Capítulo 4

Introdução à Linguagem P4 - Teoria e Prática

Luis Fernando Uria Garcia (UFES), Rodolfo S. Villaça (UFES), Moisés R. N. Ribeiro (UFES), Regis Francisco Teles Martins (UFSCar), Fábio Luciano Verdi (UFSCar), Cesar Marcondes (UFSCar)

Abstract

The goal of this short course is to present the main concepts related to network programmability, description of the data plane and packet forwarding to network devices using P4 language. This short course approaches this topic in a theoretical and practical way, presenting in the first part the PISA (Protocol-Independent Switch Architecture) model under which the P4 language works. The P4 language components used to process packets are also presented. Before the second part of this course, it presents a description of the compiler and the sofware switch native of P4, named bmv2. With all these concepts presented, the second part consists of practical exercises, which allow a better understanding of the concepts and elements initially presented. Finally, this short course also aims to show some of the most relevant projects that make use of P4 and in this way, perceive possible applications and motivate research topics in this area.

Resumo

O objetivo deste minicurso e apresentar os principais conceitos relacionados com a programabilidade de redes, descrição do plano de dados e encaminhamento de pacotes em dispositivos de rede por meio da linguagem P4. O minicurso aborda o tema de maneira teórica e prática, apresentando na primeira parte o modelo PISA (Protocol-Independent Switch Architecture), sob o qual a linguagem funciona. Apresenta os componentes da linguagem P4 pelos quais os pacotes são processados. Antes da segunda parte, apresenta uma descrição do compilador e do software switch nativo do P4, o bmv2. Com esses elementos, a segunda parte é constituída por exercícios práticos de implementação em laboratório que permitem uma melhor compreensão dos conceitos e elementos inicialmente apresentados. Por fim, este minicurso também tem como objetivo de mostrar alguns dos projetos mais relevantes que fazem uso do P4 e desta maneira, perceber possíveis aplicações e motivar temas de pesquisa nessa área.

1.1. Introdução

À medida que as redes evoluem e oferecem suporte para mais serviços, um grande volume de fluxo de dados é gerado e precisa ser transportado por tais redes. Assim, as redes atuais possuem a exigência de dar suporte para novas aplicações e serviços de natureza dinâmica surgindo a necessidade de tornar as redes flexíveis e programáveis, trazendo algumas mudanças de paradigma no que diz respeito à estrutura dessas redes.

Até pouco tempo atrás, devido à necessidade de processamento dos pacotes de forma rápida, essas funções foram encapsuladas nos ASICs (*Application Specific Integrated Circuits*) dos dispositivos de encaminhamento de rede. Como resultado, tem-se um plano de dados fixo e consequentemente engessado que impossibilitava a construção de redes de acordo com requisitos próprios, exclusivos de cada projeto. Ao fabricar dispositivos de encaminhamento com funções fixas, a questão é: o que fazer com as funções que o dispositivo não suporta e que os usuários podem necessitar? Nesse contexto, é requisito que o projetista do plano de controle possa informar ao dispositivo de encaminhamento como processar os pacotes no plano de dados.

Assim, as Redes Definidas por Software (*Software Defined Networking*, ou SDN) que, segundo a *Open Networking Foundation* (ONF), são definidas como uma arquitetura emergente, dinâmica, gerenciável e adaptável, permitem a separação entre o controle da rede e as funções de encaminhamento. Essa separação faz com que o controle da rede seja diretamente programável e a infraestrutura subjacente possa ser abstraída para o desenvolvimento de novas aplicações e serviços.

Um elemento importante dentro do paradigma SDN é o protocolo OpenFlow [McKeown et al. 2008], que fornece uma camada de abstração da rede física para o elemento de controle, permitindo assim que o plano de dados da rede seja configurado ou manipulado através de programação via software. Na sua versão atual, o OpenFlow especifica explicitamente os cabeçalhos de protocolo nos quais ele opera. Este conjunto cresceu de 12 a 41 campos em alguns anos, aumentando a complexidade da especificação porém ainda não oferece flexibilidade suficiente para se adicionar novos cabeçalhos e definir novas ações após um *flow matching*.

A linguagem P4 ajuda a superar essas limitações do OpenFlow, aumentando a programabilidade das Redes Definidas por Software. Neste minicurso pretende-se apresentar os conceitos gerais da linguagem P4 e como a linguagem permite descrever o plano de dados e o encaminhamento em dispositivos programáveis, tais como: software *switches*, *Smart NICs*[1], FPGAs (*Field Programmable Gate Array*) e *switches bare metal*. A linguagem P4 permite que o projetista recupere o controle do plano de dados e considere funções otimizadas para uma rede específica e possui três objetivos principais: Reconfigurabilidade, Independência do Protocolo e Independência de Alvo.

- Reconfigurabilidade: está relacionada com a habilidade do programador redefinir o processamento dos pacotes nos dispositivos de rede;
- Independência do Protocolo: define que os dispositivos de rede não devem ser amarrados a formatos de pacotes específicos;

[1] https://www.netronome.com/products/smartnic/overview/.

- Independência do Alvo: relaciona-se com o fato de que o compilador da linguagem de programação de redes deve gerar uma descrição independente do dispositivo, porém levando em conta as capacidades do hardware alvo específico na hora de configurá-lo.

Com base neste contexto, vários benefícios da programabilidade do plano de dados podem ser percebidos, dentre eles:

- O controle total da programabilidade do plano de dados, de tal forma que o dispositivo de rede se comporta exatamente como desejado;
- Capacidade de adicionar novas funções, uma vez que o dispositivo é completamente programável;
- Exclusividade, porque os usuários têm a capacidade de implementar protocolos personalizados;
- Eficiência, uma vez que, considerando que os dispositivos atualmente têm capacidade limitada e funções fixas, muitos dessas não são usadas e, portanto, consomem recursos sem nenhuma finalidade;
- A confiabilidade constitui outro benefício, porque é possível não usar funções implementadas por terceiros;
- O monitoramento no plano de dados permite olhar para dentro do dispositivo. Por exemplo, é possível ver os cabeçalhos dos pacotes que estão sendo processados e exportar esses dados para obter indicadores de desempenho.

Em resumo, o minicurso se propõe a apresentar a linguagem P4 sob um ponto de vista teórico e prático. Na primeira parte do minicurso as bases conceituais da linguagem P4 e o modelo PISA (*Protocol-Independent Switch Architecture*), sob o qual a linguagem funciona, são apresentados. A segunda parte está constituída de exercícios práticos de implementação em laboratório que permitirão uma melhor compreensão dos conceitos e elementos inicialmente apresentados e, consequentemente, perceber possíveis aplicações da linguagem.

1.2. A Linguagem de Programação P4

Com base na seção anterior, que diz respeito aos benefícios da programabilidade do plano de dados, observa-se que para tornar o plano de dados programável precisamos de uma linguagem de alto nível. Por volta do ano 2013 surgiram as ideias iniciais sobre a programabilidade do plano de dados por meio de uma linguagem de domínio específico de alto nível. Assim, surgiu a denominação P4, acrônimo de *Programming Protocol-Independent Packet Processors*. A linguagem tomou forma e foi publicado o primeiro artigo no SIGCOMM em 2014 [Bosshart et al. 2014] quando surgiu a primeira versão, a $P4_{14}$, e foram lançadas as primeiras especificações oficiais. No ano de 2016 surgiu uma atualização mais relevante da linguagem, nomeada $P4_{16}$, que será trabalhada neste minicurso. O nome P4,

vem deste artigo, denominado *Programming Protocol-independent Packet Processors*, e que originou o acrônimo P4.

Pode-se observar que a especificação da linguagem possui diferentes versões, numeradas com três dígitos. Incrementos no dígito mais à direita indicam uma atualização ou adição ao último padrão. Um aumento no dígito do meio indica uma mudança maior e mais significativa. Todos as versões atuais começam com o dígito 1. Pode-se considerar o padrão 1.0.3 (lançado em 2016–11–03) como o padrão mais popular, mas o padrão 1.1.0 (lançado em 2016–01–27) e o padrão 1.2 (chamada agora $P4_{16}$) trouxeram mudanças importantes na linguagem. As principais características introduzidas em 1.1.0 foram: tipagem forte e a capacidade de indicar uma ordem na qual as ações devem ser executadas. As mesmas premissas sobre processamento dos pacotes no plano de dados foram mantidas, e tenta-se introduzir um modelo flexível visando a independência de alvos (ASICs, FPGA, NICs, software *switches*).

A linguagem P4 é uma linguagem que descreve como um pacote deve ser processado pelo plano de dados de um dispositivo programável de encaminhamento, que pode ser implementado em *switches* via hardware ou software, placas de rede, roteadores ou dispositivos programáveis do tipo FPGA. Inicialmente, a P4 foi desenhada para atuar em *switches* programáveis, no entanto, seu escopo acabou sendo estendido para cobrir uma ampla variedade de dispositivos, que no escopo da linguagem, passaram a ser chamados de "alvos"(ou *targets*).

A linguagem P4 é desenhada especificamente para descrever o plano de dados do alvo e também define a interface de comunicação com o plano de controle. No entanto, a linguagem P4 não pode ser usada para descrever as funções do plano de controle de um alvo. Por esse motivo, sempre que se refere a programar um alvo, entende-se programar o plano de dados. A Figura 1.1 exemplifica as principais distinções entre um *switch* tradicional e um *switch* programável P4.

1.2.1. Arquitetura de *Switches* Independentes de Protocolo (PISA)

Esta seção tem como objetivo apresentar a abstração do modelo de encaminhamento sobre o qual os pacotes são processados. Ou seja, por meio do modelo de encaminhamento abstrato consegue-se expressar como um dispositivo deve ser configurado e como os pacotes devem ser processados.

Esse modelo está baseado no OpenFlow e tem duas operações principais: Configurar e Preencher (*Configure and Populate*). A primeira basicamente define como o *parser* é programado, e a segunda operação adiciona entradas nas tabelas *match-action* que definem o comportamento do equipamento de rede. Na Figura 1.2 pode-se observar o *pipeline* dos pacotes iniciando-se no *parser*, passando pelas tabelas *match-action*, a execução das ações e, na última etapa, o mecanismo de processamento de filas para encaminhamento dos pacotes às portas de saída. É importante ressaltar que o reconhecedor (*parser*) pode ser programado por um modelo de máquinas de estados, conforme ilustra a figura, e que após uma ação, novos reconhecimentos podem ser programados, bem como novos *matching*, antes que um pacote seja encaminhado a uma fila de saída.

