Capítulo 3

10 Anos de P4: Explorando a Programação no Plano de Dados

Dener Edson Ottolini Guedes da Silva, Lucas Trombeta, Bruna Cunha de Carvalho, Alexandre Heideker, Felipe Augusto Anon da Silva, Marcelo Antonio Marotta, João Henrique Kleinschmidt, Lisandro Zambenedetti Granville, Carlos Alberto Kamienski

Abstract

This mini-course explores the power of data plane programming with the P4 language, using the virtualized P4Docker solution. Participants learn how to create customized and efficient networks in an accessible and practical way. Software-Defined Networks (SDN) introduced a modular architecture that separates the control plane from the data plane, promoting innovation and simplifying the management of complex infrastructures. However, the evolution of the data plane faced challenges due to the difficulty of reprogramming hardware and the high criticality of performance. The P4 language and devices such as the Intel Tofino revolutionized this scenario, allowing greater customization and efficiency, although the high cost limited its initial adoption. With P4Docker, students can access a simple and efficient development environment to experiment and learn. The course will cover the fundamentals of the P4 language, SDN architecture, creating packet pipelines, implementing security policies, and using P4Docker. Upon completion, participants are expected to develop network applications, optimize infrastructure performance, and contribute to more intelligent and flexible networks.

Resumo

Este minicurso explora o poder da programação no plano de dados com a linguagem P4, utilizando a solução virtualizada P4Docker. Os participantes aprendem a criar redes personalizadas e eficientes de maneira acessível e prática. As Redes Definidas por Software (SDN) introduziram uma arquitetura modular que separa o plano de controle do plano de dados, promovendo inovação e simplificando a gestão de infraestruturas complexas. Entretanto, a evolução do plano de dados enfrentou desafios devido à dificuldade

de reprogramação de hardware e à alta criticidade de desempenho. A linguagem P4 e dispositivos como o Intel Tofino revolucionaram esse cenário, permitindo maior personalização e eficiência, embora o custo elevado tenha restringido sua adoção inicial. Com o P4Docker, os alunos têm acesso a um ambiente de desenvolvimento simples e eficiente para experimentar e aprender. O curso aborda fundamentos da linguagem P4, a arquitetura SDN, criação de pipelines de pacotes, implementação de políticas de segurança e uso do P4Docker. Ao final, os participantes estão aptos a desenvolver aplicações de rede, otimizar o desempenho de infraestruturas e contribuir para redes mais inteligentes e flexíveis.

1. Introdução

O movimento de softwarização de redes representa uma mudança de paradigma na forma como as infraestruturas de rede são projetadas, gerenciadas e operadas. Esse conceito emerge em resposta à rigidez das arquiteturas tradicionais, que dependiam de equipamentos de rede com funcionalidades fixas e protocolos padronizados, dificultando a inovação e a experimentação.

A partir do artigo publicado por [McKeown et al. 2008], abre-se um vasto campo de pesquisa na área de redes de computadores, introduzindo o OpenFlow e definindo a separação do plano de controle do plano de dados. O próximo movimento parte também do mesmo grupo de pesquisadores com o trabalho de [Bosshart et al. 2014], explorando agora a possibilidade de programar o plano de dados, melhorando o suporte do plano de dados ao plano de controle, complementando e expandindo ainda mais a programabilidade da rede.

Esta seção explora o movimento de softwarização da rede, em especial aos 10 anos desde o trabalho seminal em P4, apresentando os conceitos e avanços alcançados a partir destes dois marcos na área de redes de computadores.

1.1. Redes Definidas por Software (SDN)

Durante a década de 2010-2020 observou-se o movimento de virtualização, consolidando a computação em nuvem como padrão em infraestrutura de computação e telecomunicações. Um grande exemplo deste movimento é a tecnologia de comunicação celular 5G, desenvolvida com seus alicerces na virtualização e "softwarização"de redes ([Blanco et al. 2017]). Este movimento foi alavancado por iniciativas como as *Software Defined Networking* (SDN) ([McKeown et al. 2008]), que desacopla os planos de controle e de dados nos dispositivos de encaminhamento, ou seja, dos switches. Esta separação, ilustrada na Figura 3.1, permite migrar o controle que, anteriormente, era fortemente ligado aos dispositivos de rede individuais para dispositivos de computação acessíveis, permitindo que a infraestrutura subjacente seja abstraída para aplicativos e serviços de rede, tratando assim a rede como uma entidade lógica ou virtual ([Kaur et al. 2025]), propondo assim uma nova arquitetura de redes de computadores.

Além disso, a arquitetura SDN ainda tem a capacidade de simplificar os próprios dispositivos de rede, uma vez que eles não têm a necessidade de prover interfaces para operadores e administradores de rede, tão pouco implementar complexas regras de encaminhamento de pacotes, somente executar as instruções dos controladores de SDN ([Makde and Laxkar 2024]).

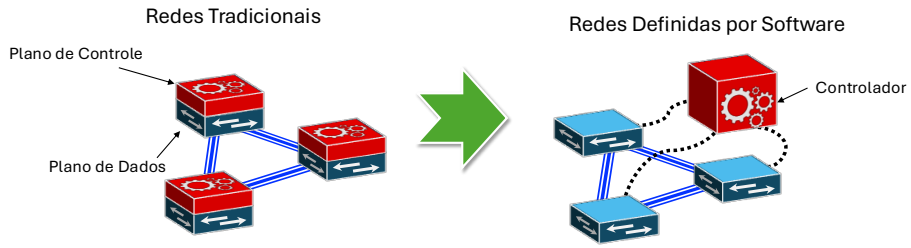


Figura 3.1: Nas redes de computadores tradicionais o controle dos dispositivos é distribuído e legado a cada modelo, fabricante e geração. Nas SDN, este controle é delegado ao Controlador.

Ressalta-se ainda que os operadores e os administradores de rede podem configurar de maneira programática essa abstração de rede, sem a necessidade de acesso individual a dezenas ou até milhares de dispositivos, físicos ou virtuais. Além disso, a inteligência centralizada no controlador SDN permite alterar o comportamento da rede em tempo real, implantando novos aplicativos e serviços de rede em pouco tempo.

Além de abstrair a rede, a arquitetura SDN pode implementar serviços como roteamento, multicast, segurança, controle de acesso, gerenciamento de largura de banda, engenharia de tráfego, qualidade de serviço, otimização de processador e armazenamento, uso de energia e todas as formas de gerenciamento de políticas, além de fornecer APIs que permitem a implementação de serviços personalizados e interação com aplicativos dedicados a atender aos objetivos de negócios. Desta maneira, os aplicativos de negócios podem operar em uma abstração da rede, aproveitando serviços e recursos disponíveis, sem estarem presos aos detalhes de sua implementação.

1.2. O Protocolo OpenFlow

O OpenFlow é o primeiro protocolo de comunicação padrão definido entre as camadas de controle e encaminhamento de uma arquitetura SDN. O OpenFlow permite o acesso direto e manipulação do plano de encaminhamento de dispositivos de rede como switches e roteadores, tanto físicos quanto virtuais. O OpenFlow usa o conceito de fluxos para identificar o tráfego de rede com base em regras de correspondência predefinidas que podem ser programadas estática ou dinamicamente pelo software de controle (SDN[McKeown et al. 2008]). Este conceito permite aos operadores e administradores de rede definirem como o tráfego que deve fluir através dos dispositivos de rede com base nos padrões de uso, aplicativos, recursos de nuvem, entre outros.

SDN utilizando OpenFlow permite o controle extremamente granular do tráfego, que pode responder a mudanças em tempo real nos níveis de aplicativo, usuário e sessão ([Alsaeedi et al. 2019]) - roteamento baseado em IP não fornece esse nível de controle, uma vez que, todos os fluxos entre os dois *endpoints* devem seguir o mesmo caminho pela rede, independentemente de seus diferentes requisitos.

O protocolo OpenFlow é tido como um facilitador fundamental para as redes definidas por software e, atualmente, é o único protocolo SDN padronizado que permite a manipulação direta do plano de encaminhamento de dispositivos de rede. Embora inicialmente aplicado a redes Ethernet, a comutação OpenFlow pode se estender a um

conjunto muito mais amplo de casos de uso ([Braun and Menth 2014]). A arquitetura SDN baseada em OpenFlow pode ser implantada em redes físicas e virtuais, suportando simultaneamente o encaminhamento com base em OpenFlow, bem como encaminhamento tradicional, facilitando a introdução desta nova abordagem mesmo em ambientes de rede com vários fornecedores.

Para a indústria e operadores de telecomunicações, as SDNs podem representar um diferencial competitivo, melhorando o aproveitamento da largura de banda disponível, se alinhando às demandas da natureza dinâmica dos aplicativos, reduzindo significativamente a complexidade no gerenciamento da rede. De forma sucinta, podemos resumir os benefícios de uma arquitetura SDN com base em OpenFlow em:

- **Controle centralizado de ambientes de vários fornecedores:** o software de controle SDN pode controlar dispositivo com suporte à OpenFlow de diferentes fabricantes, incluindo switches, roteadores e switches virtuais, utilizando ferramentas de orquestração e gerenciamento para implantar, configurar e modificar o comportamento dos dispositivos rapidamente em toda a rede.
- **Complexidade reduzida através da automação:** Sua estrutura flexível de automação e gerenciamento de rede possibilita o desenvolvimento de ferramentas que automatizem diversas tarefas de gerenciamento que são feitas manualmente, diminuindo a instabilidade da rede introduzida por erros de configuração.
- **Maior taxa de inovação:** a possibilidade de interagir diretamente com a rede, modificando seu comportamento de forma programática, permite a experimentação rápida de novas ideias, acelerando o desenvolvimento de aplicações e protocolos de rede.
- **Maior confiabilidade e segurança da rede:** a arquitetura SDN permite que os administradores da rede determinem as configurações de alto nível e políticas que são traduzidas para a infraestrutura via OpenFlow, eliminando a necessidade de configurar individualmente os dispositivos de rede sempre que um ponto final, serviço ou aplicativo é adicionado ou movido, ou uma política é alterada, reduzindo a possibilidade de falhas de rede devido a inconsistência de configuração ou política.
- **Maior controle de rede granular:** o modelo de controle baseado no fluxo do OpenFlow permite políticas em um nível mais granular, incluindo níveis de sessão, usuário, dispositivos e aplicativos, de forma altamente abstrata e automatizada. Esse controle permite que os operadores de nuvem ofereçam suporte à multilocação, mantendo o isolamento do tráfego, a segurança e a flexibilidade no gerenciamento de recursos quando os clientes compartilham da mesma infraestrutura.
- **Melhor experiência do usuário:** ao centralizar o controle da rede e tornar as informações de estado disponíveis para aplicativos de nível superior, uma infraestrutura SDN adapta-se melhor às necessidades dinâmicas do usuário. Atualmente, os usuários devem selecionar de maneira explícita uma configuração de resolução, que a rede pode ou não ser capaz de suportar, resultando em atrasos e interrupções que acabam com a experiência do usuário. Como exemplo, um aplicativo de vídeo

pode identificar a largura da banda larga disponível na rede em tempo real e ajustar automaticamente a resolução do vídeo de acordo com essa informação.

Juntamente com os benefícios apresentados pela arquitetura SDN e a adoção do protocolo OpenFlow, surge a pressão por um suporte mais amplo do plano de dados às novas versões do mesmo. Apesar do plano de controle desacoplado tornar mais versátil a manipulação do tráfego, este deve ser plenamente suportado pela camada adjacente, ou seja, o plano de dados. O próximo passo no processo de softwarização das redes é explorar a programabilidade do plano de dados.

1.3. Linguagem P4 (Programming Protocol-independent Packet Processors)

A crescente demanda por redes mais flexíveis, inteligentes e adaptáveis tem impulsionado a pesquisa em tecnologias que promovem maior controle sobre o comportamento da infraestrutura de rede. Nesse contexto, a programação do plano de dados se destaca como um tema emergente, interdisciplinar por natureza, abrangendo áreas como arquitetura de redes, engenharia de software, segurança cibernética, desempenho de sistemas e aplicações voltadas à Internet das Coisas (IoT). Diferente do modelo tradicional baseado em equipamentos com funções fixas, esse novo paradigma promove o ideal de redes personalizáveis, em que o comportamento dos dispositivos pode ser modificado conforme os requisitos do ambiente ([Bosshart et al. 2014, Silva et al. 2024, P4 Language Consortium 2024]).

O uso da programação do plano de dados surge como uma estratégia para contornar as deficiências do OpenFlow. Abordagens como o P4 permitem descrever um comportamento agnóstico de encaminhamento, independentemente do protocolo utilizado. Por ser uma linguagem de alto nível, o P4 fornece um dialeto de rede simples para descrever o caminho dos datagramas, podendo operar em conjunto com protocolos do plano de controle em arquiteturas SDN ([Bosshart et al. 2014, Heideker et al. 2023]).

A linguagem P4 (Programming Protocol-independent Packet Processors - [P4 Language Consortium 2
descreve como um pacote deve ser processado pelo plano de dados de um dispositivo programável de encaminhamento, que pode estar implementado em switches, placas de rede, roteadores ou dispositivos FPGA. Embora tenha sido inicialmente projetada para switches programáveis, seu escopo foi ampliado e passou a abranger diferentes tipos de dispositivos — referidos como “targets” pela especificação da linguagem. O P4 é utilizado para descrever tanto o comportamento interno do plano de dados quanto sua interface de comunicação com o plano de controle. Vale destacar que a linguagem não é voltada à implementação do próprio plano de controle, sendo seu foco exclusivo o plano de dados [P4 Language Consortium 2022].

O uso de P4 em ambientes introduz mecanismos para otimização de recursos, aprimoramento da segurança por meio da detecção e resposta a incidentes em tempo real, e maior flexibilidade na gestão da heterogeneidade de protocolos em gateways e dispositivos de borda. Isso permite melhorar significativamente a eficiência, a segurança e a interoperabilidade em contextos distribuídos, especialmente em aplicações críticas como cidades inteligentes, agricultura de precisão e saúde conectada. Tais avanços são fundamentais para o desenvolvimento de soluções robustas que não apenas endereçam lacunas existentes, mas também preparam o terreno para inovações futuras, promovendo uma gestão mais eficaz e segura de redes vastas e complexas ([Santos et al. 2021, Heideker et al. 2023]).

A adoção de P4 traz uma série de benefícios e alguns desafios que precisam ser considerados ao projetar soluções baseadas em redes programáveis ([Kfoury et al. 2021a, Laki et al. 2021]):

- Prós:
  - Permite a criação de protocolos personalizados, não limitados aos suportados pelo hardware tradicional.
  - Proporciona controle granular sobre o processamento de pacotes.
  - Garante independência de protocolo e de plataforma.
  - Suporta otimizações específicas para cenários como IoT, redes de baixa latência e segurança em tempo real.
  - Viabiliza experimentações com novos algoritmos de encaminhamento e políticas de QoS.
- Desafios
  - Curva de aprendizado acentuada para desenvolvedores.
  - Baixa disponibilidade de hardware P4 no mercado, especialmente para ambientes produtivos.
  - Dificuldades de integração com dispositivos legados.
  - Necessidade de ferramentas de depuração e visualização ainda em amadurecimento.

Enquanto o SDN, especialmente via OpenFlow, promoveu a descentralização e o controle programável da rede no plano de controle, a linguagem P4 avança essa proposta ao permitir que o plano de dados também seja programável. Dessa forma, P4 e SDN/OpenFlow são tecnologias complementares: o primeiro amplia a flexibilidade no processamento dos pacotes em cada switch, enquanto o segundo coordena globalmente as decisões de encaminhamento. Essa sinergia favorece o desenvolvimento de redes verdadeiramente inteligentes, capazes de responder dinamicamente a mudanças de tráfego, ameaças de segurança e requisitos de aplicações distribuídas.

2. Programação para o Plano de Dados

A linguagem P4 foi desenvolvida para oferecer uma forma flexível e independente de hardware para definir o comportamento de dispositivos de rede no plano de dados. Para isso, adota-se uma organização modular baseada no modelo PISA (Protocol-Independent Switch Architecture - [Bosshart et al. 2014]), no qual o processamento de pacotes é dividido em três estágios principais: análise (parser), aplicação de regras (tabelas de correspondência e ação) e reordenação (deparser).

