Capítulo 1

Uma Nova Revolução em Redes: Programação do Plano de Dados com P4

Daniel Lazkani Feferman (UNICAMP), Juan Sebastian Mejia (UNICAMP), Nathan Franklin Saraiva de Sousa (UNICAMP) e Christian Esteve Rothenberg (UNICAMP)

Abstract

Traditional IP networks are complex and hard to configure following pre-defined policies. Software Defined Networking (SDN) emerges as a promise to change the way data is routed, introducing programmability and flexibility in such networks. Initially, several new protocols were proposed, OpenFlow (OF) being the most famous one. Over the years, new versions have been released with support for new headers. Thus, OF became feature-rich but complex since it was necessary to maintain backward compatibility. Also, processing packets were inflexibly performed in ASICs (Application Specific Integrated Circuits) network devices, which made it impossible to change and insert new functionality. Therefore, aiming at wide data plane programmability, the P4 language was proposed: ***P****rogramming* ***P****rotocol-independent* ***P****acket* ***P****rocessors. The P4 language abstracts the data plane and allows to program how network equipment datapath pipelines should behave. The main purpose of this mini-course is to present an overview of SDN in its current context; principles of programming in the device data plan of networks through the P4 language in both versions, 14 and 16; describe and implement practical activities with P4 and a software switch; and finally, to discuss the main challenges and advances in the programming area of the data plan. This course will allows attendees to make the initial steps into the latest advances around "softwarisation" of the data plane.*

Resumo

*As tradicionais redes IP são complexas e difíceis de configurar de acordo com políticas pré-definidas. Software Defined Networking (SDN) surge então como uma tecnologia que promete mudar a forma como os dados são encaminhados, introduzindo a programabilidade e flexibilidade em tais redes. Inicialmente diversos novos protocolos foram propostos, o mais famoso deles é o OpenFlow (OF). Ao longo dos anos, novas versões foram liberadas com suporte a novos cabeçalhos. Isso o tornou mais robusto porém mais complexo, já que era necessário manter a retrocompatibilidade. Além disso, o processamento de pacotes eram realizado de forma inflexível nos ASICs (Application Specific Integrated Circuits) dos dispositivos de rede, o que tornava a alteração e inserção de novas funcionalidade um processo engessado. Sendo assim, houve então a necessidade de introduzir a programabilidade no plano de dados. Nesse contexto surgiu a linguagem P4: **P**rogramming **P**rotocol-independent **P**acket **P**rocessors. A linguagem P4 abstrai o plano de dados e permite programar como os equipamentos de rede devem se comportar. Este minicurso tem como principais objetivos: apresentar uma visão geral sobre SDN em seu contexto atual; abordar os princípios da programação no plano de dados dos dispositivos de redes através da linguagem P4 em ambas versões, 14 e 16; descrever e implementar atividades práticas com P4 e software switch bmv2; e por fim, discutir os principais desafios e avanços na área de programação do plano de dados. Este minicurso proporcionará aos participantes os passos iniciais para imersão no ambiente de "softwarização" do plano de dados.*

1.1. Introdução

As tradicionais redes de computadores e, mais especificamente, as operadoras de redes de telecomunicações por um longo tempo foram baseadas intensamente em hardware. Grandes fabricantes de redes como Cisco, Juniper, Huawei, entre outros desenvolviam soluções para atender a uma demanda específica. Muitas funcionalidades de um determinado fabricante eram incompatíveis com as de outros, dificultando consideravelmente a interoperabilidade do sistema. Dessa forma, muitos parques computacionais eram exclusivamente de um ou outro fabricante, causando uma disputa acirrada entre quem poderia oferecer a melhor solução satisfazendo os quesitos de funcionalidade e robustez associado com um baixo preço de venda.

Nos últimos anos, as redes se expandiram exponencialmente com o aumento do número de usuários, da infraestrutura física (cobertura de rede e *data centers*) e o surgimento de novos e avançados serviços de rede. Houve então uma inversão dos paradigmas. As redes passaram a evoluir do hardware para o software. Soluções que antes eram baseadas em hardware passaram a ser fundamentadas em *software*. Dessa forma, novas tecnologias baseadas em software surgiram, por exemplo: virtualização de servidores, computação em nuvem, Redes Definidas por Software (*Software Defined Networking*, ou SDN) e mais recentemente a Virtualização de Funções de Rede (*Network Function Virtualization*, ou NFV). Todos são baseadas em *software*, sendo o *hardware* usado apenas como infraestrutura física. Esse novo conceito é conhecido como softwarização de redes [Sousa and Rothenberg 2017].

A virtualização de servidores e a computação em nuvem trouxeram maior flexibil-

idade e dinamismo para a criação, implantação e gerenciamento dos serviços. As Redes Definidas por Software proporcionaram maior programabilidade no controle de fluxo de dados ao desacoplar o plano de encaminhamento do plano de controle. Já a Virtualização de Funções de redes (NFV) promete automação e versatilidade para as funções de rede implementadas via *software*.

A softwarização de rede concentra-se no projeto, arquitetura, implantação e gerenciamento de componentes de rede, sendo essencialmente baseada em propriedades programáveis de software. Dessa forma, é possível maior flexibilidade, escalabilidade e controle sobre o requisitos e comportamentos da rede, atingindo uma melhor otimização e com baixo custo de manutenção dos serviços de rede. Além disso, seu potencial se mostra revolucionário ao impactar diversos aspectos de implantação e serviços de rede, resultando em avanços em aplicações como por exemplo, rede móveis, internet das coisas e *network slicing*.

Com o advento do SDN foi possível centralizar logicamente as redes e torná-las mais flexíveis. O protocolo OpenFlow(OF) [McKeown et al. 2008a] foi um dos primeiros a implementar esse conceito. A interface OF se iniciou de forma bem simples, com uma única tabela que poderia fazer "*match*" com os campos de um cabeçalho L2 (*Layer* 2). Entretanto, novas versões foram liberadas com suporte a novos cabeçalhos o que o tornou mais complexo. Além disso, nos últimos anos o processamento de pacotes eram realizados de forma inflexível nos ASICs (*Application Specific Integrated Circuits*) dos dispositivos de rede, o que tornava a alteração e inserção de novas funcionalidade um processo engessado.

Diante de tais limitações, foi-se necessárias mudanças na forma como os pacotes são processados. A programabilidade voltou-se essencialmente para o plano de dados. Hoje não é mais suficiente dizer o que fazer com os fluxos de dados, é necessário também descrever como fazer. Surgiram então novos paradigmas para abstrair o plano de dados em uma abordagem *top-down* de forma a permitir uma softwarização do mesmo. Nesse cenário e diante de outros desafios surgiu a linguagem de processamento de pacotes P4 (**P**rogramming **P**rotocol-independent **P**acket **P**rocessors) [Bosshart et al. 2014].

Baseado nesse contexto, este minicurso tem como principais objetivos:

1. Entender porque a programabilidade das redes está acontecendo e como ela pode ser usada;
2. Definir a tecnologia de encaminhamento de dados através do protocolo OpenFlow;
3. Abordar os princípios do plano de dados programável, mais especificamente a linguagem de programação P4;
4. Nortear os participantes através de atividades práticas com a linguagem P4, adquirindo conhecimentos para desenvolver novos casos de uso;
5. Discutir os desafios e avanços na área de programação no plano de dados;

A estrutura desse minicursos desenvolve-se em quatro seções: na seção 1.2 são mostrados os princípios de Redes definidas por software e a especificação do protocolo

OpenFlow; a seção 1.3 detalha os principais conceitos no plano de dados e a linguagem P4, sendo apresentados vários casos de usos com o P4; já a seção 1.5, demonstra os desafios e avanços em pesquisas de programação no plano de dados. Por fim, as conclusões do minicurso são apresentadas na última seção.

1.2. Programação no Plano de Controle

Um dos mais recentes paradigmas de sucesso em redes de computadores é a arquitetura de *Software Defined Networking* (SDN). Através da separação do plano de controle do plano de dados, as redes definidas por software capacitaram a centralização lógica do plano de controle, através de um controlador SDN ou *Network Operating System* (NOS). Sendo assim, usando redes SDN é possível se estabelecer redes flexíveis, programáveis e robustas [McKeown et al. 2008b].

