UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Daniel Batista de Lima

RESISTING: Um Novo Mecanismo de Fast-Reroute com Distribuição de Pacotes em Switches Programáveis P4

Campinas

2024

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Daniel Batista de Lima

RESISTING: Um Novo Mecanismo de Fast-Reroute com Distribuição de Pacotes em Switches Programáveis P4

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação

Orientador: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Daniel Batista de Lima, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

Campinas

2024

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Elizangela Aparecida dos Santos Souza - CRB 8/8098

L628r

Lima, Daniel Batista de, 1981-
Resisting : um novo mecanismo de fast-reroute com distribuição de pacotes em switches programáveis P4 / Daniel Batista de Lima. – Campinas, SP : [s.n.], 2024.

Orientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Redes de computadores. 2. Linguagem de programação. I. Esteve Rothenberg, Christian Rodolfo, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações Complementares

Título em outro idioma: Resisting : a new fast-reroute mechanism with packet distribution on p4-programmable switches
Palavras-chave em inglês:
Computer networks
Programming language
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Fábio Luciano Verdi
Weverton Luis da Costa Cordeiro
Data de defesa: 07-02-2024
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0009-0006-6280-3345
- Currículo Lattes do autor: http://lattes.cnpq.br/2981805403333508

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Daniel Batista de Lima - **RA:** 208925
Data da Defesa: 7 de Fevereiro de 2024
Título da Tese: "RESISTING: Um Novo Mecanismo de Fast-Reroute com Distribuição de Pacotes em Switches Programáveis P4"

Prof. Dr. Christian Rodolfo Esteve Rothenberg (FEEC/UNICAMP)
Prof. Dr. Fábio Luciano Verdi (UFSCAR)
Prof. Dr. Weverton Luis da Costa Cordeiro (UFRGS)

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

Dedico esta tese à todo mundo.

Agradecimentos

Primeiramente, gostaria de expressar minha gratidão a Deus por me conceder saúde e disposição para vivenciar esta incrível experiência em minha carreira. Seis anos de trabalho e dedicação passaram como um "piscar de olhos". Durante esta jornada, fui privilegiado em conhecer pessoas que contribuíram para o meu desenvolvimento pessoal, técnico, bem como para o sucesso deste trabalho. Gostaria de expressar meu profundo agradecimento ao Professor Dr. Christian Rodolfo Esteve Rothenberg por proporcionar a oportunidade de atuar neste trabalho, por sua orientação, paciência e apoio constante. Suas valiosas recomendações e conhecimento profundo foram essenciais para a construção deste trabalho. Aos meus colegas de pesquisa e amigos, Fabricio Rodriguez Cesen, Francisco Germano Vogt e Alan Teixeira, agradeço pelas discussões estimulantes, colaboração e amizade.

Finalmente, mas não menos importante, quero dedicar um profundo agradecimento à minha família. O apoio e compreensão que recebi de meus pais e demais familiares foram pilares essenciais durante toda a minha jornada acadêmica. Suas palavras de encorajamento nos momentos difíceis e sua celebração nas vitórias fizeram toda a diferença. Esse sucesso também pertence a vocês. Não posso deixar de mencionar a instituição que tornou esta jornada possível. A Universidade Estadual de Campinas, junto com a Faculdade de Engenharia Elétrica e de Computação forneceram recursos, orientação e um ambiente propício para a realização deste trabalho. Agradeço a todos desta instituição.

"Mantenha-se faminto por coisas novas, mantenha-se certo de sua ignorância."

Steve Jobs

Resumo

Mecanismos Fast-Reroute (FRR) desempenham um papel fundamental na recuperação de falhas em diversas arquiteturas de redes, evitando a degradação dos fluxos de pacotes das aplicações. No entanto, os trabalhos que suportam mecanismos de FRR na arquitetura de switches programáveis na linguagem Programming Protocol-independent Packet Processor (P4) aplicam uma abordagem de recuperação de falhas sem considerar um método de distribuição de fluxos de pacotes. Como resultado, em cenários que envolvem o balanceamento de tráfego, essas soluções não contribuem para a utilização eficiente dos recursos de banda, podendo causar desequilíbrio na distribuição dos fluxos de pacotes e aumentar o risco de sobrecarga nos links durante eventos de falha. O presente trabalho propõe `RESISTING` como um novo mecanismo de FRR Equal-Cost Multi-Path (ECMP) para switches programáveis em P4, oferecendo uma distribuição equilibrada dos fluxos de pacotes entre os links operacionais utilizados no balanceamento de tráfego após a recuperação de falhas. O mecanismo de recuperação proposto é comparado ao *Primitive for Reconfigurable Fast Reroute* (PURR) – mecanismo considerado o estado da arte – durante os eventos de uma, duas e três falhas. Os resultados mostram que o método proposto não apresenta perda de fluxos de pacotes durante a avaliação experimental, ao passo que o PURR apresenta perdas a partir de duas falhas simultâneas.

Palavras-chaves: P4, SDN, FRR, Tolerância a Falhas, Convergência de Rede, Rede de Computadores.

Abstract

Fast-Reroute (FRR) mechanisms play a critical role in failure recovery in different types of network architectures, avoiding application flow performance degradation. However, Current FRR mechanisms for P4 programmable switches provide fast reroute approaches without considering flow load balancing support. Consequently, in scenarios involving traffic balancing, these solutions do not deliver efficient use of bandwidth resources, resulting in packet flow imbalance and overloaded network links during failure events. This work proposes `RESISTING` as a new FRR Equal-Cost Multi-Path (ECMP) mechanism for Programming Protocol-independent Packet Processor (P4) programmable switches, delivering a packet flow load balancing after failure recovery. The proposed method is compared against Primitive for Reconfigurable Fast Reroute (PURR) – the state-of-the-art FRR mechanism in P4 – under one, two, and three links failure experiments. The results show that the proposed prototype does not incur packet losses during the experimental evaluation, while PURR presents losses under two or more simultaneous failures.

Keywords: P4, SDN, FRR, Fast-Failover, Network Convergence, Computer Networks.

Lista de ilustrações

Lista de tabelas

Lista de Acrônimos e Abreviações

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BFD Bi-direcional Detection Forwarding

BGP Border Gateway Protocol

BIER-FRR Bit Indexed Explicit Replication Fast Reroute

BIER Bit Indexed Explicit Replication

BMv2 Behavioral Model Version 2

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DPDK Data Plane Development Kit

ECMP Equal Cost Multi Path

FCT Flow Completion Time

FPGA Field Programmable Gate Arrays

FRR Fast-Reroute

IETF Internet Engineering Task Force

INT Inband Network Telemetry

IP-FRR Internet Protocol Fast Reroute

IP Internet Protocol

LFA Loop-Free Alternate

LPM Lowest Prefix Match

MAC Media Access Control

MPLS Multiprotocol Label Switching

NAT Network Address Translation

NIC Network Interface Card

OSPF Open Shortest Path First

P4 Programming Protocol-Independent Packet Processors

PIM Protocol Independent Multicast

PURR Primitive for Reconfigurable Fast Reroute

RAM Random Access Memory

RESISTING datacenteR Equally coSt multIpath faSt re-rouTING

RMT Reconfigurable Match Tables

RX Receive

SDN Software Defined Networking

SFP Small form-factor pluggable

SRAM Static Random Access Memory

TCAM Ternary Content Addressable Memory

TCP Transmission Control Protocol

TX Transmit

UDP User Datagram Protocol

VL2 Virtual Layer 2

WAN Wide Area Network

Sumário

1 Introdução

Atualmente, a arquitetura de rede tradicional impõe dificuldades na evolução de novas soluções e aplicações na área de redes de computadores, uma vez que é construída com base em dispositivos de redes convencionais de tecnologia proprietária, que utilizam *chips* limitados e inflexíveis em termos de funcionalidade de rede (GOBATTO *et al.*, 2021). Por outro lado, a arquitetura *Software-Defined Networking* (SDN) (KREUTZ *et al.*, 2014), a linguagem *Programming Protocol-Independent Packet Processors* (P4) (BOSSHART *et al.*, 2014) e o plano de dados programável são tecnologias que promovem inovação, adaptação e flexibilidade nas redes de computadores. Juntas, possibilitam o desenvolvimento de soluções para atender a uma sociedade cada vez mais exigente em termos de desempenho e adaptabilidade.

À medida que as três tecnologias mencionadas evoluem, novas aplicações surgem para atender às demandas dos serviços de instituições públicas e privadas. Neste contexto, a área de redes de computadores desempenha um papel fundamental na implementação dessas aplicações e na oferta de conectividade confiável aos usuários finais. Destaca-se, atualmente, a diversidade de aplicações disponíveis no planos de dados programável na linguagem P4, abrangendo funcionalidades como roteamento baseado em endereços das camadas 2, 3, *Multiprotocol Label Switching* (MPLS), *Multicast*, e outros tipos de serviços, incluindo *Network Address Translation* (NAT), filtros, *Sflow*, *Equal Cost Multi Path* (ECMP), *In-band Network Telemetry* (INT). Entretanto, a concepção de aplicações que sejam capazes de lidar com a tolerância à falhas representa um desafio significativo quando se trata da linguagem P4 (SEDAR *et al.*, 2018). Isso se deve ao fato de que os hardwares utilizados na implementação dessas aplicações não contam com uma funcionalidade de recuperação de falhas incorporada nativamente (CHIESA *et al.*, 2019), exigindo o esforço de desenvolvimento do mecanismo de recuperação na arquitetura P4.

Para lidar com tal limitação, os mecanismos *Fast-Reroute* (FRR) emergem como soluções eficazes para recuperação de falhas no plano de dados, já que promovem o re-roteamento rápido dos pacotes durante eventos de falhas, evitando recorrer a estratégias que envolvam o plano de controle. Isso resulta em tempos de recuperação mais rápidos, reduzindo a degradação dos fluxos de pacotes durante os eventos de falhas. A estratégia de tolerância à falhas precisa lidar com variados cenários de infraestrutura de rede, dentre os quais destaca-se o data center. Neste cenário, o intenso volume de tráfego arbitrário requer soluções de balanceamento de tráfego entre os *enlaces*, com o propósito de assegurar uma distribuição equilibrada dos fluxos de pacotes. O balanceamento de tráfego ECMP

ganhou relevância neste panorama, sobretudo na infraestrutura de nuvem (VERDI *et al.*, 2010), ao proporcionar a distribuição dos fluxos de pacotes entre *links* com o mesmo custo. Tal habilidade otimiza o uso dos recursos de banda e amplia a capacidade de comunicação entre servidores. Por essa perspectiva, as soluções de FRR, quando implementadas em infraestruturas de data center, devem idealmente, trabalhar em conjunto com as estratégias de balanceamento de trafego, a fim de combinar a propriedade de tolerâncias a falhas com uma distribuição equilibrada dos fluxos de pacotes entre os *enlaces*, garantindo assim a eficiência no uso dos recursos de banda.

Na arquitetura P4, uma solução simples de FRR (CHIESA *et al.*, 2023) consiste em recircular continuamente todos os fluxos de pacotes provenientes da porta que sofreu a falha para uma porta *backup* dentro do plano de dados. No entanto, a técnica de recirculação de pacotes continua pode causar aumento na latência dos pacotes e redução do *throughput*, principalmente à medida que a quantidade de falhas aumenta, ocasionando o incremento progressivo das recirculações de pacotes dentro do plano de dados (SEDAR *et al.*, 2018). Outra abordagem considerada é a implementação de um mecanismo FRR baseado em decisões de rotas alternativas, as quais são pré-configuradas no plano de dados. Neste método, para cada falha que ocorre, uma rota redireciona todos os fluxos de pacotes afetados pela falha para um caminho backup. Tal solução pode atender de maneira eficiente a um cenário simples de rede, como um roteador contendo um *link* principal e um *backup*. No entanto, em cenários mais complexos, como o ambiente de data center, onde o switch topo de *rack* pode conter múltiplos caminhos alternativos, possibilitando variações de falhas, essa estratégia de recuperação pode exigir a instalação de muitas regras de redirecionamento de pacotes para lidar com diferentes combinações de falhas. Isso pode resultar no consumo de recursos valiosos de memória no plano de dados para armazenar essas regras e contribuir no aumenta da complexidade no gerenciamento das regras, principalmente à medida que a quantidade de caminhos alternativos aumenta.

Os trabalhos (MERLING *et al.*, 2020; CHIESA *et al.*, 2019) apresentam abordagens de recuperação mais eficientes em comparação com as estratégias mencionadas, possibilitando a recuperação de falhas sem comprometer a latência e a vazão, além de evitar um alto consumo de recursos de memória nos dispositivos programáveis. Porém, tanto os mecanismos FRR mais simples como os mais sofisticados podem não ser indicados para um cenário de data center, visto que o processo de recuperação dessas soluções consiste em redirecionar todos os fluxos de pacotes afetados pela falha para um caminho backup, sem considerar a possibilidade de distribuição dos fluxos entre outras opções de *links* operacionais no ambiente. Como resultado, durante incidentes de falhas nos cenários de data center, se os referidos FRR foram implementados, o processo de recuperação pode causar desequilíbrio na distribuição dos fluxos de pacotes entre os *enlaces* do ECMP, con-

tribuindo para a sobrecarga no *link* backup (ZHANG *et al.*, 2008) e reduzindo o nível de resiliência da rede.

1.1 Proposta de Pesquisa

O presente trabalho propõe o RESISTING (*datacente**R** **E**qually co**S**t mult**I**path fa**S**t re-rou**TING***): um novo mecanismo de FRR, integrado com ECMP, para switches programáveis em P4 com o propósito de oferecer resiliência, robustez e distribuição dos fluxos de pacotes no data center após a recuperação dos eventos de falhas. Nosso mecanismo é capaz de se recuperar de uma ou várias falhas simultâneas nos *links* utilizados pelo balanceamento de tráfego ECMP. Para isso, após a detecção de uma falha em um *link* do balanceamento ECMP, inicia-se o processo de recuperação FRR da arquitetura. O objetivo é remover a porta de saída do switch associada ao *link* afetado pela falha, por meio de uma rotina de reordenação das portas dos *links* remanescentes no balanceamento. Após a conclusão do processo de recuperação, os fluxos de pacotes previamente afetados pela falha são redistribuídos de forma equilibrada entre os *links* operacionais do balanceamento. O RESISTING foi compilado na linguagem P4 para o switch *Barefoot Tofino* e o switch virtual *Behavioral Model Version 2* (BMv2).

Para avaliar a eficácia do mecanismo proposto, este trabalho conduziu dois experimentos distintos. No primeiro, com o propósito de analisar o desempenho do nosso mecanismo no switch Tofino, foi realizada a avaliação da quantidade de pacotes submetidos à recirculação temporária de três fluxos distintos, variando a taxa de transmissão, durante a ocorrência de uma, três e cinco falhas. Vale enfatizar que a técnica de recirculação contínua de pacotes aumenta o tempo de permanência do pacote dentro do plano de dados, resultando no aumento da latência e na redução da vazão (SEDAR *et al.*, 2018). Assim, consideramos a avaliação do desempenho do método proposto um fator relevante para apoiar a implementação da arquitetura em um ambiente operacional real no futuro. No segundo experimento, realizou-se uma análise de degradação dos fluxos de pacotes, estabelecendo uma comparação entre a implementação proposta e o *Primitive for Reconfigurable Fast Reroute* (PURR) (CHIESA *et al.*, 2019), mecanismo reconhecido como estado da arte, em eventos de até três falhas simultâneas. O objetivo principal foi avaliar o nível de resiliência dos mecanismos no software switch BMv2, a fim de demonstrar qual estratégia de recuperação seria mais apropriada para integrar com um serviço de balanceamento ECMP em um cenário de data center.

Os resultados do primeiro experimento indicaram que, mesmo durante o aumento da taxa de transmissão e a aplicação de falhas, o mecanismo RESISTING apresentou uma taxa mínima de $\approx 0\%$ de pacotes recirculados dos três tipos de fluxos em

relação à quantidade de pacotes gerados (por exemplo, 31 pacotes recirculados para um total de 91 milhões de pacotes gerados, no pior caso). Esses resultados mostraram que o processo de recuperação recirculou apenas um contingente mínimo de pacotes dos fluxos durante a ocorrência de falhas, sugerindo que a técnica de recirculação temporária não comprometeria o desempenho do fluxo de pacotes, causando um aumento significativo da latência e redução do *throughput*. Além disso, cabe ressaltar que o aumento na taxa de transmissão dos fluxos não resultou em um aumento proporcional na quantidade de pacotes submetidos à recirculação. Essa constatação sugere que taxas de transmissão superiores às avaliadas podem manter a estabilidade na quantidade de pacotes recirculados dos fluxos.

O segundo experimento foi conduzido no ambiente virtual por meio do software BMv2, com o objetivo de identificar qual solução FRR é mais adequada para ser integrada com o ECMP. Nesse cenário, o protótipo RESISTING foi concebido com o FRR-ECMP, enquanto o protótipo PURR foi incorporado ao ECMP com o objetivo de realizar uma comparação entre os dois mecanismos FRR. Inicialmente, o ECMP distribui os fluxos de pacotes entre os *links* de forma equilibrada. Em seguida, o experimento realiza simulações de falhas nos *links* do balanceamento ECMP em ambos os protótipos, considerando todas as variações possíveis de até três falhas. Posteriormente, em cada simulação, foram coletados dados/valores sobre os fluxos de pacotes encaminhados, bem como os fluxos perdidos (degradação) resultantes das falhas. Dessa forma, foi possível calcular a diferença entre os fluxos encaminhados e os fluxos perdidos, a fim de obter o nível de resiliência de rede para as soluções FRR avaliadas.

Os resultados revelaram que o mecanismo proposto foi capaz de se recuperar de até três falhas simultâneas, garantindo o encaminhamento de 100% dos fluxos gerados durante as simulações. Por outro lado, o PURR apresentou degradação nos fluxos a partir dos cenários de duas falhas. A arquitetura proposta demonstrou um nível de resiliência superior ao mecanismo estado da arte, indicando que essa solução pode ser mais adequada para integração ao ECMP em ambientes de data centers. Tal proposta contribui para a recuperação eficiente de falhas, mantendo a otimização no uso dos recursos de banda nos *enlaces* do ECMP.

1.2 Objetivo da Tese

O principal objetivo deste trabalho é projetar, implementar e avaliar um mecanismo *Fast-Reroute* integrado com o balanceamento ECMP na arquitetura de switches programáveis em P4. Para alcançar essa finalidade, foram estabelecidos os seguintes objetivos:

- **Pesquisa e Desenvolvimento da Arquitetura:** desenvolver um mecanismo *Fast-Reroute* integrado com ECMP na linguagem P4. Para isso, criamos um primeiro protótipo chamado FRR-ECMP no switch BMv2. Esse protótipo foi concebido para realizar a funcionalidade de recuperação dos *links* do mecanismo ECMP, utilizando um algoritmo, componentes e rotinas da linguagem P4 que não são suportadas em um switch físico. Devido a essas características, a adaptação da primeira versão do protótipo BMv2 para o Barefoot Tofino não foi possível. No entanto, o algoritmo do FRR-ECMP serviu como base para a criação de um novo protótipo, agora adaptado tanto ao switch BMv2 quanto ao switch Tofino. Esse novo protótipo foi nomeado RESISTING e constitui a implementação atualmente utilizada neste trabalho;

- **Avaliação do Desempenho da Arquitetura:** realizar avaliações experimentais abrangendo o desempenho do processo de recuperação FRR. Para atingir esse objetivo, conduzimos a avaliação experimental da seguinte maneira: concebemos um cenário com base em um switch e servidor físico, com o propósito de realizar testes práticos para avaliar o desempenho do protótipo. Isso nos permitiu analisar o impacto do método de recirculação temporária nos fluxos de pacotes envolvidos no processo de recuperação durante eventos de falha;

- **Avaliação da Resiliência da Arquitetura:** realizar avaliações experimentais abrangendo a capacidade de resiliência da arquitetura em comparação ao mecanismo estado da arte. Para atingir esse objetivo, desenvolvemos um protótipo na linguagem P4, integrando tanto o PURR quanto o mecanismo ECMP, com o intuito de compará-lo com a arquitetura. O ambiente experimental foi conduzido no simulador Mininet, equipado com o software BMv2. Nesse cenário, geramos fluxos de pacotes, aplicamos falhas no ambiente experimental e realizamos cálculos para conferir o nível de resiliência dos mecanismos FRR.

1.3 Metodologia Aplicada

A metodologia empregada para alcançar os objetivos deste estudo consistiu em uma abordagem que integra fundamentos teóricos, trabalhos relacionados, desenvolvimento da arquitetura e simulações, bem como avaliações comparativas. A seguir, apresentam-se os principais passos adotados nesta metodologia:

1. **Fundamentação Teórica:** Realizou-se uma pesquisa na literatura para obter uma base sólida de conhecimento teórico sobre o tema em estudo, especialmente em relação aos conceitos do *Fast-Reroute*, balanceamento ECMP e arquiteturas de switches programáveis na linguagem P4. A fundamentação teórica foi essencial para embasar

o desenvolvimento da arquitetura proposta, visando uma solução que contribua para o tema de pesquisa.

2. **Trabalhos Relacionados:** Realizamos uma revisão bibliográfica sobre *Fast-Reroute*, que possibilitou a identificação dos principais estudos relacionados à nossa pesquisa. Começamos pela análise de métodos de recuperação mais simples, avançando para soluções mais sofisticadas e eficientes. Essa etapa foi essencial para identificar as lacunas que o presente trabalho se propõe a abordar e apresentar uma proposta de resolução.

3. **Desenvolvimento da Arquitetura e Testes:** Inicialmente, desenvolvemos a primeira versão do mecanismo FRR-ECMP na plataforma BMv2. Por meio de testes e validação, aprimoramos o protótipo, focando na necessidade de reduzir a quantidade de vezes que os pacotes eram submetidos à recirculação e, principalmente, na criação de um modelo de algoritmo padronizado para o BMv2 e Tofino.

4. **Avaliação de desempenho**. Com o objetivo de desenvolver um método capaz de avaliar o desempenho da arquitetura proposta, criou-se o cenário de teste com um switch Tofino e um servidor físico, onde o Tofino emula dois switches *leaves* lógicos. A partir desse ambiente, conduziram-se experimentos que submeteram os fluxos de pacotes a eventos de falhas, com um aumento progressivo na taxa de transmissão. Após o processo de recuperação em cada simulação, os pacotes submetidos à recirculação foram coletados através do Wireshark, a fim de permitir a estimativa da quantidade de pacotes que foram recirculados dos fluxos devido a falhas e possibilitar a análise do impacto do método temporário de recirculação de pacotes na arquitetura.

5. **Avaliação de resiliência**. O método adotado para avaliar a resiliência consiste em comparar a degradação entre o mecanismo proposto e o mecanismo estado da arte. Para alcançar esse objetivo, no cenário virtual Mininet com o switch BMv2, conduziram-se experimentos que, inicialmente, submeteram os fluxos de pacotes a uma distribuição equilibrada entre os *enlaces* que representam os *uplinks* do ECMP. Em seguida, a aplicação de uma ou até três falhas simultâneas aciona o mecanismo FRR, que, de acordo com a característica do método de recuperação (PURR/RESISTING), redireciona os fluxos de pacotes afetados para um caminho backup ou redistribui entre os caminhos operacionais remanescentes. Após cada processo de recuperação, os fluxos de pacotes degradados e os entregues são coletados e aplicados em equações que avaliam o nível de resiliência, permitindo assim constatar o impacto das falhas e comparar o grau de resiliência dos mecanismos testados.

1.4 Estrutura do Trabalho

O restante do conteúdo está estruturado da seguinte forma: No Capítulo 2, abordamos os princípios teóricos fundamentais que norteiam esta pesquisa, incluindo conceitos de redes programáveis e resiliência de rede. Os trabalhos relacionados são discutidos no Capítulo 3. No Capítulo 4, apresenta-se a arquitetura FRR-ECMP, com explicação do método de *Parser*, roteamento, balanceamento ECMP e a própria arquitetura de recuperação FRR-ECMP. O Capítulo 5 avalia o desempenho e a capacidade de resiliência da arquitetura. Por fim, discutem-se as conclusões e indicam-se os trabalhos futuros. Como contribuição adicional, o Apêndice A apresenta o artigo correlacionado a esta pesquisa. Os Apêndices B, C, D e E expõem os resultados dos cálculos relativos ao Valor Demandado e ao Valor Demandado Médio, ambos desempenhando um papel importante no contexto do experimento que avalia a resiliência da arquitetura. No Apêndice F, disponibilizamos o código-fonte P4 da arquitetura proposta, na versão BMv2. Por último, o Apêndice G concede acesso ao repositório online, que armazena os códigos-fonte P4 suplementares, bem como os scripts empregados durante o desenvolvimento deste estudo.

2 Fundamentação Teórica

Este capítulo apresenta o embasamento teórico relacionado a este trabalho, abordando as redes programáveis e a resiliência de rede da seguinte maneira: (i) discutimos os conceitos de redes programáveis, destacando a trajetória evolutiva que parte da arquitetura de rede tradicional, passa pela arquitetura SDN e termina na linguagem P4. Em seguida, realizamos uma comparação entre a arquitetura de rede tradicional e a SDN. (ii) exploramos o conceito de resiliência de rede, com foco nos aspectos fundamentais dos métodos de recuperação, tanto no plano de dados quanto no plano de controle.

2.1 Redes Programáveis: Uma Inovação na Arquitetura de Rede

Ao longo de décadas, a arquitetura de rede tradicional (CISCO, 2012), baseada em dispositivos de rede convencionais que utilizam tecnologias "fechadas"e proprietárias, desempenhou um papel crucial no transporte confiável de dados em diversos meios. Essa rede contribuiu significativamente para o desenvolvimento e a expansão de serviços em instituições e organizações. No entanto, no início dos anos 2000, notou-se uma incapacidade de adaptação e inovação dessa arquitetura (CSIKOR *et al.*, 2013), ou seja, uma inflexibilidade que limitava o progresso das redes de computadores. Nesse contexto, a incorporação de novos protocolos e funcionalidades de rede nos dispositivos torna-se uma tarefa complexa e demorada, dependendo exclusivamente dos fabricantes (NUNES *et al.*, 2014). Como consequência, a implantação ágil de novos serviços na infraestrutura de rede se tornou um desafio.