Além desse modelo abstrato, implementações práticas como o V1Model — utilizado no simulador BMv2 ([Consortium 2024]), uma solução de código aberto criada para estudo do comportamento de redes e protocolos em um ambiente controlado — detalham o fluxo completo de um pacote em um switch P4. Essa arquitetura inclui os estágios de

entrada (ingress), saída (egress), buffers e reprocessamento, além das interações entre parser, deparser e os pipelines. A Figura 3.2 ilustra esse processo interno de funcionamento de um switch P4, destacando os principais blocos que compõem o caminho do pacote desde sua chegada até o envio final.

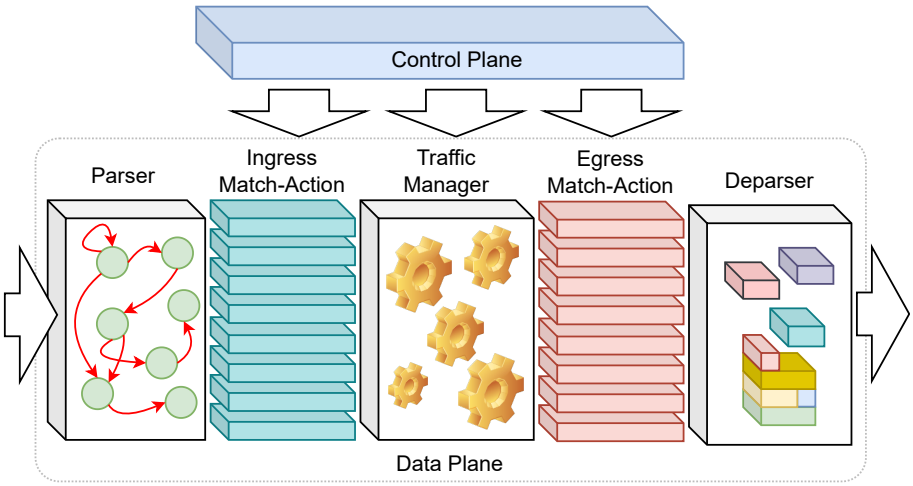


Figura 3.2: Arquitetura do Switch P4 baseado no V1Model ([Braun and Menth 2014])

Cada programa em P4 é composto por um conjunto de componentes, como mostrado no código 3.1, que definem precisamente o comportamento do dispositivo programável. Esses componentes incluem:

- Definição de *headers*: estruturas que descrevem os campos e formatos dos protocolos que o dispositivo será capaz de interpretar.
- Parser: sequência lógica para extração dos *headers*, utilizando transições condicionais entre estados.
- Tabelas de correspondência: associam combinações de campos a ações específicas, com base em critérios como endereço IP, porta ou tipo de protocolo.
- Ações: blocos de código que implementam o que deve ser feito quando uma determinada entrada é combinada.
- Fluxo de controle: lógica que conecta as tabelas e define o caminho que o pacote percorre no pipeline.
- *Deparser*: define a ordem de reescrita dos `headers` antes do envio do pacote.

```
/* Definicao de Headers */
header ethernet_t {
    mac_addr dstAddr;
    mac_addr srcAddr;
    bit<16>  etherType;
}

/* Estrutura do pacote */
struct headers {
    ethernet_t ethernet;
}
```

```
12
13 /* Parser */
14 parser MyParser(packet_in packet,
15                 out headers hdr) {
16     state start {
17         packet.extract(hdr.ethernet);
18         transition accept;
19     }
20 }
21
22 /* Acoes */
23 action forward(mac_addr dst, bit<9> port) {
24     hdr.ethernet.dstAddr = dst;
25     standard_metadata.egress_spec = port;
26 }
27
28 /* Tabela */
29 table forwarding {
30     key = {
31         hdr.ethernet.dstAddr : exact;
32     }
33     actions = {
34         forward;
35         drop;
36     }
37     size = 1024;
38 }
39
40 /* Controle de entrada */
41 control Ingress(inout headers hdr,
42                 inout standard_metadata_t standard_metadata) {
43     apply {
44         forwarding.apply();
45     }
46 }
47
48 /* Deparser */
49 control MyDeparser(packet_out packet, in headers hdr) {
50     apply {
51         packet.emit(hdr.ethernet);
52     }
53 }
54
55 /* Programa principal */
56 control MySwitch(...) {
57     MyParser() parser;
58     Ingress() ingress;
59     MyDeparser() deparser;
60
61     apply {
62         parser.apply();
63         ingress.apply();
64         deparser.apply();
65     }
66 }
```

Listing 3.1: Exemplo básico de programa P4

2.1. Independência do Hardware (abstrações)

Uma das principais vantagens da linguagem P4 é a capacidade de descrever o comportamento de dispositivos de rede de forma independente do hardware utilizado. Tradicionalmente, o plano de dados era implementado em hardware dedicado, um circuito integrado para aplicação específica (ASIC), que oferece desempenho elevado, mas com pouca ou nenhuma flexibilidade. Cada fabricante adota sua própria lógica de processamento e suporte a protocolos, o que dificulta a portabilidade de soluções entre equipamentos diferentes ([Kfoury et al. 2024]).

A proposta de abstração do hardware em P4 é viabilizada por arquiteturas intermediárias, como a PISA, que permite que o desenvolvedor defina como os pacotes devem ser tratados sem depender da implementação física do dispositivo de rede. Para que isso funcione na prática, cada dispositivo programável implementa um conjunto de funcio-

nalidades compatíveis com o modelo P4. Esses dispositivos são chamados de *targets*, e podem incluir desde switches programáveis comerciais, como o Intel Tofino ([Intel 2022]), emuladores em software como o BMv2 ([Consortium 2024]), ou switches e placas de rede de hardware customizável utilizando *Field Programmable Gate Array* (FPGA). A independência de hardware permite que o mesmo código P4 possa ser compilado para diferentes *targets*, com ajustes mínimos ou inexistentes.

Essa abordagem traz vantagens como maior portabilidade, reutilização de código, liberdade de escolha entre diferentes fornecedores de hardware e redução do acoplamento entre software e infraestrutura. Além disso, contribui para a adoção de arquiteturas mais abertas e modulares, o que favorece o desenvolvimento e a inovação em redes programáveis ([Silva et al. 2024, Heideker et al. 2023]).

2.2. Programas-alvo (*target*)

Em um programa P4, o termo *target* refere-se ao dispositivo de rede onde o código será executado. Esse dispositivo pode ser um simulador, uma placa física ou até um componente virtual em ambientes de nuvem. O objetivo do P4 é garantir que o comportamento da lógica do plano de dados possa ser descrito abstratamente, mas que seja compatível com diferentes tipos de implementação prática.

Cada *target* é responsável por mapear o programa P4 para uma infraestrutura específica. Por exemplo, no caso do simulador BMv2, a arquitetura V1Model define um pipeline fixo que guia o comportamento esperado. Já em hardware real, como os switches baseados no Intel Tofino, a implementação segue o modelo PISA, mas com otimizações de alto desempenho e limitações em termos de operações suportadas ([Intel 2022]).

Além desses, é possível utilizar P4 em plataformas baseadas em FPGA e em placas de rede inteligentes (SmartNICs), como NetFPGA e NVIDIA BlueField. Em cada caso, o compilador P4 se adapta ao conjunto de funcionalidades e restrições oferecido pelo target, garantindo que o comportamento descrito no código seja preservado dentro do que o hardware permite ([Wang et al. 2017, Ibanez et al. 2019]).

Essa flexibilidade permite a criação de protótipos realistas em ambientes simulados e sua posterior migração para produção em hardware especializado. Também viabiliza o uso de P4 em cenários educacionais, pesquisa e aplicações industriais, sem que seja necessário reescrever o programa para cada novo ambiente ([Silva et al. 2024, Trombeta et al. 2024]).

Os *targets* também definem o conjunto de metadados disponíveis, o tamanho das tabelas, suporte a tipos de correspondência e a granularidade das ações permitidas. Portanto, embora a linguagem seja padronizada, é papel do compilador e da arquitetura alvo garantir que o programa seja compatível com as capacidades do dispositivo ([P4 Language Consortium 2024]).

2.3. P4Runtime

Apesar de a linguagem P4 ser voltada exclusivamente para o plano de dados, a configuração e o controle dinâmico dos dispositivos programáveis exigem uma interface entre o plano de controle e os *targets*. O P4Runtime surge como essa interface padronizada, permitindo

que controladores externos gerenciem tabelas, inserção de regras, metadados e ações em tempo de execução ([Consortium 2023]).

Ao contrário de abordagens estáticas, em que o comportamento do dispositivo é fixo após a compilação do programa, o P4Runtime permite a atualização da lógica de encaminhamento em tempo real. Essa flexibilidade é essencial para arquiteturas baseadas em SDN, onde decisões de roteamento, balanceamento de carga, segurança e gerenciamento de rede são tomadas de forma centralizada, mas aplicadas nos planos de dados distribuídos ([Vuckovic et al. 2021]).

O P4Runtime é implementado como uma API baseada em gRPC (*Google Remote Procedure Call*), o que garante interoperabilidade com múltiplas linguagens e facilita a integração com controladores como ONOS, P4Controller e *frameworks* experimentais. Ele permite, por exemplo, adicionar ou remover entradas de tabelas, configurar contadores, e definir regras de clonagem e *multicast*, sem a necessidade de reconfigurar o dispositivo completamente [Consortium 2023].

O modelo de comunicação adotado pelo P4Runtime é bidirecional, suportando também mensagens de *feedback* dos switches para o controlador, como notificações de eventos, relatórios de erros e exportação de métricas. Com isso, é possível criar ciclos de controle mais inteligentes e reativos, aproximando ainda mais o comportamento das redes programáveis da lógica de sistemas distribuídos [Consortium 2023, Ion et al. 2020].

Considerando uma tabela `table_forwarding`, como listado no código 3.1, que realiza o encaminhamento de pacotes com base no endereço IP de destino, o seguinte comando utilizando a API P4Runtime (em Python, com `p4runtime-shell`), insere dinamicamente uma nova entrada na tabela:

```
te = table_entry["MyIngress.ipv4_lpm"](ipv4_dst="10.0.2.2/32")
te.action = action["MyIngress.forward"](port=2)
te.insert()
```

Esse comando configura o switch programável para que todos os pacotes com destinados ao endereço `10.0.2.2` sejam encaminhados pela porta 2. Internamente, o plano de dados atualiza a tabela `table_forwarding` com a nova entrada de correspondência e a ação associada, sem necessidade de recompilar o programa P4.

3. Implementações P4

A linguagem P4 possui uma série de implementações práticas que viabilizam seu uso tanto em ambientes de simulados quanto em hardware *bare metal*. Conforme introduzido na seção anterior, tais implementações abrangem desde switches virtuais e simuladores até plataformas de rede programáveis baseadas em FPGA e SmartNICs.

3.1. ASIC Switch: Intel Tofino

A Figura 3.3 apresenta uma comparação conceitual entre o funcionamento de dispositivos de rede com ASICs tradicionais e os baseados em ASICs programáveis. Em um switch tradicional, o caminho do pacote é rigidamente definido por circuitos e tabelas específicas para protocolos previamente implementados em hardware, como Ethernet, VLANs e listas de controle de acesso (ACLs). O parser é fixo e opera de acordo com uma sequência pré-configurada de protocolos, sem possibilidade de adaptação

([Bosshart et al. 2014, Intel 2022]). Por outro lado, em um dispositivo com ASIC programável, como o Intel Tofino, o parser e as tabelas de encaminhamento são definidos por meio de um programa em P4. Isso permite ao desenvolvedor configurar dinamicamente quais protocolos serão suportados, quais campos dos *headers* serão extraídos e como serão interpretados. A sequência de processamento também é determinada pelo próprio programa, o que possibilita a adaptação a novos cenários de rede sem a necessidade de modificar o hardware ([Franco et al. 2024, Intel 2022]). Essa flexibilidade é particularmente útil em ambientes que exigem suporte a protocolos emergentes, encapsulamentos personalizados ou políticas de segurança e roteamento sob demanda. O uso de ASICs programáveis representa, portanto, uma mudança de paradigma: de redes com lógica fixa para redes adaptáveis, controladas por software ([Franco et al. 2024, Intel 2022]).

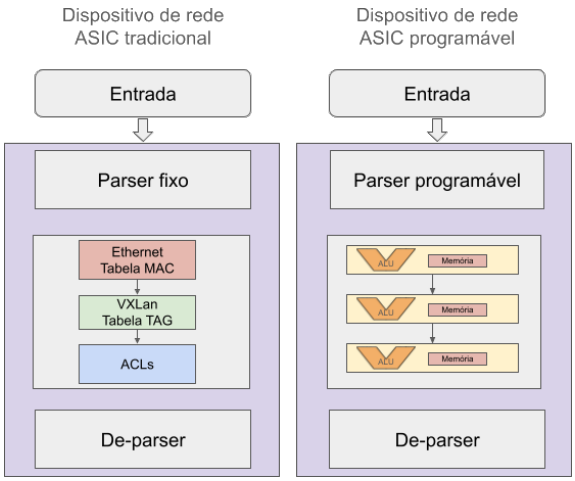


Figura 3.3: Comparação entre processamento em plano de dados entre dispositivo de rede tradicional e dispositivo de rede programável

3.2. SmartNICs

Com a constante evolução na tecnologia das placas de rede, parte das tarefas antes realizadas pelo sistema operacional passou a ser realizada pelos adaptadores de rede, por exemplo, cálculo de checksum, TOE (*TCP offload engine*), entre outras, com o objetivo de obter maior desempenho dos sistemas, uma vez que não tinham mais que performar essas funções utilizando a CPU. Posteriormente houve o desenvolvimento do chamado *SmartNIC* ([Luizelli et al. 2024]), onde a placa de rede passou a contar com dispositivo de processamento programáveis (FPGAs, ASICs). Ainda no escopo dos *SmartNICs*, podemos separar ainda aqueles que são mais sofisticados, possuindo núcleos de CPU e sistema operacional próprios, e aqueles sem essa configuração ([Kfoury et al. 2024]).

Uma configuração popular dentre os *SmartNICs* com sistema operacional próprio e CPU integrada é a utilização de processadores de arquitetura ARM, com as mais variadas configurações dentre quantidade de núcleos e velocidade das interfaces ([Kfoury et al. 2024]). Um exemplo de mercado desta nova arquitetura de SmartNICs é apresentada pela empresa NVIDIA, denominando sua linha de produtos deste tipo como DPU (*Data Center Processing Unit* - [Burstein 2021]). Tais DPUs possuem sistema operacional próprio, e núcleos ARM para processamento, utilizando um SDK (*Software De-*

velopment Kit) proprietário chamado DOCA para criar aplicativos que rodem no SmartNIC. O DOCA *P4 Developer Tools*, é uma *suite* que permite o desenvolvimento e depuração de programas P4 carregados na DPU Bluefield.

3.3. Switch virtual: BMv2 e SONiC

Por ser executado em software, o BMv2 apresenta limitações em relação ao desempenho, não sendo adequado para ambientes de produção. No entanto, sua simplicidade, flexibilidade e integração com ferramentas como Mininet, P4Runtime, P4Docker e controladores SDN o tornam uma ferramenta indispensável para aprendizado e testes funcionais. Já o *Software for Open Networking in the Cloud* (SONiC) é um sistema operacional de rede de código aberto, originalmente desenvolvido pela Microsoft. Embora não tenha sido projetado exclusivamente para uso com P4, versões recentes do SONiC têm oferecido suporte à execução de programas P4 em ambientes virtuais, incluindo integração com o SAI (*Switch Abstraction Interface*) e suporte a P4Runtime ([Microsoft 2023]). Com isso, o SONiC se posiciona como uma opção viável para explorar o uso de P4 em ambientes mais próximos dos cenários de produção, especialmente quando associado a switches virtuais ou ambientes baseados em containers e máquinas virtuais (VM). Sua estrutura modular, baseada em Docker, permite a integração de serviços de roteamento, gerenciamento e monitoramento em uma plataforma extensível. A combinação das duas abordagens oferece um espectro de possibilidades: do ensino e prototipagem com BMv2 até a experimentação mais próxima de ambientes reais com SONiC.

3.4. Emulador de hardware: P4Pi

O P4Pi é uma iniciativa que visa democratizar o acesso ao desenvolvimento com a linguagem P4, tornando possível a execução de programas de rede programável em dispositivos de baixo custo, como o Raspberry Pi. Essa plataforma permite transformar o Raspberry Pi em um switch P4 funcional, com suporte ao pipeline descrito em P4 e integração com ferramentas como BMv2 e P4Runtime ([Laki et al. 2021]).