Antes das redes SDN, existia grande limitação do que poderia ser feito com os equipamentos de redes, já que eram usados como caixas pretas com códigos proprietários e fechados. Dessa forma, para gerenciar esses equipamentos era necessário o treinamento de uma equipe técnica especializada nesses frameworks. Considerando a desvinculação do plano de controle do plano de dados (usando abordagens SDN), tornou-se possível o uso de interfaces *Open Source* com um controlador logicamente centralizado atuando como cérebro da rede. Atualmente, a interface *southbound* de referência SDN, na academia e na indústria, é a interface OpenFlow, sendo esta abordada em mais detalhes posteriormente nesse curso.

Através dos controladores SDN é possível executar diversas aplicações de rede já conhecidas, por exemplo: balanceamento de carga e roteamento. Além disso, usando esses dispositivos também é possível a aplicação de novos serviços como conectividade de rede fim-a-fim usando múltiplos domínios administrativos. Sendo assim, visando simplificar a rede, os controladores abstraem a mesma escondendo detalhes das camadas inferiores [Bernardos et al. 2015].

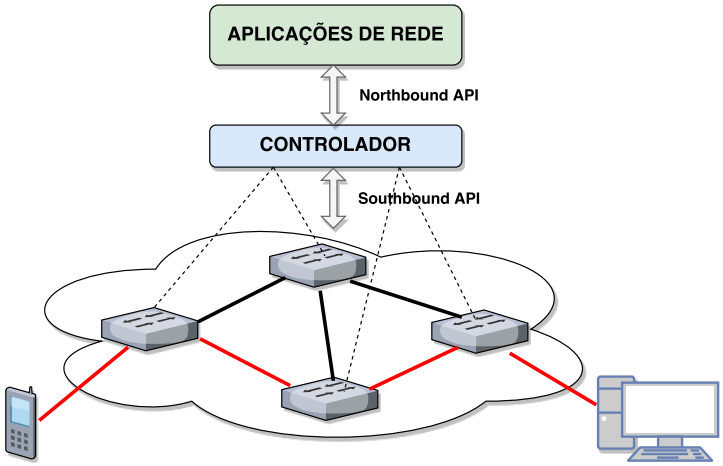

Figura 1.1. Visão simplificada de uma arquitetura SDN. Adaptado de [Kreutz et al. 2015]

1.2.1. OpenFlow

O OpenFlow foi criado com o objetivo de fornecer alto desempenho e controle de tráfego granular [McKeown et al. 2008b]. Ele é um protocolo *open source* muito adotado comercialmente em redes SDN. Algumas fabricantes com suporte ao OpenFlow são: HP, Cisco, Juniper e Extreme.

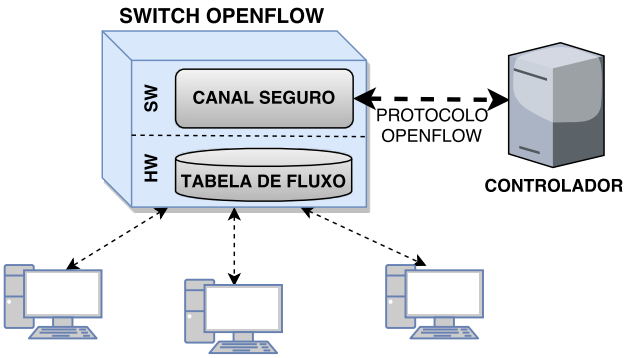

Figura 1.2. Visão geral da arquitetura do switch OpenFlow. Adaptado de [McKeown et al. 2008b]

O protocolo OpenFlow (OF) se baseia no uso das tabelas de fluxos nos equipamentos de rede. A interface OF estabelece a conexão entre o switch com suporte ao protocolo e o controlador, estando assim alinhado com o conceito de SDN. O controlador gerencia os fluxos de dados, garantindo uma centralização lógica da rede. A Figura 1.2 expõe uma visão geral da arquitetura SDN usando o OF.

Através da tabela de fluxos o switch OpenFlow verifica se o tráfego que passa por ele condiz com uma das entradas nas tabelas. Dessa forma, caso haja uma correspondência, o switch aplica aquela regra e encaminha o pacote adequadamente. Entretanto, caso não haja correspondência, o switch informa o o controlador através de um canal seguro e este último define uma nova regra na tabela de fluxos.

A primeira versão do OpenFlow (1.0 - lançada em 2009) continha apenas uma lista de regras organizadas em uma única tabela de fluxos. Dessa forma, a tabela de fluxos do protocolo era estruturada através de 4 colunas. A Tabela 1.1 ilustra a organização da primeira versão do OF, sendo detalhado abaixo cada um dos termos utilizados:

1. **Priority:** define a ordem de prioridade das regras;
2. **Pattern:** padrão de correspondência com os cabeçalhos dos pacotes;
3. **Actions:** ações a serem aplicadas no fluxo (exemplos *drop*, *forward* e *send to controller*);
4. **Counters:** estatísticas de número de bytes e pacotes;

Entretanto, a primeira versão do OF apresentava números limitados de suporte a cabeçalhos específicos, usados para as regras do tráfego. Dessa forma, eram desejadas aplicações mais complexas, com tabelas em série e múltiplos estágios de correspondência.

Tabela 1.1. Exemplo de regras do OpenFlow 1.0

Priority	Pattern	Actions	Counters
2	srcp=10.0.0.*	Forward(2)	2500
1	dstip=4.4.3.3,dstport=80	dstip:10.0.0.10, Forward(1)	3600
0	dstport=5050	Send to Controller	480

Ao longo dos anos, novas novas versões foram propostas com suporte a novos cabeçalhos e regras mais complexas.

Conforme mostrado na Tabela 1.2 a quantidade de campos do protocolo OF ao longo dos anos cresceu consideravelmente. Inicialmente, esse crescimento pode ser identificado de forma positiva, como uma consequência de um bom desenvolvimento da interface. Entretanto, grandes consequências podem ser identificadas a partir dessa mesma tabela. O aumento do número de campos também torna o protocolo "pesado", podendo sobrecarregar as redes que o utilizam. Além disso, embora novos cabeçalhos possam teoricamente surgir ao longo do tempo, o OF foi atualizado em média apenas uma vez por ano, devido a necessidade de retrocompatibilidade, padronização com a indústria, entre outros.

Tabela 1.2. Evolução do protocolo OpenFlow [Bosshart et al. 2014].

Versão	Data	# Cabeçalhos
OF 1.0	Dez 2009	12
OF 1.1	Fev 2011	15
OF 1.2	Dez 2011	36
OF 1.3	Jun 2012	40
OF 1.4	Out 2013	41

Atualmente, o protocolo OpenFlow é gerenciado pela *Open Networking Foundation* (ONF)[1]. Essa organização não tem fins lucrativos e tem como objetivo desenvolver padrões Open Source de redes SDN.

1.3. Programação no Plano de Dados

Tradicionalmente, redes corporativas são compostas por roteadores e switches, podendo conter um ou mais fabricantes. A maior parte desses equipamentos usam sistemas operacionais baseados no Unix ou Linux. Através de drivers específicos, o sistema operacional faz a comunicação com o processador ASIC (Application Specific Integrated Circuits), executando assim as aplicações de rede. A Figura 1.3 ilustra o funcionamento dos equipamentos de redes usando diversos protocolos sob um sistema operacional dedicado ao *Hardware* e que passa as instruções para o processador ASIC usando um driver.