A arquitetura SDN emergiu como uma resposta às limitações previamente mencionadas, introduzindo o conceito de redes programáveis. Isso, por sua vez, possibilita o desenvolvimento ágil de novos protocolos e aplicações, como mencionado em (NUNES *et al.*, 2014), o que impulsiona a inovação e o progresso dos serviços de rede. A característica distintiva dessa arquitetura reside na separação entre o plano de controle e o plano de dados, transformando a infraestrutura de rede em um modelo centralizado com o plano de controle no núcleo. Nesse contexto, a função do plano de controle é delegada a um dispositivo ou conjunto de dispositivos conhecidos como controlador SDN. Esses controladores gerenciam as políticas e funções de rede (KREUTZ *et al.*, 2014). A segregação entre os planos mantém o plano de dados no dispositivo de rede, onde continua responsável pelo encaminhamento de pacotes na infraestrutura, porém, agora, operando de acordo com as diretrizes definidas pelo controlador SDN. Cabe destacar que essa mudança na arquitetura do dispositivo também tem contribuído para a exploração de melhorias no plano de

dados e impulsionado inovações significativas por meio das tecnologias de plano de dados programável.

Na arquitetura SDN, a camada de aplicação, também conhecida como plano de gerenciamento, é uma estrutura de software aberta que capacita arquitetos e administradores de rede a desenvolverem algoritmos personalizados para o plano de controle. Isso possibilita a implementação de aplicações mais eficientes, flexíveis e centralizadas, como uma alternativa aos protocolos de roteamento usados no plano de controle na arquitetura de rede tradicional (HAUSER *et al.*, 2023). Tal camada fica posicionada acima do controlador SDN na hierarquia e estabelece uma comunicação direta com o controlador. O controlador, por sua vez, interage com os dispositivos de rede através de uma interface aberta – *Application Programming Interface* (API) – que estabelece um canal de comunicação eficiente e bem estruturado entre o controlador SDN e o plano de dados. Em síntese, a arquitetura SDN apresenta uma abstração que habilita a programação do plano de controle, proporcionando benefícios em relação aos dispositivos de tecnologia proprietária, nos quais tal flexibilidade seria inviável. Além disso, a comunicação com o plano de dados é estabelecida por meio de uma interface aberta, como o protocolo *OpenFlow* (MCKEOWN *et al.*, 2008), permitindo uma integração eficiente entre os componentes de diversos fabricantes.

Apesar dos avanços tecnológicos promovidos pelo SDN, a inflexibilidade intrínseca no plano de dados representava um desafio significativo para a comunidade acadêmica e a indústria. Uma solução para lidar com tal limitação foi proposta pelos autores do trabalho *Reconfigurable Match Tables* (RMT) (BOSSHART *et al.*, 2013), que apresentaram uma tecnologia de *chip* aplicada na arquitetura de switch físico capaz de habilitar a programação do plano de dados. A tecnologia oferece uma abstração que permite a reconfiguração do hardware do dispositivo de rede sem a necessidade de substituir o dispositivo ou intervenção direta por parte do fabricante. Dessa forma, os operadores de rede têm a capacidade de criar novos protocolos, personalizar o comportamento do processamento de pacotes, dimensionar os recursos de memória das tabelas, entre outros recursos. Atualmente, surgiram novas alternativas no segmento de plano de dados programável (KIANPISHEH; TALEB, 2023), que vão além do switch programável em hardware. Essas alternativas são classificadas em termos de hardware e software, abrangendo tecnologias como FPGAs, Smart NICs e soluções baseadas em software desenvolvidas para CPUs.

A programabilidade no plano de dados proporcionou o surgimento de linguagem de programação específicas para tal finalidade (KIANPISHEH; TALEB, 2023), sendo que a *Programming Protocol-Independent Packet Processors* (P4) (BOSSHART *et al.*, 2014) se destacou dentre as opções. O P4 é uma linguagem de alto nível, semelhante

à linguagem C, de código aberto, utilizada para programar o comportamento do processamento de pacotes em diferentes plataformas, independentemente das diferenças de hardware, arquitetura ou recursos específicos de cada dispositivo. Essa característica permite que os desenvolvedores escrevam programas que são independentes do *target*, ou seja, podem ser executados em diversos dispositivos de rede sem a necessidade de realizar grandes alterações no código P4. A comunicação entre o plano de controle e o plano de dados programável em P4 é estabelecida por meio de APIs. O *Openflow* é um exemplo de API que pode ser empregada em conjunto com o P4, sendo implementada como um programa P4 no plano de dados.

A evolução das tecnologias na arquitetura de redes de computadores na última década promoveu uma transformação fundamental nesse campo. Desde a introdução do SDN até as inovações trazidas pelo plano de dados programável e a linguagem P4, esses avanços permitiram uma capacidade de personalização e flexibilidade dos dispositivos de rede de maneira sem precedentes em comparação com a arquitetura de rede tradicional. A seguir, descreveremos as principais diferenças entre essas duas arquiteturas, a tradicional e a SDN, bem como faremos a comparação entre elas.

2.1.1 Arquitetura Tradicional vs SDN

Na arquitetura de rede tradicional, o modelo consiste em um controle descentralizado e distribuído, no qual cada dispositivo de rede, como switches e roteadores, requer um serviço de gerenciamento individual. Esse modelo é complexo e lento (KREUTZ *et al.*, 2014), dificultando o processo de provisionamento e gerenciamento de cada dispositivo, especialmente em infraestruturas de redes em larga escala, como os data centers de *cloud computing*, onde o operador da rede precisa lidar com um alto número de equipamentos de rede. Além disso, os dispositivos convencionais que têm servido como a 'espinha dorsal' da arquitetura de rede tradicional são equipamentos considerados verticais, uma vez que são tecnologias proprietárias que integram o plano de controle e o plano de dados dentro do mesmo equipamento. Isso acrescenta um aumento na dificuldade de inovação e adaptação (KREUTZ *et al.*, 2014) à arquitetura de rede tradicional.

A Figura 2.1 ilustra de forma sucinta o conceito vertical do dispositivo convencional. No plano de controle, são implementadas as aplicações de rede, como os protocolos de roteamento e serviços de gerenciamento. Por sua vez, o plano de dados assume a responsabilidade pelas diversas funções de rede, tais como roteamento, filtros, balanceamento de tráfego e *Network Address Translation* (NAT), entre outras funcionalidades essenciais. É importante ressaltar que a tecnologia empregada nos dispositivos não permite que o operador da rede realize modificações mais profundas nos componentes do plano de controle e do plano de dados. Isso implica na inviabilidade de personalização da rede, bem

como em uma longa espera pelo fabricante para adicionar novas aplicações, serviços e protocolos à rede (MICHEL *et al.*, 2021).

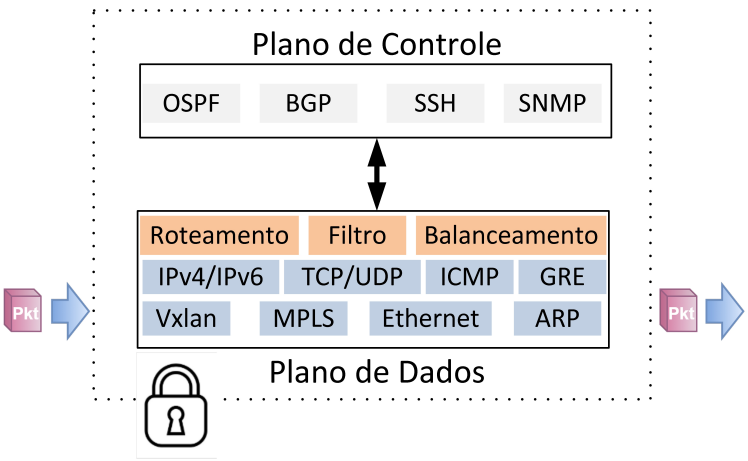

Figura 2.1 – Dispositivo convencional: arquitetura vertical.

O paradigma SDN, ao introduzir uma arquitetura de rede horizontal, juntamente com a tecnologia de plano de dados programável e a linguagem P4, surge como uma solução para lidar com os desafios mencionados anteriormente. O primeiro aspecto fundamental dessa transformação é a transição do controle distribuído para um modelo centralizado, permitindo que um controlador SDN gerencie todos os dispositivos de maneira unificada. Isso simplifica tarefas como monitoramento, configuração e implementação de políticas, reduzindo a complexidade operacional. Além do mais, contribui para o crescimento em larga escala do ambiente. A separação entre o plano de controle e o plano de dados introduziu uma perspectiva de programação. No plano de controle, o controlador e as aplicações de rede são disponibilizados como plataformas abertas, o que permite modificações tanto pelo operador da rede quanto por desenvolvedores externos. Por sua vez, no plano de dados, a tecnologia de dispositivos programáveis na linguagem P4 permite a reconfiguração do plano de dados e a personalização das funções de rede, incluindo a adição de novos protocolos. A flexibilidade intrínseca dessa nova arquitetura de rede possibilita ao operador personalizar e desenvolver soluções específicas, a fim de atender de forma precisa às necessidades da rede.

Na Figura 2.2, é apresentado um ecossistema que combina SDN, plano de dados programável e linguagem P4, mostrando a relação entre esses elementos. Com a separação entre o plano de controle e o de dados, observa-se que as aplicações essenciais do plano de controle, como roteamento e gerenciamento, agora são implementadas na camada de aplicação da arquitetura SDN, utilizando software de código aberto, que ajuda a impulsionar a evolução (KREUTZ *et al.*, 2014). Essa camada se comunica diretamente com o controlador SDN, que, de maneira centralizada, controla todos os dispositivos da rede por meio de uma API. Isso resulta em maior capacidade de provisionamento, geren-

ciamento e agilidade na rede. O plano de dados programável, por meio da linguagem P4, proporciona ao operador da rede vantagens como a personalização do comportamento de processamento de pacotes e a capacidade de adicionar novas funcionalidades e protocolos.

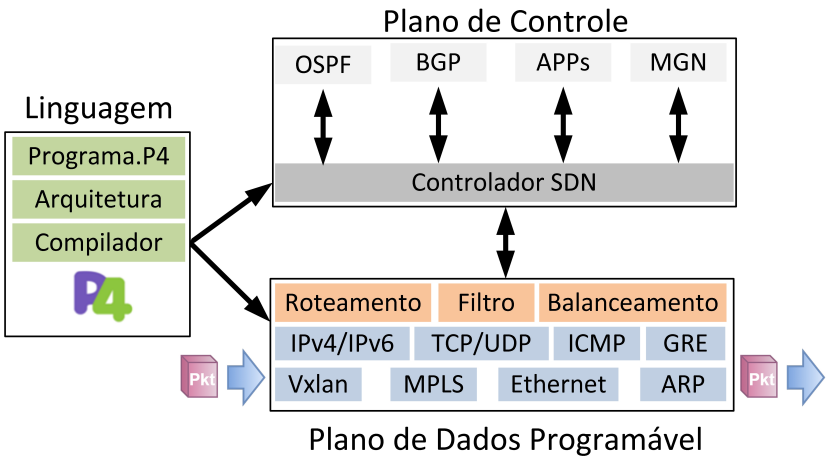

Figura 2.2 – Ecossistema: SDN, Plano de Dados Programável e P4.

2.2 Resiliência de Rede

Atualmente, vivemos em uma sociedade digital que demanda um alto nível de qualidade nos serviços e aplicações que utilizamos. Com a crescente dependência da tecnologia em nossas vidas cotidianas, seja para comunicação em rede sociais, trabalho, entretenimento ou transações comerciais, a expectativa de desempenho, segurança e confiabilidade desses serviços é cada vez maior. Nesse cenário, a infraestrutura de rede de computadores enfrenta o desafio de mitigar eventos de falhas, a fim de reduzir o tempo de interrupção dos serviços para garantir a qualidade desejada pelos usuários finais. A resiliência de rede é um atributo essencial que visa assegurar a capacidade de uma infraestrutura de comunicação resistir a diversos cenários de falhas, enquanto mantém um patamar aceitável de qualidade dos serviços e aplicações (STERBENZ *et al.*, 2011). No contexto do trabalho proposto, a resiliência de rede está diretamente relacionada aos mecanismos e estratégias de recuperação de falhas, que podem ser implementados tanto no plano de controle quanto no plano de dados dos roteadores ou switches (STERBENZ *et al.*, 2011).

Plano de Controle. As estratégias de recuperação implementadas no plano de controle são baseadas em protocolos de roteamento ou aplicações de rede, tanto na arquitetura de rede tradicional quanto na arquitetura SDN. Embora esses métodos de recuperação tenham sido aprimorados ao longo do tempo, buscando reduzir o tempo necessário para a recuperação de falhas (YE *et al.*, 2018), persiste uma latência intrínseca resultante do custo operacional do processo de recuperação ser realizado por meio de apli-

cações em software, através de um processador genérico no plano de controle, o que tende a ser mais lento em relação às implementações em hardware especializado. Além disso, após o recálculo das rotas no plano de controle, é necessário realizar mais uma etapa para concluir a recuperação. A aplicação de roteamento no plano de controle precisa instalar os novos caminhos no plano de dados para que os pacotes afetados pela falha possam finalmente ser encaminhados por um novo caminho, assegurando assim um caminho livre de falhas.

Plano de Dados. As estratégias de recuperação implementadas no plano de dados são operações realizadas em *chips* especializados, como *Application-Specific Integrated Circuit* (ASIC), *Field Programmable Gate Array* (FPGA) e outros. O processo de recuperação em hardware pode ocorrer na escala de milissegundos ou até mesmo em microssegundos (CHIESA *et al.*, 2021), enquanto que o processo de recuperação baseado em software no plano de controle, dependendo do cenário de rede e do método de recuperação utilizado, pode levar dezenas de milissegundos ou até mesmo segundos para ser concluído. Adicionalmente, o método de recuperação no plano de dados geralmente não precisa da atuação do plano de controle, uma vez que o processo de atualização dos novos caminhos são feitos diretamente no plano de dados.

A Figura 2.3 apresenta uma abstração das etapas de recuperação de falhas nas estratégias implementadas no plano de controle e no plano de dados. A primeira etapa do processo de recuperação de falha começa com o mecanismo de detecção de falha, que pode ser acionado por interrupções elétricas ou ópticas na interface do switch ou roteador, bem como pela interrupção do recebimento de pacotes de *keepalive* dos protocolos de roteamento ou aplicações de rede, como *Bi-directional Detection Forwarding* (BFD), *Open Shortest Path First* (OSPF) e Border Gateway Protocol (BGP). O tempo necessário para o mecanismo detectar uma falha varia de acordo com o tipo de falha e o método de detecção utilizado. Por exemplo, o tempo para detectar uma interrupção no circuito de uma rede de longa distância (WAN) através do protocolo BFD tende a ser mais longo em comparação com uma interrupção na rede local. Após essa etapa, o método de detecção aciona o mecanismo de recuperação no plano de controle ou no plano de dados, de acordo com a abordagem de recuperação definida.

Na estratégia de recuperação no plano de controle, primeiro ocorre uma etapa de convergência dos protocolos de roteamento ou aplicações de rede, o qual pode levar dezenas de milissegundos ou até segundos, dependendo da complexidade da rede e do tamanho da tabela de roteamento. Após o recálculo das rotas no plano de controle, a última etapa consiste em instalar os novos caminhos no plano de dados, disponibilizando assim um caminho backup para os pacotes afetados pela falha seguirem de maneira segura. Na estratégia de recuperação no plano de dados, ocorre apenas uma única etapa. Con-

siderando que a rota backup foi previamente instalada no plano de dados, o mecanismo de recuperação, ao ser acionado pelo método de detecção, redireciona imediatamente os pacotes afetados pela falha por meio de uma implementação em hardware para um caminho backup. Esse tipo de mecanismo implementado no plano de dados é chamado de ***Fast-Reroute*** (FRR), ou **re-roteamento rápido**, e destaca-se pela sua capacidade de assegurar uma rápida restauração da conectividade em nível de hardware, minimizando a degradação dos fluxos de pacotes causados pelas interrupções na rede.

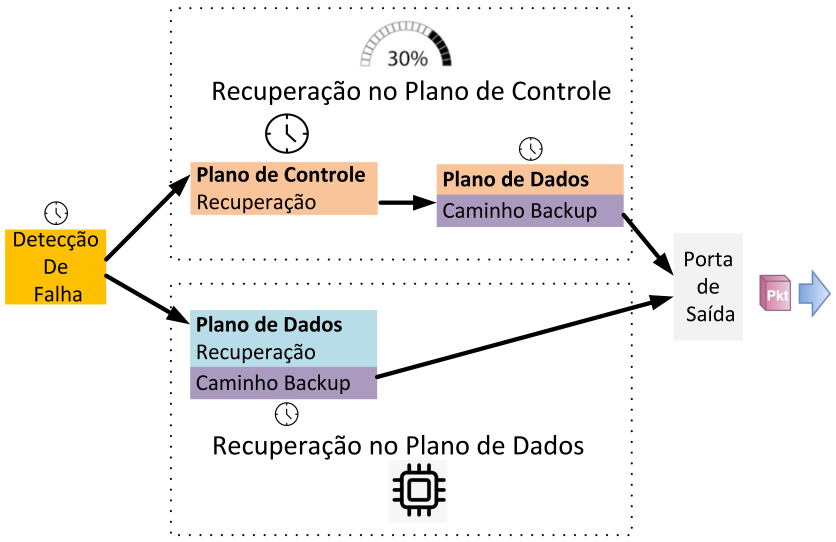

Figura 2.3 – Etapas de recuperação: Plano de Controle Vs Plano de dados.

2.2.1 Recuperação de falhas nos cenários com ECMP

O ECMP é uma técnica de roteamento que promove a distribuição dos fluxos de pacotes entre *links* com o mesmo custo (HOPPS, 2000). Inicialmente, no plano de controle ou no controlador SDN, o software de roteamento aprende todas as rotas com o mesmo custo para alcançar um destino e, em seguida, instala os múltiplos caminhos alternativos no mecanismo ECMP, implementado no plano de dados, seja ele programável ou convencional. Após o provisionamento das rotas, o mecanismo ECMP está pronto para distribuir aleatoriamente os fluxos de pacotes entre os *links* utilizando uma função *hash*.

O re-roteamento rápido (FRR) representa uma solução eficiente para a recuperação de falhas em diversos cenários de redes de computadores, sem a necessidade de recorrer a elementos externos (LUZ *et al.*, 2022), tais como o plano de controle ou o controlador SDN, reduzindo assim o tempo de recuperação. Essa estratégia permite que a rede se recupere de falhas locais, como interrupções na porta do equipamento ou cabeamento, bem como em cenários de falhas nos dispositivos adjacentes. No cenário dos data centers, onde há um intenso volume de tráfego constante, o balanceamento de tráfego é uma funcionalidade amplamente utilizada, especialmente em data centers de larga escala,

como os de computação em nuvem (PATEL *et al.*, 2013). Nesse contexto, uma estratégia de recuperação utilizada consiste em combinar o balanceamento ECMP, responsável pelo balanceamento de carga em diversas infraestruturas, com a solução de FRR, com o objetivo de mitigar falhas e, ao mesmo tempo, manter a eficiência no uso dos recursos de banda (CHIESA *et al.*, 2021). Vale ressaltar que essa combinação de funcionalidades é viável devido ao mecanismo ECMP operar com múltiplos caminhos alternativos, previamente instalados no plano de dados, permitindo o redirecionamento ou redistribuição dos pacotes afetados pela falha para um ou múltiplos caminhos alternativos. Na arquitetura da linguagem P4, os mecanismos de *Fast-Reroute* e ECMP são componentes separados (MERLING *et al.*, 2020; SIVARAMAN *et al.*, 2015), o que exige desafios tanto no desenvolvimento quanto na integração das soluções no plano de dados programável.

Durante a ocorrência de um evento de falha em um *link* utilizado pelo balanceamento ECMP, a solução integrada de recuperação assume a responsabilidade de remover ou desativar prontamente o *link* afetado pela falha, com o objetivo de minimizar o tempo de degradação dos fluxos de pacotes e, simultaneamente, manter a função *hash* operante de forma precisa, a fim de distribuir os fluxos de maneira assertiva para os caminhos operacionais, assegurando, dessa forma, a eficiência no balanceamento de tráfego.

3 Trabalhos Relacionados

Neste Capítulo, vamos explorar duas implementações básicas de *Fast-Reroute* na linguagem P4 e abordar três estudos relevantes da literatura para o nosso trabalho:

1. O IPFRR-LFA (SHAND; BRYANT, 2010) explora os conceitos fundamentais do *Fast-Reroute* e aborda o ECMP como um método com a capacidade de se recuperar de falhas. Para auxiliar a compreensão do comportamento do FRR de maneira prática, adaptamos o IPFRR (LFA) [1] para a linguagem P4 e conduzimos uma avaliação de seu comportamento em cenários de falha, utilizando o emulador Mininet;

2. O BIER FRR (MERLING *et al.*, 2020) contribui para a concepção dos mecanismos de recuperação IP-FRR e BIER-FRR. Ambos foram implementados na linguagem P4 e estão disponíveis no repositório dos autores;

3. O PURR (CHIESA *et al.*, 2019) apresenta uma primitiva de *Fast-Reroute* para o plano de dados programável, considerada o estado da arte. O mecanismo PURR destaca-se por sua simplicidade e eficiência no processo de recuperação. Utilizamos o PURR como mecanismo de referência e comparação em nosso trabalho, adaptando-o para a linguagem P4 [2] com base no código-fonte P4 disponível no trabalho (MERLING *et al.*, 2020).

Por fim, na Tabela 3.1, apresentamos uma análise comparativa teórica entre os três referidos trabalhos da literatura e o RESISTING. Esta análise considera métricas essenciais, como estratégia de recuperação, capacidade de lidar com uma ou múltiplas falhas, topologia de rede aplicada, plataforma utilizada e suporte ao balanceamento de tráfego.

3.1 Implementações simples de Fast-Reroute em P4

Na arquitetura da linguagem P4, não é implementada nativamente uma primitiva de *Fast-Reroute* em software e hardware, que permitiria ao operador da rede solicitar a funcionalidade por meio de funções no código P4. Portanto, a responsabilidade e o compromisso com o desenvolvimento do mecanismo de recuperação FRR recaem sobre o operador de rede. Uma abordagem simplificada para implementar a recuperação FRR

[1] P4-IPFRR LFA - <https://github.com/danielbl1000/P4-LFA>
[2] P4-PURR - <https://github.com/danielbl1000/P4-simple_switch_purr>

é por meio da funcionalidade de recirculação de pacotes. Basicamente, quando o mecanismo de detecção de falhas identifica um evento de indisponibilidade em uma porta, aciona uma estrutura que pode ser implementada tanto por meio de tabelas quanto por condicional (*IF/Else*). Tal estrutura desencadeia a recirculação contínua de todos os fluxos de pacotes afetados pela falha do *egress pipeline* para o *ingress pipeline*, permitindo que os fluxos de pacotes iniciem um novo ciclo de processamento no plano de dados. Esse processo tem como objetivo redirecionar os fluxos para uma nova porta de saída. Embora essa técnica seja simples de implementar e atenda aos requisitos básicos de recuperação, sua característica de exigir que os fluxos de pacotes recirculem continuamente no equipamento resulta em um aumento no *overhead* e na latência (CHIESA *et al.*, 2019).

Uma segunda solução de recuperação consiste em conceber um mecanismo baseado em uma tabela, que é responsável por analisar todas as possíveis alternativas de caminhos backup para o redirecionamento dos fluxos afetados pela falha. Essa solução se apoia nas regras instaladas no plano de dados por meio do plano de controle. Assim, para cada possibilidade de falha, uma regra é inserida na tabela, o que, por sua vez, consiste em armazenar cada regra na memória *Static Random Access Memory* (SRAM) ou *Ternary Content Addressable Memory* (TCAM) do switch. Em cenários com um número limitado de possibilidades de falhas, essa proposta pode ser adequada, pois segue um método direto de redirecionamento dos fluxos afetados pela falha para um caminho backup, sem a necessidade de recircular os pacotes. No entanto, sua viabilidade é um desafio em cenários com múltiplas possibilidades de falhas. Nesses casos, a instalação de um grande número de regras de encaminhamento se torna necessária, o que pode tornar a gestão das regras complexa, assim como consumir recursos consideráveis de memória do dispositivo de rede.

3.2 IPFRR: *Loop-Free Alternates* (LFA) e ECMP

As ocorrências de falhas na Internet, decorrentes de interrupções físicas nos *enlaces*, dispositivos de rede e outros elementos, motivaram o *Internet Engineering Task Force* (IETF) a buscar uma solução capaz de mitigar o impacto dessas falhas na infraestrutura de rede IP (CSIKOR *et al.*, 2013). Com essa missão, os engenheiros elaboraram um conjunto de técnicas de recuperação do protocolo IP, denominada "*IP Fast Reroute Framework*" (SHAND; BRYANT, 2010). Tal trabalho apresenta um estudo abrangente sobre os conceitos de recuperação relacionados com o nosso tema de pesquisa e descreve dois mecanismos de recuperação relevantes, o LFA (*Loop-Free Alternates*) e o ECMP, que também são abordados em maiores detalhas no documento técnico (ATLAS; ZININ, 2008).

O IPFRR é um método de recuperação semelhante ao mecanismo MPLS

FRR (YE *et al.*, 2018), mas aplicado à infraestrutura de roteamento IP. O conceito envolve a recuperação imediata de falhas em um *link* ou dispositivo por meio do redirecionamento dos pacotes para um caminho backup previamente instalado no plano de dados. O LFA foi introduzido ao mecanismo IPFRR com o propósito de assegurar que o caminho backup instalado no plano de dados seja livre de *loop* na rede. As rotas sem *loop* de roteamento são determinadas pelo software de roteamento no plano de controle, com base na topologia de rede, custos e métricas. Cabe ressaltar que o LFA protege contra uma falha de *link* e falha em dispositivos, como switches ou roteadores. Em algumas condições, esse mecanismo também pode proteger contra múltiplas falhas de rede (BRAUN; MENTH, 2016). Embora o LFA contribua com tais pontos positivos de recuperação de falhas, é importante destacar que o LFA não oferece proteção contra todos os tipos de topologias de rede (YE *et al.*, 2018) e, no que diz respeito ao problema que nosso trabalho propõe resolver, o LFA sozinho não considera redistribuir o tráfego afetado pela falha entre múltiplos caminhos operacionais utilizados em cenários com balanceamento.