A arquitetura do P4Pi consiste em utilizar o BMv2 como plano de dados, executado localmente no Raspberry Pi, acompanhado de interfaces de rede USB ou adaptadores Ethernet. Dessa forma, é possível conectar fisicamente outros dispositivos à placa e realizar testes reais de encaminhamento, telemetria, filtragem de pacotes e outras funcionalidades descritas no programa P4 ([Heideker et al. 2023]).

Embora limitado em desempenho por conta do hardware embarcado, o P4Pi é ideal para aplicações didáticas, oficinas, demonstrações e ambientes onde o custo e a portabilidade são fatores importantes. Além disso, sua flexibilidade permite configurar o ambiente com múltiplas interfaces, diferentes arquiteturas de rede e integração com controladores externos ([Laki et al. 2021]).

3.5. Simulador: Mininet e Mininet-WiFi

O Mininet é uma das ferramentas de emulação de redes mais consolidadas, amplamente utilizada em ambientes acadêmicos e industriais para pesquisa em SDN e experimentação de topologias virtuais. Ele permite a criação de redes completas, compostas por switches, hosts e controladores, executadas como processos isolados dentro de um mesmo kernel

Linux, utilizando `namespaces` e `bridges` ([Lantz et al. 2010]).

Essa arquitetura leve possibilita a prototipação rápida de funções de rede, tornando o Mininet uma escolha popular no desenvolvimento e avaliação de soluções SDN. No entanto, o uso de um kernel compartilhado impõe limitações de isolamento e realismo na simulação de desempenho, já que todos os elementos da rede — hosts, switches e controladores — concorrem pelos mesmos recursos do sistema operacional. Isso pode afetar a fidelidade dos testes, especialmente em cenários com maior carga ou múltiplas instâncias paralelas. Embora o suporte nativo ao BMv2 com P4 no Mininet tradicional exija configuração manual, é possível integrá-lo por meio de *scripts* personalizados e instâncias específicas do console de linha de comando do BMv2, viabilizando testes com programas P4 ([Mininet 2023]).

O Mininet-WiFi é uma extensão do Mininet que adiciona suporte à simulação de redes sem fio. Ele permite configurar mobilidade de nós, interfaces Wi-Fi, pontos de acesso virtuais e até cenários híbridos entre redes cabeadas e sem fio. Diferentemente do Mininet tradicional, o Mininet-WiFi já oferece suporte direto ao BMv2 com P4, permitindo a criação de switches programáveis de forma nativa por meio de classes como P4Switch e P4AP. Com isso, é possível simular topologias complexas que combinam comunicação sem fio, mobilidade e planos de dados personalizados ([dos Reis Fontes and Rothenberg 2015]).

3.6. Testbed: FABRIC

O FABRIC (*FABRIC: Adaptive Programmable Research Infrastructure for Computer Science and Science Applications*) é um test bed de pesquisa em larga escala desenvolvido para experimentação com redes programáveis, sistemas distribuídos e aplicações orientadas a dados. A plataforma oferece infraestrutura física distribuída geograficamente nos Estados Unidos, conectada por enlaces de alta velocidade e composta por servidores programáveis, switches reconfiguráveis e nós com suporte a FPGA e SmartNICs ([Moskowitz et al. 2023]). Um dos diferenciais do FABRIC é o suporte nativo à linguagem P4. Os pesquisadores podem executar programas P4 em ambientes realistas, testando o comportamento do plano de dados em diferentes tipos de tráfego, cenários de topologia e até mesmo com interferência de outras aplicações reais que compartilham o mesmo *backbone*. Isso permite investigar a interoperabilidade entre protocolos, realizar medições precisas de desempenho e avaliar soluções de rede sob condições próximas ao mundo real ([Moskowitz et al. 2023, Bal et al. 2024]). Além disso, o FABRIC permite a integração de nós com suporte a SDN, computação em nuvem, dispositivos IoT e análise de dados, possibilitando a construção de experimentos multidisciplinares e replicáveis. A plataforma disponibiliza APIs e ferramentas para orquestração dos testes, além de documentação extensiva e apoio da comunidade científica ([Moskowitz et al. 2023]). Por sua flexibilidade e escalabilidade, o FABRIC é amplamente utilizado por universidades e centros de pesquisa para explorar novas arquiteturas de rede, validar protocolos experimentais e testar aplicações inovadoras que exigem redes programáveis de alto desempenho.

3.7. Patch Panel: P7

O P7 ([Souza et al. 2024]) é um painel de conexões programável (*patch panel*) desenvolvido como uma aplicação prática e inovadora da linguagem P4. Em vez de depender de

conexões físicas fixas entre portas, o P7 permite que o roteamento dos sinais físicos seja feito de forma dinâmica e controlada por software, com base em políticas definidas em programas P4. A arquitetura do P7 integra switches programáveis com circuitos de comutação e microcontroladores, permitindo a criação de caminhos lógicos entre portas físicas a partir de regras configuráveis. Com isso, torna-se possível implementar funcionalidades como: redirecionamento automático de portas, isolamento dinâmico de canais, espelhamento de tráfego físico e até balanceamento de carga em nível físico. O uso da linguagem P4 nesse contexto permite que o comportamento do *patch panel* seja reconfigurado conforme o cenário ou aplicação, sem a necessidade de intervenção manual no cabeamento. Essa abordagem é especialmente útil em ambientes de data center, laboratórios acadêmicos, redes industriais ou quaisquer situações em que a reconfiguração frequente da infraestrutura física seja necessária.

3.8. Áreas de Aplicação da Linguagem P4

A linguagem P4 tem sido amplamente explorada em diferentes domínios, destacando seu potencial além do simples encaminhamento de pacotes. A seguir, são apresentadas algumas das principais áreas de aplicação onde P4 tem sido integrado com sucesso:

- Segurança: Por meio de seu controle preciso sobre os `headers` e fluxos de pacotes, P4 tem sido utilizado em *firewalls* programáveis, sistemas de detecção de intrusão, verificação de conformidade de protocolos e mitigação de ataques DoS/DDoS. A capacidade de reconfigurar a lógica de tratamento de pacotes dinamicamente torna P4 uma ferramenta poderosa para políticas de segurança adaptativas. Trabalhos recentes também propõem o uso de switches P4 para mitigar ataques de envenenamento de topologia (*topology poisoning attacks*) e outros vetores avançados ([Smyth et al. 2023]).

- Redes de IoT: Ambientes de IoT se beneficiam da flexibilidade de P4 para implementar protocolos específicos de forma eficiente. Em cenários com restrições de largura de banda e latência, é possível otimizar o caminho dos pacotes com lógica customizada diretamente nos switches. Trabalhos recentes apontam o uso de P4 como mecanismo para gerenciamento de tráfego em redes LoRaWAN e para orquestração de dispositivos em arquiteturas *edge-cloud* ([Heideker et al. 2023]). Além disso, Carvalho propôs um gateway IoT programável em P4 capaz de identificar múltiplos protocolos e processar o conteúdo do *payload* para geração de alertas ou decisões de roteamento, ampliando a interoperabilidade e a automação de serviços em redes heterogêneas ([Carvalho 2024]).

- Interoperabilidade e conversão de protocolos: A programação do plano de dados também permite a conversão de protocolos em tempo real, como demonstrado por Heideker et al. ([Heideker et al. 2025]), que apresentaram uma técnica para transformar pacotes TCP em *datagramas* UDP diretamente dentro do switch P4. Essa abordagem promove interoperabilidade entre aplicações legadas e modernas sem exigir alterações nos dispositivos finais, sendo especialmente útil em cenários industriais, redes heterogêneas e sistemas IoT com diferentes padrões de comunicação.

- Detecção de Congestionamento e Balanceamento de Carga: Trabalhos como os

de [Kulkarni et al. 2022] e [Ke and Hsu 2020] mostram como a linguagem P4 pode ser utilizada para detectar e mitigar congestionamentos em tempo real e realizar balanceamento de carga dinâmico em redes SDN, promovendo maior eficiência na distribuição de tráfego.

- Detecção de Anomalias e Monitoramento Local: Estratégias de monitoramento diretamente no plano de dados foram exploradas por [Ding et al. 2022] e [Gao et al. 2021], que demonstram como switches P4 podem ser utilizados para detectar comportamentos anômalos com baixa latência, utilizando contadores e registradores para detectar desvios de padrão em tempo de execução.
- Computação na Rede (*In-Network Computing*): P4 também tem sido explorado para acelerar tarefas que tradicionalmente seriam realizadas por servidores, como agregações de dados, pré-processamento de fluxos e filtragem de conteúdo. Isso reduz a latência e o uso de largura de banda, além de distribuir o processamento pela infraestrutura da rede [Kim et al. 2021].
- Orquestração e SDN: Combinado com arquiteturas baseadas em SDN, P4 permite que controladores centralizados configurem dinamicamente o comportamento da rede de acordo com as necessidades da aplicação. Essa integração é crucial para a criação de redes autoadaptáveis, especialmente em contextos como datacenters, redes 5G e ambientes industriais [Berde et al. 2014].

3.9. 10 Anos de P4 - Produções Acadêmicas

A trajetória da linguagem P4 ao longo dos anos reflete um crescimento gradual e consolidado na comunidade acadêmica e industrial. A especificação da linguagem teve início em 2014 com a publicação de "P4: Programming Protocol-Independent Packet Processors"e as primeiras versões da especificação P4_14, que estabeleceram as bases para o desenvolvimento de planos de dados programáveis. Nesse primeiro ano, ainda não havia eventos significativos ou trabalhos acadêmicos além da própria documentação oficial.

A partir de 2015, as primeiras pesquisas acadêmicas começaram a surgir, com dois trabalhos identificados no Google Scholar. Além disso, ocorreram os primeiros eventos dedicados à linguagem, sendo o mais marcante o primeiro *workshop* de P4, realizado em Princeton, que contou com 11 apresentações. Esse evento marcou o início de uma comunidade ativa interessada na exploração das capacidades da linguagem P4.

No ano seguinte, 2016, a linguagem evoluiu com o lançamento da especificação P4_16, trazendo mudanças significativas na estrutura e funcionalidades da linguagem. O crescimento na produção acadêmica se tornou evidente, com 13 trabalhos identificados no Google Scholar. Além disso, o *workshop* P4 contou com 20 apresentações, elevando o número total de trabalhos discutidos no ano para 33. Esse período consolidou a linguagem como uma opção viável para a programação do plano de dados.

Em 2017, a presença do P4 na comunidade acadêmica continuou se expandindo. Foram encontrados 13 artigos no Google Scholar, enquanto o *workshop* P4 contou com 13 palestras e 13 demonstrações técnicas. O aumento na quantidade de apresentações e demonstrações sugere um interesse crescente na aplicação prática da linguagem, bem como na experimentação com hardware programável.

O ano de 2018 marcou um salto expressivo na quantidade de publicações e eventos. Foram identificados 45 artigos acadêmicos, além da realização de cinco eventos importantes. O *workshop* P4, já consolidado, contou com 14 palestras e 14 demonstrações. Além disso, foi realizado o primeiro *workshop* europeu sobre P4, que trouxe 8 pôsteres e 5 demonstrações. Esse período também indica uma maior adoção da linguagem, com mais pesquisadores e desenvolvedores se envolvendo na criação de novas aplicações para redes programáveis.

Em 2019, o número de publicações e eventos continuou crescendo. Foram encontrados 56 artigos no Google Scholar e um total de 10 eventos organizados ao longo do ano. O *workshop* P4 manteve sua relevância com 14 palestras e 12 demonstrações, enquanto o *workshop* europeu ampliou sua participação, apresentando 20 trabalhos. Esse ano representou um pico na adoção da linguagem, com a comunidade expandindo seu foco para diferentes aspectos do desenvolvimento de redes programáveis.

A pandemia de COVID-19 teve um impacto notável na produção acadêmica e na realização de eventos em 2020. Apesar disso, foram encontrados 61 artigos publicados e, embora a maioria dos eventos presenciais tenha sido cancelada, o *workshop* europeu foi realizado no formato online, trazendo 13 apresentações. O impacto da pandemia pode ser observado na redução do número de eventos, mas a continuidade das pesquisas demonstra a resiliência da comunidade P4.

A recuperação começou em 2021, com um aumento expressivo na quantidade de publicações e eventos. Foram identificados 71 artigos no Google Scholar e seis eventos organizados ao longo do ano. O *workshop* P4 teve um total de 36 trabalhos apresentados, além de três tutoriais na conferência SIGCOMM. O *workshop* europeu seguiu acontecendo, desta vez com 4 palestras, 6 demonstrações e 3 aplicações discutidas. Além disso, foi realizado o primeiro *workshop* de P4 em Taiwan, trazendo 13 apresentações adicionais. A retomada dos eventos presenciais foi um marco importante para a disseminação e o avanço da linguagem.

O ano de 2022 registrou um novo crescimento, com um total de 68 artigos identificados e 10 eventos realizados. O *workshop* P4 trouxe 55 apresentações, enquanto o evento de Taiwan contou com 11 apresentações e o *workshop* europeu apresentou 12 trabalhos. A diversidade de eventos reflete o interesse global na linguagem e sua adoção em diferentes contextos, fortalecendo a colaboração internacional na área de redes programáveis.

Em 2023, o número de artigos cresceu para 83, e quatro eventos foram organizados ao longo do ano. O *workshop* EuroP4 contou com 11 apresentações, enquanto o *workshop* P4 apresentou 31 trabalhos. O número menor de eventos pode indicar uma reorganização da comunidade, com um foco maior em produções científicas e em eventos mais direcionados.

No ano de 2024, a tendência de crescimento continuou, com 88 artigos publicados e 11 eventos realizados. O *workshop* P4 trouxe 24 apresentações, enquanto o *workshop* europeu contou com 7 apresentações. O aumento na quantidade de eventos reflete uma diversificação das discussões em torno da linguagem, com novas aplicações e cenários de uso sendo explorados.

A metodologia utilizada para essa análise se baseou em buscas realizadas no Google Scholar, utilizando exclusivamente a palavra-chave "P4". Essa limitação pode significar

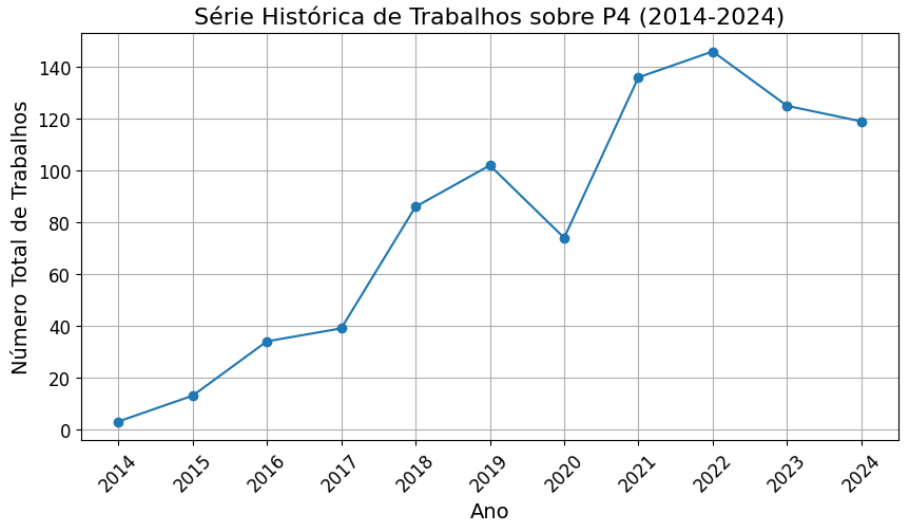


Figura 3.4: Série histórica de Trabalhos sobre P4

que alguns trabalhos relevantes não tenham sido contabilizados, especialmente aqueles que utilizam termos como "*programmable data planes*"ou variações mais específicas. A inclusão de palavras-chave adicionais pode resultar em um número ainda maior de publicações relacionadas à linguagem. Além disso, o impacto da pandemia de COVID-19 entre 2019 e 2020 afetou diretamente a realização de eventos, interrompendo o *workshop* P4 por dois anos e levando o *workshop* europeu a ocorrer apenas no formato online. Com o retorno dos eventos presenciais em 2023, o número de trabalhos apresentados nos *workshops* diminuiu em comparação ao período em que eram realizados online, reduzindo o número total de trabalhos. A Figura 3.4 resume o número de trabalhos sobre p4 durante a série histórica de 2014 à 2024. Os dados de eventos para 2025 ainda não foram compilados, mas podemos encontrar algumas referências recentes na literatura.

