Gustavo Venancio Luz

InFaRR: Um algoritmo para reroteamento rápido em planos de dados programáveis

Sorocaba, SP

2022

Gustavo Venancio Luz

InFaRR: Um algoritmo para reroteamento rápido em planos de dados programáveis

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Engenharia de Software e Sistemas de Computação.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. Fábio Luciano Verdi

Sorocaba, SP

2022

Luz, Gustavo Venancio
InFaRR: Um algoritmo para reroteamento rápido em planos de dados programáveis/ Gustavo Venancio Luz. – 2022.
100 f. : 30 cm.

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCC-So.
Orientador: Prof. Dr. Fábio Luciano Verdi
Banca examinadora: Prof. Dr. Fábio Luciano Verdi, Prof. Dr. Magnos Martinello, Profa. Dra. Yeda Venturini
Bibliografia

1. Reroteamento Rápido. 2. Redes programáveis. 3. P4. I. Luz, Gustavo Venancio. II. Universidade Federal de São Carlos. III. InFaRR: Um algoritmo para reroteamento rápido em planos de dados programáveis

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Gustavo Venancio Luz, realizada em 20/09/2022.

Comissão Julgadora:

Prof. Dr. Fabio Luciano Verdi (UFSCar)

Prof. Dr. Magnos Martinello (UFES)

Profa. Dra. Yeda Regina Venturini (UFSCar)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Dedico este trabalho a Deus.

Agradecimentos

Agradeço a todos meus familiares e amigos que, de forma direta ou indiretamente, me ajudaram no decorrer destes 4 anos de mestrado. Ser pai, ser esposo, profissional e ser estudante não seria possível sem a ajuda de vocês. Seria injusto citar nomes e correr o risco de esquecer alguém.

Agradeço a compreensão dos meus filhos Sofia, Pedro e Heitor, pelo tempo ausente tomado pelos estudos. Todo este esforço foi pensando em um futuro melhor para vocês. Espero que compreendam e tenham orgulho da minha conquista.

Agradeço à minha amada esposa Cíntia, pelo carinho, apoio e incentivo durante todo curso.

Agradeço a todos os professores pelos ensinamentos e pelo exemplo profissional, pela ajuda e pela paciência com a qual guiaram o meu aprendizado.

Agradeço a todo time LERIS, em especial, ao Prof. Dr. Fábio Verdi, meu orientador que sabiamente apontou a direção certa para minha pesquisa, me ajudou a manter o foco, a priorizar e organizar os trabalhos.

"Não é o mais forte que sobrevive,
nem o mais inteligente,
mas o que melhor se adapta às mudanças."
(Charles Darwin)

Resumo

O InFaRR (*In-network Fast ReRouting*) é um algoritmo para reroteamento rápido em planos de dados programáveis. Implementado em P4, o InFaRR é livre de cabeçalhos adicionais de gerenciamento (*overheads*) e de pacotes de gerenciamento do estado da rede (*hearbeats*). O InFaRR apresenta quatro características essenciais, não encontradas, de maneira conjunta, em outros mecanismos de recuperação: Prevenção de *loop*, *Pushback*, Reconhecimento e Restauração, e Retorno à rota principal. Os testes nas topologias *Standard Fat-Tree* e *AB Fat-Tree* com falhas em diferentes cenários apresentaram resultados positivos quando comparados aos algoritmos do estado da arte da literatura. Nos cenários em que os outros algoritmos conseguiram se recuperar, o InFaRR apresentou menor variação de tempo no atraso dos pacotes quando os mecanismo de *Pushback*, Prevenção de *loop* e Reconhecimento e Restauração foram utilizados, proporcionando menor número de saltos ao contornar a falha. Nos cenários com múltiplas falhas o InFaRR realizou com sucesso o reroteamento, quando os outros algoritmos, em alguns casos, entraram em *loop*. O mecanismo único para retorno à rota principal inovou diante da possibilidade de verificação de enlaces remotos no plano de dados, possibilitando o retorno à rota principal sem intervenção do plano de controle.

Palavras-chaves: Reroteamento Rápido. Redes Programáveis. P4.

Abstract

InFaRR (In-network Fast ReRouting) is an algorithm for fast rerouting in programmable data planes. Implemented in P4, InFaRR is free of additional management headers (*overheads*) and network state management packets (*hearbeats*). InFaRR has four essential features not jointly found in other recovery mechanisms: Loop prevention, Pushback, Recognition and Restoration and Return to the main route. Tests in a Sandard Fat-Tree and AB Fat-Tree topology with failures in different scenarios showed positive results when compared to state-of-the-art algorithms in the literature. In scenarios in which the other algorithms were able to recover, InFaRR presented less time variation in packet delay when the *Pushback*, *loop* Prevention and Recognition and Restoration mechanisms was used, resulting in fewer hops when bypassing the failure. In scenarios with multiple failures, InFaRR successfully rerouted where the others algorithms in some cases looped. The unique mechanism for returning to the main route innovated in view of the possibility of verifying remote links in the data plane, making it possible to return to the main route without intervention from the control plane.

Key-words: Fast Rerounting. Software Define Network. P4.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
CE2E	Clone Egress to Egress
CI2E	Clone Ingress to Egress
BFS	Breath-First Search
BGP	Border Gateway Protocol
DDC	Data-Driven Connectivity
DFS	Depth-First Search
FCT	Flow-Completion Time
ICMP	Internet Control Message Protocol
INFARR	In-network Fast Rerouting
IP	Internet Protocol
IPDV	IP Delay Variation (Variação de atraso de Pacotes IP)
IPER	IP packet Error Ratio (Taxa de Erro de pacote)
IPLR	IP Packet Loss Ratio (Taxa de perda de Pacotes IP)
IPTD	IP Packet Transfer Delay (Atraso máximo de Transferência de Pacotes IP)
ITU-T	Telecommunication Standardization Sector
FPGA	Field- Programmable Gate Array
FSM	Finite State Machine
LAN	Local Area Network
LPM	Longest-Prefix Match
MAC	Media Access Control
ms	Milissegundo

MTTF	Mean Time to Failure
MTTR	Mean Time to Repair
NIC	Network Interface Card
OSI	Open System Interconnection
OSPF	Open Shortest Part First
P4	Programming Protocol-independent Packet Processors
PISA	Protocol-Independent Switch Architecture
POD	Point of Delivery
QOS	Quality of Service (Qualidade de Serviço)
RTT	Round Trip Time
TCP	Transmission Control Protocol
TOR	Top of Rack (Topo de rack)
TTL	Time-to-live
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
WAN	Wide Area Network

Lista de Algoritmos

Sumário

1 Introdução

Os *datacenters* cada vez mais centralizam o processamento das aplicações que requerem respostas em tempo real, com alto grau de interatividade ou que são sensíveis à variação de latência. Tais aplicações estão cada vez mais presentes nos dias de hoje, presentes como por exemplo nos serviços de telemedicina, veículos autônomos, entre outros. Para atender estas características das aplicações atuais, muitas delas são disponibilizadas na nuvem ou em *datacenters*, pois possuem mecanismos de tolerância à falhas desde sua concepção. Exemplos destes mecanismos são disponibilização de equipamentos redundantes, conexões distintas para proteção compõem a parte física da solução, enquanto que a parte lógica é composta por algoritmos que permitem a escolha do plano de roteamento principal, priorização de pacotes e mecanismos de recuperação para o uso em caso de falhas.

Nos diferentes mecanismos de recuperação em redes de computadores, aplicados a cenários em que o roteamento não pode encaminhar pacotes diante a falha no enlace principal, destacam-se soluções de reroteamento rápido que são caracterizadas pela capacidade local do comutador de optar pelo uso da política de recuperação que foi previamente configurada. O reroteamento rápido, portanto, é uma ação tomada sem a participação do plano de controle e sem a dependência da sinalização de outros elementos externos, como, anúncios de novas rotas, *heartbeat*, *keep alives* ou do estouro do limite de tempo de uma sessão do protocolo de roteamento dinâmico (KIRANMAI et al., 2014).

As soluções de reroteamento sustentadas pelo plano de controle centralizado em Redes Definidas por Software (do inglês, *Software Defined Networking* - SDN) apresentam um mecanismo de recuperação mais lento quando comparado ao reroteamento rápido no plano de dados (KIRANMAI et al., 2014). O plano de controle centralizado tem um atraso para iniciar a recuperação, pois é preciso detectar a falha em um equipamento remoto por meio de um processo de *pooling*, ou receber a notificação da falha, o que é contrário à proposta de reroteamento rápido (SGAMBELLURI et al., 2013). No entanto, os mecanismos de recuperação executados no plano de dados provêm o reroteamento rápido, melhorando a agilidade e velocidade de recuperação durante períodos de falhas, pois não há a necessidade de consultar o plano de controle ou a dependência de algum tipo de sinalização de agentes externos (ALI et al., 2020).

A resiliência em redes de computadores é o resultado do esforço de planejamento e estratégias para combinar os recursos disponíveis para atingir o melhor nível de serviço possível. A escolha da topologia apropriada, equipamentos com características de tolerâncias a falhas e enlaces com redundância, não é o único, mas está entre os principais elementos da arquitetura de rede que devem ser considerados desde da sua construção, para possibilitar

a alta disponibilidade.

Neste contexto, o InFaRR (*In-network Fast Rerouting*) se beneficia do plano de dados programável e propõe um algoritmo de reroteamento rápido capaz de contornar até três[1] falhas em redes *Fat-Tree*. O InFaRR foi implementado na linguagem P4 (*Programming Protocol-Independent Packet Processors*) e comparado com as principais soluções encontradas na literatura atualmente (BOSSHART et al., 2014). O InFaRR possui quatro características essenciais não encontradas, de maneira conjunta, em outros mecanismos: Prevenção de *Loops* na rede, *Pushback*, Mecanismo de Reconhecimento e Restauração e Retorno à Rota Principal (KAMISINSKI, 2018; SUBRAMANIAN et al., 2021).

O controle de *loop* proposto no InFaRR age antes do tradicional TTL (*Time-To-Live*), evitando que pacotes fiquem circulando na rede. O Mecanismo de *Pushback*, também encontrado em (LIU et al., 2013a), devolve os pacotes ao comutador anterior em caso de não haver rotas alternativas a partir do ponto da falha. O Mecanismo de Reconhecimento e Restauração habilita a descoberta de quais rotas estão com falhas, evitando assim, o envio de tráfego através delas. Finalmente, o Retorno à Rota Principal é um mecanismo adicionado ao InFaRR capaz de detectar se a rota original se recuperou da falha e, então retomar o envio dos fluxos afetados através de sua rota primária.

Avaliou-se o InFaRR utilizando as topologias de rede *datacenter Standard Fat-Tree* e *AB Fat-Tree* com k=4 em ambiente emulado no Mininet. Diferentes cenários de falhas foram exercitados comparando o algoritmo proposto com trabalhos correlatos e minimamente adaptados para o funcionamento em P4 (CHIESA et al., 2019; SEDAR et al., 2018). A comparação dos algoritmos foi realizada a partir da avaliação do sucesso da recuperação diante das falhas, método de configuração, escopo de recuperação, domínio de recuperação, ocorrência de perda de pacotes durante o processo de recuperação, tempo de transmissão dos pacotes e número de saltos na rede após a recuperação da falha.

Portanto, as principais contribuições deste trabalho são: 1) projeto e implementação de um algoritmo de reroteamento rápido para redes programáveis, com a capacidade de recuperação em até três falhas em redes *Fat-tree*; 2) execução de uma prova de conceito em ambiente virtual sob a ótica das topologia *Standard Fat-Tree* e *AB Fat-Tree* para análise, interpretação e comparação com o estado da arte e; 3) disponibilização do *dataset* coletado durante a prova de conceito para fins de replicação do estudo e futuras comparações.

Este trabalho está organizado conforme descrito a seguir. O Capítulo 2 apresenta a fundamentação teórica associada a este trabalho. Os principais trabalhos relacionados que representam o estado da arte sobre o tema são apresentados na Capítulo 3. O Capítulo 4 apresenta as características do algoritmo InFaRR e detalhes de funcionamento. No Capítulo 5, apresentamos os principais resultados quantitativos em relação aos diferentes

[1] Limitação de três falhas será descrita na Sessão 4.3

cenários avaliados. O Capítulo final discute as conclusões e as propostas para trabalhos futuros.

2 Fundamentação Teórica

A resiliência pode ser definida como a capacidade de um elemento se recuperar ou superar com facilidade os problemas que aparecem (SMITH et al., 2011). A normalização dos termos utilizados para o estudo da Qualidade de Serviço (QoS) pode ser observada em diversas publicações da *Telecommunication standardization Sector (ITU-T)* (SERVICE; OPERATION, 2008).

Dividimos este capítulo em seis seções que fundamentam a teoria desta dissertação, sendo elas: Resiliência em redes computadores, Qualidade de serviços em redes de computadores, métodos de recuperação, processo de recuperação, arquitetura P4 e topologia implementada *Fat-Tree*.

2.1 Resiliência em Redes de Computadores

O estudo da resiliência se concentra em dois grandes grupos destacados na Figura 1, um voltado para disciplinas associadas à tolerância, e outro grupo de disciplinas relacionadas à confiabilidade (STERBENZ et al., 2010).

Figura 1 – Disciplinas de estudo sobre Resiliência.

2.1.1 Desafios da tolerância

As redes que precisam de elevados índices de disponibilidade devem ser planejadas de forma a contornarem possíveis rupturas em quaisquer dos seus elementos, e devem continuar a prestar um nível aceitável de funcionamento mesmo em condições de falha (*Disruption tolerance*). Um bom projeto de disponibilidade de redes deve prever meios de contornar falhas ou redundância em todos seus elementos, sejam eles hardware, software, enlaces, cabeamento, fornecimento de energia, segurança (contra *hackers*) e operacional (pessoas) (MAUTHE et al., 2016).

O planejamento de tolerância a desastres pode abordar questões complexas de sobrevivência (*survivability*), de modo que os altos níveis de disponibilidade são obtidos com o uso de ambientes geograficamente distribuídos, para prover a continuidade do serviço em caso de falha em um dos sites. Os fundamentos de sobrevivência têm como objetivo

contornar situações adversas como um incêndio, inundações ou uma greve que impeçam o acesso ao local (Mas MacHuca et al., 2016).

Para serem capazes de tolerar inesperados volumes de tráfegos (*Traffic tolerance*) as redes devem ser capazes de tratar com prioridade o tráfego legítimo e prioritário. O aumento exponencial do uso da rede pode ser proveniente de um ataque de negação de serviço (*Denial of Service*, DoS), em que a rede deve ser capaz de descartar todo tráfego inválido. Quando o ambiente apresenta uma sobrecarga, reflexo da alta utilização, os fluxos de dados devem ser classificados e as tratativas de descarte ou encaminhamento realizadas respeitando o nível de serviço de funcionamento (STERBENZ et al., 2010).

2.1.2 Desafios da confiabilidade

A confiabilidade (*trustworthiness*) em resiliência de redes de computadores visa avaliar a garantia de que a mesma está trabalhando dentro do desempenho esperado. Divide-se em três subitens: dependabilidade (*dependability*), segurança (*security*) e desempenho (*performability*) (AVIZIENIS et al., 2004; STERBENZ et al., 2010).

A dependabilidade trata os atributos mensuráveis da resiliência, sendo a disponibilidade (*availability*) a sua principal métrica de cálculo. A disponibilidade se refere à probabilidade da rede estar em funcionamento dentro de um desempenho mínimo e está diretamente relacionada às métricas: tempo médio para reparo (*mean time to repair*, MTTR) e do tempo médio para falha (*mean time to failure*, MTTF) (CHOLDA et al., 2009; CHIESA et al., 2020). O algoritmo InFaRR tem como principal objetivo reduzir o tempo de convergência da rede, que pode ser considerado como o MTTR.

As qualidades de segurança fortalecem a confiabilidade da rede. São exemplos de características de segurança: a autenticidade, credibilidade, auditabilidade, confidencialidade, integridade, e não-repudiabilidade. Podemos exemplificar que um acesso indevido a equipamento por um hacker pode resultar em indisponibilidade da rede (STERBENZ et al., 2010; CHIESA et al., 2020).

As falhas não se referem somente à incapacidade total de funcionamento. O desempenho é também um importante indicador de comprometimento de funcionamento. Situações em que o funcionamento da rede está abaixo do esperado, por exemplo uma alta latência gerada graças ‘s’ elevada utilização, ocasionará uma degradação de qualidade e confiabilidade da rede (AVIZIENIS et al., 2004; MAUTHE et al., 2016). Os critérios de desempenho serão descritos na próxima seção.

2.2 Qualidade de Serviços em redes de computadores

As redes que precisam de elevados índices de disponibilidade devem ser planejadas de forma a contornarem possíveis rupturas em quaisquer dos seus elementos. Um projeto de tolerância a falhas deve apresentar um nível aceitável de funcionamento mesmo em condições adversas. Cada tipo de fluxo suportado pela rede pode possuir níveis de qualidade de serviços específicos que devem ser atendidos para seu ideal funcionamento (MAUTHE et al., 2016). Nas redes programáveis existe a possibilidade de construção de políticas de recuperação baseadas em cada tipo de fluxo ou clientes, garantindo uma maior flexibilidade para diferentes níveis de serviço presentes na rede.

O Setor de Normatização das Telecomunicações (*Telecommunication Standardization Sector*, ITU-T) definiu parâmetros, relacionados ao desempenho da rede IP que quantificam os níveis requeridos de Qualidade de Serviços (QoS) para as diferentes classes de serviços existentes, sendo eles: Atraso máximo de Transferência de Pacotes IP (*IP Packet Transfer Delay*, IPTD), Variação de atraso de Pacotes IP (*IP Delay Variation*, IPDV), Taxa de perda de Pacotes IP (*IP Packet Loss Ratio*, IPLR) e Taxa de Erro de pacote (*IP packet Error Ratio*, IPER) (CHODOREK, 2002; ITU-T, 2002). Os requisitos para cada tipo de tráfego podem ser vistos na Tabela 1 e serão referência para análise e interpretação dos resultados dos algoritmos avaliados nesta dissertação.

Serviço	**Descrição das Aplicações**	**IPTD**	**IPDV**	**IPLR**	**IPER**
Classe 0	Tempo real, sensíveis a jitter, alta interatividade	100 ms	50 ms	1×10^{-3}	1×10^{-4}
Classe 1	Tempo real, sensíveis a jitter, interativas	400 ms	50 ms	1×10^{-3}	1×10^{-4}
Classe 2	Transação de dados, alta interatividade	100 ms	indefinido	1×10^{-3}	1×10^{-4}
Classe 3	Transação de dados, interativas	400 ms	indefinido	1×10^{-3}	1×10^{-4}
Classe 4	Tolerante a baixa perdas de pacotes	1000 ms	indefinido	1×10^{-3}	1×10^{-4}
Classe 5	Aplicações comuns de redes IP	indefinido	indefinido	indefinido	indefinido

Tabela 1 – Classes de serviços definidos pela ITU-T.

As primeiras quatro classes de serviços (0 até 3) descritas na Tabela 1 são sensíveis ao tempo de transmissão (IPTD) e requerem caminhos de transmissão fim-a-fim inferior a meio segundo (400 ms), pois estão relacionadas a aplicações que exigem comunicação em tempo real ou interativas. Além disso, as classes 0 e 1 exigem uma variação de latência (IPDV) inferior a 50 ms para um bom funcionamento (STANKIEWICZ; CHOLDA; JAJSZCZYK, 2011). Portanto, um mecanismo de recuperação com o intuito de atender a esses requisitos descritos, deve prover um reroteamento dentro do limite de IPDV, e o novo caminho de contingência não deve ultrapassar o limite do IPTD requerido.

Os protocolos de roteamento dinâmico estão entre as ferramentas utilizadas para prover tolerância a falhas e gerar confiabilidade da rede, de forma que, quando um enlace apresenta falhas, a rede pode convergir para outro caminho. Protocolos tradicionais e de ampla utilização, como o Border Gateway Protocol (BGP) ou Open Shortest Part First (OSPF), podem levar dezenas de segundos para convergir, enquanto que os requisitos das

classes de serviços de tempo real e interação necessitam que essa convergência ocorra significativamente em menos tempo ((CHIESA et al., 2020)), portanto não atendem os níveis estabelecidos de Qualidade de Serviço para redes de computadores.

Tradicionalmente, nas redes programáveis, quando ocorre a falha em enlace, o comutador remoto necessita detectar a falha e comunicar o plano de controle para reconfiguração dos planos de encaminhamento dos fluxos afetados. Este processo de convergência pode levar cerca de 100ms, caso ainda exista conectividade entre o comutador remoto e o plano de controle ((SHARMA et al., 2013)). Esta dissertação apresentará uma alternativa de reroteamento no plano de dados, de forma a atender os requisitos de qualidade de serviço para classes de serviços sensíveis a IPTD e IPDV e perda de pacotes.

2.3 Métodos de Recuperação

Os algoritmos são a parte lógica da solução que possibilita a convergência da rede, sem a programação lógica e regras de convergência, a redundância existente pela topologia física não se tornaria ativa. As diferentes formas de implementação dos algoritmos requerem uma padronização quanto aos métodos de recuperação para que seja possível interpretá-los e compará-los.

Os métodos de recuperação podem ser classificados por diversos critérios (CHOLDA et al., 2007). Quatro desses critérios possuem extrema relevância. O primeiro está associado ao método de configuração do caminho *backup*, que pode ser pré-configurado ou reativo. O método pré-configurado nas redes SDNs possibilita uma recuperação mais rápida, pois a estratégia de recuperação já esta pré-definida para ser utilizada quando necessário no plano de dados, como uma tabela de roteamento secundária. No método reativo, um processo de recuperação é iniciado no plano de controle logo quando uma falha é encontrada. Todo o processamento de reconstrução da tabela de roteamento é feito pelo plano de controle e distribuído para o plano de dados. O processo de recuperação termina quando a nova tabela de roteamento se torna ativa no plano de dados.

O segundo critério está associado ao escopo da recuperação que pode ser global, local ou de segmento. Na política global é oferecida uma proteção a todos os elementos do caminho (fim-a-fim). A política local assegura que exista uma forma de contornar pontualmente o enlace com problema. Neste cenário, o tráfego sofrerá um ajuste de encaminhamento somente a partir do ponto onde a falha foi encontrada. Por fim, a política de recuperação por segmento oferece proteção a segmentos específicos envolvendo comutadores e enlaces, e proverá contorno pontual ao trecho com problemas (CHOLDA et al., 2007).

Na Figura 2, o caminho principal entre o comutador 1 e 5 é representado por meio dos enlaces em azul, passando pelos equipamentos 2, 3 e 4. A política com escopo

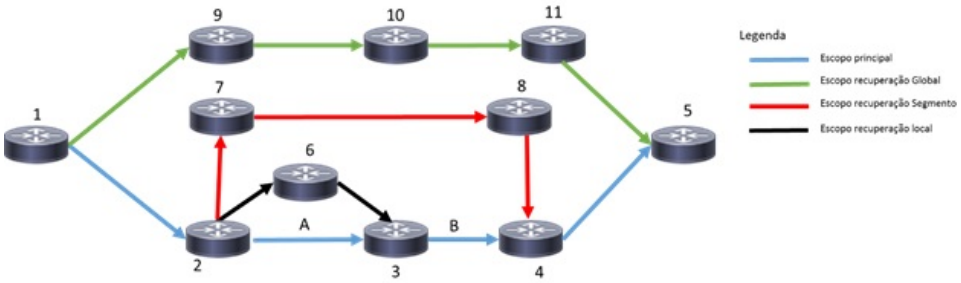


Figura 2 – Escopo de Recuperação.

de recuperação Global, representada pelos enlaces em verde, provê proteção a falhas de qualquer comutador e enlaces que compõem o caminho do escopo principal. O escopo de recuperação por segmento, representado pelos enlaces em vermelho, e os comutadores 7 e 8 provêm proteção a falhas para o comutador 3 e enlaces A e B. Finalmente, o escopo de proteção local, provê proteção somente para o enlace A por meio dos enlaces em preto passando pelo comutador 6.

Os recursos utilizados para prover a recuperação compõem o terceiro critério e podem ser classificados como dedicados ou compartilhados. Em arquiteturas que utilizam recursos dedicados, também conhecidas por abordagem 1+1, duplicam-se todos os elementos que requerem método de recuperação. A abordagem compartilhada por sua vez, possibilita que vários elementos utilizem o mesmo recurso de backup, pressupondo, estatisticamente, que não haverá mais de um elemento com falha requerendo o uso deste recurso de backup simultaneamente. Entretanto, caso isso aconteça, poderá ocorrer sobrecarga do recurso e o método de recuperação planejado não funcionará adequadamente (STERBENZ et al., 2010).