Tradicionalmente, o ECMP foi concebido com o propósito de atender à distribuição do tráfego em diversas infraestruturas de rede. Sua característica intrínseca de permitir a instalação de caminhos alternativos no plano de dados favorece a recuperação de falhas, uma vez que as rotas são previamente configuradas antes da ocorrência de eventos de falha. Isso significa que essas rotas podem servir a um duplo propósito: balanceamento e recuperação. Dessa forma, os caminhos alternativos, quando operacionais, podem ser utilizados como rotas backup durante os eventos de recuperação (CSIKOR *et al.*, 2013).

A escolha do mecanismo de recuperação mais apropriado a ser integrado ao ECMP pode variar. Por exemplo, o LFA pode ser integrado ao ECMP, com o objetivo de fornecer uma abordagem simples de recuperação e sem *loops* na rede. Por outro lado, como mencionado anteriormente, ele não leva em consideração a distribuição dos fluxos de pacotes afetados pela falha entre os caminhos alternativos disponíveis. Em resumo, o ECMP pode ser usado de maneira eficaz tanto para o balanceamento de tráfego quanto para a recuperação de falhas em redes de computadores. A combinação do ECMP com mecanismos de recuperação adequados pode melhorar o nível da resiliência da rede, além de contribuir para a eficiência na utilização dos recursos dos *enlaces*.

3.3 *Bit Indexed Explicit Replication Fast Reroute* (BIER FRR)

Os autores do trabalho (MERLING *et al.*, 2020) exploram o conceito do BIER (*Bit Indexed Explicit Replication*), uma nova arquitetura de roteamento *multicast* proposta pelo IETF (WIJNANDS *et al.*, 2017). Esta tecnologia simplifica o método

de roteamento *multicast* nos roteadores, ao mesmo tempo em que confere o benefício da redução no consumo de recursos, tais como capacidade de processamento e alocação de memória, quando comparada com a abordagem convencional de roteamento *multicast* baseada no *Protocol Independent Multicast* (PIM) [3] — estabelece o roteamento *multicast* por meio de árvores e requer a manutenção dos estados dos fluxos de pacotes no roteadores.

No que diz respeito ao nosso trabalho, o artigo destaca sua relevância ao implementar o BIER em linguagem P4 e ao integrar dois mecanismos de recuperação: um destinado à proteção do tráfego *unicast*, denominado IP-FRR, e outro voltado para a proteção do tráfego *multicast*, conhecido como BIER-FRR. Os mecanismos de recuperação propostos seguem uma estratégia que consiste em uma abordagem primária que redireciona os fluxos de pacotes do caminho principal afetado pela falha para um único caminho backup. Em cenários de contingência 1:1, essa solução pode ser eficaz. Entretanto, em cenários com múltiplos caminhos paralelos e balanceamento de tráfego implementado, os autores não consideram a possibilidade de distribuir os fluxos de pacotes afetados pela falha entre os vários caminhos operacionais.

3.4 *Primitive for Reconfigurable Fast Reroute* (PURR)

Os trabalhos (SEDAR *et al.*, 2018; CHIESA *et al.*, 2019) trouxeram a proposta de uma primitiva de *Fast-Reroute* aplicada ao plano de dados programável em P4, denominada *Primitive for Reconfigurable Fast Reroute* (PURR). Esta primitiva destaca-se pela eficiência no processo de recuperação de falhas, permitindo a recuperação de uma ou várias falhas simultâneas, sem causar degradação no *throughput* e na latência. Em outras palavras, a solução consiste na implementação de uma tabela no *ingress pipeline*, que realiza uma busca do tipo ternária pela próxima porta de saída operacional, a fim de redirecionar os fluxos de pacotes afetados pela falha para a porta disponível. O mecanismo proporciona uma indicação precisa da porta operacional subsequente à porta com falha, independentemente da quantidade ou ordem das falhas. Para esclarecer o funcionamento do mecanismo de recuperação PURR, na Figura 3.1, abordamos a ilustração da abstração do FRR em duas etapas:

(1) Considerando um cenário de rede no qual um switch *Top-of-the-Rack* (ToR) está conectado a quatro switches *Upstream* por meio de quatro *uplinks*, correspondendo às portas P1, P2, P3 e P4. O tráfego do switch 1 segue o roteamento pela porta P3. Neste contexto, ocorre um evento de falha no *link* da porta P3, exigindo atuação imediata do mecanismo PURR;

[3] Protocol Independent Multicast - <https://www.rfc-editor.org/rfc/rfc7761.html>

(2) No switch 1, observa-se o processamento dos fluxos de pacotes que percorrem o *pipeline* do equipamento. Primeiro, a tabela ***routing***, por meio do *LPM lookup*, determina a porta P3 como destino para alcançar a rede NET_C. Em seguida, a tabela `port_status` detecta a falha na porta P3 e, consequentemente, aciona a tabela de recuperação PURR. O mecanismo, então, realiza uma busca ternária pelo valor-chave P3, retornando a porta P4 como a primeira opção da sequência P4, P1 e P2. Dessa forma, o fluxo de pacotes da porta P3 é agora redirecionado para a porta P4 do switch 1. Em caso de uma segunda falha, como, por exemplo, na porta P4, o PURR redirecionaria o fluxo da porta P4 para a próxima porta operacional da sequência, que, neste exemplo, seria a porta P1.

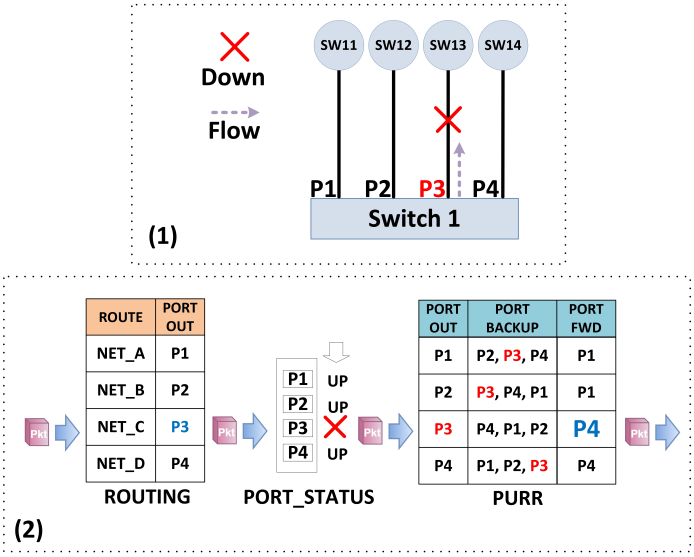

Figura 3.1 – Mecanismo de recuperação PURR.

O PURR é considerado o mecanismo *Fast-Reroute* estado da arte. No entanto, a estratégia adotada consiste em desviar todos os pacotes do caminho principal afetado pela falha para um único caminho *backup*. Em cenários com um intenso volume de tráfego e onde o balanceamento de tráfego é necessário para prevenir congestionamentos e oferecer maior eficiência no uso dos *enlaces*, a estratégia de recuperação, que apenas desvia os fluxos de pacotes afetados pela falha para um único caminho *backup*, pode resultar em um desequilíbrio na distribuição dos fluxos de pacotes entre os *links*, além de favorecer o surgimento de condições de saturação no caminho *backup* durante os eventos de falha.

3.5 Comparação entre os trabalhos

A Tabela 3.1 compara de forma sucinta as principais características dos trabalhos relacionados com o nosso mecanismo: 1) estratégia de recuperação; 2) quantidade de

falhas recuperadas; 3) topologia de rede testada; 4) validação na plataforma física e/ou software; 5) suporte a distribuição de pacotes.

Trabalho	**Recuperação**	**Falhas**	**Topologia**	**Plataforma**	**Balanceamento**
IPFRR	Rota Backup	Uma Falha	Anel	BMv2	Não
BIER-FRR	Consulta Ternária	Uma Falha	Anel	BMv2	Não
PURR	Consulta Ternária	Múltiplas	Leaf-Spine	BMv2, Tofino, FPGA	Não
RESISTING	Recirculação temporária	Múltiplas	Leaf-Spine	BMv2, Tofino	ECMP

Tabela 3.1 – Comparação entre o estado da arte e o mecanismo proposto.

A estratégia de recuperação dos trabalhos listados na Tabela 3.1 destaca-se por ser uma implementação mais simples em termos de esforço de implementação no dispositivo P4. O processo de recuperação adotado consiste em redirecionar os pacotes afetados pela falha do caminho principal para o caminho backup. Nessa perspectiva, essas estratégias podem atender adequadamente cenários simples de rede, como um roteador com um caminho principal e backup. Porém, ao avaliarmos cenários mais complexos, como a infraestrutura de data center, onde o switch pode ter mais de dois *uplinks* e as soluções de balanceamento são implementadas, a ausência da funcionalidade de redistribuição do tráfego afetado pela falha entre outros *links* disponíveis no balanceamento pode ser um fator que comprometa o desempenho do ambiente.

Nosso trabalho propõe contribuir disponibilizando um mecanismo FRR-ECMP capaz de superar a limitação encontrada nas estratégias de recuperação dos trabalhos mencionados. Tal proposta remove o *link* afetado pela falha do balanceamento de tráfego, assegurando a distribuição dos fluxos de pacotes afetados entre os caminhos operacionais remanescentes no ECMP. Em contrapartida, para alcançar esse objetivo, o RESISTING demandou um custo mais elevado em termos de esforço de implementação em comparação a outros métodos de FRR. Isso se deve à complexidade de criar uma estrutura FRR capaz de atualizar o mecanismo ECMP de maneira autônoma, sem requerer intervenção do plano de controle, visando assegurar o re-roteamento rápido dos pacotes impactados pela falha. Cabe ressaltar que o processo de recuperação da arquitetura proposta utiliza um método fundamentado na recirculação temporária de pacotes para alcançar a funcionalidade de recuperação do ECMP. Diante disso, é importante notar que esse método prolonga o tempo de permanência dos pacotes no plano de dados, constituindo uma desvantagem em relação às estratégias FRR da literatura.

4 Arquitetura RESISTING

Neste capítulo, são detalhados os componentes essenciais da nossa arquitetura de recuperação ECMP, com foco no protótipo BMv2 e abrangendo os seguintes tópicos: compreensão geral da arquitetura, o método de detecção de falhas adotado, a técnica de roteamento de pacotes, o mecanismo de balanceamento ECMP e, por fim, exploramos o conceito fundamental do FRR-ECMP proposto, demonstrando a recirculação temporária de pacotes e a atualização do balanceamento ECMP após a ocorrência de falha. Os pontos relevantes referentes ao protótipo Tofino também são abordados, embora de maneira simplificada, seguindo a explicação do BMv2. A decisão de destacar a explicação do protótipo BMv2 ocorre devido à sua capacidade de executar as funcionalidades fundamentais necessárias para alcançar a solução proposta de recuperação do ECMP. No caso do Tofino, identificamos uma limitação específica relacionada à redução da quantidade de *links* no balanceamento ECMP, conforme será detalhado mais adiante neste capítulo. Essa limitação não desqualifica o protótipo; no entanto, coloca a implementação em desvantagem nesse aspecto em comparação ao protótipo BMv2.

4.1 Compreensão geral da arquitetura

Apesar das diferenças e particularidades que as plataformas BMv2 e Tofino apresentam no contexto da linguagem P4, o RESISTING foi concebido com a finalidade de preservar a similaridade nos aspectos estruturais, algorítmicos e nos componentes inerentes à linguagem P4, em ambos os protótipos. A arquitetura oferece as seguintes funcionalidades de rede: roteamento baseado no endereço IPv4 e TAG; balanceamento de tráfego ECMP; mecanismo de detecção de falhas nas interfaces; recirculação de pacotes; uso de cabeçalho adicional no pacote para o transporte de dados entre os *pipelines*; e *fast-reroute*. Essas funcionalidades serão explicadas nas próximas seções deste capítulo. Adicionalmente, no Apêndice G, encontra-se a URL para o nosso repositório online, onde disponibilizamos os códigos-fonte P4 dos protótipos deste trabalho.

A Figura 4.1 apresenta de maneira abrangente o fluxograma de encaminhamento de pacotes, assim como a recuperação de falhas em ambos os protótipos. As setas azuis na figura indicam o fluxo de encaminhamento de pacotes, enquanto as setas vermelhas representam o fluxo de recuperação. Para facilitar o entendimento do fluxograma, considere um cenário com um switch *top of rack*, onde o encaminhamento de pacotes ocorre sem falhas nas portas de saída, o fluxo das setas azuis inicia na etapa de `Parser`, na qual os pacotes passam por uma análise dos cabeçalhos das camadas 2, 3 e 4, incluindo

a extração dos campos relevantes para o fluxo de encaminhamento. Posteriormente, na etapa de `Forwarding`, as definições de roteamento e balanceamento ECMP são executadas para determinar a porta de saída que o fluxo de pacotes deve seguir. Em seguida, na etapa `Port_status`, o mecanismo de detecção de falha permite simular o estado operacional da porta do switch por meio de regras instaladas pelo plano de controle no plano de dados. Se a porta estiver operacional, o fluxo de pacotes é direcionado para a porta de saída apropriada do switch, seja ela uma porta do balanceamento ECMP ou uma porta de acesso. Dessa forma, encerra-se o fluxo de encaminhamento de pacotes em condições sem falhas.

No caso da etapa `Port_status` simular uma falha na porta de saída do tipo acesso, ou seja, aquela que conecta diretamente um *host* ao switch, o fluxo de pacotes é continuamente descartado enquanto a porta permanecer indisponível na etapa `Port_status`, conforme indicado pela seta de cor roxa na figura. Em contrapartida, em caso de falha na porta de saída associada ao *link* do balanceamento, o fluxo de pacotes não é descartado; ele é encaminhado para a rotina de recuperação, conforme demonstrado pelas setas vermelhas na figura.

Na etapa FRR, a estrutura de recuperação disponibiliza um conjunto de componentes que simulam o balanceamento ECMP, preservando a mesma sequência de *links* associados às portas de saída. Por meio desse cenário de reprodução do ECMP, a arquitetura é capaz de lidar com a falha, removendo a porta de saída indisponível desse componente de simulação ECMP e reordenando simultaneamente as portas remanescentes operacionais do balanceamento. Após a conclusão desta etapa, a nova sequência de portas operacionais é registrada no cabeçalho FRR do pacote, com a finalidade de permitir a atualização do balanceamento ECMP na última etapa. Então, o pacote contendo o cabeçalho FRR, por sua vez, é encaminhado para uma nova etapa de `Parser` por meio da funcionalidade de recirculação de pacotes. Na etapa de `Parser`, ocorre a análise do cabeçalho FRR e a extração dos campos essenciais para conclusão do processo de recuperação da falha no *link* ECMP. Em seguida, a etapa de `Updating` realiza a atualização dos *links* do ECMP com a nova ordem de portas operacionais, sobrescrevendo a porta indisponível do *link* ECMP. Simultaneamente, ocorre a redução na quantidade de *links*, visando remover o caminho inativo do serviço de balanceamento de forma definitiva.

Por fim, após a conclusão da atualização do ECMP na etapa de `Forwarding`, o ECMP é capaz de redirecionar o fluxo de pacotes de maneira permanente e com segurança para um porta de saída operacional, que representa um *uplink* conectado a um switch *upstream*. Cabe ressaltar que o processo de recuperação da porta indisponível associada ao *link* ECMP ocorre sem interferir no roteamento e balanceamento dos fluxos de pacotes das demais portas operacionais do switch.

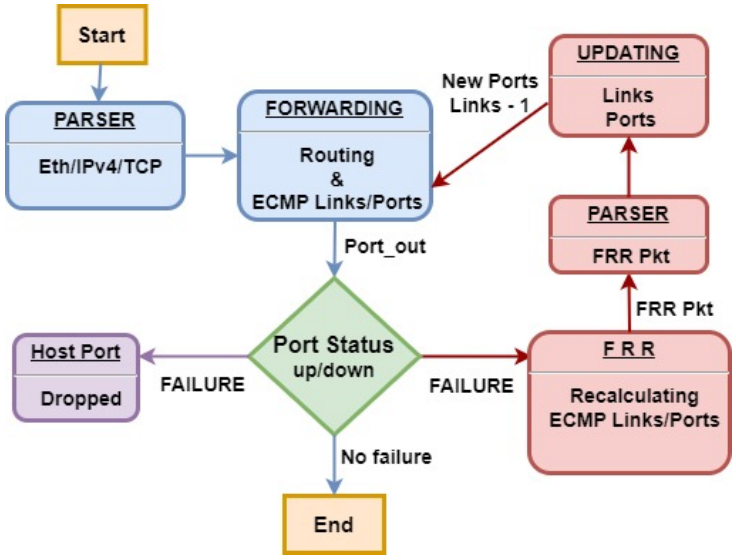

Figura 4.1 – Fluxo de encaminhamento de pacotes e recuperação de falhas.

4.2 Detecção de falha

Na infraestrutura de rede, a falha causa um efeito de incapacidade operacional no elemento de rede, impactando assim o funcionamento do mesmo. Portanto, quando nos deparamos com uma falha em um *link*, as consequências dessa falha se manifestam na interrupção do tráfego que flui através desse meio físico. Nesse contexto, a recuperação de falha é acionada pelo mecanismo de detecção de falhas, o qual deve ser assertivo e rápido na sinalização da falha ao mecanismo de recuperação, seja no plano de dados ou no plano de controle. Na Figura 4.2, apresentamos a abstração do mecanismo de detecção de falha, que, juntamente com o mecanismo de recuperação, desempenha um papel relevante no tempo de recuperação da falha e na mitigação do impacto nos fluxos de pacotes.

O método de detecção de falha na camada física visa identificar as interrupções no sinal elétrico ou óptico na interface do equipamento. Tal abordagem pode levar alguns milissegundos para detectar a falha (SHAND; BRYANT, 2010). Em métodos de detecção de falha que monitoram a disponibilidade de dispositivos na rede por meio de abordagens que enviam e recebem pacotes de *hello* ou BFD, o tempo de detecção de falha costuma ser ainda mais prolongado. Na arquitetura P4, diferentes métodos são empregados para detectar falhas, variando de acordo com a plataforma. Por exemplo, no switch BMv2, a comunidade de desenvolvimento P4 sugere a criação de uma tabela ou registro chamado *port_status* para armazenar o estado operacional (*up/down*) de cada porta (LUZ *et al.*, 2022). No caso do switch Tofino, é possível utilizar o mesmo método ou um mecanismo de *trigger* que envia pacotes para sinalizar a falha ao plano de controle do equipamento ou a um controlador SDN. Essa abordagem permite iniciar o processo de recuperação ou simplesmente notificar a indisponibilidade da porta no sistema de gerenciamento.

Para padronizar o método de detecção de falha nos protótipos BMv2 e Tofino utilizados neste trabalho, optamos por implementar o conceito de detecção de falha sugerido pela comunidade de desenvolvimento P4. Dessa forma, criamos uma tabela chamada `port_status`, que permite simular falhas nas portas do switch por meio do plano de controle, inserindo regras que indicam o estado operacional de cada porta. A referida tabela foi posicionada após a etapa `Forwarding`, a qual é constituída pelas funcionalidades de roteamento e ECMP, visto que, a princípio, não encontramos na arquitetura P4 um método de identificar o estado operacional da porta de saída antes da sua definição pelos métodos de roteamento. Adicionalmente, como parte de nossos planos para trabalhos futuros, pretendemos avaliar a viabilidade de implementar a detecção de falha no switch Tofino por meio do mecanismo *trigger*, assim como a implementação do protocolo BFD, tanto para BMv2 quanto Tofino. A expansão do escopo de detecção de falhas ampliará a capacidade de nosso mecanismo em detectar falhas, tornando-o mais adequado para cenários reais.

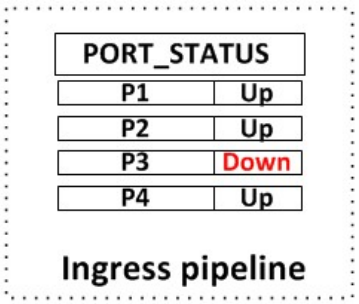

Figura 4.2 – Tabela de detecção de falha.

4.3 Método de *Parser* e Aplicação do *Header* FRR

O *parser* na linguagem P4 é uma etapa essencial que ocorre no início do processamento dos pacotes no *ingress pipeline* e no *egress pipeline*. Essa etapa é responsável por interpretar os cabeçalhos dos pacotes que são recebidos pelas portas físicas ou lógicas do dispositivo. Na arquitetura proposta, o *parser* implementado analisa sequencialmente os cabeçalhos de camada 2, 3 e 4 dos pacotes recebidos pelas portas de entrada do switch no *ingress pipeline*. Durante o processo de análise, o *parser* extrai os campos essenciais dos cabeçalhos utilizados pela arquitetura. À medida que o processamento do pacote avança pelo *ingress pipeline*, esses campos extraídos, juntamente com os metadados, são consultados e alterados pelas tabelas e ações nos estágios de processamento.

Os campos de metadados na arquitetura P4 são espaços de memória alocados para armazenar dados associados a cada pacote que atravessa o plano de dados. O escopo de atuação dos metadados está restrito ao *ingress* ou *egress*, ou seja, não é possível a troca

de metadados entre os *pipelines*, o que resulta na impossibilidade do envio e recebimento de dados entre o *ingress* e o *egress* por meio desses campos. Na nossa arquitetura, a estrutura de recuperação FRR reside no *egress pipeline*, enquanto o mecanismo ECMP está localizado no *ingress*. Para permitir a sinalização entre esses mecanismos, que residem em *pipelines* diferentes, empregamos uma técnica que utiliza um cabeçalho adicional no pacote, denominado FRR, com o objetivo de transportar os campos necessários para a recuperação de falhas entre os *pipelines* do switch durante o processo de recuperação. O protótipo BMv2 introduz o cabeçalho FRR no espaço entre o cabeçalho Ethernet e o IP no pacote. No caso do protótipo Tofino, a implementação utiliza o espaço ocupado pelo cabeçalho Ethernet no pacote e sobrescreve o Ethernet pelo FRR, evitando assim um *overhead* adicional no pacote.

Com o objetivo de exemplificar os métodos de *parses* dos protótipos, na Figura 4.3, apresentamos uma ilustração dos métodos aplicados e fazemos uma comparação entre eles. Considerando um cenário em que o switch recebe um pacote [1] de um *host* por meio da porta de entrada. Primeiramente, no *ingress pipeline*, o método de *parser* proposto interpreta sequencialmente (`Eth->IP->TCP`) os cabeçalhos do pacote e extrai os campos considerados relevantes para garantir o processamento do pacote no switch, tanto no protótipo BMv2 quanto no Tofino. Este é o método de *parser* utilizado em situações sem a ocorrência de falhas, destinado apenas ao encaminhamento dos pacotes.

Quando um evento de recuperação está em andamento, o pacote envolvido na recuperação que ingressa no *ingress pipeline* é proveniente do método de recirculação vindo do *egress pipeline*. Nesse cenário, cada plataforma possui um tratamento diferente em relação à recirculação de pacotes. O BMv2 recebe o pacote recirculado do *egress pipeline* pela mesma porta de entrada em que o pacote foi recebido inicialmente do *host*. Entretanto, o pacote agora transporta, além dos cabeçalhos tradicionais (Ethernet, IP, TCP/UDP), o *header* FRR contendo os campos essenciais para a recuperação da arquitetura. No caso do switch Tofino[2] empregado no nosso trabalho, o pacote recirculado proveniente do *egress pipeline* utiliza duas opções de portas internas e dedicadas para recirculação de pacotes. Para atender às abordagens de recirculação de pacotes das plataformas, os métodos de *parser* foram adaptados conforme o tipo de plataforma.

***Parser* BMv2.** O pacote recebido no *ingress*, proveniente da recirculação, analisa primeiro o cabeçalho *Ethernet*, o que extrai o conteúdo do campo *EtherType*. No caso desse campo armazenar o valor `0x255`, que representa o *header* FRR, o *parser* segue para análise e extração dos campos relevantes do FRR para a recuperação. O *parser* FRR se relaciona com os cabeçalhos de camada 3 e 4, o que possibilita adicionar novas

[1] Um pacote constituído de *payload* e cabeçalhos Ethernet, IP e TCP/UDP.
[2] O Barefoot Tofino BFN-T10-032 suporta 32 x 100Gbps (QSFP28) e 2 x 100Gbps recirculation ports.

funcionalidades e modificações nos cabeçalhos superiores em caso de necessidade.

***Parser* Tofino**. No estágio inicial do *parser* (denominado `state_start`), especificamos as duas portas dedicadas à recirculação, que requisitam diretamente o *parser* do *header* FRR. Essa abordagem é necessária para distinguir os pacotes provenientes da recirculação de pacotes daqueles que não estão relacionados aos eventos de falha e precisam seguir o *parser* padrão, que normalmente começa com o *Ethernet*. Desta forma, o pacote recebido no *ingress*, vindo da porta de recirculação, passa por uma análise do cabeçalho FRR e extrai os conteúdos dos campos essenciais para o processo de recuperação. A partir desse ponto, os protótipos BMv2 e Tofino seguem um similar *parser*.

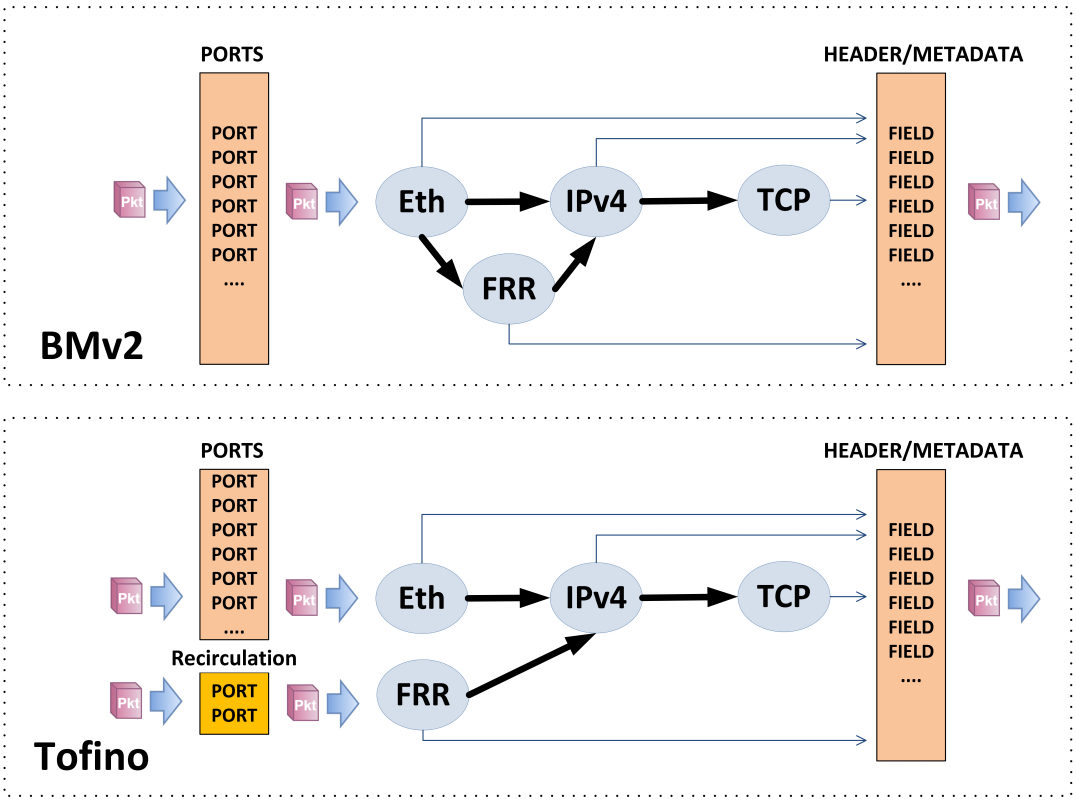

Figura 4.3 – *Parser* implementado nas plataformas BMv2 e Tofino.