Considerando esse rígido cenário, novas funcionalidades podem ser implementadas apenas pelo fabricante original e de acordo com um ciclo de atualização limitado e longo. Além de necessária a atualização do sistema operacional do equipamento, em alguns momentos são necessários novos drivers que devem ser testados antes de

[1] https://www.opennetworking.org/

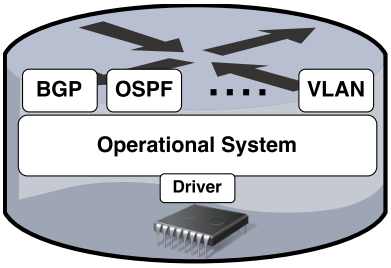

Figura 1.3. O funcionamento de roteadores e switches tradicionais. [Sousa and Rothenberg 2017]

serem lançados. Dessa forma, esse ciclo de *update* pode levar de meses até anos para estarem disponíveis aos usuários finais. Considerando ambientes corporativos dinâmicos e flexíveis, esse processo prejudica o funcionamento da rede.

As redes tradicionais foram desenvolvidas através de configurações fixas com uma hierarquia do tipo *bottom-up*, isto é, de baixo para cima, em que os software era desenhado para se adequar as características do processador (*Hardware*). Entretanto, esse modelo rígido e extremamente dependente do *Hardware* impossibilita rápidos *updates*, tornando-o atrasado e dificultando em muito a adoção de novas tecnologias nas redes. Entretanto, nos últimos anos novas pesquisas têm sido feitas com o objetivo de habilitar a abordagem do tipo *top-down*, isto é, de cima para baixo, em que o *Software* dita o que deve ser feito para o *Hardware*. Dessa forma, é possível (re)programar o plano de dados de forma a disponibilizar novos serviços e funcionalidades sem necessidade de se adquirir novos equipamentos. As consequências dessa mudança de paradigma são drásticas: o que antes era-se feito em anos, hoje pode ser feito em meses, dias ou até mesmo horas. A figura 1.4 demonstra uma visão geral do funcionamento de um equipamento de rede usando uma linguagem de programação de alto nível.

Figura 1.4. Uma visão geral do funcionamento de equipamentos de redes programáveis. Adaptado de [Bosshart et al. 2014]

Considerando os problemas apontados anteriormente, foi-se estabelecida a proposta de uma linguagem que possibilite programabilidade a nível de plano de dados. Essa linguagem foi chamada de P4, uma uma nova proposta tem sido estabelecida visando três principais objetivos:

1. Os analisadores dos pacotes devem ser configuráveis, não havendo necessidade de ter um cabeçalho específico;

2. Na tabela de Match+Action deve ser possível fazer a correspondência de todos os campos definidos e suportar múltiplas tabelas;

3. Cabeçalhos e metadados podem ser modificados usando funções básicas como: *copy*, *add*, *remove*, *modify*;

Existem diversos desafios identificados e que têm sido abordados pela indústria e pela comunidade acadêmica. Entretanto, a abordagem referência nesse contexto de programabilidade em plano de dados é a linguagem P4 [Bosshart et al. 2014].

1.3.1. A linguagem P4

Considerando as dificuldades encontradas no protocolo OpenFlow citadas anteriormente e a necessidade de redes com plano de dados programáveis, a linguagem P4 é estabelecida em uma parceria entre grandes nomes da indústria (Google, Microsoft, Intel e Barefoot Networks) e da academia (Universidade de Stanford e Princeton) [Bosshart et al. 2014]. Para a implementação do P4 três desafios eram previstos inicialmente [Bosshart et al. 2014], são eles:

1. **Processador de pacotes configurável:** a análise (*parser*) do programa deve suportar a declaração de cabeçalhos, de forma que essa análise **não** esteja atrelada a um formato de pacote específico ou cabeçalho, permitindo assim a inserção de novos cabeçalhos ao longo dos anos;

2. **Flexibilidade na tabela de pacotes:** configuração das tabelas de correspondência, permitindo estruturas em série, paralelas ou até híbridas. Dessa forma, é possível inserir novas funcionalidades no programa;

3. **Primitivas genéricas de processamento de pacotes:** primitivas genéricas que alterem metadados e dados no pacote devem estar disponíveis, podendo ser facilmente utilizadas, primitivas como: *copy*, *add*, *remove* e *modify*;

Esses três desafios têm sido pesquisados pela indústria e pela academia em diversas abordagens promissoras. Dessa forma, percebemos uma grande movimentação em direção de tecnologias programáveis [Kim 2016]

- Nova geração de switches Application Specific Integrated Circuits (ASIC): Barefoot Tofino, Cisco Doppler, Cavium (Xpliant), Intel Flexpipe;

- FPGA: Altera, Xilinx;

- CPU: DPDK, Open Vswitch, VPP, eBPF;

- Network Processor Unit (NPU): Netronome, EZchip;

Através desses equipamentos é possível realizar a programação do processamento de dados com alta performance e um custo que justifique essa mudança. Entretanto, sem a linguagem P4 (**P**rogramming **P**rotocol-independent **P**acket **P**rocessors) [Bosshart et al. 2014] a programação desses equipamentos se mostra difícil e muitas vezes única, já que possuem interfaces proprietárias com programação de baixo nível. Através da linguagem P4 é possível então abstrair o plano de dados tornando a programação do comportamento dos equipamentos de rede mais simples e uma programação de alto nível.

A linguagem P4 se baseia na arquitetura PISA (Protocol-Independent Switch Architecture), com chips de capacidade de até Tb/s. Através do PISA temos um novo paradigma com hardware baseado em um *pipeline* de (re)configuração usando o modelo de Match+Action. Dessa forma, diferentemente dos tradicionais ASICs, a arquitetura provê grande flexibilidade sem comprometer a performance. A Figura 1.5 demonstra de forma genérica o processo de um equipamento usando PISA.

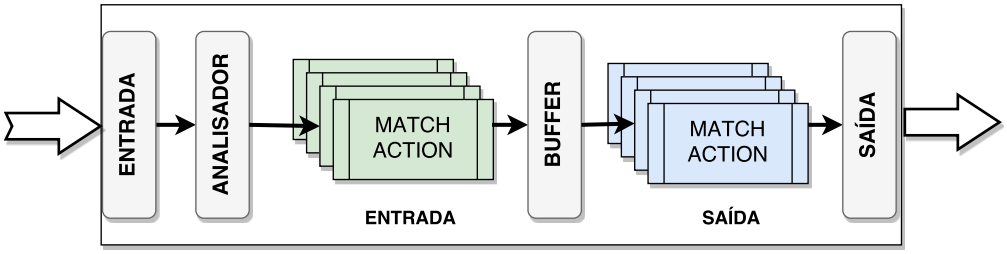

Figura 1.5. Arquitetura PISA. Adaptado de [Bosshart et al. 2014]

A linguagem P4 é gerenciada por uma organização chamada *P4 Language Consortium* [2], tendo sua especificação padronizada pelo grupo. A linguagem abstrai a programação de redes em busca de três objetivos [Bosshart et al. 2014]:

- **Independência do protocolo:** O switch deve ser capaz de analisar qualquer tipo de protocolo, desde que declarado anteriormente no programa como analisar esse cabeçalho e as ações usando o modelo de Match+Action;
- **Independência de arquitetura:** O programa P4 deve abstrair a camada física, sendo ele o mesmo para diferentes equipamentos (*targets)*. Dessa forma, as peculiaridades de cada equipamento deverá ser tratado pelo próprio compilador;
- **Reconfiguração em campo:** O processamento dos pacotes pode ser alterado uma vez implantado, até mesmo em tempo de execução;

Conforme mostrado na Figura 1.5, quando um pacote chega no switch/roteador P4, os cabeçalhos são primeiramente analisados de acordo com os critérios definidos previamente no programa. Dessa forma, os campos são extraídos. Feito isso, as tabelas passam a fazer o Match de acordo com os campos extraídos e tomam as devidas ações de acordo com instruções passadas pelo controlador ao popular as tabelas. A especificação da linguagem sugere subdividir as tabelas em duas partes:

[2]https://p4.org

- **Ingress:** define portas de saída e fila para o pacote;

- **Egress:** modifica o cabeçalho do pacote, removendo ou adicionando campos (por exemplo, GRE ou VxLAN);

Para a construção de um programa P4 é necessário preencher cinco campos [Bosshart et al. 2014], conforme mostrado na Figura 1.6. Por padrão esses campos são definidos na seguinte ordem no programa P4 (versão 14):

1. **Headers:** são os campos que contém dados dentro do pacote e são analisados pelo *parser*;

