2
```

3.3.2 Contador de pacotes por porta física

Projeto desenvolvido com o intuito de validar a hipótese de que é possível contabilizar pacotes de diversas maneiras utilizando apenas as construções nativas da linguagem P4. Neste caso, o intuito foi contabilizar pacotes por porta física.

Da mesma forma que o projeto anterior, a topologia deste foi construída de forma simples. Existem dois *hosts* e um comutador central, de forma a possibilitar o envio de pacotes de um *host* para outro.

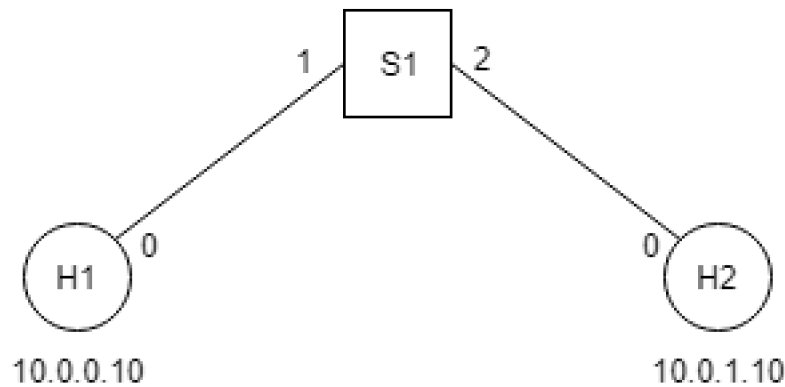


Figura 4 – Topologia utilizada para o projeto: Contador de pacotes por porta física.

O código fonte P4 exibido na Listagem 3.7 também se assemelha bastante ao anterior, com a diferença de que agora existe mais uma tabela a ser verificada.

Listagem 3.7 – Código fonte P4 do projeto: Contador de pacotes por porta física.

```
1 #include "includes/headers.p4"
2 #include "includes/parser.p4"
3
4 counter my_static_counter {
5     type: packets;
6     static: process_port;
7     instance_count: 1;
8 }
9
10 action count_port() {
11     count(my_static_counter, 0);
12 }
13
14 action set_destination(dmac, port) {
15     modify_field(ethernet.dstAddr, dmac);
16     modify_field(standard_metadata.egress_spec, port);
17 }
18
19 table process_port {
20     reads {
21         standard_metadata.ingress_port : exact;
```

```
22     }
23     actions {
24         count_port;
25     }
26 }
27
28 table forward {
29     reads {
30         ipv4.dstAddr : exact;
31     }
32     actions {
33         set_destination;
34     }
35 }
36
37 control ingress {
38     apply(process_port);
39     apply(forward);
40 }
```

A tabela `process_port` é encarregada de verificar qual a porta de entrada do pacote e possui apenas uma ação nomeada `count_port`, que como o próprio nome diz, é encarregada de contabilizar o pacotes oriundos de uma determinada porta física. Os comandos necessários para o funcionamento desta rede são os seguintes:

```
table_add process_port count_port 1 =>
table_add forward set_destination 10.0.0.10 => 00:04:00:00:00:00 1
table_add forward set_destination 10.0.1.10 => 00:04:00:00:00:01 2
```

Como pode ser observado mais uma vez, esta rede se diferencia da anterior apenas na tabela adicional, o que também reflete nos comandos necessários. Desta vez, estamos contabilizando todos os pacotes oriundos da porta física 1. Ou seja, todos os pacotes enviados pelo *host* h1.

3.3.3 Contador de pacotes por drop programado

Projeto desenvolvido com o intuito de validar a hipótese de que é possível contabilizar pacotes de diversas maneiras utilizando apenas as construções nativas da linguagem P4. Neste caso, o intuito foi contabilizar pacotes recusados de forma programática.

A única diferença da topologia desta rede em comparação com as duas anteriores está no fato de que existe um *host* a mais. Portanto, foi possível enviar pacotes para mais de um destino a partir de determinado host.

A nível de código, este projeto se assemelha bastante ao primeiro no sentido de que existe somente uma tabela, porém ambos diferem no número de ações possíveis. Neste, é possível que pacotes destinados a determinado *IP* sejam recusados arbitrariamente. O código fonte pode ser visto na Listagem 3.8.

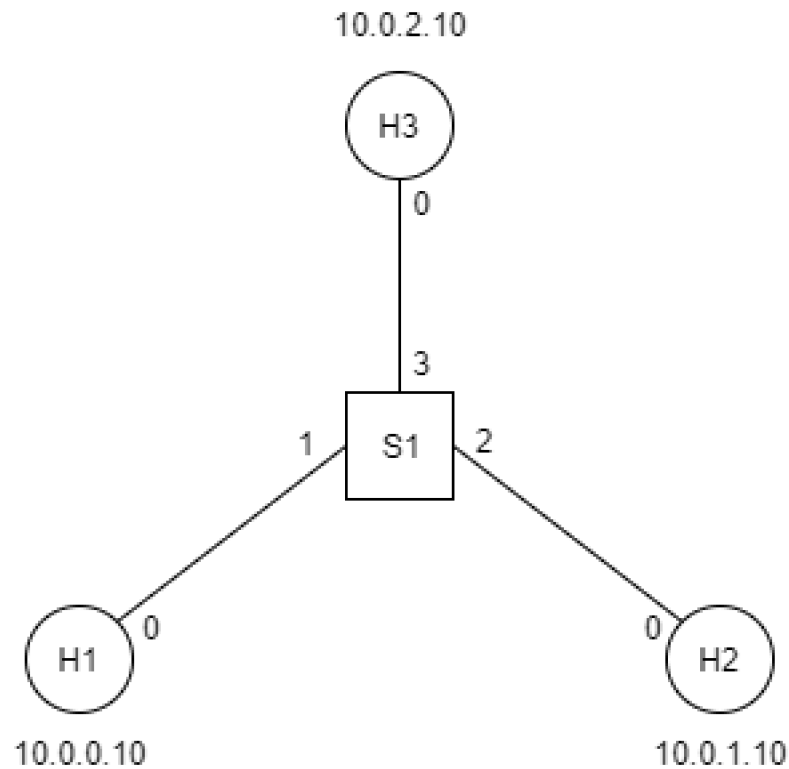


Figura 5 – Topologia utilizada para o projeto: Contador de pacotes por drop programado.

Listagem 3.8 – Código fonte P4 do projeto: Contador de pacotes por drop programado.

```

1  #include "includes/headers.p4"
2  #include "includes/parser.p4"
3
4  counter my_static_counter {
5      type: packets;
6      static: forward;
7      instance_count: 1;
8  }
9
10 action _drop() {
11     count(my_static_counter, 0);
12     drop();
13 }
14
15 action set_destination(dmac, port) {
16     modify_field(ethernet.dstAddr, dmac);
17     modify_field(standard_metadata.egress_spec, port);
18 }
19
20 table forward {
21     reads {
22         ipv4.dstAddr : exact;
23     }
24     actions {
25         set_destination;
26         _drop;
27     }
28 }
29
30 control ingress {

```

```

31     apply(forward);
32 }

```

Os comandos utilizados darão uma visão mais clara de como essa diferenciação foi feita:

```

table_add forward _drop 10.0.2.10 =>
table_add forward set_destination 10.0.0.10 => 00:04:00:00:00:00 1
table_add forward set_destination 10.0.1.10 => 00:04:00:00:00:01 2

```

Nesta rede, todos os pacotes destinados ao *host* h2, identificado pelo *IP* 10.0.2.10, serão recusados. A partir do momento que são recusados, uma unidade é somada ao contador de nome `my_static_counter` na posição 0. Pacotes destinados aos *hosts* h1 e h2 serão repassados sem maiores problemas.

3.3.4 Multicaminhos utilizando abordagem Probabilística

Projeto desenvolvido com o intuito de validar a hipótese de que é possível utilizar construções nativas da linguagem P4 para decidir, baseando-se em métricas da rede, qual rota deve ser tomada a partir de um comutador qualquer. Ou seja, validar a possibilidade de substituição da implementação de multicaminhos que utiliza OpenFlow e Floodlight[7] por uma que utiliza apenas comutadores P4.

Neste projeto, foi construída uma arquitetura capaz de informar a um determinado comutador P4 quais probabilidades ele deve considerar quando precisar decidir entre mais de um caminho. Para isso, foi necessário estender o projeto *behavioral-model* de forma a criar métodos que viriam a ser utilizados para a tomada de decisão entre caminhos. Conforme dito anteriormente, para este projeto a decisão foi aleatória com pesos configuráveis para cada caminho.

Lógicamente, foi necessário criar um mecanismo para a inserção de metadados que seriam utilizados pelo *behavioral-model*. Metadados que, deveriam informar quantos caminhos são possíveis, e também qual o peso para cada um deles. A estrutura escolhida foi um sequencia de bytes onde o primeiro byte representa a quantidade de caminhos possíveis (numero de portas físicas consideradas no switch) e os bytes subsequentes contem a probabilidade de escolha de cada um dos caminhos. Por exemplo, a definição

```
0b00000011000111100001111000101000
```

indica que há três caminhos possíveis (0b00000011) e os 3 bytes subsequentes representam indicam as probabilidades 30% (00011110), 30% (00011110) e 40% (00101000) para cada um dos mesmos. Não é necessário que o somatório dos valores seja igual a 100, apenas

que tenham a proporção desejada. Isto é, se em vez de 30, 30, e 40 os valores fossem 6, 6 e 8, o resultado seria exatamente o mesmo.

Após a geração do metadado, cuja implementação será discutida mais a frente no trabalho, o código *behavioral-model* exibido na Listagem 3.9 e que está no comutador executará o método `probabilistic_simple_multipath`.

Listagem 3.9 – Código fonte C++ responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Probabilística.

```

1  uint32_t get_probabilistic_simple_sum(const char *buf, size_t max_paths) {
2      uint32_t probability_sum = 0;
3      for (size_t i = 1; i < max_paths + 1; i++) {
4          probability_sum += 0 | buf[i];
5      }
6      return probability_sum;
7  }
8
9  uint32_t get_probabilistic_simple_path(const char *buf, uint32_t max_paths, uint32_t
    random) {
10     uint32_t path = 0;
11     uint32_t accum = 0.0;
12
13     for (size_t i = 1; i < max_paths + 1; i++) {
14         accum += 0 | buf[i];
15         if (random < accum) {
16             path = static_cast<uint32_t>(i - 1);
17             break;
18         }
19     }
20     return path;
21 }
22
23 uint32_t get_probabilistic_simple_max_paths(const char *buf, size_t s) {
24     uint32_t array_max = static_cast<uint32_t>(s - 1);
25     uint32_t meta_max = static_cast<uint32_t>(0 | buf[0]);
26     return array_max < meta_max ? array_max : meta_max;
27 }
28
29 struct probabilistic_simple_multipath {
30     uint32_t operator()(const char *buf, size_t s) const {
31         uint32_t max_paths = get_probabilistic_simple_max_paths(buf, s);
32         uint32_t probability_sum = get_probabilistic_simple_sum(buf, max_paths);
33         uint32_t random = rand() % probability_sum;
34         return get_probabilistic_simple_path(buf, max_paths, random);
35     }
36 };

```

Primeiro, calcula-se a quantidade possível de caminhos extraindo o primeiro byte do metadado de nome `buf`. Após isso, calcula-se a soma de todas as probabilidades e é gerado um número aleatório com valor contido no intervalo de 0 a `probability_sum - 1`. O cálculo do caminho é então feito utilizando a abordagem de nome *roulette wheel* a partir da soma acumulada de pesos, como pode ser visto no método que possui o nome `get_probabilistic_simple_path`. Vale observar que os valores retornados para o comutador estão no intervalo de 0 a `max_paths - 1`. Estes valores representam os caminhos, e a

forma com que o programa P4 os identifica será explicada adiante.

Para possibilitar a realização de testes que comprovam a viabilidade da rede proposta, foi desenvolvida uma topologia com quatro *hosts* (h1, h2, h3, h4) e cinco comutadores P4 (s1, s2, s3, s4, s5).

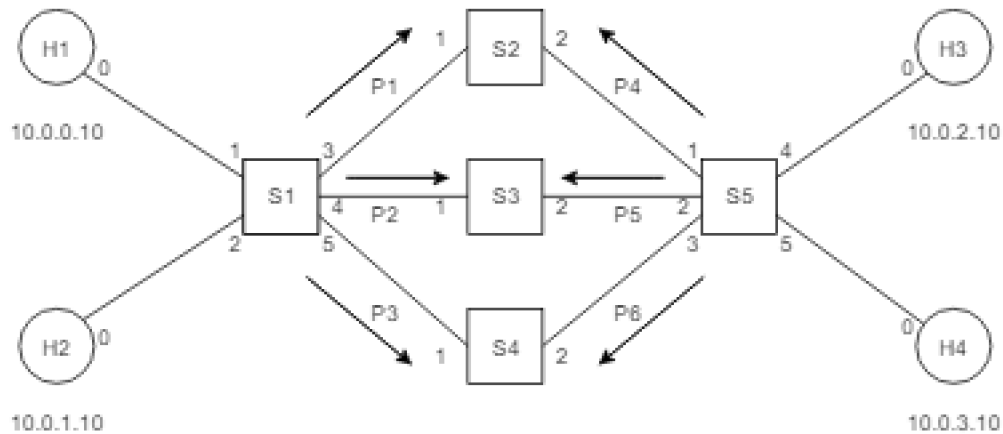


Figura 6 – Topologia utilizada para o projeto: multicaminhos utilizando abordagem probabilística.

Conforme pode ser observado na Figura 6, existem pesos associados a cada caminho possível para pacotes que saem de S1 e S5. Por exemplo, pacotes que estão em S1 e devem chegar em S5 possuem uma probabilidade P1 de passar pelo caminho constituído pelo comutador P4 identificado como S2. A Tabela 1 demonstra quais são as probabilidades que cada caminho da rede possui.

Tabela 1 – Tabela de probabilidades de roteamento

Origem	Destino	Probabilidade
S1	S2	P1
S1	S3	P2
S1	S4	P3
S5	S2	P4
S5	S3	P5
S5	S4	P6

Para pacotes que não precisam atravessar a parte interna da rede, como por exemplo os pacotes enviados do *host* h1 para o *host* h2, a tabela de probabilidades não é considerada, pois o comutador P4 não precisa decidir entre mais de uma possível rota para a entrega da mensagem.

Conforme mencionado anteriormente, é necessário gerar um metadado que contém as informações referentes às probabilidades. Para isso, foi criado um programa P4 que recebe tal informação e a guarda internamente para alimentar o calculador de caminhos no projeto *behavioral-model*.

Listagem 3.10 – Código fonte P4 responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Probabilística.