No cenário apresentado na Figura 3, pode-se observar que o comutador 1 possui uma rota principal até o comutador 6 passando pelo comutador 2 (enlaces em azul), enquanto que o comutador 3 possui como rota principal a passagem do comutador 4 (enlaces em verde). Os enlaces em preto são recursos de proteção dedicados, de modo que o enlace entre o comutador 1 e 5 é dedicado para prover proteção ao comutador 1, enquanto que o enlace entre o comutador 3 e 5 protege exclusivamente o comutador 3. O comutador 5 (vermelho) e o enlace entre o comutador 5 e 6 (vermelho) são recursos de proteção compartilhados, pois tanto podem servir para proteção ao comutador 1 quanto para o comutador 3, e devem ser devidamente dimensionados para suportar tráfegos dos comutadores 1 e 3, se necessário.

O último critério trata as características relacionadas ao domínio de operação e recuperação, sendo este item diretamente associado ao tamanho da rede e suas fronteiras. O atributo domínio de operação pode ser observado nas redes SDNs pelo número de planos de controle existentes, sendo que, quanto mais domínios de operações mais complexo será o processo de sincronismo entre os diferentes planos de controle (RAK, 2015). Se

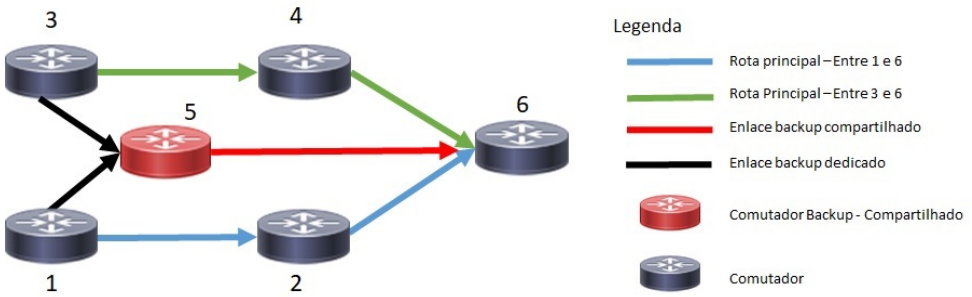


Figura 3 – Recursos de Recuperação.

observarmos as redes *Standard Fat-Tree* e *AB Fat-Tree*, objeto deste estudo, podemos, por exemplo, interpretar um *POD*[1] (*Point of Delivery*) como um domínio de recuperação, mapeando se o processo de reroteamento ocorre exclusivamente no mesmo *POD* ou se existe dependência de comutadores de outros *PODs*.

O InFaRR adota um método de recuperação pré-configurado que o classifica como um algoritmo de reroteamento rápido. O escopo de recuperação adotado no InFaRR é por segmento. O domínio de operação será interno ao *POD* trazendo independência ao processo de reroteamento, evitando que, no momento da recuperação da falha, pacotes trafeguem por equipamentos desnecessários.

2.4 Processo de Recuperação

Inicialmente, é importante definir o conceito básico referente aos tipos de tabela de roteamento existentes, em que a "tabela de roteamento principal" refere-se ao processo primário utilizado pelo comutador para definir a porta de saída para encaminhamento dos pacotes, utilizando como chave para pesquisa o endereço IP destino. Enquanto que a "tabela de roteamento secundário", ou *backup*, será utilizada somente se a porta sugerida pela tabela de roteamento principal estiver com problemas.

A Figura 4 descreve as etapas existentes relacionadas ao processo de detecção e recuperação de falhas. Na etapa inicial $E0$, têm-se o funcionamento da rede com tráfego enviado por meio de um plano de encaminhamento ótimo, determinado pelo plano de controle. A detecção da falha, que acontece na etapa $E1$, define o momento em que a porta foi para o estado de *down*, dando início ao processo de recuperação na etapa $E2$. Na etapa $E3$, a rede passa a funcionar por um caminho alternativo até que o mecanismo de controle possa restabelecer a conectividade, o que acontece na etapa $E4$. O gatilho para mudança de estado para *up* da etapa anterior, inicia o processo de atualização das tabelas de roteamento, etapa $E5$, retornando ao plano de encaminhamento inicial em $E6$.

O mecanismo de controle que ocorre nas etapas $E1$ e $E4$ não são escopo deste

[1] O conceito POD faz parte da topologia de rede FAT-TREE e será descrito na Sessão 2.6

trabalho, associadas aos mecanismos de sinalização no meio físico, perda de pacotes ou outros fatores que levam à penalização da porta com estado *down*; ou admitem uma porta com estado *up*. Todavia, a mudança do estado da porta para *down* ou *up* é o gatilho para início do algoritmo de recuperação. Para fins de avaliação nesta dissertação, o estado da porta foi imposto arbitrariamente na simulação de falhas e respectivo processo de recuperação.

O Mecanismo de Prevenção de *Loop* proposto pelo algoritmo InFaRR implementa funcionalidades de detecção de falha e gatilho para o início do reroteamento através da detecção do *loop* na etapa $E1$, enquanto que o Mecanismo de Retorno à Rota Principal promove o retorno a rota principal descrito na etapa $E4$

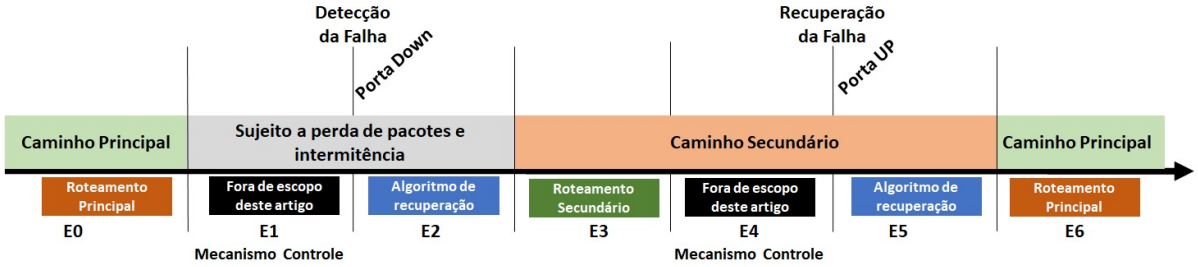


Figura 4 – Etapas do processo de detecção e recuperação de falhas.

O InFaRR tem entre seus objetivos reduzir o tempo de recuperação, que é a soma da duração das etapas $E1$ e $E2$. Neste intervalo, de $E1$ a $E2$, conjectura-se que pode ocorrer perda de pacotes, portanto quanto menor for este tempo, mais rápido será o processo de convergência (CHOLDA et al., 2007). Visto que na etapa $E1$, ocorre a detecção do enlace com status $DOWN$[2], observa-se que é importante avaliar a duração da etapa $E2$, tendo em vista que ela pode sofrer influência direta do algoritmo de convergência utilizado. Realizar as medições relacionadas ao desempenho da rede durante o funcionamento normal no momento $E0$, e compará-las com o funcionamento no momento de contingência $E3$, possibilitará avaliar a eficácia do algoritmo de contingência. Conjectura-se que mesmo no processo de convergência, as medições de desempenho permaneçam dentro dos limites apresentados na Seção 2.2.

As condições de rede no momento de contingência serão equivalentes ou piores se comparadas com o estado de funcionamento normal, tomando como premissa que o plano de controle ofereceu a rota ótima para o funcionamento normal na etapa $E0$. Portanto, as condições de rede serão equivalentes caso a estrutura de contingência possua as mesmas condições de utilização e latência de rede, encontradas em topologias com proteção dedicada (1+1). Em cenários de proteção com recursos compartilhados, o plano de encaminhamento contingente estará sujeito a utilização de trechos com maior ocupação, perda de pacotes, maior latência ou caminhos com maior número de saltos que impactarão diretamente os

[2] Não é escopo deste trabalho

parâmetros de desempenho do fluxo contingenciado. Para efeito de estudo, consideraremos que as conexões que farão parte do processo de redundância serão dedicadas e apresentarão as mesmas condições de rede (latência e utilização), de forma que não será necessário considerar nenhum processo de priorização ou descarte de pacotes neste estudo.

2.5 Linguagem P4

A linguagem Programming Protocol-independent Packet Processors (P4) possibilita uma programação em alto nível aplicada ao plano de dados de equipamentos que compõem redes definidas por software (SDN). Possui como característica principal o processamento de pacotes em *pipelines* baseado em uma arquitetura PISA (Protocol-Independent Switch Architecture) (BOSSHART et al., 2014).

Diferentes tipos de equipamentos e fabricantes usufruem da capacidade do compilador P4 para abstrair características especificas do hardware tornando a linguagem de programação P4 viável para hardwares ASICs (*Application-Specific Integrated Circuits*), FPGAs (*Field- Programmable Gate Arrays*), Network Interface Cards (NICs) e software switchs (HAUSER et al., 2021).

O administrador de rede possui assim flexibilidade para programar e explorar os recursos computacionais dos seus equipamentos conforme a necessidade de sua topologia e tipo de tráfego existente. Toda estrutura de funcionamento de um programa P4 se baseia nos seguintes componentes e modelo de funcionamento:

- *Cabeçalhos* (*Headers*): a definição da estrutura dos cabeçalhos habilitam os tipos de pacotes que serão processados pelo comutador. Para o processamento de pacotes IPv4 o programa deverá possuir no mínimo a definição de cabeçalhos *Ethernet* e IPv4. Caso o comutador tenha que processar pacotes IPv6, este cabeçalho também deverá ser definido. A facilidade de manipulação dos cabeçalhos permite ainda, ao administrador da rede definir novos cabeçalhos caso necessário (HAUSER et al., 2021).

  A Figura 5 demonstra o código em P4 utilizado para definir os cabeçalhos Ethernet, IPv4, TCP, UDP e ICMP que são usadas pelo InFaRR. A estrutura de tupla permite a definição dos campos e tamanho em bits que formam cada cabeçalho. Como exemplo, na estrutura Ethernet temos a definição de três campos sendo o primeiro associado ao MAC destino com 48 bits (t_dstAddr), o segundo campo associado ao MAC origem, também com 48 bits (t_srcAddr) e um campo para o tipo Ethernet com 16 bits (etherType).

- *Analisador (Parser)*: possui a responsabilidade de interpretar a estrutura do pacote recebido diante dos cabeçalhos definidos. É nesta etapa que o analisador distingue

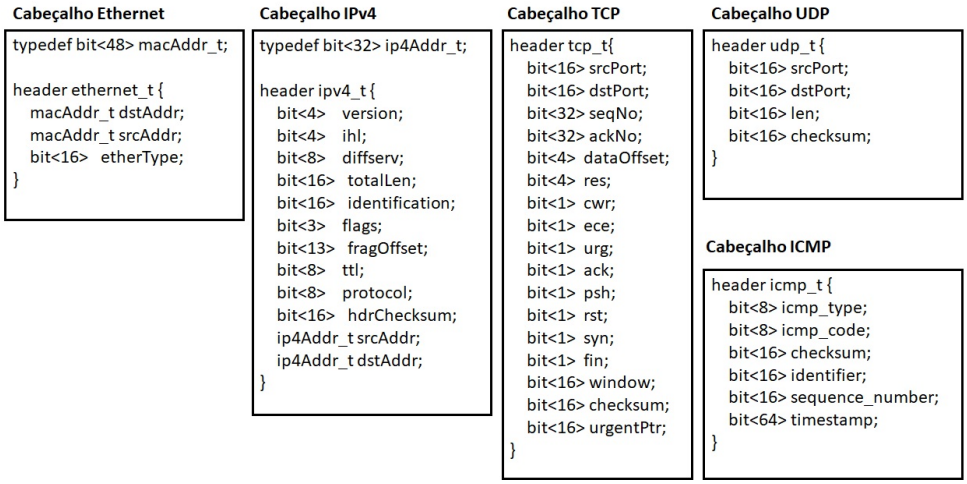


Figura 5 – Modelo de cabeçalho P4.

qual tipo de pacote está processando e poderá dar sequência ao processamento conforme a respectiva tabela *Match-Action*. Desta forma, os pacotes, TCP seguirão para a respectiva tabela *Match-Action*, enquanto que os pacotes UDP ou ICMP serão encaminhados para a respectiva tabela *Match-Action* (GIBB et al., 2013).

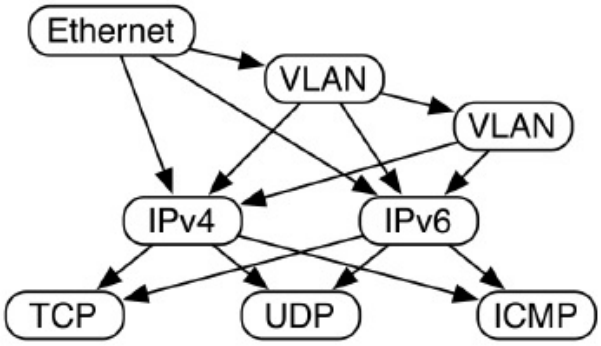


Figura 6 – Exemplo de Analisador P4 de cabeçalhos.

A Figura 6 representa o modelo de máquina de estado infinito responsável por analisar a estrutura de cabeçalho Ethernet com IPv4 ou IPv6, com possíveis marcações de *tag* de *VLAN*. Observa-se que este analisador não está apto para tratar pacotes *ARP* ou *MPLS* pois esses tipos de pacotes possuem estruturas de cabeçalhos diferentes das que ele foi programado para tratar.

- *Tabelas Match-Action*: definem todas as ações que serão tomadas para o processamento do pacote. As ações correspondentes, quando uma consulta realiza *Match* na tabela, possibilitam ao plano de dados manipular os cabeçalhos de forma que o pacote possa ser encaminhado (KFOURY; CRICHIGNO; BOU-HARB, 2021).

  As principais ações possíveis de serem implementadas são:

  - definir um campo: atualizar o valor do campo de um cabeçalho.

  - copiar um campo: copiar o valor de um campa para outro.

- adicionar um cabeçalho: Incluir um cabeçalho na estrutura do pacote.
- remover um cabeçalho: Excluir um cabeçalho da estrutura do pacote.
- ajustar: Incrementar ou decrementar o valor de um campo.

A flexibilidade na construção das ações possibilita uma programação de redes que atenda aos requisitos de reroteamento rápido, assim as ações de recuperação são inseridas diretamente no plano de dados. O administrador de rede pode, por exemplo, definir como processo de recuperação a consulta a uma tabela de roteamento secundário sem a necessidade de interferência do plano de controle (BOSSHART et al., 2014).

A linguagem P4 permite a programação dos cabeçalhos, do analisador e das tabelas *match-action*. O modelo representado na Figura 7 (BOSSHART et al., 2014) exemplifica o fluxo percorrido pelo pacote em cada *pipeline*. O *pipeline* de entrada (*ingress*) interpreta os cabeçalhos (*Parser*) e os encaminha para o processamento da tabela *match-action*, em que modificações no cabeçalho do pacote podem ser realizadas. A tabela *match-action* do *pipeline* de entrada é responsável pela definição da porta de saída pelo qual o pacote será encaminhado. A última etapa é o processo de remontagem dos pacotes denominados *defaser* (HAUSER et al., 2021). Sequencialmente, o pacote é encaminhado para o *pipeline* de saída em que novas alterações, se necessárias, poderão acontecer no cabeçalho do pacote, mas não é possível alterar a porta de saída do pacote neste *pipeline*. No *pipeline* de saída, as funções de *broadcast*, *multicast* ou espelhamento de porta necessitam recorrer à função CE2E (*Clone Egress to Egress*, e quando necessária a alteração da porta de saída, o pacote deverá ser recirculado para o *pipeline* de entrada para ser novamente processado (GOMEZ et al., 2022).

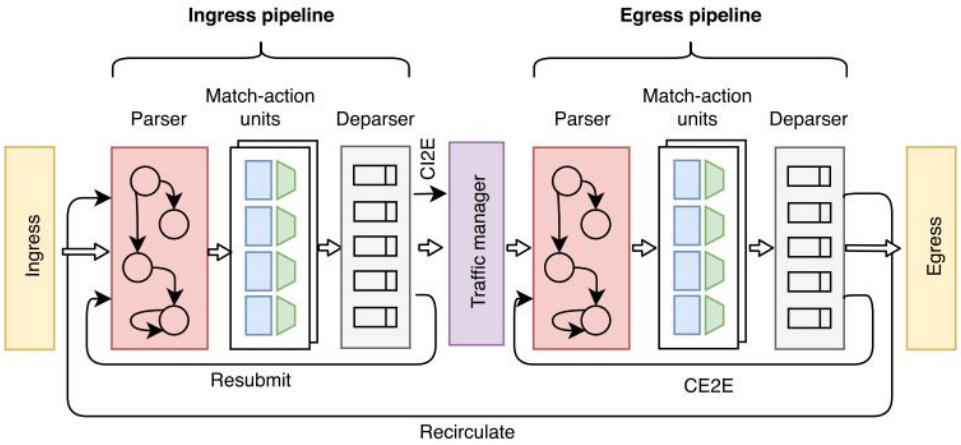


Figura 7 – Modelo abstrato de encaminhamento PISA.

As principais e comuns atribuições de um comutador P4 são:

- Envio de pacote *unicast*, operação tradicional de encaminhamento de pacote;

- Descarte (*drop*) de pacote;
- Envio de pacotes *broadcast*, *multicast* ou espelhamento de porta;
- Reenvio do pacote (*Resubmit*) após processado pelo *pipeline* de entrada para o início do *pipeline* de entrada;
- Recirculação pacote (*Recirculate*) após processado pelo *pipeline* de saída para o início do *pipeline* de entrada;
- CI2E (*Clone Ingress to Egress*) - cria um clone de um pacote do *pipeline* de entrada para o início do *pipeline* de saída;
- CE2E (*Clone Egress to Egress*) - cria um clone de um pacote do *pipeline* de saída para o o início do *pipeline* de saída;

Na linguagem de programação P4, as ações são realizadas somente quando um pacote é tratado pela tabela *match-action*, portanto não é possível agendar ações no plano de dados para que ocorram esporadicamente, ou definir ações que devam ser realizadas em casos de *time-out* para um determinado fluxo de dados. Durante a ociosidade da rede o plano de dados fica impossibilitado de realizar qualquer ação (BOSSHART et al., 2014). Para contornar a ausência de estruturas de repetições, como *while...loop* ou *for...next*, o administrador de redes necessita implementar soluções de reenvio ou de recirculação de pacotes (conforme trabalho de (QU et al., 2019)) ou utilizar a função clone de pacotes para manter um pacote ativo em processamento pelo comutador (MENTH et al., 2019).

2.6 Topologias de redes *Fat-Tree*

As redes *Fat-Tree*, normalmente utilizadas em *datacenters*, possuem uma estrutura de rede hierárquica caracterizada por sua construção em três camadas, com redundância de equipamentos e enlaces. As camadas são denominadas Núcleo, Agregação e Topo de Rack (em inglês, *CORE, Aggregation e Top of Rack* - ToR) as quais possuem funções distintas. O número de portas presentes nos equipamentos dimensiona o tamanho da rede e sua capacidade de tolerância a falhas. Sendo k o número de portas presentes em um equipamento do Núcleo, k será o número de conexões existentes para cada um dos diferentes k *POD*. O conjunto de equipamentos de Agregação, de topo de rede e *hosts* interconectados são denominados *PODs* (LEISERSON, 1985; GILL; JAIN; NAGAPPAN, 2011).

Os equipamentos de Agregação se posicionam na camada intermediária da rede e são responsáveis por discernir sobre o encaminhamento de pacotes internos de interesse de um mesmo *POD* ou por processar o encaminhamento de pacotes com destinos externos

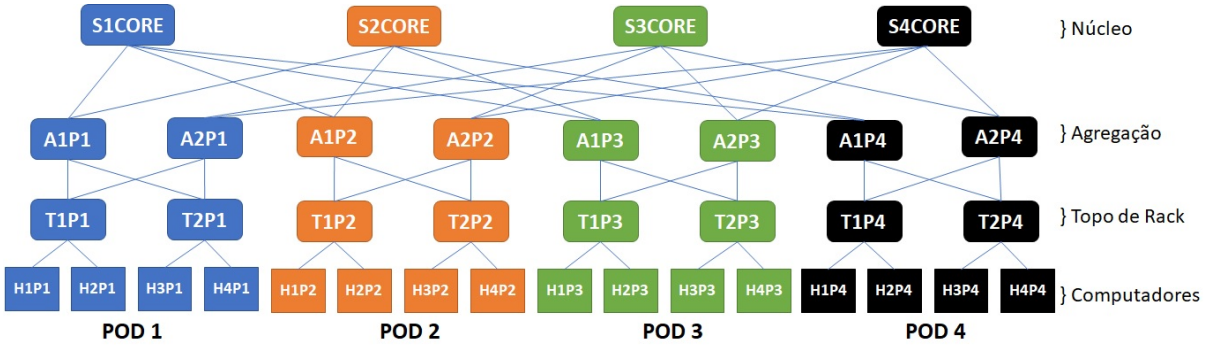


Figura 8 – Topologia de Rede *Standard Fat-Tree*.

para equipamentos Núcleo de forma que possam alcançar outros *PODs*. Destaca-se que um equipamento de agregação possui conexão direta à $k/2$ Cores e conexão direta à $k/2$ TOR distintos, todavia não possui conexão direta ao outro equipamento de agregação do mesmo *POD*; no caso de haver necessidade de comunicação entre eles, faz-se necessário um roteamento por meio do TOR do mesmo *POD* (camada inferior) ou equipamentos de núcleo (camada superior), conforme Figura 8.

Os equipamentos de Topo de Rede se caracterizam por possuírem $k/2$ conexões a equipamentos de agregação distintos, e proporcionam conexão direta aos $k/2$ *hosts* com conexão direta por meio deste equipamento. Por concepção as redes *Fat-Tree* não proporcionam conexão direta entre os hosts de um mesmo *POD* ligados a equipamentos TOR diferentes, sendo obrigatório que esta conexão entre eles realize-se por meio da camada intermediária de agregação.

As redes *Fat-Tree* possuem tolerância a falhas de $k^2 - 1$ nos enlaces entre os *PODs* e os equipamentos do Núcleo. Os equipamentos posicionados no Núcleo possuem como função prover roteamento entre os *PODs*. Cada dispositivo do Núcleo possui um enlace a pelo menos um equipamento de Agregação de um *POD* diferente, conforme Figura 8.

A variação de k não reflete no aumento do número de camadas na topologia de rede, reflete apenas no número de conexões, equipamentos e *PODs*.

A nomenclatura utilizada, nesta dissertação para nomear os equipamentos do *POD* consiste na utilização da primeira letra para nomear a função (H para host, T para topo de rede e A para agregação), um número de identificação, a letra P para representação do *POD* e, finalmente, o número do *POD* que o equipamento faz parte. Assim, ao observar a nomenclatura A2P4, deve se interpretar que a nomeação se refere ao equipamento de Agregação 2 do *POD*4.

Nas topologias *Standard Fat-Tree* os equipamentos do Núcleo *S1CORE* e *S2CORE* possuem enlaces diretos com os equipamentos de Agregação *A1P1, A1P2, A1P3 e A1P4*, enquanto os equipamentos *S3CORE* e *S4CORE* possuem enlaces diretos com *A2P1*, *A2P2*, *A2P3* e *A2P4*, conforme Figura 8. Esse padrão de interconexão entre núcleo e agregação

limita as opções de reroteamento entre os *PODs* tornando-o incapaz de contornar falhas duplas em um mesmo equipamento de agregação (LEISERSON, 1985).