2. **Parser:** responsáveis pela análise e extração dos campos do pacote;

3. **Tables:** mecanismo que faz o processamento do pacote. Dentro das tabelas tem as correspondências (*matches*) e as ações a serem executadas;

4. **Actions:** conjunto de primitivas usadas para uma certa função customizada, podendo conter primitivas como: *copy_header*, *remove_header*, *add*, etc;

5. **Control:** Última parte do programa P4, ele estabelece o controle de fluxo das tabelas;

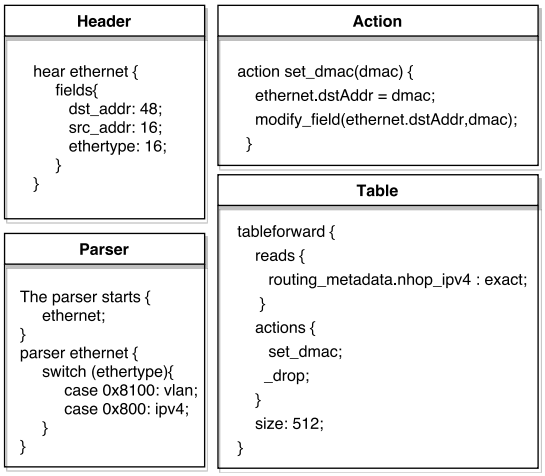

Figura 1.6. Exemplo de um programa $P4_{14}$.

Como a maioria das linguagens, um programa P4 precisa de um compilador para traduzir o programa para uma linguagem de baixo nível, esse compilador pode utilizar *hardware* ou *software*. No caso do *parser* programável, o compilador faz a conversão da descrição de como analisar os cabeçalhos dentro de uma máquina de estados. A versão mais atual da linguagem *p*4 é a versão 16, ou simplesmente $P4_{16}$[3].

[3]https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

1.3.2. O $P4_{16}$

Os objetivos da versão 16 é composto por:

1. Suporte à múltiplas arquiteturas: A especificação do $P4_{14}$ inclui uma arquitetura fixa e rígida, representando um switch com *ingress* e *egress*. Um dos objetivos principais do $P4_{16}$ é a execução de programas P4 em múltiplos targets, aceitando diferentes arquiteturas e com diferentes capacidades(por exemplo FPGAs, placas de rede, software switches, etc.).

2. Simplicidade: Através do $P4_{14}$ é necessário diversos comandos e *primitives* para uma determinada ação. No $P4_{16}$ muitas dessas ações tem sido compactadas em expressões menores (por exemplo: na versão 14 o tínhamos o comando `add_to_field(a,b)` que foi simplificado na versão 16 para `a=a+b`).

3. Compatibilidade: Idealmente as futuras versões de P4 devem ser retro-compatíveis com as atuais. Dessa forma, os programas escritos hoje devem continuar compilando e executando nas futuras versões da linguagem.

4. Extensibilidade: muitas funções de $P4_{14}$ foram adicionadas nas características do modelo sendo características próprias de uma arquitetura de um target específico. Tais funções têm sido convertidas em objetos e funções **extern**, sendo partes específicas da linguagem no Target.

A sintaxe da linguagem $P4_{16}$ foi baseada em linguagens familiares, por exemplo a linguagem C, que se baseia na execução sequencial do código, adiciona bibliotecas, programação orientada a objetos e declarações.

1.3.2.1. Arquiteturas de $P4_{16}$:

Uma das maiores mudanças do $P4_{16}$ foi permitir que programas compilados em P4 serem executados em diferentes targets. Portanto, essa nova versão permite definir a arquitetura de cada target (Software switch, FPGA, ASIC etc.) como uma biblioteca que será adicionada no início do codigo P4 sendo dos fabricantes de dispositivos de rotamento a responsabilidade de definir sua própria arquitetura para ser usada no código de P4. A arquitetura usada para o target de Behavioral-model 2 (Bmv2) é denominada “v1model”. Usualmente, os programas P4 começam incluindo a biblioteca padrão “core.p4” e a biblioteca que descreve a arquitetura. Na próxima sub-seção introduziremos um pouco o P4C compiler e seus principais parâmetros para compilar código de JSON para Bmv2 tanto para a versão 14 como também a versão 16. Em [Mejia and Rothenberg 2017] podemos ver a arquitetura e implementação de um Switch para o target de Bmv2 com P4_16.

1.3.2.2. Compilador P4C:

O compilador P4C foi criado para executar ambas versões, isto é $P4_{14}$ e $P4_{16}$, sendo escrito em C++ e provendo suporte para diferentes *backends*. Dependendo do tipo de

arquitetura do programa P4 de entrada (v1 model ou eBNF) este pode compilar para quatro tipos diferentes de *backend*:

- **p4c-bm2-ss:** usado para o BMV2 fornecendo um arquivo JSON de saída;
- **p4c-ebpf:** usado para gerar código C para o target eBNF (extended Berkeley Packet Filter);
- **p4test:** usado para converter de $P4_{14}$ para $P4_{16}$ com o proposito de aprendizado, testing, ou debugging;
- **p4c-graphs:** Pode ser usado para gerar representações visuais de um programa em P4;

Quando a arquitetura de entrada é da arquitetura $P4_{14}$, o código é primeiro convertido para $P4_{16}$ aplicando a arquitetura v1model. Esta versão pode ser baixada do repositório: https://github.com/p4lang/p4c
Para gerarmos o JSON apartir do codigo $P4_{16}$ executamos o seguinte comando:

```
p4c-bm2-ss l2switch.p4 -o l2switch.json
```

Para gerarmos o JSON apartir do codigo $P4_{14}$ executamos o seguinte comando:

```
p4c-bm2-ss --p4v 14 l2switch.p4 -o l2switch.json
```

1.4. Atividade Prática

Essa seção de atividade prática tem como objetivo ser um guia introdutório da linguagem P4. A seguir apresentaremos dois estudos de caso usando $P4_{14}$, geração de representações através de um programa $P4_{14}$ e por último, outros dois exemplos dessa vez usando $P4_{16}$. Dessa forma, utilizaremos uma máquina virtual[4] contendo ambas as versões da linguagem P4 e o software `mininet`. Vale ressaltar que para todos os experimentos a seguir devemos ter as interfaces virtuais (veths) criadas. Sendo assim, é necessário executar o seguinte comando:

```
cd /home/p4/behavioral-model/
./tools/veth_setup.sh
```

1.4.1. simple_router ($P4_{14}$)

Nesse ambiente usaremos o `mininet` para simular a topologia estabelecida na Figura 1.7, em que conectamos duas estações (*hosts*) a um único roteador. Feito isso, efetuaremos um simples teste de roteamento usando o programa `simple_router`:

1. Nos direcionamos para a pasta com o target simple_router e convertemos o arquivo p4 para o formato json usando p4c-bmv2:
   ```
   cd /home/p4/behavioral-model/targets/simple_router/
   sudo p4c-bmv2 --json simple_router.json simple_router.p4
   ```

[4]Link da máquina virtual: https://goo.gl/2GeyL6

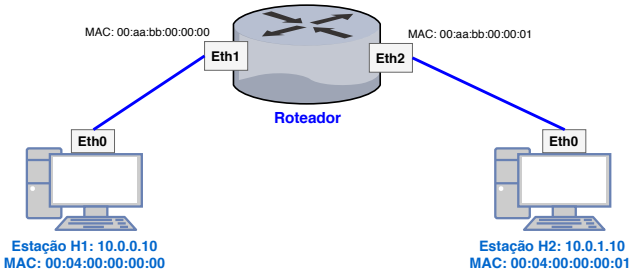

Figura 1.7. Topologia de camada 3 usando a linguagem P4.

