Capítulo 3

Introdução à linguagem de programação P4, o futuro das redes

Pedro Eduardo Camera e Alisson Borges Zanetti

Abstract

This short course aims to explain the functioning of the P4 language, demonstrating its relevance today through examples of practical application, such as monitoring links between network equipment. With a theoretical approach and exercises, this paper discusses the main points of SDN networking and how the programming of packets in the data plane, in the figure of P4, enables the increase of user experience, troubleshooting, better resources usage, and network control centralization.

Resumo

Este minicurso tem como objetivo explanar o funcionamento da linguagem P4, demostrando sua relevância nos dias atuais por meio de exemplos de aplicação prática, como o monitoramento e links entre os equipamentos de rede. Dispondo de um referencial teórico e exercícios, este documento discorre sobre os principais pontos das redes SDN e de como a programação de pacotes no plano de dados, na figura do P4, possibilita o aumento da experiência de usuário, capacidade de troubleshooting, otmização do uso de recursos e centralização do controle de rede.

3.1. Introdução

O monitoramento de rede é crucial na identificação e resolução de problemas, sendo implementado desde empresas de pequeno porte a operadoras de alta performance. Todavia, à medida que as redes se expandem, cresce a complexidade de gerenciamento e *troubleshooting* em suas estruturas. Outra adversidade é o destaque da experiência de usuário na formulação das aplicações, compondo-se insuficiente, frente às exigências atuais de desempenho, a mera disponibilidade e redundância dos serviços. A heterogeneidade também é um fator contribuinte, posto que os fabricantes, a fim de diferenciar seus produtos, empregam soluções proprietárias que dificultam o desenvolvimento de ferramentas de

monitoramento abrangentes e efetivas, impelindo o uso de utilitários reativos e datados (como *traceroute*, SNMP, ICMP, dentre outros) para tal tarefa.

Com o surgimento das redes definidas por software (*Software Defined Networking*), tratadas daqui para frente como SDN, novas formas de monitoramento tornaram-se possíveis. Apresentada em 2008, esse tipo de rede tem como principais características a separação das camadas de controle e dados, a centralização da gerência dos ativos e a visualização global do estado da rede [Singh and Jha 2017]. Esse panorama é alcançado por meio da definição de programas customizáveis que intermedeiam (e ditam) as ações do hardware. Nos últimos anos, o paradigma SDN se popularizou especialmente em redes que comportam aplicações de alta demanda [Haleplidis et al. 2015].

Apesar disso, com o passar do tempo, a centralização SDN deslocou a programabilidade ao baixo nível (camada de dados), a fim de balancear a carga imposta ao gerenciador SDN (camada de controle). A contar de 2014, o aparecimento de linguagens de processamento de pacotes, como o P4 (*Programming Protocol-Independent Packet Processors*), bem como seu suporte em hardware, possibilitaram ao plano de dados a execução de tarefas atreladas unicamente aos controles superiores. Com isso, uma nova gama de aplicações, incluindo o monitoramento de redes em tempo real, começaram a ser esboçadas.

O monitoramento tem como base a busca por informações de *status* da rede e sua demonstração aos administradores. O presente artigo propõe um exercício sobre essa nova tecnologia, a saber o P4, utilizando-se de um algoritmo nele programado para realizar varreduras acerca do estado de rede dos equipamentos. Por encontrar-se no baixo nível, implementações P4 conseguem capturar metadados restritos ao hardware. Com isso, seu nível de precisão e detalhamento aprofundam a compreensão das atividades de rede.

Este trabalho divide-se da seguinte forma: uma breve fundamentação teórica a respeito dos conceitos SDN (seção 3.2.1) e P4 (seção 3.2.2.1 em diante) são apresentados, seguidos de uma descrição das ferramentas essenciais à execução desta proposta (seção 3.3) e das particularidades da implementação do programa P4 em si (seção 3.4), encerrando-se com a conclusão e deposições finais (seção 3.5). Demais detalhes são providos pelas informações anexas a este minicurso (seção 3.6).

3.2. Referencial Teórico

3.2.1. SDN

O paradigma SDN foi desenvolvido por cientistas da Universidade de Stanford, sendo fruto de esforços de egressos de outras universidades americanas, como Berkeley, Carnigie Mellon e Princeton [Sood and Xiang 2017]. Segundo a ONF (*The Open Networking Foundation*) [1], o objetivo da SDN é reduzir custos e melhorar a experiência do usuário, automatizando toda a gama de serviços de rede. Seus princípios incluem o desacoplamento entre controle e dados, a capacidade de interação direta dos elementos da rede com este e a centralização do gerenciamento [OPEN NETWORK FOUNDATION 2016].

A capacidade de dissociar os planos dá-se ao tornar a camada de controle a respon-

[1] Consórcio sem fins lucrativos que desenvolve, padroniza e comercializa soluções SDN.

sável pela decisão de roteamento, analisando os pacotes e resolvendo a melhor forma de lidar com o tráfego, enquanto a camada de dados apenas encaminha de acordo com a deliberação tomada [Singh and Jha 2017]. Através de fluxos, que qualificam uma sequência de movimentos entre origem e destino, as determinações de envio são formuladas, sendo os pacotes balizados por critérios de correspondência e ação (*match-action*). Nos equipamentos, pacotes de mesmo fluxo recebem políticas de serviço idênticas. Essa abstração uniformiza o comportamento dentro dos diferentes dispositivos de rede, tais quais roteadores, *switches*, *firewalls* e *middleboxes* [Kreutz et al. 2015].

O aprovisionamento de um plano de controle unificado, onde um único software gere diversos planos de dados com múltiplos membros, ocorre por meio de protocolos como o OpenFlow [Singh and Jha 2017]. A Figura 3.1 representa todas essas vertentes, contrastando-as com a arquitetura tradicional. Percebe-se a forma consolidada com que as relações SDN ocorrem, onde tanto as aplicações de alto nível quanto as operações de baixo têm a supervisão do controlador.