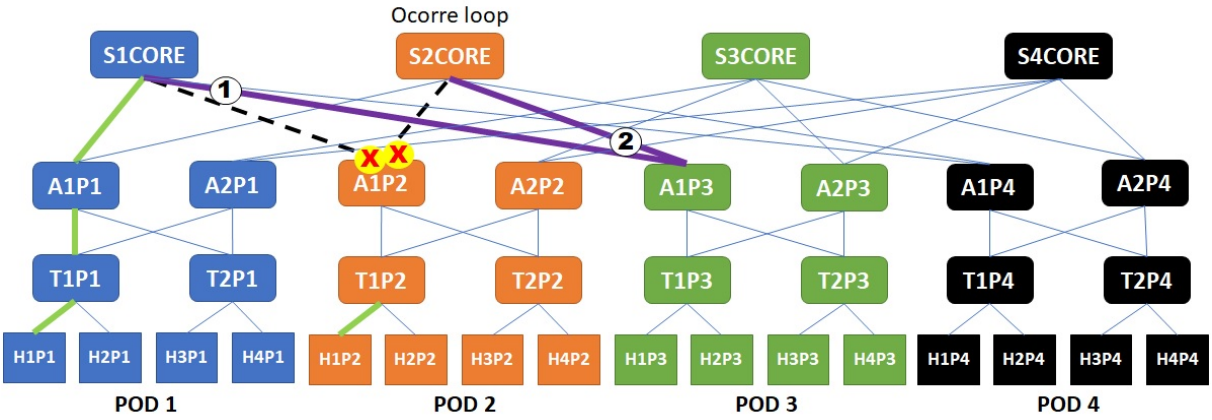


Figura 9 – Standard Fat Tree com falhas no A1P2.

A Figura 9 representa a comunicação entre um *host* H1P1 do *POD*1 e um *host* H2P2 do *POD*2. Tradicionalmente em uma rede com roteamento por IP destino, o equipamento de núcleo *S1CORE*, quando impossibilitado de rotear pacotes para o *POD*2, utiliza seu plano de encaminhamento secundário enviando o pacote para ser reroteado por meio do *POD*3. O equipamento *A1P3*, tem em sua tabela de roteamento principal, como destino a mesma porta pelo qual ele recebeu o pacote, forçando-o a encaminhar o pacote a partir de seu plano de encaminhando secundário por meio do *S2CORE*. O equipamento de núcleo *S2CORE* também está impossibilitado de encaminhar pacotes para o *POD*2, e não poderá devolver pacotes pela porta que o pacote chegou, restando-lhe a opção de encaminhamento para o reroteamento por meio do POD1 ou do POD4, que em ambos os cenários encaminharam o pacote para *S1CORE*, ocasionando um *looping*.

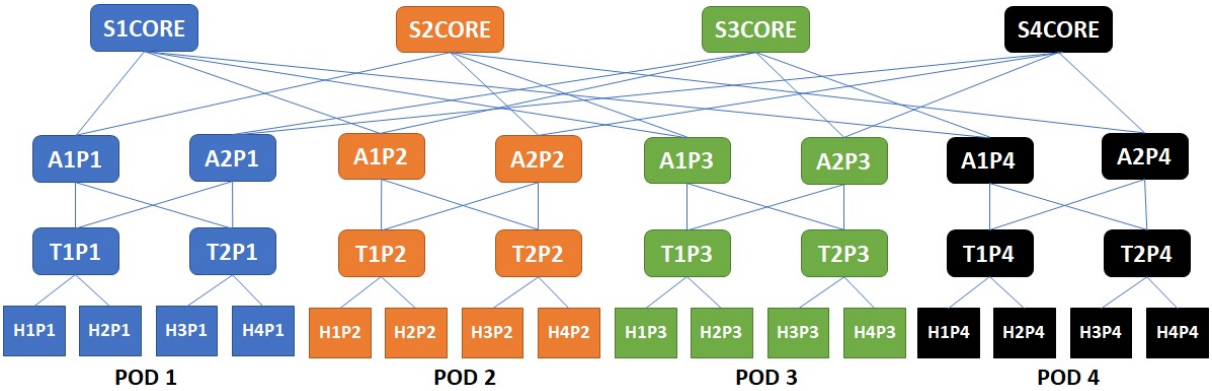


Figura 10 – Topologia de Rede *AB Fat-Tree*.

Uma outra organização bem conhecida é a topologia *AB Fat-Tree*, que propõe um padrão de interconexão cruzada conforme apresentado na Figura 10. O equipamento de Núcleo *S1CORE* está conectado aos equipamentos *A1P1*, *A1P2*, *A1P3* e *A1P4*. O equipamento de Núcleo *S2CORE* está conectado aos equipamentos *A1P1*, *A2P2*, *A1P3* e

A2P4. O equipamento de Núcleo *S3CORE* está conectado aos equipamentos *A2P1*, *A1P2*, *A2P3* e *A1P4*. O equipamento de Núcleo *S4CORE* está conectado aos equipamentos *A2P1*, *A2P2*, *A2P3* e *A2P4*. Este padrão de interconexão cruzada entre núcleo e agregação torna viável o contorno de falhas duplas em um mesmo equipamento de agregação (LIU et al., 2013b).

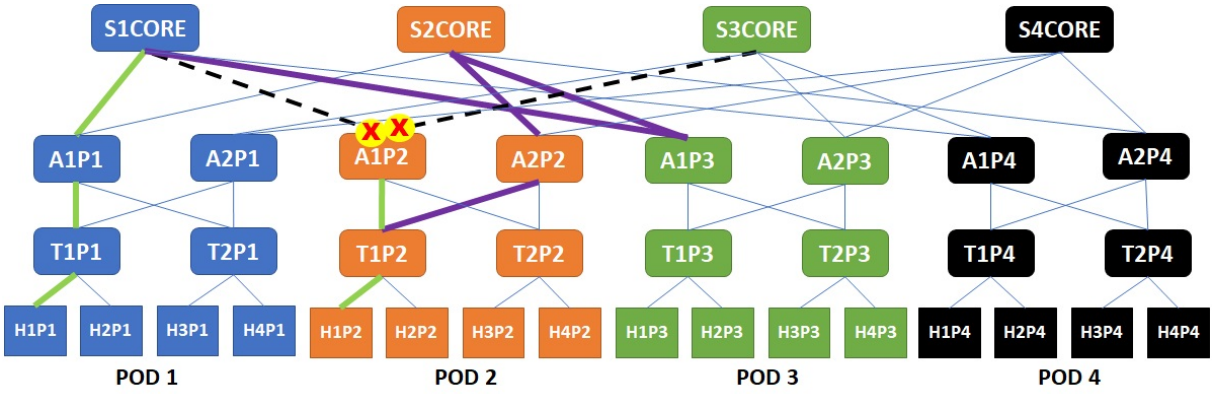


Figura 11 – AB Fat-Tree com falhas no A1P2.

A Figura 11 demonstra como é possível contornar a falha de dois enlaces em *A1P2* por meio da restruturação das interconexões dos enlaces propostos pela *AB Fat-Tree*. Graças à estrutura cruzada de interconexão, o *S2CORE* possui conexão ao *POD2* por intermédio de *A2P2*, possibilitando assim, o encaminhamento dos pacotes aos seus destinos.

3 Trabalhos relacionados

A disponibilidade da rede e seus mecanismos de recuperação são tema de extensos trabalhos de pesquisa e apresentam diferentes propostas de soluções. A Figura 12 classifica os trabalhos relacionados sobre o mecanismo de recuperação na visão das redes programáveis. Os algoritmos podem ser associados ao plano de controle, ou implementados no plano de dados e sub-classificados em mecanismos de recuperação com rotas pré-configuradas ou possuir mecanismo de busca de caminhos.

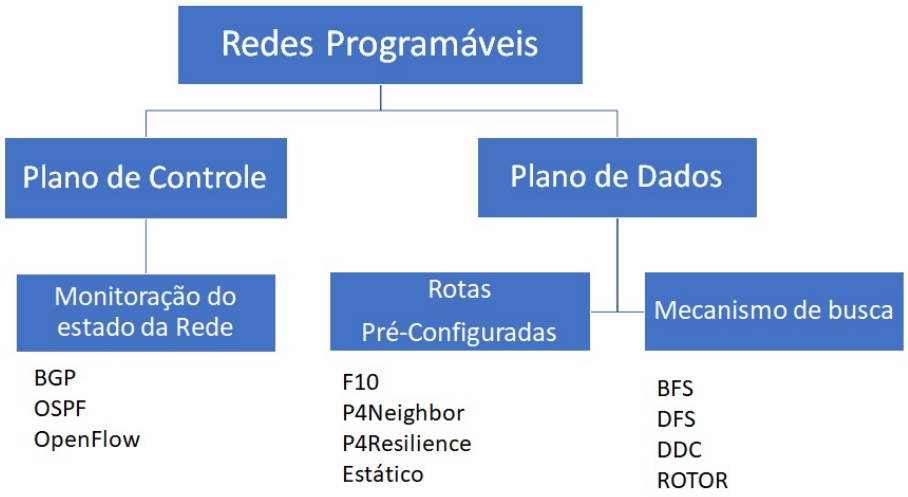


Figura 12 – Classificação Algoritmos nas Redes programáveis.

Este capítulo esta dividido em duas seções: a Seção 3.1 tratará dos principais trabalhos da literatura relacionados aos reroteamentos que inspiraram a criação do algoritmo InFaRR e a Seção 3.2 tratará especificamente dos trabalhos de reroteamento livre de cabeçalhos que utilizaremos como comparação no Capítulo 5, na execução da experimentação e da avaliação.

3.1 Propostas de Reroteamento na literatura

Os protocolos de roteamento dinâmico estão entre as ferramentas utilizadas para prover tolerância a falhas e garantir confiabilidade na rede. Com isso, quando uma conexão entre equipamentos apresenta falhas a rede pode convergir para outro caminho. Protocolos tradicionais e de ampla utilização como o Border Gateway Protocol (BGP) ou Open Shortest Part First (OSPF), podem levar dezenas de segundos para convergirem, impactando nos requisitos das classes de serviços 0 e 1 que necessitam que esta convergência seja rápida e com a menor perda de pacotes possível (CHIESA et al., 2020). Esses protocolos se caraterizam pelo constante envio de anúncios do estado da rede para atualização das

tabelas de roteamento que, consequentemente, geram *overhead* e também podem promover a convergência da rede diante do estouro de tempo (*time-out*) devido a ausência da chegada de anúncios.

O estudo voltado para a tolerância a falhas denominado F10 (LIU et al., 2013b) destaca a necessidade de uma abordagem física e lógica para implementação de um ambiente altamente resiliente. A discussão sobre pontos de falhas e processo de convergência sugere uma adaptação da topologia *Fat-Tree*, de modo que passamos para um esquema de conexão entre equipamentos chamada *AB Fat-Tree*. A proposta lógica destes autores contempla um escopo de proteção local e requer que o reroteamento dos pacotes seja executado em outro POD de forma que não possua um domínio de recuperação independente. Apesar que sua conceituada relevância científica, e de servir como inspiração para este trabalho, a proposta F10 está condicionada a uma reformulação das conexões entre os equipamentos, descaracterizando a topologia *standard Fat-Tree* e utiliza um processo de anúncios a todos seus vizinhos na ocorrência das falhas.

A proposta do algoritmo P4-Protect, implementado na linguagem P4, apresenta uma solução de escopo de recuperação global 1+1 para redes IP. Para esta solução, o algoritmo requer uma topologia com dois caminhos disjuntos ao destino. O primeiro comutador da rede é responsável por encaminhar um pacote para cada caminho (clone), adicionando ao pacote um cabeçalho de controle. O último comutador remove o cabeçalho e encaminha o primeiro pacote recebido ao destino, descartando o pacote clonado. Embora este algoritmo não execute reroteamento, ele oferece uma interessante camada de resiliência e apresenta excelentes resultados na redução de *jitter*, mas ainda requer estudos em topologias hierárquicas datacenter como *Fat-Tree* (LINDNER et al., 2020).

Oriundos da Teoria dos Grafos, os algoritmos de busca em profundidade (DFS - *Depth-First-Search*) e busca em largura (BFS - *Breadth-First- Search*) possibilitam a recuperação de falhas mesmo sem possuir uma tabela de roteamento secundário predefinido (KRISHNA; SURI; ATHITHAN, 2011). Quando o pacote não pode ser encaminhado para a porta destino, o mecanismo incia o processo de busca por um caminho viável, encaminhando o pacote sistematicamente conforme o algoritmo escolhido. A implementação dos algoritmos DFS ou BFS em P4 adiciona um cabeçalho à estrutura dos pacotes, para que os comutadores possam controlar por onde o pacote já passou. Todavia, embora não exista *loop*, a ausência de uma Mecanismo de Reconhecimento e Restauração faz com que todos os pacotes percorram um longo caminho até o destino (BOROKHOVICH; SCHIFF; SCHMID, 2014).

O amplo *survey* sobre mecanismos de reroteamento rápido em plano de dados de (CHIESA et al., 2020) apresenta o estudo mais atual sobre as diferentes abordagens de implementação desses mecanismos. Inclui-se um capítulo dedicado a redes definidas por software, em que são abordadas estratégias em ambientes Openflow e em planos de

dados programáveis (P4). O *survey* destaca a implementação do algoritmo *Data-Driven Connectivity* (DDC) em P4 que utiliza cabeçalhos adicionais para armazenar o número de falhas ao longo do caminho percorrido pelo pacote, apresentando o conceito de *link-reversal* (LIU et al., 2013a). Tal abordagem gera um *overhead* de sinalização no pacote.

Os trabalhos P4Neighbor e P4Resilience são implementações na linguagem P4 de propostas de algoritmos de reroteamento rápido e utilizam cabeçalhos adicionais na estrutura do pacote para encapsular informações do caminho *backup*; assim otimizam o reroteamento pelo comutador, já que a proposta de reroteamento está contida no cabeçalho do pacote. O algoritmo P4Neighbor quando impossibilitado de encaminhar um pacote por um enlace com falhas, adiciona um cabeçalho de recuperação ao pacote e o recircula dentro do mesmo comutador, para que o mesmo pacote seja reroteado por uma porta em funcionamento. O mecanismo de recuperação é predefinido, de modo que os múltiplos caminhos *backup* são calculados pelo plano de controle (XU; XIE; ZHAO, 2021). O P4Resilience possui como diferencial a garantia da criação de caminhos livres de *loop*; todavia os múltiplos caminhos *backup* necessitam ser armazenados no plano de dados e possuem crescimento linear em relação ao número de fluxos (LI et al., 2022).

O trabalho PURR (CHIESA et al., 2019), implementado em P4, apresenta uma solução com duas consultas a tabelas *match-action*: a primeira consulta realizada à tabela de roteamento principal, retorna à porta de saída pela qual o pacote deve ser encaminhado. A segunda consulta utiliza como chave de pesquisa a porta de saída e a matriz de *status* das portas do comutador, e oferece uma porta com o *status* ativo como porta de saída. Esta proposta otimiza o tamanho da segunda tabela mas inviabiliza o tratamento do reroteamento por fluxo ou outro tipo de segregação.

A Tabela 2 destaca os principais conceitos e desafios encontrados nos algoritmos de reroteamento que nortearam os requisitos de funcionamento do algoritmo proposto InFaRR. A coluna "conceitos e desafios" destaca as restrições ou problemas do algoritmo, enquanto que a coluna "Proposta InFaRR" apresenta como este trabalho tratará dos desafios apresentados.

Algoritimo	**Conceitos e Desafios**	**Proposta InFaRR**
BGP / OSPF Open Flow	Plano de Controle Alto tempo de convergência Plano de Encaminhamento Otimizado	Plano de dados Baixo tempo convergência Mecanismo de Reconhecimento e Restauração Mecanismos de Retorno à Rota Principal
F10	Topologia *AB Fat-Tree*	Funciona em ambas topologias: *Standard Fat-Tree* e *AB Fat-Tree*
P4 Protect	Duplicação de pacotes	Mecanismo de Retorno à rota principal
BFS / DFS / Rotor	Longos caminhos até o destino	Mecanismo de Reconhecimento e Restauração
DDC	Link-Reversal, cabeçalho adicional	*Pushback*, sem cabeçalho
P4Neighbor	Sujeito a *loop*	Mecanismo Prevenção de *Loop*
P4Resilience	Tabela com múltiplos de caminhos *backup* para cada rota	Tabela de roteamento *backup* para cada fluxo
Estático	Recuperação para porta de saída Sujeito a *loop*	Recuperação para o fluxo Mecanismo Prevenção de *Loop*

Tabela 2 – Resumo motivações para algoritmos InFaRR.

3.2 Algoritmos de Reroteamento Selecionados

Os principais trabalhos da literatura, considerados estado-da-arte no contexto de recuperação de falhas em redes programáveis, que não requerem anúncios de estado da rede e não utilizam cabeçalhos adicionais na estrutura dos pacotes, foram selecionados como trabalhos diretamente relacionados. Tais trabalhos foram adaptados para a linguagem P4 e utilizados para fins de comparação com o algoritmo InFaRR.

O primeiro algoritmo de reroteamento rápido denominado Estático, conta com rotas pré-configuradas e propõe uma tabela de roteamento secundária para fornecer contorno a falhas encontradas pela tabela de roteamento principal. O segundo algoritmo, chamado Rotor, implementa um mecanismo de busca ao destino utilizando sequencialmente as portas ativas do comutador.

Também selecionamos uma abordagem tradicional para redes programáveis com funcionamento no plano de controle, para comprovação da eficiência e dos benefícios do reroteamento rápido.

3.2.1 Reroteamento Estático

No trabalho (NIKOLAEVSKIY, 2016), intitulado de "*Scalability and Resiliency of Static Routing*", é apresentado um amplo estudo sobre a viabilidade da implementação de tolerância a falhas por meio do uso de roteamento estático e suas variações. A abordagem mais básica de roteamento estático como mecanismo de recuperação consiste em prover uma tabela de roteamento principal e uma tabela de roteamento secundária, a ser utilizada na ocorrência de falhas, ambas previamente configuradas pelo plano de controle. Esta abordagem oferece escopo de recuperação local e, quando implementada no comutador de Núcleo, necessita de um domínio de convergência de outro *POD* para encontrar um caminho alternativo até seu destino.

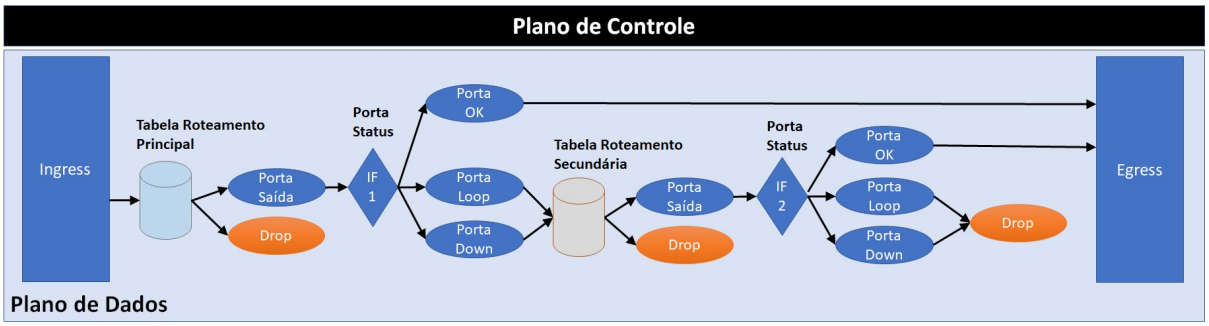


Figura 13 – Fluxo do algoritmo reroteamento Estático.

A Figura 13 apresenta o fluxo macro de funcionamento do algoritmo de reroteamento estático. Todas as ações de roteamento e reroteamento são realizadas exclusivamente no plano de dados por meio de consultas a tabelas de roteamentos previamente definidas. Assim

que o pacote passa pela fase de *Ingress* do comutador é realizada a consulta à tabela de roteamento principal. Havendo *match*, retorna-se uma porta de saída para encaminhamento do pacote, caso contrário será realizado o *Drop* do pacote. Na verificação do *status* da porta (IF 1), se a porta de saída estiver com o *status Ok* o pacote é encaminhado para *Egress*. Caso a porta esteja com o *status loop* ou *down* a próxima etapa será a consulta à tabela de roteamento secundário, em que será definida uma nova porta de saída ou a realização do *Drop* caso não exista uma rota secundária predefinida. Finalmente, uma nova verificação de *status* da porta de saída (IF 2) é realizada, e se o estiver com *status ok*, o pacote é encaminhado para o *Egress*, do contrário o pacote sofrerá *Drop*.

3.2.2 Reroteamento Rotor

O trabalho de (SEDAR et al., 2018) é uma implementação no plano de dados programável sem a necessidade do uso de cabeçalhos adicionais. O algoritmo Rotor realiza uma consulta à tabela de roteamento principal e, uma vez que a porta de saída esteja com problemas, o pacote será submetido para a próxima porta ativa disponível. O escopo de recuperação é local, e sua implementação no dispositivo de Núcleo requer que o pacote seja roteado por outro *POD*. O método de configuração foi definido previamente, visto que não existem consultas ao plano de controle. Embora não exista uma tabela de roteamento *backup* preconfigurada, o algoritmo assume a responsabilidade de descobrir um caminho alternativo até seu destino (LEISERSON, 1985).

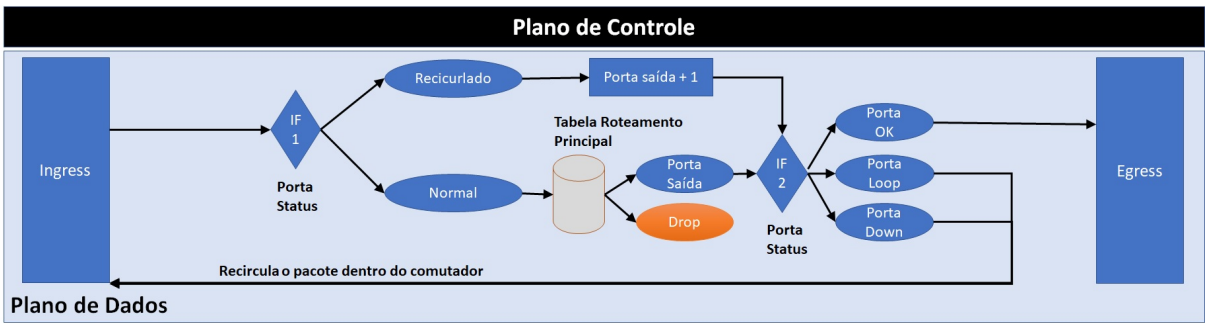


Figura 14 – Fluxo do algoritmo reroteamento Rotor.

O fluxo de funcionamento do algoritmo rotor é todo processado no plano de dados e se inicia com a verificação do tipo de pacote que será processado (IF 1). Os pacotes, quando processados pela primeira vez, são classificados como normal e seguem para a tradicional consulta a tabela de roteamento principal e verificação de *status* da porta de saída (IF 2). Estando *Ok*, o pacote é enviado para o *Egress*; do contrário o pacote é recirculado dentro do comutador, passando novamente pelo *Ingress*; todavia, agora o pacote será sinalizado como "recirculado" e será encaminhado para a próxima porta ("Porta saída + 1"). Mais uma vez é realizada a verificação do *status* da porta de saída, repetindo-se o processo até

que uma porta de saída com o *status Ok* seja encontrada. O fluxo de funcionamento do algoritmo Rotor é representado pela Figura 14.

3.2.3 Plano de Controle

Em uma arquitetura com o plano de controle centralizado, é possível oferecer escopo de recuperação global, mecanismo de configuração reativo e domínio de operação independente. Uma das atribuições do plano de controle é avaliar constantemente as condições da rede e, sempre que necessário, atualizar as tabelas de roteamento dos comutadores (ONF, 2013). O processo de avaliação pode envolver recorrentes consultas (*pooling*) do plano de controle a cada um dos equipamentos de rede, ocasionando uma considerável utilização da rede para sua própria monitoração. O intervalo do ciclo de atualização é determinante para o consumo de largura de banda para gerência e para o tempo de detecção de problemas, podendo se tornar um grande desafio (SGAMBELLURI et al., 2013).

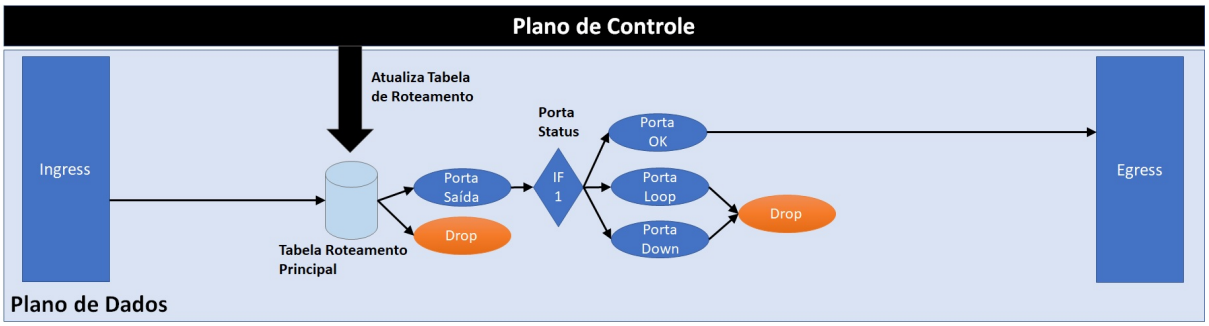


Figura 15 – Fluxo do algoritmo reroteamento no Plano de controle.