2. Iniciamos então a topologia usando o `mininet`:
   ```
   cd /home/p4/behavioral-model/mininet/
   sudo python 1sw_demo.py --behavioral-exe
   ../targets/simple_router/simple_router --json
   ../targets/simple_router/simple_router.json
   ```

3. Em um novo terminal populamos a tabela com as entradas necessárias:
   ```
   ./runtime_CLI < commands.txt
   ```

4. Então inserimos o seguinte comando no `mininet` para fazer o teste:
   ```
   mininet >h1 ping h2
   ```
   A saída deve ser similar as linhas abaixo:
   ```
   PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
   64 bytes from 10.0.1.10: icmp_seq=7 ttl=63 time=4.71 ms
   64 bytes from 10.0.1.10: icmp_seq=8 ttl=63 time=4.48 ms
   64 bytes from 10.0.1.10: icmp_seq=9 ttl=63 time=0.960 ms
   ```

1.4.2. simple_switch_lb ($P4_{14}$)

Nesse exemplo efetuaremos o teste de um roteador fazendo o balanceamento de carga através da função *csum16*. A Figura 1.8 representa a arquitetura do teste de balanceamento de carga. Vale ressaltar que o *simple_switch* é um modelo de roteador mais genérico que suporta uma gama maior de programas P4 que o simple_router, isto é, programas que rodam no simple_router, podem ser executados também no simple_switch, no entanto, a reversa não é necessariamente válida.

1. Nos direcionamos para a pasta com o target simple_switch e convertemos o arquivo p4 para o formato json usando p4c-bmv2:
   ```
   cd targets/simple_switch_lb
   sudo p4c-bmv2 --json simple_switch_lb.json simple_switch_lb.p4
   ```

2. Iniciamos então o switch com o seguinte comando:
   ```
   sudo ./simple_switch -i 0@veth1 -i 0@veth0 -i 1@veth3
   simple_switch_lb.json
   ```
   Sendo as veths interfaces virtuais usadas para simular as interfaces conectadas ao switch/roteador P4.

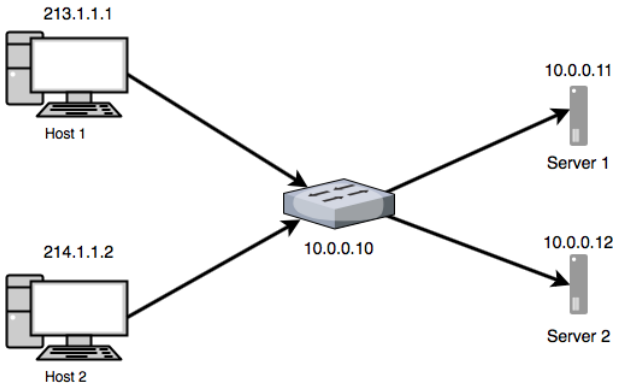

Figura 1.8. Arquitetura de teste *simple_switch_lb.p4*

3. Feito isso, em um novo terminal populamos a tabela com as entradas necessárias (porta 9090):
```
cd /home/p4/behavioral-model/targets/simple_switch_lb
sudo ./runtime_CLI < commands.txt
```

4. Feito isso, executamos um analisador de pacotes, usaremos tcpdump para fazer a análise:
```
sudo tcpdump -XXi veth3
```

5. Por último, criamos diferentes pacotes (alterando apenas o endereço IP de origem) com scapy e enviamos os mesmos. O pacote encaminhado pode ser analisado no terminal rodando o tcpdump. A Figura 1.9 demonstra o balanceamento sendo distribuído entre dois IPs exemplos:
```
sudo scapy
>>> pkt1 = Ether(dst='a2:5e:37:ac:a1:7f',src='fa:4f:e8:df:b1:5f')
/IP(dst='10.0.0.10',src='214.1.1.2')
>>> pkt2 = Ether(dst='a2:5e:37:ac:a1:7f',src='fa:4f:e8:df:b1:5f')
/IP(dst='10.0.0.10',src='213.1.1.1')
>>> sendp(pkt1,iface="veth0",count=1);
>>> sendp(pkt2,iface="veth0",count=1);
```

```
root@p4-VirtualBox:/home/p4# tcpdump -XXi veth3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth3, link-type EN10MB (Ethernet), capture size 262144 bytes
22:15:36.766854 IP 214.1.1.2 > 10.0.0.12:  hopopt 0
        0x0000:  0004 0000 0000 00aa bb00 0000 0800 4500  ..............E.
        0x0010:  0014 0001 0000 3f00 9ada d601 0102 0a00  ......?.........
        0x0020:  000c                                     ..
22:16:03.902003 IP host213-1-1-1.range213-1.btcentralplus.com > 10.0.0.11:  hopopt
 0
        0x0000:  0004 0000 0000 00aa bb00 0000 0800 4500  ..............E.
        0x0010:  0014 0001 0000 3f00 9bdc d501 0101 0a00  ......?.........
        0x0020:  000b                                     ..
```

Figura 1.9. Balanceamento de carga usando $P4_{14}$

Percebemos que o resultado é o balanceamento da carga aleatoriamente com base na função *csum16* entre o servidor com ip 10.0.0.11 e 10.0.0.12. Entretanto, vale ressaltar que o balanceamento aleatório é feito apenas no primeiro acesso do *host*, já que uma vez

determinado o servidor que atenderá o cliente, esse será sempre o mesmo. Dessa forma, o IP de origem 213.1.1.1 será sempre atendido pelo servidor com IP 10.0.0.11.

1.4.3. Geração de representações do programa $P4_{14}$

Visando facilitar a compreensão dos programas P4, criou-se um *framework* que gera fluxogramas de diferentes partes do P4. Nesse tópico usaremos como exemplo o programa `simple_router` fornecido no exemplo da subseção 1.4.1.

Para gerar esses fluxogramas executamos o seguinte comando no terminal da VM disponibilizada:

```
$ p4-graphs simple_router.p4
```

A Figura 1.10 ilustra as dependências das tabelas no `ingress`. Dessa forma, é possível notar que existem duas condições básicas para a execução de qualquer tabela no programa `simple_router`, são elas: verificação da existência de um cabeçalho IPv4 no pacote recebido e um *time-to-live* (ttl) maior que 0. Caso uma ou ambas as condições não sejam satisfeitas, o programa passa automaticamente para a as tabelas de `egress`.

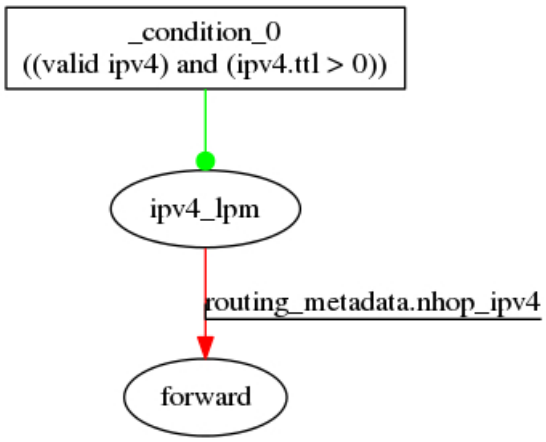

Figura 1.10. Fluxograma de dependências do *ingress*.

Além disso, também é possível fazer um fluxograma da análise (o *parser*) do programa. A Figura 1.11 ilustra a análise de um pacote com seu cabeçalhos. Percebemos que em nosso programa `simple_router` é feita uma análise do pacote *ethernet* por padrão. Dessa forma, caso o roteador encontra um *Ethertype* 0x0800, ele passa a análise para o pacote ipv4. Caso contrário, o programa assume por padrão que não há mais campos no pacote e termina sua análise.