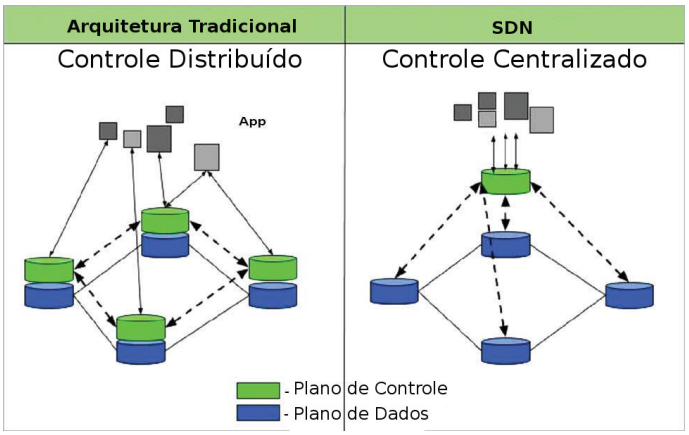

Figura 3.1: Diferenças entre rede tradicional e SDN. Fonte: Autores.

3.2.1.1. OpenFlow

Conforme mencionando, a centralização do controle é obtida através do uso do protocolo aberto OpenFlow [Huang et al. 2016], responsável pelo encorajamento dos fornecedores na implementação de alguns conceitos SDN em produtos de rede [Kreutz et al. 2015]. Proposto por McKeown et al. [McKeown et al. 2008] [Sood and Xiang 2017], o protocolo é utilizado na maioria das propostas SDN, possuindo diversas aplicações *open source* de controladores. Experimentos iniciais utilizando-se OpenFlow foram criados para separar a rede em uma fatia de software controlável, com foco no encaminhamento de pacotes. O protocolo aproveita-se do fato de *switches* e roteadores modernos possuírem tabelas de fluxo para funções de roteamento, máscaras de sub-rede, proteção de *firewall* e análise estatística do fluxo de dados [Xia et al. 2015].

3.2.1.2. Comutação SDN

Dependendo das regras instaladas pelo aplicativo controlador, um *switch* OpenFlow pode se comportar como um roteador, comutador, *firewall*, *load balancer*, *traffic shaper* ou quaisquer outras funções que caracterizam um *middlebox* de rede [Kreutz et al. 2015]. A Figura 3.2 traz de modo simplificado o funcionamento da comutação SDN. Os *switches* OpenFlow suportam a operação de tabelas de fluxo, por onde o protocolo OpenFlow combina e processa pacotes de rede pelas regras nelas estabelecidas. Cada entrada da tabela constitui-se de três componentes (cabeçalho, instrução e estatísticas) em vez da entrada de roteamento tradicional (quíntupla IP). Os pacotes são correspondidos (*matching*) por seus campos de cabeçalho (*header*) e, em seguida, processados de acordo com a instrução (*action*) no fluxo de entrada. As estatísticas indicam o *status* da rede, incluindo prioridade, contadores, e assim por diante [Gong et al. 2015].

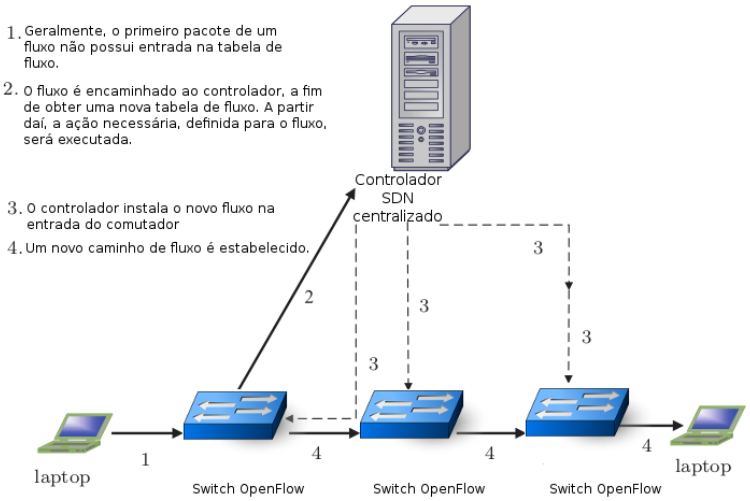

Figura 3.2: Comutação com Switch OpenFlow. Fonte: SOOD, K; XIANG, Y. [Sood and Xiang 2017].

3.2.2. Programabilidade no plano de dados

Consoante o já exposto, as camadas de controle e dados caracterizam-se como fisicamente separadas [Santiago da Silva et al. 2018] [Bosshart et al. 2014], o que permite a evolução de ambas distintamente. Essa separação permitiu que os dispositivos de encaminhamento atingissem níveis maiores de programabilidade. Não obstante essa capacidade, houve um enredamento destes ao plano de controle. Ao utilizar-se de uma interface comum, aberta e independente de provedor (OpenFlow), a camada de controle revelou-se imprescindível ao plano de dados [Bosshart et al. 2014].

Inicialmente, a simplicidade satisfez a demanda. Porém, para permitir que dispositivos vigentes exponham mais de seus recursos ao controlador, a especificação OpenFlow (diretiva do uso do protocolo) tem-se tornado cada vez mais complexa, acrescendo campos e estágios de regras [Bosshart et al. 2014]. Embora continuamente atualizada, a catalogação fica aquém da proliferação de novos campos de cabeçalho. Em outras palavras, a padronização da programabilidade no plano de controle produziu um plano de

dados “fixo”, no sentido de que os usuários restringiram-se a trabalhar com protocolos identificados na especificação OpenFlow [Hancock and van der Merwe 2016].

3.2.2.1. P4

Para contornar as deficiências do OpenFlow [Santiago da Silva et al. 2018], o emprego de linguagens de programação no plano de dados vem ganhando adoção tanto na academia quanto na indústria. Abordagens como o P4 permitem descrever um comportamento agnóstico de encaminhamento no plano de dados. Por ser uma linguagem de alto nível, o P4 fornece um dialeto de rede simples para descrever o caminho dos datagramas, podendo trabalhar em conjunto com protocolos de controle SDN [Bosshart et al. 2014]. Com a proposta do P4, busca-se a evolução do modelo por meio da:

- **Reconfigurabilidade no campo:** os programadores devem ser capazes de mudar a forma como os *switches* processam os pacotes depois de implementados;
- **Independência do protocolo:** os comutadores não devem estar vinculados a nenhum protocolo de rede específico;
- **Independência do alvo:** os programadores devem ser capazes de descrever a funcionalidade de processamento de pacotes independentemente das especificidades do hardware subjacente;

3.2.2.2. PISA

Baseado na arquitetura PISA (*Protocol-Independent Switch Architecture*), compõe-se um programa P4 por declarações de cabeçalho (*header*), máquina de estado do analisador de pacotes (*parser*) e das tabelas de ação de correspondência (*match-action*). A linguagem P4 assume a implementação de um *parser* que percorre os *headers* do início ao fim, extraindo os valores de campo conforme o percurso. As medidas extraídas são enviadas para o processamento das tabelas de *match-action* [Bosshart et al. 2014], similar ao tratamento adotado pelo OpenFlow a nível de controle. Assim, é descrita a máquina de estado como o conjunto de transições de um cabeçalho para o próximo. A Figura 3.3 demonstra um caminho hipotético rastreado pelo *parser*, conforme a sequência de cabeçalhos extraídos.