```

1  #include "includes/headers.p4"
2  #include "includes/parser.p4"
3
4  field_list multi_path_hash_fields {
5      meta.meta_handle;
6  }
7
8  field_list_calculation multi_path_port_selector {
9      input {
10         multi_path_hash_fields;
11     }
12     algorithm : probabilistic_simple_multipath;
13     output_width: 32;
14 }
15
16 action_selector multi_path_action_selector {
17     selection_key : multi_path_port_selector;
18 }
19
20 action _nop() {
21 }
22 }
23
24 action set_multi_path_destination(port) {
25     modify_field(standard_metadata.egress_spec, port);
26 }
27
28 action_profile probability_multipath_profile {
29     actions {
30         _nop;
31         set_multi_path_destination;
32     }
33     size : 64;
34     dynamic_action_selection : multi_path_action_selector;
35 }
36
37 table multi_path_profile_forward {
38     reads {
39         ipv4.srcAddr : exact;
40         ipv4.dstAddr : exact;
41     }
42     action_profile: probability_multipath_profile;
43     size : 1024;
44 }
45
46 table multi_path_regular_forward {
47     reads {
48         ipv4.dstAddr : exact;
49     }
50     actions {
51         set_multi_path_destination;
52     }
53 }
54
55 action set_meta_handle(port) {
56     modify_field(meta.meta_handle, port);
57 }

```

```

58
59 table multi_path_compute_meta {
60     reads {
61         ipv4.srcAddr : exact;
62         ipv4.dstAddr : exact;
63     }
64     actions {
65         set_meta_handle;
66     }
67 }
68
69 action set_dmac(dmac) {
70     modify_field(ethernet.dstAddr, dmac);
71 }
72
73 table multi_path_set_dmac {
74     reads {
75         ipv4.dstAddr : exact;
76     }
77     actions {
78         set_dmac;
79     }
80 }
81
82 control ingress {
83     apply(multi_path_compute_meta);
84     apply(multi_path_set_dmac);
85     apply(multi_path_regular_forward);
86     apply(multi_path_profile_forward);
87 }

```

Como pode ser observado, o nível de complexidade deste código P4 aumentou consideravelmente caso comparado com os três projetos anteriores.

Da mesma forma que havia sido feita previamente, existe uma tabela responsável pela definição da porta de saída que o pacote utilizará para alcançar seu destino. Neste caso, como a intenção é dinamizar a escolha, foi necessário incluir uma construção da linguagem que não havia sido utilizada antes. O nome dessa construção é *action_profile*.

Uma *action_profile* é constituída por um conjunto de ações possíveis e por um algoritmo responsável por escolher qual dentre elas será executada. Isso pode ser visto no trecho exibido na Listagem 3.11.

Listagem 3.11 – Trecho de código P4 responsável pela declaração da *action_profile* no projeto: multicaminhos utilizando abordagem Probabilística.

```

1 action_profile probability_multipath_profile {
2     actions {
3         _nop;
4         set_multi_path_destination;
5     }
6     size : 64;
7     dynamic_action_selection : multi_path_action_selector;
8 }

```

As possíveis ações são *_nop* e *set_multi_path_destination*, onde a primeira representa uma ação vazia, onde nada é aplicado ao pacote, e a segunda realiza a escolha da porta física destino.

O algoritmo de escolha das ações é definido na linha 7 por meio da construção *dynamic_action_selection*. Para este projeto foi utilizado o seletor *multi_path_action_selector*, e o mesmo foi definido de forma idêntica a exibida na Listagem 3.12.

Listagem 3.12 – Trecho de código P4 responsável pela decisão de qual ação será tomada em tempo de execução.

```

1 field_list multi_path_hash_fields {
2     meta.meta_handle;
3 }
4
5 field_list_calculation multi_path_port_selector {
6     input {
7         multi_path_hash_fields;
8     }
9     algorithm : probabilistic_simple_multipath;
10    output_width: 32;
11 }
12
13 action_selector multi_path_action_selector {
14     selection_key : multi_path_port_selector;
15 }
```

O algoritmo *probabilistic_simple_multipath* foi selecionado como o algoritmo de seleção de ações. A implementação dele pode ser vista anteriormente neste mesmo capítulo. Como parâmetros, o algoritmo recebe um array de caracteres e o tamanho do mesmo. Este array contém as informações necessárias para o cálculo da rota a partir das probabilidades definidas. No contexto P4, definimos este array de caracteres como sendo um *field_list*, onde, de cima para baixo, os bytes do array de caracteres são preenchidos. Para esta topologia, foi utilizado somente um metadado de nome *meta.meta_handle* de tamanho 32 bytes (1 byte para a quantidade de caminhos e 3 bytes para a probabilidade de cada caminho).

Para o preenchimento deste metadado em tempo de execução, utilizamos outra tabela de nome *multi_path_compute_meta*. Esta tabela é indexada por ambos os *IPs* de origem e destino. Ou seja, é possível diferenciar probabilidades baseando-se em caminhos diferentes de um pacote originado de um mesmo *IP*. Um possível trabalho futuro é a implementação de uma diferenciação de fluxos por porta de origem e destino, de forma a viabilizar a priorização do tráfego de certas aplicações, por exemplo.

Listagem 3.13 – Trecho de código P4 responsável pelo armazenamento do metadado utilizado como input do algoritmo de seleção de ações.

```

1
2 action set_meta_handle(metadata) {
3     modify_field(meta.meta_handle, metadat);
```

```

4  }
5
6  table multi_path_compute_meta {
7      reads {
8          ipv4.srcAddr : exact;
9          ipv4.dstAddr : exact;
10     }
11     actions {
12         set_meta_handle;
13     }
14 }

```

A tabela *multi_path_compute_meta* possui somente uma ação possível, de nome *set_meta_handle*. Essa ação possui a única função de guardar o valor informado no metadado que será utilizado como *input* do algoritmo.

Com a estrutura pronta, o último passo necessário para a utilização da rede é a inserção dos comandos. Por sua natureza dinâmica, o conjunto de comandos escolhidos para a rede e os resultados obtidos serão discutidos na seção de validação da proposta.

3.3.5 Multicaminhos utilizando abordagem Round-Robin

Projeto desenvolvido com o intuito de validar a hipótese de que é possível utilizar construções nativas da linguagem P4 para decidir, baseando-se em métricas da rede, qual rota deve ser tomada a partir de um comutador qualquer. Ou seja, validar a possibilidade de substituição da implementação de multicaminhos que utiliza OpenFlow e Floodlight[7] por uma que utiliza apenas comutadores P4.

Neste projeto, diferente do mencionado anteriormente, não foi utilizada uma abordagem probabilística, mas sim uma determinística. Nessa rede, cria-se uma iteração infinita onde o primeiro pacote é enviado para o primeiro comutador P4, depois para o segundo, e assim em diante. Repetindo o ciclo após o término de cada sequência.

O nome dado para essa abordagem foi multicaminhos Round-Robin devido à semelhança com o algoritmo de agendamento de processos de mesmo nome. O código P4 se assemelha bastante com o anterior, salvo uma alteração no *input* utilizado para o algoritmo presente no projeto *behavioral_model*. Desta vez, o *IP* de origem e de destino estão sendo enviados para serem utilizados como identificador do fluxo. Novamente, um trabalho futuro interessante seria a adequação desse identificador para comportar diferenciações baseadas em porta de origem e porta de destino. O código utilizado no *behavioral_model* pode ser analisado na Listagem `reflst:c++probabilisticroundrobin`.

Listagem 3.14 – Código fonte C++ responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Round-Robin.