Por último, também é possível gerar um fluxograma das tabelas que são usadas em nosso programa, incluindo tanto o `ingress` quanto o `egress`. A Figura 1.12 representa o fluxograma de tabelas. A seguir definiremos brevemente a função de cada tabela:

- **ipv4_lpm:** essa tabela faz a correspondência (match) do IP (IPv4) de destino. Existem duas possíveis ações a serem tomadas:

  _drop: recusa/ignora o pacote recebido;

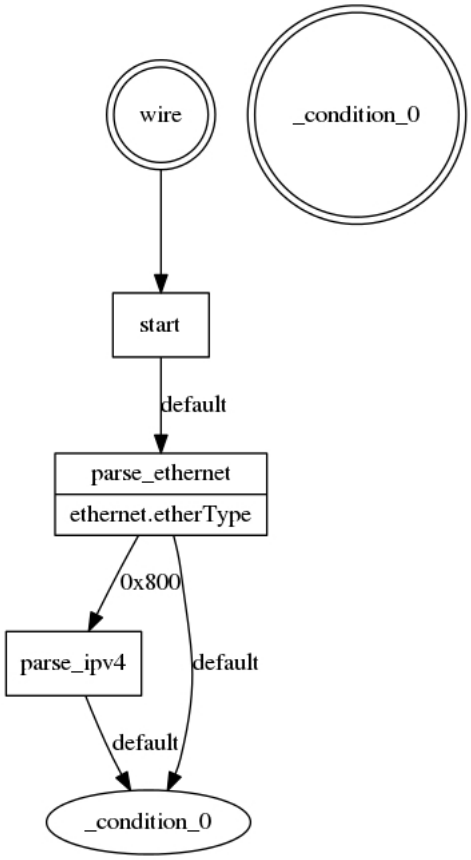

Figura 1.11. Fluxograma da análise de pacotes.

set_nhop: configura o próximo *hop* IPv4, a porta de saída e diminui o valor do ttl em uma unidade;

- **forward:** essa tabela tem a função de fazer o encaminhamento do pacote, para isso ela faz a correspondência (match) no próximo *hop* IPv4. Existem duas possíveis ações a serem tomadas:

  _drop: conforme comentado anteriormente, recusa/ignora o pacote recebido;

  set_dmac: configura o mac de destino do protocolo *ethernet*;

- **send_frame:** essa tabela é responsável por reescrever o MAC de origem de acordo com a porta de saída do roteador, por isso ela faz a correspondência (match) no metadado que corresponde a porta de saída. Existem duas possíveis ações a serem tomadas:

  rewrite_mac: como o nome já diz, essa ação reescreve o MAC de origem de acordo com o mac da porta estabelecida;

  _drop: conforme comentado anteriormente, recusa/ignora o pacote recebido;

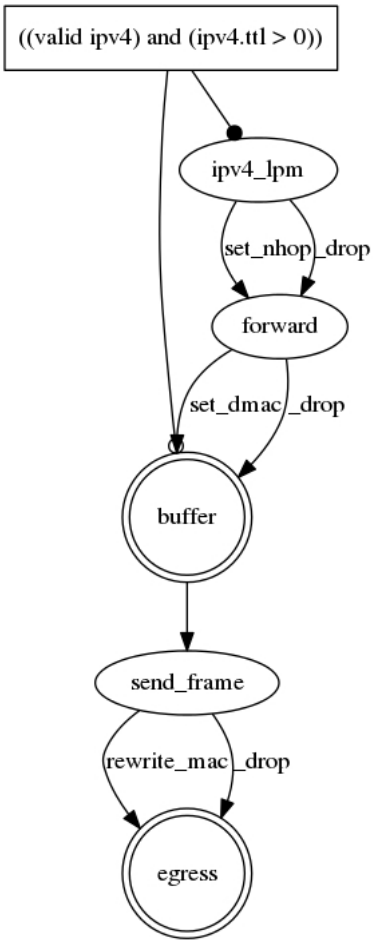

Figura 1.12. Fluxograma das tabelas contidas no simple_router.

1.4.4. Simple router+acl ($P4_{16}$)

Nesse ambiente usaremos o `mininet` para simular a topologia estabelecida na Figura 1.7, em que conectamos duas estações (*hosts*) a um único roteador. Feito isso, efetuaremos um simples teste de roteamento usando uma versão modificada do `simple_router` anterior e com uma lista de controle de acesso.

1. Nos direcionamos para a pasta com o target `acl` e convertemos o arquivo .p4 para o formato JSON usando o compilador para a versão $P4_{16}$:
   ```
   cd ~/minicurso_p4/examples/acl/
   sudo p4c-bm2-ss simple_acl_16.p4 -o ss_acl.json
   ```

2. Iniciamos então a topologia usando o `mininet`:
   ```
   cd ~/behavioral-model/mininet/
   sudo python 1sw_demo.py --behavioral-exe
   ../targets/simple_switch/simple_switch --json
   ~/minicurso_p4/examples/acl/ss_acl.json
   ```

3. Usando um novo terminal populamos a tabela com as entradas necessárias:

```
cd ~/behavioral-model/targets/simple_switch
./runtime_CLI < ~/minicurso_p4/examples/acl/acl_comm.txt
```

4. Então inserimos o seguinte comando no `mininet` para fazer o teste:
   `mininet >h1 ping h2`

   A saída deve ser similar as linhas abaixo:
   ```
   PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
   64 bytes from 10.0.1.10: icmp_seq=7 ttl=63 time=4.71 ms
   64 bytes from 10.0.1.10: icmp_seq=8 ttl=63 time=4.48 ms
   64 bytes from 10.0.1.10: icmp_seq=9 ttl=63 time=0.960 ms
   ```

5. Então executamos o script de CLI e inserimos a entrada que bloqueia qualquer tráfego que passe pela tabela *acl*:
   ```
   ./runtime_CLI
   RuntimeCmd:table_set_default acl _drop
   ```

6. Feito isso, no terminal do `mininet` efetuamos um ping novamente com o seguinte comando:
   `mininet >h1 ping h2`

   A saída esperada deverá ser similar as linhas abaixo:
   ```
   PING 10.0.1.10 (10.0.1.10) 56(84) bytes of data.
   Request timeout for icmp_seq 0
   Request timeout for icmp_seq 1
   Request timeout for icmp_seq 2
   ```

   Percebemos que conforme esperado todo o tráfego está sendo bloqueado.

7. Podemos então fazer o teste com filtro do trafego através do comando *curl*.
   No console do mininet, abra o *xterm* do h1 e h2.

   No h2 executamos o seguinte (ver Fig.1.13):
   ```
   sudo fuser -k 80/tcp
   pushd ~/minicurso_p4/examples/acl/webpage/; python3 -m http.server
   ```

   No h1 (ver Fig.1.14):
   `curl http://10.0.1.10`

8. Executamos então novamente o script CLI e adicionamos a seguinte entrada na tabela *acl* visando permitir o trafego:
   ```
   ./runtime_CLI
   RuntimeCmd: table_set_default acl _nop
   ```

9. Novamente efetuamos o teste com o comando *curl*, mas dessa vez sem filtragem do trafego:
   No h2 (ver Fig.1.13):
   ```
   sudo fuser -k 80/tcp
   pushd /home/webpage/; python3 -m http.server 80 &
   ```

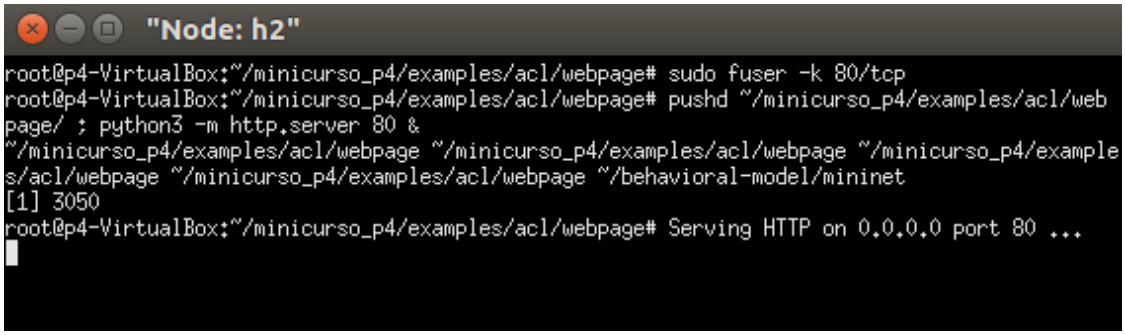

Figura 1.13. Iniciando o servidor web no Host 2.

Figura 1.14. Executando uma requisição Http do Host 1 para o servidor web sem sucesso.