4.4 Roteamento de pacotes

A função básica de um roteador, seja ele físico ou virtual, consiste na capacidade de encaminhar pacotes na rede. Para tanto, quando o dispositivo recebe um pacote em uma porta de entrada, com base em decisões de roteamento, ele direciona o pacote para uma porta de saída apropriada ou, no caso de pacotes *multicast*, para múltiplas portas de saída. As decisões de encaminhamento são tomadas por meio de tabelas no plano de dados que analisam os endereços de destino, como os de camada 2 (Ethernet), 2,5 (MPLS) ou 3 (IP) presentes nos pacotes. No contexto do nosso trabalho, apresentamos uma estratégia de recuperação para cenários de data center, onde o balanceamento

de tráfego é amplamente utilizado. Isso implica na necessidade de conceber uma técnica de roteamento adequada para essa infraestrutura de rede. Sendo assim, o método que empregamos para encaminhar pacotes em nossa arquitetura baseia-se no roteamento de camada 2 e 3, seguindo um conceito semelhante ao aplicado no trabalho *Virtual Layer 2* (VL2) (GREENBERG *et al.*, 2009).

A arquitetura de roteamento adotada incorpora duas categorias de endereçamento: (1) o endereço IP, que é alocado aos *hosts* na rede e proporciona a comunicação fim-a-fim entre os elementos; (2) o endereço ***D**e**S**tina**T**ion **ID**entification* (DST_ID), concedido aos switches *leaf*/ToR na topologia Clos *leaf-spine* (ALIZADEH; EDSALL, 2013) adotada. O endereço `DST_ID`, por vezes referenciado como TAG em nosso trabalho, corresponde a um valor numérico capaz de identificar o switch na rede e que se sobrepõe ao endereço MAC (*Media Access Control*) no campo de destino dos pacotes. Sua finalidade é viabilizar o roteamento dos pacotes entre os switches *leaves*, nos quais os *hosts* estão conectados, por meio dos *spines* na topologia Clos.

O RESISTING combina as tabelas de roteamento `ipv4_lpm` e ECMP no *ingress pipeline* para garantir a comunicação entre *hosts* no mesmo switch, bem como entre *hosts* remotos que estão conectados em switches diferentes. A tabela `ipv4_lpm` utiliza o endereço IP de destino do pacote como valor-chave para realizar a busca pela rota mais adequada para alcançar o *host* de destino. No caso em que o *host* de destino está diretamente conectado ao mesmo switch, o pacote segue o processo de encaminhamento para a porta de saída apropriada. No entanto, se o *host* de destino está conectado a um switch remoto na rede, ocorre um passo adicional. Primeiramente, o `DST_ID` apropriado para alcançar o switch remoto é escrito no campo MAC de destino no *header* Ethernet do pacote, sobrepondo o endereço MAC de destino original. Em seguida, o pacote segue para processamento na tabela ECMP.

A tabela ECMP utiliza o endereço `DST_ID` do pacote como valor-chave para buscar a rota que permita alcançar o switch remoto ao qual o *host* de destino está conectado. Essa tabela desempenha outras funções relacionadas ao mecanismo ECMP, as quais são detalhadas na Seção 4.5. Para simplificar a explicação e focar na compreensão do roteamento proposto, após a localização da rota para o switch de destino, o pacote segue o processo de encaminhamento por uma das opções de *links* do balanceamento ECMP, que estão conectados aos switches *spines*. Assim que o pacote é recebido pelo switch *spine*, ele segue um fluxo direto para a tabela ECMP, ignorando a `ipv4_lpm` que precede a tabela ECMP no *ingress pipeline*. Não é necessário realizar uma segunda busca pelo endereço IP de destino, uma vez que, nessa camada de switches da rede, os *spines* precisam apenas conhecer as rotas para alcançar os *leaves*.

A Figura 4.4 apresenta os passos do roteamento implementado considerando

uma topologia Clos *leaf-spine* simplificada, onde o *host* (`h1`) que está conectado no *leaf 1* envia um pacote com destino o *host* (`h2`) conectado no *leaf 10*.

(1) Inicialmente, o switch `leaf 1` recebe o pacote do `h1` através da porta de entrada 15 (P15). Em seguida, no *ingress pipeline*, realiza-se uma busca na tabela `ipv4_lpm` para localizar a rota correspondente ao IP de destino 20.0.0.11 (`h2`). Como resultado, o `DST_ID` de valor 10 é obtido, o qual está associado ao switch `leaf 10` e, consequentemente, o valor 10 é escrito no *header* Ethernet do pacote. Com o `DST_ID` L10 devidamente inserido, o pacote segue para a próxima etapa de processamento no `leaf 1`;

(2) No mecanismo ECMP, a tabela busca pela rota `DST_ID` L10, que está mapeada para quatro opções de *links* no balanceamento ECMP (P1-P4). Posteriormente, neste exemplo, a aplicação da função *hash* resulta no encaminhamento do pacote pela porta de saída 1 (P1) no `Leaf 1`, que corresponde ao *uplink* conectado ao `spine 11`;

(3) O switch `spine 11` recebe o pacote do `leaf 1` e, consequentemente, realiza a busca pela rota `DST_ID` L10 na tabela ECMP, que em conjunto com a função *hash*, retorna a porta de saída 2 (P2), conectada ao `leaf 10`. O switch `spine` realiza apenas o roteamento via TAG, simplificando assim o processo de encaminhamento dos pacotes e reduzindo o consumo de memória nos `spines`;

(4) Finalmente, o switch `leaf 10` recebe o pacote do `spine`11 e realiza a busca pela rota correspondente ao endereço IP 20.0.0.11 (`h2`) na tabela `ipv4_lpm.` A busca retorna a porta P16, resultando no encaminhamento do pacote para o destino final, juntamente com a operação que altera no pacote o endereço `DST_ID` para o apropriado endereço MAC do `host` (h2).

4.5 *Equal Cost Multi Path* (ECMP)

O ECMP é um método de roteamento que distribui o tráfego entre múltiplos caminhos de custo igual e menor distância para alcançar um destino na rede (CHIESA *et al.*, 2017). Cabe ao software de roteamento com suporte ao ECMP, como por exemplo o BGP e o OSPF, a responsabilidade de calcular as rotas (CAO *et al.*, 2000). Posteriormente, por meio de um componente de interface específico, as rotas são instaladas no mecanismo ECMP incorporado ao plano de dados. Após essa etapa, o balanceamento de tráfego entra em operação, utilizando uma função *hash* para distribuir os fluxos de pacotes de maneira equilibrada entre os diversos caminhos disponíveis. Com o objetivo de tornar a

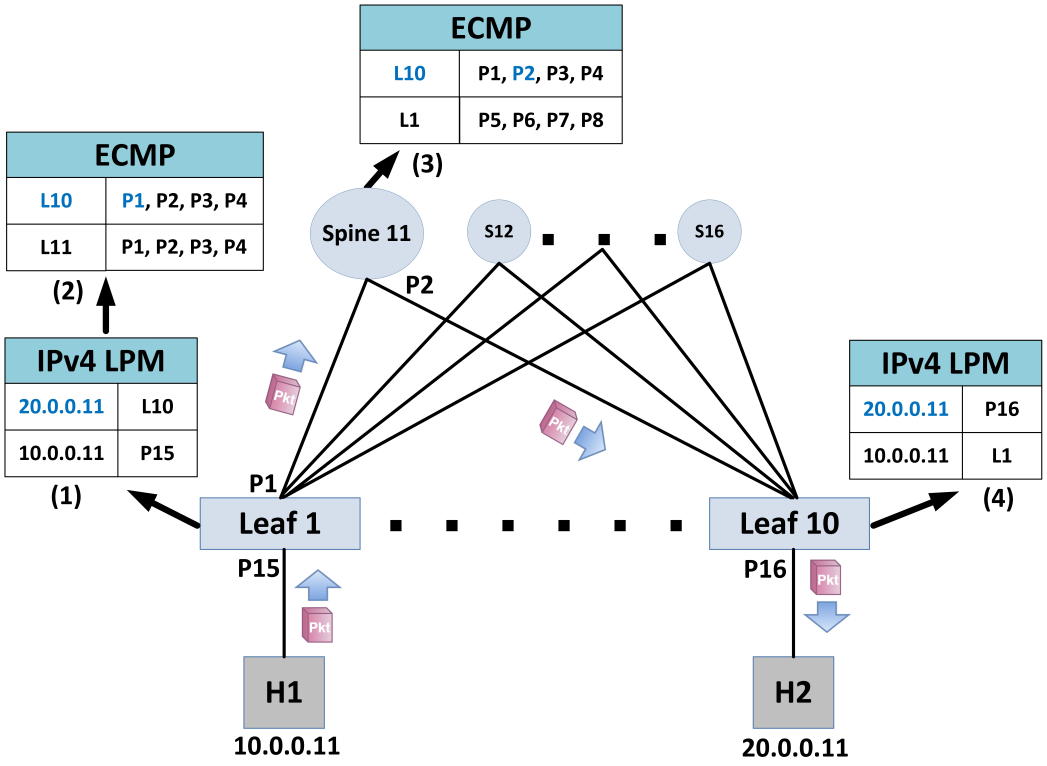

Figura 4.4 – Topologia Clos *leaf-spine*: roteamento IP e TAG

compreensão do mecanismo ECMP proposto mais acessível e organizar o texto, dividimos os elementos que constituem o ECMP nas partes de plano de controle e de dados:

- **Plano de controle**. A criação de um software de roteamento destinado a calcular rotas para o ECMP está além do escopo de nosso trabalho. No entanto, para efetuar essa funcionalidade, as rotas foram criadas manualmente no plano de controle das plataformas. Posteriormente, essas rotas foram instaladas por meio das interfaces Thrift API [3] no BMv2 e pela *Barefoot Runtime Interface* (BRI) no Tofino. Todas as rotas dos protótipos, bem como os demais tipos de regras utilizados no trabalho, estão disponíveis no repositório online, conforme detalhado no Apêndice G.

- **Plano de dados**. Na arquitetura P4, uma estratégia para desenvolver um mecanismo ECMP implica na criação de uma tabela que realiza buscas com base em um valor-chave do tipo inteiro. Esse valor representa a rota e a identificação que referencia um grupo, o qual, por sua vez, mapeia os elementos que representam os caminhos de destinos no equipamento, como *links* ou portas. No RESISTING, o campo-chave na tabela ECMP é a identificação do switch de destino (`DST_ID`), que desempenha o papel dublo, representando uma rota e mapeando o grupo de *links* utilizados no balanceamento. Cada *link*, por sua vez, está diretamente associado a uma porta de saída específica do switch.

[3] Thrift - <https://thrift.apache.org/>

A função *hash* aplicada no mecanismo ECMP é baseada no algoritmo *Cyclic Redundancy Check* (CRC), o qual é geralmente implementado em hardware e amplamente usado para detecção de erros na comunicação de rede. Para realizar a distribuição do tráfego entre os *links*, primeiramente, a função recebe como parâmetro de entrada uma tupla de cinco elementos do pacote: IP de origem, IP de destino, protocolo, porta de origem e porta de destino. Em seguida, o CRC é aplicado a essa tupla e, por meio de uma operação de módulo (`5-tuple mod N`), obtém-se o resultado do *link* de saída (CAO *et al.*, 2000). Nesse contexto, o valor definido no parâmetro `Max_Links` especifica o divisor na operação de módulo e é responsável por definir a quantidade de caminhos disponíveis no balanceamento ECMP.

Cabe destacar que, considerando a integração do ECMP com o mecanismo de recuperação, os valores que representam as portas de saída do switch e o valor do parâmetro `Max_Links` estão sujeitos a alterações durante o processo de FRR. Isso implica na necessidade de se adotar uma abordagem que permita a modificação por meio do mecanismo FRR no plano de dados. Portanto, a técnica de armazenamento por meio de metadados (*metadata*) na linguagem P4 não atenderia. Sendo assim, todos os valores dos parâmetros que o FRR requer modificar, como as portas e a quantidade máxima de *links*, foram armazenados em registros dedicados, o que permite a flexibilidade de serem controlados por meio do plano de dados durante o processo de recuperação.

A Figura 4.5 apresenta a abstração do mecanismo ECMP em operação em um cenário com quatro fluxos de pacotes distintos. Considerando que o plano de controle previamente instalou as regras no plano de dados, como a rota `DST_ID` 10, os *links* 0, 1, 2 e 3, as portas de saída P1-P4 e definiu o registro `Max_Links` com a quantidade máxima de 4 *links* disponíveis para o balanceamento. Inicialmente, os pacotes ingressam no *ingress pipeline* do switch, onde o mecanismo ECMP primeiro interpreta e distingue os fluxos de pacotes com base na tupla de cinco elementos dos pacotes recebidos. Em seguida, os fluxos são processados pela função *hash*, resultando na distribuição dos fluxos entre os quatro *links* disponíveis, numerados de 0 a 3, que estão associados à rota `DST_ID` 10. Neste exemplo, o valor 10 representa a rota para alcançar o switch de destino em uma topologia Clos *leaf-spine*. Por fim, cada *link* é mapeado para um registro específico que armazena a porta de saída, possibilitando o encaminhamento dos pacotes para a porta de saída apropriada do switch.

4.6 FRR-ECMP

Conforme mencionado neste trabalho, a arquitetura FRR-ECMP remove o *link* afetado pela falha do mecanismo ECMP, ao mesmo tempo que reduz o valor que repre-

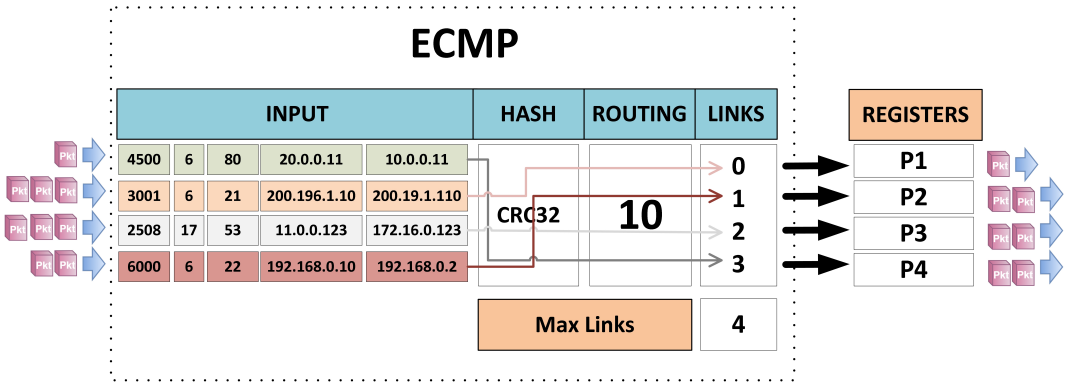

Figura 4.5 – Mecanismo ECMP

senta a quantidade de *links* operacionais, o qual é usado como parâmetro na operação modular da função *hash* do ECMP. Essa operação assegura uma nova distribuição do tráfego afetado entre os *enlaces* disponíveis após o término do *Fast-Reroute*. Com o objetivo de construir uma arquitetura de recuperação robusta, capaz de lidar com uma ou múltiplas falhas simultâneas nos *links* utilizados pelo mecanismo ECMP, nosso trabalho desenvolveu um algoritmo que utiliza uma série de elementos e componentes na linguagem P4, aplicados nas seguintes etapas do processo de recuperação:

- **Início da Recuperação:** a tabela `frr_recirculation` desempenha um papel duplo, permitindo habilitar e encerrar o processo de recuperação na arquitetura proposta. Após a detecção de uma falha no *ingress pipeline*, a tabela é prontamente acionada, dando início ao mecanismo *fast-reroute*. Inicialmente, é adicionado o cabeçalho FRR aos pacotes afetados pela falha, sendo que ao primeiro pacote subsequente à detecção da falha cabe a responsabilidade de transportar as informações essenciais para o funcionamento da recuperação. Os demais pacotes seguem um caminho de recuperação semelhante ao do primeiro pacote, porém com o único propósito de evitar o descarte enquanto a operação de recuperação está em andamento. Para cada pacote, um número de identificação é atribuído por meio de uma rotina de contador, com o intuito de distinguir o primeiro pacote dos demais pacotes do fluxo. Vale ressaltar que uma das informações transportadas pelo primeiro pacote é a posição do *link* indisponível no balanceamento ECMP. Essa informação permite tanto à Estrutura FRR quanto à rotina de atualização do ECMP identificar dentre os *links* possíveis, qual precisa ser tratado;

- **Processamento da Falha:** a Estrutura FRR consiste em um conjunto de tabelas, nomeadas como `frr_port_out`, registros e uma operação condicional *(IF/Else)*, residindo de maneira integrada no *egress pipeline*. O propósito fundamental é a replicação dos elementos que compõem o mecanismo ECMP, mantendo a mesma

sequência de *links*, com suas respectivas portas de saída do switch. Essa estrutura remove a porta de saída do *link* indisponível por meio da reordenação das portas operacionais dos *links* remanescentes, em um ambiente isolado e controlado. Nesta etapa, a implementação inicia o algoritmo de movimentação das portas dos *links* remanescentes, movendo a porta da posição que representa o último *link* do ECMP para a posição do *link* anterior. Esse processo se repete até alcançar a posição do *link* que representa o *link* que sofreu a falha no ECMP, onde a porta que estava alocada na posição do *link* anterior sobrepõe a porta de saída ao *link*, causando a exclusão da porta na Estrutura FRR. O processo de movimentação das portas provoca uma realocação das portas nos *links* que representam o balanceamento ECMP. Em seguida, a nova sequência de portas realocadas é escrita nos campos apropriados do cabeçalho FRR do primeiro pacote subsequente à falha, visando possibilitar a atualização dos *links* do ECMP na próxima etapa da recuperação;

- **Atualização do ECMP:** a etapa de atualização do mecanismo ECMP envolve a substituição das portas alocadas nos registros de encaminhamento `port_out` pela nova sequência de portas operacionais dos *links* remanescentes geradas pela Estrutura FRR, ao mesmo tempo em que ocorre a redução da quantidade de *links* operacionais, por meio do decremento do valor do registro `Max_Links` no mecanismo ECMP. Essa operação é executada por uma estrutura de tabelas conhecida como `update_port_out`, juntamente com a tabela `update_ecmp_hash`. Basicamente, a rotina de atualização identifica qual posição do *link* precisa ser tratada e, a partir dessa informação, copia a nova sequência de portas armazenadas nos campos apropriados do cabeçalho FRR para os registros de encaminhamento que precisam de atualização;

- **Fim da Recuperação:** a tabela `frr_recirculation` atua tanto no início quanto no encerramento do processo de recuperação na arquitetura. Com a devida atualização realizada no mecanismo ECMP e nos registros de portas, o processo de recuperação pode ser considerado como encerrado. No entanto, os pacotes afetados pela falha e recirculados precisam passar por um novo processo de encaminhamento após a atualização do balanceamento. Adicionalmente, o mecanismo FRR requer que alguns componentes sejam retornados ao estado inicial, como o reinício da rotina do contador. Portanto, mais algumas tarefas são realizadas pela tabela, como a remoção do cabeçalho FRR dos pacotes envolvidos no processo de recuperação, bem como outras tarefas explicadas com mais detalhes neste capítulo.

4.6.1 Método de recirculação temporária de pacotes

O P4 é uma linguagem de domínio específico voltada para a programação de dispositivos de rede. Diferentemente de linguagens de propósito geral, como C++ e Python, o P4 não oferece recursos de laço, como declarações *"for"* e *"while"* para controle de fluxo. Essa restrição ocorre porque os dispositivos de rede que utilizam a linguagem P4 precisam executar as operações de processamento de pacotes em altíssima velocidade. Entretanto, a linguagem disponibiliza a funcionalidade de recirculação de pacotes do *egress pipeline* para o *ingress pipeline*, promovendo uma operação que se assemelha com a funcionalidade de laço dentro do plano de dados. Nesse contexto, a execução dos blocos de código P4 em cada estágio são repetidos a cada recirculação do pacote. Adicionalmente, a inclusão de um cabeçalho adicional no pacote possibilita a comunicação/integração entre os blocos de código P4 que residem em *pipelines* isolados. Vale ressaltar que o aumento da permanência dos pacotes no plano de dados causa um acréscimo na latência. No entanto, em arquiteturas baseadas em hardware, especialmente quando se emprega um número reduzido de recirculações, o impacto na latência tende a ser imperceptível.

Consideramos o método de recirculação de pacotes adotado no nosso trabalho como temporário, pois os pacotes afetados pela falha são recirculados apenas duas vezes durante o processo de recuperação. A primeira recirculação atualiza o balanceamento ECMP por meio do primeiro pacote subsequente à falha. A última recirculação tem o objetivo de proporcionar aos pacotes um novo processamento no plano de dados, a fim de buscar um caminho operacional para saída dos pacotes. A seguir, ilustramos na Figura 4.6 as etapas de recirculação de pacotes do primeiro pacote subsequente à falha na arquitetura sugerida.

(1) Primeiramente, o switch recebe o pacote pela porta de entrada 11 (P11). Em seguida, o pacote é processado pelo pipeline de entrada (*ingress pipeline*), onde o roteamento define a porta de saída. O pacote então segue para o (*egress pipeline*), que o direciona para a porta de saída 1 (P1). Basicamente, essa é a estratégia de encaminhamento de pacotes em cenários sem a ocorrência de falhas;

(2) Logo que a falha na porta 1 (P1) é detectada no *ingress pipeline*, a estratégia de recuperação é acionada. Essa estratégia gera uma nova sequência de portas operacionais no *egress pipeline*. A nova sequência é então inserida no cabeçalho FRR, que é adicionado ao pacote e recirculado de volta ao *ingress*;

(3) No *ingress pipeline*, após o pacote ser recirculado, a estrutura de encaminhamento de pacotes é atualizada com a nova sequência de portas operacionais, por meio do cabeçalho FRR. Em seguida, o pacote segue para o *egress*, onde o cabeçalho FRR

é removido e o pacote é recirculado pela última vez. Essa recirculação inicia uma nova estratégia de encaminhamento de pacote no switch. Com isso, o processo de recuperação está concluído;

(4) Por fim, após a segunda e última etapa de recirculação de pacote no plano de dados, o pacote segue um novo processo de encaminhamento, no qual é direcionado para uma nova porta de saída operacional no switch.

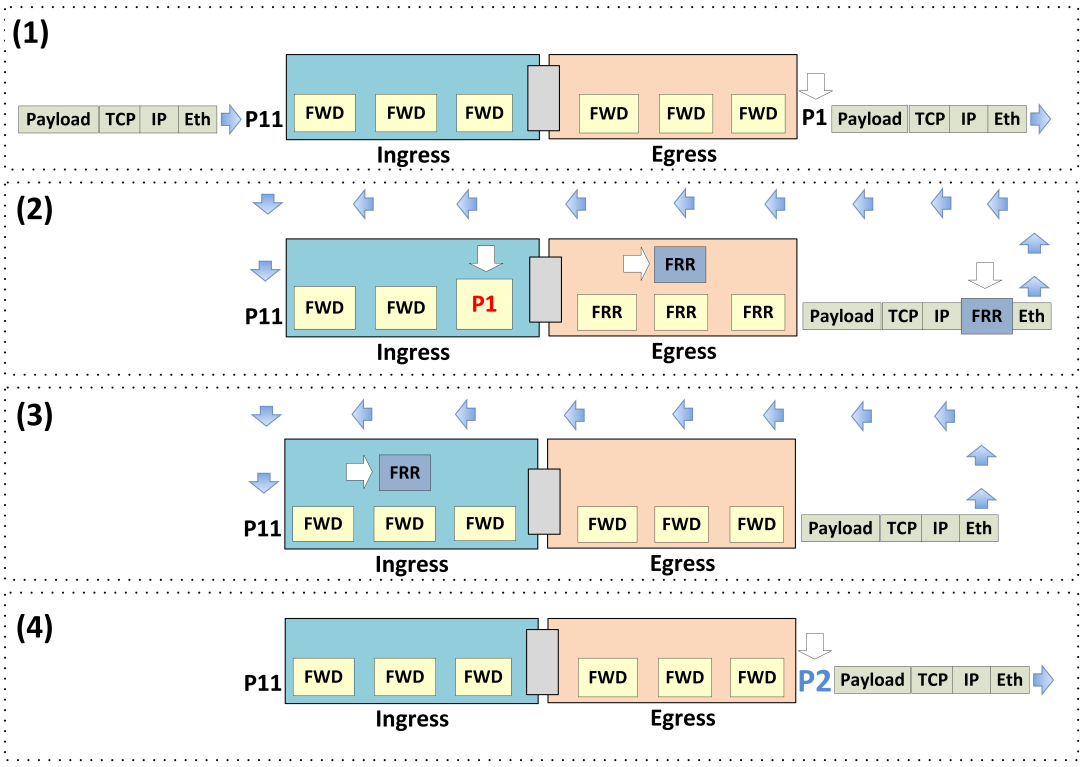

Figura 4.6 – Recirculação temporária de pacotes.