A arquitetura PISA possui os seguintes componentes principais:

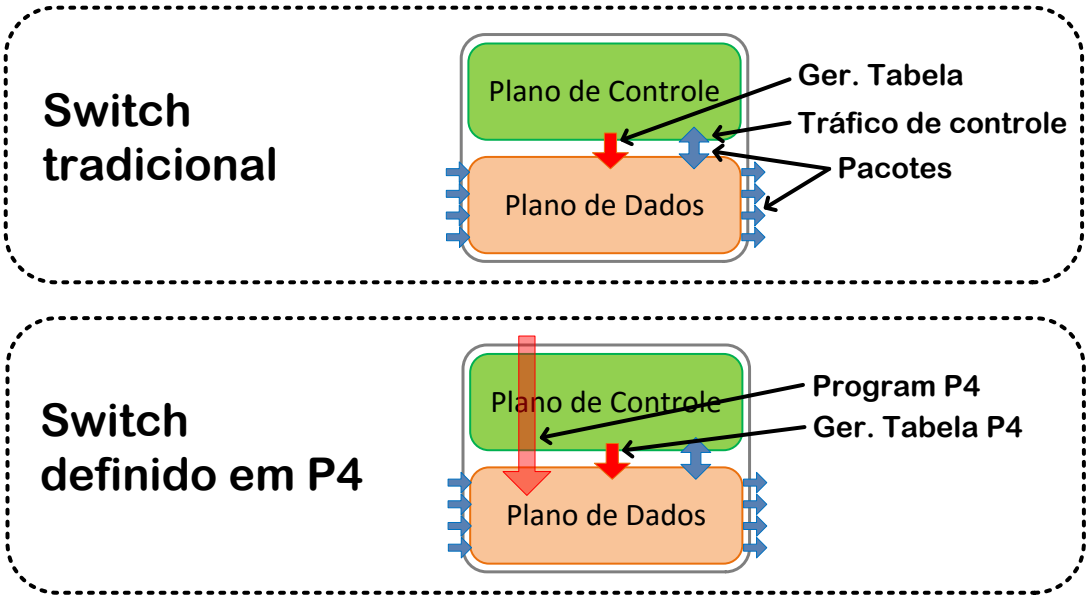

Figura 1.1. Comparação entre um *switch* tradicional e um *switch* programável P4.

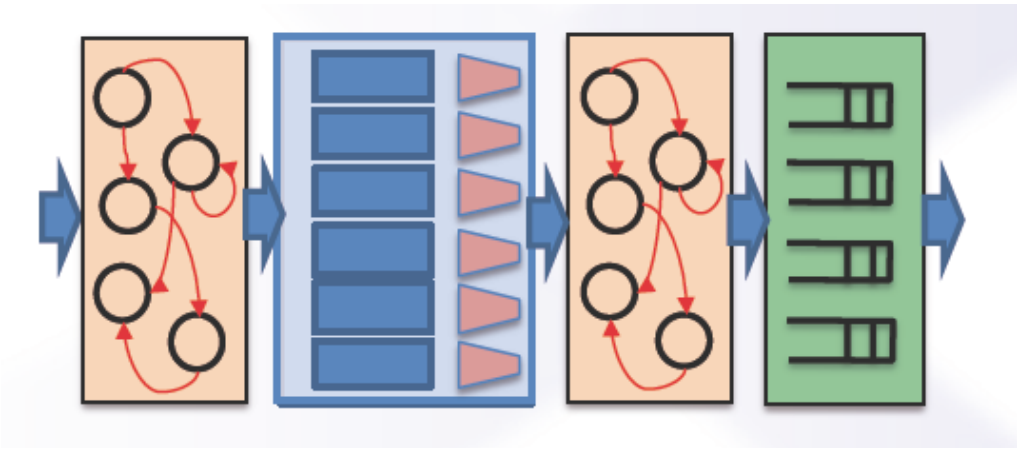

Figura 1.2. Pipeline entre os componentes da linguagem P4.

- *Parser Programável*: o *parser* tem por função identificar, em um *stream* de *bits*, os cabeçalhos que encontram-se presentes no pacote, ou seja, permite especificar o formato para processamento e identificar os protocolos contidos no pacote;
- *Match-Action*: nessa etapa faz-se um *match* e executa-se uma ação de acordo com as entradas que encontram-se nas tabelas *match-action*;
- *Deparser*: nessa etapa, monta-se de novo o pacote, para enviá-lo como um *stream* de *bits* na saída;
- Metadados: os metadados contém informações necessárias ao processamento mas que não estão no pacote, por exemplo, a identificação da porta de ingresso de um determinado pacote em um *switch*;

Pode-se observar, na Figura 1.3, o fluxo do processamento dos pacotes e as etapas do modelo PISA. O *parser* basicamente é uma máquina de estados finita que tem por função realizar a conversão do fluxo de *bits* que chega no dispositivo em um conjunto de cabeçalhos *headers*. Como exemplo, pode-se identificar e extrair primeiramente o cabeçalho Ethernet, olhar o identificador de VLAN e, na etapa seguinte do processamento no *pipeline*, os identificadores extraídos do cabeçalho identificado é repassado para as tabelas *match-action* onde para cada correspondência (*match*) executa-se uma ação (*action*) associada. Se temos pacotes L2 (*Layer 2*) pode-se fazer o *match* com o endereço MAC de destino e a ação correspondente seria encaminhar para uma porta de saída no *switch*. Em outro exemplo, se for um encaminhamento L3, com IPv4, o *match* poderia ser com o endereço IPv4 de destino e a ação poderia ser a substituição do endereço de origem pelo endereço do roteador, decrementar o TTL, e encaminhar. Cada vez que se faz uma nova ação, novos resultados serão gerados, alguns cabeçalhos podem ser modificados e portanto novos resultados intermediários podem ser produzidos exigindo novo processamento. Na última etapa haverão cabeçalhos modificados que precisam ser encaminhados à saída do dispositivo para remontagem do pacote. É nesse ponto que aparece o *deparser*, o qual monta os cabeçalhos modificados dos pacotes e os coloca nas filas de saída.

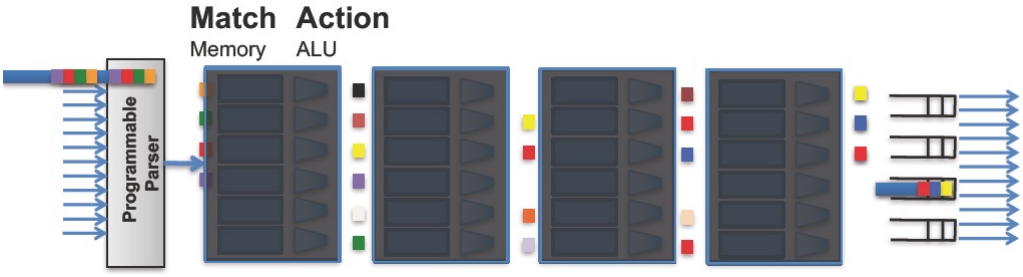

Figura 1.3. Fluxo dos pacotes dentro da Arquitetura PISA. Fonte: [Changhoon Kim 2016].

Os principais componentes da linguagem P4 associados ao modelo PISA estão descritos a seguir:

Headers: A definição do cabeçalho descreve a sequência e estrutura de uma série de campos de *bits* definidos pelo programador. Inclui especificações de largura, restrições de tipos e valores.

Parsers: A definição de *parser* especifica como identificar (reconhecer) cabeçalhos ou sequências de cabeçalho válidos nos pacotes.

Tables: As tabelas *match-action* são o mecanismo para realizar o processamento de pacotes, nas quais são associadas ações (*actions*) aos pacotes de acordo com o *matching* realizado nos *parsers*.

Actions: P4 suporta construção de ações complexas construídas usando primitivas simples e independentes de protocolo.

Em geral, o *parser* identifica os *headers* presentes em cada pacote ingressando no dispositivo. Cada tabela *match-action* realiza uma busca (*lookup*) em um subconjunto de campos de cabeçalho e aplica as ações correspondentes ao primeiro matching encontrado na tabela.

1.2.2. Elementos da Linguagem P4

Em $P4_{16}$ podem-se encontrar 5 (cinco) categorias principais de elementos da linguagem que permitem a programação do *pipeline* do plano de dados:

- Tipos de Dados: P4 tem um conjunto de tipos de dados básicos que servem para a construção de tipos mais complexos, tais como *arrays*, *headers* e *structures*;
- Expressões: constituídas por uma lista de operadores e operações básicas;
- Elementos de Fluxo de Controle: são elementos e estruturas para expressar o fluxo de dados entre as tabelas *match-action*;
- *Parsers*: elementos de programação de máquina de estados e extração de campos;
- *Externs*: conjunto de bibliotecas para suporte de componentes especializados.

1.2.2.1. *Headers* e *Structures*

O *Header* constitui o mais importante dos tipos de dados. Pode conter os tipos básicos *bit*, *int*, *varbit*. A declaração de um cabeçalho é dada pela sintaxe a seguir:

```
header HeaderName {

    fields.......

}
```

Por exemplo, pode-se definir o cabeçalho Ethernet da seguinte forma:

```
header Ethernet_h {
   bit<48> dstAddr;
   bit<48> srcAddr;
   bit<16> etherType;
}
```

A seguinte declaração de variável, ou instanciação, usa o tipo *Ethernet_h* definido anteriormente:

```
Ethernet_h ethernetHeader;
```

A *Struct* é uma estrutura composta. Nesse caso, a estrutura é formada por campos que são cabeçalhos (*headers*) definidos previamente. Um exemplo deste tipo de definição é o seguinte:

```
header Tcp_h { ... }
header Udp_h { ... }
struct Parsed_headers {
    Ethernet_h ethernet;
    Ip_h       ip;
    Tcp_h      tcp;
    Udp_h      udp;
}
```

1.2.2.2. Operadores e Expressões

P4 possui uma lista de operadores e um conjunto de expressões parecidas com outras linguagens de uso geral, como C, por exemplo. Os operadores básicos são: +, -, *, «, », &, |, ^ , &&, ||, !, <, <=, >, >=, ==, != (soma, subtração, multiplicação, deslocamento de *bits* à esquerda, deslocamento de *bits* à direita, "e"*bit* a *bit*, "ou"*bit* a *bit*, "ou exclusivo"*bit* a *bit*, negação, "e"lógico", "ou"lógico, negação e comparadores). Também suporta operadores ternários da forma: $(cond)?(true_expr) : (false_expr)$.

1.2.2.3. *Match-Actions*

No pipeline, em uma operação *match-action* tem-se 3 elementos principais: Controles, Tabelas e Ações. Em P4, os controles são muito parecidos com funções em C, sem *loops* e recursão. Os blocos do *pipeline* seguem uma ordem sequencial e vão sempre para a frente com a possibilidade de uso de declarações *if* e/ou declarações do tipo *switch*. Ou seja, o corpo do bloco de controle é executado, semelhante à execução de um programa imperativo tradicional. Sintaticamente, um bloco de controle é declarado com um nome, parâmetros obrigatórios, opcionais e uma sequência de declarações de constantes, variáveis, ações, tabelas e outras instâncias. Dentro de um bloco de controle, utiliza-se a sentença *apply(nome_tabela)* para invocar as ações correspondentes declaradas na tabela.

```
control ingress {
    apply(nome_tabela);
}

control egress {
}
```

Parsers são funções especiais escritas em um estilo de máquina de estados finita que basicamente buscam converter ou mapear fluxos de *bits* em cabeçalhos. Têm uma sequencia de sentenças que representam um conjunto de estados e as regras de como faz-se a transição entre eles. Todo *parser* tem três estados predefinidos. O primeiro é o estado *Start*, que é onde inicia-se o processo. O outro estado é o *Accept*, o qual é usado

para indicar que o pacote é aceito. O terceiro estado é o *Reject*, o qual é onde envia-se aqueles pacotes que não se quer tratar. A Figura 1.4 mostra a estrutura geral da máquina de estados do *parser*.