A Figura 15, representa o algoritmo de funcionamento proposto por uma solução controlado pela Plano de Controle. Nesta implementação, há uma consulta a uma tabela de roteamento e verificação do *status* da porta de saída, todavia não há nenhuma ação pré programada no plano de dados para reroteamento. Cabe ao Plano de Controle atualizar a tabela de roteamento com rotas viáveis sempre que necessário. O processo de detecção da falha e atualização da tabela de roteamento é o grande desafio deste algoritmo, e enquanto esta atualização não acontece, os pacotes sofrerão *Drop* na ocorrência de porta *Down* ou *loop*.

A Tabela 3 resume as principais características, analisadas nos trabalhos relacionados, que farão parte da experimentação deste trabalho em relação ao algoritmo proposto InFaRR.

A Tabela 3 também posiciona os algoritmos descritos acima com o InFaRR. A tabela apresenta as características dos mecanismos de recuperação, a destacar o domínio de recuperação em que se observa a necessidade de uso de outro POD para o contorno à falha. O algoritmo InFaRR se diferencia das outras abordagens de execução no plano de

Algoritmos	**Plano de Controle**	**Estático**	**Rotor**	**InFaRR**
Reroteamento Rápido	Não	Sim	Sim	Sim
Camada de funcionamento	Plano de controle	Plano de dados	Plano de dados	Plano de dados
Métodos de configuração	Reativo	Pré-configurado	Mecanismo de Busca	Pré-configurado
Escopo de recuperação	Global	Local	Local	Segmento
Domínio de recuperação	Independente	Dependente	Dependente	Independente
Plano de encaminhamento	Otimizado	Não	Não	Otimizado
Pooling de gerenciamento	Sim	Não	Não	Não

Tabela 3 – Resumo dos algoritmos selecionados.

dados pois oferece um domínio de recuperação de forma independente, sem a necessidade de uso de outros PODs, possibilitando assim um plano de encaminhamento otimizado com o menor número de saltos. O escopo de recuperação é um atributo que, em conjunto com o domínio de recuperação, viabiliza uma proteção por segmento ao prover o contorno ao local onde a falha se encontra.

4 *In-network Fast ReRouting* - InFaRR

As funcionalidades do reroteamento rápido têm início assim que o comutador detecta ser inviável o encaminhamento de pacotes através da porta indicada pela tabela de roteamento principal. O InFaRR propõe um algoritmo de reroteamento rápido para redes programáveis capaz de contornar até três falhas em redes *Fat-Tree* sem uso de cabeçalhos adicionais nos pacotes e sem mecanismos de monitoração dos estados dos enlaces (*heartbeats* ou *keep alive*).

O InFaRR foi implementado na linguagem P4 e tem como princípio básico de recuperação a consulta a uma tabela de roteamento secundária predefinida para ser utilizada em situações de falhas. As funcionalidades avançadas complementam seu funcionamento, como: 1) Mecanismo de *Pushback*; 2) Prevenção de *Loop* na rede; 3) Mecanismo de Reconhecimento e Restauração; e 4) Mecanismo de Retorno à Rota Principal.

O algoritmo InFaRR visa gerenciar, em cada comutador, informações dos fluxos ativos pertinentes ao processo de encaminhamento dos pacotes, de forma a viabilizar o reroteamento rápido dos pacotes quando necessário. O armazenamento dessas informações em memória local está condicionado à capacidade física do comutador, mas critérios de seleção dos fluxos podem ser utilizados para definir quais serão priorizados e farão uso do algoritmo proposto.

Na Seção 2.4, no capítulo de Fundamentação Teórica, descrevemos todas as etapas do Processo de Recuperação de forma detalhada. A etapa $E1$, denominada Mecanismo de Controle, que é responsável por monitorar o *status* físico das portas, avaliando se estão *UP* ou *DOWN* não é escopo do algoritmo InFaRR, embora esta informação seja o gatilho para o início do processo de reroteamento[1].

Este capítulo esta organizado em três seções: a Seção 4.1 descreve os principais conceitos utilizados pelo algoritmo InFaRR. O detalhamento do funcionamento do algoritmo é descrito na Seção 4.2 e, na Seção 4.3, apresenta-se uma análise sobre o funcionamento do algoritmo InFaRR em diferentes topologias de rede.

4.1 Taxonomia - InFaRR

A taxonomia do algoritmo InFaRR é importante para compreender a terminologia utilizada para descrever as funcionalidade apresentadas no processo de roteamento e reroteamento.

[1] Tal implementação poderia ser feita conforme descrito em https://github.com/p4lang/p4-spec/issues/709.

- **Mecanismo de Prevenção de *Loop*:** o Mecanismo de Prevenção de *Loop* é a habilidade que o comutador possui para prevenir o envio de pacotes pela mesma porta em que chegaram. Embora sua detecção seja uma atividade simples (comparação se a porta de entrada é a mesma que a de saída), exige que o comutador tenha mecanismos para tratamento desta situação; o InFaRR propõe para isso o reroteamento ou o uso do Mecanismo de *Pushback* para esta finalidade.

- **Mecanismo de *Pushback*:** habilidade do algoritmo InFaRR de encaminhar pacotes para o comutador antecessor em caso de ocorrência de falhas no processo de roteamento e reroteamento. Esta opção de encaminhamento evita que o pacote seja descartado e o encaminha ao comutador antecessor que é um caminho ativo e conhecido.

  A Figura 16 apresenta a execução do Mecanismo de *Pushback* no comutador S3: impossibilitado de encaminhar o pacote para o comutador S4 e, diante da ausência de uma rota secundária, o pacote é encaminhado para seu antecessor, S2.

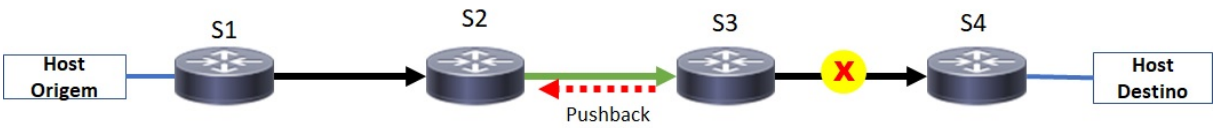


Figura 16 – *Pushback* envia pacote para o comutador antecessor.

- **Antecessor:** o conceito de antecessor se difere do conceito de comutador anterior pois é implementado durante o Mecanismo de *Pushback*. O roteamento é constituído por uma série de comutadores pelos quais o pacote necessita passar até chegar ao seu destino. O termo antecessor se aplica ao comutador $n-1$ a considerar n o comutador atual que está processando o pacote. Veja-se um exemplo com $n = 4$, os comutadores denominados como S1, S2, S3 e S4, cujo roteamento se sucede sequencialmente conforme demonstrado na Figura 16. O pacote gerado no *host* origem é encaminhado através da rede, passando sequencialmente pelos comutadores S1 -> S2 -> S3, até que encontra uma falha entre os enlaces S3-S4; o comutador S3 (n= 3) necessita realizar o Mecanismo de *Pushback* pois não possui um roteamento secundário. Assim, o pacote é encaminhado para o comutador S2 ($n = 2$). O comutador S2 tem como seu antecessor o comutador S1 ($n - 1$), enquanto que o comutador S3 é o comutador anterior pelo qual o pacote passou.

- **Mecanismo de Reconhecimento e Restauração:** o comutador, ao detectar uma situação de *loop*, inviabiliza o uso deste caminho para o fluxo que está sendo tratado e prossegue para o plano de reroteamento ou *Pushback*. Esta característica habilita o fluxo a utilizar uma tabela de roteamento otimizado já que o pacote não necessita

percorrer trechos desnecessários ocasionados pelo *loop*, e torna o comutador um PIVO no processo de reroteamento para o fluxo.

- **PIVO:** todo comutador da rede que possui uma tabela de roteamento *backup* pode vir a ser um PIVO. O comutador que possui a porta desabilitada para o fluxo pelo Mecanismo de Reconhecimento e Restauração, e possui a porta do reroteamento em funcionamento é denominado como comutador PIVO. Esta identificação é importante pois determinará o funcionamento do Mecanismo de Retorno à Rota Principal.

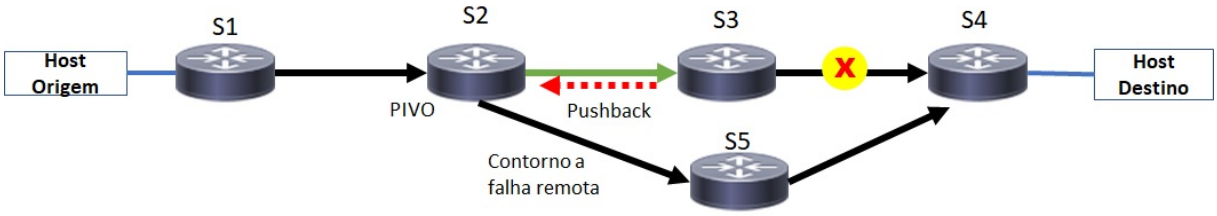


Figura 17 – Comutador PIVO otimiza o roteamento.

A Figura 17 apresenta o funcionamento do PIVO no comutador S2. O pacote segue o roteamento principal através dos comutadores S1 -> S2 -> S3, diante a falha no enlace entre S3-S4 o comutador S3 faz um *Pushback* devolvendo o pacote para S2. O Mecanismo de Prevenção de *Loop* faz que o comutador S2 encaminhe pacotes por meio do comutador S5, enquanto que o Mecanismo de Reconhecimento e Restauração faz com que os próximos pacotes do fluxo sejam encaminhados pela rota livre de falhas. O comutador PIVO possibilita o contorno a uma falha remota e onde será implementado a função do Mecanismo de Retorno à Rota Principal.

- **Mecanismo de Retorno à Rota Principal:** o Mecanismo de Retorno à Rota Principal é uma proposta do algoritmo InFaRR para detecção de recuperação a falhas remotas no plano de dados. Diante da proposta de um algoritmo livre de cabeçalhos adicionais ou de mecanismos de monitoração de enlace, quando o Mecanismo de Reconhecimento e Restauração desabilita um caminho para uso, criaria-se uma dependência do plano de controle para que o fluxo retornasse ao caminho principal. A proposta do Mecanismo de Retorno à Rota Principal é gerar um pacote duplicado, de forma a verificar se o caminho está pronto para recuperação; embora esta ação funcione de forma similar à monitoração do enlace traz independência para o plano de dados para determinar o retorno ao funcionamento do fluxo pelo enlace. A duplicação do pacote garante que um deles seja entregue ao destino por meio do caminho *backup*, caso o caminho principal ainda não tenha sido recuperado.

- **Clone:** a duplicação de pacotes é uma das funcionalidades utilizadas pelo InFaRR para o funcionamento do Mecanismo de Retorno à Rota Principal. O comutador gera uma cópia do pacote (cabeçalho + *payload*) e o envia pelo caminho logicamente

desabilitado (porta de saída sugerida pela tabela de principal), enquanto que o outro pacote é encaminhado pela porta de saída definida pela tabela de roteamento secundária (reroteamento).

- **Chave de busca:** são os parâmetros passados como índice para realização de consultas às tabelas de roteamento. As chaves de busca podem ser simples, contendo somente o campo de endereço destino do pacote, sendo comum à maioria dos equipamentos de roteamento. Também pode ser composta, com mais de um campo para busca de forma a possibilitar uma granularidade maior ao mecanismo de busca. A linguagem P4 possibilita a flexibilização da criação de tabelas e de suas chaves de buscas. Pode-se, por exemplo, criar tabelas indexadas pelo cabeçalho de endereço IP Origem e Destino, e/ou porta TCP/UDP Origem e Destino.

- **Comutador:** nas redes programáveis, os equipamentos são denominados comutadores sem distinção de em qual camada da pilha TCP/IP eles trabalham, diferentemente das redes tradicionais, em que há o conceito explicito de que equipamentos de camada rede são *switchs* e equipamentos de camada internet são roteadores. Os comutadores podem ainda trabalhar em outras camadas de rede, como na camada de transporte para realização do roteamento por tipo de serviço, ou exercer políticas de restrição, como os *firewall*.

- **Fluxo:** conjunto de pacotes que fazem parte da mesma sessão TCP/IP e que devem usar a mesma tabela de roteamento e reroteamento.

- **Controle por Fluxo:** o Mecanismo de Reconhecimento e Restauração oferece o monitoramento granular sobre fluxos e enlaces de forma que é possível identificar caminhos a serem evitados pelo fluxo. Uma vez que um pacote retorna ao comutador ocasionando um *loop*, o comutador aprende que este caminho de encaminhamento tem algum problema à frente, assim a porta é colocada logicamente como *DOWN* para este fluxo específico, continuando a porta ativa para os demais.

- **DROP:** trata-se do mecanismo utilizado para descarte de pacotes quando não é possível completar nenhum mecanismo de encaminhamento do pacote.

- **Hash:** a linguagem P4 disponibiliza registradores que funcionam como vetores, de modo que cada posição pode ser acessada por números inteiros, agindo como índices. O InFaRR utiliza a função *hash* (tabela de dispersão) como artifício para converter fluxos em números inteiros. São apresentados campos dos cabeçalhos do pacote e convertidos em números aceitos pelo índice (HAUSER et al., 2021). A função *hash* não está livre de colisões (gerar o mesmo número chave para fluxos diferentes), sendo a probabilidade de ocorrência associada ao número de fluxos processados e ao número de posições reservadas nos registadores. Este trabalho não adotou nenhum

mecanismo de tratamento de colisões causada pelo *hash*, tema que poderá ser tratado em trabalhos futuros.

- **Livre de Cabeçalhos Adicionais:** a flexibilidade da linguagem P4 em implementar e manipular novos cabeçalhos é uma das características mais exploradas e sinônimo da usabilidade da linguagem. Todavia, o crescimento da capacidade computacional dos comutadores viabiliza o desenvolvimento de algoritmos de reroteamento livres de cabeçalhos adicionais que geram *overhead* nos pacotes. A proposta do InFaRR é trazer ao comutador a habilidade de prover resiliência à rede, sem a necessidade de sobrecarregar os enlaces com cabeçalhos adicionais.

- **Livre de Monitoração:** a proposta do algoritmo InFaRR gira em torno da construção de um mecanismo de recuperação livre de *overhead* sobre a rede para monitoração dos enlaces, sejam eles *heartbeat*, anúncios de rotas, *timeout* de sessões de protocolos de roteamento ou *keep alive*. O Mecanismo de controle descrito na Seção 2.4, que não é escopo deste trabalho, pode usufruir deste tipo de solução.

- **Tabela de roteamento primária:** refere-se à tabela de roteamento utilizada para realização do roteamento, predefinida pelo plano de controle, podendo possuir diferentes políticas de roteamento para cada tipo de destino ou fluxo.

- **Tabela de roteamento secundária:** também denominada como tabela de roteamento *backup*, é utilizada pelo mecanismo de recuperação de forma a oferecer um caminho alternativo ao roteamento primário. Esta tabela também é predefinida pelo plano de controle, podendo ser constituída somente de entradas para destinos ou fluxos que requerem mecanismo de recuperação.

- ***Timestamp*:** o Mecanismo de Reconhecimento e Restauração e de Recuperação necessitam controlar o horário corrente de processamento, para isso utilizam a função *Timestamp*, a fim de coletar o horário interno do comutador. Esta informação é utilizada para mapear quando a porta para determinado fluxo foi desativada e, quando comparada ao horário atual, determinar se a mesma está apta para verificação por meio do Mecanismo de Retorno à Rota Principal.

4.1.1 Exemplo de funcionamento

A Figura 18 será utilizada como exemplo para visualização dos termos que foram descritos. A topologia apresenta uma comunicação entre o *host* origem e o *host* destino A, e uma comunicação entre o *host* origem e o *host* destino B.

A tabela de roteamento principal (destacado em azul) entre o *host* origem e o *host* destino A, tem duas falhas e as usaremos para ilustrar o Mecanismo de *Pushback* nos comutadores 2, 3 e 6, Mecanismo de Prevenção de *Loop* nos comutadores 1, 2 e 3,

Mecanismo de Reconhecimento e Restauração e Mecanismo de Retorno à Rota Principal no comutador 1. O *host* de origem só possui conectividade com o *host* destino A por meio do caminho secundário (destacado em verde), todavia para isso necessita que o comutador 1 seja capaz de detectar uma falha remota dois saltos à frente.

Existe somente uma rota direta entre o *host* origem e o *host* destino B: através do comutador 2.

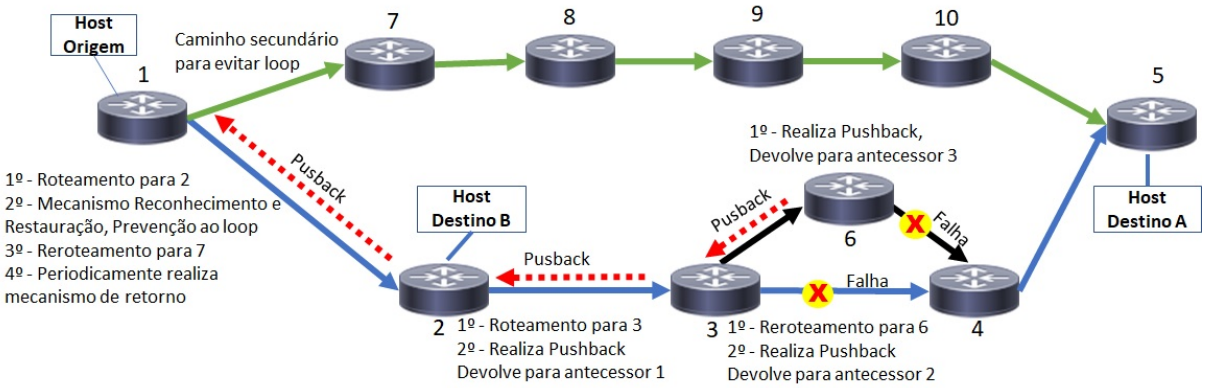


Figura 18 – Características do Mecanismo de *pushback* e Mecanismo de Reconhecimento e Restauração.

Os comutadores 1 e 2 utilizam a tabela de roteamento principal para encaminhar pacotes para o *host* destino A. O comutador 3 detecta a falha no enlace 3-4 e realiza o reroteamento via comutador 6. Porém, há também uma falha no enlace 6-4 e, neste caso, o InFaRR aciona o Mecanismo de *Pushback* devolvendo o pacote para o seu antecessor (comutador 3)[2]. É observado que o comutador 3 não possui opções viáveis de encaminhamento, pois possui uma porta *DOWN* e outra em *loop*. Neste caso, o Mecanismo de *Pushback* encaminha o pacote ao seu antecessor (comutador 2). O comutador 2 também necessita recorrer ao Mecanismo de *Pushback* e encaminha o pacote ao seu antecessor (comutador 1).

O comutador 1, ao receber de volta o pacote encaminhado pelo comutador 2, também detecta o *loop*, disparando assim o reroteamento rápido via comutador 7. Neste momento, o comutador 1 utiliza o Mecanismo de Reconhecimento e Restauração para armazenar a informação de que o caminho através do comutador 2 está indisponível, ou seja, os pacotes subsequentes deste fluxo deverão ser encaminhados via comutador 7. Após isso, o comutador 1 eventualmente inicia um Mecanismo de Retorno à Rota Principal para verificar a disponibilidade de encaminhamento de pacotes via comutador 2. Neste caso, o pacote original (encaminhado para o comutador 7) deverá ser duplicado, tendo suas cópia enviada para a interface de saída que o conecta ao comutador 2. Caso as falhas não tenham sido corrigidas no comutador 2, o pacote duplicado retornará ao comutador 1 conforme

[2] Claramente, se não existisse a falha no enlace 6-4, o pacote chegaria ao *host* destino A. Provendo o reroteamento e recuperação a falha no enlace 3-4

Mecanismo de *Pushback* já descrito. Caso o pacote duplicado não retorne, assume-se que as falhas foram recuperadas e esta rota volta a ser a principal para o processo de roteamento.

Observa-se uma leve diferença entre a execução do InFaRR no comutador 1, 2 e 3. No comutador 2 e 3, não ocorreu atuação do Mecanismo de Reconhecimento e Restauração, visto que o Mecanismo de *Pushback* enviou o pacote de volta ao comutador 1. Neste caso, o comutador 1 se beneficiou do Mecanismo de Reconhecimento e Restauração, visto que a rota secundária estava operacional, tornando-se um PIVO no processo de reroteamento. Sendo assim, observa-se que o Mecanismo de Reconhecimento e Restauração só deverá ser utilizado quando a rota secundária estiver operacional, o que não ocorre com o comutador 2 e 3 para este cenário. O InFaRR é capaz de detectar este evento acionando o Mecanismo de Reconhecimento e Restauração em diferentes pontos da rede quando necessário.

A comunicação entre o *host* origem e o *host* destino B não é afetada pelo Mecanismo de Reconhecimento e Restauração aplicado ao fluxo entre *host* origem e o *host* destino A.

4.1.2 Pseudocódigo Macro funcionamento

O pseudocódigo do InFaRR está descrito no Algoritmo 1. Inicialmente, o algoritmo gera um *hash* (linha 1) com base no cabeçalho TCP/IP[3]. O valor do *hash* computado é utilizado para indexação de vetores, a fim de armazenar as informações associadas por fluxo. A primeira informação a ser armazenada de cada fluxo é a porta de entrada, que dá acesso ao comutador antecessor (linhas 2-4), informação essencial para o Mecanismo de *Pushback*. Na linha 5, ocorre a consulta à tabela de roteamento principal.

O Mecanismo de Retorno à Rota Principal pode exercer atuação sobre os pacotes de um mesmo fluxo de duas formas diferentes: 1) O primeiro cenário avalia se eventualmente uma porta desabilitada pelo Mecanismo de Reconhecimento e Restauração deve ser verificada (linha 6), quando isso é necessário realiza-se a duplicação do pacote (clone). O pacote duplicado é encaminhado pela porta desabilitada[4] (linha 8), enquanto que o pacote original é encaminhado pela rota secundária. Dois registradores são utilizados pelo Mecanismo de Retorno à Rota Principal: 1) um registrador armazena o *timestamp* de quando o pacote foi duplicado; 2) outro registrador identifica que o comutador está no modo "recuperação"(linhas 9-10).

A segunda forma de atuação do Mecanismo de Retorno à Rota Principal ocorre quando o comutador esta no modo de "recuperação". Os pacotes do fluxo enviados posteriores a etapa de duplicação tem com objetivo controlar o tempo transcorrido desde a duplicação. Para cada pacote recebido do fluxo, verifica-se o tempo transcorrido desde

[3] Neste trabalho, a operação de *hash* utiliza os endereços IP de origem e destino, portas de origem e destino. Qualquer outra combinação de campos também pode ser utilizada.