As declarações de *header* especificam os nomes e as larguras dos campos para os cabeçalhos de protocolo nos quais o programa P4 se destina a operar. Em geral, cada cabeçalho contém uma lista ordenada de nomeações e seus respectivos tamanhos. A tabela *match-action* desempenha a principal função, identificando os campos de pacotes e metadados a serem lidos, além das possíveis ações executadas em resposta. As *actions* são funções parametrizáveis que invocam uma ou mais primitivas de linguagem [2]. Uma vez que as tabelas e ações são definidas, resta especificar o fluxo de controle entre uma tabela e outra, por meio de um conjunto de funções, condicionais e referências de tabela [Bosshart et al. 2014] [Hancock and van der Merwe 2016].

[2]Elementos mais simples existentes numa linguagem de programação.

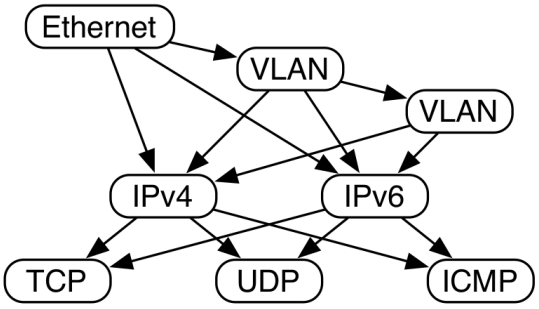

Figura 3.3: Representação do *parser* numa rede hipotética. Fonte: GIBB, G. et al. [Gibb et al. 2013].

A Figura 3.4 traz a abstração presente no envio de dados dentro do modelo P4/PISA. Primeiramente, os pacotes são manipulados pelo *parser*. Este reconhece e retira campos do *header* e, assim, define os protocolos suportados pelo comutador. Os campos extraídos são passados para as tabelas de *match-action*, divididas entre entrada (*ingress*) e saída (*egress*). Embora ambas possam modificar o *header*, a *ingress match-action* determina a(s) porta(s) de *egress* e a fila na qual o pacote é colocado. Com base nesse processamento, o pacote pode ser encaminhado, replicado (para *multicast*, *span* ou até ao plano de controle), descartado ou mesmo acionar o controle de fluxo. A *egress match-action* executa modificações por instância no *header*, por exemplo, as cópias *multicast*. As tabelas de ação (contadores, *policers* e assim por diante) podem ser associadas a um fluxo para rastrear o estado *frame-to-frame*, bem como informações de metadados que ofertam detalhes entre os estágios percorridos.

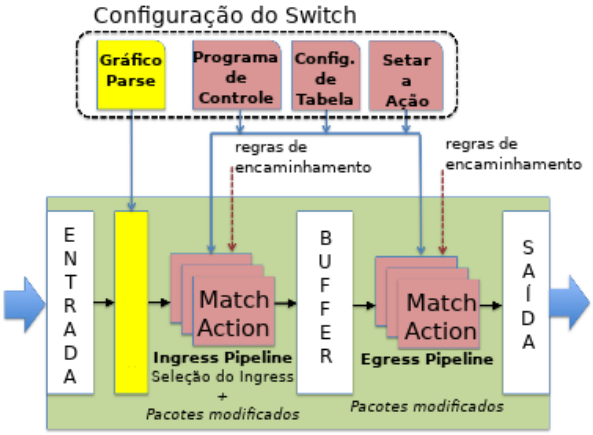

Figura 3.4: Abstração do modelo de encaminhamento. Fonte: BOSSHART, P. et al. [Bosshart et al. 2014].

3.2.2.3. Compilador P4

Com a popularização, diversos compiladores P4 emergiram recentemente. Neste trabalho, atentaremos ao P4C (compilador de referência), onde o processo de compilação é definido

em dois períodos. Num primeiro momento, o programa de controle P4 é convertido para uma representação gráfica das dependências entre as tabelas, conhecida como TDG (*Table Dependency Graphs*). Essa dependência estipula quais tabelas podem ser executadas em paralelo. Em analogia, podemos realizar a leitura simultânea dos conteúdos da tabela IP e ARP, pois ambas possuem dependência de dados entre si. Posteriormente, essa exibição é utilizada por um *back-end* específico de destino, que consome os dados na construção dos recursos especificados para o dispositivo [Bosshart et al. 2014]. Para uma melhor compreensão, observa-se o esquema expresso na Figura 3.5, que traz um exemplo de TDG para um contexto de *switches* L2/L3. Os nós TDG são mapeados diretamente para as tabelas de *match-action*, na qual uma análise de dependência identifica a colocação de cada tabela dentro do *pipeline*.

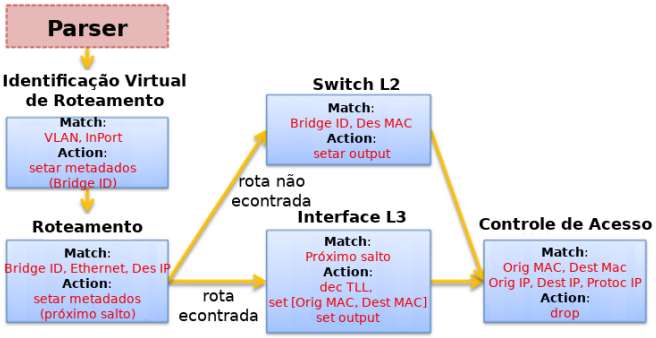

Figura 3.5: Gráfico de Dependência de Tabela (TDG) para um dispostivo L2/L3. Fonte: Adaptado de BOSSHART, P. et al. [Bosshart et al. 2014].