```

1
2  std::map<std::string, std::pair<uint32_t, uint32_t>> round_map;
3

```

```

4 uint32_t get_deterministic_round_path(std::string route_id, uint32_t paths_number) {
5     uint32_t default_route = 0;
6     round_map[route_id] = !round_map.count(route_id) ?
7         std::make_pair(default_route, paths_number) :
8         std::make_pair((round_map[route_id].first + 1) % round_map[route_id].second,
9             paths_number);
10
11     return round_map[route_id].first;
12 }
13 struct deterministic_round_multipath {
14     uint32_t operator()(const char *buf, size_t s) const {
15         std::string route_id(buf, 8);
16         uint32_t paths_number = 0 | buf[s - 1];
17         return get_deterministic_round_path(route_id, paths_number);
18     }
19 };

```

Antes de tudo, foi criada uma estrutura do tipo *std::map* que funciona como um dicionário. A chave utilizada para indexar esta estrutura é o par *IP* origem e *IP* destino. O valor associado ao identificador no dicionário é um par de inteiros não negativos que representam a ação escolhida e o número máximo de ações possíveis, respectivamente.

Para cada pacote cuja rota está ligada à *action_profile*, o dicionário é atualizado para refletir a rota selecionada. O cálculo utilizado é o módulo simples aplicado entre a rota anterior + 1 e o número máximo possível de rotas. Desta forma, os caminhos são atualizados de uma forma similar à de um relógio. Ou seja, de forma circular e sem ultrapassar o máximo estipulado pelo metadado.

A topologia utilizada é a mesma do experimento anterior, com a única diferença sendo a ausência de probabilidades para rotas.

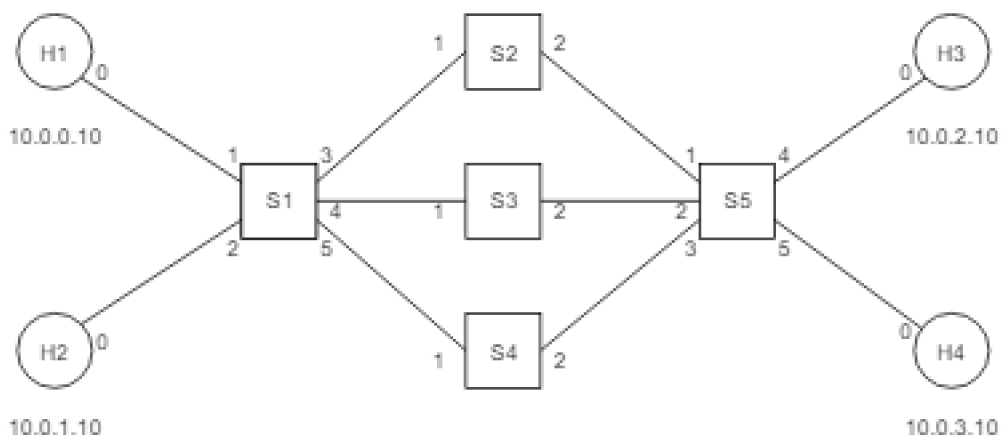


Figura 7 – Topologia utilizada para o projeto: multicaminhos utilizando abordagem Round-Robin.

Da mesma forma que a topologia foi pouco alterada, o código P4 também se manteve parecido com o desenvolvido para o experimento anterior, conforme pode ser observado na Listagem 3.15.

Listagem 3.15 – Código fonte P4 responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Round-Robin.

```

1  #include "includes/headers.p4"
2  #include "includes/parser.p4"
3
4  field_list multi_path_hash_fields {
5      ipv4.srcAddr;
6      ipv4.dstAddr;
7      meta.meta_handle;
8  }
9
10 field_list_calculation multi_path_port_selector {
11     input {
12         multi_path_hash_fields;
13     }
14     algorithm : deterministic_round_multipath;
15     output_width: 32;
16 }
17
18 action_selector multi_path_action_selector {
19     selection_key : multi_path_port_selector;
20 }
21
22 action _nop() {
23 }
24 }
25
26 action set_multi_path_destination(port) {
27     modify_field(standard_metadata.egress_spec, port);
28 }
29
30 action_profile probability_multipath_profile {
31     actions {
32         _nop;
33         set_multi_path_destination;
34     }
35     size : 64;
36     dynamic_action_selection : multi_path_action_selector;
37 }
38
39 table multi_path_profile_forward {
40     reads {
41         ipv4.dstAddr : exact;
42     }
43     action_profile: probability_multipath_profile;
44     size : 1024;
45 }
46
47 table multi_path_regular_forward {
48     reads {
49         ipv4.dstAddr : exact;
50     }
51     actions {
52         set_multi_path_destination;
53     }
54 }
55
56 action set_dmac(dmac) {
57     modify_field(ethernet.dstAddr, dmac);

```

```

58 }
59
60 table multi_path_set_dmac {
61     reads {
62         ipv4.dstAddr : exact;
63     }
64     actions {
65         set_dmac;
66     }
67 }
68
69 action set_meta_handle(portNumber) {
70     modify_field(meta.meta_handle, portNumber);
71 }
72
73 table multi_path_compute_meta {
74     reads {
75         ipv4.srcAddr : exact;
76         ipv4.dstAddr : exact;
77     }
78     actions {
79         set_meta_handle;
80     }
81 }
82
83 control ingress {
84     apply(multi_path_compute_meta);
85     apply(multi_path_set_dmac);
86     apply(multi_path_regular_forward);
87     apply(multi_path_profile_forward);
88 }

```

A única diferença está no metadado enviado para o método de seleção de ações no projeto *behavioral_model*. Neste projeto o *IP* de origem e *IP* de destino devem ser enviados, caso contrário não seria possível indexar o dicionário com essas informações. Esta implementação pode ser observada na Listagem 3.16.

Listagem 3.16 – Trecho de código P4 responsável pelo armazenamento do metadado utilizado como input do algoritmo de seleção de ações.

```

1 field_list multi_path_hash_fields {
2     ipv4.srcAddr;
3     ipv4.dstAddr;
4     meta.meta_handle;
5 }

```

Com a definição da rede pronta, o conjunto de comandos escolhidos para a rede e os resultados obtidos serão discutidos na seção de validação da proposta.

3.3.6 Multicaminhos utilizando abordagem Mista

Projeto desenvolvido com o intuito de validar a hipótese de que é possível utilizar construções nativas da linguagem P4 para decidir, baseando-se em métricas da rede, qual rota deve ser tomada a partir de um comutador qualquer. Ou seja, validar a possibilidade

de substituição da implementação de multicaminhos que utiliza OpenFlow e Floodlight[7] por uma que utiliza apenas comutadores P4.

Este projeto foi desenvolvido com a intenção de mesclar ambas abordagens mencionadas anteriormente. Ou seja, foi implementada uma abordagem determinística semelhante à Round-Robin com a diferença de que agora é possível configurar quantos pacotes devem ser enviados por um caminho até que o próximo seja escolhido. Ou seja, a partir de um metadado idêntico ao do experimento "multicaminhos utilizando abordagem Probabilística", o método de seleção no projeto *behavioral_model* define qual o número de vezes que pacotes devem passar por uma determinada rota até que a próxima seja escolhida. Por exemplo, no seguinte metadado está especificado que existem 3 rotas (0b00000011), onde 30 (00011110) pacotes devem passar pelo primeiro, 30 (00011110) pelo segundo e 40 (00101000) pelo terceiro, respectivamente e repetindo após o total de 100 pacotes serem enviados.

0b00000011000111100001111000101000

O código responsável pela lógica de seleção de rotas está presente no projeto *behavioral_model* e pode ser visto na Listagem 3.17.

Listagem 3.17 – Código fonte C++ responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Mista.