[4] Neste momento a porta de saída foi definida pela tabela de roteamento principal e assim o pacote será encaminhado pela porta saída que estava desabilitada pelo Mecanismo de Reconhecimento e Restauração para este fluxo

Algoritmo 1: Pseudocódigo - InFaRR

```
1  hashFluxo ← hash(cabecalhoIP, TCP);                                          ▷ gera hash
2  if ( porta_antecessor[ var_hashfluxo ] = vazio ) then
3  |  porta_antecessor[hashFluxo] ← porta_entrada                              ▷ Caminho para antecessor
4  end
5  Consulta Tabela_Roteamento_Principal                                         ▷ Retorna porta_saída;
6  hashfluxoAtivo ← hash(hashFluxoAntecessor, porta_saida);                     ▷ Gera hash
7  if ( ( Hora_fluxo_down[ hashFluxo , porta_saida ] ) + Tempo_Recuperacao ) < timestamp ) and (
     Modo_Recuperacao[ hashFluxo , porta_saida ] = False) and (PIVO[ hashFluxo ] = True) then
8  |  clone pacote                                                             ▷ Duplica pacote
9  |  Modo_Recuperacao[hashFluxo, porta_saida] ← True                          ▷ Ativa modo "recuperação"
10 |  hora_Envio_Clone[hashFluxo, porta_saida] ← timestamp
11 end
12 if ( porta_entrada = porta_saida ) then
13 |  fluxo_ativo[hashFluxo, porta_saida] ← False                              ▷ Desativa Fluxo
14 |  Hora_fluxo_down[hashFluxo, porta_saida] ← timestamp                      ▷ Registra Hora Desativação
15 |  Modo_Recuperacao[hashFluxo, porta_saida] ← False                         ▷ Desativa modo "recuperação"
16 end
17 if ( Modo_Recuperacao[ hashFluxo , porta_saida ] = True ) and ( ( hora_Envio_Clone[ hashFluxo,porta_saida ]
     + Tempo_timeout ) > timestamp ) then
18 |  fluxo_ativo[hashFluxo, porta_saida] ← True                              ▷ Recupera Fluxo
19 end
20 if ( porta_saida = DOWN ) ou ( fluxo_ativo[ hashFluxo , porta_saida ] = False ) then
21 |  Consulta Tabela_Roteamento_Secundário                                    ▷ Retorna porta_saída
22 |  if ( porta_entrada = porta_saida ) then
23 |  |  fluxo_ativo[hashFluxo, porta_saida] ← False                           ▷ Antiloop
24 |  |  PIVO[hashFluxo] ← True
25 |  end
26 |  if ( porta_saida = DOWN ) ou ( fluxo_ativo[ hashFluxo , porta_saida ] = False ) then
27 |  |  porta_saida ← porta_antecessor[hashFluxo]                             ▷ Pushback
28 |  |  PIVO[hashFluxo] ← False
29 |  end
30 end
31 Encaminha pacote através da porta de saída
```

a geração do pacote duplicado (linha 17). Quando o tempo transcorrido é maior que o intervalo estipulado (pacote duplicado não retornou), a porta pode ser reativada (linha 18). O intervalo de tempo transcorrido deve ser no mínimo de um RTT até o comutador com falha. É importante reforçar que caso a falha ainda exista, o pacote duplicado retornará ao comutador PIVO que gerou o pacote duplicado, ocasionando um *loop*. Neste caso, o Mecanismo de Reconhecimento e Restauração fará com que o comutador saia do modo "recuperação" e a porta permaneça desativada.

Conforme apresentado na Seção 2.5, a linguagem P4 não possui mecanismos para executar ações em *background*, sendo necessário sempre a existência de um pacote para execução qualquer atualização dos registradores. A chegada de pacotes de um mesmo fluxo é o que possibilita mensurar o tempo transcorrido desde a duplicação do pacote, sem os pacotes não seria possível mensurar se o intervalo estipulado foi atingido.

As próximas etapas representadas no pseudo código do algoritmo InFaRR realizam verificações se o pacote necessitará ser reroteado. Na linha 12, caracteriza o Mecanismo de Prevenção de *Loop* que verifica se a porta de entrada é igual à porta de saída. Quando o *loop* é identificado o Mecanismo de Reconhecimento e Restauração utiliza o *hash* do fluxo e a porta de saída como índice no vetor *fluxo_ativo* para armazenar que aquela porta não pode ser utilizada pelo fluxo (valor *False*) (linhas 13-15), para que esta porta não seja

utilizada novamente por pacotes futuros deste fluxo. O Mecanismo de Reconhecimento e Restauração também armazena o *timestamp* de quando a porta foi desativado para este fluxo, informação que será utilizada pelo Mecanismo de Retorno à Rota Principal.

Na linha 21, o InFaRR realiza duas verificações: 1) se a porta de saída está fisicamente *UP/DOWN*; 2) se a porta de saída está ativa para o fluxo[5]. Se a porta estiver em estado físico *UP* e não estiver em *loop*, o algoritmo envia o pacote através da porta de saída sinalizada pela tabela de roteamento principal. Caso contrário, o reroteamento é iniciado com uma consulta a tabela de roteamento secundária (linha 22) e as duas verificações são realizadas novamente antes do encaminhamento do pacote (linhas 22-27). Entretanto, caso a porta de saída ainda esteja indisponível (estado de *DOWN* ou em *loop*), o Mecanismo de *Pushback* (linha 27) é disparado. Finalmente, o comutador encaminha o pacote para porta de saída (linha 32).

O pacote duplicado, durante o Mecanismo de Retorno à Rota Principal, quando não consegue alcançar seu destino, é encaminhado aos seus antecessores por meio do Mecanismo de *Pushback*, momento em que a variável *Hora_Iniciar_Recuperacao* é atualizada diante da detecção do *loop* (Algoritmo 1, linha 14) e o modo de "recuperação" é desativado (linha 15). Assim, o Mecanismo de Retorno à Rota Principal usufrui da habilidade do Mecanismo de Prevenção de *Loop* para definir e manter o fluxo como inativo por meio da porta em que o Mecanismo de *Pushback* atuou. As seguintes estruturas de vetores foram definidas:

- *vetores_auxiliares_pushbak*: armazenam informações para realização do Mecanismo de *Pushback*, cada posição ocupando 57 bits de memória. Neste vetor, são armazenados a primeira porta de entrada do pacote no comutador (antecessor) com 9 bits[6] e o endereço MAC de origem com 48 bits[7] para futura utilização como MAC de destino.

- *vetores_auxiliares_reconhecimento_loop*: armazenam informações utilizadas pelo Mecanismo de Reconhecimento e Restauração e Prevenção de *Loop* para o fluxo, cada posição ocupando 1 bit de memória. Quando o fluxo está ativo para esta porta, é armazenado 0, quando o fluxo está inativo armazena-se 1.

- *vetores_auxiliares_recuperação*: armazenam informações utilizadas pelo Mecanismo de Retorno à Rota Principal, cada posição ocupando 97 bits de memória. Esta estrutura armazenará as seguintes informações: o *timestamp* em que a porta entrou em *loop* com 48 bits[6], armazenará se o comutador é do tipo PIVO com 1 bit (sendo o bit 0 para o caso negativo e o bit 1 para quando o comutador é um PIVO) e armazenará o *timestamp* em que se iniciou o processo de recuperação para controle do intervalo transcorrido com 48 bits[6].

[5] Não está em *loop* e não foi marcada pelo Mecanismo de Reconhecimento e Restauração (*vetor fluxo_ativo*).
[6] Tamanho definido pelo P4.
[7] Tamanho definido padrão Ethernet.

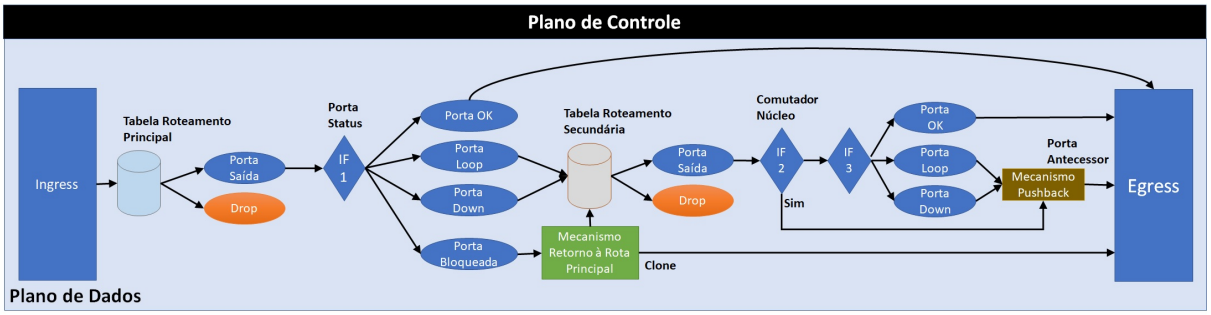


Figura 19 – Fluxo algoritmo reroteamento InFaRR.

A Figura 19 apresenta o fluxo macro de funcionamento do algoritmo InFaRR, em que se podem observar as características descritas no pseudocódigo descrito anteriormente. O algoritmo InFaRR realiza *Drops* de pacotes somente quando não existe um rota predefinida na tabela de roteamento principal ou na tabela de roteamento secundária. Em todos os outros casos, o comutador é capaz de realizar o encaminhamento do pacote contornando portas com *status DOWN*, situações de *loop* e fluxo marcados pelo Mecanismo de Reconhecimento e Restauração. Quando identificado que o comutador é um núcleo de rede, o pacote é automaticamente enviado ao Mecanismo de *Pushback* conforme demonstrado no *"IF2"*.

As estruturas de vetores alocadas para cada fluxo totalizam 155 bits (aproximadamente 20 bytes). Assim, foram alocados 2 Mbytes de memória para o tratamento de 100.000 fluxos diferentes. A escalabilidade dos números de fluxos tratados está diretamente associada à capacidade de memória disponível. Nesta implementação, não realizamos nenhum processo de otimização, agrupamento e seleção de quais fluxos seriam protegidos pelo algoritmo InFaRR, seja por meio dos hosts envolvidos (endereços IPs), tipo de serviço utilizado (portas TCP) ou qualquer outro mecanismo de classificação de QoS.

A escalabilidade do algoritmo é uma combinação de dois fatores: o primeiro fator determinante está relacionado ao espaço de memória que o comutador utilizará para armazenar os vetores do algoritmo InFaRR. Como mencionado anteriormente, os vetores são indexados pela fórmula *hash* e estão sujeitos a colisões, sendo que a probabilidade de colisão aumenta com o crescimento do número de fluxos e diante da proporção de posições disponíveis nos vetores. O segundo fator está implícito à existência de uma entrada na tabela de roteamento secundária, na qual somente os fluxos prioritários precisam ter configurado o mecanismo de reroteamento.

4.2 Máquina de Estados Finita

O algoritmo InFaRR pode se encontrar em diferentes estados de funcionamento durante o processamento do fluxo de dados: em uma situação normal o comutador estará no estado de roteamento. Diante de uma falha na porta de encaminhamento ou

situação de *loop* estará reroteando seus pacotes. Impossibilitado de realizar o reroteamento, passará ao estado do Mecanismo de *Pushback*. O estado de Mecanismo de Reconhecimento e Restauração possibilitará otimizar o contorno a falhas remotas e ainda habilitará, ocasionalmente, o estado do Mecanismo Retorno à Rota Principal. Assim, diante desses diferentes estados de processamento em que o algoritmo InFaRR pode se encontrar, entendemos que o uso de uma máquina de estado finita é a melhor forma para documentar o funcionamento do algoritmo, possibilitando o detalhamento da codificação realizada na linguagem P4.

A máquina de estados finita representada na Figura 20 (*Finite State Machine* - FSM), demonstra o funcionamento em detalhes do algoritmo InFaRR. Nesta seção demonstraremos a codificação em linguagem P4 para cada estado. Assim que os pacotes passam pelo *PARSER*, as ações de encaminhamento do pacote são iniciadas tomando como base a tabela de roteamento principal, a tabela de roteamento secundária e as funcionalidades apresentadas pelo algoritmo InFaRR. Desta forma, os pacotes podem ser encaminhados para um estado de aceitação *DEPARSER*, com uma porta de saída determinada, ou os pacotes podem ser encaminhados para descarte (*drop*). Na nomenclatura dos estados adaptamos os nomes dos mecanismos apresentados pelo InFaRR para melhor interpretação de suas funcionalidades, o Mecanismo de Prevenção de *Loop* pode ser observado como condição para transição dos estados. Descreveremos a seguir o papel de cada estado e as características necessárias para cada transição (mudança de estado).

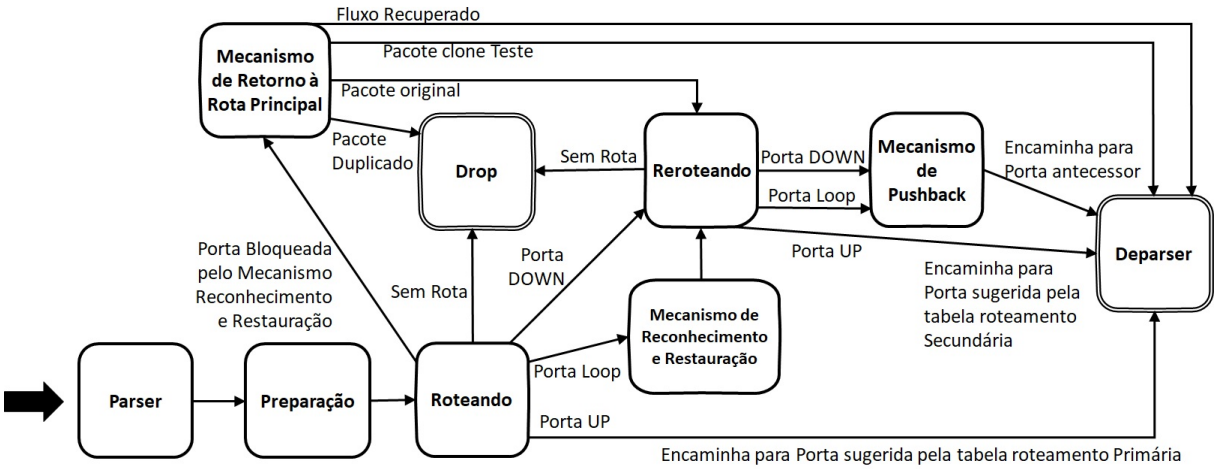


Figura 20 – Máquina de estado finita do algoritmo InfaRR.

4.2.1 Estado Parser

O estado Parser monitora a chegada de pacotes; assim que um pacote com a estrutura de cabeçalhos válida é reconhecido ocorre a transição para o estado Preparação. O Algoritmo 2 demonstra o trecho do código fonte do InFaRR no qual ocorre o PARSER. O InFaRR foi programado para aceitar pacotes do tipo *ethernet* (linha 11) do tipo IPv4 e os

cabeçalhos válidos da camada de transporte ICMP, UDP ou TCP (respectivamente linhas 18, 19 e 20). Qualquer outro formato de cabeçalho não será aceito por este comutador.

Algoritmo 2: InFaRR - PARSER

```
1  parser MyParser(packet_in packet,
2  out headers hdr,
3  inout metadata meta,
4  inout standard_metadata_t standard_metadata) {
5    state start {
6      transition parse_ethernet;
7    }
8    state parse_ethernet {
9      packet.extract(hdr.ethernet);
10     transition select(hdr.ethernet.etherType) {
11       TYPE_IPV4: parse_ipv4;
12       default: accept;
13     }
14   }
15   state parse_ipv4 {
16     packet.extract(hdr.ipv4);
17     transition select(hdr.ipv4.protocol){
18       TYPE_ICMP: parse_icmp;
19       TYPE_UDP: parse_udp;
20       TYPE_TCP: parse_tcp;
21       default: accept;
22     }
23   }
24   state parse_tcp {
25     packet.extract(hdr.tcp);
26     transition accept;
27   }
28   state parse_udp {
29     packet.extract(hdr.udp);
30     transition accept;
31   }
32   state parse_icmp {
33     packet.extract(hdr.icmp);
34     transition accept;
35   }
36 }
```

4.2.2 Inicialização das variáveis

A fase inicial do processo necessita que sejam realizadas algumas inicializações de variáveis e constantes que serão utilizadas durante todo o processo de comutação do pacote. A linguagem P4, fundamentada em uma arquitetura PISA, apresentada na Sessão 2.5, interpreta o processamento de cada pacote de forma autônoma e independente, e não permite a troca de informações diretas entre variáveis ou informações, do cabeçalho do pacote com outros pacotes e *pipelines*. A linguagem P4 possui três mecanismos para armazenamento de informações que podem ser utilizados para troca de infomações: contadores (*counters*), medidores (*meters*) e registradores (*registers*) (BOSSHART et al., 2014).

O algoritmo InfaRR utiliza registradores para controlar o funcionamento do fluxo do pacote, o que lhe permite implementar os Mecanismos de Reconhecimento e Restauração, Mecanismo de Retorno à Rota Principal e Mecanismo de *Pushback*.

O Algoritmo 3 descreve as constantes, registradores e variáveis utilizadas pelo InFaRR. As constantes são definidas entre as linhas 1 e 8, e têm como função parametrizar o funcionamento do algoritmo e auxiliar na construção e interpretação do código em alto nível. A nomenclatura das constantes foi padronizada com nomes declarados em maiúsculo e será utilizada durante todo o algoritmo. Descreveremos a seguir a usabilidade de cada uma delas:

- *PORTA_DOWN* e *PORTA_UP*: são definidas respectivamente com bit 1 e 0, para abstração de verdadeiro e falso para auxiliarem a interpretação e no desenvolvimento lógico do código fonte.

- *PIVO_FRR*: define com o bit 1 se o comutador utilizou o Mecanismo de Reconhecimento e Restauração.
- *SIM* e *NAO*: são definidas respectivamente com bit 1 e 0, para abstração de verdadeiro e falso para auxiliar a interpretação e o desenvolvimento lógico do código fonte.
- *REVALIDA* e *NAO_REVALIDA*: são definidas respectivamente com bit 1 e 0, para abstração de verdadeiro e falso, para auxiliarem a interpretação e no desenvolvimento lógico do código fonte;
- *NUMERO_FLUXOS*: define o número de fluxos que poderão ser tratados pelo algoritmo InFaRR.
- *NUMERO_PORTAS*: define o número de portas existentes no comutador.
- *TIME_OUT_RECUPERA_FLUXO*: Define o intervalo de tempo que o Mecanismo de Retorno à Rota Principal deve aguardar para executar a função de recuperação.
- *TIME_OUT_VALIDA_CLONE*: Define o intervalo de tempo que o Mecanismo de Retorno à Rota Principal deve aguardar para validar se o pacote duplicado não retornou através do Mecanismo de *Pushback*.

Algoritmo 3: InFaRR - Constantes, Registradores e Variáveis

```
1  const bit<1> PORTA_DOWN = 1;
2  const bit<1> PORTA_UP = 0;
3  const bit<1> PIVO_FRR = 1;
4  const bit<1> NAO = 0;
5  const bit<1> SIM = 1;
6  const bit <1> REVALIDA = 1;
7  const bit <1> NAO_REVALIDA = 0;
8  const bit <32> NUMERO_FLUXOS = 100000;
9  const bit <32> NUMERO_PORTAS = 5;
10 const bit <48> TIME_OUT_RECUPERA_FLUXO =
     250000;
11 const bit <48> TIME_OUT_VALIDA_CLONE =
     100000;
12
13 register <bit <9 > (NUMERO_FLUXOS)
     fluxo_porta_entrada;
14 register <bit <48 > (NUMERO_FLUXOS)
     fluxo_mac_entrada;
15 register <bit <1> > (NUMERO_FLUXOS) fluxo_status;
16 register <bit <48> > (NUMERO_FLUXOS)
     fluxo_tempo;
17 register <bit <1> > (NUMERO_FLUXOS)
     fluxo_pivo_saida;
18 register <bit <1> > (NUMERO_FLUXOS)
     clone_valida_fluxo;
19
20 bit <9> var_fluxo_porta_entrada;
21 bit <1> var_status_fluxo_saida;
22 bit <48> var_fluxo_tempo;
23 bit <1> var_fluxo_pivo_saida;
24 bit <1> var_clone_valida_fluxo;
25 bit <9> var_saida_primeira_rota;
26 bit <48> var_hash_fluxo_porta_entrada;
27 bit <48> var_hash_fluxo_porta_saida;
```

A segunda parte do Algoritmo 3, entre as linhas 14 e 19, descreve os registradores que serão utilizados para o funcionamento dos mecanismos de recuperação propostos pelo algoritmo InFaRR. Os registradores funcionam como *arrays* de uma dimensão que podem ter seus conteúdos acessados diretamente. A constante *NUMERO_FLUXOS* especifica o números de posições que o registrador possuirá e sua indexação será feita por meio do cálculo do *hash* utilizando como parâmetro campos específicos dos cabeçalhos do pacote. O cálculo da função *hash* retornará um valor entre 0 (zero) e o valor definido por *NUMERO_FLUXOS*. Abaixo a descrição dos registrados utilizados pelo InFaRR:

- *fluxo_porta_entrada*: utilizado pelo Mecanismo de *Pushback*, este registrador armazenar a porta de entrada do fluxo em sua primeira passagem pelo comutador. Esta informação possibilitará o envio do pacote ao seu antecessor.
- *fluxo_mac_entrada*: armazena o endereço MAC de origem do pacote em sua primeira passagem pelo comutador. É um registrador complementar ao registrador anterior;
- *fluxo_status*: Utilizado pelo Mecanismo de Reconhecimento e Restauração, armazena o *status* lógico do funcionamento para um determinado fluxo. Por meio dele é possível mapear falhas remotas e otimizar o encaminhamento dos pacotes.
- *fluxo_tempo*: complementar ao registrador anterior armazena a hora em que o fluxo ficou logicamente DOWN para o fluxo. A partir de informações do registador, o Mecanismo de Retorno à Rota Principal poderá saber a hora de duplicar o pacote.
- *fluxo_pivo_saida*: O Mecanismo de Retorno à Rota Principal deve ser utilizado somente nos comutadores que realizam o reroteamento devido ao Mecanismo de Reconhecimento e Restauração. Este registrador é responsável por identificar os comutadores que fazem o papel de PIVO para o fluxo na estrutura de roteamento.
- *clone_valida_fluxo*: os fluxos podem ser compostos por vários pacotes por segundo. Este registrador é utilizado pelo Mecanismo de Retorno à Rota Principal para garantir que somente um pacote seja utilizado pelo algoritmo para duplicação do pacote.

Na última parte do Algoritmo 3, entre as linhas 21 e 28, são declaradas as variáveis utilizadas pelo algoritmo InFaRR. A maioria das variáveis possui correlação direta com os registradores, sejam elas para armazenar o conteúdo ou indexar o registrador. As variáveis possuem conteúdo momentâneo válido somente para o processamento do pacote. Assim, a cada novo pacote, os valores das variáveis são reiniciados e precisarão ser recalculados ou atribuídos novamente, com valores dos registradores. As variáveis são descritas a seguir:

- *var_fluxo_porta_entrada*: variável correlacionada ao registrador *fluxo_porta_entrada*.
- *var_status_fluxo_saida*: variável correlacionada ao registrador *fluxo_status*.
- *var_fluxo_tempo*: variável correlacionada ao registrador *fluxo_tempo*.
- *var_fluxo_pivo_saida*: variável correlacionada ao registrador *fluxo_pivo_saida*.
- *var_clone_valida_fluxo*: variável correlacionada ao registrador *clone_valida_fluxo*.
- *var_saida_primeira_rota*: variável auxiliar que armazena a rota sugerida pelo estado de Roteamento, que será utilizada durante o estado de Reroteamento para definição de comutador é um PIVO.

- *var_hash_fluxo_porta_entrada*: variável utilizada como índice dos registadores *fluxo_porta_entrada* e *fluxo_mac_entrada*.

- *var_hash_fluxo_porta_saida*: variável utilizada como índice dos registadores *fluxo_status*, *fluxo_tempo*, *clone_valida_fluxo* e *fluxo_pivo_saida*.

Durante o estado de Preparação, o algoritmo InFaRR realiza o mapeamento da porta de entrada do pacote durante a primeira passagem do fluxo pelo comutador. Esta importante informação é utilizada pelo Mecanismo de *Pushback* para o envio do pacote ao seu antecessor.

O Algoritmo 4 descreve, entre as linhas 1 e 8, o cálculo do *hash* do cabeçalho do pacote que será utilizado como índice para consulta e gravação do registrador *fluxo_porta_entrada* e *fluxo_mac_entrada*. Cada tipo de pacote ICMP, TCP e UDP possui uma estrutura de cabeçalho formada por diferentes campos, assim o cálculo da variável tem diferentes parâmetros conforme o tipo de pacote. Na linha 10, é realizada a consulta ao registrador *fluxo_porta_entrada* utilizando como índice o valor *hash*, calculado anteriormente.

A linguagem P4 tem como característica a inicialização de suas variáveis com valor 0 (zero). A linha 12 faz uma verificação de se o resultado da consulta à posição indicada pelo *hash* ao *fluxo_porta_entrada* é zero. Se sim, o comutador entende que este pacote está passando pela primeira vez e armazenará o número da porta de entrada e o endereço MAC de entrada (linhas 13 e 14). Caso a consulta retorne um valor diferente de 0 (zero), significa que o fluxo já passou pelo comutador e nenhuma ação necessita ser tomada.