Corroborando o processo acima, a Figura 3.6 repassa as etapas de *parsing* do código-fonte P4 para então produzir uma representação intermediária (IR) de alto nível. Um *back-end* de compilador converte esta IR numa conformação específica de destino (por exemplo, código binário ou JSON). Como ofertas de *back-end* do compilador P4, podemos incluir o próprio P4C [Hancock and van der Merwe 2016]. Em dispositivos com *parsers* programáveis, o compilador converte a descrição em máquina de estado, enquanto nos fixos apenas verifica a descrição quanto à consistência com o alvo (*target*) pretendido [Bosshart et al. 2014]. Ou seja, a máquina de estado identifica, sequencialmente, a ordem e as relações do cabeçalho dentro do pacote. Começando do nó raiz, as transições de estado são tomadas em resposta aos próximos valores de campo de cabeçalho [Gibb et al. 2013].

3.3. Implementação

3.3.1. Mininet, BMv2 e P4Runtime

Antecipadamente, para o bom andamento do experimento, é necessário principiar o cursista a algumas noções básicas sobre o ambiente de testes. Para emular o âmbito SDN/P4, utiliza-se a ferramenta Mininet, um sistema computacional que imita o funcionamento de *switches*, controladores e *hosts* em tempo real. Neste sentido, o Mininet é um dos aplicativos mais empregados para avaliação de redes SDN, permitindo a emulação a partir de uma única máquina. Aproveitando-se de premissas de virtualização, este utilitário per-

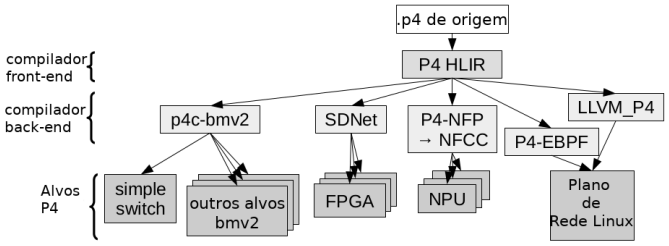

Figura 3.6: Compilando um programa P4 para vários alvos. Fonte: HANCOCK, D.; MERWE, J. van der. [Hancock and van der Merwe 2016].

mite ao usuário criar, interagir, personalizar e compartilhar um protótipo de rede completo que integra-se a controladores reais e demais atores externos.

Como a emulação tem suas limitações, é preciso estabelecer um padrão de procedimento fidedigno a uma rede P4 real. Nesta situação, um simulador de redes reproduz o comportamento de equipamentos reais através de um modelo matemático. Somando-se, o Mininet suporta o plano de dados programável com a ajuda do BMv2 (*Behavioral Model version 2*), que dita o procedimento dos *switches* P4 dentro da estrutura emulada. Outro agente a ser considerado é o *framework* P4Runtime que, nesta aplicação, desempenha as funções do plano de controle, gerenciando os elementos do plano de dados definidos em P4. A Figura 3.7 engloba todos os itens citados e suas correspondentes vinculações.

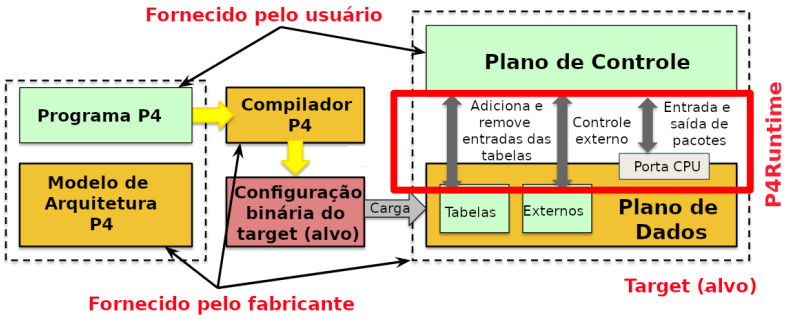

Figura 3.7: Fluxo de implementação do programa P4. Fonte: Autores.

3.3.2. Máquina Virtual

Visando o benefício máximo do tempo e a organização, dispõe-se uma máquina virtual contendo pré-instalações de pacotes de sistema e softwares que serão manipulados durante o laboratório. A mesma pode ser obtida através deste *hyperlink*[3]. Para se abrir o arquivo, recomenda-se a instalação do programa VirtualBox em sua versão 6.0.10 (ou superior) na máquina hospedeira.

[3]https://drive.google.com/a/upf.br/file/d/1E9KXKYV6cbyO-X-qR_5FCtaiLjVONkn0/view?usp=sharing

3.4. Programa P4 para monitorar *links*

Nesta seção apresenta-se uma prática P4 que objetiva um programa de monitoramento de ativos no plano de dados, por meio da coleta de informações sobre a rede, em especial do estado do *link*, seguindo a topologia exposta na Figura 3.8. Nela, ilustra-se a estrutura criada no emulador Mininet, na qual a simulação de comportamento dos comutadores fica a cargo do BMv2. Dispondo de quatro *switches*, nota-se que os superiores (Switch3 e Switch4) interligam-se com os da base (Switch1 e Switch2) e que estes, por sua vez, comunicam-se com os *hosts* 1, 2, 3 e 4, respectivamente. Para facilitar o entendimento, os *links* são também enumerados, posto que *switches* e *hosts* conectam-se por vários canais, quase formando uma *mesh*.

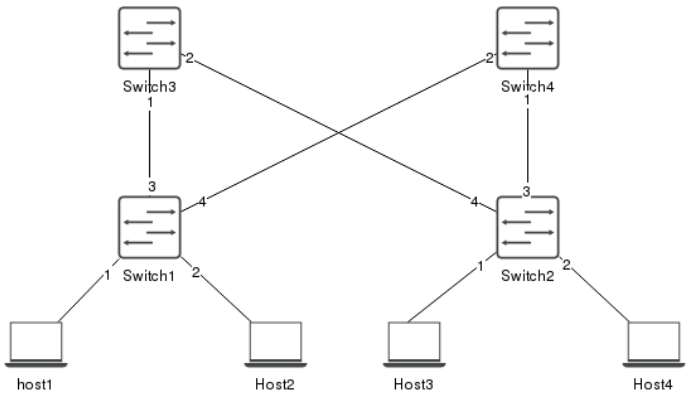

Figura 3.8: Topologia de rede básica do minicurso. Fonte: Autores.