```

1  std::map<std::string, std::pair<uint32_t, uint32_t>> block_map;
2
3  uint32_t calculate_deterministic_block_path(const char *buf, std::string route_id,
4      uint32_t max_paths) {
5      uint32_t path = 0;
6      uint32_t accum = 0;
7
8      for (size_t i = 9; i < max_paths + 9; i++) {
9          accum += 0 | buf[i];
10         if (block_map[route_id].first < accum) {
11             path = static_cast<uint32_t>(i - 9);
12             break;
13         }
14     }
15     return path;
16 }
17
18 uint32_t get_deterministic_blocks_num(const char *buf, uint32_t max_paths) {
19     uint32_t accum = 0;
20     for (size_t i = 9; i < max_paths + 9; i++) {
21         accum += 0 | buf[i];
22     }
23     return accum;
24 }
25
26 uint32_t get_deterministic_block_path(const char *buf, std::string route_id, uint32_t
27     max_paths) {
28     uint32_t blocks_num = get_deterministic_blocks_num(buf, max_paths);
29     block_map[route_id] = !block_map.count(route_id) ?

```

```

28     std::make_pair(static_cast<uint32_t>(0), blocks_num) :
29     std::make_pair((block_map[route_id].first + 1) % block_map[route_id].second,
        blocks_num);
30     return calculate_deterministic_block_path(buf, route_id, max_paths);
31 }
32
33 uint32_t get_deterministic_block_max_paths(const char *buf, size_t s) {
34     uint32_t array_max = static_cast<uint32_t>(s - 9);
35     uint32_t meta_max = static_cast<uint32_t>(0 | buf[8]);
36     return array_max < meta_max ? array_max : meta_max;
37 }
38
39 struct deterministic_block_multipath {
40     uint32_t operator()(const char *buf, size_t s) const {
41         std::string route_id(buf, 8);
42         uint32_t max_paths = get_deterministic_block_max_paths(buf, s);
43         return get_deterministic_block_path(buf, route_id, max_paths);
44     }
45 };

```

De forma idêntica à que foi feita no experimento anterior, foi definido um dicionário utilizando a construção C++ *std::map*. A função do dicionário neste projeto é praticamente a mesma, com a diferença de que o número da ação atual é utilizado para selecionar para qual rota o pacote será encaminhado em um algoritmo *wheel of fortune*, que também foi utilizado em outro projeto neste trabalho.

Calcula-se primeiro a chave do dicionário, que está presente nos oito primeiros bits do metadado. Estes bits representam o par *IP* de origem e *IP* de destino. Feito isso, o número máximo de caminhos é extraído do metadado e a rota é definida dentro do método *get_deterministic_block_path*.

A topologia é a mesma do projeto "multicaminhos utilizando abordagem Probabilística", onde existem pesos para cada caminho saindo dos comutadores P4 S1 e S5.

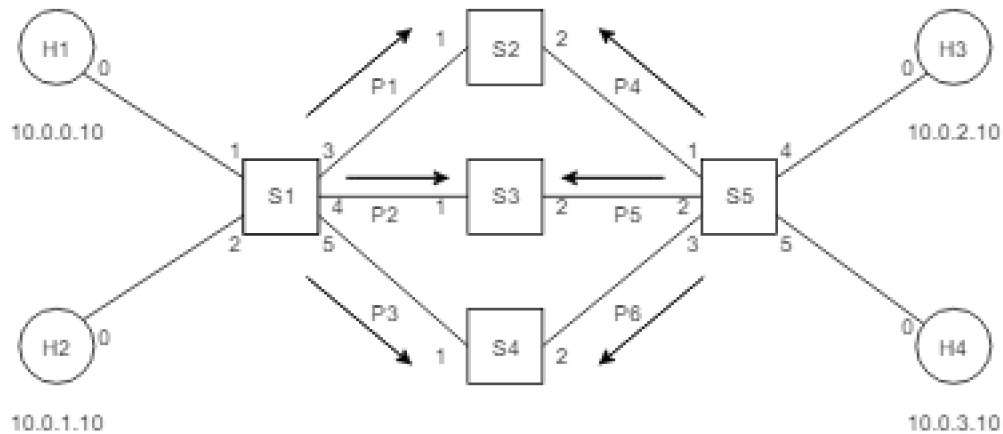


Figura 8 – Topologia utilizada para o projeto: multicaminhos utilizando abordagem Mista.

O código P4 deste projeto é o mesmo do projeto "multicaminhos utilizando abordagem Round-Robin", onde os *IPs* de origem e destino estão presentes no metadado. Isso pode ser observado na Listagem 3.18.

Listagem 3.18 – Código fonte P4 responsável pelo cálculo multicaminhos no projeto: multicaminhos utilizando abordagem Mista.

```

1  #include "includes/headers.p4"
2  #include "includes/parser.p4"
3
4  field_list multi_path_hash_fields {
5      ipv4.srcAddr;
6      ipv4.dstAddr;
7      meta.meta_handle;
8  }
9
10 field_list_calculation multi_path_port_selector {
11     input {
12         multi_path_hash_fields;
13     }
14     algorithm : deterministic_block_multipath;
15     output_width: 32;
16 }
17
18 action_selector multi_path_action_selector {
19     selection_key : multi_path_port_selector;
20 }
21
22 action _nop() {
23
24 }
25
26 action set_multi_path_destination(port) {
27     modify_field(standard_metadata.egress_spec, port);
28 }
29
30 action_profile probability_multipath_profile {
31     actions {
32         _nop;
33         set_multi_path_destination;

```

```

34     }
35     size : 64;
36     dynamic_action_selection : multi_path_action_selector;
37 }
38
39 table multi_path_profile_forward {
40     reads {
41         ipv4.srcAddr : exact;
42         ipv4.dstAddr : exact;
43     }
44     action_profile: probability_multipath_profile;
45     size : 1024;
46 }
47
48 table multi_path_regular_forward {
49     reads {
50         ipv4.dstAddr : exact;
51     }
52     actions {
53         set_multi_path_destination;
54     }
55 }
56
57 action set_meta_handle(port) {
58     modify_field(meta.meta_handle, port);
59 }
60
61 table multi_path_compute_meta {
62     reads {
63         ipv4.srcAddr : exact;
64         ipv4.dstAddr : exact;
65     }
66     actions {
67         set_meta_handle;
68     }
69 }
70
71 action set_dmac(dmac) {
72     modify_field(ethernet.dstAddr, dmac);
73 }
74
75 table multi_path_set_dmac {
76     reads {
77         ipv4.dstAddr : exact;
78     }
79     actions {
80         set_dmac;
81     }
82 }
83
84 control ingress {
85     apply(multi_path_compute_meta);
86     apply(multi_path_set_dmac);
87     apply(multi_path_regular_forward);
88     apply(multi_path_profile_forward);
89 }

```

Com a definição da rede pronta, o conjunto de comandos escolhidos para a rede e os resultados obtidos serão discutidos na seção de validação da proposta.

4 Validação da Proposta

Este capítulo visa validar a proposta deste trabalho a partir de experimentos realizados nas redes criadas durante o desenvolvimento. Ao todo, foram realizados três experimentos. Um para cada rede multicaminho definida.

4.1 Multicaminhos utilizando abordagem Probabilística

Conforme descrito no capítulo anterior, esta rede foi criada com o intuito de selecionar rotas de forma probabilística a partir de um metadado pré-definido. Para isso, foi necessário aplicar uma série de comandos nos comutadores P4.

Para esta rede, os comutadores S1 e S5 devem conter toda a lógica de encaminhamento multicaminhos, enquanto os outros comutadores S2, S3 e S4 só repassam os pacotes que recebem e contabilizam os mesmos com diferenciação de fluxo.

Os comandos utilizados para alimentar tanto os comutadores responsáveis pelo repasse quanto os comutadores responsáveis pelo encaminhamento multicaminhos serão descritos a seguir:

4.1.1 Comandos

Para os comutadores S2, S3 e S4, foram aplicados os comandos listados na Listagem 4.1.

Listagem 4.1 – Lista de comandos aplicados para os comutadores S2, S3 e S4 no projeto: multicaminhos utilizando abordagem Probabilística.