4.6.2 Processo de recuperação de falha

Na seção anterior, discutimos a integração e a troca de dados entre os *pipelines*, utilizando o método de recirculação de pacotes, empregando uma técnica baseada na inclusão de um cabeçalho de recuperação no pacote. Agora, prosseguimos abordando a arquitetura, direcionando nossa atenção para uma explicação dos componentes que compõem os elementos fundamentais do FRR-ECMP e a interação entre eles durante o processo de recuperação

O RESISTING segue quatro principais etapas para concluir o processo de recuperação: (1) **Início da Recuperação** – corresponde à etapa que reúne informações essenciais utilizadas nas etapas subsequentes, como, por exemplo, qual *link* sofreu a falha e parâmetros de controle do mecanismo FRR, envolvendo numeração de pacote, rotina

de contador e acionamento da recirculação; (2) **Processamento da Falha** – esta etapa proporciona a remoção da porta indisponível e geração de uma nova sequência de portas operacionais em um ambiente separado do ECMP. A partir disso, as portas operacionais são encaminhadas para o ECMP; (3) **Atualização do ECMP** – nesta etapa, a nova sequência de portas operacionais é recebida, permitindo, assim, a atualização dos *links* remanescentes do balanceamento ECMP, ao mesmo tempo em que a quantidade de *links* operacionais é subtraída; (4) **Fim da Recuperação** – nesta etapa final, ocorre a remoção do cabeçalho FRR dos pacotes envolvidos na recuperação e a submissão da recirculação pela última vez.

Nas Figuras 4.7, 4.8, 4.9 e 4.10, apresentamos a ilustração das abstrações das quatro etapas utilizadas no processo de recuperação de falha na arquitetura baseada no protótipo BMv2, cujo código fonte P4 está disponível no Apêndice F. Neste cenário de explicação, um fluxo de pacotes tipicamente percorre o plano de dados. Quando ocorre um evento de falha na porta 3 (P3), desencadeia-se o início da recuperação. A configuração do switch envolve seis *links* (0-5) e seis portas de saída (*P1-P6*).

4.6.2.1 Início da Recuperação

Considerando que o plano de controle previamente instalou todas as regras no plano de dados, o primeiro passo inicia quando o switch recebe um fluxo de pacotes através da porta de entrada. Assim, os pacotes no *ingress pipeline* passam pelo método de `parser`, onde os cabeçalhos das camadas 2, 3 e 4 são interpretados, e seus campos relevantes são extraídos para dar suporte ao pacote durante sua progressão nos estágios subsequentes. Para simplificar e focar na compreensão da recuperação, as etapas das tabelas `IPv4_lpm` e ECMP resultam no *link* 2, o que determina a porta de saída 3 (P3) neste exemplo. Em seguida, na tabela `port_status`, responsável por detectar falhas com base no estado operacional da porta, é realizada uma busca pelo valor-chave 3, correspondente à porta P3. O resultado da busca indica que o estado da porta está indisponível. Portanto, a tabela `frr_recirculation` é acionada para iniciar o processo de recuperação.

Para desempenhar a dupla função de iniciar e encerrar o processo de recuperação de maneira integrada em um único bloco de código P4, a tabela `frr_recirculation` utiliza uma série de parâmetros de busca. Entre eles, destacam-se a busca pelos valores-chave que indicam o número do *link* associado à porta inativa, neste caso, o número 2, e o estado da recuperação, sinalizado como 0, representando o início deste processo. Este parâmetro é essencial, pois indica o acionamento da *action* de início ou fim da recuperação. Nessa etapa da recuperação, a *action* de início associada à tabela é necessária. Ela é composta pelas operações que desencadeiam o início do *Fast-Reroute*, sendo destacadas as seguintes:

- Adição do cabeçalho FRR: este cabeçalho é inserido entre os cabeçalhos Ethernet e IP no pacote. Essa operação permite o transporte de dados entre componentes que residem em *pipelines* distintos. Um exemplo disso é a inclusão da informação sobre o *link* que requer recuperação, como, neste caso, o *link* 2 do balanceamento ECMP;

- Rotina de contador: o registro `dr_rg_first_failure_count_wr` é acionado para realizar uma contagem e atribuir um número de identificação específico no campo apropriado do cabeçalho FRR em cada pacote, com o objetivo de distinguir o primeiro pacote subsequente à falha dos demais na arquitetura;

- Sinalização de um novo cabeçalho: o valor 255 é definido no campo *EtherType* do cabeçalho Ethernet, com o propósito de identificar o cabeçalho FRR e instruir o correto fluxo do *parser* durante o processo de recuperação;

- Habilitar a recirculação: a operação de recirculação dos pacotes ocorre no final do *egress pipeline*, mas na linguagem P4 a funcionalidade é habilitada no *ingress*.

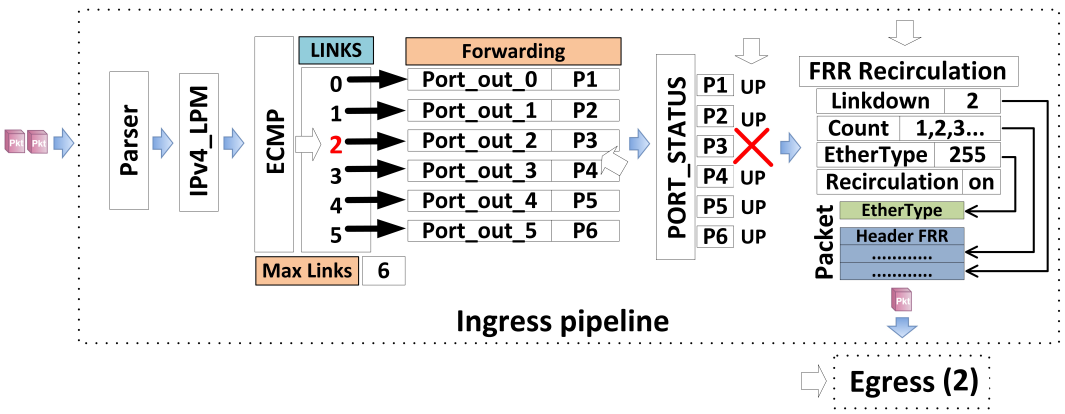

Figura 4.7 – (1) Início da Recuperação.

4.6.2.2 Processamento da Falha

Esta etapa é composta por uma rotina de contagem decrescente de portas que podem ser recuperadas. Essa rotina é formada pela tabela `count_port_up`, pelo registro `rg_count_port_up_wr`, bem como pela Estrutura de *Fast-Reroute*. Tal estrutura fornece uma réplica do cenário ECMP, simulando a remoção da porta do *link* interrompido no balanceamento. A seguir, detalhamos as operações utilizadas nesta etapa do processo de recuperação:

- Contador de portas operacionais: este elemento tem o propósito de controlar a quantidade de *links* operacionais que podem ser recuperados. Por exemplo, no cenário que abordamos como explicação, são utilizados seis *links* no balanceamento ECMP,

o que permitiria até cinco recuperações de falhas, desde que pelo menos um *link* esteja operacional para o encaminhamento do tráfego, ou seja, cinco *links* backup. Para isso, a tabela `count_port_up` e o registro `rg_count_port_up_wr` são usados para controlar a quantidade de *links* que podem ser recuperados. Isso é realizado armazenando o valor referente à quantidade máxima de portas backups possíveis de recuperação no `rg_count_port_up_wr`, que, neste exemplo, é definido como 5. Posteriormente, o valor é decrementado na etapa seguinte conforme ocorrem falhas e processos de recuperação. Em caso de falha em todos os *links* utilizados no ECMP, esse mecanismo impede a tentativa de recuperação do último *link* que estava operacional, evitando assim um *loop* ininterrupto dos pacotes envolvidos na falha, já que não haveria mais nenhum caminho backup disponível para o encaminhamento do tráfego;

- Estrutura FRR: a estrutura em questão inicia com uma operação condicional *IF*, que verifica se o pacote possui as credenciais necessárias para acessar a operação de processamento de falhas. Essa condição abrange seis tabelas, denominadas `frr_port_out_0` a `frr_port_out_5`, bem como seis registros, denominados `rg_frr_port_out_0_wr` a `rg_frr_port_out_5_wr`. Além disso, inclui uma operação que decrementa o registro `rg_count_port_up_wr`, o qual foi mencionado anteriormente. A condição verifica entre os pacotes envolvidos na recuperação qual atende aos seguintes critérios: 1) se é o pacote número 1, ou seja, o primeiro pacote subsequente à falha; 2) durante o processo de recuperação, o pacote número 1 passa pelo *egress pipeline* mais de uma vez. Por isso, a condição identifica se o pacote número 1 está passando pela primeira vez, por meio de um campo de controle no cabeçalho FRR. Assim, a implementação assegura o acesso ao pacote número 1 proveniente da etapa inicial da recuperação, que detectou a falha; 3) verifica por meio do valor armazenado no `rg_count_port_up_wr` se ainda há portas operacionais disponíveis para recuperação. Se o valor do registro `rg_count_port_up_wr` for igual a zero, isso indica que todos os *links* alternativos do balanceamento ECMP foram utilizados no tratamento de outros incidentes de falhas. Portanto, os pacotes que não atenderem aos três critérios mencionados seguem para o final do *pipeline*, seguindo o fluxo de recirculação e prevenção de descarte, enquanto ocorre o processo de recuperação e atualização do ECMP.

  As tabelas `frr_port_out`, em conjunto com os registros `rg_frr_port_out`, proporcionam um cenário de replicação dos elementos do mecanismo ECMP, mantendo a mesma sequência de *links* com suas respectivas portas de saída. Isso é realizado por meio do armazenamento dos *links* $L0 - L5$ e das portas de saída $P1 - P6$. Nesse ambiente de replicação, ocorre a remoção da porta de saída do *link* afetado pela falha

por meio da reordenação das portas operacionais nos *links* remanescentes. Posteriormente, a nova sequência de portas reordenadas é escrita nos campos apropriados do cabeçalho FRR do pacote número 1, que segue para o *ingress pipeline*, com o objetivo de atualizar o mecanismo ECMP com as novas portas de saída, conforme explicado na etapa (3) de recuperação;

- Reordenação das portas: o RESISTING segue uma lógica decrescente de reordenação com base no deslocamento das portas da posição do último *link* para a posição do *link* anterior. Essa movimentação de portas se repete até que a posição do *link* que sofreu a falha seja alcançada. Além disso, o último *link* recebe o valor 255, que representa uma porta *default* no protótipo Tofino. Essa atribuição visa corrigir uma limitação do Tofino (discutida na próxima seção) e indicar que ocorreu uma falha, ou dependendo do cenário, várias falhas nos *links* ECMP. Considerando o mecanismo ECMP com seis *links*, o qual inicia na posição zero e termina na posição cinco, a movimentação das portas começa na posição cinco e termina na posição dois, que representa o *link* que se tornou indisponível. Os *links* nas posições um e zero não necessitam de atualização. Por fim, o processo de reordenação das portas sobrepõe a porta na posição do *link* afetado pela falha, resultando na sua remoção no cenário de replicação dos elementos ECMP e, posteriormente, no mecanismo ECMP.

  Neste exemplo, o processo de reordenação atua na tratativa do *link* número dois, que está associado à porta 3 (P3), resultando na seguinte reordenação e nova sequência de portas operacionais: $(L2 \rightarrow P4)$, $(L3 \rightarrow P5)$, $(L4 \rightarrow P6)$ e $(L5 \rightarrow P255)$.

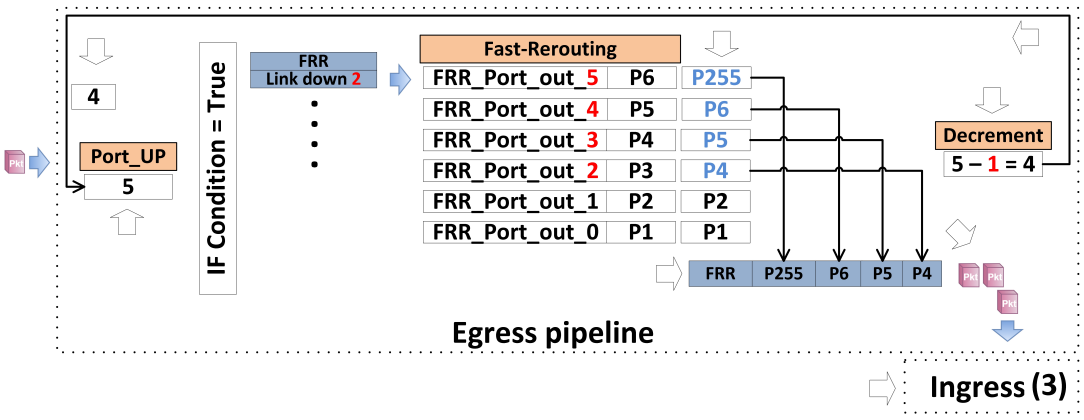

Figura 4.8 – (2) Processamento da falha.

4.6.2.3 Atualização do ECMP

O processo de atualização consiste na ação de copiar cada porta dos *links* envolvidos no processo de reordenação realizado na etapa (2) para os registros de encaminhamento `rg_port_out` apropriados, ao mesmo tempo em que decrementa o valor

que controla a quantidade de *links* operacionais no balanceamento ECMP por meio do registro `rg_max_link_wr`. Para realizar tal operação, foi concebida uma rotina baseada em tabelas e *actions*, dedicada à atualização dos elementos no mecanismo ECMP. Primeiramente, a tabela `update_ecmp_hash` executa uma operação que reduz a quantidade de *links* operacionais de seis para cinco no `Max_Link` (registro `rg_max_link_wr`), permitindo, assim, que a função *hash* empregada no ECMP considere os *links* de 0 a 4 no cálculo modular, excluindo o último *link* do balanceamento. Este último *link*, por sua vez, agora corresponde a um *link* indisponível que foi tratado pela implementação. Na sequência, as tabelas, que variam de `update_port_out_0` a `update_port_out_5`, realizam uma busca pelo valor-chave que representa o *link* que sofreu a falha, no campo responsável pelo transporte dessa informação no cabeçalho FRR. As tabelas `update_port_out` atualizam as portas dos registros de encaminhamento `rg_port_out` envolvidos no processo de recuperação. Cabe ressaltar que, dependendo da posição do *link* indisponível no balanceamento, alguns registros `rg_port_out` não necessitam de atualização.

Neste exemplo, ao tratar a falha no *link* número dois, a atualização das novas portas ocorre exclusivamente nos registros `rg_port_out_2` a `rg_port_out_5`. Por fim, a rotina de atualização copia as portas dos campos apropriados transportados pelo cabeçalho FRR para os registros na seguinte ordem: `rg_port_out_2=P4`, `rg_port_out_3=P5`, `rg_port_out_4=P6`, e `rg_port_out_5=P255`. A porta `P3` é substituída pela porta `P4` no registro `rg_port_out_2`, resultando na remoção da porta associada à falha no ECMP.

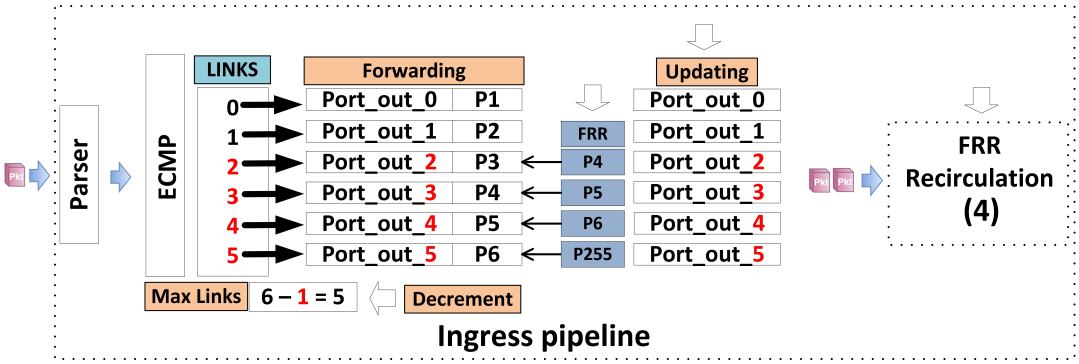

Figura 4.9 – (3) Atualização do ECMP.

4.6.2.4 Fim da Recuperação

O processo de *Fast-Reroute* é encerrado assim que a atualização do ECMP é concluída. A partir desse momento, o balanceamento assume a tarefa de distribuir o tráfego afetado pela falha entre os *enlaces* operacionais. Apesar disso, ainda há alguns pontos relevantes a serem abordados, como o tratamento dos pacotes que foram recirculados durante a recuperação e a rotina do contador que está em funcionamento. Para

tratar esses pontos pendentes, a tabela `frr_recirculation`, por meio de uma regra de encerramento da recuperação (diferente da regra que inicia a recuperação), redefine o registro `dr_rg_first_failure_count_wr`, responsável pela operação do contador, com o valor zero. Essa operação simples habilita o mecanismo de recuperação para atuar em uma nova ocorrência de falha subsequente ou em um momento posterior. Em seguida, os pacotes envolvidos no processo de recuperação têm o cabeçalho FRR removido, e o campo *EtherType* do cabeçalho Ethernet é definido com o valor 0x800, que representa o IPv4. Posteriormente, esses pacotes são direcionados para a *egress pipeline*, onde são submetidos a uma última recirculação de volta ao *ingress*. Nessa fase, o *ingress* dá início a um novo ciclo de processamento desses pacotes, marcando o encerramento completo da atuação de nossa arquitetura.

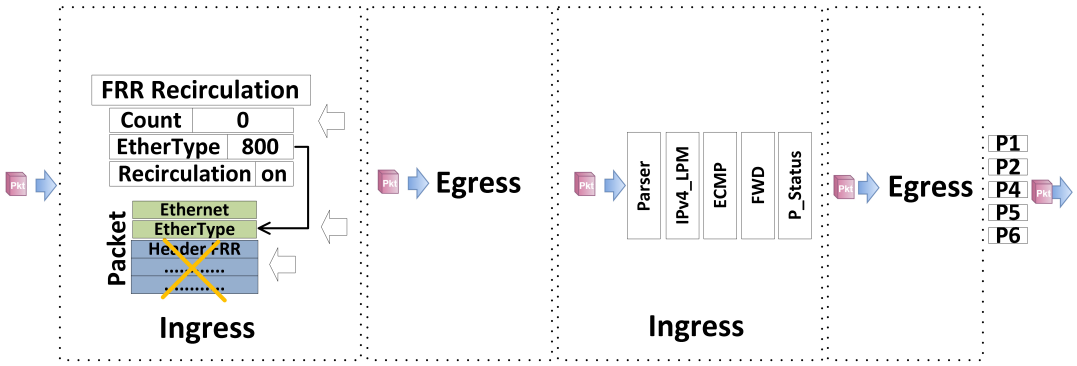

Figura 4.10 – (4) Fim da Recuperação.

4.6.3 Limitações no Tofino

A primeira versão da arquitetura FRR-ECMP, que serviu como precursora do protótipo RESISTING, foi desenvolvida na plataforma BMv2[4]. Essa versão empregava uma estrutura baseada em condicionais (*If/Else*) e, principalmente, operações que acessavam os mesmos registros múltiplas vezes por pacote nos *pipelines*. Na tentativa de implementar essa arquitetura na plataforma Tofino, a lógica e a estrutura adotadas no BMv2 se mostraram incompatíveis com o switch físico. Embora o Tofino supere consideravelmente o BMv2 em termos de desempenho, é importante destacar que sua flexibilidade e capacidade de personalização são mais limitadas em comparação com o BMv2. Um exemplo disso é a restrição de permitir apenas um único acesso por pacote ao registro, o que pode comprometer sua versatilidade e adaptabilidade em cenários específicos. Para superar essas limitações, desenvolvemos uma segunda versão no ambiente BMv2, que é o protótipo atualmente utilizado neste trabalho. Essa versão levou em conta uma lógica e

[4] P4-LFA - <https://github.com/danielbl1000/FRR-ECMP-First-Version>

estrutura mais compatíveis com o Tofino. Contudo, durante a fase de desenvolvimento do RESISTING no Tofino, novas restrições e divergências em relação ao BMv2 surgiram.

A principal restrição no aspecto de funcionalidade na plataforma Tofino em relação à nossa proposta de arquitetura, concebida no BMv2, foi a incapacidade de reduzir a quantidade de *links* operacionais utilizados na função *hash* através do plano de dados. Isso ocorre porque o mecanismo ECMP disponível na plataforma não oferece a possibilidade de armazenar o valor do parâmetro que define a quantidade de *links* no ECMP por meio de registro. Esse parâmetro é fundamental para o processo de recuperação, pois é manipulado visando reduzir os caminhos de saída usados na operação modular da função *hash*. Como solução paliativa, desenvolvemos uma estrutura adicional na implementação Tofino que utiliza um conceito semelhante ao de roteamento `default`, redirecionando o tráfego do *link* que não pode ser subtraído no ECMP para uma porta de saída disponível.

O processo de recuperação no protótipo Tofino é semelhante ao aplicado no BMv2, conforme ilustrado na Figura 4.9. No entanto, quando a função *hash* resulta no encaminhamento de tráfego para o último *link*, associado à porta 255 (P255), uma rotina adicional é acionada para redirecionar o tráfego para uma porta operacional. A tabela `default_path`, localizada na sequência das tabelas de encaminhamento `forwarding_tag` no *ingress pipeline*, intercepta o tráfego com o objetivo de encaminhá-lo para uma porta que permanece sempre operacional, desde que pelo menos uma porta esteja operacional no ECMP. Essa abordagem garante que o tráfego não seja descartado, mesmo em casos de novas falhas subsequentes.

É relevante ressaltar que essa solução é temporária e visa contornar a limitação enfrentada na plataforma. Dessa forma, como parte de trabalhos futuros, estamos explorando novas alternativas para abordar essa questão sem comprometer o desempenho do *Fast-Reroute*. O protótipo Tofino pode ser encontrado no repositório online mencionado no Apêndice G.

5 Avaliação Experimental

A seguir é apresentado um conjunto de experimentos que visam avaliar dois requisitos fundamentais para um mecanismo FRR-ECMP: o desempenho do processo de recuperação e a resiliência da rede. Os protótipos P4 utilizados nos experimentos estão disponíveis nas versões BMv2[1] e Tofino[2] e os logs dos experimentos estão disponíveis no repositório[3].

5.1 Metodologia

Nesta seção, apresentamos a metodologia aplicada para a avaliação do desempenho e da capacidade de resiliência da arquitetura proposta. Com o objetivo de alcançar tais metas, foram concebidos dois ambientes de avaliação: o cenário 01 analisa o desempenho da recirculação temporária de pacotes do RESISTING em um equipamento físico durante a ocorrência de 1, 3 e 5 falhas; o cenário 02 avalia a capacidade de resiliência do RESISTING em comparação com o PURR durante a ocorrência de 1, 2 e 3 falhas.

Cenário 01 - Hardware Tofino. A Figura 5.1 apresenta a topologia criada com um único switch físico emulando dois *leaves*. Para fins de compreensão, denominamos os *leaves* de *leaf* 1 e *leaf* 2. No entanto, é importante ressaltar que essa técnica não deve ser confundida com virtualização. O Tofino executa um único programa P4 com adaptações de regras e componentes para emular dois switches nos cenários de testes. Neste contexto, os *leaves* foram interconectados fisicamente por meio de seis cabos de 10 Gbps, representando os *uplinks*. O tráfego foi gerado pelo servidor *Trex/DPDK*, equipado com duas placas de rede de 10 Gbps que simulam um *host* em cada uma dessas placas. O protótipo, juntamente com as regras adaptadas, permite que o *leaf 1* realize o balanceamento ECMP, além da recuperação FRR no ambiente. O balanceamento foi configurado com seis *links*: 1, 2, 3, 4, 5 e 6, associados às portas lógicas P25, P26, P27, P180, P181 e P182, respectivamente. Enquanto isso, o *leaf 2* complementa o cenário, apenas realizando o balanceamento de tráfego, sem a aplicação da funcionalidade de recuperação. O servidor empregado para originar o tráfego via DPDK e conduzir a análise com a ferramenta *Wireshark* está conectado ao switch *leaf 1* por meio da interface de rede P0 (TX) e ao switch *leaf 2* via interface P1 (RX).

Os equipamentos utilizados neste cenário foram um *Tofino Wedge 100B-32X*

[1] RESISTING BMv2 - <https://github.com/danielbl1000/P4-RESISTING-bmv2> e Apêndice F
[2] RESISTING Tofino - <https://github.com/danielbl1000/P4-RESISTING-tofino>
[3] Logs - <https://github.com/danielbl1000/WGRS>

com *Software Develpment Environment* (SDE) *Bf-Sde-9.9.0* e um servidor DPDK com processador *Intel Xeon* de 2,20 GHz, 62 GB de RAM e placa de rede *Intel Ethernet Connection X552 10GbE SFP*.

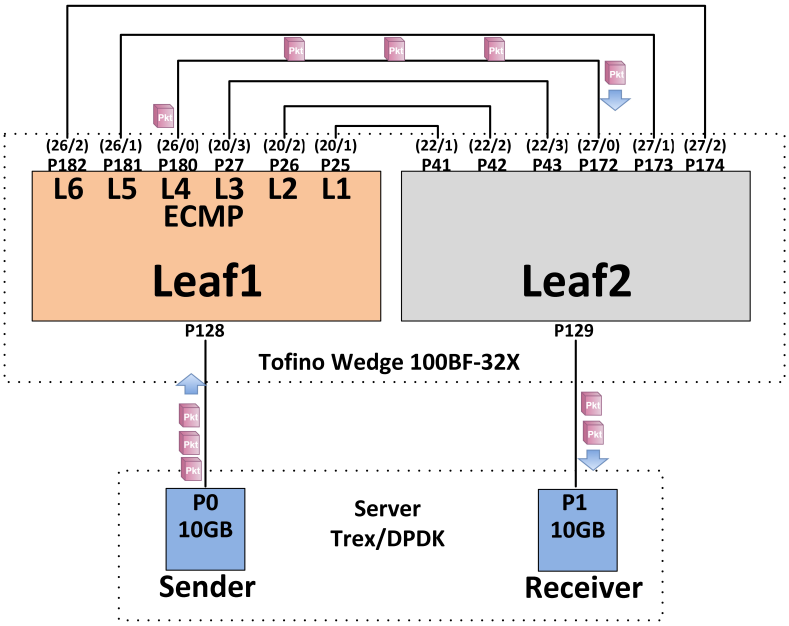

Figura 5.1 – Cenário 01 - Tofino.