Algoritmo 4: InFaRR - Mapeamento do Antecessor

```
1  if (hdr.ipv4.protocol == TYPE_ICMP){
2     hash(var_hash_fluxo_porta_entrada , HashAlgorithm.crc32, (bit <32>) 0 , {hdr.ipv4.srcAddr ,
     hdr.ipv4.dstAddr} , (bit <32>) NUMERO_FLUXOS);
3  } else if (hdr.ipv4.protocol == TYPE_TCP){
4     hdr.ipv4.dstAddr , hdr.tcp.srcPort , hdr.tcp.dstPort , hdr.tcp.seqNo} , (bit <32>)NUMERO_FLUXOS);
5     hash(var_hash_fluxo_porta_entrada , hashHashAlgorithm.crc32, (bit <32>) 0 , {hdr.ipv4.srcAddr ,
     hdr.ipv4.dstAddr , hdr.tcp.srcPort , hdr.tcp.dstPort} , (bit <32>) NUMERO_FLUXOS);
6  } else if (hdr.ipv4.protocol == TYPE_UDP){
7     hash(var_hash_fluxo_porta_entrada , HashAlgorithm.crc32 , (bit <32>) 0 , {hdr.ipv4.srcAddr ,
     hdr.ipv4.dstAddr , hdr.udp.srcPort , hdr.udp.dstPort} , (bit <32>) NUMERO_FLUXOS);
8  }
9
10 fluxo_porta_entrada.read(var_fluxo_porta_entrada , (bit <32>) var_hash_fluxo_porta_entrada);
11
12 if (var_fluxo_porta_entrada == 0) {
13    fluxo_porta_entrada.write( (bit <32>) var_hash_fluxo_porta_entrada , (bit <48>)
     standard_metadata.ingress_port);
14    fluxo_mac_entrada.write( (bit <32>) var_hash_fluxo_porta_entrada , hdr.ethernet.srcAddr);
     var_fluxo_porta_entrada = (bit <48>) standard_metadata.ingress_port;
15 }
```

4.2.3 Estado Roteando

O estado Roteamento é responsável por exercer a função básica de comutação de pacotes e está presente em todos comutadores de redes programáveis. Os pacotes que necessitam de comutação na camada de rede, em que os *hosts* são da mesma sub-rede IP, ou os pacotes que necessitam de roteamento camada de internet para prover interconexão entre redes diferentes, são processados da mesma forma no comutador P4, por meio de uma consulta à tabela de roteamento.

A flexibilidade da linguagem P4 habilita a construção de tabelas de roteamento personalizadas; as chaves de buscas podem ser customizadas e compostas por outros campos além do endereço IP destino utilizado tradicionalmente. Assim, é possível o roteamento com base em qualquer campo dos cabeçalhos do pacote, sejam eles IP, TCP ou outros.

Avaliamos durante a execução deste trabalho a viabilidade de seleção de fluxos prioritários que necessitariam de políticas de roteamento diferenciadas. Consideramos como estrutura para seleção os campos endereço IP de origem, endereço IP de destino e portas TCP destino, conforme Algoritmo 6. A implementação da tabela de roteamento com esta estrutura de chave de pesquisa se demonstrou viável e de fácil implementação bastando ao administrador da rede inserir os fluxos necessários na tabela de roteamento. Os fluxos que não possuem rotas definidas na tabela de roteamento sofrem a transição para a estado *Drop*.

Algoritmo 5: InFaRR - Roteamento

Chave IP Destino

```
1  table ipv4 {
2    key = {
3      hdr.ipv4.srcAddr: lpm;
4    }
5    actions = {
6      ipv4_forward;
7      drop;
8    }
9    size = 1024;
10   default_action = drop();
11 }
```

Algoritmo 6: InFaRR - Roteamento

Chave Seleção de fluxo

```
1  table ipv4_flow {
2    key = {
3      hdr.ipv4.srcAddr: exact;
4      hdr.ipv4.dstAddr: exact;
5      hdr.tcp.dstPort: exact;
6    }
7    actions = {
8      ipv4_forward;
9      drop;
10   }
11   size = 1024;
12   default_action = drop();
13 }
```

O Algoritmo 5, extraído do código fonte do InFaRR, representa a consulta à tabela de roteamento que utiliza endereço IP destino como chave, enquanto que o Algoritmo 6 representa a opção de uma estrutura para seleção de fluxos prioritários que considera como chave os campos: endereço IP de origem, endereço IP de destino e porta TCP de destino. Embora as tabelas possuam chaves de pesquisas diferentes, utilizam a mesma estrutura de ações para realização do roteamento, no caso *ipv4_forward*. As informações contidas

dentro do parâmetro *actions* denominam as sub-funções que poderão ser acionadas em caso de *match* pela tabela.

A lista de comandos na Figura 21 representa a construção de uma tabela de roteamento pelo Plano de Controle que utiliza endereço IP destino. Por meio do comando "*table_add*" que possui os seguintes parâmetros: 1) nome da tabela (destacado em rosa) onde a rota será inserida, 2) sub-função (cinza) que será utilizada em caso de *match*, 3) chave de pesquisa (verde), 4) o endereço MAC destino (amarelo) e 5) porta por onde o pacote deve ser encaminhado (amarelo). O endereço de rede cadastrado como chave de pesquisa pode utilizar recursos de máscara de rede (verde), de forma que não se faz necessária a inclusão de *host* a *host*, podendo ser utilizado a partir de apenas uma entrada na tabela de roteamento uma rede inteira. O MAC destino e porta de saída destacados em amarelo são passados como parâmetro para o acionamento da sub-função (cinza) quando o *match* ocorre na tabela.

```
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.1.0.0/16 => 08:00:00:01:03:00 1
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.2.0.0/16 => 08:00:00:02:03:00 2
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.3.0.0/16 => 00:00:00:03:03:00 3
table_add MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.4.0.0/16 => 00:00:00:04:03:00 4
```

Figura 21 – Plano de controle - Comandos para inclusão de rotas na tabela MyIngress.ipv4_lpm.

A sub-função "*MyIngress.ipv4_forward*" descrita no Algoritmo 7, realiza as seguintes operações: 1) atualizam o campo MAC origem e MAC destino do cabeçalho *ethernet*, 2) o decremento do campo TTL do cabeçalho IP e 3) define a porta de saída para a qual o pacote deve ser encaminhado. O acionamento da função é realizado pelo *match* da tabela e pela passagem dos respectivos parâmetros de resposta.

A sintaxe de inicialização de variáveis na linguagem P4 requer a definição do número de bits antes do nome da variável. A constante "*macAddr_t*" de 48 bits dimensiona a variável "*dstAddr*" (destacado em amarelo), e a constante "*egressSpec_t*" de 9 bits dimensiona a variável "port" (azul).

Algoritmo 7: InFaRR - Sub-função *actions*

```
1 typedef bit <9> egressSpec_t;
2 typedef bit <48> macAddr_t;
3 action ipv4_forward( macAddr_t dstAddr , egressSpec_t port ) {
4     standard_metadata.egress_spec = port;
5     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
6     hdr.ethernet.dstAddr = dstAddr;
7     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
8 }
```

Existem 5 (cinco) transições possíveis para o próximo estado, que são escolhidas

de acordo com o *status* físico[8] ou lógico da porta de saída:

- Porta de saída com status OK: a transição ocorrerá para o estado Deparser;
- Porta de saída com status DOWN: a transição ocorrerá para o estado Reroteando;
- Porta de saída não especificada: em casos em que não ocorrereu *match* na tabela de roteamento, a transição ocorrerá para o estado de DROP;
- Porta de saída em *loop*: em casos em que a porta de saída seja mesma que a porta de entrada, a transição ocorrerá para o estado de Mecanismo de Reconhecimento e Restauração;
- Porta de saída bloqueada pelo Mecanismo de Reconhecimento e Restauração: a transição ocorrerá para o estado de Mecanismo de Retorno à Rota Principal.

4.2.4 Estado Deparser

O estado de aceitação "Deparser" é responsável por recalcular os *checksum* dos cabeçalhos que foram atualizados. A passagem do pacote pelo comutador termina com encaminhamento do pacote pela rede; a linguagem P4 abstrai as funções da camada física de rede.

4.2.5 Estado Reroteando

Assim que o pacote é atribuído ao estado de reroteamento será realizado um processo similar ao de roteamento, entretanto com consulta a uma tabela de roteamento secundária (*backup*). A tabela de roteamento secundária é definida previamente e pode ser ajustada sempre que necessário pelo plano de controle.

Algoritmo 8: InFaRR - Reroteamento

Chave IP Destino

```
1  tableipv4_backup {
2     key = {
3        hdr.ipv4.srcAddr: lpm;
4     }
5     actions = {
6        ipv4_forward_backup;
7        drop;
8     }
9     size = 1024;
10    default_action = drop();
11 }
```

Algoritmo 9: InFaRR - Sub-função

actions Secundário

```
1 typedef bit <9> egressSpec_t;
2 typedef bit <48> macAddr_t;
3 action
    ipv4_forward_backup( macAddr_t dstAddr ,
    egressSpec_t port ) {
4    standard_metadata.egress_spec = port;
5    hdr.ethernet.dstAddr = dstAddr;
6 }
```

[8] Conforme descrito na Seção 2.4, o mecanismo de controle sobre o *status* físico *UP/DOWN* das portas não é escopo deste trabalho.

A estratégia de resiliência da rede definirá quais *hosts* ou sub-redes possuirão mecanismo de recuperação a falhas, em que diferentes estratégias poderão ser aplicadas aos diferentes tipos de fluxos e necessidades de qualidade de serviço. Um fluxo prioritário poderá ser encaminhado através de uma porta que dá acesso a um enlace *backup* dedicado; os fluxos de menor prioridade poderão ser encaminhados a uma porta com recursos compartilhados e um fluxo sem prioridade não necessitaria ter uma rota de proteção configurada.

A tabela de roteamento secundária pode ser configurada de acordo com o tipo de chave de pesquisa a ser utilizado. O Algoritmo 8 descreve a pesquisa a uma tabela de roteamento cuja chave de pesquisa é o endereço IP destino. Destaca-se que as mesmas estruturas de pesquisa e de ações são utilizadas por essa tabela, cabendo a única alteração do nome da tabela e do nome da sub-função a ser utilizada.

A sub-função "*ipv4_forward_backup*", descrita no Algoritmo 9, é similar à sub-função utilizada durante o roteamento, tendo como diferença a realização de menos operações nesta fase. As operações de ajuste do campo endereço MAC de origem e decremento do campo TTL, foram realizadas durante o roteamento e, desse modo, não precisam ser realizadas novamente. Como a consulta à tabela de roteamento secundária estabelece um caminho alternativo para a proposta de roteamento, é necessário realizar as operações para atualizar a porta de saída (*egress*) e o respectivo endereço MAC de destino. Existem 4 (quatro) transições possíveis para o próximo estado que são escolhidas de acordo com o *status* físico[8] da porta de saída:

- Porta de saída com status OK: a transição ocorrerá para o estado Deparser.
- Porta de saída com status DOWN: a transição ocorrerá para o estado Mecanismo de *Pushback*.
- Porta de saída não especifica: em casos em que não ocorrer *match* na tabela de roteamento secundário, a transição ocorrerá para o estado de DROP.
- Porta de saída em *loop*: em casos em que a porta de saída é a mesma que a porta de entrada, a transição ocorrerá para o estado de Mecanismo de *Pushback*.

No estado de reroteamento não se faz necessária a gestão do estado lógico da porta por parte do fluxo, já que uma vez que não seja possível o encaminhamento pela porta recomendada, ocorrerá um *Pushback* e consequentemente o Mecanismo de Reconhecimento e Restauração do comutador antecessor evitará que pacotes deste fluxos cheguem novamente a este comutador.

4.2.6 Estado Drop

Os pacotes são transacionados para o estado de aceitação *DROP* quando não existem planos de roteamento ou reroteamento configurados para o encaminhamento do pacote. Uma vez que o algoritmo se encontre neste estado, o tratamento do pacote é encerrado (descartado).

4.2.7 Estado Mecanismo de Reconhecimento e Restauração

O estado Mecanismo de Reconhecimento e Restauração é transacionado quando o estado Roteamento detecta um *loop*. Nesta etapa, o InFaRR armazenará que a porta de saída está inativa para este fluxo. Desta forma, o Mecanismo de Reconhecimento e Restauração possibilita que os pacotes subsequentes deste fluxo não sejam encaminhados por esta porta. Também será armazenada a hora em que o status do fluxo foi alterado para futura utilização pelo Mecanismo de Retorno à Rota Principal.

Algoritmo 10: InFaRR - Mecanismo de Reconhecimento e Restauração

```
1  if(hdr.ipv4.protocol == TYPE_ICMP){
2    hash(var_hash_fluxo_porta_saida , HashAlgorithm.crc32, (bit <32>) 0 , {hdr.ipv4.srcAddr , hdr.ipv4.dstAddr
     , standard_metadata.egress_spec} , (bit <32>) NUMERO_FLUXOS);
3  } else if(hdr.ipv4.protocol == TYPE_TCP){
4    hash(var_hash_fluxo_porta_saida,HashAlgorithm.crc32, (bit <32>) 0, {hdr.ipv4.srcAddr , hdr.ipv4.dstAddr ,
     hdr.tcp.srcPort , hdr.tcp.dstPort , standard_metadata.egress_spec} , (bit <32>) NUMERO_FLUXOS);
5  } else if(hdr.ipv4.protocol == TYPE_UDP){
6    hash(var_hash_fluxo_porta_saida,HashAlgorithm.crc32, (bit <32>) 0, {hdr.ipv4.srcAddr , hdr.ipv4.dstAddr ,
     hdr.udp.srcPort , hdr.udp.dstPort , standard_metadata.egress_spec} , (bit <32>) NUMERO_FLUXOS);
7  } if(standard_metadata.egress_spec == standard_metadata.ingress_port) {          ▷ verifica loop
8    fluxo_status.write((bit <32>)var_hash_fluxo_porta_saida , PORTA_DOWN);
9    fluxo_tempo.write((bit <32>)var_hash_fluxo_porta_saida , standard_metadata.ingress_global_timestamp);
10 }
```

O Algoritmo 10 demonstra a codificação do Mecanismo de Reconhecimento e Restauração. A primeira fase deste processo é o cálculo da variável "*var_hash_fluxo_porta_saida*", que armazenará o *hash* dos campos necessários para identificação do fluxo (destacados em azul), acrescido da porta de saída (linhas 3 a 6). Cada tipo de cabeçalho TCP, UDP ou ICMP, possui uma estrutura de campos diferentes para o cálculo do *hash* que será utilizado para indexar os registradores. Na linha 7, o código fonte do InFaRR realiza a checagem do *loop*. A variável "*standard_metadata.egress_spec*"; contém o número da porta de saída, informado durante a consulta à tabela de roteamento, enquanto que a variável "*standard_metadata.ingress_port*", é preenchida automaticamente quando o Parser aceita o pacote e contém o número da porta de entrada. As próximas duas ações descritas se referem à gravação da informações nos registradores do comutador tomando como índice a variável "*var_hash_fluxo_porta_saida*" para que possam ser consultadas por outros pacotes futuramente. O registrador "*fluxo_status*" armazenará o status do fluxo para esta porta como DOWN (linha 8), e o registrador "*fluxo_tempo*" armazenará o horário (*timestamp*) em que a porta foi colocada com o *status* DOWN (Linha 9).

Após concluídas todas as etapas descritas, a máquina de estados finita será transicionada para o estado Reroteando.

4.2.8 Estado Mecanismo de Retorno à Rota Principal

Quando o fluxo está marcado pelo Mecanismo de Reconhecimento e Restauração para não utilizar a porta de saída ele é transacionado para o Mecanismo de Retorno à Rota Principal antes de ser transacionado para o Reroteamento.

Algoritmo 11: InFaRR - Mecanismo de Retorno à Rota Principal - Envia Clone

```
1  fluxo_tempo.read(var_fluxo_tempo , (bit<32>)var_hash_fluxo_porta_saida);
2  clone_valida_fluxo.read(var_clone_valida_fluxo , (bit<32>) var_hash_fluxo_porta_saida);
3  if((var_fluxo_tempo + TIME_OUT_RECUPERA_FLUXO) <standard_metadata.ingress_global_timestamp) {
4     fluxo_pivo_saida.read (var_fluxo_pivo_saida , (bit<32>) var_hash_fluxo_porta_saida);
5     fif((var_fluxo_pivo_saida == PIVO_FRR) && (var_status_porta_saida == PORTA_UP) &&
      (var_clone_valida_fluxo == NAO_REVALIDA)){
6        clone_valida_fluxo.write((bit<32>) var_hash_fluxo_porta_saida,REVALIDA);
7        clone(CloneType.I2E,(bit<32>) standard_metadata.egress_spec);
8     } else if((var_fluxo_pivo_saida != PIVO_FRR) && (var_status_porta_saida == PORTA_UP)){
9        var_status_fluxo_saida = PORTA_UP;
10       fluxo_status.write((bit<32>) var_hash_fluxo_porta_saida,PORTA_UP);
11    }
12 }
```

O estado de Mecanismo de Retorno à Rota Principal pode ser dividido em duas funções: 1) realizar periodicamente o envio dos pacotes de verificação, esta ação duplica um pacote de dados e o envia para a rota principal, além de enviar o pacote original via rota secundária. 2) validar o resultado do pacote de verificação. Caso o pacote duplicado não retorne dentro do período pré-estabelecido, por exemplo um *Round Trip Time* (RTT) como explicado anteriormente, o InFaRR assume que a rota principal foi restabelecida. A partir deste momento, os pacotes poderão ser enviados pela rota principal. Caso o pacote retorne, conclui-se que a falha ainda persiste.

A codificação do trecho responsável por envios periódicos de pacotes de verificação é demonstrada pelo Algoritmo 11. Por meio da consulta ao registrador "*fluxo_tempo*" e "*clone_valida_fluxo*" nas linhas 1 e 2 é possível identificar o horário em que o fluxo foi colocado no modo logicamente DOWN e se o comutador esta no modo de monitoração de resultados.

Na próxima linha, verifica-se se é o momento de realizar o teste de recuperação ao se comparar a hora atual com a hora em que o fluxo foi colocado no modo logicamente DOWN, somada ao tempo predefinido de intervalo de recuperação e se ele não está no modo validação ativado. Para os casos positivos, na linha 5 realiza-se uma consulta ao registrador "*fluxo_pivo*". Caso o comutador seja um PIVO, é realizada a escrita no registrador "*clone_valida_fluxo*", para que a função de monitoração do pacote de verificação seja iniciada e outros pacotes clone não sejam criados (linha 7). Na linha 6, ocorre a duplicação do pacote (clone). Para os casos negativos (*else*) da condição avaliada

na linha 5, quando o comutador não é um PIVO, verifica-se se a porta de encaminhamento está UP (linha 8). Em caso positivo, é possível alterar o *status* do fluxo para UP (linha 10).

É importante destacar que os comutadores que não são PIVO não recebem pacotes até que o Mecanismo de Retorno à Rota Principal duplique um pacote. Desta forma, quando o pacote chega ao comutador o tempo de recuperação já expirou e o pacote poderá ser encaminhado ao próximo comutador.

A função que valida o resultado de verificação do Mecanismo de Retorno à Rota Principal é codificada e apresentada pelo Algoritmo 12. Caso o registrador "*clone_valida_fluxo*" associado ao fluxo do pacote em processamento esteja ativo (linha 3), será verificado se o pacote está em condição de *loop* (linha 4). Caso a condição de *loop* seja confirmada o fluxo ainda permanecerá na condição mapeada pelo Mecanismo de Reconhecimento e Restauração, pois o pacote duplicado (clonado) percorreu o caminho até a falha e retornou até o comutador PIVO. As ações para este caso são: atualizar o registrador fluxo_tempo com a hora atual (linha 5), retirar o fluxo do modo de verificação (linha 6) e transacionar o pacote para o estado DROP (linha 7). Caso a condição de *loop* (*else*) não aconteça, será verificado se o tempo pré-estabelecido de monitoração esgotou (linha 9); em casos positivos os registradores que ativam o Mecanismo de Retorno à Rota Principal serão desativados e o pacote transacionado para o estado Deparser.

Algoritmo 12: InFaRR - Mecanismo de Retorno à Rota Principal - Reativa porta

```
1  fluxo_tempo.read(var_fluxo_tempo , (bit<32>) var_hash_fluxo_porta_saida);
2  clone_valida_fluxo.read(var_clone_valida_fluxo , (bit<32>) var_hash_fluxo_porta_saida);
3  if(var_clone_valida_fluxo == REVALIDA) {
4     if(standard_metadata.egress_spec == standard_metadata.ingress_port) {
5        fluxo_tempo.write((bit<32>) var_hash_fluxo_porta_saida ,
     standard_metadata.ingress_global_timestamp);
6        clone_valida_fluxo.write((bit<32>) var_hash_fluxo_porta_saida , NAO_REVALIDA);
7        drop();
8     } else
9     if((var_fluxo_tempo + TIME_OUT_RECUPERA_FLUXO + TIME_OUT_VALIDA_CLONE) <
     standard_metadata.ingress_global_timestamp) {
10       var_status_fluxo_saida = PORTA_UP;
11       fluxo_status.write((bit<32>) var_hash_fluxo_porta_saida , PORTA_UP);
12       clone_valida_fluxo.write((bit<32>) var_hash_fluxo_porta_saida , NAO_REVALIDA);
13       fluxo_pivo_saida.write((bit<32>) var_hash_fluxo_porta_saida , NAO);
14    }
15 }
```

Existem 4 (quatro) transições possíveis para o próximo estado que são escolhidas de acordo com o *status* físico[8] da porta de saída:

- Não está na hora de recuperar o pacote: a transição ocorrerá para o estado Reroteando.
- Quando o clone do pacote é gerado, a transição ocorrerá para o estado Deparser e Reroteando simultaneamente.

- Caso o fluxo seja recuperado, a transição ocorrerá para o estado de Deparser;
- Caso de pacote duplicado, ocorre a transição para o estado *Drop*.

4.2.9 Estado Mecanismo de *Pushback*

O Mecanismo de *Pushback* é transacionado quando o estado de Reroteamento não pode encaminhar pacotes por causa da porta DOWN ou em situações de *loop*. Diante desta condição o comutador enviará o pacote ao comutador antecessor através da porta mapeada no estado de Preparação.

Algoritmo 13: InFaRR - Mecanismo de *Pushback*

```
1 if ((var_status_porta_saida == PORTA_DOWN) || (standard_metadata.egress_spec ==
    standard_metadata.ingress_port)) {                    ▷ verifica status da porta e loop
2   standard_metadata.egress_spec = (bit <9>) var_fluxo_porta_entrada;
3   fluxo_mac_entrada.read(hdr.ethernet.dstAddr , (bit <32>) var_hash_fluxo_porta_entrada);
4   fluxo_pivo_saida.write((bit <32>) var_hash_fluxo_porta_saida , NAO);
5 }
```

O Algoritmo 13 demonstra o código fonte das funções implementadas pelo Mecanismo de *Pushback*. A linha 1 do código se refere à verificação sobre o *status* da porta ou *loop* para início do *Pushback*. Na linha 2, é ajustada a porta de saída com a informação da variável "*var_fluxo_porta_entrada*", criada na fase de recuperação, enquanto que, na linha 3 é ajustado o endereço MAC destino.

Após todas as etapas descritas concluídas, a máquina de estados finita será transacionada para o estado de aceitação Deparser.

4.3 Topologias e usabilidade

Nos *datacenters*, os comutadores normalmente são organizados em topologias hierárquicas e podem ser classificadas como de núcleo, agregação ou topo de rede, conforme descritos na Seção 2.6. A organização dos *PODs* (*Point of Delivery*) segrega a rede diante do interesse de tráfego e processamento entre os *hosts*. Uma característica tipicamente encontrada em outros algoritmos se refere ao fato de que o tráfego com destino a um *POD* utilize outros *PODs* durante o processo de recuperação (LIU et al., 2013b).

O InFaRR não utiliza outros *PODs* para realizar o reroteamento, visto que os comutadores do Núcleo estão configurados para executarem o Mecanismo de *Pushback* quando necessitarem recorrer ao reroteamento. Essa estratégia possibilita a recuperação sem a utilização de outros *PODs* em uma topologia *Fat-tree*. Para exemplificar esta característica, a Figura 22 descreve a comunicação entre os *hosts* finais H1P1 e H1P2. O comutador S1CORE, diante da falha de conexão ao *POD*2, recorre ao Mecanismo de *Pushback* (linha em vermelho tracejada) que enviará o pacote ao seu antecessor. O comutador A1P1, ao receber o pacote por meio do Mecanismo de Prevenção ao *Loop*,

iniciará o reroateamento para S2CORE, e o Mecanismo de Reconhecimento e Restauração atuará para que o caminho alternativo (linhas em roxo) seja utilizado também pelos pacotes subsequentes.