Ao longo dos anos, a realização de eventos especializados ajudou a consolidar a linguagem, com a criação de *workshops* específicos em diferentes regiões, como a Europa e Taiwan, além do tradicional *workshops* P4, que continua sendo o evento central para a comunidade. O levantamento histórico mostra que, a partir de 2018, o número de trabalhos apresentados nesses eventos passou a representar uma fração significativa do total de produções científicas sobre P4, reforçando a importância desses encontros para a disseminação da pesquisa e a troca de conhecimento na área.

4. Ambiente P4Docker

A adoção do P4 em larga escala enfrenta diversos desafios, incluindo a necessidade de profissionais com habilidades especializadas, dificuldades para garantir interoperabilidade, suporte de hardware limitado, preocupações com segurança e a complexidade inerente ao processamento personalizado de pacotes. Além disso, a transição de sistemas legados para redes programáveis adiciona outra camada de dificuldade ([Kfoury et al. 2021b]).

Embora o interesse acadêmico pelo desenvolvimento da programabilidade do plano de dados esteja crescendo, o custo elevado dos dispositivos de rede compatí-

veis com P4 restringe a maioria dos experimentos a simulações ([Kfoury et al. 2021b, Goswami et al. 2023]). Mesmo quando tais equipamentos estão disponíveis, sua quantidade pode ser insuficiente para viabilizar experimentos em larga escala e explorar todo o potencial do P4.

Ferramentas de virtualização de redes, como o Mininet [Mininet 2023], surgem como uma alternativa para testar e validar aplicações P4. Essa plataforma permite a criação de ambientes de rede realistas com múltiplos switches P4, hosts e até controladores programáveis ([Chen et al. 2023]). O Mininet cria redes virtuais executando o kernel real, switches e código de aplicação em uma única máquina. Além disso, ele suporta OpenFlow e P4 por meio dos switches BMv2 (Behavioral Model Version 2 - [Consortium 2024]), tornando-se uma ferramenta essencial para estudar redes e protocolos em um ambiente controlado.

Apesar de seu amplo uso e aceitação na comunidade acadêmica, o Mininet apresenta limitações em termos de escalabilidade e arquitetura, dificultando a simulação de cenários de grande porte. Em contrapartida, a tecnologia de contêineres tem sido amplamente adotada em ambientes acadêmicos, facilitando a transição de configurações experimentais para ambientes de produção.

Nesse contexto, o P4Docker surge como um testbed alternativo baseado em contêineres, projetado para oferecer maior isolamento entre componentes de rede, ao mesmo tempo que atende às demandas de experimentação em redes voltadas para IoT. Diferentemente do Mininet, onde todos os elementos de rede compartilham o mesmo sistema operacional, o P4Docker encapsula cada componente em um contêiner independente. Esse isolamento reduz a interferência de recursos e melhora a reprodutibilidade dos experimentos. Além disso, essa abordagem é particularmente vantajosa para cenários de IoT, onde o processamento de baixa latência, a personalização de protocolos e a priorização de tráfego são essenciais para lidar com redes de sensores em larga escala e infraestruturas de computação em névoa.

4.1. Arquitetura

A Figura 3.5(a) ilustra como o Docker Engine orquestra os contêineres, isolando cada um dos outros e interagindo com o Docker Engine, que faz interface com o Sistema Operacional Host. A combinação de namespaces e contêineres garante que a comunicação entre os dois hosts por meio dos pares Ethernet virtual (veth) atravesse um caminho de rede virtualmente indistinguível daquele de uma rede física. Essa configuração contrasta com os recursos de ferramentas de emulação de rede como o Mininet, que, embora versátil, executa todos os componentes de rede dentro do mesmo kernel do sistema operacional host, conforme ilustrado na Figura 3.5(b), potencialmente levando à contenção de recursos compartilhados e à mistura do tráfego de rede.

Ao garantir a separação do tráfego e a independência dos dispositivos de rede, o P4Docker oferece uma plataforma valiosa para pesquisadores interessados em estudar o comportamento das redes e o impacto da programação do plano de dados P4. Seu ambiente de teste reflete de forma realista as topologias de redes do mundo real, proporcionando bases teóricas sólidas e implicações práticas para o desenvolvimento e validação de protocolos, sistemas e arquiteturas de rede em um ambiente virtual controlado.

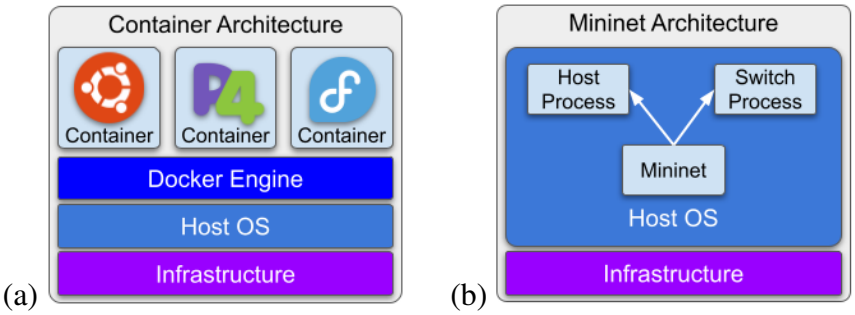


Figura 3.5: (a) Arquitetura Docker com isolamento de containers. (b) Arquitetura Mininet com conceito de isolamento de processos.

O P4Docker integra contêineres Docker com o switch P4 do Behavioral Model Version 2 (BMv2) ([Consortium 2024]), criando ambientes de rede isolados que simulam cenários reais. Sua arquitetura utiliza namespaces do Linux para separar a pilha de rede de cada contêiner, garantindo que cada host simulado opere de maneira independente. Essa abordagem espelha o isolamento encontrado em redes físicas, resultando em um ambiente de teste realista e de alta fidelidade.

A estrutura do P4Docker segue uma arquitetura frontend-backend direta, ilustrada na Fig. 3.6. O frontend, desenvolvido em JavaScript, fornece uma interface gráfica intuitiva baseada no Cytoscape2 para criação de topologias de rede. O backend, por sua vez, gera scripts para implantação e gerenciamento dessas topologias, além de interagir com o P4 Compiler, executado dentro de um contêiner Docker, para compilar e armazenar os arquivos de configuração.

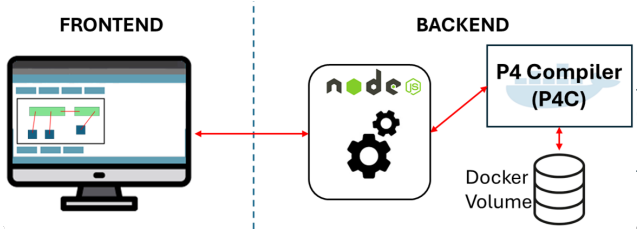


Figura 3.6: Arquitetura do P4Docker

Para facilitar o compartilhamento de arquivos entre os contêineres, o P4Docker estabelece um volume compartilhado ("Share") durante sua instalação. Esse volume garante armazenamento persistente de dados, permitindo a retenção de informações entre reinicializações e atualizações dos contêineres. Além disso, o contêiner P4 Compiler utiliza esse volume para armazenar códigos P4 compilados e compartilhá-los com os contêineres de switches. Desta forma, qualquer código compilado pela interface gráfica do P4Docker será automaticamente disponibilizado no volume compartilhado e poderá ser utilizado por qualquer switch na topologia de testes especificada.

O P4Docker permite ainda o gerenciamento, execução e limpeza dos ambientes e topologias definidos, simplificando a construção de topologias de rede baseadas em

switches P4.

4.2. Principais Funções

O P4Docker oferece uma abordagem modular e escalável, permitindo a criação e gerenciamento de topologias personalizadas com suporte para múltiplos switches P4 e hosts. Suas principais funções incluem:

- Compilação e depuração: Ferramentas integradas permitem compilar e testar programas P4 diretamente no ambiente, facilitando a identificação de erros e ajustes antes da implantação em hardware real.
- Criação de topologias virtuais: É possível montar redes com switches P4, hosts e controladores, com emulação realista garantida por contêineres isolados.
- Configuração automatizada: A interface gráfica ou scripts geram automaticamente endereços, regras de roteamento e parâmetros de rede, simplificando a criação dos ambientes.
- Execução com BMv2: Utiliza o BMv2 como backend para testes de pipelines P4 personalizados, com suporte ao V1Model.
- Isolamento e escalabilidade: Cada componente roda em seu próprio contêiner, permitindo testes independentes e escaláveis.
- Automação de ambientes: Scripts de inicialização e limpeza são gerados automaticamente, agilizando a reconfiguração dos experimentos.
- Persistência de dados: Os contêineres compartilham volumes para armazenar arquivos de configuração, logs e códigos compilados.
- Suporte a múltiplos cenários: Pode ser usado para testes em SDN, IoT, redes distribuídas e computação em névoa, entre outros contextos.

Essas funções fazem do P4Docker uma plataforma flexível para pesquisa e desenvolvimento em redes programáveis, fornecendo um ambiente completo para a validação de novas arquiteturas e protocolos sem a necessidade de hardware dedicado.

4.3. Topologias

As topologias no P4Docker são modeladas como grafos, onde os nós representam hosts e switches, enquanto as arestas correspondem às conexões de rede entre eles. Essa estrutura possibilita a criação de cenários de experimentação altamente configuráveis, permitindo testes em diferentes arquiteturas e garantindo flexibilidade na definição das redes.

Diferente de algumas ferramentas de emulação, o P4Docker adota um modelo de comunicação em pares, exigindo que cada conexão entre dois nós possua um endereço de rede distinto. Como consequência, não é possível conectar múltiplos enlaces a uma mesma porta de switch, exigindo que cada host esteja vinculado a uma rede específica e que o switch intermediário seja corretamente configurado para encaminhar o tráfego. Essa

restrição decorre do alto grau de isolamento entre os componentes da topologia, onde cada nó opera dentro de um namespace de rede separado, garantindo independência total entre as pilhas TCP/IP. Esse isolamento evita interferências, garantindo que cada nó funcione autonomamente e reproduza fielmente o comportamento de uma rede física. A comunicação ocorre por meio de pares virtuais de Ethernet (veth), replicando as características de conexões físicas e tornando os experimentos mais realistas.

As topologias são representadas em formato JSON, permitindo armazenamento, carregamento e compartilhamento de configurações de forma simples e eficiente. Isso possibilita que os usuários reutilizem definições complexas sem a necessidade de reconfiguração manual, aumentando a portabilidade dos experimentos. Além disso, a integração com scripts automatizados para deploy e limpeza das topologias reduz o tempo de preparação dos testes e melhora a eficiência na execução de múltiplos cenários.

A flexibilidade do P4Docker permite sua aplicação em uma ampla variedade de experimentos, desde redes convencionais até redes programáveis, IoT e computação em névoa. A possibilidade de definir parâmetros avançados como largura de banda, latência e perda de pacotes possibilita a simulação de condições reais de rede, sendo útil para a avaliação do desempenho de protocolos distribuídos, políticas de roteamento e controle de tráfego. Essa abordagem também possibilita a criação de cenários de segurança para testar estratégias de mitigação de ataques e isolamento de tráfego entre domínios distintos.

O P4Docker oferece um grau superior de isolamento e independência entre os elementos da topologia, tornando os experimentos mais previsíveis e reprodutíveis. A arquitetura baseada em containers Docker em conjunto com a capacidade de portabilidade de topologias garante que cada teste seja executado de forma consistente, independentemente do ambiente, permitindo a replicação dos experimentos sem ajustes manuais. Essa característica facilita o compartilhamento de cenários entre pesquisadores e equipes de desenvolvimento, promovendo maior colaboração e padronização na execução de testes.

Com essa estrutura modular e altamente configurável, o P4Docker se torna uma solução poderosa para testar, depurar e validar protocolos P4 em diferentes contextos, suportando desde simulações simples até topologias complexas compostas por múltiplos switches e regras de encaminhamento personalizadas.

4.4. Interface Gráfica

A interface gráfica do P4Docker foi desenvolvida para oferecer uma experiência intuitiva e eficiente no gerenciamento de topologias e no desenvolvimento de código P4. Ela é dividida em três seções principais: a interface de gerenciamento de topologias, a interface de depuração e a interface de projetos, cada uma com funções específicas que facilitam a criação, configuração e execução de experimentos em redes programáveis.

A interface de gerenciamento de topologias é a tela principal é o ponto central da interface, onde o usuário pode definir, criar, alterar e gerenciar topologias de rede. Nessa seção, é possível adicionar novos nós, estabelecer conexões entre dispositivos, configurar parâmetros individuais de cada elemento da topologia e visualizar a estrutura geral da rede em um ambiente interativo.

A interface de depuração permite a compilação e validação de código P4 direta-

mente no ambiente do P4Docker. O usuário pode definir um nome para o código-fonte, compilar o programa e visualizar a saída do processo em tempo real. Caso a compilação seja bem-sucedida, o arquivo gerado pode ser exportado e utilizado em testes. Em caso de erro, a interface exibe mensagens detalhadas e a linha exata onde o problema ocorreu, facilitando o processo de correção e depuração.

Por fim, a interface de projetos possibilita o gerenciamento de diferentes topologias criadas na tela principal. Através dessa seção, o usuário pode carregar configurações previamente definidas, fazer o deploy das topologias para execução e realizar a limpeza do ambiente ao finalizar um experimento. Essa funcionalidade garante flexibilidade no desenvolvimento e reuso de cenários de teste, permitindo alternar entre diferentes configurações sem a necessidade de recriação manual.

Essas três seções trabalham em conjunto para oferecer um ambiente completo de experimentação, permitindo desde a criação de redes até a execução e depuração de programas P4, promovendo um fluxo de trabalho contínuo e otimizado.

4.4.1. Interface de Compilação e Depuração

A interface de compilação e depuração, ilustrada na Figura 3.7, permite que o usuário defina um nome para o código a ser compilado, o qual será sempre salvo com esse nome e a extensão .json. A interface apresenta um botão "Compile", que, ao ser acionado, envia o código para o container P4 Compiler, configurado durante a instalação do P4Docker.

A saída do processo de compilação é exibida no painel "Output", localizado no lado direito da tela, conforme ilustrado na Figura 3.7. Quando a compilação ocorre sem erros, a mensagem "Code Compiled"é exibida, indicando sucesso. Além disso, o arquivo gerado pode ser baixado diretamente através do botão "Download Compiled File", facilitando sua exportação e aumentando a portabilidade dos projetos.

Caso ocorra um erro, a tela "Output"exibe a mensagem gerada pelo compilador em tempo real, destacando a linha onde o problema foi identificado. Esse mecanismo, ilustrado na Figura 3.8, auxilia na depuração do código, permitindo correções rápidas e precisas.