Cenário 02 - BMv2 Software. O experimento de resiliência, juntamente com os cálculos das Equações (5.1) e (5.2), foi conduzido neste cenário, instanciado no emulador Mininet com uma topologia *Clos* em escala reduzida utilizando o BMv2, conforme ilustrado na Figura 5.2. O propósito desse cenário é emular uma estrutura de rede, capaz de realizar um balanceamento equilibrado dos fluxos de pacotes entre os *uplinks* operacionais, mantendo uma ocupação inferior a 50% por *link*. Assim, partindo desse cenário base, as falhas são aplicadas nos *uplinks* do switch *leaf 1* e, posteriormente, os mecanismos RESISTING e PURR são ativados. Para viabilizar essa configuração, o ambiente experimental se fundamenta em algumas premissas: cada um dos seis *uplinks* é capaz de suportar no máximo 1000 fluxos, totalizando 6000 fluxos de pacotes; o *host* (h1) gera simultaneamente 2400 fluxos de pacotes TCP distintos, com o objetivo de permitir que o ECMP realize uma distribuição equilibrada entre os seis *uplinks*, mantendo uma ocupação aproximada de 400 fluxos em cada *uplink*, ou seja, cerca de 40% de ocupação.

Cabe salientar que o balanceamento ECMP é uma técnica de roteamento que promove a distribuição dos fluxos de pacotes entre os enlaces, sem considerar os aspectos de tamanho do pacote e/ou a taxa de transmissão do fluxo. Diante disso, o experimento enfatiza a distribuição por fluxos, conforme determinado pelo resultado da função *hash* aplicada no mecanismo ECMP.

Os dispositivos utilizados neste cenário foram um servidor *Intel Xeon* de 2,13

GHz, 25 GB de RAM, SO Linux (Ubuntu 5.4.0-6ubuntu1-16.04.11), Mininet v2.2.1, BMv2 e *target* `simple_switch1.13.0-33e221fd`.

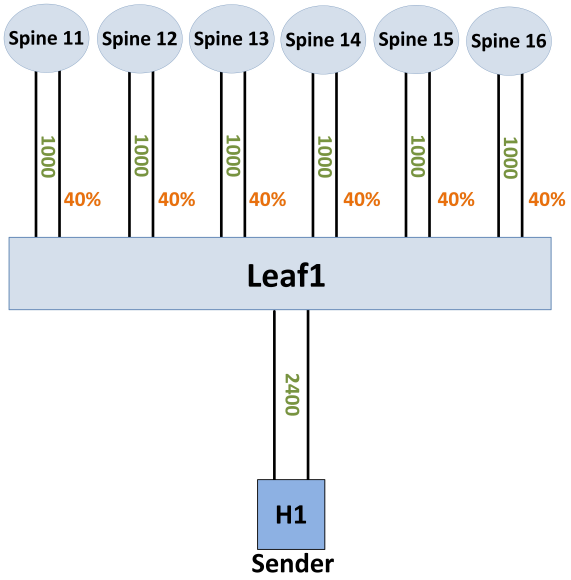

Figura 5.2 – Cenário 02 - Topologia *Clos leaf-spine*

5.2 Desempenho da arquitetura

O mecanismo de recuperação, ao empregar a estratégia de recirculação persistente de pacotes no plano de dados, acarreta um aumento na latência do processamento de pacotes e uma redução no *throughput* (CHIESA *et al.*, 2019). Diante disso, surge a necessidade de avaliar se a técnica de recirculação temporária de pacotes adotada possui um impacto negativo no desempenho da arquitetura proposta.

5.2.1 Experimento 1: Impacto da recirculação de pacotes no Tofino

Este experimento tem como objetivo avaliar a métrica da quantidade de pacotes submetidos à recirculação durante a ocorrência de uma, três e cinco falhas, a fim de identificar os elementos que possam influenciar a quantidade de pacotes recirculados durante a recuperação e mensurar o impacto da recirculação no fluxo de pacotes. Com esse objetivo, foram executadas avaliações contemplando o cenário 01 (Figura 5.1). O experimento consiste em gerar no servidor *Trex/DPDK* a cada 10 segundos um fluxo de pacotes UDP de tamanho grande (1514 B), médio (814 B) e pequeno (114 B), da placa de rede P0 (h1) para a placa de rede P1 (h2), abrangendo quatro diferentes taxas de transmissão: 1) 100 Mbps; 2) 1 Gbps; 3) 5 Gbps; e 4) 9 Gbps, e aplicação de 1, 3 e 5 falhas simultâneas.

Para avaliar a quantidade de pacotes submetidos à recirculação durante a ocorrência de uma, três e cinco falhas, foram capturadas amostras para análise de cada fluxo de pacotes na ferramenta *Wireshark*. Além disso, é importante destacar que as falhas foram introduzidas por meio do plano de controle na tabela `port_status` do *leaf* 1, o que causou o acionamento do FRR-ECMP proposto. Por exemplo, o cenário da avaliação do fluxo UDP de 114 bytes com aplicação de 5 falhas enviou 95.339.027 pacotes UDP na taxa de 9 Gbps e apenas 31 pacotes foram recirculados durante o processo de recuperação. A Figura 5.3 apresenta os resultados da quantidade de pacotes submetidos à recirculação devido a falhas, na forma de gráficos, para os fluxos UDP com tamanhos de 1514, 814 e 114 Bytes, juntamente com as taxas de transmissão mencionadas anteriormente. Nas Figuras 5.4, 5.5 e 5.6, apresentamos os resultados da proporção entre o total de pacotes gerados e a quantidade de pacotes submetidos à recirculação por falha, em cada fluxo de pacote, comparando as taxas de transmissão mencionadas anteriormente. Adicionalmente, disponibilizamos a tabela e os dados do experimento[4] como contribuição extra.

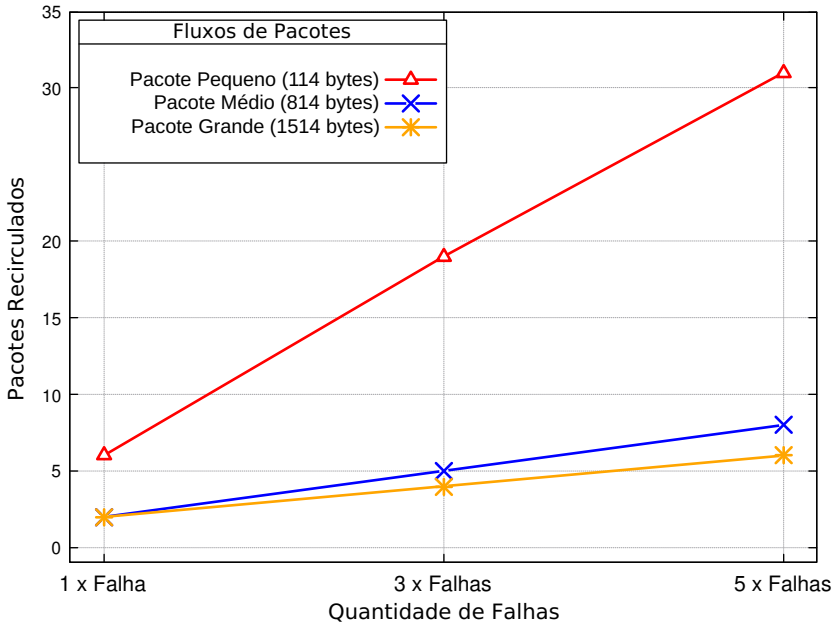

Figura 5.3 – Quantidade de pacotes recirculados por falha nos três fluxos.

Os resultados apresentados no gráfico da Figura 5.3 evidenciaram uma correlação entre o tamanho do pacote e o número de falhas aplicadas. À medida que o tamanho do pacote diminui e a quantidade de falhas aumenta, observa-se um aumento na quantidade de pacotes submetidos à recirculação, independentemente da taxa de transmissão aplicada. Por outro lado, quando o tamanho do pacote aumenta e a quantidade de falhas reduz, ocorre o inverso: uma redução na quantidade de pacotes recirculados. É importante destacar que o aumento progressivo da taxa de transmissão, de 100 Mbps para 1 Gbps, 5 Gbps e 9 Gbps, nos fluxos de pacotes durante os experimentos, ocasionou um

[4] Disponível em: https://github.com/danielbl1000/P4-RESISTING-tofino/tree/main/ex01-1

aumento proporcional na quantidade total de pacotes gerados. No entanto, essa variação nas taxas de transmissão não teve impacto sobre a quantidade de pacotes recirculados, a qual permaneceu constante em todos os cenários testados. Por exemplo, ao analisarmos o fluxo do pacote de 114 bytes, observamos a geração de 1 milhão de pacotes a uma taxa de 100 Mbps e 95 milhões de pacotes na avaliação com uma taxa de 9 Gbps. Apesar das diferentes taxas de transmissão, a quantidade de pacotes recirculados permaneceu a mesma em ambos os casos durante cada evento de falhas.

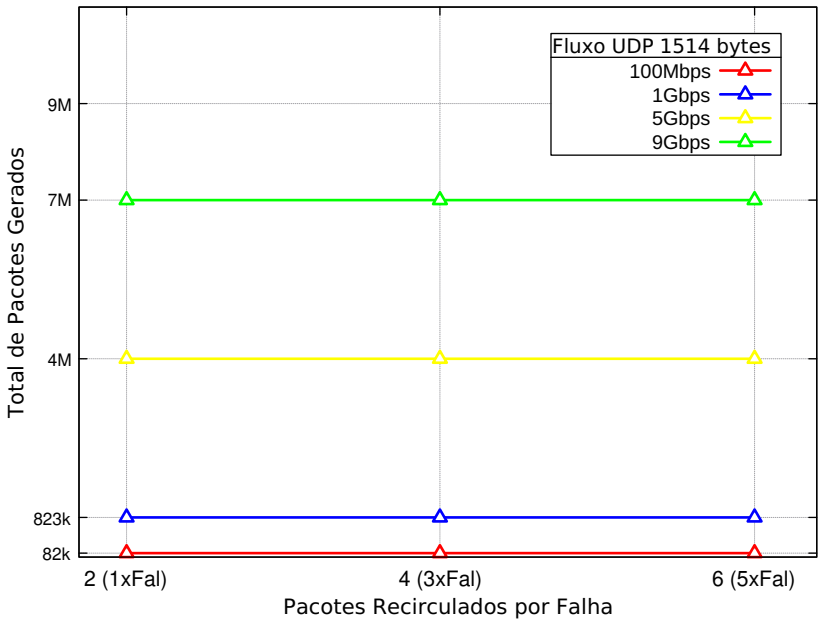

Figura 5.4 – Quantidade de pacotes recirculados por falha no fluxo de 1514 Bytes.

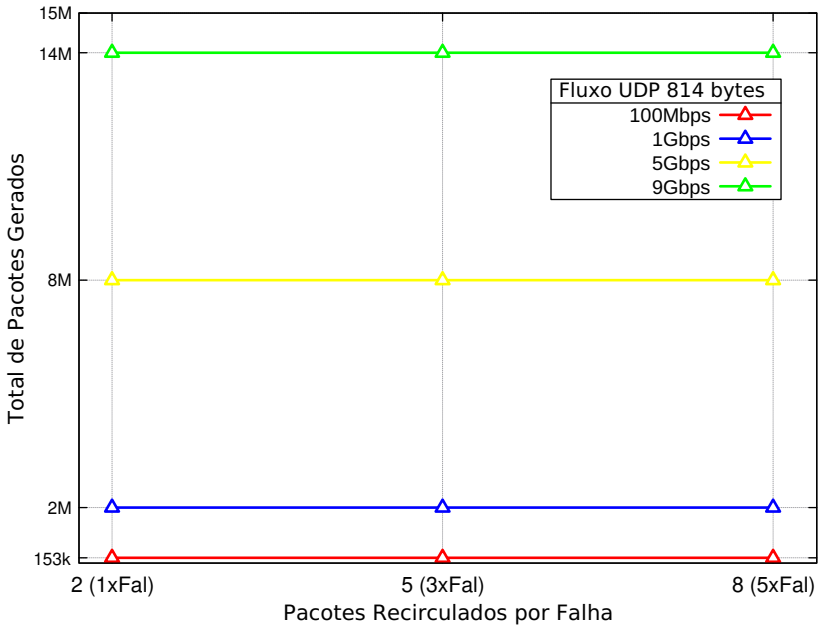

Figura 5.5 – Quantidade de pacotes recirculados por falha no fluxo de 814 Bytes.

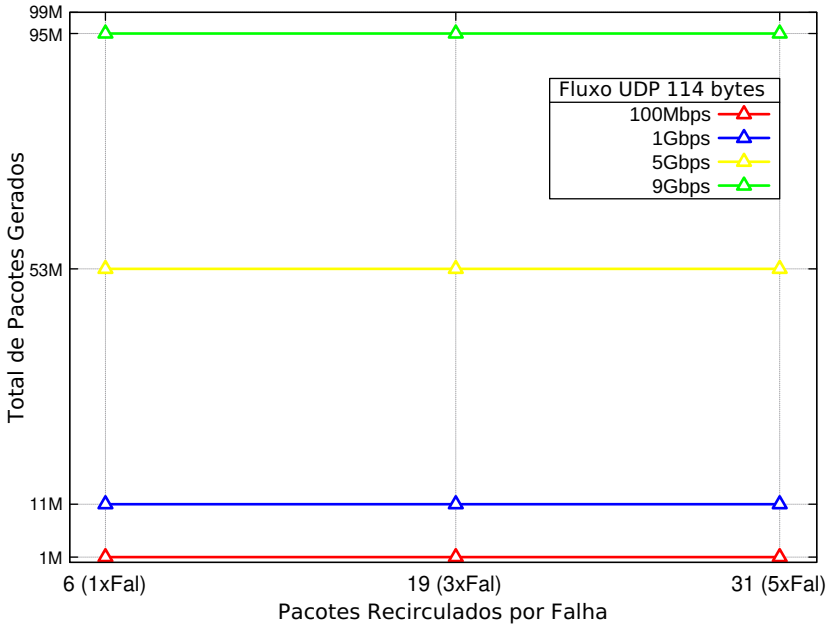

Figura 5.6 – Quantidade de pacotes recirculados por falha no fluxo de 114 Bytes.

Os resultados apresentados nos gráficos das Figuras 5.4, 5.5 e 5.6 conferiram ênfase à relação entre o total de pacotes gerados durante os testes e a proporção de pacotes que se submetem à recirculação devido a falhas. Nesses gráficos, que abrangem três distintos fluxos e pacotes - pequeno (114 B), médio (814 B) e grande (1514 B) - observa-se um contingente mínimo de pacotes que passaram pelo processo de recirculação temporária durante a fase de recuperação, em comparação com a quantidade de pacotes gerados. A estabilidade na quantidade de pacotes recirculados por falha, independentemente da taxa de transmissão aplicada, é outro aspecto destacado pelos resultados dos gráficos. Contudo, devido a limitações de recursos computacionais no laboratório de avaliação, o *throughput* máximo aplicado nos experimentos foi de 9 Gbps.

O experimento 1 foi realizado em um cenário de testes composto por equipamentos reais com o propósito de analisar o desempenho do processo de recuperação do RESISTING, levando em consideração o impacto do método de recirculação temporária de pacotes nos fluxos em condições de falhas. Os resultados do experimento sugerem a viabilidade operacional da arquitetura proposta, visto que apenas um contingente mínimo de pacotes dos fluxos foi recirculado pela implementação, e o aumento na taxa de transmissão não interferiu nos pacotes recirculados. A variação na taxa de transmissão sugere a manutenção da estabilidade na quantidade de pacotes recirculados nos cenários que exploram taxas de transmissão acima de 9 Gbps. Com base nesses resultados, concluímos que o custo operacional, decorrente do método de recirculação empregado nos fluxos de pacotes durante a recuperação, não causaria um impacto significativo no aumento da latência nem na redução do *throughput*.

5.3 Resiliência de rede

O objetivo deste experimento é avaliar a capacidade de resiliência da arquitetura proposta em comparação ao estado da arte em cenários de falha. Durante as simulações, as falhas são aplicadas, desencadeando uma resposta imediata de recuperação por parte dos mecanismos *Fast Reroute*. Nos testes envolvendo o PURR, a estratégia de recuperação redireciona todos os fluxos de pacotes para um único *link* backup, sem considerar a possibilidade de balanceamento de fluxo. Por outro lado, nos cenários com o RESISTING, o método de recuperação remove os links afetados pelas falhas, permitindo que o ECMP continue a distribuir os fluxos de pacotes entre os caminhos operacionais. Nesse contexto, é possível comparar a habilidade de recuperação dos mecanismos envolvidos, levando em consideração o equilíbrio na distribuição dos fluxos de pacotes após a indisponibilidade dos *enlaces*.

Em um cenário de infraestrutura de data center, onde o mecanismo *Fast Reroute* pode ser empregado para recuperar os *links* utilizados pelo balanceamento ECMP, consideramos uma solução de FRR potencialmente resiliente, quando ela é integrada com o ECMP e demonstra a capacidade de lidar com uma ou várias falhas nos caminhos utilizados pelo ECMP, sem comprometer a eficiência na utilização dos recursos de banda e reduzindo o risco de sobrecarga nos *enlaces* backup. Para calcular **o grau de resiliência** dos mecanismos de recuperação abordados neste trabalho em todos os cenários possíveis de falha, seguimos o conceito de resiliência de rede abordado nos trabalhos da literatura (OMER *et al.*, 2009; FIGUEIRA; VASCONCELOS, 2011). Esses trabalhos definem a resiliência de rede com base na quantidade de fluxo ou tráfego entregue após uma redução ou perda de pacotes. É importante destacar que os fluxos de pacotes gerados durante os experimentos possuem um tamanho fixo, uma vez que o ECMP distribui os fluxos entre os caminhos sem considerar o tamanho específico de cada pacote, seja ele grande ou pequeno.

Na Equação (5.1), o Valor Entregue (VE) de fluxo expressa o nível de resiliência de rede para cada topologia de rede testada durante o experimento. É importante considerar que o termo “topologia de rede”, utilizado no cálculo de resiliência de rede abordado nos trabalhos anteriormente referenciados, representa cada cenário no qual o VE calcula o fluxo/tráfego que foi entregue com ou sem falha. Basicamente, o Valor Entregue (VE) possibilita avaliar a capacidade de resiliência em preservar o fluxo de pacotes em várias situações de falhas. Com esse propósito, o VE é calculado a partir da diferença entre o Valor Demandado (VD) e a PERDA. O VD corresponde à quantidade de fluxo/tráfego que a rede deve encaminhar, enquanto a PERDA representa a quantidade de demanda que foi perdida decorrente a falha.

$$VE = \frac{VD - PERDA}{VD} \tag{5.1}$$

Com o intuito de consolidar os resultados dos cálculos VE, a Equação (5.2) calcula os valores médios dos resultados obtidos na Equação (5.1) para todas as topologias de rede testadas. A soma dos resultados dos calculados VE é então dividida pela quantidade de topologias de rede avaliadas, ou seja, pelo número de cenários de falha considerados.

$$VEMedio = \frac{\sum VE}{Quantidade.Topologia.Rede} \tag{5.2}$$

As simulações do experimento de resiliência, acompanhadas dos cálculos correspondentes, foram conduzidas no Cenário 02, conforme ilustrado na Figura 5.2. Para demonstrar o cenário de simulação e a execução do cálculo VE, examinamos três cenários emulados com base no Cenário 02, conforme representado na Figura 5.7. Inicialmente, o *host* (h1), conectado ao *leaf 1*, gerou uma demanda de 2400 fluxos distintos nos três cenários. Essa demanda corresponde ao Valor Demandado (VD) no cálculo do Valor Entregue (VE), conforme definido na Equação (5.1). Em seguida, o balanceamento ECMP realizou uma distribuição equilibrada desses fluxos de pacotes entre os *uplinks*, suportando no máximo 1000 fluxos cada, nas seguintes proporções: 429 (42%) para P1, 406 (40%) para P2, 371 (37%) para P3, 393 (39%) para P4, 399 (39%) para P5 e 402 (40%) para P6. Essa distribuição está ilustrada na Figura 5.7(a). Para efeitos de avaliação, focamos exclusivamente no tráfego de saída do *leaf 1* em direção aos *spines*, desconsiderando o tráfego de retorno. Isso ocorre porque o mecanismo de recuperação é aplicado ao fluxo de pacotes no sentido de saída das interfaces do switch. Conforme as premissas do Cenário 02, cada *uplink* tem a capacidade máxima de suportar 1000 fluxos de pacotes. Assim, quando um *uplink* alcança essa marca, consideramos que o *enlace* atingiu 100% de sua capacidade.

Neste cenário base, a taxa de ocupação de fluxos de pacotes entre os *uplinks* é inferior a 50%, proporcionando uma margem significativa para acomodar os fluxos de pacotes de *links* indisponíveis devido a eventos de falhas. Como exemplo, a Figura 5.7(a) ilustra o cenário do switch *leaf 1* sem falhas, atendendo a 100% da demanda de fluxo de pacotes. Nas Figuras 5.7(b) e (c), foram aplicadas falhas simultâneas nas portas 1 (P1), 5 (P5) e 6 (P6) no *leaf 1*. Após o evento de falha, o mecanismo de recuperação PURR redirecionou os fluxos de pacotes dos três *uplinks* para a porta 2 (P2), conforme ilustrado na Figura 5.7(b). Como resultado, a porta 2 (P2) ultrapassou 100% de ocupação, resultando em um Valor Entregue (VE) de 0,735 (73%), ou seja, apenas 73% do Valor Demandado (2400 fluxos) foram entregues, resultando na PERDA de 648 (27%) fluxos.

Na Figura 5.7(c), após a detecção das falhas, o FRR-ECMP proposto removeu as portas P1, P5 e P6 do balanceamento ECMP, permitindo que a demanda dos três *uplinks* indisponíveis fosse redistribuída de forma uniforme entre os *links* remanescentes, evitando a sobrecarga de fluxos de pacotes e entregando 100% da demanda (VE = 1).

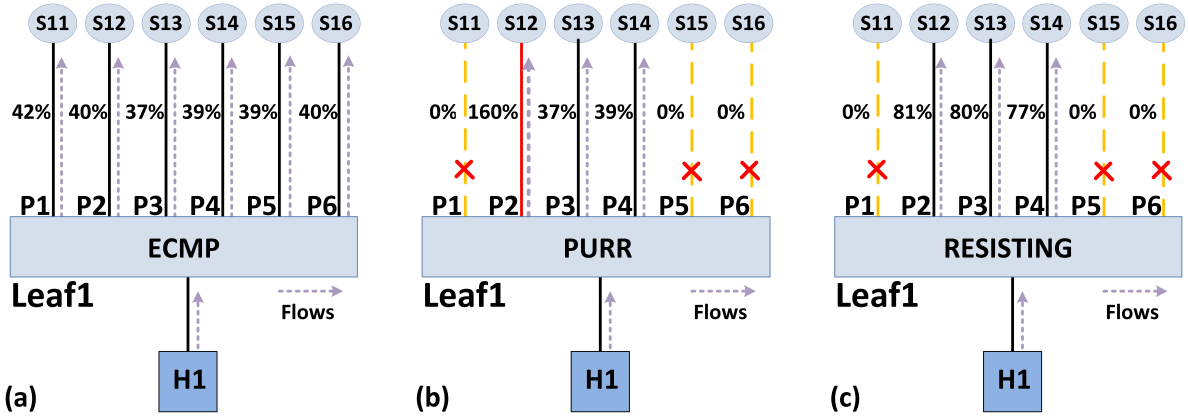

Figura 5.7 – Quantidade de fluxos entregues nos cenários de falhas com os mecanismos PURR e RESISTING.

5.3.1 Experimento 2: Capacidade de resiliência da arquitetura

Sem mais delongas, vamos agora apresentar o experimento que avalia a capacidade de resiliência da arquitetura proposta em cenários com uma, duas e três falhas, em comparação com o PURR, um mecanismo considerado estado da arte. No Cenário 2 (Figura 5.2), foram geradas todas as simulações possíveis de falhas envolvendo até três *uplinks*. Para cada simulação de falha, que também é referenciada como topologia de rede no contexto do cálculo VE, coletamos e aplicamos os valores de VD e PERDA na Equação (5.1). A Tabela E.1 apresenta a quantidade de falhas aplicadas nas simulações para os mecanismos de recuperação, bem como a quantidade de topologias de rede correspondentes. Por exemplo, no cenário de uma única falha de porta no *leaf* 1, as simulações incluem as portas P1, P2, P3, P4, P5 e P6. Já no cenário de duas falhas, as combinações possíveis são: (P1, P2), (P1, P3), (P1, P4) e assim por diante. Nos Apêndices B, C, D e E, as sequências das simulações e os resultados dos cálculos são documentados.

Inicialmente, após a conclusão dos cálculos VE, os resultados foram aplicados à Equação (5.2), que, por sua vez, calcula o VE Médio do experimento. Em seguida, os resultados do VE Médio foram consolidados em um gráfico, apresentado na Figura 5.8, destacando a porcentagem média de perda dos fluxos de pacotes pelos mecanismos durante as simulações para 1, 2 e 3 falhas. Em relação a uma falha, os resultados mostram que o RESISTING e o PURR obtiveram um nível de resiliência de 100% entregando todos os fluxos de pacotes nas simulações, independente de qual porta de *uplink* sofreu indisponibilidade no *leaf 1*. Não ocorreram perdas durante os experimentos com uma falha.

Quantidade de Falhas	Mecanismo FRR	Quantidade de Top. de Rede (simulações)
1 Falha	PURR RESISTING	6
2 Falhas	PURR RESISTING	15
3 Falhas	PURR RESISTING	20

Tabela 5.1 – Cálculo VE: quantidade de simulações.

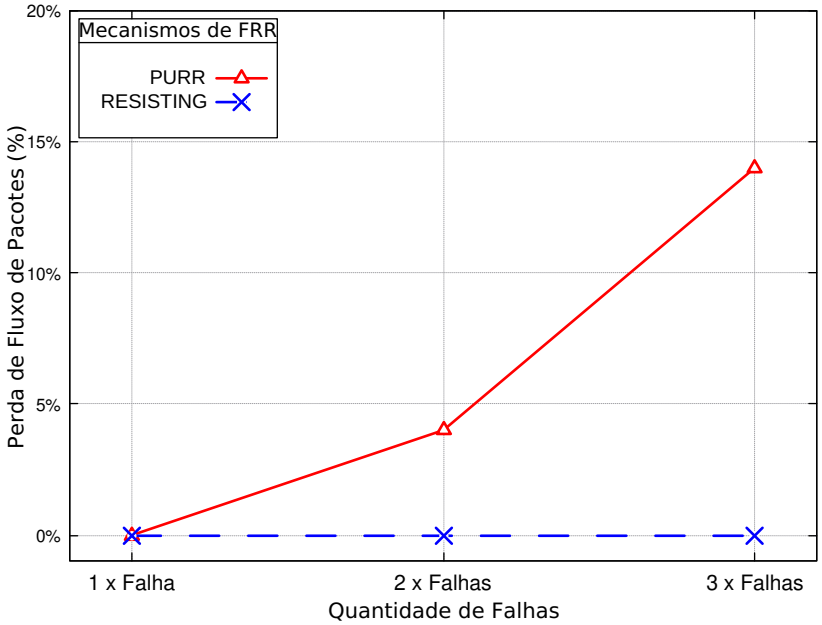

Figura 5.8 – Comparação de perda média entre o PURR e o RESISTING.