Para que o objetivo de monitoramento seja alcançado, o *host*1 encaminha um pacote que coleta metadados referentes ao estado dos *links* nos *switches*. Por este motivo, o pacote tem de trafegar por todos caminhos entre os comutadores, retornando para a sua origem ao final. A Figura 3.9 descreve como o pacote "sonda" realiza a captura da quantidade de *bytes* da porta de saída (*egress*) de cada equipamento. Acompanhando o caminho traçado na imagem, vê-se que a "sonda" percorre os *switches* 1, 4, 2 e 3 na primeira volta (trajeto em vermelho), retornando por 1, 3, 2, 4 e, subsequentemente, 1 novamente (destacado em azul).

Inicialmente, os cabeçalhos foram estruturados com base no que é apresentado na Figura 3.10. Foram definidos os *headers* padrão Ethernet e IPv4. Para o perfeito funcionamento da aplicação, o último tem seu tráfego verificado com base na tabela de rota LPM (*Longest Prefix Match*) [4]. Acima, formulou-se novos cabeçalhos, como o *probe*, que conta a quantidade de saltos (*hops*), e o *probe_data*, onde armazenam-se as informações coletadas em forma de pilha. Por último, temos o *probe_fwd*, que conserva a informação da interface de saída (*egress*) precedente de cada *hop* em que o pacote transitou.

Conforme explicado na seção 3.2.2.3, os programas P4 observam uma ordem de operações. A primeira delas é o *parser*, que realizará a extração dos dados dos pacotes, seguido do processamento de *ingress* e *egress* e, posteriormente, o *deparser*, passo em que os cabeçalhos são remontados. Como visto, de cada estado do *parser* deriva-se um tipo de informação diferente. No nosso programa, o analisador inicia pelo estado de *start*

[4] Algoritmo que procura o prefixo IP de destino do próximo *hop*.

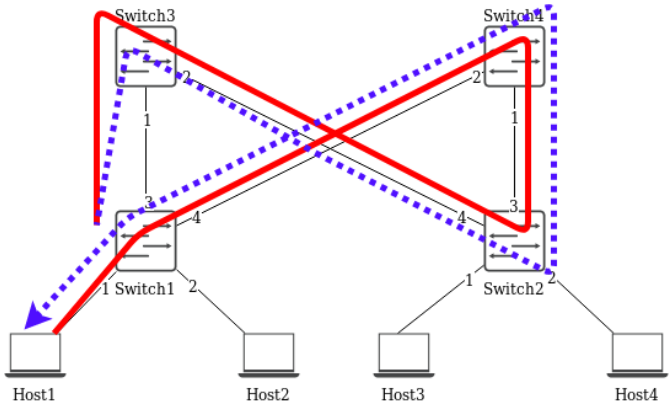

Figura 3.9: Caminho do pacote “sonda”. Fonte: Autores.

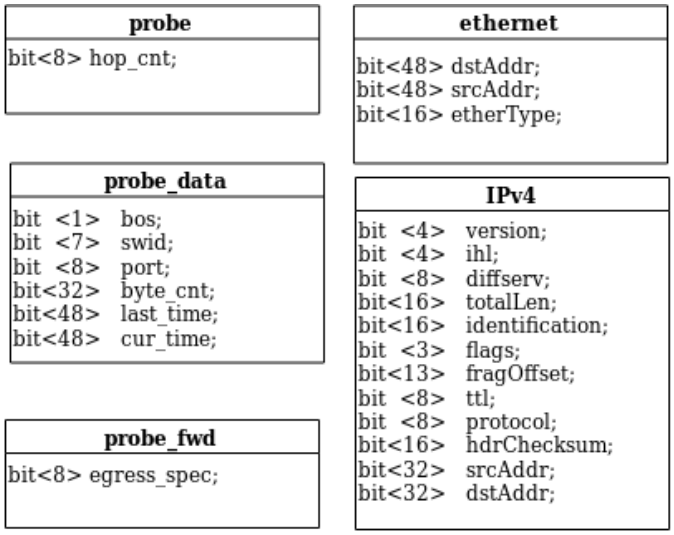

Figura 3.10: Cabeçalhos definidos neste exemplo. Fonte: Autores.

que, uma vez invocado, destina-se ao próximo estágio (que efetuará a retirada Ethernet). Conforme a Figura 3.11, percebe-se que este é um dos pontos críticos do procedimento. Caso o tipo aferido no pacote corresponder a 0x800, trata-se de um pacote IPv4 ordinário que, após ter sua informação de *header* coletada, é direcionado para o processamento de entrada (*ingress*) imediatamente. Se o tipo for 0x812, significa que é um cabeçalho *probe* (e consequentemente um pacote “sonda”), devendo deslocar-se por mais verificações. Do contrário, nenhum *header* se deduz nesta etapa, passando restritamente ao *ingress*.

Ainda na Figura 3.11, para coletar dados do *header probe* inicia-se apurando a quantidade de *hops* por onde o pacote passou. Se esse número for igual a 0, concerne-se que o pacote ainda não passou por nenhum *switch*, continuando assim para o estado de encaminhamento. Caso contrário, assume-se que o pacote adentrou previamente nas estruturas de rede, partindo para a extração do *probe_data* até que a coleta da pilha LIFO (*Last In, First Out*) chegue ao valor 1, o que equivale a leitura completa da fila, destinando-se também para o estado de encaminhamento. Neste último, é realizada a coleta do *probe_fwd*