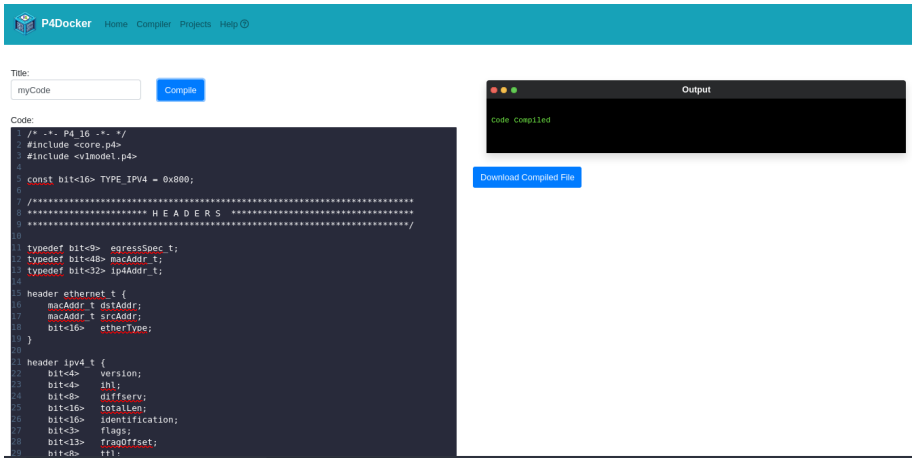


Figura 3.7: Interface do Compilador com código compilado corretamente

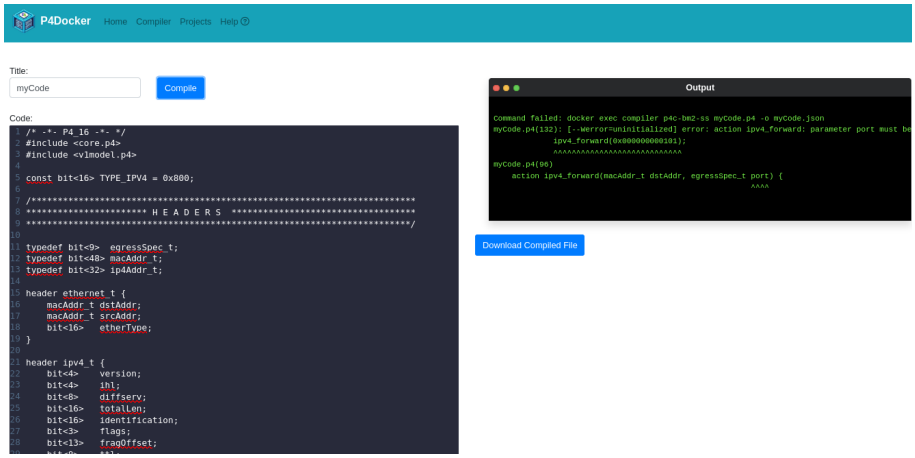


Figura 3.8: Interface do Compilador com código apresentando erro

4.4.2. Interface de Gerenciamento de Topologias

A interface de gerenciamento de topologias, ilustrada na Fig. 3.9 permite que os usuários:

- Adicionem nós à topologia, representando hosts, switches P4 e outros dispositivos de rede.
- Configurem portas para cada nó, especificando parâmetros como endereço IP, MAC e largura de banda.
- Criem conexões entre os nós e definam características de enlaces, como latência e taxa de transmissão.
- Editem e removam elementos da topologia conforme necessário.
- Salvem e carreguem topologias em formato JSON para reutilização e compartilhamento.

- Gerem scripts para criação e limpeza de ambientes de teste.

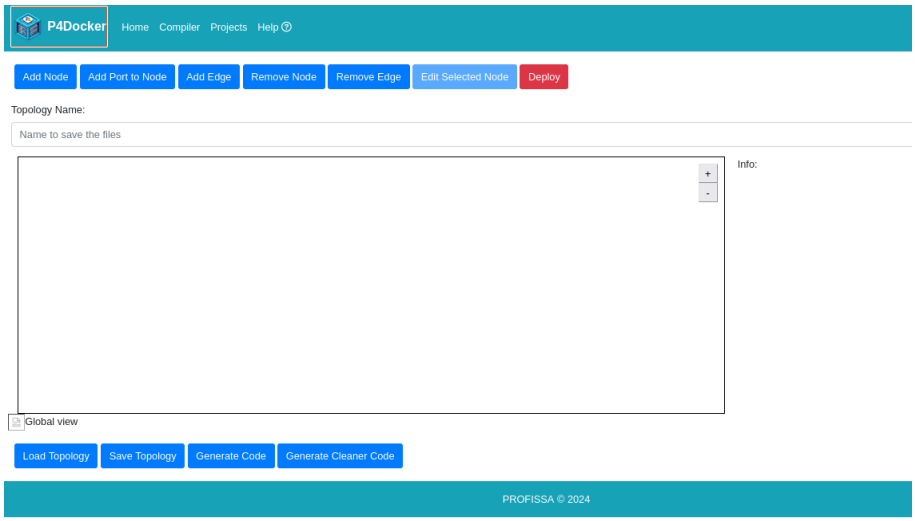


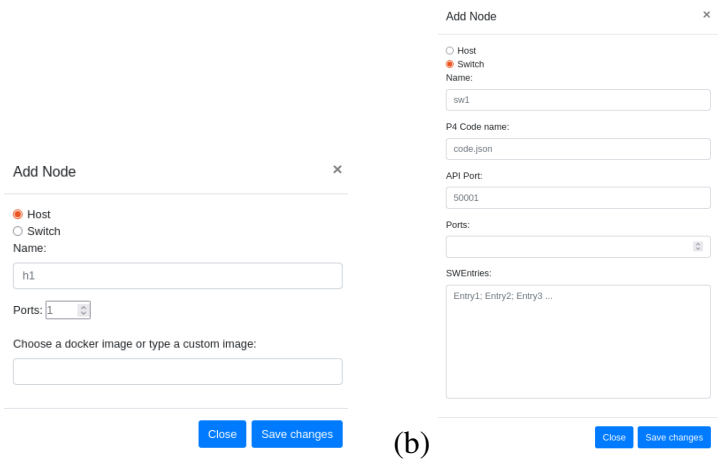
Figura 3.9: Interface principal do P4Docker

A Figura 3.10 apresenta os modais exibidos ao clicar no botão "Add Node". Nessa interface, é possível adicionar dois tipos de nós à topologia existente. Os nós do tipo "host"representam os componentes da topologia e podem ser configurados por meio dos seguintes parâmetros:

- Name: define o nome do nó na topologia.
- Ports: determina a quantidade de portas disponíveis no nó.
- Docker image: especifica a imagem Docker que servirá como base para o nó. Essa imagem pode ser tanto uma versão local quanto um repositório remoto disponível no Docker Hub.

Os nós do tipo "Switch"representam os switches P4 baseados em BMv2, e podem ser configurados com os seguintes parâmetros:

- Name: define o nome do nó na topologia.
- Ports: determina a quantidade de portas disponíveis no switch.
- P4 Code name: define o nome do código P4 compilado que será utilizado no switch.
- API Port: define a porta da API do BMv2 para receber comandos.
- Ports: determina a quantidade de portas disponíveis no switch.
- SWEntries - define as regras de entrada openflow que podem ser definidas no switch quando necessário. As entradas são passadas via interface de linha de comando do switch BMv2.



(a) (b)

Figura 3.10: (a) Modal com opções para criação de um nó host. (b) Modal com opções para criação de um nó switch.

Ao criar switches P4, a ordem das portas desempenha um papel fundamental, pois é necessário especificar corretamente as portas de entrada e saída. Por padrão, a interface atribui os identificadores das portas de forma sequencial, iniciando em 1 e seguindo a ordem em que os nós aparecem na topologia. Dessa forma, em um nó chamado "switch1"com três portas, os identificadores atribuídos serão "switch-p1", "switch-p2"e "switch-p3".

A interface permite reordenar as portas simplesmente arrastando-as para novas posições. Essa alteração modifica a maneira como o switch interpreta a numeração das portas internamente, sem alterar os nomes visíveis. Por exemplo, se a porta "switch-p3"for movida para a primeira posição na interface, seu nome permanecerá inalterado, mas o switch passará a tratá-la como a porta 1 para fins de codificação no P4. Esse mecanismo oferece flexibilidade na configuração, garantindo que a numeração das portas esteja alinhada com as regras e lógica implementadas no código P4.

A Figura 3.11 ilustra a criação de dois componentes na topologia atual: um host e um switch. Ao selecionar qualquer um desses componentes, suas configurações são exibidas no painel à direita da tela. No exemplo apresentado na figura, ainda não há configurações definidas para o host além da imagem Docker associada.

Ao selecionar um nó na topologia, novos botões são habilitados, permitindo a realização de ações específicas sobre o elemento escolhido:

- Add Port to Node – Adiciona uma nova porta ao nó selecionado.
- Remove Node – Remove o nó e todas as conexões associadas a ele.
- Edit Selected Node – Permite a edição das configurações do nó.

Além de remover nós inteiros, o botão Remove Node também permite excluir portas específicas, sem a necessidade de remover o nó ao qual estão vinculadas.

Figura 3.11: Interface de gerenciamento de topologias com componentes

O botão "Add Edge"define as conexões entre os nós, permitindo a configuração das portas. Essa funcionalidade é essencial para a construção de topologias de rede. As conexões são criadas de maneira direcional, exigindo a seleção de um nó de origem (source) e um nó de destino (target). No entanto, essa direção é utilizada apenas para a representação gráfica da topologia e não afeta o comportamento da comunicação, que permanecerá bidirecional.

Para configurar uma conexão, primeiro selecionamos o nó pai, que contém a porta a ser configurada, e em seguida, escolhemos a porta específica. Após essa seleção, as seguintes configurações devem ser definidas: Endereço IP - Definido no formato CIDR (Classless Inter-Domain Routing), que utiliza uma notação como /24 para especificar o tamanho da máscara de rede; MAC Address: Especificado no formato hexadecimal.

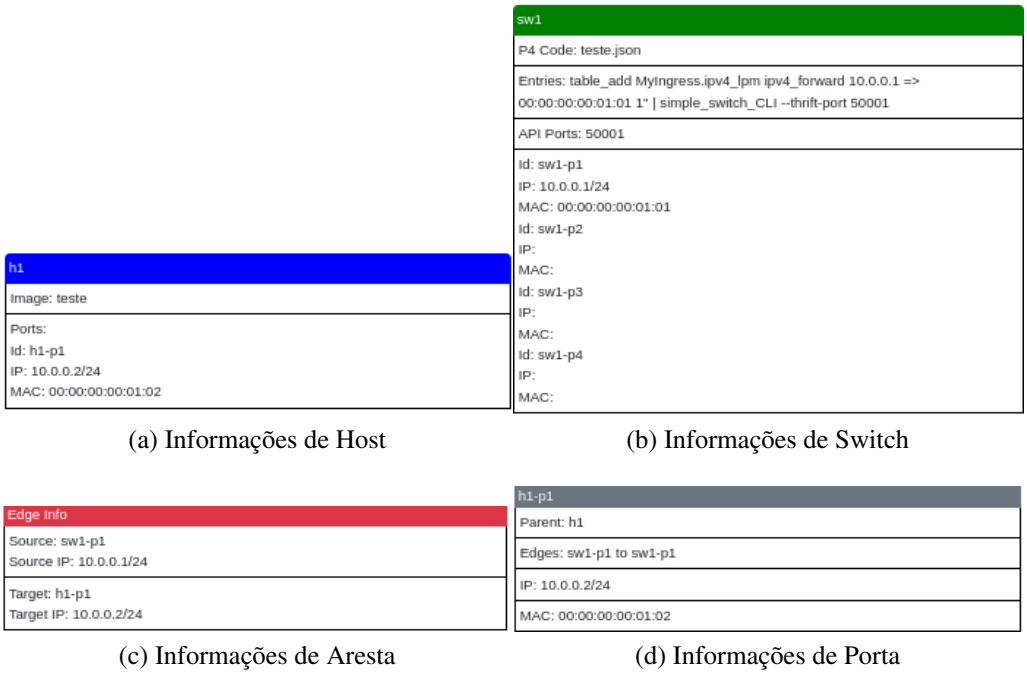
Essas mesmas configurações devem ser repetidas para o nó de destino, garantindo que não haja conflitos de IP. Além disso, a interface permite a definição de limites de banda (fornecida em formato kbit, mbit sem espaços) e atraso de rede (delay), funcionalidades úteis para simulações e testes específicos. Essas opções ficam disponíveis ao ativar a opção em "Define Bandwidth and Delay".

Após configurar todos os parâmetros, clicar em "Save Changes"cria a conexão entre os nós e aplica as configurações nas portas correspondentes. Vale destacar que não é possível editar conexões já existentes; caso seja necessário modificar uma aresta, o usuário deve selecioná-la, clicar em "Remove Edge"e criar uma nova conexão com os ajustes desejados.

É importante ressaltar que o P4Docker não realiza verificações automáticas para evitar conflitos de IPs ou endereços MAC. Dessa forma, a responsabilidade por garantir a consistência das configurações recai sobre o usuário.

Uma vez criadas as arestas, as novas configurações podem ser visualizadas ao selecionarmos os nós, como ilustrado na Figura 3.12, que mostra as informações para um nó host, um nó switch, uma porta e uma aresta.

O menu inferior da interface gráfica do P4Docker fornece opções essenciais para o gerenciamento e exportação de topologias, facilitando a reutilização e compartilhamento



(a) Informações de Host (b) Informações de Switch

(c) Informações de Aresta (d) Informações de Porta

Figura 3.12: Caixas de informações na interface do P4Docker

de ambientes de experimentação. Entre as funcionalidades disponíveis, destacam-se:

- Load Topology: Permite carregar uma topologia salva anteriormente para visualização e edição.
- Save Topology: Salva a topologia no estado atual, incluindo todas as configurações de portas e conexões, exportando o ambiente no formato JSON.
- Generate Code: Gera um script Bash para a criação da topologia configurada, permitindo sua implantação automatizada em outro ambiente.
- Generate Cleaner Code: Gera um script para a remoção da topologia criada, garantindo que o ambiente de testes possa ser redefinido de forma rápida e eficiente.

A capacidade de exportação das topologias, bem como dos códigos compilados pelo P4 Compiler, proporciona ao P4Docker um alto nível de portabilidade. Isso permite que cenários de experimentação sejam facilmente compartilhados com colaboradores, garantindo a replicabilidade dos testes.

Além disso, a adoção do Docker assegura o isolamento dos ambientes, eliminando problemas de compatibilidade entre diferentes máquinas. Dessa forma, uma configuração que funciona em um sistema poderá ser executada em outro sem a necessidade de ajustes manuais, garantindo maior consistência nos experimentos realizados.

A interface também permite que os usuários gerem scripts auxiliares para desmontar o ambiente criado, bem como salvar e carregar topologias. Essa funcionalidade

aprimora a usabilidade do sistema, possibilitando a exportação de configurações para diferentes máquinas e promovendo um fluxo de trabalho mais eficiente e adaptável no desenvolvimento de redes.

A última funcionalidade da interface de configuração é o botão *Deploy*, responsável por transformar a configuração atual da topologia em um projeto executável. Ao acionar essa opção, a topologia é implantada e passa a ser gerenciada na interface de projetos, acessível através da opção **"Projects"** no menu principal.

4.4.3. Interface de Projetos

A interface de projetos permite carregar e gerenciar projetos previamente definidos na interface de gerenciamento de topologias. Através dessa seção, é possível carregar topologias salvas, executar o *deploy* de ambientes e realizar sua limpeza quando necessário.

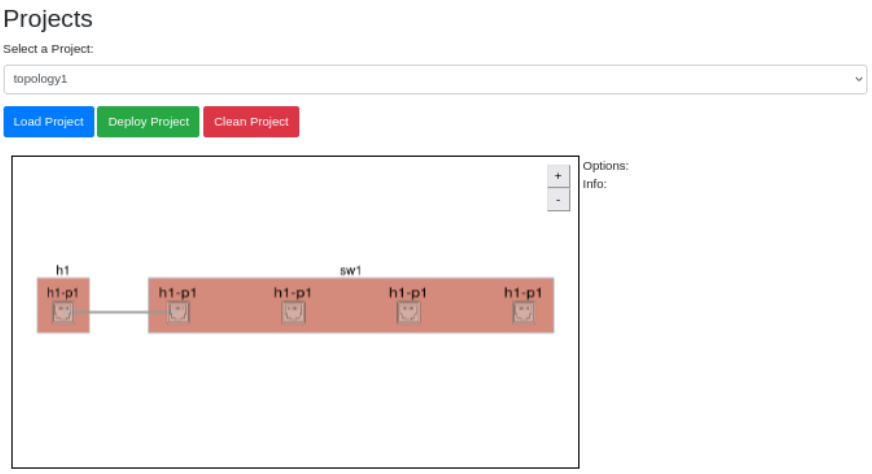


Figura 3.13: Interface de projetos - projeto não está em execução

A Figura 3.13 ilustra um exemplo de topologia de testes carregada na interface. Os nós destacados em vermelho representam componentes que não estão operacionais, ou seja, os contêineres correspondentes não estão em execução. Ao clicar no botão **Deploy**, o script de implantação é executado, iniciando os contêineres e configurando o ambiente.

Quando os nós estão operacionais, eles são marcados com a cor verde, conforme ilustrado na Figura 3.14. Uma vez que a topologia esteja ativa, é possível selecionar qualquer nó e conectar-se ao seu terminal através do botão **Connect**, como mostrado na Figura 3.14. Essa funcionalidade abre uma nova aba contendo o terminal do contêiner selecionado, permitindo acesso direto aos hosts e possibilitando a verificação dos logs dos switches P4 em tempo real. Para isso, basta conectar-se ao switch desejado e acessar o arquivo de log localizado em `/tmp/switch.log`.