No cenário com duas falhas, o PURR obteve resultados de perda em 6 combinações de portas: (P1, P2), (P1, P6), (P2, P3), (P3, P4) e (P4, P5). O VE médio obtido foi de 96%, portanto, 4% dos fluxos médio sofreram perda. Neste cenário, a nossa arquitetura entregou 100% da demanda para todas as possíveis combinações de portas durante as simulações. Finalmente, no drástico cenário com três falhas, somente duas combinações de falhas não obtiveram perda pelo PURR: (P1, P3, P5) e (P2, P4, P6). As demais simulações mostraram perda e, por este motivo, o resultado médio de degradação foi de 14%, enquanto o mecanismo de recuperação exposto RESISTING não apresentou perda de fluxos de pacotes nas 20 topologias avaliadas. Com base nesses resultados, o RESISTING demonstrou uma capacidade de resiliência superior ao mecanismo estado da arte, especialmente durante eventos de três falhas. Isso indica que a arquitetura proposta pode oferecer uma solução eficiente, capaz de manter uma distribuição uniforme dos fluxos de pacotes em conjunto com o serviço de balanceamento.

6 Conclusões e Trabalhos Futuros

Este estudo propõe um novo mecanismo de *Fast Reroute* integrado ao *Equal Cost Multipath* em switches programáveis na linguagem P4. O mecanismo tem como objetivo a recuperação de falhas nos caminhos do balanceamento ECMP, removendo a porta de saída do switch associada ao *link* afetado pela falha. Isso é realizado por meio de um processo de recuperação baseado na realocação das portas dos *links* remanescentes do ECMP. Desse modo, após a conclusão do processo de recuperação da falha, a implementação assegura a uniformidade na distribuição dos fluxos de pacotes entre múltiplos *links* operacionais no balanceamento. Essa solução contribui efetivamente para otimizar o uso da largura de banda disponível, ao mesmo tempo que previne o risco de sobrecarga nos *enlaces*, responsáveis por absorver o fluxo excedente do caminho afetado pela falha. Para alcançar esses resultados, foram realizadas as seguintes atividades: (i) desenvolvimento de dois protótipos na linguagem P4, um para o switch virtual BMv2 e outro para o switch Tofino; (ii) condução de um primeiro experimento que avaliou o impacto da recirculação temporária de pacotes na arquitetura proposta no Tofino; (iii) realização de um segundo experimento no ambiente virtual Mininet, que avaliou a capacidade de resiliência do RESISTING em relação ao PURR, mecanismo de recuperação estado da arte.

Esta pesquisa aborda os desafios inerentes à concepção de uma arquitetura FRR-ECMP em switches programáveis P4. Dentre esses desafios, destaca-se a complexa tarefa de desenvolver uma arquitetura capaz de realizar a recuperação de uma ou múltiplas falhas no balanceamento ECMP por meio do plano de dados, sem a necessidade de intervenções diretas do plano de controle. Este esforço visa assegurar uma recuperação FRR eficiente e autônoma, apresentando uma solução indicada para cenários de redes com múltiplos caminhos que empregam o balanceamento ECMP, como, por exemplo, infraestruturas de data center.

Os experimentos foram retratados, evidenciando a eficácia dos métodos propostos: (1) impacto da recirculação temporária de pacotes da arquitetura no Tofino e (2) capacidade de resiliência da arquitetura. Os resultados do experimento (1) demonstram que o protótipo apresentado foi capaz de se recuperar de uma ou múltiplas falhas sem afetar significativamente a latência e o *throughput* dos fluxos de pacotes envolvidos no processo de recuperação, recirculando apenas uma quantidade mínima de pacotes temporariamente em relação à quantidade de pacotes que foram gerados, indicando estabilidade na quantidade de pacotes recirculados mesmo com o aumento progressivo da taxa de transmissão. No experimento (2), os resultados expressaram que o mecanismo

RESISTING não apresentou perdas de fluxos de pacotes nos eventos de até três falhas em relação ao FRR do estado da arte. Os resultados sugerem que a arquitetura implementada está capacitada a lidar com cenários de falhas nos *links* do balanceamento ECMP, garantindo uma redistribuição equilibrada dos fluxos de pacotes afetados entre os *enlaces* operacionais.

Planejamos dar continuidade a este trabalho explorando os seguintes temas em trabalhos futuros: (1) a implementação do RESISTING em outras topologias de rede, como a *Fat-Tree*, e a introdução de mecanismos de balanceamento mais sofisticados em comparação ao ECMP, considerando o tamanho dos fluxos de pacotes e/ou taxa de transmissão como critérios adicionais para a distribuição de fluxo de pacotes; (2) a integração de métodos reais de detecção do estado operacional da porta e/ou *link* com a arquitetura de recuperação para oferecer uma solução mais adequada para cenários operacionais reais; (3) expandir o escopo dos experimentos, incluindo a análise do *Flow Completion Time* e a avaliação da latência dos pacotes recirculados por meio de telemetria.

Embora o RESISTING tenha apresentado resultados promissores, identificamos uma limitação no protótipo desenvolvido para o Tofino. O componente encarregado do controle dos *links* operacionais no ECMP não permite a redução da quantidade de caminhos operacionais após a conclusão do processo de recuperação, através do plano de controle. Portanto, é crucial explorar alternativas para superar essa limitação na plataforma, com o objetivo de equalizar as funcionalidades entre os protótipos Tofino e BMv2.

Referências

ALIZADEH, M.; EDSALL, T. On the data path performance of leaf-spine datacenter fabrics. In: *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*. [S.l.: s.n.], 2013. p. 71–74. Citado na página 43.

ATLAS, A.; ZININ, A. D. *Basic Specification for IP Fast Reroute: Loop-Free Alternates*. RFC Editor, 2008. RFC 5286. (Request for Comments, 5286). Disponível em: <https://www.rfc-editor.org/info/rfc5286>. Citado na página 32.

BOSSHART, P.; DALY, D.; GIBB, G.; IZZARD, M.; MCKEOWN, N.; REXFORD, J.; SCHLESINGER, C.; TALAYCO, D.; VAHDAT, A.; VARGHESE, G.; WALKER, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, v. 44, n. 3, p. 87–95, 7 2014. ISSN 0146-4833. Disponível em: <https://dl.acm.org/doi/10.1145/2656877.2656890http://arxiv.org/abs/1312.1719>. Citado 2 vezes nas páginas 16 e 24.

BOSSHART, P.; GIBB, G.; KIM, H.-S.; VARGHESE, G.; MCKEOWN, N.; IZZARD, M.; MUJICA, F.; HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. New York, NY, USA: Association for Computing Machinery, 2013. (SIGCOMM '13), p. 99–110. ISBN 9781450320566. Disponível em: <https://doi.org/10.1145/2486001.2486011>. Citado na página 24.

BRAUN, W.; MENTH, M. Loop-free alternates with loop detection for fast reroute in software-defined carrier and data center networks. *Journal of Network and Systems Management*, v. 24, 07 2016. Citado na página 33.

CAO, Z.; WANG, Z.; ZEGURA, E. Performance of hashing-based schemes for internet load balancing. In: *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*. [S.l.: s.n.], 2000. v. 1, p. 332–341 vol.1. Citado 2 vezes nas páginas 44 e 46.

CHIESA, M.; KAMISIŃSKI, A.; RAK, J.; RETVARI, G.; SCHMID, S. A survey of fast recovery mechanisms in the data plane. *Authorea Preprints*, Authorea, 2023. Citado na página 17.

CHIESA, M.; KAMISIńSKI, A.; RAK, J.; RéTVáRI, G.; SCHMID, S. A survey of fast-recovery mechanisms in packet-switched networks. *IEEE Communications Surveys Tutorials*, v. 23, n. 2, p. 1253–1301, 2021. Citado 2 vezes nas páginas 28 e 30.

CHIESA, M.; KINDLER, G.; SCHAPIRA, M. Traffic engineering with equal-cost-multipath: An algorithmic perspective. *IEEE/ACM Transactions on Networking*, v. 25, n. 2, p. 779–792, 2017. Citado na página 44.

CHIESA, M.; SEDAR, R.; ANTICHI, G.; BOROKHOVICH, M.; KAMISIŃSKI, A.; NIKOLAIDIS, G.; SCHMID, S. PURR: a primitive for reconfigurable fast reroute. *In Proceedings of the 15th International Conference on Emerging Networking Experiments*

And Technologies, CoNEXT '19, New York, NY, USA, p. 1–14, 2019. Citado 7 vezes nas páginas 16, 17, 18, 31, 32, 34 e 60.

CISCO. *Designing Cisco Data Center Infrastructure*. 2012. Disponível em <https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book/DCInfra_2a.html>. Acesso em: 11/07/2023. Citado na página 23.

CSIKOR, L.; RÉTVÁRI, G.; TAPOLCAI, J. High availability in the future internet. In: GALIS, A.; GAVRAS, A. (Ed.). *The Future Internet*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 64–76. ISBN 978-3-642-38082-2. Citado 3 vezes nas páginas 23, 32 e 33.

FIGUEIRA, M.; VASCONCELOS, D. E. Emprego de resiliência na gerência de redes. In: *Revista Militar de Artigos Ciência e Tecnologia*. Rio de Janeiro: [s.n.], 2011. XXXIII, p. 32–41. Disponível em: <https://rmct.ime.eb.br/arquivos/RMCT_3_tri_2016_web/RMCT_3TRI_WEB_2016.pdf>. Citado na página 64.

GOBATTO, L.; RODRIGUES, P.; SAQUETTI, M.; CORDEIRO, W.; AZAMBUJA, J. R. Programmable data planes meets in-network computing: A review of the state of the art and prospective directions. *Journal of Integrated Circuits and Systems*, v. 16, n. 2, 2021. Citado na página 16.

GREENBERG, A.; HAMILTON, J. R.; JAIN, N.; KANDULA, S.; KIM, C.; LAHIRI, P.; MALTZ, D. A.; PATEL, P.; SENGUPTA, S. VL2. *ACM SIGCOMM Computer Communication Review*, v. 39, n. 4, p. 51–62, aug 2009. ISSN 0146-4833. Disponível em: <https://dl.acm.org/doi/10.1145/1897852.1897875https://dl.acm.org/doi/10.1145/1594977.1592576>. Citado na página 43.

HAUSER, F.; HäBERLE, M.; MERLING, D.; LINDNER, S.; GUREVICH, V.; ZEIGER, F.; FRANK, R.; MENTH, M. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, Elsevier BV, v. 212, p. 103561, mar 2023. Disponível em: <https://doi.org/10.1016%2Fj.jnca.2022.103561>. Citado na página 24.

HOPPS, C. RFC, *RFC2992: Analysis of an Equal-Cost Multi-Path Algorithm*. USA: RFC Editor, 2000. RFC 2992, RFC Editor https://www.rfc-editor.org/info/rfc2992. Disponível em: <https://www.rfc-editor.org/info/rfc2992>. Citado na página 29.

KIANPISHEH, S.; TALEB, T. A survey on in-network computing: Programmable data plane and technology specific applications. *Commun. Surveys Tuts.*, IEEE Press, v. 25, n. 1, p. 701–761, jan 2023. ISSN 1553-877X. Disponível em: <https://doi.org/10.1109/COMST.2022.3213237>. Citado na página 24.

KREUTZ, D.; RAMOS, F. M. V.; VERISSIMO, P.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. *Software-Defined Networking: A Comprehensive Survey*. 2014. Citado 4 vezes nas páginas 16, 23, 25 e 26.

LUZ, G.; ROCHA, A.; ALMEIDA, L.; VERDI, F. Infarr: Um algoritmo para reroteamento rápido em planos de dados programáveis. In: *Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. Porto Alegre, RS, Brasil: SBC, 2022. p. 154–167. ISSN 2177-9384. Disponível em: <https:

//sol.sbc.org.br/index.php/sbrc/article/view/21168>. Citado 2 vezes nas páginas 29 e 39.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar 2008. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/1355734.1355746>. Citado na página 24.

MERLING, D.; LINDNER, S.; MENTH, M. P4-based implementation of BIER and BIER-FRR for scalable and resilient multicast. *Journal of Network and Computer Applications*, v. 169, p. 102764, nov 2020. ISSN 10848045. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S1084804520302381>. Citado 4 vezes nas páginas 17, 30, 31 e 33.

MICHEL, O.; BIFULCO, R.; RéTVáRI, G.; SCHMID, S. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 54, n. 4, may 2021. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3447868>. Citado na página 26.

NUNES, B. A. A.; MENDONCA, M.; NGUYEN, X.-N.; OBRACZKA, K.; TURLETTI, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys Tutorials*, v. 16, n. 3, p. 1617–1634, 2014. Citado na página 23.

OMER, M.; NILCHIANI, R.; MOSTASHARI, A. Measuring the resilience of the global internet infrastructure system. In: *2009 3rd Annual IEEE Syscon*. [S.l.: s.n.], 2009. p. 156–162. Citado na página 64.

PATEL, P.; BANSAL, D.; YUAN, L.; MURTHY, A.; GREENBERG, A.; MALTZ, D. A.; KERN, R.; KUMAR, H.; ZIKOS, M.; WU, H.; KIM, C.; KARRI, N. Ananta: Cloud scale load balancing. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 43, n. 4, p. 207–218, aug 2013. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2534169.2486026>. Citado na página 30.

SEDAR, R.; BOROKHOVICH, M.; CHIESA, M.; ANTICHI, G.; SCHMID, S. Supporting Emerging Applications With Low-Latency Failover in P4. In: *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*. New York, NY, USA: ACM, 2018. p. 52–57. ISBN 9781450359078. Disponível em: <https://dl.acm.org/doi/10.1145/3229574.3229580>. Citado 4 vezes nas páginas 16, 17, 18 e 34.

SHAND, M.; BRYANT, S. RFC, *IP Fast Reroute Framework*. RFC Editor, 2010. RFC 5714, RFC Editor https://www.rfc-editor.org/info/rfc5714. (Request for Comments, 5714). Disponível em: <https://www.rfc-editor.org/info/rfc5714>. Citado 3 vezes nas páginas 31, 32 e 39.

SIVARAMAN, A.; KIM, C.; KRISHNAMOORTHY, R.; DIXIT, A.; BUDIU, M. Dc.p4: Programming the forwarding plane of a data-center switch. In: *Proceedings of the 1st ACM SIGCOMM SOSR*. New York, NY, USA: ACM, 2015. ISBN 9781450334518. Disponível em: <https://doi.org/10.1145/2774993.2775007>. Citado na página 30.

STERBENZ, J.; CETINKAYA, E.; HAMEED, M.; JABBAR, A.; QIAN, S.; ROHRER, J. Evaluation of network resilience, survivability, and disruption tolerance: Analysis, topology generation, simulation, and experimentation: Invited paper. *Springer Telecommunication Systems*, 12 2011. Citado na página 27.

VERDI, F.; L ROTHENBERG, C.; E PASQUINI, R.; MAGALHAES, M.; F. Novas Arquiteturas de Data Center para a cloud computing. *Em Minicursos do XXVIII SBRC*, v. 28, p. 103–152, 3 2010. Disponível em: <https://dcomp.ufscar.br/verdi/MCSBRC2010.pdf>. Citado na página 17.

WIJNANDS, I.; ROSEN, E.; DOLGANOW, A.; PRZYGIENDA, T.; ALDRIN, S. *Multicast Using Bit Index Explicit Replication (BIER)*. [S.l.], 2017. <https://www.rfc-editor.org/info/rfc8279>. Disponível em: <https://www.rfc-editor.org/info/rfc8279>. Citado na página 33.

YE, J.-L.; CHEN, C.; CHU, Y. A weighted ecmp load balancing scheme for data centers using p4 switches. In: . [S.l.: s.n.], 2018. p. 1–4. Citado 2 vezes nas páginas 27 e 33.

ZHANG, M.; LIU, B.; ZHANG, B. Load-balanced ip fast failure recovery. In: AKAR, N.; PIORO, M.; SKIANIS, C. (Ed.). *IP Operations and Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 53–65. ISBN 978-3-540-87357-0. Citado na página 18.

APÊNDICE A – Trabalho Publicado

- **Daniel B. de Lima**, Christian E. Rothenberg, Francisco Vogt, Alan Teixeira da Silva. "RESISTING: Um Novo Mecanismo de Fast-Reroute com Distribuição de Pacotes em Switches Programáveis P4". In SBRC WGRS – XXVIII Workshop de Gerência e Operação de Redes e Serviços, May 2023. – *Link para o repositório do artigo.*

APÊNDICE B – Cálculo VE: Um Falha

Neste apêndice, consolidamos os resultados dos cálculos do Valor Demandado (conforme a Equação 5.1) nos cenários de simulação que avaliaram a capacidade de resiliência para uma falha, usando os mecanismos PURR e RESISTING.

Top	**Failure**	**Links**	**Flows**	**PURR**	**RESISTING**
0	No Failure	**P1** **P2** **P3** **P4** **P5** **P6**	429 406 371 393 399 402	VE= 1	VE= 1
1	(P1)	-	-	VE=1	VE=1
2	(P2)	-	-	VE=1	VE=1
3	(P3)	-	-	VE=1	VE=1
4	(P4)	-	-	VE=1	VE=1
5	(P5)	-	-	VE=1	VE=1
6	(P6)	-	-	VE=1	VE=1

Tabela B.1 – Resultados dos cálculos VE para uma única falha.

APÊNDICE C – Cálculo VE: Duas Falhas

Neste apêndice, consolidamos os resultados dos cálculos do Valor Demandado (conforme a Equação 5.1) nos cenários de simulação que avaliaram a capacidade de resiliência para duas falhas simultâneas, usando os mecanismos PURR e RESISTING.

Top	**Failure**	**Links**	**Flows**	**PURR**	**RESISTING**
0	No Failure	**P1** **P2** **P3** **P4** **P5** **P6**	429 406 371 393 399 402	VE= 1	VE= 1
1	(P1,P2)	-	-	VE=0,9141	VE=1
2	(P1,P3)	-	-	VE=1	VE=1
3	(P1,P4)	-	-	VE=1	VE=1
4	(P1,P5)	-	-	VE=1	VE=1
5	(P1,P6)	-	-	VE=0,9012	VE=1
6	(P2,P3)	-	-	VE=0,9291	VE=1
7	(P2,P4)	-	-	VE=1	VE=1
8	(P2,P5)	-	-	VE=1	VE=1
9	(P2,P6)	-	-	VE=1	VE=1
10	(P3,P4)	-	-	VE=0,9320	VE=1
11	(P3,P5)	-	-	VE=1	VE=1
12	(P3,P6)	-	-	VE=1	VE=1
13	(P4,P5)	-	-	VE=0,9191	VE=1
14	(P4,P6)	-	-	VE=1	VE=1
15	(P5,P6)	-	-	VE=0,9041	VE=1

Tabela C.1 – Resultados dos cálculos VE para duas falhas.

APÊNDICE D – Cálculo VE: Três Falhas

Este apêndice consolida os resultados dos cálculos do Valor Demandado (conforme a Equação 5.1) nos cenários de simulação que avaliaram a capacidade de resiliência para três falhas simultâneas, usando os mecanismos PURR e RESISTING.

Top	**Failure**	**Links**	**Flows**	**PURR**	**RESISTING**
0	No Failure	**P1** **P2** **P3** **P4** **P5** **P6**	429 406 371 393 399 402	VE= 1	VE= 1
1	(P1,P2,P3)	-	-	VE=0,7504	VE=1
2	(P2,P3,P4)	-	-	VE=0,7629	VE=1
3	(P3,P4,P5)	-	-	VE=0,7645	VE=1
4	(P4,P5,P6)	-	-	VE=0,7404	VE=1
5	(P5,P6,P1)	-	-	VE=0,735	VE=1
6	(P6,P1,P2)	-	-	VE=0,7466	VE=1
7	(P1,P2,P4)	-	-	VE=0,9141	VE=1
8	(P1,P2,P5)	-	-	VE=0,9141	VE=1
9	(P1,P3,P4)	-	-	VE=0,9320	VE=1
10	(P1,P3,P5)	-	-	VE=1	VE=1
11	(P1,P3,P6)	-	-	VE=0,9012	VE=1
12	(P1,P4,P5)	-	-	VE=0,9191	VE=1
13	(P1,P4,P6)	-	-	VE=0,9012	VE=1
14	(P2,P3,P5)	-	-	VE=0,9291	VE=1
15	(P2,P3,P6)	-	-	VE=0,9291	VE=1
16	(P2,P4,P5)	-	-	VE=0,9191	VE=1
17	(P2,P4,P6)	-	-	VE=1	VE=1
18	(P2,P5,P6)	-	-	VE=0,9041	VE=1
19	(P3,P4,P6)	-	-	VE=0,9320	VE=1
20	(P3,P5,P6)	-	-	VE=0,9041	VE=1

Tabela D.1 – Resultados dos cálculos VE para três falhas.

APÊNDICE E – Cálculo VE Médio

Neste apêndice, apresentamos a consolidação dos resultados obtidos por meio dos cálculos do Valor Demandado Médio (conforme a Equação 5.2) nos cenários de simulação que investigaram a capacidade de resiliência para até três falhas, utilizando os mecanismos PURR e RESISTING.

Failure Scenarios	**FRR**	**Average VE**	**Average Loss**
1 Falha	PURR	1	0
	RESISTING	1	0
2 Falhas	PURR	0,96	0,4
	RESISTING	1	1
3 Falhas	PURR	0,86	0,14
	RESISTING	1	0

Tabela E.1 – Resultados dos cálculos VE Médio para os três cenários de falhas.

APÊNDICE F – Protótipo RESISTING BMv2

Neste apêndice, é apresentado o código-fonte P4 do protótipo RESISTING BMv2, cuja estrutura é composta por cinco distintos arquivos P4:

1. bmv2_6p_frr_v2_resisting.p4;
2. bmv2_6p_frr_v2_resisting_deparser_recirculation.p4;
3. bmv2_6p_frr_v2_resisting_egress_recirculation.p4;
4. bmv2_6p_frr_v2_resisting_ingress_recirculation.p4;
5. bmv2_6p_frr_v2_resisting_parser_recirculation.p4;

Listing F.1 – bmv2_6p_frr_v2_resisting.p4

```
1  #include <core.p4>
2  #include <v1model.p4>
3  #include "common/headers.p4"
4
5  /* PARSER */
6  #include "bmv2_6p_frr_v2_resisting_parser_recirculation.p4"
7  /* INGRESS PROCESSING */
8  #include "bmv2_6p_frr_v2_resisting_ingress_recirculation.p4"
9  /* EGRESS PROCESSING */
10 #include "bmv2_6p_frr_v2_resisting_egress_recirculation.p4"
11 /* DEPARSER */
12 #include "bmv2_6p_frr_v2_resisting_deparser_recirculation.p4"
13
14 /* CHECKSUM VERIFICATION */
15 control MyVerifyChecksum(inout headers hdr,
16  inout metadata meta) {
17   apply { }
18 }
19
20 /* CHECKSUM UPDATE */
```

```
21 control MyComputeChecksum(inout headers hdr,
22  inout metadata meta) {
23  apply {
24   update_checksum(
25   hdr.ipv4.isValid(),
26   {hdr.ipv4.version,
27    hdr.ipv4.ihl,
28    hdr.ipv4.diffserv,
29    hdr.ipv4.total_len,
30    hdr.ipv4.identification,
31    hdr.ipv4.flags,
32    hdr.ipv4.frag_offset,
33    hdr.ipv4.ttl,
34    hdr.ipv4.protocol,
35    hdr.ipv4.src_addr,
36    hdr.ipv4.dst_addr},
37    hdr.ipv4.hdr_checksum,
38    HashAlgorithm.csum16);
39  }
40 }
41
42 /* SWITCH */
43 V1Switch(
44  MyParser(),
45  MyVerifyChecksum(),
46  MyIngress(),
47  MyEgress(),
48  MyComputeChecksum(),
49  MyDeparser()
50 ) main;
```

Listing F.2 – bmv2_6p_frr_v2_resisting_deparser_recirculation.p4

```
1  control MyDeparser(
2          packet_out pkt,
3          in headers hdr) {
4  apply {
5             pkt.emit(hdr.ethernet);
6             pkt.emit(hdr.frr);
```

```
7             pkt.emit(hdr.ipv4);
8             pkt.emit(hdr.tcp);
9   }
10 }
```