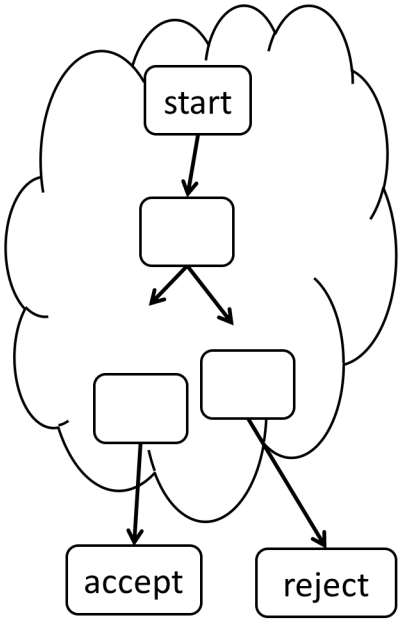

Figura 1.4. Estados predefinidos de um *parser*. Fonte: [Consortium 2017].

Os estados definidos pelo programador descrevem como o pacote será processado pelo *parser*. Como exemplo de declaração de uma estrutura de um *parser* pode-se observar a seguinte estruturação, que será detalhada na próxima seção.

```
parser start {
    return parse_ethernet;
}

parser parse_ethernet {
    extract(ethernet);
    return ingress;
}
```

Os objetos externos (*Externs*) são construções específicas da arquitetura que podem ser manipuladas por programas P4 através de APIs bem definidas, mas cujo comportamento interno é *hard-wired* (por exemplo, unidades de *checksum*) e, portanto, não programáveis usando P4.

1.2.3. Seções de um Programa P4

Um programa em P4 pode-se dividir em 3 principais seções, de acordo com o *pipeline* da Arquitetura PISA que os pacotes atravessam. Com base na seção anterior, onde apresentaram-se os elementos centrais da linguagem, a seguir descreve-se as seções de um programa em P4 procurando agrupar aqueles elementos em exemplos de declarações de cada seção.

1.2.3.1. Declaração de Dados

Basicamente, um programa P4 começa com a definição do cabeçalho. Por isso, o tipo de dado mais importante é o tipo *Header*. Na seção sobre elementos da linguagem já mostramos como fazer a declaração dos cabeçalhos. Com base nos tutoriais que encontram-se no repositório oficial da linguagem P4, especificamente o exercício chamado `basic.p4`, apresenta-se como exemplo a definição de um quadro Ethernet, onde tem-se especificado os campos endereço destino, endereço de origem e *ethertype*. Assim também define-se um cabeçalho IPv4.

```
typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}
```

1.2.3.2. *Parser* Programável

Uma vez definidos os *headers*, precisa-se expressar a lógica do *parser*, a mesma que será usada para identificar a transição da máquina de estados finita. Como exemplo, pode-se ver uma sequencia de estados simples, desde o estado inicial, para extrair os cabeçalhos Ethernet e IPv4 em um pacote.

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
```

```
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}
```

1.2.3.3. Tabelas

Nas tabelas, são encontradas as chaves com as quais procura-se fazer o *matching* assim como as ações associadas. As ações são basicamente funções compostas por ações primitivas. Como exemplo, pode-se observar a seguir, o código para uma tabela IPv4, onde o campo para realizar o *matching* é o endereço IP de destino sob o algoritmo de *Longest Prefix Matching* e as ações possíveis são *setnhop* (definir o próximo salto) e *drop* (descarte).

```
table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
}
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

Em outro exemplo, o bloco de controle a seguir considera a condição de verificação de validade do cabeçalho para aplicar a tabela *ipv4_lpm* por meio da sentença *apply*:

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
inout standard\_metadata\_t standard\_metadata) {
apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
```

```
}
```

1.3. Compilando Programas na Linguagem P4

Esta seção apresenta uma introdução geral sobre o processo de compilação de programas na linguagem P4. Começamos destacando que a linguagem P4 é fruto de ampla discussão e trabalho colaborativo da comunidade de redes, trabalhando pela evolução da programabilidade do plano de dados. Nessa comunidade existe um processo de criação e refinamento do P4, onde a cada 2 anos são preparadas novas **especificações** da linguagem e da **implementação** de compilador de referência. Esses dois recursos, especificação e compilador de referência se inter-relacionam, e permitem a melhoria consecutiva de ambos, ao longo do tempo. A versão corrente da linguagem P4, considerada estável, é a **P4_16** que conta com um esforço colaborativo de dezenas de desenvolvedores, licença *open source* Apache, e milhares de linhas de código e teste. Dessa forma, destacamos que o P4 é um projeto maduro que pode inclusive ser usado em alguns cenários em modo de produção.

Atrelada à linguagem, o desenvolvimento de um compilador para a linguagem P4, traz por si, uma série de desafios de projeto. Por exemplo, a compilação de um programa P4 precisa possuir suporte a versões mais antigas da linguagem, garantindo retro-compatibilidade, bem como permitir uma extensibilidade flexível para suportar versões futuras com diferentes sintaxes. Além dos desafios de análise sintática e semântica do código P4, ele precisa também, dentro da arquitetura geral, suportar um modelo de abstração *frontend* único e estável. Esse *frontend* deve ser de código aberto e não deve ser constantemente modificado por desenvolvedores.

Para ilustrar o fluxo de trabalho do compilador P4 desde a entrada do programa ".p4"até a geração de código específico do alvo, como o Verilog de código P4, ou mesmo código C, detalharemos a Figura 1.5. O primeiro trecho do fluxo do compilador é expressar o código P4 em um formato intermediário **único** (IR). Para tanto, é feito o uso de uma gramática que está especificada nos documentos de padronização, e permite fazer o segmentação precisa dos construtores da linguagem P4 (P4_16). Para manter a compatibilidade com versões antigas, o compilador possui dois segmentadores (*parsers*) baseados em duas gramáticas ligeiramente diferentes e ambas devem ser convertidas para uma mesma representação intermediária comum "segmentada"(primeira referência de IR), antes de serem processadas pelo *frontend*.

O primeiro código IR, com as transformações iniciais, é então processado pelo *frontend* e passa por um *pipeline* de processamento de transformação. Por exemplo, no P4_16 o *frontend* introduz mais de 25 passos de transformação sintática e semântica, de modo que seja colocado em um formato de uma plataforma abstrata P4 (expressa em classes de linguagem de C++) e portanto independente do hardware e software de baixo nível, gerando assim o segundo IR que aparece na Figura 1.5.

Esse IR genérico pode então passar por um estágio de transformação de baixo nível. Esse estágio já dependente da arquitetura, e focado para a preparação do processamento *midend* e depois *backend*. A diferença do *midend* e do *backend* é somente para fins de reaproveitamento de código em plataformas específicas que podem ser muito pró-

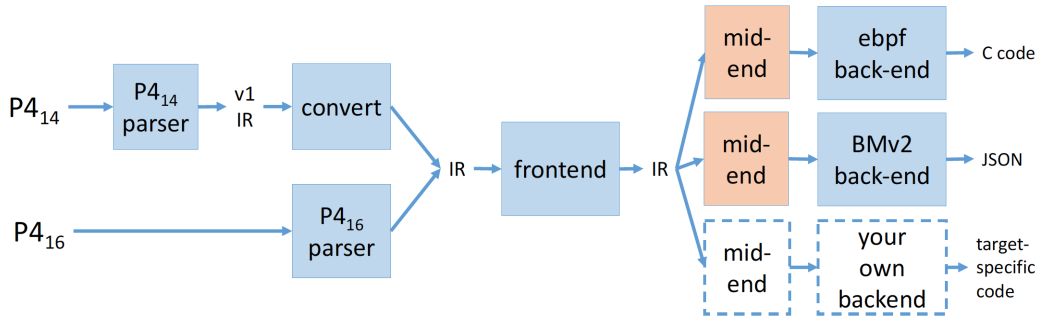

Figura 1.5. Fluxo do Compilador P4. Fonte: [Changhoon Kim 2016].

ximas. Finalizando o fluxo do compilador, é no *backend* que o compilador deverá suportar uma ampla variedade de arquiteturas **alvo** diferentes. Desse modo, o *backend* aumenta o ecossistema de plataformas específicas que suportam a possibilidade de reprogramação do plano de dados via P4. Essas plataformas específicas podem ser desde *chips* integrados (ASIC) baseados em PISA configuráveis como a plataforma Tofino[2] que tem capacidade de processar Terabits/s, passando por FPGA[3] com seu modelo de referência NetFPGA-P4. Finalmente, até mesmo a conversão do programa P4 em plataformas puramente de software, como código binário de filtragem baseado no *kernel* eBPF (*extended Berkeley Packet Filter*)[4] e código de *openvswitch* instrumentado e parametrizado para suportar a arquitetura PISA, como o projeto PISCES[5]. Esse poder de suportar uma variedade de plataformas é um objetivo importante, pois o compilador começa abstraindo o suficiente o processamento do plano de dados, e depois deixa os pontos específicos da plataforma real para esse estágio final de *backend* da compilação.

Dentre as plataformas usadas para testes e simulação para criar provas de conceito está o BMv2, que é a implementação do *"Behavioral Model"* da arquitetura PISA, e esse por sua vez implementa um modelo simples de *switch* no *backend* específico do compilador: *p4c-bm*. Uma vez feita a compilação do programa P4, o *backend* será o responsável pela programação ou configuração do plano de dados apropriado. Por exemplo, no caso do BMv2, o programa P4 será transformado em um descritivo JSON como saída do *backend*, conforme Figura 1.5. O arquivo JSON, por sua vez, será a entrada para o *switch* BMv2 que fará na entrada a segmentação dos pacotes, seguindo a orientação do que estiver especificado no arquivo de configuração JSON, e depois populará as tabelas do pipeline do *switch* PISA de referência. Todo esse processo é específico do BMv2. Na próxima sub-seção, apresentaremos o passo a passo de como compilar um programa P4 usando o BMv2.

Para outros tipos de *backend*, tais como as FPGAs, a saída final pode ser a geração de *soft cores* de FPGA, que são, em última instância, representados em linguagem Verilog ou VHDL e que permitem refazer o *pipeline* de baixo nível PISA, como é o caso do projeto NetFPGA-P4. Para outros *backend* baseados em *software* como o eBPF, a saída do

[2]https://barefootnetworks.com/products/brief-tofino/.
[3]https://github.com/NetFPGA/P4-NetFPGA-public.
[4]http://prototype-kernel.readthedocs.io/en/latest/bpf/.
[5]http://pisces.cs.princeton.edu/.

backend produz o equivalente a um *bytecode* de baixo nível que instrui a criação de regras de filtragem, de alto desempenho, dentro da pilha de redes do sistema operacional Linux. Após a instrução passo-a-passo de como compilar usando o *backend* BMv2, voltaremos em sub-seção subsequente a discutir detalhes de outro *backend*, o NetFPGA-P4, onde detalharemos o seu funcionamento interno.

1.3.1. Passo a Passo na Compilação de Programa P4 no *backend* BMv2

Conforme vimos no começo dessa seção, para implementar um programa em P4, precisa-se mapear a descrição de como serão processados os pacotes (em programa ".p4") em um dispositivo alvo (*target*) específico, seja hardware ou software. Esse é basicamente o trabalho do compilador.

O fluxo de etapas mostrado na Figura 1.6 representa uma configuração de ferramentas que permitem implementar programas escritos em P4 em um dispositivo alvo. Dentre essas ferramentas temos o compilador `p4c` e o *target* que nesse caso é o software *switch* nativo de referência, denominado BMv2. Nessa representação, pode-se observar que o programa escrito com código P4 é processado pelo `p4c` (compilador P4), para em seguida carregar a representação comportamental gerada pelo compilador no BMv2 software switch. É dessa forma que o dispositivo processará os pacotes no plano de dados da maneira desejada e descrita no programa. Além disso, pode-se observar como o plano de controle poderia interagir com o dispositivo *target* em tempo de execução por meio do *framework* chamado P4Runtime. No caso do BMv2, tem-se disponível uma ferramenta de tipo CLI (*Command Line Interface*) para preencher as tabelas do dispositivo alvo.

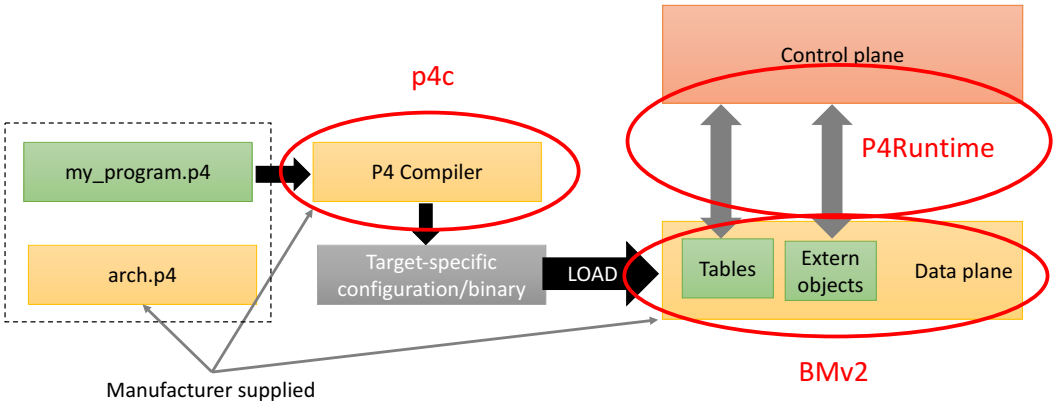

Figura 1.6. Fluxo para implementar um programa P4. Fonte: [Carmelo Cascone 2016].

1.3.2. Compilador P4C

Em um breve histórico, o `p4c-behavioral` foi a primeiro compilador P4 e era conhecido também como BMv1. Esse compilador gerava um código C para cada programa P4, que seria compilado para um executável usando o `gcc`, ou seja, para cada programa P4 precisava-se um novo executável.

Assim, mudou-se para o novo compilador *Behavioral Model version 2* (BMv2). O fluxo com esse novo compilador é simples: o programa P4 é primeiramente compi-

lado para uma representação JSON (*JavaScript Object Notation*) por meio do módulo chamado `p4c-bm`. Este JSON é então carregado no BMv2 e as estruturas de dados são inicializadas para refletir o comportamento de encaminhamento desejado.

Para executar seus próprios programas P4 no BMv2, primeiramente você precisa transformar o código P4 em uma representação JSON que pode ser consumida pelo software *switch*. Esta representação dirá ao BMv2 quais tabelas inicializar e como configurar o *parser*. Pode-se gerar o arquivo JSON a partir de um programa P4 da seguinte forma:

```
p4c-bm -json <path to JSON file> <path to P4 file>
```

O arquivo JSON agora pode ser usado como entrada para o *switch* BMv2. Como exemplo, para realizar esta ação usando um *simple_switch* como alvo a seguinte sequência de comandos é necessária:

```
sudo ./simple_switch -i 0@<iface0> -i 1@<iface1> <path
to JSON file>
```

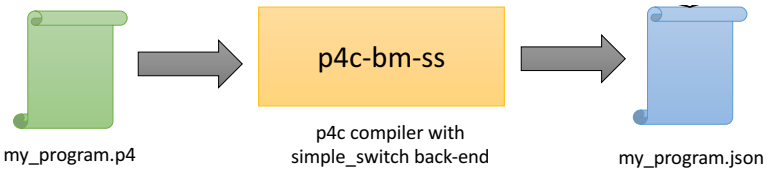

Figura 1.7. Fluxo de compilação. Fonte: [Carmelo Cascone 2016].

A última versão do compilador `p4c` aceita compilar programas tanto em $P4_{14}$ quanto $P4_{16}$, podendo então ser usado no lugar do `p4c-bm`. Com o `p4c-bm2-ss` compila-se diretamente para o alvo *simple_switch*, que é escrito usando o Behavioral Model (Ver Figura 1.7).

Desta forma, para compilarmos programas em P4_16 para o *simple_switch* (target BMv2) utiliza-se a seguinte linha de comando:

```
p4c-bmv2-ss -o program.json program.bmv2.p4
```

1.3.3. Detalhamento do *Backend* NetFPGA-P4

A plataforma de experimentação NetFPGA é uma das mais difundidas plataformas para ensino e pesquisa de novos projetos de dispositivos de rede. Com ela é possível projetar um equipamento de rede: a) completamente a partir de FPGA puro e; b) reutilizando módulos de alta qualidade escritos em projetos de referência. A plataforma possui diversos modelos de placas comercializados para fins acadêmicos, de um espectro de opções de placas com interfaces de 1G até 10G. Da família de placas NetFPGA, a mais sofisticada é a NetFPGA SUME com chip Xilinx Virtex-7 com capacidade de 690K células lógicas e capacidade de suportar projetos de grande porte, bancos de memória DRAM DDR3 SoDIMM com 4GB e SRAM QDRII+, 4 transceptores SFP+ com suporte a cabo direto e interfaces seriais de expensão QTH-DP de 8 x 12.5Gbps, e FMC HPC de 10 x 12.5 Gbps.

Nessa família de placas, para suportar a possibilidade de programação de projetos de alto desempenho usando a linguagem P4, foi feita uma parceria entre a empresa Xilinx e os gestores do projeto NetFPGA (Universidade de Stanford e Universidade de

Cambridge - UK) para a liberação de um *backend* apropriado para a NetFPGA SUME, o projeto P4-netFPGA [Sultana et al. 2017]. Além do *backend*, a comunidade e outros entusiastas de NetFPGA produziram parte do compilador P4 que suporta o fluxo de compilação para a NetFPGA.

A Figura 1.8 apresenta o fluxo de programação do plano de dados da NetFPGA à partir do programa P4. O programa p4 é observado no quadro à esquerda superior e ele é baseado em uma arquitetura de referência P4 para a NetFPGA, representada logo abaixo na mesma figura. Observe que a arquitetura de referência trata-se da arquitetura **SimpleSwitch**, muito parecida com a arquitetura vista com o BMv2, em termos de *pipelines Match* e *Actions*. Essa arquitetura **SimpleSwitch** é a forma abstrata de apresentar um modelo de *switch* compatível com PISA. Continuando no fluxo de programação, temos o compilador P4 que produz como saída: uma instanciação *runtime* do plano de dados e simultaneamente uma API (*Application Program Interface*) que permite a comunicação com registradores e configurações internas do hardware.

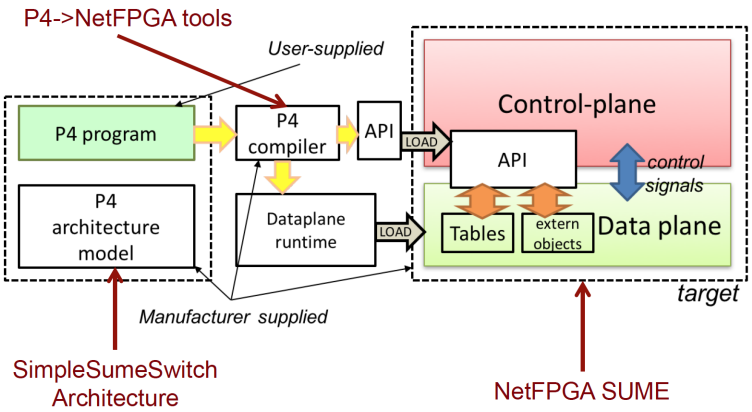

Figura 1.8. Fluxo de Programação P4 para o Alvo da placa netFPGA. Fonte: [Ibanez 2017].

Ambas essas saídas são carregadas na plataforma netFPGA para execução. O *runtime* é sintetizado, e o *bitstream* é carregado no chip Xilinx Virtex-7, enquanto que a API é carregada como um conjunto de *drivers*, programas em C e interfaces que permitem a posterior configuração do plano de dados através da Tabela *Match* e *Actions*. Também é carregado na plataforma *runtime*, código adicional em Verilog para algum tipo de processamento especial, acessível pela interface chamada *extern objects*.

O *extern objects* é a maneira pela qual a plataforma P4 pode ser estendida se o alvo permitir mudanças e inclusão de funções que não existem na arquitetura PISA, como por exemplo, funções *hashing* específicas, CRC (*Cyclic Redundance Checking*) ou outras que necessitam de processamento especial por pacote. Dessa forma, o fluxo na camada do plano de dados pode ser incrementado com processamento de funções matemáticas ou outros processos computacionais que seriam implementados na plataforma alvo. A plataforma alvo da NetFPGA é baseada em código Verilog, portanto, a adição de código FPGA específico para casos particulares podem ser feitos, configurados e manuseados diretamente na plataforma.

Outro destaque na Figura 1.8 são o conjunto de setas que apontam o material su-

portado pelo fabricante (*Manufacturer Supplied*), ou seja, o projeto tem partes em código aberto e partes que são ferramentas proprietárias da Xilinx. A figura a seguir, Figura 1.9, apresenta a cadeia de ferramentas para o processo de compilar alvos P4 em NetFPGA. Começando no topo, o programa P4 faz uso do compilador com partes proprietárias Xilinx P4_16. Essa ferramenta pode ser obtida por meio de solicitação diretamente para a Xilinx.

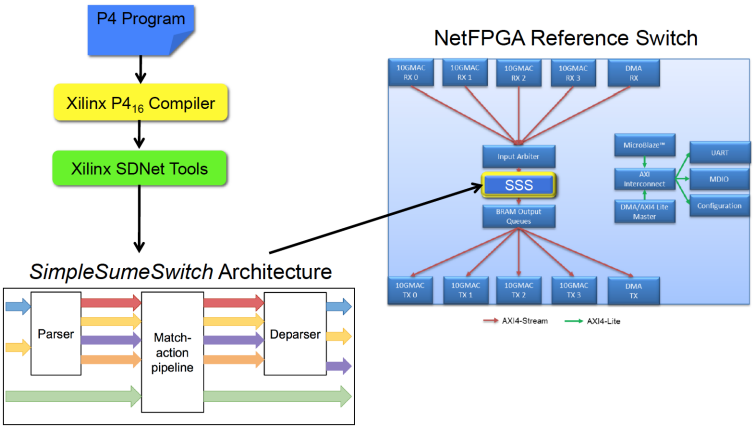

Figura 1.9. Visão Geral das Ferramentas Abertas e Proprietárias usados no P4-NetFPGA. Fonte: [Ibanez 2017].

Em seguida, a próxima ferramenta Xilinx SDNet Tools é um conjunto de *softcores* e bibliotecas proprietárias que são específicas para Redes Definidas por Software e que permitem a implementação eficiente de cadeias de *match* e *actions* como as usadas em OpenFlow e outras tecnologias. O resultado da aplicação das ferramentas em sequencia permite especificar em Verilog, com detalhes, a arquitetura *SimpleSumeSwitch* (SSS) que basicamente contém os elementos da arquitetura PISA de segmentação (*parsing*), o conjunto de pipeline de ações de *match* e *action* e o serializador final (*deparsing*).

Esse elemento da arquitetura PISA então é embarcado (lado direito da Figura 1.9) no **NetFPGA Reference Switch**. Em outras palavras, todo o entorno do núcleo SSS reaproveita a implementação do *switch* de referência em NetFPGA, que foi escrito em Verilog. Com isso se reaproveita o código que permite lidar com a sincronização das filas TX e RX, o árbitro de entrada que detecta se um pacote está sendo enviado pela mesma porta de entrada e saída, o gerenciador de filas de saída que tem um mecanismo de controle de *buffers* em DRAM. Além disso, registradores também ficam expostos, e *soft* microprocessador MicroBlaze estão automaticamente integrados no projeto.

Outros detalhes dessa implementação P4-FPGA podem ser obtidos em [Sultana et al. 2017] e [Wang et al. 2017]. Por exemplo, para o aproveitamento completo de um desenvolvedor nesta plataforma é preciso ter noção dos aspectos específicos da linguagem P4, tais como: os metadados na arquitetura SSS que estão organizados na estrutura $sume_metadata_t$. Também é preciso ter uma ideia do mapeamento dos metadados específicos da NetFPGA com os metadados nativos da linguagem P4. Outro aspecto bastante relevante é o uso das funções externas, que podem ser implementação caixa-preta de códigos em Verilog para melhorar a funcionalidade de um projeto. Essas funções externas permitem especificar coisas como variáveis de estado, e variáveis atômicas, bem como

funções de medição de latência feitas pela Xilinx.

O passo a passo da execução final de um projeto P4-NetFPGA passa pelos seguintes passos:

- Escrever um programa P4;
- Com o código do P4-NetFPGA baixado, executar o *script* `gen_testdata.py`;
- Na pasta do projeto P4-netFPGA compilar o programa P4 e gerar o código Verilog;
- Gerar as ferramentas de API e CLI, ambas com a de cima, com um único `make`;
- Executar o simulador SDNet com o comando `vivado_sim.bash`;
- Executar o simulador NetFPGA com o comando `./nf_test sim –major switch –minor default`;
- Sintetizar o projeto e criar o bitstream com o comando `make`;
- Testar seu novo hardware.

Com isso, encerramos o detalhamento do processo interno de compilação de programas P4 e sua aderência para plataformas alvo variadas como BMv2 e NetFPGA, que permitem a construção de provas de conceito altamente sofisticadas. A seguir, seguiremos para os laboratórios P4 usando BMv2.

1.4. Laboratórios

A fim de entender melhor como funciona uma rede formada por dispositivos de encaminhamento programáveis em linguagem P4, este minicurso conta com uma parte prática onde os participantes, utilizando-se de uma máquina virtual fornecida e instalada previamente, poderão criar, compilar e implementar programas em linguagem P4.

A parte prática, ou de laboratório, conta com três atividades propostas, onde todo o ambiente já foi previamente configurado e disponibilizado[6], cabendo ao participante complementar o código P4 de forma a cumprir com o objetivo de cada atividade.

1.4.1. Atividade 1 - Encaminhamento de Pacotes

O objetivo desta atividade é simplesmente encaminhar um pacote recebido de um *host* para outro. O diagrama apresentado na Figura 1.10 demonstra a topologia configurada para essa tarefa.

Para simplificar ainda mais, deve ser implementado somente a lógica de encaminhamento para o protocolo IPv4. Para isso, o elemento de encaminhamento ou *switch*, deverá desempenhar as seguintes ações para cada pacote recebido:

- Atualização do endereço MAC de origem e destino;

[6]https://www.dropbox.com/sh/r29oa0abzluj9n2/AABFsTTVZrh1yQ8WFK01_HRna?dl=0

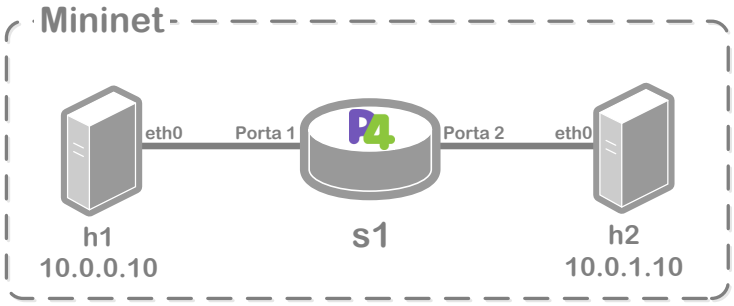

Figura 1.10. Topologia proposta para a Atividade 1.

- O decremento do valor do TTL (*Time To Live*) do pacote;
- Encaminhamento do pacote para a porta de saída apropriada.

O *switch* P4 contará com uma única tabela, a qual será populada com regras estáticas implementadas pelo plano de controle usando CLI. Cada uma das regras mapeia um endereço IP para o endereço MAC e a porta de saída para o próximo salto. Essas regras já foram previamente implementadas, então será necessário somente a implementação da lógica do plano de dados através do código P4.

1.4.1.1. Passo 1: Execução Inicial

O diretório `/home/student/labs/task_1/p4src` contém o código P4 `task_1.p4`, que pode ser compilado. No entanto, em seu estado atual ele simplesmente descarta todos os pacotes que chegam até o dispositivo. O objetivo dessa atividade é completar esse código inicial de forma que o *switch* P4 faça o encaminhamento do pacote ao *host* correto.

Primeiramente, compile o código de forma que está e execute o Mininet para testar o seu funcionamento.

Na linha de comando, dentro do diretório `/home/student/labs/task_1/` execute o *script*: `sudo ./run_task.sh`. Esse *script* irá compilar o programa `task_1.p4` e inicializar o Mininet com a topologia configurada para a atividade.

A instância do Mininet para essa atividade conta com um *switch* ($s1$) e dois *hosts* ($h1$ e $h2$). Os *hosts* possuem os seguintes endereços IPs associados: $h1$ - 10.0.0.10 e $h2$ - 10.0.1.10.

Uma vez que a instância do Mininet esteja carregada, obtem-se o *prompt* >. Abra então terminais para $h1$ e $h2$, respectivamente:

```
mininet> xterm h1 h2
```

Em cada um dos *hosts* existem aplicações baseadas em Python para enviar e receber mensagens. Inicie primeiramente a aplicação para receber mensagens no terminal do *host* 2 ($h2$).

```
sudo ./receive.py
```

Em seguida, inicie a aplicação para enviar mensagem a partir do *host* 1 ($h1$). A sintaxe do comando é `send.py <ip do receiver> <"mensagem">`:

```
sudo ./send.py 10.0.1.10 "Olá mundo P4!"
```

A mensagem não será recebida.

Saia do emulador digitando `exit` na linha de comando.

```
mininet> exit
```

A mensagem não foi entregue porque o *switch* está programado com o código P4 em `task_1.p4` original, que descarta os pacotes recebidos. O objetivo dessa tarefa é estender o código P4 para que os pacotes consigam alcançar o seu destino.

1.4.1.2. Plano de Controle

O programa em P4 define o *pipeline* de processamento dos pacotes, mas as regras em cada uma das tabelas são inseridas através do plano de controle. Quando uma regra encontra um pacote correspondente, sua ação é invocada com os parâmetros fornecidos pelo plano de controle.

Nessa atividade, a lógica do plano de controle já foi implementada. Assim que o Mininet é instanciado, as regras para o processamento dos pacotes são instaladas nas tabelas do *switch* P4. Essas regras estão definidas no arquivo `command_s1.txt`.

O código P4 também define a interface entre o *pipeline* do *switch* e o plano de controle. Os comandos CLI no arquivo `command_s1.txt` referem-se a tabelas, chaves e ações, especificadas pelo nome e qualquer alteração que ocorra no programa P4 que adicione ou renomeie uma tabela, chave ou ação, precisa ser refletida nestes comandos.

1.4.1.3. Passo 2: Implementação

O arquivo `task_1.p4` contém o código P4, com as principais partes da lógica a ser implementada, assinaladas com o comentários **PARA COMPLETAR**. A implementação a ser feita deverá seguir a estrutura proposta nesse arquivo, substituindo os comentários pelo código P4 necessário para que a implementação funcione.

O arquivo `task_1.p4` completo deve ter os seguintes componentes:

1. Definições de cabeçalho para as camadas *ethernet* (`ethernet_t`) e IPv4 (`ipv4_t`).
2. **PARA COMPLETAR:** Os *parsers* para *ethernet* e IPv4 que populem os campos de `ethernet_t` e `ipv4_t`.
3. Pelo menos uma ação para descartar os pacotes não reconhecidos usando `mark_to_drop()`.

4. **PARA COMPLETAR:** Uma ação (chamada `ipv4_forward`) que:
   (a) Determina a porta de saída para o próximo salto;
   (b) Atualiza o endereço *ethernet* de destino com o endereço do próximo salto;
   (c) Atualiza o endereço *ethernet* de origem, com o endereço *ethernet* do *switch*;
   (d) Decrementa o TTL.
5. **PARA COMPLETAR:** Um controle que:
   (a) Defina a tabela para ler o endereço IPv4 de destino e invoque a ação `drop` ou a ação `ipv4_forward`;
   (b) Defina um bloco `apply` que referencie a tabela.
6. **PARA COMPLETAR:** Defina um `deparser` que selecione a ordem que cada campo deve ser inserido no pacote que está saindo do *switch*;
7. Um instanciação do pacote composta por `parser`, `control` e `deparser`.

Eventualmente esse processamento também requer instâncias de verificação de *checksum* e controles. Isso não será necessário para essa tarefa e foram substituídas por controles vazios.

1.4.1.4. Passo 3: Execução Final

Siga as instruções do Passo 1 para testar. Após as alterações de código as mensagens do *host h*1 deverão ser entregues ao *host h*2.

Reflexões e Desafios

Os programas utilizados para testar a sua implementação não são muito robustos e não suportam o envio de um tráfego mais elevado. Qual seria outra forma de testar a implementação? Além disso, outras questões podem ser consideradas: Como você poderia melhorar o seu código para suportar vários saltos? Essa implementação é suficiente para substituir um roteador comercial? O que estaria faltando?

Solução de Problemas

Existem vários problemas que podem acontecer durante o desenvolvimento da sua implementação:

1. O código `task_1.p4` pode falhar durante a compilação. Nesse caso o *script* `run_task_1.sh` apresentará na tela o erro de compilação e será terminado.
2. O código `task_1.p4` pode compilar, mas pode falhar ao carregar as regras do plano de controle contidas no arquivo `command_s1.txt`, que o *script* `run_task.sh` tentará instalar usando o CLI BMv2. Neste caso, o `run_task.sh` irá gerar mensagens de erro sobre a saída do CLI no diretório de *logs*. Utilize essa mensagem de erro para consertar a sua implementação.

3. O código `task_1.p4` pode compilar e as regras do plano de controle descritas no arquivo `command_s1.txt` podem ser carregadas sem problemas, mas ainda assim, o *switch* P4 pode não conseguir processar os pacotes corretamente. O arquivo `/tmp/p4s.s1.log` contém informações detalhadas descrevendo como o pacote foi processado pelo *switch*. Este *log* pode ajudar a encontrar erros de lógica na sua implementação.

Reiniciando o Mininet

Alguns casos de falha podem ser causados devido a inicialização do ambiente. Nestes casos, reiniciar o Mininet pode ajudar. Utilize o comando `mn -c` para terminar as instâncias que ficaram travadas.

1.4.2. Atividade 2 - Implementando uma Calculadora em P4

O objetivo desta atividade é implementar uma calculadora básica utilizando um cabeçalho customizado escrito em P4. O cabeçalho deve conter a operação a ser executada e dois operandos. Quando o *switch* receber o cabeçalho para calcular, ele deverá executar a operação com os operandos e retornar ao endereço de origem o resultado. Muito embora esta atividade pode não possuir uma aplicação efetivamente prática, ela é muito útil para demonstrar o potencial da linguagem P4 para definir novos cabeçalhos. O diagrama apresentado na Figura 1.11 demonstra a topologia configurada para essa tarefa.

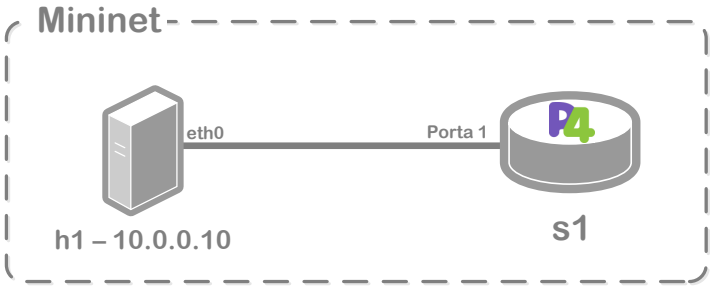

Figura 1.11. Topologia proposta para a atividade 2.

1.4.2.1. Passo 1: Execução Inicial

O diretório `/home/student/labs/task_2/p4src` contém o código P4 `task_2.p4`, e pode ser compilado. No entanto, em seu estado atual ele simplesmente descarta os pacotes que chegam até o *switch*. O objetivo dessa atividade é completar esse código inicial de forma que o *switch* P4 faça o cálculo de acordo com as informações recebidas no cabeçalho e devolva o pacote à sua origem.

Primeiramente compile o código da forma que se encontra e execute o Mininet para testar o seu funcionamento.

Na linha de comando, dentro do diretório `/home/student/labs/task_2/` execute o *script*: `sudo ./run_task.sh`. Esse *script* irá compilar o programa

`task_2.p4` e inicializar o Mininet já com a topologia configurada para a atividade.

A instância do Mininet para essa atividade conta com um *switch* (*s*1) e um *host* (*h*1). O *host* tem o seguintes endereço IP associado: h1 - 10.0.0.10

Para essa atividade um pequeno programa em Python foi escrito para testar o funcionamento da sua calculadora. O programa será carregado no *host h*1 diretamente no *prompt* e permitirá a execução dos testes. O *script* proverá um novo *prompt*, no qual você poderá escrever expressões matemáticas básicas. Essa aplicação vai verificar a expressão e preparar um pacote com o operador e os operandos correspondentes, e então, enviará esse pacote para o *switch* P4. Quando o *switch* retornar o pacote com o resultado da operação, a aplicação imprimirá na tela o valor. Uma vez que a lógica em P4 ainda não foi implementada, você deve ver uma mensagem de erro na tela, conforme exemplo a seguir:

```
     > 1+1
Didn't receive response
>
```

1.4.2.2. Passo 2: Implementação

Para a implementação da calculadora, será necessário definir um cabeçalho customizado (Figura 1.12 e definir a lógica no *switch* P4 para verificar o cabeçalho, efetuar a operação indicada, escrever o resultado no cabeçalho e retornar o pacote ao seu emissor. O seguinte formato de cabeçalho foi utilizado para essa atividade:

- P é o caracter ASCII 'P' (0x50)
- 4 é o caracter ASCII '4' (0x34)
- Versão é atualmente a 0.1 (0x01)
- Op é a operação a ser efetuada:
- '+' (0x2b) Resultado = Operando A + Operando B
- '-' (0x2d) Resultado = Operando A - Operando B
- '&' (0x26) Resultado = Operando A & Operando B
- '|' (0x7c) Resultado = Operando A | Operando B
- '^' (0x5e) Resultado = Operando A ^ Operando B

Assume-se que o cabeçalho a ser calculado será carregado sobre o protocolo *ethernet*, e que será usado o tipo de *ethernet* 0x1234 para indicar a presença desse cabeçalho. Dado o que foi aprendido até agora, o objetivo é implementar o código P4 para fazer os cálculos. Não existe lógica do plano de controle, então só é necessário a implementação da lógica para o plano de dados.

Figura 1.12. Cabeçalho customizado para a calculadora P4.

Uma implementação correta deverá verificar os cabeçalhos customizados, executar as operações matemáticas, escrever o resultado no campo apropriado e retornar o pacote ao emissor do mesmo.

1.4.2.3. Passo 3: Execução Final

Siga as instruções do Passo 1. Agora, ao invés de uma mensagem de erro, deverá aparecer o resultado correto da operação.

```
> 1+1
2
>
```

1.4.3. Atividade 3 - Implementação Inicial

O objetivo desta atividade é implementar um esquema de roteamento estabelecido na origem do pacote. Com o roteamento na origem, o *host* de origem especifica para qual porta cada *switch* na rede deverá encaminhar o pacote em cada salto. O *host* de origem estabelece uma série de portas de saída no cabeçalhos do pacote. Neste exemplo, será introduzida a lista das portas para encaminhamento após o cabeçalho *ethernet* e será estabelecido um *etherType* especial para indicar a presença dessa lista. Cada *switch* deverá tomar um item dessa lista e encaminhar o pacote de acordo com o número da porta especificada.

O *switch* P4 deve verificar a lista de roteamento de origem. Cada item tem um *bit* de fim de lista e o número da porta. O *bit* de fim lista será igual a 1 somente no último item da lista. Então, no ingresso do pacote, é necessário tomar um item da lista e configurar a porta de saída de acordo com o valor do item tomado. Note que no último salto o pacote deve ser revertido ao *etherType* `TYPE_IPV4` antes de ser entregue.

O diagrama apresentado na Figura 1.13 demonstra a topologia configurada para essa tarefa.

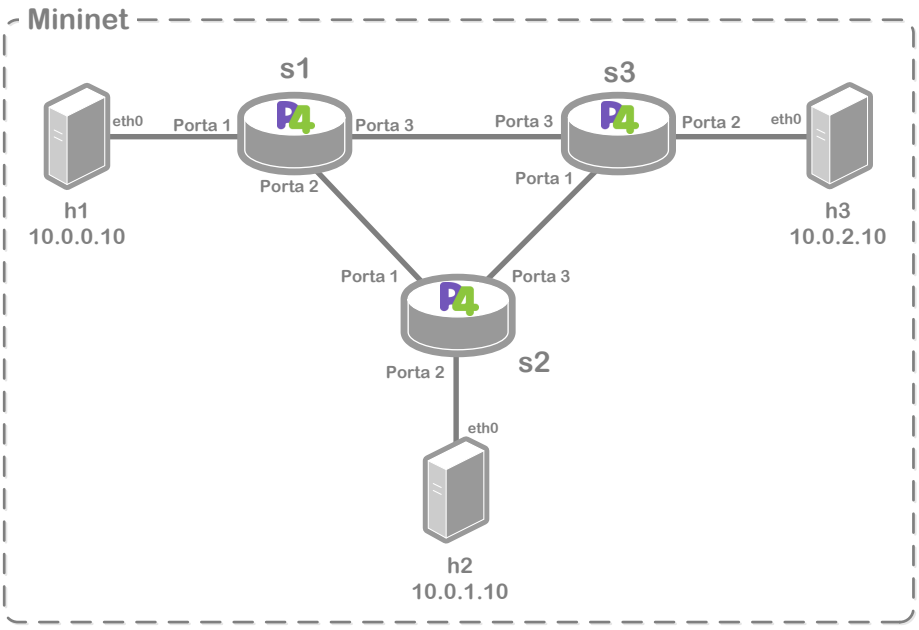

Figura 1.13. Topologia proposta para a atividade 3.

1.4.3.1. Passo 1: Execução

O diretório `/home/student/labs/task_3/p4src` contém o código P4 `task_3.p4`, que pode ser compilado, no entanto, em sua versão original, simplesmente descarta os pacotes que chegam até ele. O objetivo dessa atividade é completar esse código inicial, de forma que o *switch* P4 faça o encaminhamento do pacote ao *host* correto.

Primeiramente, compile o código de forma em que se encontra e execute o Mininet para testar o seu funcionamento.

Na linha de comando, dentro do diretório `/home/student/labs/task_3/` execute o *script*: `sudo ./run_task.sh`. Esse *script* irá compilar o código `task_3.p4` e inicializar o Mininet já com a topologia configurada para a atividade.

A instância do Mininet para essa atividade conta com três *switches* ($s1$, $s2$ e $s3$) e três *hosts* ($h1$, $h2$ e $h3$). Os *hosts* possuem os seguintes endereços IPs associados: $h1$ - 10.0.0.10, $h2$ - 10.0.1.10 e $h3$ - 10.0.2.10.

Uma vez que a instância do Mininet esteja carregada, obtem-se o *prompt* >. Abra então dois terminais, para $h1$ e $h2$, respectivamente:

```
mininet> xterm h1 h2
```

Em cada um dos *hosts* existem aplicações baseadas em Python para enviar e receber mensagens. Inicie primeiramente a aplicação para receber mensagens no terminal do *host* 2 ($h2$).

```
sudo ./receive.py
```

Em seguida, inicie a aplicação para enviar mensagem a partir do *host* 1 (*h*1). A sintaxe do comando é `send.py <ip do receiver>`:

```
sudo ./send.py 10.0.1.10
```

Digite então a lista de roteamento de origem, digamos, `2 3 2 2 1`. Essa lista deveria fazer com que o pacote percorra uma rota passando por: *h*1, *s*1, *s*2, *s*3, *s*1, *s*2 e *h*2. Entretanto o pacote não será recebido por *h*2.

Saia do *prompt* do *script* Python digitando `q` e então digite `exit` para sair do `xterm` e voltar a linha de comando do Mininet. Então saia do Mininet digitando `exit` na linha de comando.