No h1 (ver Fig.1.15):
`curl http://10.0.1.10`

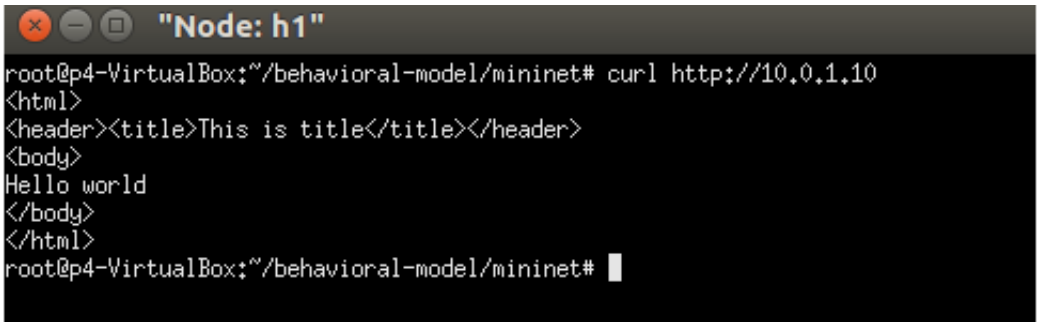

Figura 1.15. Executando uma requisição Http do Host 1 para o servidor web com sucesso.

1.4.5. NAT ($P4_{16}$)

Nesse ambiente usaremos o `Wireshark` e `Behavioral model 2` para simular a topologia estabelecida na Figura 1.16, em que conectamos duas estações (*hosts*) em diferentes redes a um servidor NAT. Feito isso, efetuaremos um simples teste de roteamento.

1. Nos direcionamos para a pasta que contém o exemplo e convertemos o arquivo p4 para o formato JSON usando o compilador do $P4_{16}$:
   ```
   cd ~/minicurso_p4/examples/nat
   sudo p4c-bm2-ss nat.p4 -o nat.json
   ```
2. Iniciamos as interfaces virtuais através do seguinte comando:
   ```
   sudo ./veth_create.sh
   ```

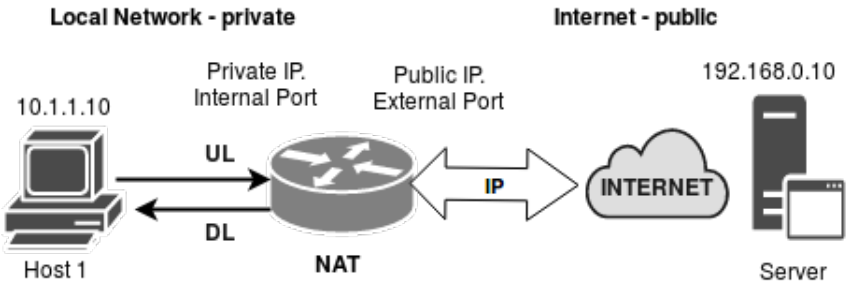

Figura 1.16. Topologia de rede NAT. Fonte [Mejia et al. 2018].

3. Feito isso, iniciamos servidor NAT com o target `simple_switch`:
   `sudo simple_switch -i 1@veth1 -i 2@veth3 --thrift-port 9090 --nanolog ipc:///tmp/bm-0-log.ipc --device-id 0 nat.json`

4. Em um segundo terminal populamos a tabela com as entradas necessárias:
   `./runtime_CLI < nat_entries.txt`

5. Em um terceiro terminal iniciamos Wireshark com permissão de root e adicionamos as interfaces veth1 e veth3:
   `sudo wireshark`

6. Então inserimos o seguinte comando no terminal para fazer o teste de Upload:
   `sudo python send_packet.py`
   Na Fig.1.17 é possível identificar os pacotes capturados no programa Wireshark nas interfaces selecionadas.

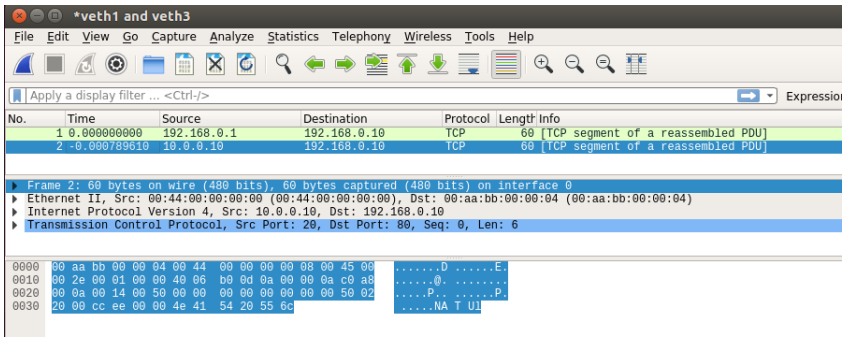

Figura 1.17. Captura de pacotes com Wireshark nas interfaces de NAT UL.

7. Então inserimos o seguinte comando no terminal para o teste de Download:
   `sudo python send_packet.py --dl`
   Na Fig.1.18 é possível identificar os pacotes capturados no programa Wireshark nas interfaces selecionadas.

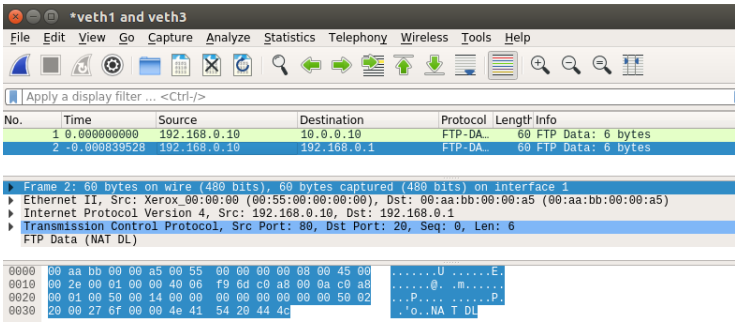

Figura 1.18. Captura de pacotes com Wireshark nas interfaces de NAT DL.

1.5. Desafios

Na área de programação do plano de dados existem diversos desafios que necessitam de atenção da comunidade científica, indústrias e órgãos de padronização. Muitos avanços foram obtidos a partir de processadores programáveis e linguagens que abstraem o plano de dados, como a linguagem P4, mas ainda há muito a ser desenvolvido. Discuti-se nos próximos parágrafos os principais desafios nesse campo.

Atualmente os sistemas de redes estão cada vez mais virtualizados, utilizando para isso tecnologias como SDN e NFV. Nesse ambiente altamente virtualizado com foco no software, um dos principais desafios é o desempenho da rede. Tal desafio reflete-se principalmente no plano de dados onde o desempenho é essencial para atender aos requisitos dos serviços.

Inicialmente, o desempenho era avaliado sobre sistemas baseados em hardware como *appliances*, roteadores e switches. Nesse quesito o hardware específico sempre tinha melhor desempenho ao ser comparado com softwares rodando em hardwares genéricos. Entretanto, esse cenário vem mudando e o *gap* de desempenho entre hardware e software vem diminuindo ao longo dos anos, principalmente devido às novas tecnologias como: DPDk, ClickOS e FPGA. Nesse sentido, o P4 permite levar para o plano de dados os vários benefícios introduzido pelo SDN de forma a otimizar o desempenho das redes.

Uma infraestrutura de rede softwarizada e virtualizada modifica a forma como os fluxos de dados e serviços são implementados. Dessa forma, existe um impacto relevante na segurança dessas redes [Freire et al. 2018]. Novas funcionalidades e recursos necessitam ser implantados, incluindo a capacidade de gerenciamento avançado de autenticação, identidade e controle de acesso. A programabilidade e flexibilidade oferecida pela programação de processadores de pacotes não deve deixar brechas para acesso indevidos às tabelas Match+Action e manipulação de metadados [Freire et al. 2017].