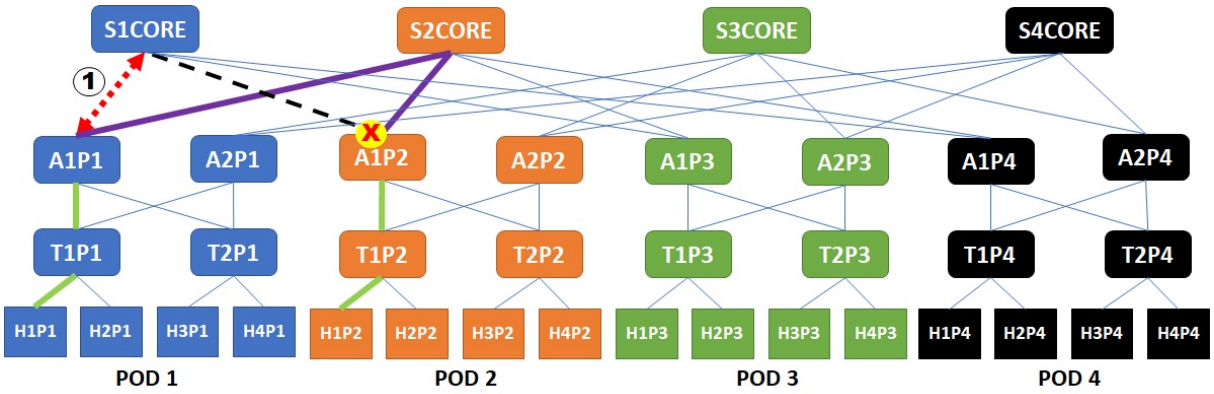


Figura 22 – Processo de Recuperação InFaRR - Cenário com 1 falha.

O algoritmo InFaRR foi idealizado para explorar até 3 falhas simultâneas em redes *Fat-Tree* com k=4. Outros tamanhos de rede como ilustrado na Figura 23, com k=6, podem suportar o aumento de *PODs*, comutadores e enlaces. Todavia, se faz necessário que as tabelas de roteamento sejam capazes de explorar as novas opções de encaminhamento entre os *PODs*. Na ausência de viabilidade de encaminhamento por meio da tabela de roteamento principal e da tabela de roteamento secundário, o algoritmo InFaRR realiza o *Pushback*. Embora existam outras opções de encaminhamento, elas não fazem parte das tabelas de roteamento. O comutador A1P1 possui 3 enlaces de acesso aos comutadores de núcleo, todavia pode utilizar somente 2.

Da mesma forma, o comutador T1P1 pode encaminhar pacotes somente através de A1P1 e A2P1, embora exista um enlace com A3P1. Assim, o algoritmo InFaRR possui o limite de recuperação de 3 falhas simultâneas ao acesso a outros *PODs* em uma rede hierárquica com três camadas. O crescimento de k não habilita maior capacidade de tolerância a falhas simultâneas no InFaRR, já que neste caso o limitador para o Mecanismo de *Pushback* é o número de camadas. Mas, em conjunto com o Plano de Controle pode se adaptar e explorar novas alternativas de roteamento. Uma alternativa para o aumento da resiliência e uso de mais enlaces entre os *PODs* seria a inclusão de uma tabela de roteamento terciária.

Assim que uma falha é detectada, os algoritmos de reroteamento rápido atuam de imediato, de forma a reduzir o impacto em fluxos que necessitam atender requisitos de qualidade de serviços, conforme descritos na Seção 2.2. O plano de controle se mantém como elemento chave de inteligência da rede, sendo capaz de atualizar os comutadores com planos de encaminhamento que ignoram os enlaces com falhas e assim configurar conexões que antes não eram possíveis por meio de comutadores com apenas duas tabelas de roteamento. Na ocorrência de uma falha, a tabela de roteamento secundária é utilizada

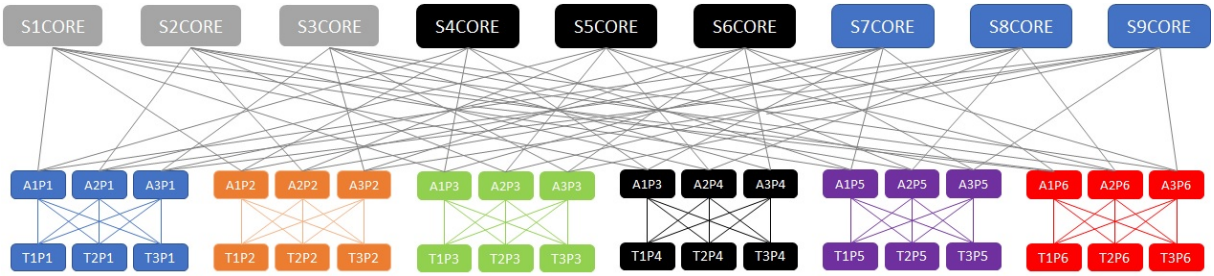
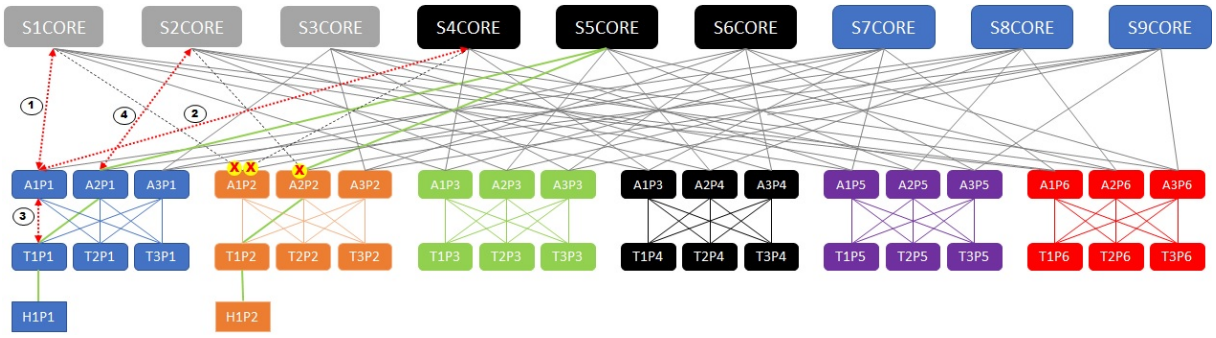


Figura 23 – Rede *Fat-Tree* com k=6.

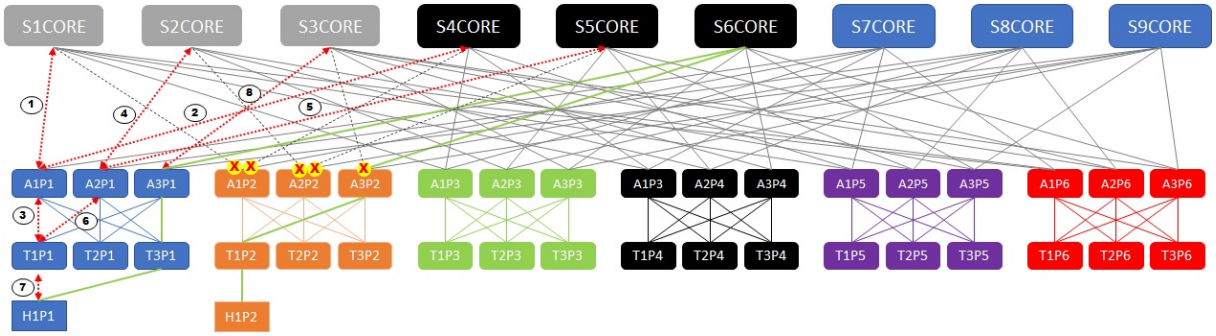
imediatamente pelo comutador, até que o Plano de Controle atualize a tabela de roteamento principal com uma rota em funcionamento. Esta atualização pode promover a rota contida na tabela de roteamento secundária para rota principal e inserir uma nova rota secundária ou pode oferecer uma nova rota à tabela de roteamento principal.

O uso de placas de redes programáveis é uma tendência atual em redes de datacenter e pode habilitar o uso do algoritmo InFaRR a partir do *host* (YAN et al., 2020). Um *host* com uma placa programável com duas ou mais interfaces poderia utilizar as características de reroteamento e habilitar novos caminhos de proteção para os fluxos prioritários estendendo a capacidade de roteamento até o *host* (SHAN et al., 2022). Um *host* com uma placa de rede programável, com duas portas conectadas a comutadores TOR diferentes, aumentaria a resiliência em uma topologia.

A Figura 24a mostra que o *host* H1P1 possui capacidade de contornar até 3 falhas simultâneas em uma rede *Fat-tree*, indepedente do tamanho de k, cenário que exploramos exaustivamente no Capítulo 5. A Figura 24b simula o *host* H1C1, com uma placa programável com duas portas conectadas a comutadores TOR diferentes de modo que, por meio de uma rede *Fat-Tree* com k=6, é possível contornar até 5 falhas simultâneas entre o *POD*1 (origem) e o *POD*2 (destino) com o uso do algoritmo InFaRR. Destaca-se a realização de 8 *pushbacks*, sendo que antes era possível somente 4.



(a) Recuperação de até 3 falhas simultâneas.



(b) Recuperação de até 5 falhas simultâneas, com uso de uma placa *smartnic*.

Figura 24 – *Smart Fat-tree* k=6 - Tolerância falhas.

5 Experimentação e Avaliação

O algoritmo proposto, conforme descrito na Seção 4.2, foi implementado em linguagem P4. Para estudo e análise de viabilidade, foi criado um ambiente Mininet com suporte ao *software switch* Bmv2[1]. Os códigos fontes dos algoritmos apresentados nesta dissertação podem ser encontrados em repositório público[2].

Este capítulo está dividido nas seguintes seções: 1) cenários avaliados em que será descrito como os testes foram realizados; 2) como os algoritmos estado da arte foram adaptados para comparação com o algoritmo InFaRR; 3) resumo da parametrização dos testes; 4) resultados obtidos diante da topologia *Standard Fat-tree*; e 5) resultados obtidos diante da topologia *AB Fat-tree*.

5.1 Cenários Avaliados

O experimento foi executado nas topologias datacenter *Standard Fat-Tree* e *AB Fat-Tree* com k=4, de modo que as particularidades de cada topologia foram descritas na Seção 2.6. Com o objetivo de avaliar o mecanismo de recuperação da rede em diferentes cenários de falhas entre os *hosts* de *PODs* distintos, simulou-se fluxos com diferentes tamanhos de pacotes, para mensurar o impacto das perdas de pacotes versus o tamanho do pacote. Avaliou-se os diferentes tamanhos de *payload* nos pacotes na camada IP[3]: 40, 300, 500, 1.000 e 1.500 bytes. Cada execução foi realizada de maneira independente para cada um dos quatro algoritmos mencionados na Seção 5.2. Conforme descrito na Sessão 4.3, o algoritmo InFaRR possui habilidade de se recuperar até três falhas, assim todos os algoritmos foram submetidos aos seguintes cenários:

1. **Cenário sem falhas:** para criação de um *baseline* e validação de conectividade entre os elementos da rede, realizou-se um teste em um ambiente sem falhas.
2. **Cenário com 1 falha[4]:** simulou-se uma falha no enlace principal do *POD* de destino.
3. **Cenário com 2 falhas:** simulou-se uma falha no enlace principal e outra no enlace secundário ao *POD* de destino.
4. **Cenário com 3 falhas:** simularam-se falhas em três enlaces sequenciais no processo de roteamento (principal, secundário e terciário) do *POD* de destino.

[1] https://github.com/p4lang/behavioral-model.
[2] https://github.com/dcomp-leris/InFaRR
[3] Refere-se ao tamanho do campo “IP.len”.
[4] As falhas sempre ocorrem entre os equipamentos de Núcleo e Agregação do *POD* de destino.

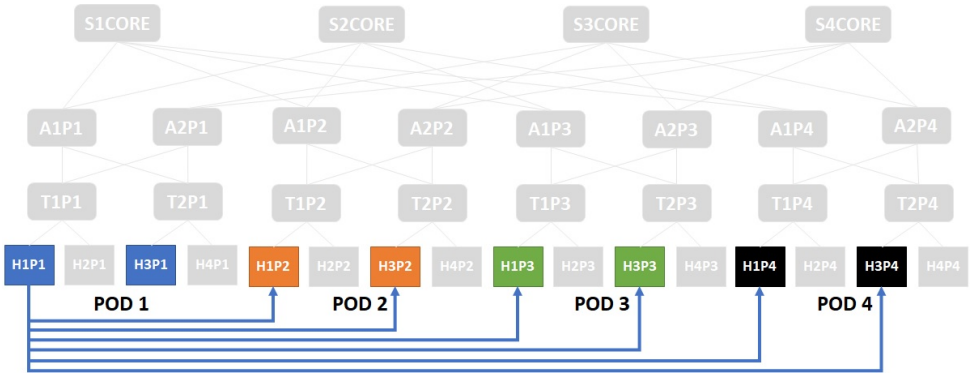


Figura 25 – Cenário de Teste com origem em H1P1 aos demais *hosts*.

Não foram considerados testes de *hosts* do mesmo *POD* e, para otimização da experimentação, consideramos apenas um *host* de cada comutador TOR, já que todos os *hosts* conectados ao mesmo TOR possuem a mesma tabela de roteamento. Assim, foram escolhidos 2 *hosts* de cada *POD*, cada um conectado a um TOR diferente, totalizando 8 *hosts*. Cada *host* fará uma conexão a outros 6 *hosts* vizinhos de outros *PODs*, totalizando 48 testes por bateria (8 *hosts* origem x 6 *hosts* destino).

A Figura 25 demonstra que o *host* H1P1 fará testes se conectando a 6 *hosts*, conforme linhas em azul. Assim, seguindo este mesmo conceito, todos os *hosts* selecionados são tomados como origem e farão conexão aos *hosts* nos outros *PODs*.

5.1.1 Detalhamento dos Testes Realizados

Cada teste foi estruturado para coletar informações (*(logs)* nas diferentes etapas existentes, conforme apresentado na Figura 26. Para realização dos testes optamos arbitrariamente pelo envio de 750 pacotes entre um *host* origem e um *host* destino. Entendemos que este número é suficiente grande para gerar dados estatísticos e não impactar o tempo de execução de cada teste.

Detalhando o envio dos 750 pacotes, na etapa inicial de cada cenário de experimentação $E0$, são gerados 249 pacotes que seguem pelo caminho principal até o destino. Este primeiro lote gera informações sobre o estado normal de funcionamento da rede. As falhas ocorrem no 250º pacote (etapa $E1$); quando simulamos que as portas ficam *DOWN*, o pacote 250º é incapaz de ser roteado por meio da porta sugerida pela tabela de roteamento principal, assim o algoritmo de recuperação é imediatamente acionado (etapa $E2$). A monitoração do pacote 250 é uma importante métrica para discernirmos se o algoritmo foi capaz de contornar as falhas sem perder pacotes. Durante a etapa $E3$, o algoritmo de reroteamento já conseguiu contornar a falha e os pacotes são encaminhados pelo caminho secundário; nesta etapa, não são mais esperadas perdas de pacotes e pode-se avaliar o quanto a rede foi degradada pelo uso do caminho secundário, seja pelo aumento do número de saltos, seja pelo aumento do *IP Packet Transfer Delay* (IPTD) ou do *IP Delay Variation*

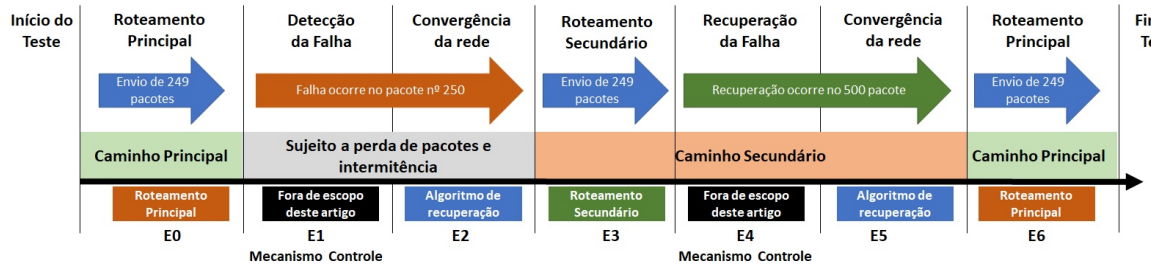


Figura 26 – Cenário de Teste com origem em H1P1 aos demais *hosts*.

(IPDV). A recuperação da falha acontece durante o 500º pacote, caracterizando a etapa $E4$. Na etapa $E5$, avaliamos a habilidade do algoritmo de voltar ao caminho principal; os pacotes subsequentes trafegam normalmente pelo caminho principal e pertencem à etapa $E6$. O processo de experimentação termina após o 750º pacote.

A partir da análise dos resultados obtidos, foi possível obter as seguintes medições: 1) perda de pacotes durante o processo de recuperação; 2) IPTD; 3) número de saltos (TTL) durante as diferentes etapas; e 4) IPDV. A perda de pacotes foi obtida por meio da diferença entre o número de pacotes enviados e recebidos. O IPDT (ver Tabela 1) foi computado através da diferença entre o instante de tempo (*timestamp*) em que pacote entra na rede, e o instante de tempo (*timestamp*) em que o pacote sai da rede. Já o número de saltos foi medido a partir do campo *Time to Live* (TTL) do cabeçalho IPv4. O IPDV é calculado pela diferenças da média do IPTD, da fase em que os pacotes são encaminhados sem falhas e da fase em que os pacotes necessitam contornar falhas.

Para o cálculo do IPTD, criamos uma ferramenta em *Python* utilizando as bibliotecas *Scapy* e *Time*. Esta ferramenta é composta por um programa para envio de pacotes e outro com funcionalidade de recebimento. O Algoritmo 14 é utilizado no *host* origem e está programado para o envio de 750 pacotes (linha 3), possuindo como principal característica o envio do *timestamp* dentro do *paylod* do pacote (linha 6), o tempo de processamento para inclusão desta informação é relativa, pois todos os pacotes possuíram esta informação. Já o *host* destino, fará uso do Algoritmo 15, que interpreta no *payload* do pacote o *timestamp* gerado durante a criação do pacote. O IPTD é resultado do cálculo da diferença do *timestamp* do *paylod* e o seu *time* atual. Como ambos os *hosts* são executados na mesma maquina virtual, o *time* entre as máquinas estará sincronizado, sendo possível medir com precisão o tempo que o pacote levou para atravessar a rede.

A Figura 27 representa o cenário de experimentação utilizado para avaliação deste trabalho. Cada teste consiste em iniciar o ambiente virtual que simula a rede *Fat-Tree* com k=4 em *Mininet* (passo 1), e carregar o respectivo algoritmo P4 nos comutadores referentes ao teste que será avaliado (passo 2). Assim que o ambiente está pronto, inicia-se nos *hosts* origem e destino a ferramenta de cálculo de IPDT (passo 3). As informações de cada pacote são armazenadas em arquivo de log para análise futura. O teste termina com a

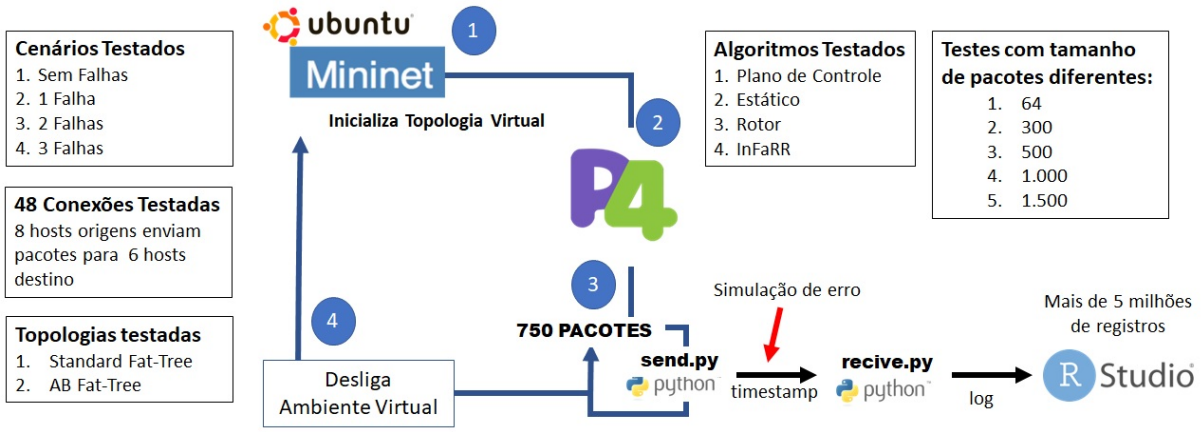


Figura 27 – Ambiente de Testes, cenários avaliados.

desativação do ambiente virtual e a limpeza das variáveis (passo 4). Os dados coletados em todas as baterias de testes e cenários foram analisados por meio da plataforma *R Studio*.

Algoritmo 14: Programa de envio de pacotes - send.py

```
1  import time
2  import scapy
3  while ( i< 750 ):
4     nSeq = nSeq +1
5     pkt = Ether( src = get_if_hwaddr( iface ), dst =
   'ff:ff:ff:ff:ff:ff' )
6     pkt = pkt / IP( dst=addr ) / TCP( dport=1234,
   sport=54321, seq=nSeq) / "{ :.11f }".format(
   time.time( ) )
7     if len( pkt ) < ( int( tamanho_Pacote ) + 14 ):
8        myString = "\x00 "* ( int( tamanho_Pacote ) -
   len( pkt ) + 14)
9        pkt=pkt / myString
10    i = i + 1
11    sendp( pkt, iface=iface, verbose=False )
```

Algoritmo 15: Programa de recebimentos de pacotes - recive.py

```
1  import time
2  import scapy
3  def handle_pkt( pkt ):
4     if TCP in pkt and pkt[TCP].dport == 1234:
5        timeatual = float( ":.11f".format( time.time() ) )
6        timeorigem = float( pkt[ Raw ].load[ :22 ] )
7        deltatime = timeatual - timeorigem
8        newlog=[ "{ :.11f }".format( timeatual ) ,
   pkt[IP].src , pkt[IP].dst , pkt[IP].ttl , pkt[IP].len ,
   pkt[TCP].sport , pkt[TCP].dport , pkt[TCP].seq ,
   pkt[TCP].ack,"{ :.11f }".format( timeorigem ) , "{
   :.11f }".format( deltatime ) ]
9        for j in newlog:
10          file.write ( str( j ) )
11          file.write ( ";")
12       sys.stdout.flush ( )
13       if pkt [ TCP ].seq >= 750:
14          exit( 1 )
```

A realização dos testes em ambiente controlado e individualizado permitiu coordenar a ocorrência das falhas de forma a impactar diretamente a tabela de roteamento principal e nos enlaces utilizados pelo algoritmos de recuperação. Assim, foram obtidos resultados de todas as combinações possíveis válidas para este estudo. Esta abordagem se demonstrou mais promissora em relação à realização de testes simultâneos e de falhas randômicas para obtenção de registros de cenários com convergência da rede e perda de pacotes. Os testes individualizados permitiram a rastreabilidade e a interpretação das falhas ocorridas durante os processos de recuperação.

A Tabela 4 consolida os números relacionados à experimentação deste trabalho. Para cada pacote enviado pelo *host* origem, espera-se uma linha de *log* gerada pelo *host*

destino. Assim, a se considerar todos os cenários avaliados chegamos ao número de cinco milhões e setecentos mil registros (multiplicando todas variáveis envolvidas). Os pacotes enviados durante simulações de falhas das quais o algoritmo não se recuperou, não chegam ao *host* destino e assim não geraram *log* caracterizam perda de pacotes.

Algoritmos	**Topologias**	***Host* Origem**	***Host* destino**	**Falhas**	**Tamanhos**	**Pacotes**
4	2	6	8	4	5	750

Total de registros log: 5.760.000

Tabela 4 – Números dos experimentos realizados.

5.2 Adaptações aos algoritmos selecionados

Os algoritmos do estado da arte foram implementados, com algumas adaptações, em linguagem P4, para fins de comparação com a proposta deste estudo. Todos os algoritmos utilizam a mesma tabela de roteamento principal e utilizam como chave de pesquisa o endereço de IP destino.

O código fonte dos algoritmos adaptados estão disponibilizados na seguinte página do github: https://github.com/gvenancioluz/InFaRR2022.

5.2.1 Reroteamento Estático

O algoritmo de roteamento estático, descrito na Seção 3.2.1, consulta a tabela de roteamento principal e em caso de falha na porta de saída é realizada uma consulta à tabela de roteamento secundária (reroteamento). Na adaptação deste algoritmo em P4, a ocorrência de *loop* após a consulta da tabela de roteamento principal será também gatilho para inicio do reroteamento.