```
mininet> exit
```

A mensagem não foi entregue porque o *switch* está programado com o código P4 original em `task_3.p4`, que descarta todos os pacotes recebidos. O objetivo dessa tarefa é estender o código P4 para que os pacotes consigam alcançar o seu destino.

1.4.3.2. Passo 2: Implementação

O arquivo `task_3.p4` contém o código P4 com indicações das partes a serem implementadas, assinaladas com o comentários **PARA COMPLETAR**. A implementação a ser feita deverá seguir a estrutura proposta nesse arquivo, substituindo os comentários pelo código P4 necessário para que a implementação funcione.

O arquivo `task_3.p4` completo deve ter os seguintes componentes:

1. Definições de cabeçalho para as camadas *ethernet* (`ethernet_t`), IPv4 (`ipv4_t`) e o roteamento de origem (`srcRoute_t`).
2. **PARA COMPLETAR:** Os *parsers* para *ethernet*, IPv4 e roteamento de origem que populem os campos de `ethernet_t`, `ipv4_t` e `srcRoute_t`.
3. Pelo menos uma ação para descartar os pacotes, usando `mark_to_drop()`.
4. **PARA COMPLETAR:** Uma ação (chamada `srcRoute_nhop`) que irá:
   (a) Determinar a porta de saída para o próximo salto;
   (b) Remover o primeiro item da lista roteamento de origem.
5. **U**m controle com um bloco de aplicação que:
   (a) Verifica a existência das rotas de origem.
   (b) PARA COMPLETAR: Verifica se o `ethernet.etherType` deve ser alterado, caso o pacote esteja sendo enviado ao último *host*.
   (c) PARA COMPLETAR: Chamar a ação `srcRoute_nhop`
6. Um `deparser` que selecione a ordem que cada campo deve ser inserido no pacote que está saindo do *switch*.

7. Um instanciação do pacote composta por `parser`, `control` e `deparser`.

Eventualmente esse processamento também requer instâncias de verificação de *checksum* e controles de recálculo. Isso não será necessário para essa tarefa e foram substituídas com controles vazios.

1.4.3.3. Passo 3: Execução Final

Siga as instruções do Passo 1. Agora as mensagens do *host h*1 deverão ser entregues ao *host h*2.

Verifique o `TTL` do cabeçalho IP. A cada salto o `TTL` será decrementado. A sequência de portas `2 3 2 2 1` força o pacote e fazer uma volta a mais na rede (*loop*), então o `TTL` deverá ser 59 no *host h*2.

Reflexões

- É possível alterar a lógica do código P4 para manipular o encaminhamento IPv4 e o roteamento de origem ao mesmo tempo?
- Como você poderia melhorar seu código para permitir que o primeiro *switch* adicionasse a rota ao pacote, tornando o roteamento de origem transparente para todos os *hosts*?

Solução de Problemas

Existem vários problemas que podem acontecer durante o desenvolvimento da implementação:

1. O código `task_3.p4` pode falhar durante a compilação. Nesse caso o *script* `run_task_3.sh` apresentará na tela o erro de compilação e será terminado.
2. O código `task_3.p4` pode compilar, mas falhar ao carregar as regras do plano de controle descritas no arquivo `command_s<número_do_switch>.txt` que o *script* `run_task.sh` tentará instalar usando o CLI BMv2. Neste caso, o `run_task.sh` irá registrar a saída do CLI no diretório de *logs*. Utilize essa mensagem de erro para consertar a sua implementação.
3. O código `task_3.p4` pode compilar, e as regras do plano de controle descritas no arquivo command_s1.txt podem ser carregadas sem problemas, mas ainda assim, o *switch* P4 pode não conseguir processar os pacotes corretamente. O arquivo `/tmp/p4s.s1.log`, contém informações detalhadas descrevendo como o pacote foi processado pelo *switch*. Este *log* pode ajudar a encontrar erros de lógica na sua implementação. Um captura de pacotes também será gerada para as interfaces de entrada e saída de cada um dos *switchs* no formato `<switch-name>-<interface-name>.pcap`. Utilize o comando `tcpdump -r <filename> -xxx` e imprima a saída em hexadecimal (*hexdump*) dos pacotes capturados.

Reiniciando o Mininet

Alguns casos de falha podem ser causados devido a inicialização do ambiente. Nestes casos, reiniciar o Mininet pode ajudar. Utilize o comando `mn -c` para terminar as instâncias que ficaram travadas.

1.4.4. Respostas Sugeridas

Não exite somente uma resposta correta para resolver as atividades propostas. A seguir são apresentadas uma das possíveis soluções para cada uma das atividades.

1.4.4.1. Atividade 1 - Encaminhamento de Pacotes

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
*************************************************************
****************** H E A D E R S  ***********************
*************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
```

```
}

/*************************************************************
******************* P A R S E R  ************************
*************************************************************/

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }

}

/*************************************************************
******   C H E C K S U M    V E R I F I C A T I O N   ******
*************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta)
   {
    apply {  }
}

/*************************************************************
*********  I N G R E S S   P R O C E S S I N G   ***********
*************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop();
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
```

```
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}

/*************************************************************
**********  E G R E S S   P R O C E S S I N G   ************
*************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}

/*************************************************************
******   C H E C K S U M    C O M P U T A T I O N   ********
*************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta
    ) {
     apply {
        update_checksum(
            hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
```

```
}

/*************************************************************
****************  D E P A R S E R  ***********************
*************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}

/*************************************************************
******************  S W I T C H  ****************************
*************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

1.4.4.2. Atividade 2 - Implementando uma calculadora em P4

```
/* -*- P4_16 -*- */

/*
 * P4 Calculator
 *
 * This program implements a simple protocol. It can be carried
    over Ethernet
 * (Ethertype 0x1234).
 *
 * The Protocol header looks like this:
 *
 *        0             1             2             3
 * +-------------+-------------+-------------+-------------+
 * |      P      |      4      |    Version  |     Op      |
 * +-------------+-------------+-------------+-------------+
 * |                         Operand A                     |
 * +-------------+-------------+-------------+-------------+
 * |                         Operand B                     |
 * +-------------+-------------+-------------+-------------+
 * |                         Result                        |
 * +-------------+-------------+-------------+-------------+
 *
 * P is an ASCII Letter 'P' (0x50)
 * 4 is an ASCII Letter '4' (0x34)
 * Version is currently 0.1 (0x01)
 * Op is an operation to Perform:
 *   '+' (0x2b) Result = OperandA + OperandB
```

```
 *   '-' (0x2d) Result = OperandA - OperandB
 *   '&' (0x26) Result = OperandA & OperandB
 *   '|' (0x7c) Result = OperandA | OperandB
 *   '^' (0x5e) Result = OperandA ^ OperandB
 *
 * The device receives a packet, performs the requested operation
    , fills in the
 * result and sends the packet back out of the same port it came
    in on, while
 * swapping the source and destination addresses.
 *
 * If an unknown operation is specified or the header is not
    valid, the packet
 * is dropped
 */

#include <core.p4>
#include <v1model.p4>

/*
 * Define the headers the program will recognize
 */

/*
 * Standard ethernet header
 */
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}

/*
 * This is a custom protocol header for the calculator. We'll use
 * ethertype 0x1234 for is (see parser)
 */
const bit<16> P4CALC_ETYPE = 0x1234;
const bit<8>  P4CALC_P     = 0x50;   // 'P'
const bit<8>  P4CALC_4     = 0x34;   // '4'
const bit<8>  P4CALC_VER   = 0x01;   // v0.1
const bit<8>  P4CALC_PLUS  = 0x2b;   // '+'
const bit<8>  P4CALC_MINUS = 0x2d;   // '-'
const bit<8>  P4CALC_AND   = 0x26;   // '&'
const bit<8>  P4CALC_OR    = 0x7c;   // '|'
const bit<8>  P4CALC_CARET = 0x5e;   // '^'

header p4calc_t {
    bit<8>  p;
    bit<8>  four;
    bit<8>  ver;
    bit<8>  op;
    bit<32> operand_a;
    bit<32> operand_b;
    bit<32> res;
}
```

```
/*
 * All headers, used in the program needs to be assembed into a
    single struct.
 * We only need to declare the type, but there is no need to
    instantiate it,
 * because it is done "by the architecture", i.e. outside of P4
    functions
 */
struct headers {
    ethernet_t   ethernet;
    p4calc_t     p4calc;
}

/*
 * All metadata, globally used in the program, also  needs to be
    assembed
 * into a single struct. As in the case of the headers, we only
    need to
 * declare the type, but there is no need to instantiate it,
 * because it is done "by the architecture", i.e. outside of P4
    functions
 */

struct metadata {
    /* In our case it is empty */
}

/*************************************************************
 ***************  P A R S E R  ***************************
*************************************************************/
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            P4CALC_ETYPE : check_p4calc;
            default      : accept;
        }
    }

    state check_p4calc {
        transition select(packet.lookahead<p4calc_t>().p,
        packet.lookahead<p4calc_t>().four,
        packet.lookahead<p4calc_t>().ver) {
            (P4CALC_P, P4CALC_4, P4CALC_VER) : parse_p4calc;
            default                          : accept;
        }
    }

    state parse_p4calc {
        packet.extract(hdr.p4calc);
```

```
        transition accept;
    }
}

/*************************************************************
******   C H E C K S U M    V E R I F I C A T I O N   ******
*************************************************************/
control MyVerifyChecksum(inout headers hdr,
                         inout metadata meta) {
    apply { }
}

/*************************************************************
********  I N G R E S S   P R O C E S S I N G   ************
*************************************************************/
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action send_back(bit<32> result) {
        bit<48> tmp;

        /* Put the result back in */
        hdr.p4calc.res = result;

        /* Swap the MAC addresses */
        tmp = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = hdr.ethernet.srcAddr;
        hdr.ethernet.srcAddr = tmp;

        /* Send the packet back to the port it came from */
        standard_metadata.egress_spec = standard_metadata.
            ingress_port;
    }

    action operation_add() {
        send_back(hdr.p4calc.operand_a + hdr.p4calc.operand_b);
    }

    action operation_sub() {
        send_back(hdr.p4calc.operand_a - hdr.p4calc.operand_b);
    }

    action operation_and() {
        send_back(hdr.p4calc.operand_a & hdr.p4calc.operand_b);
    }

    action operation_or() {
        send_back(hdr.p4calc.operand_a | hdr.p4calc.operand_b);
    }

    action operation_xor() {
        send_back(hdr.p4calc.operand_a ^ hdr.p4calc.operand_b);
    }
```

```
    action operation_drop() {
        mark_to_drop();
    }

    table calculate {
        key = {
            hdr.p4calc.op        : exact;
        }
        actions = {
            operation_add;
            operation_sub;
            operation_and;
            operation_or;
            operation_xor;
            operation_drop;
        }
        const default_action = operation_drop();
        const entries = {
            P4CALC_PLUS : operation_add();
            P4CALC_MINUS: operation_sub();
            P4CALC_AND  : operation_and();
            P4CALC_OR   : operation_or();
            P4CALC_CARET: operation_xor();
        }
    }


    apply {
        if (hdr.p4calc.isValid()) {
            calculate.apply();
        } else {
            operation_drop();
        }
    }
}

/*************************************************************
 **********  E G R E S S   P R O C E S S I N G   **********
*************************************************************/
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply { }
}

/*************************************************************
*******   C H E C K S U M    C O M P U T A T I O N   *******
*************************************************************/

control MyComputeChecksum(inout headers hdr, inout metadata meta)
    {
    apply { }
}

/*************************************************************
```

```
*****************  D E P A R S E R  ***********************
************************************************************/
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.p4calc);
    }
}

/************************************************************
******************  S W I T T C H ************************
************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

1.4.4.3. Atividade 3 - Implementando roteamento na origem

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

const bit<16> TYPE_IPV4 = 0x800;
const bit<16> TYPE_SRCROUTING = 0x1234;

#define MAX_HOPS 9

/************************************************************
********************* H E A D E R S  ***********************
************************************************************/

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header srcRoute_t {
    bit<1>    bos;
    bit<15>   port;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
```

```
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct metadata {
    /* empty */
}

struct headers {
    ethernet_t              ethernet;
    srcRoute_t[MAX_HOPS]    srcRoutes;
    ipv4_t                  ipv4;
}

/*************************************************************
***************** P A R S E R  ***************************
*************************************************************/

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_SRCROUTING: parse_srcRouting;
            default: accept;
        }
    }

    state parse_srcRouting {
        packet.extract(hdr.srcRoutes.next);
        transition select(hdr.srcRoutes.last.bos) {
            1: parse_ipv4;
            default: parse_srcRouting;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
```

```
}

/*************************************************************
******   C H E C K S U M    V E R I F I C A T I O N   ******
*************************************************************/

control MyVerifyChecksum(inout headers hdr, inout metadata meta)
   {
    apply {  }
}

/*************************************************************
*********  I N G R E S S   P R O C E S S I N G   ***********
*************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop();
    }

    action srcRoute_nhop() {
        standard_metadata.egress_spec = (bit<9>)hdr.srcRoutes[0].
           port;
        hdr.srcRoutes.pop_front(1);
    }

    action srcRoute_finish() {
        hdr.ethernet.etherType = TYPE_IPV4;
    }

    action update_ttl(){
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    apply {
        if (hdr.srcRoutes[0].isValid()){
            if (hdr.srcRoutes[0].bos == 1){
                srcRoute_finish();
            }
            srcRoute_nhop();
            if (hdr.ipv4.isValid()){
                update_ttl();
            }
        }else{
            drop();
        }
    }
}

/*************************************************************
*********  E G R E S S   P R O C E S S I N G   *************
```

```
*************************************************************/

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}

/*************************************************************
*******   C H E C K S U M    C O M P U T A T I O N   *******
*************************************************************/

control MyComputeChecksum(inout headers  hdr, inout metadata meta
    ) {
    apply {  }
}

/*************************************************************
*****************  D E P A R S E R  ***********************
*************************************************************/

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.srcRoutes);
        packet.emit(hdr.ipv4);
    }
}

/*************************************************************
*********************  S W I T C H  *************************
*************************************************************/

V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

1.5. Tendências e Conclusões

O P4 vem como uma segunda onda de transformação nas redes de computadores, segue a trilha aberta pela mudança de paradigma arquitetônico trazida pelo SDN. A separação entre o plano de controle e o plano de dados foi estabelecida pelo SDN, mas a programabilidade da rede ainda era papel quase exclusivo do plano de controle. Apesar do sucesso das implementações feitas pela primeira onda liderada pelo OpenFlow, a questão da programabilidade direta do plano de dados ainda não tinha sido resolvida de forma apropriada.