```

1 table_add process_path count_path 10.0.0.10 10.0.2.10 => 0
2 table_add process_path count_path 10.0.2.10 10.0.0.10 => 1
3 table_add process_path count_path 10.0.0.10 10.0.3.10 => 2
4 table_add process_path count_path 10.0.3.10 10.0.0.10 => 3
5 table_add process_path count_path 10.0.1.10 10.0.2.10 => 4
6 table_add process_path count_path 10.0.2.10 10.0.1.10 => 5
7 table_add process_path count_path 10.0.1.10 10.0.3.10 => 6
8 table_add process_path count_path 10.0.3.10 10.0.1.10 => 7
9 table_add forward set_destination 10.0.0.10 => 00:04:00:00:00:00 1
10 table_add forward set_destination 10.0.1.10 => 00:04:00:00:00:01 1
11 table_add forward set_destination 10.0.2.10 => 00:04:00:00:00:02 2
12 table_add forward set_destination 10.0.3.10 => 00:04:00:00:00:03 2

```

Os oito primeiros comandos são responsáveis pela contabilização dos pacotes que passam pelo comutador, sendo cada um deles referente a um fluxo diferente. O restante dos comandos são responsáveis pelo redirecionamento do pacote. Como pode ser obser-

vado, não existe a possibilidade de um pacote destinado a um *host* qualquer passar por caminhos diferentes.

Os comutadores S1 e S5 receberam ambos o mesmo conjunto de comandos, com a única diferença nos valores recebidos como parâmetros em alguns deles, como por exemplo as interfaces de saída para os hosts H3 e H4 em S5. Os comandos recebidos pelo comutador S1 podem ser vistos na Listagem 4.2.

Listagem 4.2 – Lista de comandos aplicados para o comutador S1 no projeto: multicaminhos utilizando abordagem Probabilística.

```

1 table_add multi_path_set_dmac set_dmac 10.0.0.10 => 00:04:00:00:00:00
2 table_add multi_path_set_dmac set_dmac 10.0.1.10 => 00:04:00:00:00:01
3 table_add multi_path_set_dmac set_dmac 10.0.2.10 => 00:04:00:00:00:02
4 table_add multi_path_set_dmac set_dmac 10.0.3.10 => 00:04:00:00:00:03
5 table_add multi_path_regular_forward set_multi_path_destination 10.0.0.10 => 1
6 table_add multi_path_regular_forward set_multi_path_destination 10.0.1.10 => 2
7 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.2.10 =>
  0b000000011000111100001111000101000
8 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.3.10 =>
  0b000000011001100100010100000001010
9 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.2.10 =>
  0b000000011000111100011001000010100
10 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.3.10 =>
  0b000000011001010000010100000010100
11 act_prof_create_group probability_multipath_profile
12 act_prof_create_member probability_multipath_profile _nop
13 act_prof_create_member probability_multipath_profile set_multi_path_destination 3
14 act_prof_create_member probability_multipath_profile set_multi_path_destination 4
15 act_prof_create_member probability_multipath_profile set_multi_path_destination 5
16 act_prof_add_member_to_group probability_multipath_profile 1 0
17 act_prof_add_member_to_group probability_multipath_profile 2 0
18 act_prof_add_member_to_group probability_multipath_profile 3 0
19 table_indirect_set_default multi_path_profile_forward 0
20 table_indirect_add_with_group multi_path_profile_forward 10.0.0.10 10.0.2.10 => 0
21 table_indirect_add_with_group multi_path_profile_forward 10.0.0.10 10.0.3.10 => 0
22 table_indirect_add_with_group multi_path_profile_forward 10.0.1.10 10.0.2.10 => 0
23 table_indirect_add_with_group multi_path_profile_forward 10.0.1.10 10.0.3.10 => 0
24 table_dump multi_path_profile_forward

```

As primeiras quatro linhas são responsáveis por definir qual será o endereço MAC destino para cada pacote recebido. Isso é feito tendo como base o endereço *ip* destino que está contido no cabeçalho do pacote. Nas linhas 5 e 6 os casos onde pacotes são destinados para *hosts* que estão diretamente ligados ao comutador são tratados. Para estes, não há necessidade de multicaminhos.

As linhas 7, 8, 9 e 10 são responsáveis por uma das partes mais importantes do processo. Os metadados. Estes são definidos a gosto pelo administrador da rede, e representam neste caso, a quantidade possível de caminhos e as probabilidades associadas a cada um deles. Nesta rede os metadados são definidos tomando como base o fluxo de pacotes, e possuem a mesma formatação mencionada previamente neste trabalho, onde o primeiro byte indica que há três caminhos possíveis e os 3 bytes subsequentes representam

indicam as probabilidades de cada um dos mesmos.

A partir da linha 11 a construção *action_profile* é definida e populada. As linhas 12, 13, 14 e 16 têm o papel de definir as ações e parâmetros disponíveis para escolha do algoritmo presente no projeto *behavioral_model*. Nesta rede existem quatro ações possíveis. São elas: *_nop* (o mesmo que não fazer nada), e redirecionamentos do pacote para as portas físicas de números 3, 4 e 5.

Cada ação criada é identificada internamente por um *id* único. Esse *id* é gerado de forma incremental partindo do número zero e é utilizado para o próximo passo.

As linhas 16, 17 e 18 adicionam as entradas de *id* 1, 2 e 3 para o grupo de número zero. Este grupo pode ser entendido como o conjunto de ações possíveis que ser levadas em consideração no momento da escolha feita pelo código do *behavioral_model*.

A linha 19 define qual a ação padrão a ser executada quando algum pacote entrante no comutador P4 receber um *match* na tabela associada à *action_profile*. Nesta rede, a ação padrão a ser tomada é a *_nop*, cujo *id* é 0.

Feito isso, precisamos associar os fluxos ao grupo zero da *action_profile*. Isso é feito das linhas 20 a 23. Poderíamos por exemplo ter criado dois grupos de ações e associar fluxos diferentes a grupos diferentes, porém não houve necessidade de criar mais de um grupo para esta rede.

A última linha tem a única função de realizar um *dump* da tabela associada à *action_profile*. Após a inserção dos comandos, foram realizados testes de validação da rede.

Para isso, associamos as probabilidades definidas na Tabela 2 à rede.

4.1.2 Resultados

Cada linha na tabela apresentada anteriormente representa um fluxo de dados e cada probabilidade de caminhos associada ao mesmo. Para cada uma, foram enviados dez mil pacotes para avaliar se a probabilidade estipulada se manteve consistente.

Foi possível validar a quantidade de pacotes por caminho devido ao programa P4 e comandos associados aos roteadores S2, S3 e S4. Uma conexão *thrift* foi aberta a partir do comando:

```
python run_dummy_cli.py --thrift-port <thrift-port>
```

Onde *<thrift-port>* representa a porta *thrift* associada aos comutadores P4 internos. Após a interface para comandos no comutador estar aberta, o valor do contador por fluxo foi lido da seguinte forma:

```
counter_read dummy_switch_counter <flow-id>
```

Tabela 2 – Tabela de probabilidades de roteamento aplicadas a rede do projeto: multicaminhos utilizando abordagem Probabilística

Host Origem	Host Destino	Comutador Origem	Comutador Destino	Probabilidade
H1	H3	S1	S2	30%
H1	H3	S1	S3	30%
H1	H3	S1	S4	40%
H1	H4	S1	S2	50%
H1	H4	S1	S3	40%
H1	H4	S1	S4	10%
H2	H3	S1	S2	30%
H2	H3	S1	S3	50%
H2	H3	S1	S4	20%
H2	H4	S1	S2	40%
H2	H4	S1	S3	40%
H2	H4	S1	S4	20%
H3	H1	S5	S2	30%
H3	H1	S5	S3	30%
H3	H1	S5	S4	40%
H3	H2	S5	S2	50%
H3	H2	S5	S3	40%
H3	H2	S5	S4	10%
H4	H1	S5	S2	30%
H4	H1	S5	S3	50%
H4	H1	S5	S4	20%
H4	H2	S5	S2	40%
H4	H2	S5	S3	40%
H4	H2	S5	S4	20%

Onde $\langle flow-id \rangle$ representa o *id* referente ao fluxo que deseja ser observado. Os *ids* associados a cada fluxo podem ser consultados neste mesmo capítulo na listagem de comandos inseridos nos comutadores S2, S3 e S4.

Após a execução do comando *ping* para o envio de pacotes entre cada par de *hosts*, os resultados obtidos podem ser vistos na Tabela 3.

4.1.3 Conclusão

Foi possível validar após o experimento, que as probabilidades configuradas a partir dos metadados foram levadas em consideração da forma correta no momento da escolha do caminho a ser tomado pelo pacote. Portanto, foi confirmado que, utilizando apenas a linguagem P4 e uma rede de comutadores P4, é possível configurar rotas multicaminhos.

Tabela 3 – Resultados obtidos após a execução dos testes para o projeto: multicaminhos utilizando abordagem Probabilística

Host Origem, Destino	Comutador Origem, Destino	Probabilidade	Pacotes
H1, H3	S1, S2	30%	2983
H1, H3	S1, S3	30%	3017
H1, H3	S1, S4	40%	4000
H1, H4	S1, S2	50%	5006
H1, H4	S1, S3	40%	4015
H1, H4	S1, S4	10%	979
H2, H3	S1, S2	30%	2991
H2, H3	S1, S3	50%	4998
H2, H3	S1, S4	20%	2011
H2, H4	S1, S2	40%	4010
H2, H4	S1, S3	40%	3998
H2, H4	S1, S4	20%	1992
H3, H1	S5, S2	30%	3003
H3, H1	S5, S3	30%	2994
H3, H1	S5, S4	40%	4003
H3, H2	S5, S2	50%	5010
H3, H2	S5, S3	40%	4007
H3, H2	S5, S4	10%	983
H4, H1	S5, S2	30%	2996
H4, H1	S5, S3	50%	5002
H4, H1	S5, S4	20%	2002
H4, H2	S5, S2	40%	3998
H4, H2	S5, S3	40%	4003
H4, H2	S5, S4	20%	1999

4.2 Multicaminhos utilizando abordagem Round-Robin

Para o segundo experimento a mesma topologia foi utilizada, com a exceção de que agora os caminhos são definidos de forma determinística, um a um, obedecendo o sentido horário. Ou seja, para a topologia apresentada de cinco comutadores, caso dois pacotes sejam enviados de H1 para H3, o primeiro pacote passará primeiro por S2 na ida e na volta, e o segundo por S3 na ida e na volta.

4.2.1 Comandos

Os comandos inseridos nos comutadores internos S2, S3 e S4 são os mesmos do experimento anterior, portanto não serão mostrados aqui. O objetivo continua o mesmo, que é contabilizar pacotes com diferenciação de fluxo, de forma a possibilitar a validação do experimento.

Para os comutadores externos S1 e S5 a única mudança se deu nos metadados enviados para o método responsável no projeto *behavioral_model*. As novas entradas com

os novos metadados estão na Listagem 4.3.

Listagem 4.3 – Metadados aplicados ao comutador S1 e S5 do projeto: multicaminhos utilizando abordagem Round-Robin.

```
1 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.2.10 => 0b000000011
2 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.3.10 => 0b000000011
3 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.2.10 => 0b000000011
4 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.3.10 => 0b000000011
```

A única informação necessária a ser passada para o método responsável pela escolha dos caminhos é a quantidade de caminhos possíveis. Internamente, o programa que está executando no comutador guarda o estado atual dos pacotes de determinado fluxo. Isso possibilita a natureza determinística Round-Robin da escolha dos multicaminhos.

Para a validação desta proposta de rede, foram disparadas pequenas quantidades de pacotes de forma a validar que os caminhos eram sempre alterados de forma sequencial e cíclica.

4.2.2 Resultados

A Tabela 4 mapeia o fluxo, quantidade de pacotes enviados e quantidade de pacotes recebidos por comutador interno (S2, S3 e S4).

4.2.3 Conclusão

Foi validado que a lógica implementada no projeto *behavioral_model* obteve resultados práticos. Também foi possível guardar estados, algo que não havia sido feito no experimento anterior e que abre portas para diversas implementações interessantes. Portanto, foi confirmado mais uma vez que, utilizando apenas a linguagem P4 e uma rede de comutadores P4, é possível configurar rotas multicaminhos utilizando algoritmos personalizados.

4.3 Multicaminhos utilizando abordagem Mista

O terceiro e último experimento foi concebido a partir de uma junção dos dois últimos experimentos. Para este, a abordagem determinística e sequencial foi mantida, com a diferença de que agora foi possível configurar quantos pacotes devem passar por cada caminho até que o próximo seja escolhido. O mecanismo de estados utilizado na abordagem Round-Robin e os metadados da abordagem Probabilística Simples foram mantidos.

Tabela 4 – Resultados obtidos após a execução dos testes para o projeto: multicaminhos utilizando abordagem Round-Robin

Host Origem	Host Destino	Pacotes Enviados	S2, S3 e S4
H1	H3	1	1, 0 e 0
H1	H3	2	1, 1 e 0
H1	H3	6	2, 2 e 2
H1	H4	7	3, 2 e 2
H1	H4	8	3, 3 e 2
H1	H4	10	4, 3 e 3
H2	H3	1	1, 0 e 0
H2	H3	2	1, 1 e 0
H2	H3	3	1, 1 e 1
H2	H4	10	4, 3 e 3
H2	H4	15	5, 5 e 5
H2	H4	20	7, 7 e 6
H3	H1	31	11, 10 e 10
H3	H1	35	12, 12 e 11
H3	H1	36	12, 12 e 12
H3	H2	3000	1000, 1000 e 1000
H3	H2	3001	1001, 1000 e 1000
H3	H2	3002	1001, 1001, 1000
H4	H1	0	0, 0 e 0
H4	H1	7	3, 2 e 2
H4	H1	1	1, 0 e 0
H4	H2	10000	3334, 3333 e 3333
H4	H2	10003	3335, 3334 e 3334
H4	H2	10006	3336, 3335 e 3335

4.3.1 Comandos

Os comandos aplicados aos comutadores internos (S2, S3 e S4) foram os mesmos dos outros experimentos e os comandos aplicados aos comutadores externos (S1 e S5) foram os mesmos da abordagem Probabilística Simples.

Para lembrar, ambos comutadores P4 externos foram alimentados com os mesmos metadados, com a única diferença no fluxo mapeado para a inserção. Os comandos inseridos para o comutador externo S1 estão exibidos na Listagem 4.4.

Listagem 4.4 – Lista de comandos aplicados para os comutadores S1 e S5 no projeto: multicaminhos utilizando abordagem Mista.