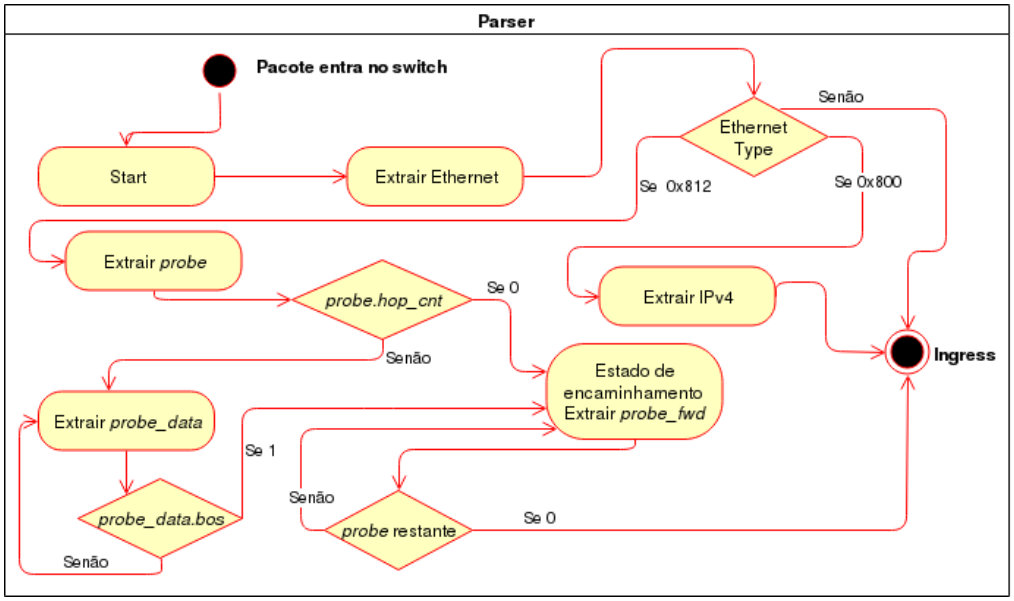

Figura 3.11: *Parser* do programa de monitoramento de *links*. Fonte: Autores.

e, ulteriormente, enviado ao *ingress*.

A execução do *ingress* caracteriza-se como o momento de entrada dos pacotes nos *switches*. Se o pacote for IPv4, suas informações de cabeçalho são transferidas para a tabela *match-action* correspondente (neste caso IPv4). Em caso de primeira execução de fluxo, a tabela consulta o P4Runtime (que orienta-se por uma tabela LPM) a fim de descobrir quais ações serão consumadas. Ilustradas na Figura 3.12, as ações condizentes apresentam dois caminhos possíveis: o encaminhamento do pacote para a porta de saída (*egress*), sofrendo incrementação do *ttl* (tempo de vida) e substituição de MAC ou então o descarte deste. Doutra feita, se o pacote contiver a função de coleta de dados do *switch* ("sonda"), empreendem-se a expedição do pacote para a porta de saída (*egress*) e incrementação da quantidade de equipamentos no cabeçalho *probe*.

A fase *egress*, pormenorizada na Figura 3.13, pode ser encarada como a mais crucial deste programa de monitoramento. Nela, além de recolhidas as informações de estado do *link*, efetivam-se os cálculos quanto a sua utilização. Para isso, logo que o pacote adentra à etapa, busca-se determinar o *timestamp* [5] atual do *egress*, assim como recuperar o da passagem anterior, com o propósito de avaliar qual foi o intervalo de tempo desde a última coleta. Um dos componentes mais importantes para a aferição do uso do *link* é o registro *byte_cnt_reg*, que se ocupa da quantidade de *bytes* trafegados, sendo persistido no plano de controle (P4Runtime). Através de uma *action* (ação P4), pode-se acessá-lo no plano de dados. Neste caso, se já houver informação de *bytes*, requesta-se o valor antigo e soma-se ao atual, modificando a variável local *byte_cnt*. Na hipótese de ser uma nova entrada, ocorre somente a escrita por meio de outra variável (*new_byte_cnt*).

Atendo-se novamente à Figura 3.13, o programa avalia se o cabeçalho é uma *probe*. Em caso positivo, o plano de dados envia uma *action* para o controle (P4Runtime) resetar o registro de *bytes* contido nele. Em oposto (ou após o *reset*), analisa-se uma vez

[5] Cadeia de caracteres que denota hora ou data de certo evento.

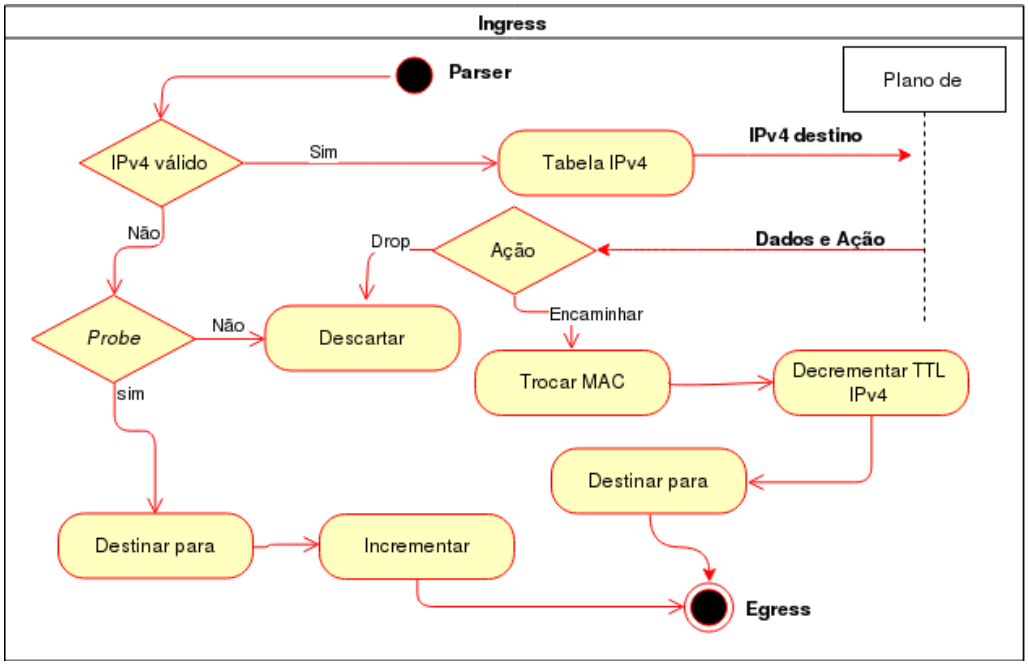

Figura 3.12: Esquema do funcionamento do *ingress* na aplicação. Fonte: Autores.

mais se o pacote é uma "sonda". Se a condição for invalidada reiteradamente, o *switch* encaminha o pacote ao *deparser*. Do contrário, tem-se a certeza que é uma *probe* válida, perfazendo-se os próximos passos. Sucessivamente, é ativado o *header* para armazenar os dados sobre a pilha (*probe_data*). Na circunstância deste ser o primeiro da fila, ativa-se o *bit* do campo *bos*, um controle que possibilitará a leitura dos metadados no *parser* sem *looping*. Ao final, há a condução para a tabela que solicita o ID do *switch* ao P4Runtime, salvando-o na estrutura.