Sínteses das funções de rede no plano de dados vinham sendo implementadas por programas escritos em linguagens de alto nível, dai havia uma posterior compilação

em linguagem da plataforma alvo, como o Verilog e VHDL. Todavia, a expressividade requerida para a programabilidade de rede no plano de dados sofria com a falta de bibliotecas e módulos que combinassem as abstrações comuns desses módulos com objetos para desenvolver funcionalidades específicas. Some-se a isso fatores complicantes como a ausência de camadas intermediárias, e.g., sistema operacional e *drivers*, que nos habituamos em sistemas baseados em CPUs. Portanto, havia uma exposição de complexidade do hardware ao usuário dessas plataformas baseadas em FPGA, por exemplo. Isso criava uma barreira de entrada difícil de ser transposta para a implementação de funções de rede no plano de dados [Sultana et al. 2017].

É importante também entender que a fase de demonstração do potencial das novas funcionalidades de rede pelo uso de SDN/OpenFlow já foi vencida, e desempenho é o que se espera alcançar no momento. Busca-se também redução do tempo requerido prototipação de serviços de rede para acelerar os ciclos de inovação. Porém, nota-se a ausência de plataformas convenientes de *debug* para agilizar desenvolvimento de protótipos quando seguimos as soluções convencionais descritas acima.

Como uma proposta para preencher essas lacunas surge a linguagem P4 para, finalmente, habilitar programabilidade do plano de dados de dispositivos de rede. A baixa barreira de entrada e suas descrições de funcionalidades de rede independentes da plataforma alvo podemos resumir como os principais atrativos.

A linguagem P4 traz portanto benefícios para as inúmeras inovações funcionais já feitas em SDN que agora podem ser transferidas para o plano de dado de forma a melhorar desempenho. Todavia, a sua expressividade também viabiliza novas soluções para problemas clássicos. Por exemplo, medições ampliadas à toda rede valendo-se de monitoramento simplificado usando estruturas de dados probabilísticas (*i.e., sketches*) [Liu et al. 2016] e balanceamento de carga sem manutenção de estados em *datacenters* [Olteanu et al. 2018].

Para analisarmos as tendências, primeiro temos que entender que o sucesso da linguagem P4 vai depender, naturalmente, de adoção ampla por fabricantes de equipamentos, de criação e da manutenção de comunidades para a difusão de soluções básicas e na criação de bibliotecas. Pela origem comum, acreditamos que os mesmos caminhos eficientes de difusão das soluções OpenFlow sejam empregados também para as desenvolvidas via linguagem P4.

Notamos um número crescente de compiladores para a linguagem P4, o que pode ser um importante sinal de interesse da comunidade e as variadas plataformas alvo existente ajudam muito na difusão da linguagem. Há inclusive propostas de criação de uma *benchmark suite* para permitir comparação entre as diversas soluções de compilação [Dang et al. 2017].

Outro fator importante é que o P4 vem gradualmente casando o seu nível de expressividade com o da linguagem nativa da plataforma alvo. Discussões sobre a tradução de P4 para descrição de hardware (VHDL) pode ser encontrada em [Benek et al. 2018]. O objetivo buscado é, naturalmente, o aumento de eficiência das implementações. Funcionalidades de alto desempenho (i.e., em 100Gbps) podem ser dai geradas automaticamente para componentes principais de processamento de pacotes, como apresenta essa outra ini-

ciativa recente [Benáček et al. 2018].

Há ainda uma nova plataforma de alto desempenho que serve como alternativa ao uso de *switches bare metal* e FPGAs. A incorporação de programabilidade em P4 às interfaces de rede de servidores já é uma realidade via *SmartNICs*. Isso torna possível elevar implementações flexíveis de programação de rede, que são usualmente feitas com software *switches* dos servidores, ao patamar de alto desempenho - porém a custos bem reduzidos.

O espaço dos software *switches* com P4 também está bem ativo, afinal a expressividade permitida pelo P4 tem um papel importantíssimo no contexto de computação em nuvem, onde redes são cada vez mais compostas com elementos virtualizados. Neste cenário há trabalhos de extensão do OvS tradicional para a inclusão, por exemplo, de filtros com estados e funções de monitoramento em tempo de execução via P4 [Chaignon et al. 2018]. De outro lado, temos propostas como o PISCES [Shahbaz et al. 2016] que é um protótipo independente que pode ser programado a partir da linguagem P4.

Um outro sinal de vitalidade de novas propostas é o surgimento de replicas de ferramentas bem difundidas na comunidade de redes, e também o desenvolvimento de capacidade de se implementar testes, monitoramentos e validações para vencer as limitações de *debug* comentadas anteriormente. Um exemplo recente nessa linha é o *p4pktgen* [Nötzli et al. 2018], uma ferramenta automática de geração de casos de teste, e com capacidade de avaliação de códigos P4 usando execução simbólica.

O surgimento de novas soluções, antes limitadas pelo excesso de estados a serem mantido no plano de dados, dão prova do potencial futuro do P4 na resolução de problemas complexos [Jepsen et al. 2018]. Com capacidade de processamento em algumas aplicações já ultrapassando Terabits por segundo [Cabal et al. 2018], a utilização da linguagem P4 claramente eleva o patamar dos protótipos produzidos, facilitando a transição de soluções implementadas para a fase de pilotos e dai para produtos, reduzindo o tempo dos ciclos de inovação em redes de computadores. Isso permite avançar em diversas áreas de aplicação, como por exemplo naquelas pouco exploradas como a aviônica [Geyer and Winkel 2018].

A comunidade de redes deve atentar, entretanto, para o fato de que à medida que as soluções abertas aproximam-se dos níveis mais baixos, barreiras são erguidas por fornecedores de hardware com suas usuais licenças de utilização. E isso pode frustrar iniciativas de inovação lembrando-nos que a fronteira do *opensource* precisa avançar ainda mais.

Por último, não podemos deixar de comentar que há críticas apontadas à solução baseada em linguagem especializada e, portanto, engessada num paradigma específico de processamento de pacotes; como é o caso da linguagem P4. Argumenta-se que o baixo nível de abstração do P4 inibe o desenvolvimento de funcionalidades mais abrangentes. A plataforma *Emu* [Wang et al. 2017], por exemplo, alega desempenho similar ao P4 utilizando-se bibliotecas padronizadas para a criação ágil de funcionalidades de rede. Como no P4, as plataformas alvo do *Emu* podem ser CPUs, Mininet e FPGAs; porém desfrutando de ambiente de *debug* mais rico. Indicamos ainda a Tabela 1 de [Wang et al. 2017] para o leitor interessado numa discussão comparativa do *Emu* com o P4 e outras ferramentas que também habilitam serviços de rede implementados em hardware.

Referências

[Benek et al. 2018] Benek, P., Pu, V., Kubtov, H., and ejka, T. (2018). P4-to-vhdl. *Microprocess. Microsyst.*, 56(C):22–33.

[Benáček et al. 2018] Benáček, P., Puš, V., Kubátová, H., and Čejka, T. (2018). P4-to-vhdl: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems*, 56:22 – 33.

[Bosshart et al. 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

[Cabal et al. 2018] Cabal, J., Benáček, P., Kekely, L., Kekely, M., Puš, V., and Kořenek, J. (2018). Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 249–258, New York, NY, USA. ACM.

[Carmelo Cascone 2016] Carmelo Cascone (2016). P4 support in ONOS. `https://wiki.onosproject.org`.

[Chaignon et al. 2018] Chaignon, P., Lazri, K., François, J., Delmas, T., and Festor, O. (2018). Oko: Extending open vswitch with stateful filters. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 13:1–13:13, New York, NY, USA. ACM.

[Changhoon Kim 2016] Changhoon Kim (2016). Programming the network dataplane. `http://netseminar.stanford.edu/seminars/03_31_16.pdf`.

[Consortium 2017] Consortium, T. P. L. (2017). The P4 Language Specification - version 1.0.0. Specification, The P4 Language Consortium, https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[Dang et al. 2017] Dang, H. T., Wang, H., Jepsen, T., Brebner, G., Kim, C., Rexford, J., Soulé, R., and Weatherspoon, H. (2017). Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 95–101, New York, NY, USA. ACM.

[Geyer and Winkel 2018] Geyer, F. and Winkel, M. (2018). Towards Embedded Packet Processing Devices for Rapid Prototyping of Avionic Applications. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France.

[Ibanez 2017] Ibanez, S. (2017). P4 netfpga tutorial. `https://github.com/NetFPGA/P4-NetFPGA-public/blob/master/slides/P4-NetFPGA-SIGCOMM-17/P4-NetFPGA_SIGCOMM_2017_v2.pdf`. [Accesado 9-Nov-2017].

[Jepsen et al. 2018] Jepsen, T., Moshref, M., Carzaniga, A., Foster, N., and Soulé, R. (2018). Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 10:1–10:7, New York, NY, USA. ACM.

[Liu et al. 2016] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. (2016). One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA. ACM.

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

[Nötzli et al. 2018] Nötzli, A., Khan, J., Fingerhut, A., Barrett, C., and Athanas, P. (2018). P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 5:1–5:7, New York, NY, USA. ACM.

[Olteanu et al. 2018] Olteanu, V., Agache, A., Voinescu, A., and Raiciu, C. (2018). Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI) 18)*, pages 125–139. USENIX.

[Shahbaz et al. 2016] Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., and Rexford, J. (2016). Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA. ACM.

[Sultana et al. 2017] Sultana, N., Galea, S., Greaves, D., Wojcik, M., Shipton, J., Clegg, R., Mai, L., Bressana, P., Soulé, R., Mortier, R., Costa, P., Pietzuch, P., Crowcroft, J., Moore, A. W., and Zilberman, N. (2017). Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, Santa Clara, CA. USENIX Association.

[Wang et al. 2017] Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. (2017). P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA. ACM.