```

1 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.2.10 =>
  0b000000011000111100001111000101000
2 table_add multi_path_compute_meta set_meta_handle 10.0.0.10 10.0.3.10 =>
  0b000000011001100100010100000001010
3 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.2.10 =>
  0b000000011000111100011001000010100
4 table_add multi_path_compute_meta set_meta_handle 10.0.1.10 10.0.3.10 =>
  0b000000011001010000010100000010100

```

Cada metadado contém o número de caminhos possíveis e a quantidade de pacotes associada a cada caminho possível. Por exemplo, para o fluxo de H1 para H3 existem três caminhos possíveis (0b00000011), onde o primeiro deve receber 30 (00011110) pacotes até que o segundo receba o primeiro, que por sua vez deve receber 30 (00011110) pacotes até que o último receba o primeiro dos 40 (00101000) possíveis.

Para validar este experimento, foram consideradas as probabilidades listadas na Tabela 2. Da mesma forma que no experimento anterior, também foram disparadas diversas quantidades de pacotes, de forma a validar que a natureza sequencial e cíclica da escolha de caminhos ocorre da maneira correta.

4.3.2 Resultados

A Tabela 5 mapeia o fluxo, quantidade de pacotes enviados e quantidade de pacotes recebidos por comutador P4 interno (S2, S3 e S4).

Tabela 5 – Resultados obtidos após a execução dos testes para o projeto: multicaminhos utilizando abordagem Mista

Host Origem	Host Destino	Pacotes Enviados	Percentual S2, S3 e S4	S2, S3 e S4
H1	H3	100	30%, 30%, 40%	30, 30 e 40
H1	H3	101	30%, 30%, 40%	31, 30 e 40
H1	H3	102	30%, 30%, 40%	32, 30 e 40
H1	H4	100	50%, 40%, 10%	50, 40 e 10
H1	H4	200	50%, 40%, 10%	100, 80 e 20
H1	H4	201	50%, 40%, 10%	101, 80 e 20
H2	H3	100	30%, 50%, 20%	30, 50 e 20
H2	H3	10	30%, 50%, 20%	10, 0 e 0
H2	H3	31	30%, 50%, 20%	30, 1 e 0
H2	H4	100	40%, 40%, 20%	40, 40 e 20
H2	H4	120	40%, 40%, 20%	60, 40 e 20
H2	H4	205	40%, 40%, 20%	85, 80 e 40
H3	H1	100	30%, 30%, 40%	30, 30 e 40
H3	H1	1	30%, 30%, 40%	1, 0 e 0
H3	H1	2	30%, 30%, 40%	2, 0 e 0
H3	H2	100	50%, 40%, 10%	50, 40 e 10
H3	H2	300	50%, 40%, 10%	150, 120 e 30
H3	H2	91	50%, 40%, 10%	50, 40 e 1
H4	H1	100	30%, 50%, 20%	30, 50 e 20
H4	H1	29	30%, 50%, 20%	29, 0 e 0
H4	H1	35	30%, 50%, 20%	30, 5 e 0
H4	H2	100	40%, 40%, 20%	40, 40 e 20
H4	H2	101	40%, 40%, 20%	41, 40 e 20
H4	H2	102	40%, 40%, 20%	42, 40 e 20

4.3.3 Conclusão

Foi novamente validado que a regra personalizada construída no projeto *behavioral_model* apresentou os resultados esperados. Foi possível também demonstrar que a junção de metadados e métodos de seleção é possível e não possui limitações criativas. Portanto, mais uma vez ficou claro que é possível configurar rotas multicaminhos utilizando apenas a linguagem P4 e uma rede de comutadores P4.

5 Trabalhos Futuros

Este trabalho teve como objetivo validar a possibilidade de transição da arquitetura de roteamento multicaminhos desenvolvido por [7], que utiliza OpenFlow [11], por uma que utiliza somente a linguagem P4 e seu ecossistema. Como resultado foi possível validar tal hipótese, mas alguns pontos podem ser melhorados tanto na implementação dos projetos, quanto no prosseguimento da pesquisa. São eles:

- Utilização de apenas um programa P4 tanto para comutadores externos e comutadores internos. Nos projetos desenvolvidos por este trabalho foram construídos dois programas separados, um para decisão de rotas multicaminhos e um para a contagem de pacotes com diferenciação por fluxo.
- Desenvolvimento de um projeto de rede com rotas com larguras de banda limitadas, de forma a realizar testes de estresse em conjunto com os algoritmos de decisão de rotas multicaminhos.
- Melhoramento do ambiente de desenvolvimento *p4-dev* criado, de forma a atualiza-lo com as últimas especificações da linguagem.
- Portar inteiramente os algoritmos multicaminhos propostos pelo projeto desenvolvido por [7] para a arquitetura P4 proposta por este trabalho.

6 Conclusão

Conforme mencionado durante o trabalho, o protocolo OpenFlow [11] está cada vez mais complexo devido a sua natureza generalista. Portanto, viu-se necessária a criação de uma alternativa mais dinâmica e expressiva. Uma das alternativas criadas foi a linguagem P4 em conjunto com seu ecossistema.

Para este trabalho, foi proposta uma alternativa que utiliza P4 à arquitetura *SDN* de rotas multicaminhos proposta por [7], que por sua vez utiliza OpenFlow. Com os resultados dos experimentos foi possível validar a possibilidade de transição da arquitetura proposta previamente, e também foi possível contribuir com a comunidade OSS por meio do ambiente de desenvolvimento *p4-dev*.

A área de Redes Definidas por Software está crescendo em um ritmo muito acelerado e a própria comunidade de desenvolvimento da linguagem P4 prova isso. Por mais que ainda esteja em um ponto de alta volatilidade, onde não existem muitos padrões definidos pela indústria, o ecossistema P4 se destaca pelo seu número de contribuidores.

Portanto, pode-se esperar um grande amadurecimento na área de Redes Definidas por Software, e também na linguagem P4, que devido ao seu grande potencial, pode-se dizer que possui um futuro intimamente ligado com a área de redes de computadores como um todo.

Referências

- [1] P. A. R. Antunes. Soluções de segurança em redes utilizando a linguagem p4, 2016. Citado na página 11.
- [2] D. M. Batista, G. Blair, F. Kon, R. Boutaba, D. Hutchison, R. Jain, R. Ramjee, and C. E. Rothenberg. Perspectives on software-defined networks: interviews with five leading scientists from the networking community. *Journal of Internet Services and Applications*, 6(1):22, Oct 2015. Citado na página 9.
- [3] T. P. L. Consortium. The p4 language specification. Citado 2 vezes nas páginas 5 e 10.
- [4] A. C. Drummond, N. L. S. da Fonseca, X. Chen, and A. Jukan. Roteamento multicaminho para provisão eficiente de recursos interdomínio com qualidade de serviço em redes ópticas wdm, 2011. Citado na página 11.
- [5] O. N. FOUNDATION. Software-defined networking: The new norm for networks. Citado na página 5.
- [6] T. P. A. W. Group. Portable switch architecture (psa). Citado na página 6.
- [7] P. Henrique Amorim Rezende. Extensões na arquitetura sdn para o provisionamento de qos através do monitoramento e uso de múltiplos caminhos. Master's thesis, Universidade Federal de Uberlândia. Citado 9 vezes nas páginas 5, 6, 7, 8, 27, 33, 37, 50 e 51.
- [8] C. Hopps. Analysis of an equal-cost multi-path algorithm. Citado na página 12.
- [9] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhligand. Software-defined networking: A comprehensive survey. Citado 2 vezes nas páginas 9 e 10.
- [10] W. Li, W. Meng, and L. F. Kwoka. A survey on openflow-based software defined networks: Security challenges and countermeasures. Citado na página 10.
- [11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turnerand. Openflow: Enabling innovation in campus networks. Citado 3 vezes nas páginas 10, 50 e 51.
- [12] G. Neri, R. Morling, G. Cain, E. Faldella, M. Longhi-Gelati, T. Salmon-Cinotti, and P. Natali. Mininet: A local area network for real-time instrumentation applications. Citado na página 11.

-
- [13] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, and M. F. Magalhães. Open-flow e redes definidas por software: um novo paradigma de controle e inovação em redes de pacotes. Citado 2 vezes nas páginas [5](#) e [10](#).