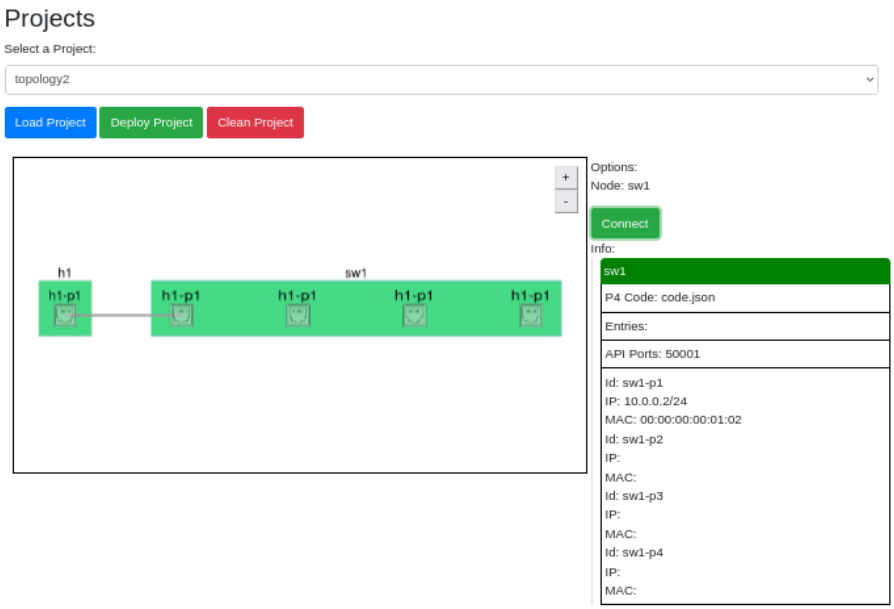


Figura 3.14: Interface de projetos - projeto em execução

Essa abordagem facilita o monitoramento e depuração do ambiente, proporcionando maior controle sobre a execução dos experimentos e garantindo que os componentes da topologia estejam corretamente configurados e operacionais.

4.5. Depuração (debug)

O P4Docker oferece um ambiente robusto para a depuração de redes programáveis, proporcionando a captura de logs em tempo real dos switches BMv2. Por padrão, os containers que executam switches BMv2 no P4Docker armazenam os registros no arquivo `/tmp/switch.log`, permitindo o monitoramento contínuo do comportamento da rede. Essa funcionalidade possibilita que os usuários acessem os logs a qualquer momento, registrando eventos relevantes e analisando detalhadamente o processamento de pacotes na topologia. A Figura 3.15 ilustra o log em tempo real de um pacote recebido pelo switch P4, mostrando todas as funções acionadas, processamento do pacote até sua saída.

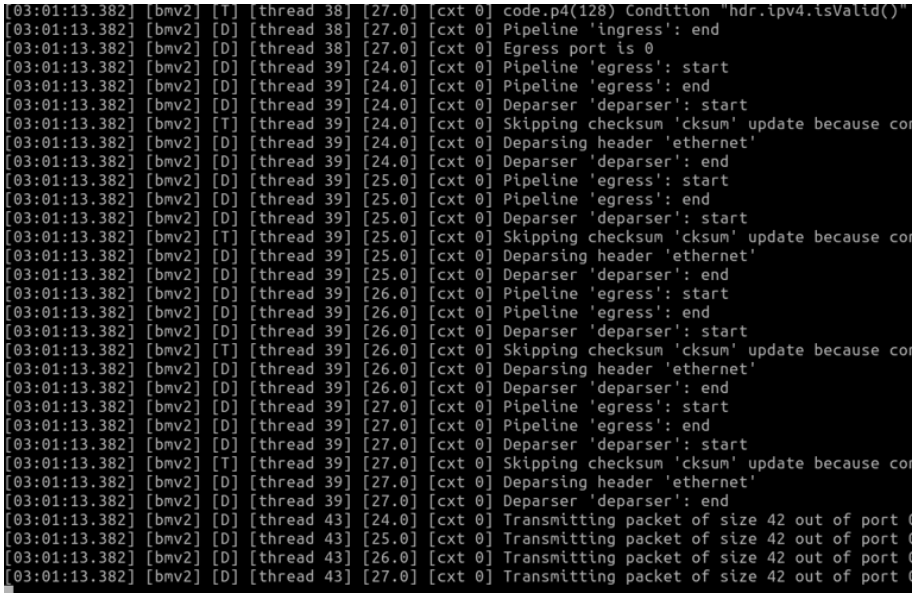


Figura 3.15: Log em tempo real de um switch P4

Uma das grandes vantagens do P4Docker em relação a ferramentas tradicionais de simulação, como o Mininet, é a capacidade de tratar cada switch de forma independente. Em topologias complexas, onde múltiplos switches e hosts interagem simultaneamente, é possível interromper apenas um switch específico, modificar seu código P4 e reiniciá-lo sem afetar o restante da topologia. Isso permite que os desenvolvedores testem alterações incrementais no código, observem os impactos diretamente nos logs e ajustem o comportamento da rede sem a necessidade de reiniciar todo o ambiente de simulação.

Essa abordagem é um diferencial significativo, pois evita a necessidade de reconfigurar toda a topologia sempre que uma alteração for realizada em um dos switches. Em ferramentas como o Mininet, qualquer modificação no código P4 exige a reinicialização completa da simulação, o que pode ser inviável para testes frequentes e iterativos. No P4Docker, as mudanças podem ser aplicadas de maneira dinâmica, possibilitando uma depuração mais eficiente e ágil, acelerando o processo de desenvolvimento e análise de redes programáveis.

Além disso, o acesso direto aos logs individuais de cada switch facilita a compreensão do impacto de uma mudança específica, tornando mais simples a identificação de erros e otimização do código P4. Essa capacidade de testar modificações isoladas e visualizar os efeitos em tempo real sem interromper a operação dos demais dispositivos da rede reforça o P4Docker como uma ferramenta ideal para experimentação, ensino e pesquisa em redes definidas por software e programação de planos de dados.

É possível ainda fazer a depuração com Wireshark e capturar os pacotes diretamente nas interfaces de rede dos containers. No entanto, como os nós da topologia são executados em containers isolados por namespaces, as interfaces virtuais (`veth`) não ficam visíveis no Wireshark executado no sistema host.

Para contornar essa limitação, é necessário executar o Wireshark dentro do namespace de rede do container desejado. Isso pode ser feito obtendo o PID do processo do

container e utilizando o comando `nsenter`. Por exemplo, para acessar as interfaces do container `host1`:

```
PIDHost1=$(docker inspect -f '{{.State.Pid}}' host1)
sudo nsenter -t $PIDHost1 -n wireshark
```

O primeiro comando armazena o PID do container na variável `PIDHost1`. Em seguida, o comando `nsenter` abre o Wireshark no contexto de rede daquele container, permitindo que todas as suas interfaces internas sejam acessadas para captura de pacotes. Essa abordagem é especialmente útil para acompanhar o tráfego real entre os nós e observar como os pacotes são manipulados pelos switches programáveis em tempo real.

Além da captura básica de logs, a depuração no P4Docker pode ser aprimorada com estratégias avançadas que combinem análise automatizada de eventos, integração com ferramentas externas e geração de métricas de desempenho. Um exemplo prático é a criação de filtros personalizados nos logs para destacar apenas pacotes que acionam ações específicas, como redirecionamentos, drops ou modificações em headers. Essa funcionalidade pode ser futuramente incorporada à interface gráfica por meio de uma aba de visualização de eventos relevantes.

Outra possibilidade promissora é o uso de scripts de análise que processam os logs após a execução dos testes, extraindo estatísticas como número de pacotes por porta, número de entradas acionadas por tabela e tempo médio de processamento por switch. Esses dados podem ser visualizados com ferramentas como Grafana ou visualizadores em tempo real integrados ao frontend, promovendo uma visão mais abrangente sobre o comportamento da rede.

A interface de depuração também pode evoluir para oferecer um modo passo a passo, permitindo que o usuário inspecione o caminho percorrido por um pacote dentro do pipeline, analisando cada estágio: parser, match-action ingress, egress, e deparser. Esse recurso funcionaria de forma semelhante a um debugger tradicional, e seria especialmente valioso para fins educacionais.

A exportação estruturada dos logs em formatos como JSON ou CSV viabiliza a análise por ferramentas externas de machine learning e detecção de anomalias. Isso abre espaço para investigações mais sofisticadas sobre falhas de rede, comportamento inesperado ou mesmo ataques. Com essas extensões, o P4Docker pode se consolidar não apenas como um ambiente de testes, mas como uma plataforma completa de observabilidade para redes programáveis.

4.6. Instalação do P4Docker

A instalação do P4Docker pode ser realizada de forma automatizada através do script *install.sh* para distribuições Ubuntu ou *installYUM.sh* para distribuições RedHat, disponível no repositório oficial da ferramenta. Esse script instala todas as dependências necessárias, incluindo Docker, Node.js e NPM, além de clonar o repositório completo.

1. Clone o repositório ou acesse diretamente o arquivo de instalação em: `https://github.com/dnredson/P4Docker`
2. Conceda permissão de execução ao script:

```
chmod +x install.sh
```

3. Execute o script com permissões administrativas:

```
./install.sh
```

Após a instalação, acesse a pasta *P4Docker* e execute o servidor backend com o comando:

```
cd P4Docker
sudo node app.js
```

A interface gráfica estará disponível em: `http://localhost:3000`

Alternativamente, também é possível utilizar uma imagem virtual pré-configurada no formato OVA[1] com usuário e senha padrão `p4d`.

4.7. Permissões para execução automática com sudo

Como o backend do P4Docker interage diretamente com o Docker Engine, é necessário que o processo `app.js` seja executado com permissões de superusuário. Para facilitar a execução sem necessidade de digitar a senha toda vez, pode-se configurar o usuário atual para executar o comando `node app.js` com privilégios sudo sem senha.

Edite o arquivo de configuração do `sudoers` com o comando:

```
sudo visudo
```

E adicione a seguinte linha ao final do arquivo, substituindo `seu_usuario` pelo nome do seu usuário no sistema:

```
seu_usuario ALL=(ALL) NOPASSWD: /usr/bin/node /caminho/para/app.js
```

Essa configuração permite que o comando seja executado com sudo sem solicitar senha, garantindo o funcionamento adequado da interface sem intervenção manual. Certifique-se de ajustar o caminho conforme a localização real do `app.js` em seu sistema.

5. P4: da Teoria à Prática

Esta seção explora a criação, edição e compilação de códigos P4 utilizando o ambiente integrado do P4Docker. Para iniciar, certifique-se de que o P4Docker está instalado e em execução no seu sistema. A interface gráfica pode ser acessada através do navegador web no endereço `http://localhost:3000`. Para mais detalhes e tutoriais adicionais, consulte a documentação oficial do P4Docker[2].

[1] disponível em: `hhttps://drive.google.com/file/d/1Hlv1EAcj4YnxV7yyqgx41kiPIiU8yWWm/view?usp=sharing`

[2] disponível em: `https://dnredsons-organization.gitbook.io/p4docker`

5.1. Escrevendo e compilando códigos P4

Nesta etapa, será abordado o processo de compilação de um código P4 utilizando o compilador p4c, integrado ao P4Docker. O ambiente gráfico permite que o código seja compilado com um único clique, sem necessidade de utilizar a linha de comando. As etapas para compilação do código P4 são, na sequência:

1. Acesse a interface gráfica do P4Docker em `http://localhost:3000`.
2. Clique na opção “Compiler” no menu superior.
3. Insira o título do código no campo “Title”, como em “exemplo”
4. No editor de código, insira o conteúdo do programa P4 conforme o exemplo utilizado neste minicurso (ver código 3.2).
5. Clique em Compilar.
6. Caso o processo de compilação seja bem-sucedido, aparecerá uma mensagem na tela de Output informando que o código está compilado com sucesso.
7. O código `exemplo.json` será salvo no volume compartilhado para uso nos switches p4.

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

const bit<16> TYPE_IPV4 = 0x800;
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;
}

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){
            TYPE_IPV4: ipv4;
            default: accept;

        }

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

}


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

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

    action drop() {
        mark_to_drop(standard_metadata);
    }

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

    }

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

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

        }
    }
}

control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {
    apply {  }
}
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);
    }
}
control MyDeparser(packet_out packet, in headers hdr) {
```

```
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);

    }
}

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

Listing 3.2: Exemplo básico de programa P4

O código 3.2 utilizado neste primeiro exemplo implementa um switch programável simples com suporte ao encaminhamento de pacotes IPv4. Ele define dois cabeçalhos, Ethernet e IPv4, que serão extraídos dos pacotes recebidos para análise. O `parser` identifica se o pacote é IPv4 e, nesse caso, extrai também o cabeçalho IP. Após isso, o controle de ingresso processa o pacote com base em uma tabela chamada `ipv4_lpm`, que associa endereços IP de destino a ações de encaminhamento. Se houver uma correspondência na tabela, a ação `ipv4_forward` é aplicada, modificando os endereços MAC do pacote, definindo a porta de saída e decrementando o TTL. Se não houver regra definida, o pacote é descartado. O controle de egresso está presente, mas não realiza nenhuma modificação neste exemplo. O `deparser` é responsável por remontar os cabeçalhos no pacote antes que ele seja enviado. A arquitetura V1Switch conecta todos esses blocos, formando o pipeline completo. Esse código será utilizado na prática para permitir a comunicação entre dois hosts em uma topologia criada no P4Docker, com inserção de regras via terminal.

5.2. Criando infraestruturas utilizando a *dashboard*

Com o código P4 já compilado, a próxima etapa consiste na criação da topologia de rede. Neste exemplo, será criada uma infraestrutura simples com dois hosts conectados por um único switch programável. Na tela inicial do P4Docker, clique em "Add Node"e selecione a opção "Host". Para o primeiro host, use o nome “h1”, defina apenas uma porta e utilize a imagem Docker "`dnredson/net`". Repita o processo para criar o “h2”. Em seguida, adicione um switch, selecione a opção "Switch"e configure o nome como “sw1”, defina 2 portas, e informe o nome do arquivo gerado na compilação (`exemplo.json`). No campo "API Port"adicione a porta "50001". No campo "Entries", insira os comandos para preencher a tabela do plano de dados, como:

```
table_add MyIngress.ipv4_lpm ipv4_forward 10.0.1.2 => 00:00:00:00:01:02 1;
table_add MyIngress.ipv4_lpm ipv4_forward 10.0.2.2 => 00:00:00:00:02:02 2
```

As regras informam ao switch como proceder para o encaminhamento de pacotes baseando-se no IP de destino. Pacotes para `10.0.1.2` serão enviados pela porta 1 com destino ao MAC `00:00:00:00:01:02`, e pacotes para `10.0.2.2` seguirão pela porta 2. As linhas devem ser inseridas separadas por “;”. Depois de posicionar os nós, a topologia será adicionada ao *canvas* como ilustrado na Figura 3.16. O próximo passo é adicionar as arestas e criar a comunicação.

Clique em "Add Edge"para conectar os hosts ao switch. Conecte o host1 à porta 1 do switch, atribuindo o endereço IP `10.0.1.2/24` e MAC `00:00:00:00:01:02`

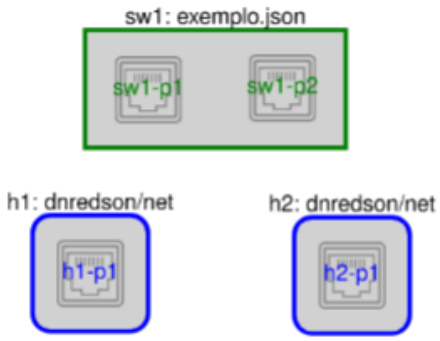


Figura 3.16: Topologia inicial para o Exemplo 1 - Comunicação entre dois hosts

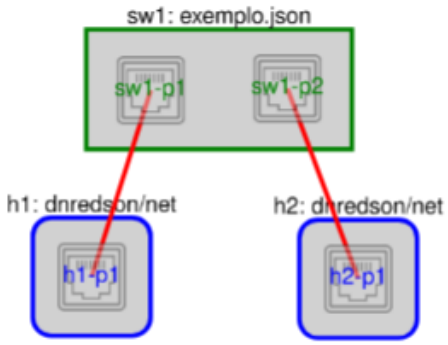


Figura 3.17: Topologia completa para o Exemplo 1 - Comunicação entre dois hosts

para o host, e IP `10.0.1.1/24` e MAC `00:00:00:00:01:01` para o switch. Faça o mesmo com o host2 e a porta 2 do switch, atribuindo o endereço IP `10.0.2.2/24` e endereço MAC `00:00:00:00:02:02` para o host, e IP `10.0.2.1/24` e MAC `00:00:00:00:02:01` para o switch.