Listing F.3 – bmv2_6p_frr_v2_resisting_egress_recirculation.p4

```
1  /* EGRESS PROCESSING */
2    control MyEgress(
3          inout headers hdr,
4          inout metadata eg_md,
5          inout standard_metadata_t eg_intr_md) {
6
7  register<bit<9>>(3) rg_frr_port_out_5_wr;
8
9     action set_add_copy_port_out_add_255(bit<32> idx){
10          hdr.frr.setValid();
11          hdr.frr.idx_5_or_cp_5_4 = 255;
12          rg_frr_port_out_5_wr.write(idx,255);
13     }
14
15    action set_add_copy_port_out_5_to_4(bit<32> idx){
16          hdr.frr.setValid();
17          bit<9> cp_port;
18          rg_frr_port_out_5_wr.read(cp_port,idx);
19          hdr.frr.idx_5_or_cp_5_4 = cp_port;
20          rg_frr_port_out_5_wr.write(idx,255);
21     }
22
23   table frr_port_out_5 {
24          key = {
25              hdr.frr.loop : exact;
26              hdr.frr.idx_port_down : exact;
27              hdr.frr.dst_id : exact;
28          }
29          actions = {
30              set_add_copy_port_out_5_to_4();
31              set_add_copy_port_out_add_255();
32          }
```

```
33          size = 64;
34      }
35
36  register<bit<9>>(3) rg_frr_port_out_4_wr;
37
38      action set_add_copy_port_out_4_to_3(bit<32> idx){
39          hdr.frr.setValid();
40          bit<9> cp_port;
41          rg_frr_port_out_4_wr.read(cp_port,idx);
42          hdr.frr.idx_4_or_cp_4_3 = cp_port;
43          rg_frr_port_out_4_wr.write(idx,hdr.frr.idx_5_or_cp_5_4);
44      }
45
46    table frr_port_out_4 {
47          key = {
48              hdr.frr.loop : exact;
49              hdr.frr.idx_port_down : exact;
50              hdr.frr.dst_id : exact;
51          }
52          actions = {
53              set_add_copy_port_out_4_to_3();
54          }
55          size = 64;
56      }
57
58  register<bit<9>>(3) rg_frr_port_out_3_wr;
59
60      action set_add_copy_port_out_3_to_2(bit<32> idx){
61          hdr.frr.setValid();
62          bit<9> cp_port;
63          rg_frr_port_out_3_wr.read(cp_port,idx);
64          hdr.frr.idx_3_or_cp_3_2 = cp_port;
65          rg_frr_port_out_3_wr.write(idx,hdr.frr.idx_4_or_cp_4_3);
66      }
67
68    table frr_port_out_3 {
69          key = {
70              hdr.frr.loop : exact;
```

```
71              hdr.frr.idx_port_down : exact;
72              hdr.frr.dst_id : exact;
73          }
74          actions = {
75              set_add_copy_port_out_3_to_2();
76          }
77          size = 64;
78      }
79
80  register<bit<9>>(3) rg_frr_port_out_2_wr;
81
82      action set_add_copy_port_out_2_to_1(bit<32> idx){
83          hdr.frr.setValid();
84          bit<9> cp_port;
85          rg_frr_port_out_2_wr.read(cp_port,idx);
86          hdr.frr.idx_2_or_cp_2_1 = cp_port;
87          rg_frr_port_out_2_wr.write(idx,hdr.frr.idx_3_or_cp_3_2);
88      }
89
90      action set_add_copy_port_out_2_to_1_and_255(bit<32> idx){
91          hdr.frr.setValid();
92          hdr.frr.idx_2_or_cp_2_1 = 255;
93          rg_frr_port_out_2_wr.write(idx,255);
94      }
95
96        action set_add_copy_port_out_2_to_1_range(bit<32> idx){
97          hdr.frr.setValid();
98          bit<9> cp_port;
99          rg_frr_port_out_2_wr.read(cp_port,idx);
100         hdr.frr.idx_2_or_cp_2_1 = cp_port;
101         rg_frr_port_out_2_wr.write(idx,255);
102     }
103
104   table frr_port_out_2 {
105         key = {
106             hdr.frr.loop : exact;
107             hdr.frr.idx_port_down : exact;
108             hdr.frr.dst_id : exact;
```

```
109         }
110         actions = {
111             set_add_copy_port_out_2_to_1();
112             set_add_copy_port_out_2_to_1_and_255();
113             set_add_copy_port_out_2_to_1_range();
114         }
115         size = 64;
116     }
117
118 register<bit<9>>(3) rg_frr_port_out_1_wr;
119
120     action set_add_copy_port_out_1_to_0(bit<32> idx){
121         hdr.frr.setValid();
122         bit<9> cp_port;
123         rg_frr_port_out_1_wr.read(cp_port,idx);
124         hdr.frr.idx_1_or_cp_1_0 = cp_port;
125         rg_frr_port_out_1_wr.write(idx,hdr.frr.idx_2_or_cp_2_1);
126     }
127
128   table frr_port_out_1 {
129         key = {
130             hdr.frr.loop : exact;
131             hdr.frr.idx_port_down : exact;
132             hdr.frr.dst_id : exact;
133         }
134         actions = {
135             set_add_copy_port_out_1_to_0();
136         }
137         size = 64;
138     }
139
140 register<bit<9>>(3) rg_frr_port_out_0_wr;
141
142     action set_add_port_out_1_to_0(bit<32> idx){
143         hdr.frr.setValid();
144         hdr.frr.idx_0 = hdr.frr.idx_1_or_cp_1_0;
145         rg_frr_port_out_0_wr.write(idx,hdr.frr.idx_1_or_cp_1_0);
146     }
```

```
147
148   table frr_port_out_0 {
149         key = {
150             hdr.frr.loop : exact;
151             hdr.frr.idx_port_down : exact;
152             hdr.frr.dst_id : exact;
153         }
154         actions = {
155             set_add_port_out_1_to_0();
156         }
157         size = 64;
158     }
159
160 register<bit<24>>(3) rg_count_port_up_wr;
161
162   action set_count_port_up(bit<32> idx){
163                   eg_md.port_count_idx = idx;
164                   rg_count_port_up_wr.read(eg_md.port_count,eg_md
                          .port_count_idx);
165    }
166
167   table count_port_up {
168         key = {
169             hdr.frr.update_bit : exact;
170             hdr.frr.first_failure : exact;
171             hdr.frr.idx_port_down : range;
172             hdr.frr.dst_id : exact;
173         }
174         actions = {
175             set_count_port_up();
176         }
177         size = 64;
178     }
179
180  apply {
181
182          count_port_up.apply();
183
```

```
184             if(hdr.frr.update_bit == 1 && hdr.frr.first_failure ==
                    1 && eg_md.port_count > 0){
185                     hdr.frr.setValid();
186                     hdr.frr.loop = 4w1;
187                     frr_port_out_5.apply();
188                     frr_port_out_4.apply();
189                     frr_port_out_3.apply();
190                     frr_port_out_2.apply();
191                     frr_port_out_1.apply();
192                     frr_port_out_0.apply();
193                     hdr.frr.setValid();
                            rg_count_port_up_wr.write(eg_md.
                            port_count_idx,eg_md.port_count - 1);
194             }
195     }
196 }
```

Listing F.4 – bmv2_6p_frr_v2_resisting_ingress_recirculation.p4

```
1  /* INGRESS PROCESSING */
2
3   control MyIngress(
4         inout headers hdr,
5         inout metadata ig_md,
6         inout standard_metadata_t ig_intr_md) {
7
8     action drop_act() {
9        mark_to_drop(ig_intr_md);
10    }
11
12    action set_nhop_local(mac_addr_t dstAddr, PortId_t port) {
13          hdr.ethernet.setValid();
14          hdr.ethernet.ether_type = 0x0800;
15          hdr.ethernet.dst_addr = dstAddr;
16          hdr.ipv4.setValid();
17          hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
18          ig_intr_md.egress_spec = port;
19          ig_md.dst_id = 0;
20    }
```

```
21
22      action set_nhop_network(dst_id_t dst_id) {
23          ig_md.dst_id = dst_id;
24      }
25
26      table ipv4_lpm {
27          key = {
28              hdr.ethernet.ether_type: exact;
29              ig_intr_md.ingress_port: exact;
30              hdr.ipv4.dst_addr: lpm;
31          }
32          actions = {
33              set_nhop_network;
34              set_nhop_local;
35              drop_act;
36          }
37          //const default_action = drop_act;
38          size = 64;
39      }
40
41 //Version 1
42 /*
43  action set_ecmp_path_selector(bit<9> idx){
44     ig_md.ecmp_path_selector = idx;
45  }
46  // Act_Sel(HashAlg.crc32, max_group_size, num_groups)
47   action_selector(HashAlgorithm.crc32, 32w12, 32w256)
       hash_path_selector;
48
49  table ecmp_hash_selector {
50          key = {
51              ig_md.dst_id: exact;
52              hdr.ethernet.ether_type: exact;
53              hdr.ipv4.src_addr: selector;
54              hdr.ipv4.dst_addr: selector;
55              hdr.ipv4.protocol: selector;
56              hdr.tcp.src_port: selector;
57              hdr.tcp.dst_port: selector;
```

```
58          }
59          actions = {
60              drop_act;
61              set_ecmp_path_selector;
62           }
63          implementation = hash_path_selector;
64      }
65  */
66   register<bit<16>>(3) rg_base_link_wr;
67   register<bit<32>>(3) rg_max_link_wr;
68
69   action set_add_ecmp_hash_loop_bit_1(bit<32> idx){
70           bit<32> max_link;
71           rg_max_link_wr.read(max_link,idx);
72           rg_max_link_wr.write(idx,max_link - 1);
73          }
74
75   table update_ecmp_hash {
76          key = {
77              hdr.frr.idx_port_down : range;
78              hdr.frr.loop : exact;
79              hdr.frr.dst_id : exact;
80              hdr.frr.first_failure : exact;
81          }
82          actions = {
83              set_add_ecmp_hash_loop_bit_1;
84          }
85
86          size = 64;
87      }
88
89   action set_ecmp_hash(bit<32> idx) {
90           bit<16> base_link;
91           bit<32> max_link;
92
93           rg_base_link_wr.read(base_link,idx);
94           rg_max_link_wr.read(max_link,idx);
95           hash(ig_md.ecmp_path_selector, HashAlgorithm.crc32,
```

```
96                      base_link,
97                      {hdr.ipv4.src_addr,
98                       hdr.ipv4.dst_addr,
99                       hdr.ipv4.protocol,
100                      hdr.tcp.src_port,
101                      hdr.tcp.dst_port},
102                      max_link);
103         }
104
105  table ecmp_hash {
106          key = {
107              ig_md.dst_id: exact;
108              hdr.ethernet.ether_type: exact;
109          }
110          actions = {
111              drop_act;
112              set_ecmp_hash;
113           }
114          size = 64;
115      }
116
117 register<bit<9>>(3) rg_default_path_wr;
118
119   action set_add_default_path_loop_bit_1(bit<32> idx) {
120               rg_default_path_wr.write(idx,hdr.frr.idx_0);
121     }
122
123  action set_add_default_path_loop_bit_1_ecmp_idx_range_1(bit<32>
        idx) {
124               rg_default_path_wr.write(idx,hdr.frr.idx_0);
125     }
126
127  action set_add_default_path_loop_bit_1_ecmp_idx_range_2_idx_3(
       bit<32> idx) {
128               rg_default_path_wr.write(idx,hdr.frr.
                     idx_4_or_cp_4_3);
129     }
130
```

```
131   action set_add_default_path_loop_bit_1_ecmp_idx_range_2_idx_0(
          bit<32> idx) {
132               rg_default_path_wr.write(idx,hdr.frr.idx_0);
133    }
134
135   action set_add_default_path_loop_bit_1_ecmp_idx_range_3(bit
          <32> idx) {
136               rg_default_path_wr.write(idx,hdr.frr.idx_0);
137    }
138
139  table update_default_path {
140         key = {
141             hdr.frr.idx_port_down : exact;
142             hdr.frr.loop : exact;
143             hdr.frr.dst_id : exact;
144         }
145         actions = {
146             set_add_default_path_loop_bit_1;
147             set_add_default_path_loop_bit_1_ecmp_idx_range_1;
148             set_add_default_path_loop_bit_1_ecmp_idx_range_2_idx_0
                    ;
149             set_add_default_path_loop_bit_1_ecmp_idx_range_2_idx_3
                    ;
150             set_add_default_path_loop_bit_1_ecmp_idx_range_3;
151         }
152         size = 64;
153     }
154
155    action set_default_path_loop_bit_0(bit<32> idx){
156          bit<9> port_out;
157          rg_default_path_wr.read(port_out,idx);
158          ig_intr_md.egress_spec = port_out;
159          ig_md.ecmp_path_selector = 0;
160     }
161    action set_default_path_loop_bit_0_ecmp_idx_range_1(bit<32>
          idx){
162             bit<9> port_out;
163             rg_default_path_wr.read(port_out,idx);
```

```
164             ig_intr_md.egress_spec = port_out;
165             ig_md.ecmp_path_selector = 0;
166         }
167         action set_default_path_loop_bit_0_ecmp_idx_range_2(bit<32>
                idx){
168             bit<9> port_out;
169             rg_default_path_wr.read(port_out,idx);
170             ig_intr_md.egress_spec = port_out;
171             ig_md.ecmp_path_selector = 3;
172         }
173         action set_default_path_loop_bit_0_ecmp_idx_range_3(bit<32>
                idx){
174             bit<9> port_out;
175             rg_default_path_wr.read(port_out,idx);
176             ig_intr_md.egress_spec = port_out;
177             ig_md.ecmp_path_selector = 0;
178         }
179
180       table default_path {
181             key = {
182                 ig_intr_md.egress_spec : exact;
183                 ig_md.ecmp_path_selector : range;
184                 hdr.frr.loop : exact;
185                 ig_md.dst_id : exact;
186             }
187             actions = {
188                 set_default_path_loop_bit_0;
189                 set_default_path_loop_bit_0_ecmp_idx_range_1;
190                 set_default_path_loop_bit_0_ecmp_idx_range_2;
191                 set_default_path_loop_bit_0_ecmp_idx_range_3;
192                 drop_act;
193             }
194             size = 64;
195            // const default_action = drop_act;
196         }
197
198 register<bit<9>>(3) rg_port_out_5_wr;
199
```

```
200     action set_add_port_out_5_loop_bit_1(bit<32> idx) {
201             rg_port_out_5_wr.write(idx,255);
202       }
203
204   table update_port_out_5 {
205         key = {
206             hdr.frr.idx_port_down : exact;
207             hdr.frr.loop : exact;
208             hdr.frr.dst_id : exact;
209         }
210         actions = {
211             set_add_port_out_5_loop_bit_1;
212         }
213         size = 64;
214       }
215
216     action set_fowarding_port_out_5_loop_bit_0(bit<32> idx) {
217           hdr.ethernet.setValid();
218           hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
219           hdr.ethernet.ether_type = 0x800;
220           bit<9> port_out;
221           rg_port_out_5_wr.read(port_out,idx);
222           ig_intr_md.egress_spec = port_out;
223        }
224
225   table fowarding_tag_5 {
226        actions = {
227               set_fowarding_port_out_5_loop_bit_0();
228           }
229           key = {
230               ig_md.dst_id : exact;
231               ig_md.ecmp_path_selector : exact;
232               hdr.frr.loop : exact;
233            }
234           size = 64;
235       //    default_action = drop_act;
236      }
237
```

```
238 register<bit<9>>(3) rg_port_out_4_wr;
239
240     action set_add_port_out_4_loop_bit_1(bit<32> idx) {
241             rg_port_out_4_wr.write(idx,hdr.frr.idx_5_or_cp_5_4);
242       }
243
244   table update_port_out_4 {
245         key = {
246             hdr.frr.idx_port_down : exact;
247             hdr.frr.loop : exact;
248             hdr.frr.dst_id : exact;
249         }
250         actions = {
251             set_add_port_out_4_loop_bit_1;
252         }
253         size = 64;
254       }
255
256     action set_fowarding_port_out_4_loop_bit_0(bit<32> idx) {
257           hdr.ethernet.setValid();
258           hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
259           hdr.ethernet.ether_type = 0x800;
260           bit<9> port_out;
261           rg_port_out_4_wr.read(port_out,idx);
262           ig_intr_md.egress_spec = port_out;
263        }
264
265   table fowarding_tag_4 {
266        actions = {
267              set_fowarding_port_out_4_loop_bit_0();
268         }
269          key = {
270              ig_md.dst_id : exact;
271              ig_md.ecmp_path_selector : exact;
272              hdr.frr.loop : exact;
273           }
274          size = 64;
275       //    default_action = drop_act;
```

```
276     }
277
278 register<bit<9>>(3) rg_port_out_3_wr;
279
280     action set_add_port_out_3_loop_bit_1(bit<32> idx) {
281           rg_port_out_3_wr.write(idx,hdr.frr.idx_4_or_cp_4_3);
282       }
283
284   table update_port_out_3 {
285         key = {
286             hdr.frr.idx_port_down : exact;
287             hdr.frr.loop : exact;
288             hdr.frr.dst_id : exact;
289         }
290         actions = {
291             set_add_port_out_3_loop_bit_1;
292         }
293         size = 64;
294       }
295
296     action set_fowarding_port_out_3_loop_bit_0(bit<32> idx) {
297           hdr.ethernet.setValid();
298           hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
299           hdr.ethernet.ether_type = 0x800;
300           bit<9> port_out;
301           rg_port_out_3_wr.read(port_out,idx);
302           ig_intr_md.egress_spec = port_out;
303        }
304
305   table fowarding_tag_3 {
306        actions = {
307              set_fowarding_port_out_3_loop_bit_0();
308         }
309          key = {
310             ig_md.dst_id : exact;
311             ig_md.ecmp_path_selector : exact;
312             hdr.frr.loop : exact;
313           }
```

```
314             size = 64;
315         //    default_action = drop_act;
316       }
317
318 register<bit<9>>(3) rg_port_out_2_wr;
319
320     action set_add_port_out_2_loop_bit_1(bit<32> idx) {
321             rg_port_out_2_wr.write(idx,hdr.frr.idx_3_or_cp_3_2);
322       }
323
324       action set_add_port_out_2_loop_bit_1_add_255(bit<32> idx) {
325             rg_port_out_2_wr.write(idx,255);
326       }
327
328     table update_port_out_2 {
329           key = {
330               hdr.frr.idx_port_down : exact;
331               hdr.frr.loop : exact;
332               hdr.frr.dst_id : exact;
333           }
334           actions = {
335               set_add_port_out_2_loop_bit_1;
336               set_add_port_out_2_loop_bit_1_add_255;
337           }
338           size = 64;
339       }
340
341     action set_fowarding_port_out_2_loop_bit_0(bit<32> idx) {
342           hdr.ethernet.setValid();
343           hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
344           hdr.ethernet.ether_type = 0x800;
345           bit<9> port_out;
346           rg_port_out_2_wr.read(port_out,idx);
347           ig_intr_md.egress_spec = port_out;
348         }
349
350     table fowarding_tag_2 {
351           actions = {
```

```
352             set_fowarding_port_out_2_loop_bit_0();
353         }
354          key = {
355             ig_md.dst_id : exact;
356             ig_md.ecmp_path_selector : exact;
357             hdr.frr.loop : exact;
358           }
359          size = 64;
360       //    default_action = drop_act;
361     }
362
363 register<bit<9>>(3) rg_port_out_1_wr;
364
365    action set_add_port_out_1_loop_bit_1(bit<32> idx) {
366          rg_port_out_1_wr.write(idx,hdr.frr.idx_2_or_cp_2_1);
367      }
368
369   table update_port_out_1 {
370         key = {
371             hdr.frr.idx_port_down : exact;
372             hdr.frr.loop : exact;
373             hdr.frr.dst_id : exact;
374         }
375
376         actions = {
377             set_add_port_out_1_loop_bit_1;
378         }
379
380         size = 64;
381      }
382
383    action set_fowarding_port_out_1_loop_bit_0(bit<32> idx) {
384          hdr.ethernet.setValid();
385          hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
386          hdr.ethernet.ether_type = 0x800;
387          bit<9> port_out;
388          rg_port_out_1_wr.read(port_out,idx);
389          ig_intr_md.egress_spec = port_out;
```

```
390         }
391
392     table fowarding_tag_1 {
393           actions = {
394                 set_fowarding_port_out_1_loop_bit_0();
395           }
396            key = {
397                 ig_md.dst_id : exact;
398                 ig_md.ecmp_path_selector : exact;
399                 hdr.frr.loop : exact;
400             }
401            size = 64;
402         //    default_action = drop_act;
403       }
404
405 register<bit<9>>(3) rg_port_out_0_wr;
406
407      action set_add_port_out_0_loop_bit_1(bit<32> idx) {
408            rg_port_out_0_wr.write(idx,hdr.frr.idx_0);
409        }
410
411     table update_port_out_0 {
412           key = {
413                 hdr.frr.idx_port_down : exact;
414                 hdr.frr.loop : exact;
415                 hdr.frr.dst_id : exact;
416            }
417
418           actions = {
419                 set_add_port_out_0_loop_bit_1;
420           }
421
422           size = 64;
423        }
424
425      action set_fowarding_port_out_0_loop_bit_0(bit<32> idx) {
426              hdr.ethernet.setValid();
427              hdr.ethernet.dst_addr = (bit<48>)ig_md.dst_id;
```

```
428             hdr.ethernet.ether_type = 0x800;
429             bit<9> port_out;
430             rg_port_out_0_wr.read(port_out,idx);
431             ig_intr_md.egress_spec = port_out;
432         }
433
434     table fowarding_tag_0 {
435         actions = {
436                 set_fowarding_port_out_0_loop_bit_0();
437             }
438             key = {
439                 ig_md.dst_id : exact;
440                 ig_md.ecmp_path_selector : exact;
441                 hdr.frr.loop : exact;
442             }
443             size = 64;
444         //    default_action = drop_act;
445       }
446
447      action set_port_status_down(bit<16> bit_down){
448             ig_md.port_status_up_down = (bit<1>)bit_down;
449       }
450
451     table port_status {
452             key = {
453                 ig_intr_md.egress_spec : exact;
454                 hdr.frr.loop : exact;
455                 hdr.ethernet.ether_type: exact;
456       }
457
458             actions = {
459                 set_port_status_down;
460                 drop_act;
461             }
462
463             size = 64;
464             const default_action = drop_act;
465       }
```

```
466
467 register<bit<24>>(3) dr_rg_first_failure_count_wr;
468
469     action set_frr_recirculation_first_failure_loop_bit_0(bit<32>
            idx){
470             hdr.ethernet.setValid();
471             hdr.ethernet.ether_type = 0x255;
472             hdr.frr.setValid();
473 // adiciona a posicao do index/posicao que armazena a porta down
        de saida
474             hdr.frr.idx_port_down =  ig_md.ecmp_path_selector;
475             hdr.frr.update_bit = 1;
476             hdr.frr.ether_type_eth = 0x255;
477             bit<24> count;
478             dr_rg_first_failure_count_wr.read(count,idx);
479             dr_rg_first_failure_count_wr.write(idx,count + 1);
480             hdr.frr.first_failure = count + 1;
481             hdr.frr.dst_id = ig_md.dst_id;
482             recirculate<headers>(hdr);
483  }
484
485     action set_frr_recirculation_first_failure_loop_bit_1(bit<32>
            idx){
486             dr_rg_first_failure_count_wr.write(idx,0);
487             hdr.ethernet.setValid();
488             hdr.ethernet.ether_type = 0x800;
489             hdr.frr.setInvalid();
490             hdr.frr.update_bit = 0;
491             hdr.frr.first_failure = 0;
492             recirculate<headers>(hdr);
493   }
494
495     table frr_recirculation {
496          key = {
497              ig_md.port_status_up_down : exact;
498              hdr.frr.loop : exact;
499              ig_md.ecmp_path_selector : range;
500              hdr.frr.idx_port_down : range;
```

```
501             ig_md.dst_id : exact;
502             hdr.frr.dst_id : exact;
503
504     }
505
506         actions = {
507             set_frr_recirculation_first_failure_loop_bit_1;
508             set_frr_recirculation_first_failure_loop_bit_0;
509            // drop_act;
510         }
511
512         size = 6;
513        // const default_action = drop_act;
514
515     }
516
517     action set_frr_no_recovery_loop_bit_0(){
518          hdr.frr.setInvalid();
519          hdr.ethernet.setValid();
520          hdr.ethernet.ether_type = 0x800;
521    }
522
523     table frr_no_recovery {
524         key = {
525             ig_md.port_status_up_down : exact;
526             hdr.frr.loop : exact;
527
528     }
529         actions = {
530             set_frr_no_recovery_loop_bit_0;
531        }
532         size = 64;
533     }
534
535   apply {
536
537
538         ipv4_lpm.apply();
```

```
539 //V1
540 //          ecmp_hash_selector.apply();
541           update_ecmp_hash.apply();
542           ecmp_hash.apply();
543           update_port_out_5.apply();
544           update_port_out_4.apply();
545           update_port_out_3.apply();
546           update_port_out_2.apply();
547           update_port_out_1.apply();
548           update_port_out_0.apply();
549 //V1
550 //          update_default_path.apply();
551           fowarding_tag_5.apply();
552           fowarding_tag_4.apply();
553           fowarding_tag_3.apply();
554           fowarding_tag_2.apply();
555           fowarding_tag_1.apply();
556           fowarding_tag_0.apply();
557 //V1
558 //          default_path.apply();
559
560           port_status.apply();
561
562         @atomic{
563           frr_recirculation.apply();
564           }
565
566     }
567
568 }
```

Listing F.5 – bmv2_6p_frr_v2_resisting_parser_recirculation.p4

```
1
2 parser MyParser(
3         packet_in pkt,
4         out headers hdr,
5         inout metadata ig_md,
6         inout standard_metadata_t ig_intr_md) {
```

```
7
8       state start {
9            transition parse_ethernet;
10      }
11
12      state parse_ethernet {
13          pkt.extract(hdr.ethernet);
14          ig_md.dst_id = (bit<16>)hdr.ethernet.dst_addr;
15          transition select (hdr.ethernet.ether_type) {
16               0x800 : parse_ipv4;
17               TYPE_FRR: parse_frr;
18               default : accept;
19           }
20       }
21
22       state parse_frr {
23          pkt.extract(hdr.frr);
24          hdr.ethernet.ether_type = hdr.frr.ether_type_eth;
25          transition select (hdr.frr.update_bit) {
26                0: parse_ipv4;
27                default : accept;
28          }
29       }
30
31       state parse_ipv4 {
32          pkt.extract(hdr.ipv4);
33          //hdr.ethernet.ether_type = 0x800;
34          transition select (hdr.ipv4.protocol) {
35              0x006 : parse_tcp;
36
37              default : accept;
38          }
39       }
40
41        state parse_tcp {
42           pkt.extract(hdr.tcp);
43           transition accept;
44       //   default : accept;
```

```
45              }
46  }
```

APÊNDICE G – Repositório Online

Este apêndice apresenta o repositório online que inclui os códigos-fonte P4 dos protótipos RESISTING BMv2 e Tofino, bem como as implementações do PURR, P4-LFA e da primeira versão do FRR-ECMP BMv2 (que foi precursor do RESISTING). Além disso, estão disponíveis as regras e scripts que sustentaram as realizações deste trabalho:

1. RESISTING BMv2 – <https://github.com/danielbl1000/P4-RESISTING-bmv2>;
2. RESISTING Tofino – <https://github.com/danielbl1000/P4-RESISTING-tofino>;
3. PURR – <https://github.com/danielbl1000/P4-simple_switch_purr>;
4. LFA – <https://github.com/danielbl1000/P4-LFA>;
5. FRR-ECMPv1 – <https://github.com/danielbl1000/FRR-ECMP-First-Version>;