Subsequentemente, adiciona-se o ID do *egress*, os *bytes* contabilizados no *byte _cnt_reg* e o último *timestamp* do pacote final trafegado no *switch*, bem como o *timestamp* atual. Nesta etapa, também é realizada a atualização de registro do último *timestamp* no plano de controle. Finalizando, o pacote é redirecionado ao *deparser*, que realiza a montagem dos cabeçalhos dentro do pacote. Desta forma, todas as atualizações dos *headers* anteriores desempenhadas sobre estrutura na memória do plano de dados são alocados no pacote que irá afluir pelos *switches*.

Devido à lógica do programa estar plenamente baseada sobre o processamento do *ingress* e *egress*, em anexo está a etapa do processo de entrada e saída, nas seções A.1, A.2, na devida ordem. Nota-se que os algoritmos apresentados estão de absoluto acordo com os diagramas retratados nesta seção.

3.5. Execução do projeto

Similar ao exposto na seção 3.3.2, para realizar a programação no interior do plano de dados, disponibiliza-se uma máquina virtual com os softwares necessários. Também foi criado um repositório GitHub, acessível pelo *hyperlink*[6], onde os arquivos básicos para a

[6]https://github.com/PedroEduardo68/without-code

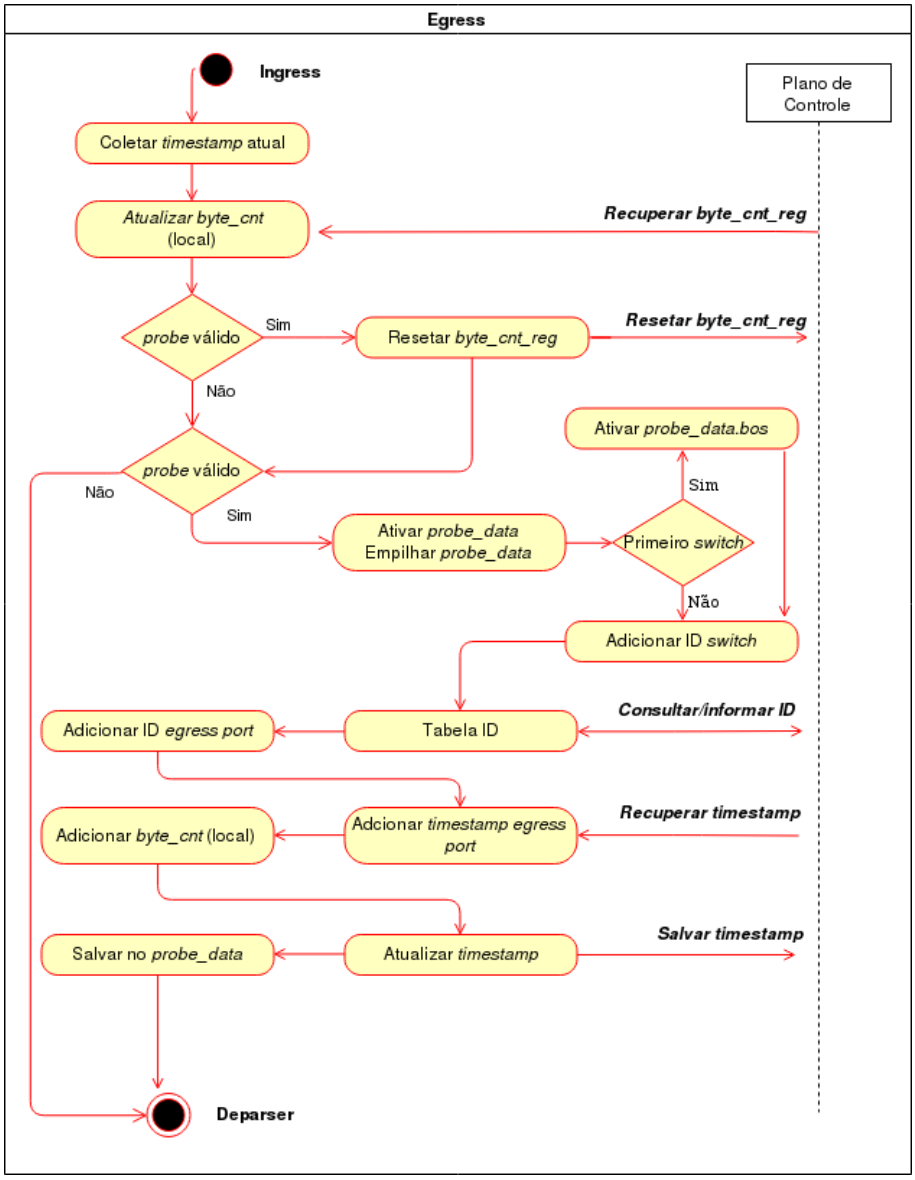

Figura 3.13: Fluxograma do *egress* adotado. Fonte: Autores.

compilação dos códigos encontram-se providenciados. Abaixo, estão descritas as etapas a serem seguidas para execução do código e do ambiente:

- **Clonar arquivos-** Primeiramente, deve-se clonar o repositório GitHub para dentro do diretório da máquina virtual, preferencialmente em `/home/p4`:

```
$cd /home/p4/
$git clone https://github.com/PedroEduardo68/without-code
```

- **Adicionar diretório-** Com o objetivo organizar os arquivos para compilação, copiar o diretório `exercises` dentro da pasta `/home/p4/tutorials/`:

```
$mv exercises /home/p4/tutorials/
```

- **Codificar-** Para codificar, conforme o detalhamento presente na seção 3.4, deve-se aceder ao diretório `/home/p4/tutorials/exercises/link_monitor`, sendo obrigatório o desenvolvimento do código dentro do arquivo `link_monitor.p4`.