Com os nós e conexões criados, a infraestrutura estará pronta, como ilustrado na Figura 3.17 para execução. Esta interface permite ainda salvar e exportar a topologia, facilitando testes futuros e reuso em diferentes experimentos.

5.3. Implementação, remoção e alteração de ambientes de teste

Após a criação da infraestrutura com os hosts e switches conectados, o ambiente de teste pode ser iniciado diretamente pela interface do P4Docker. O primeiro passo é utilizar os botões do menu inferior da *dashboard*: clique em "Save Topology" para armazenar a topologia atual, em "Generate Code" para gerar o script de configuração *exemploConfig.sh*, responsável por criar os containers, `namespaces` e interfaces de rede, e em "Generate Cleaner Code" para gerar o script `cleanexemplo.sh`, que remove e limpa o ambiente.

Os *scripts* gerados podem ser executados diretamente pelo terminal Linux. Para isso, é necessário primeiro atribuir permissões de execução com os comandos:

```
chmod +x exemploConfig.sh
chmod +x cleanexemplo.sh
```

Após salvar os arquivos, clique em “Deploy” para que o projeto seja registrado e possa ser gerenciado através da aba “Projects”, localizada no menu superior da interface. Essa etapa é fundamental para que o ambiente esteja disponível futuramente, já que projetos executados apenas via terminal, sem uso da opção “Deploy”, não são listados automaticamente na interface gráfica.

Na tela “Projects”, todos os projetos salvos via *dashboard* estarão listados. Selecione o projeto recém-criado, por exemplo, “exemplo”, e clique em “Load Project”. A topologia será carregada na tela, e os nós aparecerão inicialmente em vermelho, indicando que os containers ainda não estão em execução.

Se o P4Docker estiver configurado com permissões administrativas, basta clicar em “Deploy Project” para iniciar a criação do ambiente diretamente pela interface gráfica. Caso contrário, o script `exemploConfig.sh` deverá ser executado manualmente com permissões de super usuário:

```
sudo ./exemploConfig.sh
```

Como a configuração envolve a criação de `namespaces` e o uso do Docker Engine, é necessário utilizar o comando com `sudo` para que as permissões do sistema operacional sejam respeitadas.

A Figura 3.18 ilustra a topologia criada com base nas configurações definidas.

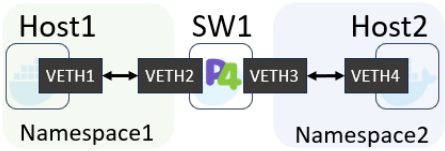


Figura 3.18: Topologia criada pelo P4Docker

Após o início do ambiente, atualize a página e recarregue o projeto. Os nós serão exibidos em verde, indicando que os containers estão ativos e em funcionamento. Na primeira execução, esse processo pode levar alguns minutos, especialmente se as imagens dos containers ainda precisarem ser baixadas.

Ao clicar em um nó ativo, o botão “Connect” ficará disponível no painel lateral, permitindo acesso direto ao terminal daquele container. Por exemplo, clique sobre o nó “h1” e selecione “Connect” para abrir seu terminal em uma nova aba do navegador.

Caso seja necessário interromper a execução de um container específico, clique em “Stop”. Isso pausará apenas aquele container, sem afetar os demais elementos da topologia. Para limpar completamente o ambiente, clique em “Clean Project”, o que encerrará todos os containers, redes e configurações associadas ao projeto. Caso as permissões administrativas não estejam ativas na interface, a limpeza também pode ser feita manualmente com o comando:

```
sudo ./cleanexemplo.sh
```

Além disso, o P4Docker permite editar o ambiente mesmo após sua criação. É

possível alterar nomes, imagens, endereços IP e até substituir o arquivo P4 de um switch por outro compilado.

5.4. Gerenciando o ambiente de teste em tempo de execução

Uma das vantagens do P4Docker é a capacidade de controlar os containers individualmente. É possível, por exemplo, parar apenas um dos hosts ou um switch específico sem afetar o restante da topologia. Isso permite simular falhas, reinicializações parciais ou reconfigurações pontuais, proporcionando maior controle sobre cada componente do ambiente. Essa granularidade é especialmente útil para testes de resiliência, depuração de comportamentos isolados ou para o ensino de conceitos como tolerância a falhas em redes programáveis.

Com o ambiente configurado e operacional, selecione o nó "sw1"que representa o switch para que o painel lateral de informações seja atualizado. Clique na opção "Connect"para abrir um terminal conectado diretamente ao container do switch. Repita o processo para os nós "h1"e "h2", abrindo uma nova aba de terminal para cada um deles.

Por padrão, os switches BMv2 criados no P4Docker mantêm seus *logs* ativos no diretório "`/tmp/switch.log`". Para acompanhar os eventos em tempo real, acesse o terminal do container "sw1"e execute o comando:

```
tail -f /tmp/switch.log
```

Em seguida, vá até a aba correspondente ao terminal do host "h1"e envie pacotes ICMP com o comando:

```
ping -c 10 10.0.2.2
```

Esse comando envia quatro pacotes para o endereço IP de "h2". Retornando ao terminal do switch, será possível visualizar em tempo real o registro da chegada, do processamento e do encaminhamento dos pacotes, evidenciando o funcionamento interno de um switch programável controlado por P4.

5.5. Aplicação: Comunicação entre dois hosts utilizando entradas de *Control Plane* e *Data Plane*

No exemplo da seção 5.3, os comandos "`table_add`"foram utilizados para configurar a tabela de encaminhamento diretamente pelo plano de controle. Essa abordagem reflete o uso tradicional de redes SDN, onde a lógica de processamento está no switch, mas as decisões de encaminhamento são definidas por controladores externos.

Entretanto, graças à flexibilidade da linguagem P4, esse mesmo comportamento pode ser implementado exclusivamente no plano de dados. Ou seja, o próprio programa P4 pode conter a lógica necessária para decidir o encaminhamento dos pacotes, sem depender de entradas fornecidas dinamicamente em tempo de execução.

O código da Listagem 3.3 apresenta uma versão modificada do exemplo anterior, onde o encaminhamento é feito diretamente com base nos valores dos cabeçalhos, eliminando a necessidade de configurar a tabela via terminal.

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

```
4
5  const bit<16> TYPE_IPV4 = 0x800;
6  typedef bit<9>  egressSpec_t;
7  typedef bit<48> macAddr_t;
8  typedef bit<32> ip4Addr_t;
9
10 header ethernet_t {
11     macAddr_t dstAddr;
12     macAddr_t srcAddr;
13     bit<16>   etherType;
14 }
15
16 header ipv4_t {
17     bit<4>    version;
18     bit<4>    ihl;
19     bit<8>    diffserv;
20     bit<16>   totalLen;
21     bit<16>   identification;
22     bit<3>    flags;
23     bit<13>   fragOffset;
24     bit<8>    ttl;
25     bit<8>    protocol;
26     bit<16>   hdrChecksum;
27     ip4Addr_t srcAddr;
28     ip4Addr_t dstAddr;
29 }
30
31 struct metadata {
32     /* empty */
33 }
34
35 struct headers {
36     ethernet_t   ethernet;
37     ipv4_t       ipv4;
38 }
39
40 parser MyParser(packet_in packet,
41                 out headers hdr,
42                 inout metadata meta,
43                 inout standard_metadata_t standard_metadata) {
44
45     state start {
46
47         packet.extract(hdr.ethernet);
48         transition select(hdr.ethernet.etherType){
49             TYPE_IPV4: ipv4;
50             default: accept;
51         }
52     }
53     state ipv4 {
54         packet.extract(hdr.ipv4);
55         transition accept;
56     }
57
58 }
59
60 control MyVerifyChecksum(inout headers hdr, inout metadata meta) {
61     apply {  }
62 }
63
64 control MyIngress(inout headers hdr,
65                   inout metadata meta,
66                   inout standard_metadata_t standard_metadata) {
67
68     action drop() {
69         mark_to_drop(standard_metadata);
70     }
71
72     action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
73
74
75         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
76         hdr.ethernet.dstAddr = dstAddr;
77         standard_metadata.egress_spec = port;
78         hdr.ipv4.ttl = hdr.ipv4.ttl -1;
79
80     }
81
82     table ipv4_lpm {
83         key = {
84             hdr.ipv4.dstAddr: exact;
85         }
86         actions = {
```

```
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    apply {
        if (hdr.ipv4.isValid()){
            ipv4_lpm.apply();
              if (hdr.ipv4.dstAddr == 0x0a000102) {
              ipv4_forward(0x000000000102, 1);
   } else if (hdr.ipv4.dstAddr == 0x0a000202) {
              ipv4_forward(0x000000000202, 2);
        }
        }
    }
}

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

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);
    }
}

control MyDeparser(packet_out packet, in headers hdr) {
    apply {

        //parsed headers have to be added again into the packet.
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);

    }
}

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

Listing 3.3: Código P4 para encaminhamento utilizando apenas o plano de dados

Embora a estrutura geral do código permaneça a mesma, a lógica dentro do bloco `apply` foi alterada. Após verificar que o pacote possui um cabeçalho IPv4 válido, o código inspeciona o campo `dstAddr` e identifica para qual endereço IP o pacote deve ser encaminhado. Quando o endereço de destino é reconhecido como `0x0a000102`, que representa o endereço `10.0.1.2` em formato hexadecimal, o pacote é enviado pela porta 1 com destino ao endereço MAC `0x000000000102`, que representa o endereço `00:00:00:00:01:02`; se for `0x0a000202`, segue pela porta 2 com destino ao endereço MAC `0x000000000202`, que representa o endereço `00:00:00:00:02:02`.

A função `ipv4_forward` é chamada diretamente com os parâmetros esperados, sem depender de qualquer entrada na tabela de encaminhamento.

Apesar de não ser a abordagem recomendada para redes reais — onde os dispositivos podem trocar de IP, e tabelas de encaminhamento precisam ser atualizadas dinamicamente — esse exemplo é útil para fins educacionais. Ele demonstra com clareza como um switch programável pode tomar decisões diretamente no plano de dados, sem qualquer interação com o plano de controle.

5.6. Aplicação: P4Checkers

O P4Checkers ([Trombeta et al. 2024]) é uma aplicação educacional interativa que integra conceitos de redes de computadores com o jogo de damas (checkers). Desenvolvido com o objetivo de tornar o ensino de redes mais atrativo, ele utiliza a linguagem P4 para interpretar ações de jogo como pacotes de rede, processando-as diretamente no plano de dados. A arquitetura da aplicação consiste em dois hosts e um switch P4, todos implementados como containers Docker no ambiente P4Docker.

Nesta atividade prática, os participantes executarão o P4Checkers utilizando o P4Docker, reforçando conceitos como manipulação de pacotes, uso de registradores e controle de fluxo diretamente no plano de dados.

Passo 1 – Compilação do código P4

Utilize o código do P4Checkers[3] e compile-o utilizando a interface do P4Docker. Nomeie o programa como `p4checkers`.

O código do P4Checkers implementa a lógica de um jogo de damas diretamente no plano de dados do switch P4. As principais funções e estruturas estão descritas a seguir:

- **Registradores:** Os registradores armazenam o estado do jogo diretamente no switch. Eles são definidos logo após as estruturas de cabeçalhos e são essenciais para manter a lógica do jogo entre os pacotes:

```
// 0 brancas - 1 pretas

register<bit<1>>(1) jogo_iniciado;
//verificador de jogo iniciado


register<bit<376>>(1) map;
```

Listing 3.4: Registradores do P4Checkers

- **Parser:** O parser extrai os campos relevantes dos pacotes recebidos e redireciona os pacotes UDP e TCP para os estados adequados. Ele prepara os dados para que a lógica do jogo possa atuar nos campos corretos dos pacotes. Um exemplo da transição após extração do header IPv4 é mostrado a seguir:

```
    state parse_ethernet {
        pkt.extract(hdr.ethernet);
        meta.parsed_bytes = meta.parsed_bytes + 14;
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
```

[3]disponível em: https://github.com/lucastrombet/P4CHECKERS/blob/main/main.p4

```
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        meta.parsed_bytes = meta.parsed_bytes + 20;
        transition select(hdr.ipv4.protocol){
            17 : parse_udp;
            6  : parse_tcp;
            default: chain_ipv4_raw;
        }
    }

    state parse_udp{
        pkt.extract(hdr.udp);
        meta.parsed_bytes = meta.parsed_bytes + 8;
        transition chain_ipv4_udp;
    }

    state parse_tcp{
        pkt.extract(hdr.tcp);
        meta.parsed_bytes = meta.parsed_bytes + 20;
        transition chain_ipv4_tcp;
    }
```

Listing 3.5: Transição do parser para pacotes UDP ou TCP

- **Ação SendBack:** Essa ação é utilizada para enviar uma resposta diretamente ao jogador, invertendo os campos de origem e destino dos *headers* Ethernet, IP e UDP/TCP. Isso permite que o switch responda imediatamente ao jogador que enviou uma jogada inválida ou incorreta:

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

    apply{
        standard_metadata.egress_spec = standard_metadata.ingress_port;

        if (hdr.ethernet.isValid()){
            bit<48> tmp = hdr.ethernet.srcAddr;
            hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
            hdr.ethernet.dstAddr = tmp;
        }

        if (hdr.ipv4.isValid()){
            bit<32> tmp = hdr.ipv4.src;
            hdr.ipv4.src = hdr.ipv4.dst;
            hdr.ipv4.dst = tmp;
        }

        if (hdr.udp.isValid()){
            bit<16> tmp = hdr.udp.srcPort;
            hdr.udp.srcPort = hdr.udp.dstPort;
            hdr.udp.dstPort = tmp;
            hdr.udp.csum = 0;
        }

        if (hdr.tcp.isValid()){
            bit<16> tmp = hdr.tcp.srcPort;
            hdr.tcp.srcPort = hdr.tcp.dstPort;
            hdr.tcp.dstPort = tmp;
            hdr.tcp.csum = 0;
        }

    }
```

Listing 3.6: Ação SendBack: resposta do switch para o remetente

- **Controle MyIngress:** O bloco de controle de entrada é o coração da lógica do jogo. Ele aplica a tabela de roteamento, executa a lógica dos registradores e inspeciona o conteúdo dos pacotes para validar e processar movimentos.

```
    apply {
/*
        portfwd.apply();
*/
        if (hdr.ipv4.isValid()){
            ipv4_lpm.apply();

        }
        #include "a_apply.p4"

        // erase checksums an update packet sizes
        if (hdr.tcp.isValid()){
            hdr.tcp.csum = 0;
        }
        else if (hdr.udp.isValid()){
            hdr.udp.csum = 0;
            hdr.udp.len = hdr.udp.len + meta.size_growth;
            hdr.udp.len = hdr.udp.len - meta.size_loss;
            if (meta.truncated_to!=0){
                hdr.udp.len = (bit<16>) meta.truncated_to - 34;
            }
        }
        if (hdr.ipv4.isValid()){
            hdr.ipv4.totalLen = hdr.ipv4.totalLen + meta.size_growth;
            hdr.ipv4.totalLen = hdr.ipv4.totalLen - meta.size_loss;
            if (meta.truncated_to!=0){
                hdr.ipv4.totalLen = (bit<16>) meta.truncated_to - 14;
            }
            // Note: checksum is updated later
        }

        if (meta.postprocessing==SENDBACK){
            SendBack.apply(hdr,meta,standard_metadata);
        }
        else if (meta.postprocessing==DROP){
            #ifdef TARGET_NFP
                mark_to_drop();
            #else
                mark_to_drop(standard_metadata);
            #endif
        }
    }
```

Listing 3.7: Trecho da lógica principal de movimentação no plano de dados

Esses blocos evidenciam como o P4Checkers implementa a lógica de um jogo interativo inteiramente no plano de dados, utilizando estruturas como registradores, inspeção de payloads e respostas imediatas aos jogadores. O uso de arquivos externos para modularizar a lógica do jogo também facilita a manutenção e expansão da aplicação.