As tecnologias, tais como SDN, NFV e mais recentemente P4, propõem diversas inovações e arquiteturas voltadas para a programabilidade em redes. No entanto, a integração dessas tecnologias é um grande desafio. Nota-se que P4, SDN e NFV são ferramentas **complementares** com o objetivo de alcançar uma programabilidade completa da rede. É importante ressaltar que SDN e NFV são independentes, mas isso não ocorre entre P4 e SDN. A linguagem P4 pode ser comparada a uma "versão 2.0" do protocolo OpenFlow. Enquanto SDN traz "inteligência" e programabilidade no plano de controle,

a linguagem P4 permite uma maior flexibilidade diretamente no plano de dados [Feferman and Rothenberg 2017]. Então é necessário existir uma estreita relação entre o plano de dados e o de controle. No entanto, não existe uma padronização de como deve ser essa relação SDN e P4, por exemplo, novas funcionalidades devem ser propostas pelo controlador ou diretamente aplicada no dispositivo de rede com a compilação do P4? É necessário uma maior investigação no processo de integração entres essas tecnologias.

1.6. Avanços na programação do plano de dados

A programação do plano de dados associada com a linguagem P4 tem feito grandes avanços, em termos de processadores programáveis. Nesse contexto dois hardwares se destacam: o Tofino [5] (produzido pela Barefoot Networks) e FPGA. O Tofino é resultado de pesquisas da empresa Barefoot Networks, uma start-up fundada por alguns dos líderes da organização P4. O dispositivo é considerado o primeiro switch Ethernet programável, tendo ainda suporte para velocidade de até 6.5 Tb/s com total suporte à linguagem P4. Por outro lado, o FPGA é um circuito integrado um pouco mais difundido no ambiente de redes do que o Tofino. Ele também permite a programação mais aberta para o usuário final sem interferência do fabricante. O FPGA é vendido hoje em boa parte para ambiente de pesquisa, diferentemente do Tofino que é vendido como um produto B2B (Business to Business). Os resultados do FPGA comparados aos do Tofino são um pouco mais humildes, mas ainda assim surpreendem, chegando a mais de 10 Gbps[6], com baixíssima latência.

O P4 proporciona vários benefícios para a plano de dados como já listados durante esse minicurso. Com ele novas soluções surgiram como por exemplo, balanceador de carga de camada de transporte [Lee et al. 2017], controle de congestionamento de baixa latência [Handley et al. 2017], telemetria de rede *in-band* [Kim et al. 2015], cache rápido na rede para armazenamentos de valores-chave [Jin et al. 2017], desempenho na velocidade de rede [Dang et al. 2015], agregação para aplicações de *MapReduce* [Sapio et al. 2017], entre outros. Além disso, a linguagem P4 vem aproximando seu nível de abstração a linguagens específicas da plataforma alvo como Verilog, VHDL ou System Verilog.

Por fim, nota-se um crescimento no número de compiladores para a linguagem P4. Isso indica uma aceitação da indústria e comunidade científica e permite expandir a sua abrangência para os vários tipos de dispositivos alvos. Como exemplo de novos compiladores temos o MACSAD [Patra et al. 2017], compilador modular multi-arquitetura capaz de suportar novas plataformas DSL e de rede, e t4p4s [7], compilador *multi-target* específico para o P4.

1.7. Conclusão

No passado, as redes eram tradicionalmente rígidas com *hardwares* fixos e proprietários. Porém, nos últimos anos, a inflexibilidade às mudanças e a não escalabilidade tem se mostrado fortes limitantes nas redes atuais. Além disso, esse modelo tem um alto valor

[5] https://www.barefootnetworks.com/products/brief-tofino/

[6] http://p4fpga.github.io/

[7] https://github.com/P4ELTE/t4p4s

de CAPEX e OPEX.

Apesar dos avanços das redes definidas por software, a programabilidade do plano de dados ainda não tinha sido resolvida de forma apropriada. Algumas linguagens como Verilog e VHDL expõem a complexidade do hardware, o que dificulta a abstração e programabilidade para as camadas superiores. Nesse contexto, voltou-se a atenção para a programabilidade para o plano de dados.

Através de tecnologias como SDN e a programabilidade do plano de dados, é possível proporcionar automação, escalabilidade e flexibilidade às redes de computadores. Nesse minicurso abordamos: o protocolo OpenFlow, programação no plano de dados e principalmente a linguagem P4, incluindo ambas versões, 14 e 16. As atividades apresentadas nesse minicurso são apenas introdutórias e visam direcionar os seus participantes em alguns dos mais recentes avanços no campo de redes de computadores.

Vale ressaltar também que apesar dessas ferramentas representarem grandes avanços tecnológicos, elas ainda apresentam grandes desafios, muitos já vêm sendo estudados pela indústria e academia. Nesse aspecto, apresentamos apenas alguns dos desafios presentes nesse novo ambiente. Embora a linguagem P4 mostra-se bem recente observamos também muitos avanços, e por ter a mesma origem do SDN espera-se o mesmo sucesso.

Agradecimentos

Este minicurso foi financiado pelo Centro de Inovação da Ericsson, Brasil, projeto UNI.63 em parceira com a Universidade Estadual de Campinas (UNICAMP).

Referências

[Bernardos et al. 2015] Bernardos, C. J., Dugeon, O., Galis, A., Morris, D., and Simon, C. (2015). 5G Exchange (5GEx) – Multi-domain Orchestration for Software Defined Infrastructures.

[Bosshart et al. 2014] Bosshart, P., Varghese, G., Walker, D., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., and Vahdat, A. (2014). P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[Dang et al. 2015] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. (2015). Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA. ACM.

[Feferman and Rothenberg 2017] Feferman, D. L. and Rothenberg, C. E. (2017). Modeling P4 programmable devices using YANG. page 4.

[Freire et al. 2018] Freire, L., Neves, M., Leal, L., Levchenko, K., Schaeffer-filho, A., and Barcellos, M. (2018). Uncovering Bugs in P4 Programs with Assertion-based Verification. *Sosr '18*.

[Freire et al. 2017] Freire, L. M., Neves, M. C., Schaeffer-filho, A. E., and Barcellos, M. P. (2017). POSTER : Finding vulnerabilities in P4 programs with assertion-based verification. *Acm Ccs*, pages 2–4.

[Handley et al. 2017] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. (2017). Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 29–42, New York, NY, USA. ACM.

[Jin et al. 2017] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA. ACM.

[Kim 2016] Kim, C. (2016). Tutorial-Programming The Network Data Plane in P4.

[Kim et al. 2015] Kim, C., Sivaraman, A., Katta, N. P., Bas, A., Dixit, A., and Wobker, L. J. (2015). In-band network telemetry via programmable dataplanes.

[Kreutz et al. 2015] Kreutz, D., Ramos, F. M. V., Esteves Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S., and Uhlig, S. (2015). Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76.

[Lee et al. 2017] Lee, J., Miao, R., Kim, C., Yu, M., and Zeng, H. (2017). Stateful layer-4 load balancing in switching asics. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, pages 133–135, New York, NY, USA. ACM.

[McKeown et al. 2008a] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008a). OpenFlow. *ACM SIGCOMM Computer Communication Review*, 38(2):69.

[McKeown et al. 2008b] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008b). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

[Mejia et al. 2018] Mejia, J. S. M., Feferman, D. L., and Rothenberg, C. E. (2018). Network Address Translation using a Programmable Dataplane Processor. *In 7º Workshop em Desempenho de Sistemas Computacionais e de Comunicação (Wperformance) at XXVIII Congresso da Sociedade Brasileira de Computação - CSBC 2018, Natal-RN, Brazil, 22 a 26 de Julho*, page 4.

[Mejia and Rothenberg 2017] Mejia, J. S. M. and Rothenberg, C. E. (2017). Broadband Network Gateway implementation using a programmable data plane processor. *X EADCA Workshop at University of Campinas, October 26-27, 2017*, page 4.

[Patra et al. 2017] Patra, P. G., Rothenberg, C. E., and Pongracz, G. (2017). Macsad: High performance dataplane applications on the move. In *2017 IEEE 18th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6.

[Sapio et al. 2017] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., and Kalnis, P. (2017). In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA. ACM.

[Sousa and Rothenberg 2017] Sousa, N. F. S. D. and Rothenberg, C. E. (2017). Softwarização em Redes: Do Plano de Dados ao Plano de Orquestração. In *Anais Eletônicos ENUCOMP 2017*, chapter 1, pages 628–650. FUESPI, Parnaíba, 1º edition.