Com o programa concluído, parte-se para a avaliação do estado da rede, verificando-se a transmissão do tráfego de dados pelas interfaces por meio do software P4. Ao acessar a pasta `link_monitor`, executar o comando:

```
$make run
```

O comando acima procede com a montagem da topologia de rede (*switches*, *hosts* e *links*) ao mesmo tempo que compila o arquivo `link_monitor.p4`, já incorporado dentro dos alvos de rede. Subsequentemente, abre-se o *shell* do Mininet, possibilitando operações sobre os equipamentos. Para entrar no console do Host1 duas vezes (para operar os arquivos `receive.py` e `send.py`, separadamente), usa-se:

```
$xterm h1 h1
```

O programa `receive.py` "escuta"a interface em que os pacotes com dados provenientes dos *switches* ingressam. O `send.py` os encaminha. Retornando ao *shell* do Mininet, executa-se o software `iperf` para gerar tráfego entre os computadores finais, por meio do comando:

```
$iperf h1 h4
```

Durante a execução do aplicativo gerador de trânsito, ao conectar no *prompt* do Host1 (que recebe os dados), nota-se que as interfaces estão demostrando o valor real de circulação. Com o projeto operando, para finalizar o ambiente de *testbed* Mininet, utilizam-se os comandos:

```
$quit
$make stop
$make clean
```

3.6. Conclusão

Conclui-se, pelos exemplos aqui demonstrados, a potencialidade da programabilidade no plano de dados por meio da linguagem P4. Alinhada aos conceitos SDN neste abarcados, assevera-se sua importância na transformação das redes de computadores, viabilizando o provisionamento de demandas atuais e futuras. Ao expressar os dispositivos independentemente de protocolos, o paradigma P4 quebra o enredamento imposto pelo plano de controle, até então o único elemento coordenável, resolvendo diversas dificuldades na programabilidade direta dos ativos de rede.

Buscou-se nas explanações contidas neste artigo denotar o funcionamento básico dos principais componentes P4, familiarizando o cursista às nomenclaturas e o *modus operandi* da linguagem, através da exposição de esquemas e figuras que simplificam o modelo. Com uso prático, os exemplos são facilmente assimilados, proporcionando uma rápida curva de aprendizagem e fomentando o interesse neste tipo de tecnologia. Logo, espera-se que este trabalho seja um instrumento válido de difusão do conhecimento acerca do SDN e P4 e, conseguintemente, contribua no crescimento de produções vindouras.

A. Anexos

A.1. Código do processamento de entrada

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

action drop() {
    mark_to_drop(standard_metadata);
}

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 = drop();
}

apply {
    if (hdr.ipv4.isValid()) {
        ipv4_lpm.apply();
    }
    else if (hdr.probe.isValid()) {
        standard_metadata.egress_spec = (bit<9>)meta.egress_spec;
        hdr.probe.hop_cnt = hdr.probe.hop_cnt + 1;
    }
}
```

A.2. Código do processamento de saída

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

register<bit<32>>(MAX_PORTS) byte_cnt_reg;
register<time_t>(MAX_PORTS) last_time_reg;

action set_swid(bit<7> swid) {
    hdr.probe_data[0].swid = swid;
}
table swid {
    actions = {
        set_swid;
        NoAction;
    }
    default_action = NoAction();
}

apply {
    bit<32> byte_cnt;
    bit<32> new_byte_cnt;
    time_t last_time;
    time_t cur_time = standard_metadata.egress_global_timestamp;
    byte_cnt_reg.read(byte_cnt, (bit<32>)standard_metadata.egress_port);
    byte_cnt = byte_cnt + standard_metadata.packet_length;
    new_byte_cnt = (hdr.probe.isValid()) ? 0 : byte_cnt;
    byte_cnt_reg.write((bit<32>)standard_metadata.egress_port, new_byte_cnt);

    if (hdr.probe.isValid()) {
        hdr.probe_data.push_front(1);
        hdr.probe_data[0].setValid();

        if (hdr.probe.hop_cnt == 1) {
            hdr.probe_data[0].bos = 1;
        }
        else {
            hdr.probe_data[0].bos = 0;
        }
        swid.apply();
        hdr.probe_data[0].port = (bit<8>)standard_metadata.egress_port;
        hdr.probe_data[0].byte_cnt = byte_cnt;
        last_time_reg.read(last_time, (bit<32>)standard_metadata.egress_port);
        last_time_reg.write((bit<32>)standard_metadata.egress_port, cur_time);

        hdr.probe_data[0].last_time = last_time;
        hdr.probe_data[0].cur_time = cur_time;
    }
```

}

Referências

[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.

[Gibb et al. 2013] Gibb, G., Varghese, G., Horowitz, M., and McKeown, N. (2013). Design principles for packet parsers. In *Architectures for Networking and Communications Systems*, pages 13–24, San Jose, CA, USA. IEEE.

[Gong et al. 2015] Gong, Y., Huang, W., Wang, W., and Lei, Y. (2015). A survey on software defined networking and its applications. *Frontiers of Computer Science*, 9(6):827–845.

[Haleplidis et al. 2015] Haleplidis, E., Hadi Salim, J., Denazis, S., and Koufopavlou, O. (2015). Towards a network abstraction model for sdn. *Journal of Network and Systems Management*, 23(2):309–327.

[Hancock and van der Merwe 2016] Hancock, D. and van der Merwe, J. (2016). Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 35–49, New York, NY, USA. ACM.

[Huang et al. 2016] Huang, T., Yu, F. R., and Liu, Y.-j. (2016). Special issue on future network: Software-defined networking. *Frontiers of Information Technology & Electronic Engineering*, 17(7):603–605.

[Kreutz et al. 2015] Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. In *Proceedings of the IEEE*, volume 105, pages 14–76, Singapore. IEEE Computer Society.

[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.

[OPEN NETWORK FOUNDATION 2016] OPEN NETWORK FOUNDATION (2016). Sdn architecture - a primer. Disponível em: <https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2013/05/7-26%20SDN%20Arch%20Glossy.pdf. Acessado: 19-08-2019.

[Santiago da Silva et al. 2018] Santiago da Silva, J., Boyer, F.-R., and Langlois, J. P. (2018). P4-compatible high-level synthesis of low latency 100 gb/s streaming packet parsers in fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 147–152, New York, NY, USA. ACM.

[Singh and Jha 2017] Singh, S. and Jha, R. K. (2017). A survey on software defined networking: Architecture for next generation network. *Journal of Network and Systems Management*, 25(2):321–374.

[Sood and Xiang 2017] Sood, K. and Xiang, Y. (2017). The controller placement problem or the controller selection problem? *Journal of Communications and Information Networks*, 2(3):1–9.

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