Passo 2 – Criação da topologia

Na dashboard do P4Docker, crie uma nova topologia com os seguintes nós:

- `h1` – container com imagem <imagem> e IP `10.0.1.1/24`;
- `h2` – mesma imagem de h1, com IP `10.0.2.2/24`;
- `sw1` – switch programável P4 com o código `p4checkers.json`, duas portas.

Conecte `h1` à porta 1 de `sw1` e `h2` à porta 2. Nomeie o projeto como `p4checkers` e gere os scripts de configuração e limpeza.

Passo 3 – Execução do ambiente

Atribua permissão de execução ao script gerado e inicie o ambiente como superusuário:

```
chmod +x p4checkersConfig.sh
sudo ./p4checkersConfig.sh
```

Abra o terminal dos hosts pela interface do P4Docker e execute:

```
// No h1:
python3 p4checkers_player1.py

// No h2:
python3 p4checkers_player2.py
```

Movimentos são feitos no formato:

```
linha_origem coluna_origem linha_destino coluna_destino
Exemplo: 3 3 4 4
```

Passo 4 – Acompanhamento da execução

Use o log do switch para acompanhar o processamento:

```
tail -f /tmp/switch.log
```

Ou utilize o Wireshark com namespace:

```
PIDHost1=$(docker inspect -f '{{.State.Pid}}' h1)
sudo nsenter -t $PIDHost1 -n wireshark
```

Os pacotes são UDP com payload textual (ex: `3 3 4 4`). O switch P4 intercepta o pacote, utiliza registradores para verificar o turno (`checkers_turn`), se o jogo foi iniciado (`jogo_iniciado`) e o estado do tabuleiro (`map`). Movimentos inválidos são detectados e modificados com respostas como:

```
invalid move
not your turn
```

O comportamento é implementado diretamente no P4, demonstrando o poder da linguagem para manipulação de estado e conteúdo do pacote. Abaixo, o trecho de código que grava o estado do tabuleiro em registrador:

```
1 register<bit<376>>(1) map;
2 map.write(0, 0x2D322D322D320A302D302D302D0A...);
```

Esse exemplo evidencia como switches P4 podem atuar como elementos computacionais autônomos, sem depender de plano de controle, sendo capazes de executar lógicas complexas e interativas diretamente no plano de dados.

6. Discussão e Conclusão

6.1. Próximos passos da linguagem

A linguagem P4 tem se consolidado como um dos pilares da programabilidade em redes, especialmente com sua adoção em ambientes acadêmicos e o crescente interesse da indústria. Seu potencial vai além da simples reconfiguração de pipelines: P4 vem sendo

cada vez mais integrado a outras tecnologias emergentes, como SDN, computação na borda, IoT e *in-network computing*. O futuro da linguagem aponta para uma expansão ainda maior de seu ecossistema, com interfaces mais acessíveis, suporte a novos *targets* e integração facilitada com sistemas legados.

6.2. Dependência do suporte dos dispositivos alvo

Apesar das vantagens, a evolução da linguagem ainda depende fortemente do suporte dos dispositivos de rede programáveis. Muitos ambientes de produção ainda operam com equipamentos que não são compatíveis com P4, o que limita a adoção em larga escala. Essa dependência exige soluções híbridas e o uso de plataformas de simulação, como BMv2, até que o mercado amadureça. A padronização das interfaces e a evolução de arquiteturas abertas, como PISA, buscam mitigar esse gargalo.

6.3. Produção de hardware P4

A produção de hardware com suporte nativo a P4 ainda é restrita, mas iniciativas recentes têm buscado mudar esse cenário. Embora a Intel tenha descontinuado a linha Tofino, a iniciativa de abertura do design como projeto *open source* pode incentivar novas empresas e instituições a produzirem ASICs programáveis com suporte à linguagem. Além disso, iniciativas da Linux Foundation voltadas à padronização e desenvolvimento colaborativo indicam uma tendência de fortalecimento da comunidade em torno de P4.

6.4. DPU, GPU, IA e P4

Com os recentes avanços nas tecnologias de Inteligência Artificial, em especial com o lançamento de bibliotecas de *deep learning* como o pytorch entre outros, tornou-se possível o desenvolvimento de tecnologias de classificação que antes não eram possíveis ou práticas. Isso, juntamente com bibliotecas como o *GPUNetIO* da NVIDIA, torna possível um desenvolvimento integrado de aplicações que unem P4 e IA para roteamento, classificação, filtragem de pacotes entre outros. Além disso, o uso de GPUs para realização de processamentos massivamente paralelos apresenta notável vantagem quando comparados com CPUs tradicionais, proporcionando assim uma grande oportunidade de desenvolvimento.

6.5. Projeto PROFISSA

O projeto PROFISSA (Programa de Formação em Infraestruturas de Software, Segurança e Aplicações em Rede), financiado pela FAPESP e coordenado pela RNP, é um exemplo prático do investimento em tecnologias disruptivas para redes de computadores. Entre os diversos resultados obtidos pelo projeto, destaca-se a criação da ferramenta P4Docker, que democratiza o acesso à experimentação com a linguagem P4 por meio de containers. O projeto também fomenta o desenvolvimento de soluções de rede baseadas em programação de plano de dados, promovendo a formação de recursos humanos especializados e a consolidação da comunidade brasileira em torno de redes programáveis.

6.6. Conclusão

Ao longo deste minicurso, foram apresentados os fundamentos da programação do plano de dados com a linguagem P4, sua relação com arquiteturas SDN, e a evolução dessa abordagem ao longo da última década. Discutimos os principais avanços tecnológicos

que permitiram a consolidação da linguagem, explorando desde dispositivos programáveis como Intel Tofino e SmartNICs até ambientes de experimentação como o BMv2, Mininet-WiFi, SONiC e P4Pi.

A introdução da linguagem P4 representa um marco no desenvolvimento de redes programáveis, permitindo a criação de aplicações personalizadas diretamente no plano de dados. Essa capacidade impulsiona inovações em áreas como segurança, IoT, balanceamento de carga, interoperabilidade de protocolos, e até mesmo computação na rede, temas que foram brevemente discutidos ao longo do texto.

Dentro desse cenário, o P4Docker foi apresentado como uma solução acessível e eficiente para o ensino, experimentação e prototipagem de aplicações em P4. Ao utilizar contêineres Docker para simular topologias com switches P4, o P4Docker oferece maior isolamento, reprodutibilidade e escalabilidade em relação a soluções tradicionais. Sua interface gráfica facilita a criação de topologias, compilação de programas e visualização de logs, contribuindo significativamente para o aprendizado e o desenvolvimento de novas soluções em redes programáveis.

Como trabalhos futuros, destacam-se três direções principais. A primeira envolve o avanço na usabilidade de ambientes como o P4Docker, especialmente na integração com sistemas de visualização de métricas, suporte nativo a múltiplos targets, e debug gráfico. A segunda é o aprofundamento no uso da linguagem P4 para contextos específicos, como segurança distribuída, aplicações industriais e redes 6G. Por fim, a terceira frente propõe a criação de plataformas colaborativas que fomentem o ensino de P4 por meio de cursos online, repositórios de código aberto e laboratórios virtuais baseados em contêineres.

Consolidar o conhecimento adquirido e fomentar uma comunidade ativa em torno de P4 é essencial para que essa tecnologia continue evoluindo. Espera-se que os participantes deste minicurso estejam aptos a explorar, implementar e compartilhar suas próprias soluções, contribuindo para o crescimento da área e o avanço da próxima geração de redes inteligentes e adaptativas.

7. Agradecimentos

Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) sob processo número 2020/05152-7 (projeto PROFISSA). Também faz parte do INCT de Redes Inteligentes de Comunicações e Internet das Coisas Inteligentes (ICoNIoT), financiado pelo CNPq (proc. 405940/2022-0) e pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES), Código Financeiro 88887.954253/2024-00.

8. Bibliografia

Referências

[Alsaeedi et al. 2019] Alsaeedi, M., Mohamad, M. M., and Al-Roubaiey, A. A. (2019). Toward adaptive and scalable openflow-sdn flow control: A survey. *IEEE Access*, 7:107346–107379.

[Bal et al. 2024] Bal, S., Han, Z., Handagala, S., Cevik, M., Zink, M., and Leeser, M.

(2024). P4-based in-network telemetry for fpgas in the open cloud testbed and fabric. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6.

[Berde et al. 2014] Berde, P. et al. (2014). Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6.

[Blanco et al. 2017] Blanco, B., Fajardo, J. O., Giannoulakis, I., Kafetzakis, E., Peng, S., Pérez-Romero, J., Trajkovska, I., Khodashenas, P. S., Goratti, L., Paolino, M., Sfakianakis, E., Liberal, F., and Xilouris, G. (2017). Technology pillars in the architecture of future 5g mobile networks: Nfv, mec and sdn. *Computer Standards Interfaces*, 54:216–228. SI: Standardization SDNNFV.

[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. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[Braun and Menth 2014] Braun, W. and Menth, M. (2014). Software-defined networking using openflow: Protocols, applications and architectural design choices. *Future Internet*, 6(2):302–336.

[Burstein 2021] Burstein, I. (2021). Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE.

[Carvalho 2024] Carvalho, B. (2024). Um gateway iot programável com p4 para interoperabilidade e processamento inteligente de tráfego. Dissertação de mestrado, Universidade Federal do ABC (UFABC), Santo André, SP. Disponível mediante solicitação ou repositório institucional da UFABC.

[Chen et al. 2023] Chen, G., Hu, Z., and Jin, D. (2023). Enhancing fidelity of p4-based network emulation with a lightweight virtual time system. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '23, page 34–43, New York, NY, USA. Association for Computing Machinery.

[Consortium 2023] Consortium, P. L. (2023). P4runtime specification. `https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html`.

[Consortium 2024] Consortium, P. L. (2024). BMV2: Behavioral Model version 2 (P4 Runtime environment). Git repository.

[Ding et al. 2022] Ding, D., Savi, M., Pederzolli, F., and Siracusa, D. (2022). Design and development of network monitoring strategies in p4-enabled programmable switches. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6.

[dos Reis Fontes and Rothenberg 2015] dos Reis Fontes, R. and Rothenberg, C. E. (2015). Mininet-wifi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 384–389.

[Franco et al. 2024] Franco, D., Ollora Zaballa, E., Zang, M., Atutxa, A., Sasiain, J., Pruski, A., Rojas, E., Higuero, M., and Jacob, E. (2024). A comprehensive latency profiling study of the tofino p4 programmable asic-based hardware. *Computer Communications*, 218:14–30.

[Gao et al. 2021] Gao, S., Handley, M., and Vissicchio, S. (2021). Stats 101 in p4: Towards in-switch anomaly detection. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, HotNets '21, page 84–90, New York, NY, USA. Association for Computing Machinery.

[Goswami et al. 2023] Goswami, B., Kulkarni, M., and Paulose, J. (2023). A survey on p4 challenges in software defined networks: P4 programming. *IEEE Access*, 11:54373–54387.

[Heideker et al. 2025] Heideker, A., Silva, D., Kamienski, C. A., and Trotta, A. (2025). Achieving seamless iot interoperability through data plane programmability. In *2025 IEEE 22st Consumer Communications Networking Conference (CCNC)*, pages 1–6.

[Heideker et al. 2023] Heideker, A., Silva, D., Kleinschmidt, J., and Kamienski, C. (2023). Otimização de tráfego iot-lorawan usando programação de plano de dados em p4. In *Anais do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 239–252, Porto Alegre, RS, Brasil. SBC.

[Ibanez et al. 2019] Ibanez, S., Brebner, G., McKeown, N., and Zilberman, N. (2019). The P4→NetFPGA workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 1–9.

[Intel 2022] Intel (2022). Intel® tofino™ series. `https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html`.

[Ion et al. 2020] Ion, M., Scholz, S., and Dumitrescu, C. (2020). Programming the network data plane with p4runtime: Concepts and tools. In *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks*, pages 155–160.

[Kaur et al. 2025] Kaur, A., Rama Krishna, C., and Patil, N. V. (2025). A comprehensive review on software-defined networking (sdn) and ddos attacks: Ecosystem, taxonomy, traffic engineering, challenges and research directions. *Computer Science Review*, 55:100692.

[Ke and Hsu 2020] Ke, C.-H. and Hsu, S.-J. (2020). Load balancing using p4 in software-defined networks. *Journal of Internet Technology*, 21(6):1671–1679.

[Kfoury et al. 2021a] Kfoury, E., Elgendy, I., and Jararweh, Y. (2021a). P4 in iot: A survey on data plane programmability and applications. *Computer Networks*, 190:107930.

[Kfoury et al. 2024] Kfoury, E. F., Choueiri, S., Mazloum, A., AlSabeh, A., Gomez, J., and Crichigno, J. (2024). A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access*, 12:107297–107336.

[Kfoury et al. 2021b] Kfoury, E. F., Crichigno, J., and Bou-Harb, E. (2021b). An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155.

[Kim et al. 2021] Kim, H. et al. (2021). Experience-driven research on programmable networks. *ACM SIGCOMM Computer Communication Review*, 51(1):10–17.

[Kulkarni et al. 2022] Kulkarni, M., Goswami, B., and Paulose, J. (2022). P4 based load balancing strategies for large scale software-defined networks. In *2022 Second International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, pages 1–7.

[Laki et al. 2021] Laki, S., Kis, Z., Keresztesi, R., Horpácsi, D., and Szabó, R. (2021). P4pi: P4 on raspberry pi for flexible home and iot networking. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*.

[Lantz et al. 2010] Lantz, B., Heller, B., and McKeown, N. (2010). A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6.

[Luizelli et al. 2024] Luizelli, M. C., Vogt, F., de Matos, G. M. V., Cordeiro, W., Schaeffer Filho, A. E., RNP, M. S., Verdi, F. L., and Rothenberg, C. E. (2024)). Smartnics: The next leap in networking. In *Minicursos do 42. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2024)*.

[Makde and Laxkar 2024] Makde, R. R. and Laxkar, P. R. (2024). A comprehensive introduction to sdn architectural foundations. *International Journal of Science and Research (IJSR)*, 13(2).

[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. *ACM SIGCOMM computer communication review*, 38(2):69–74.

[Microsoft 2023] Microsoft (2023). Sonic: Software for open networking in the cloud.

[Mininet 2023] Mininet (2023). Mininet: An instant virtual network on your laptop (or other pc). [Online; accessed 14-January-2024].

[Moskowitz et al. 2023] Moskowitz, I. S., Baldine, I., Nikolich, A., Ruth, P., Liu, J., and Chase, J. (2023). Fabric: A national-scale programmable research infrastructure. In *Proceedings of the 2023 ACM SIGCOMM Conference*, pages 37–44.

[P4 Language Consortium 2022] P4 Language Consortium (2022). The p4 language specification. `https://p4.org/p4-spec/docs/P4-16-spec.html`. Version 1.2.3.

[P4 Language Consortium 2024] P4 Language Consortium (2024). P4 language specifications. `https://p4.org/specs/`. Accessed: 2024-07-11.

[Santos et al. 2021] Santos, J., Costa, M., and Almeida, F. (2021). P4 and iot: A survey on data plane programmability for smart systems. *Journal of Network and Systems Management*, 29(4):1–24.

[Silva et al. 2024] Silva, D., Heideker, A., Trombeta, L., Carvalho, B., Kleinschmidt, J., and Kamienski, C. (2024). P4docker: Enabling efficient p4 switch testbeds with docker integration. In *Anais Estendidos do XLII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 1–8, Porto Alegre, RS, Brasil. SBC.

[Smyth et al. 2023] Smyth, D., Scott-Hayward, S., Cionca, V., McSweeney, S., and O'Shea, D. (2023). Secap switch—defeating topology poisoning attacks using p4 data planes. *Journal of Network and Systems Management*, 31(1):28.

[Souza et al. 2024] Souza, J., Oliveira, M., and Silva, P. (2024). P7: Patch panel programável com p4. Projeto apresentado no Workshop de Ferramentas do SBRC.

[Trombeta et al. 2024] Trombeta, L., Da Silva, D. E. O. G., Carvalho, B. C., Anon Da Silva, F. A., Kleinschmidt, J. H., and Kamienski, C. A. (2024). P4Checkers: Exploring Modern Network Concepts Through a Classic Game . In *2024 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, Los Alamitos, CA, USA. IEEE Computer Society.

[Vuckovic et al. 2021] Vuckovic, P., Santic, J., and Tomašić, D. (2021). Integrating p4runtime with onos for real-time sdn control. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 127–134.

[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)*, pages 122–135.