Considerando uma rede *Fat-Tree*, cada porta do comutador de núcleo está fisicamente conectada ao respectivo *POD*; a porta 1 está conectada ao *POD1*, a porta 2 está conectada ao *POD2* e assim sucessivamente. A tabela de roteamento principal foi construída para apontar a porta diretamente conectada ao *POD* como sua opção de encaminhamento.

Nos comutadores de núcleo, a tabela de roteamento secundária foi construída sistematicamente para apontar para a próxima porta que a utilizada pela tabela de roteamento principal, de modo que a porta de saída secundária para o *POD*1 será a porta 2, para o *POD*2 será a porta 3 e assim sucessivamente. Entretanto, a tabela de roteamento secundária também está sujeita a *loop*.

A Figura 28 descreve a ocorrência de uma falha entre S1CORE e A1P1. O pacote com origem em *host* H1P2 e destino ao *host* H1P1, quando processado pelo comutador

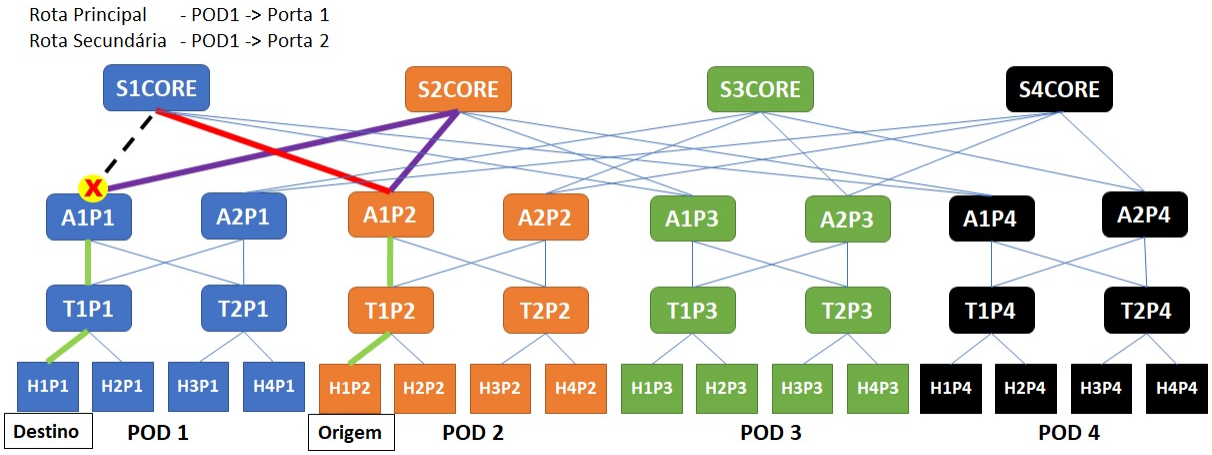


Figura 28 – *Roteamento Estático* - Desafio Tabela de Roteamento Secundária.

S1CORE, diante da falha até A1P1, necessita consultar a tabela de roteamento secundária que irá apontar para a porta de saída número 2 e assim o pacote é devolvido para A1P2, ocasionando um *loop*. O comutador A1P2, ao consultar a tabela de roteamento principal, percebe a ocorrência do *loop* e assim executa o reroteamento por meio de uma consulta a tabela de roteamento secundária e o pacote é encaminhado para S2CORE. Finalmente, o pacote pode ser encaminhado para o destino. Assim, a tabela de roteamento para este cenário será H1P2 -> T1P2 -> A1P2 -> S1CORE -> A1P2 -> S2CORE -> A1P1 -> T1P1 -> H1P1.

Na adaptação do algoritmo Estático optou-se por realizar o encaminhamento do pacote através da porta sugerida pela tabela de roteamento secundária, mesmo em situação de *loop*, ao invés de realizar o descarte do pacote, o que viabilizou a recuperação da falha descrita no parágrafo anterior.

5.2.2 Reroteamento Rotor

O algoritmo Rotor, descrito na Seção 3.2.2, se detectada a falha na porta de saída, procede com o reroteamento, sequencialmente, para a próxima porta disponível. Na adaptação realizada para este algoritmo, se detectado o envio de pacotes para porta de entrada em situação de *loop*, um novo reroteamento será realizado e o pacote será encaminhado para a próxima porta disponível. Sempre que o pacote precisa ser reroteado, ele é recirculado dentro do comutador, até que uma porta de encaminhamento viável seja encontrada.

5.2.3 Plano de Controle

O algoritmo que simulou o Plano de Controle centralizado, descrito na Seção 3.2.3, foi implementado em linguagem *Python* e funciona em um servidor segregado aos comutadores, realizando consultas periódicas (varredura) em cada um dos comutadores

P4. O objetivo deste algoritmo é detectar a ocorrência de falhas na rede, recalcular as rotas e atualizar as tabelas de roteamento nos comutadores. Neste caso, observou-se que o tempo médio para executar as consultas em todos os comutadores em nosso ambiente de experimentação, sequencialmente, variou entre 3 e 5 segundos. Sendo assim, para fins de padronização e coleta dos dados, acrescentamos um atraso no algoritmo, para que as consultas tenham um intervalo fixo de 5 segundos.

O trabalho de (CHAN et al., 2018) aborda o mecanismo de "Detecção de Falhas e Planejamento do Ciclo de Monitoramento" implementados no Plano de Controle. Neste trabalho foram utilizando 1,5 segundos como intervalo de cada ciclo, considerando outra topologia e quantidade de equipamentos. O intervalo de cada ciclo de detecção pode variar conforme a topologia rede, número de comutadores na rede e se as consultas ocorrem de forma sequencial ou em paralelo.

5.3 Parametrização para os Testes

Esta seção contém os parâmetros utilizados para os testes:

- Número de algoritmos testes: 4 (InFaRR, Estático, Rotor e Plano de Controle);
- Número de cenários testados: 4 (sem falhas, 1 falha, 2 falhas e 3 falhas);
- Número de topologias avaliadas: 2 (*Standard Fat-Tree* e *AB Fat-Tree*);
- Número de tamanhos[3] de pacotes: 5 (40, 300, 500, 1.000 e 1.500);
- Número de *hosts* origem testados por lote: 8;
- Número de *hosts* destino testados por lote: 6;
- Número total de testes por lote: 48;
- Número de pacotes enviados, etapa $E0$ Caminho Principal: 250;
- Número de pacotes enviados, etapa $E3$ Caminho Secundário: 250;
- Número de pacotes enviados, etapa $E6$ Caminho Principal: 250;
- Número de *PODs*: 4;
- Intervalo de monitoração do Plano de Controle: 5 segundos.
- Constante InFaRR, número de posições de registradores: 100.000;
- Constante InFaRR, intervalo para revalidação do enlace: 2,5 segundos;
- Constante InFaRR, intervalo espera retorno pacote duplicado: 1 segundo;

As constantes utilizadas pelo InFaRR foram definidas arbitrariamente para o funcionamento do algoritmo em laboratório. O número de posições de registradores deve ser dimensionado com tamanho suficiente para diminuir a probabilidade do número de colisões causadas pelo função *hash*, como discutido anteriormente. O intervalo para revalidação do enlace foi definido com a metade do tempo de intervalo de monitoração do Plano de controle, de forma a proporcionar uma redução significativa no tempo de recuperação por meio do Mecanismo de Retorno à Rota Principal. A constante que define o intervalo de espera do retorno do pacote duplicado foi definida e padronizada com 1 segundo, após a realização dos testes inicias onde mensuramos os valores. Este valor deve ser no mínimo equivalente ao RTT do pacote até o comutador com falha[5].

5.4 Resultados - Topologia *Standard Fat-Tree*

A Tabela 5 consolida os resultados obtidos para as execuções realizadas junto à topologia *Standard Fat-Tree*, podendo se observar que o InFaRR foi o único algoritmo a atender os critérios do ITU em todos os cenários.

Algoritmos /Cenário de Testes Topologia *Standard Fat-Tree*	**Plano de Controle**	**Estático**	**Rotor**	**InFaRR**
Cenário 1 Falha				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	7	7	7
N.º de saltos - Caminho redundante ($E3$)	5	7	7	5
Atende critérios ITU	Não	Sim	Sim	Sim
Cenário 2 Falhas				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	Não Recupera	Não Recupera	11
N.º de saltos - Caminho redundante ($E3$)	5	Não Recupera	Não Recupera	5
Atende critérios ITU	Não	Não	Não	Sim
Cenário 3 Falhas				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	Não Recupera	Não Recupera	13
N.º de saltos - Caminho redundante ($E3$)	5	Não Recupera	Não Recupera	5
Atende critérios ITU	Não	Não	Não	Sim

Tabela 5 – Avaliação consolidada dos algoritmos estudados em Topologias *Standard Fat-Tree*.

Como descrito conceitualmente na Seção 3.2.3, o algoritmo que utiliza o Plano de Controle apresenta um atraso na detecção da falha que ocasiona perda de pacotes, inviabilizando sua utilização para fluxos sensíveis a perda de pacotes. A Figura 33a demonstra a ocorrência da perda de pacotes durante o processo de recuperação usando o Plano de Controle. Observa-se ainda que, independente do número de falhas e do tamanho dos pacotes, ocorreram perdas de pacotes. A quantidade de perda de pacotes mensurada em cada teste, não gerou diferença estatística diante aos diferentes cenários testados. O

[5] A monitoração do tempo do RTT dos pacotes, embora seja possível sua implementação em P4, não é escopo do algoritmo InFaRR.

algoritmo InFaRR não apresentou perda de pacotes durante o processo de recuperação em nenhum cenário.

Os testes realizados focaram na habilidade de recuperação diante dos cenários com falhas. Não foram avaliados o desempenho dos algoritmos diante da carga máxima de transmissão da rede. O processo de envio de pacotes sequenciais em ambiente de rede controlado, sem outros tráfegos ocorrendo simultaneamente, promoveu um baixa variação do IPDV, de forma que o estudo do tempo de transmissão do fluxo (*Flow-Completion Time* - FCT) não agregaria valor a esta pesquisa.

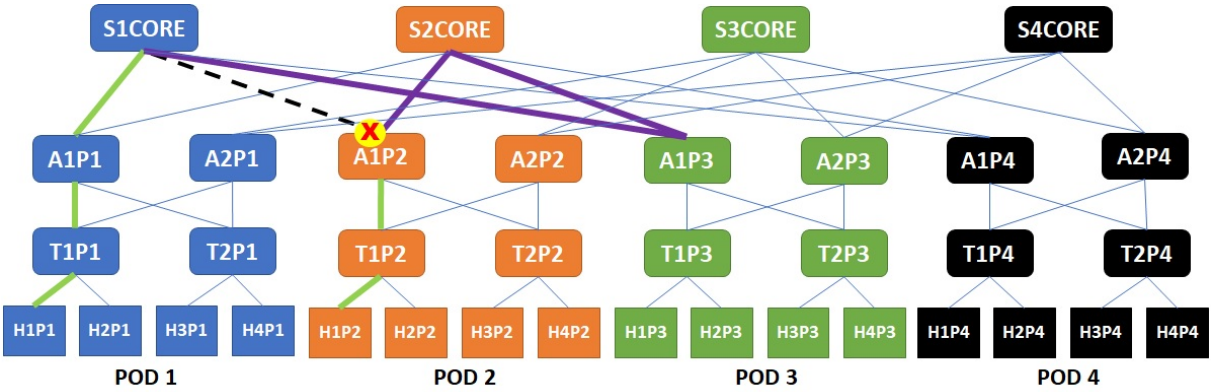


Figura 29 – *Standard Fat-tree* - Cenário com 1 falha - Algoritmos Estático e Rotor.

A Figura 29 representa a comunicação entre H1P1 e H1P2 cujo tabela de roteamento principal utiliza 5 comutadores (saltos), sendo eles: T1P1 -> A1P1 -> S1CORE -> A1P2 -> T1P2 (destacado pelos enlaces em verde). Esta figura representa o cenário com uma falha, em que o enlace tracejado entre S1CORE e A1P2 está interrompido. Para contornar a falha, os algoritmos Estático e Rotor realizam o reroteamento por meio do comutador A1P3 do *POD*3 e S2CORE, destacado em roxo. Esta estratégia de recuperação a falhas, utilizando um comutador em outro *POD*, penaliza o fluxo de dados que agora precisará ser processado por mais comutadores; neste exemplo, o número de saltos passa de 5 para 7.

O Plano de Controle, embora tenha desvantagens sobre o tempo de detecção e processamento das novas rotas, se comparado com soluções implementadas no plano de dados, como é o caso dos algoritmos de reroteamento rápido, sempre oferece uma tabela de roteamento otimizada e livre de *loop*. Assim, para contornar a falha apresentada na Figura 29, o plano de controle apresentaria uma tabela de roteamento com 5 saltos, utilizando os seguintes comutadores: T1P1 -> A1P1 -> S2CORE -> A1P2 -> T1P2. O algoritmo InFaRR, por meio dos mecanismos de recuperação, também oferece uma conexão otimizada, conforme apresentamos na Figura 30.

Os algoritmos de roteamento Estático e Rotor não concluíram a recuperação durante os cenários com 2 e 3 falhas, devido ao problema já descrito na Seção 2.6 sobre as redes *standard Fat-Tree* que podem ocasionar *loop*. A limitação para o funcionamento do reroteamento está associada às interconexões físicas propostas pela topologia e não à

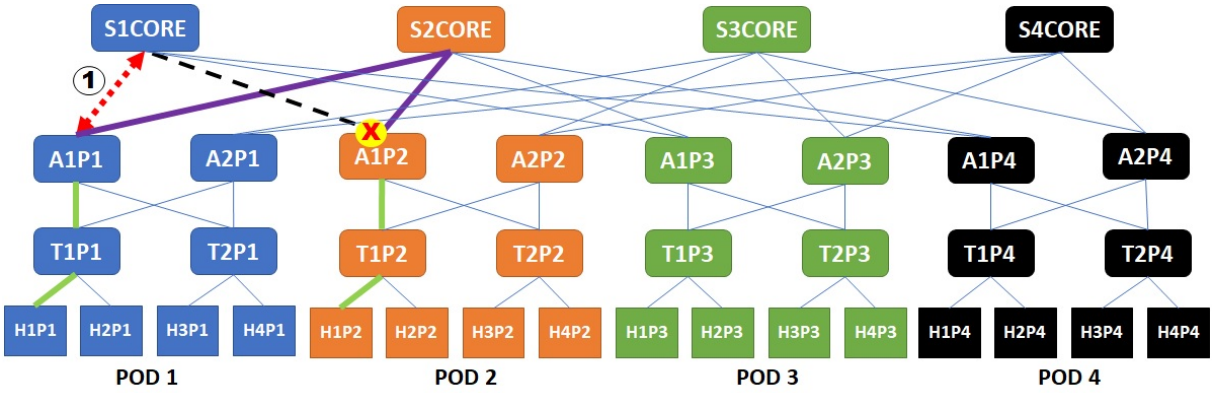


Figura 30 – Tabela de roteamento otimizada - InFaRR - Cenário com 1 falha.

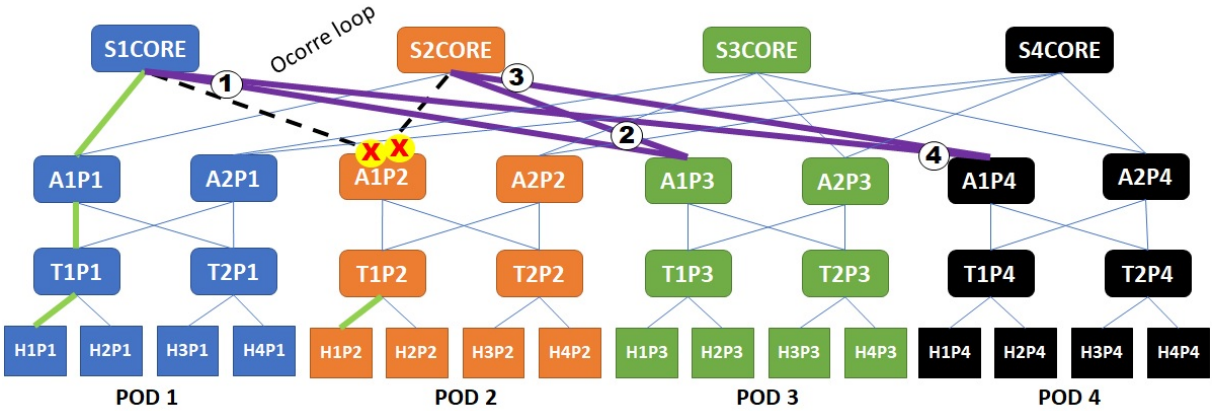


Figura 31 – *Standard Fat-tree* - Cenário com 2 falhas - Algoritmos Estático e Rotor.

política de encaminhamento lógico. A Figura 31 ilustra a conexão entre H1P1 e H1P2 e detalha o cenário com 2 falhas. Neste caso, a utilização do caminho alternativo, conforme sequência de enlaces numerados, acarreta um *loop* no dispositivo S1CORE tanto para o algoritmo Estático quanto para o Rotor.

Os resultados obtidos comprovaram a habilidade do InFaRR em concluir o processo de recuperação dentro dos parâmetros requeridos em relação a perda de pacotes, IPDT e IPDV, conforme descrito na Seção 2.2, em todos os cenários avaliados.

A opção pelo Mecanismo de *Pushback* como reroteamento nos comutadores do Núcleo previne a ocorrência do *loop*, conforme descrito na Capítulo 4, e possibilita a recuperação. A Figura 32 demonstra o Mecanismo de *Pushback* em funcionamento, de forma que o algoritmo InFaRR consegue contornar as duas falhas sequenciais. O comutador S1CORE, diante da falha do enlace que o interliga ao *POD*2, realiza o *Pushback* (passo 1); o reroteamento no comutador A1P1 enviará o pacote para S2CORE, que também está impossibilitado de enviar o pacote diretamente para o *POD*2 e, novamente, é realizado um *Pushback* (passo 2); o comutador A1P1, diante da inviabilidade da tabela de roteamento principal e secundário, realiza o envio do pacote para seu antecessor (passo 3). O reroteamento do comutador T1P1 leva o pacote por um caminho viável até H1P2, passando

pelos comutadores A2P1, S3CORE, A2P3 e T1P2 (destacado em roxo).

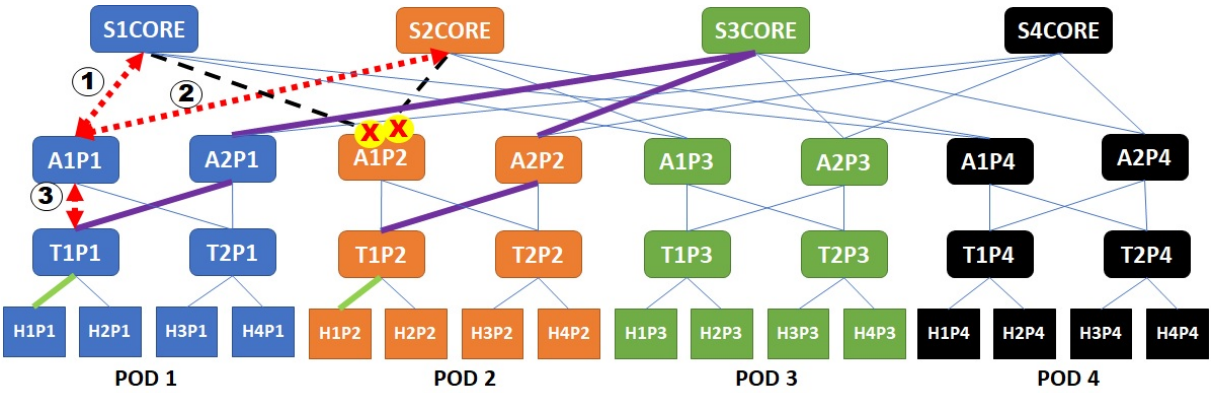
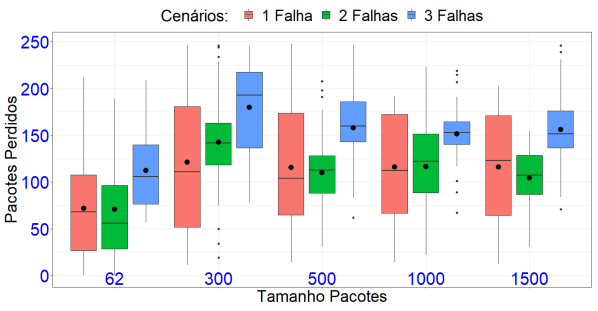


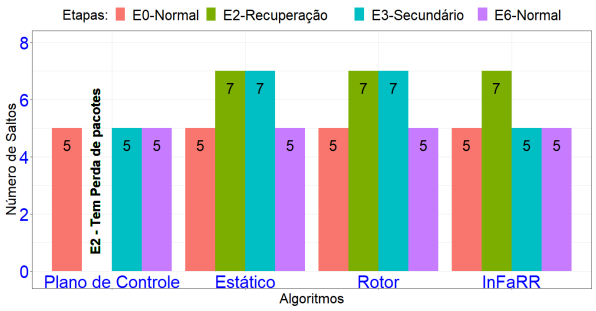
Figura 32 – *Standard Fat-tree* - Cenário com 2 falhas - InFaRR recuperação através do *Mecanismo de Pushback*.

A Figura 33b apresenta o número de saltos observados durante as etapas do processo de recuperação para o cenário de 1 falha. As etapas de recuperação estão descritas na Seção 5.1.1. As barras em vermelho demonstram a etapa sem falhas ($E0$). Em relação à etapa de recuperação, instante $E2$, observa-se que os algoritmos (Estático, Rotor e InFaRR) obtiveram 7 saltos. O algoritmo Plano de Controle por possuir um método de recuperação reativo, não pode encaminhar pacotes, até que uma atualização do Plano de Controle seja encaminhada a ele. Neste caso, conforme indicado na Figura 33b, existe perda de pacotes. A barra em azul representa o funcionamento no caminho redundante, instante $E3$, no qual o InFaRR possui 5 saltos, enquanto os algoritmos Estático e Rotor possuem 7. A barra em roxo, instante $E6$, representa o funcionamento normal (retorno à rota principal).

A Figura 34a apresenta os resultados do cenário com 1 falha para os algoritmos Estático e Rotor. O início do reroteamento ocorre na linha vermelha, instante $E1$, e finaliza na linha azul, instante $E4$. As etapas de recuperação (instante $E2$) e funcionamento pelo encaminhamento secundário (instante $E3$) ocorrem simultaneamente, visto que todos os pacotes passaram pela etapa $E2$ e possuem um aumento no IPDT e no número de saltos. A Figura 34b apresenta o tempo do IPTD durante a etapa $E2$, especificamente para o algoritmo InFaRR nos diferentes cenários de falhas. A etapa $E2$ consiste em buscar

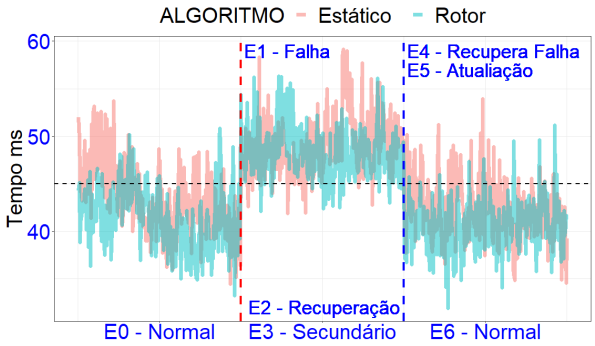


(a) Perda de pacotes - Plano de controle.

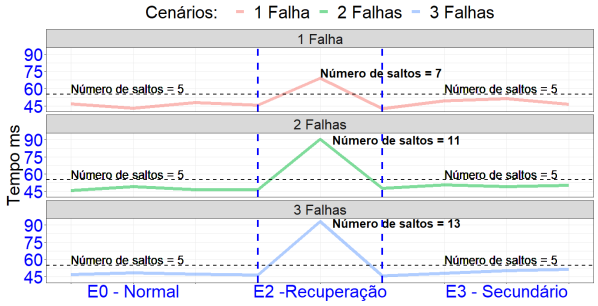


(b) Número de saltos por etapas. Cenário com 1 falha.

Figura 33 – *Standard Fat-tree* - Resultados Obtidos.



(a) IPTD - Algoritmo Estático e Rotor - 1 Falha.



(b) IPTD - Detalhe do Processo de Recuperação do algoritmo InFaRR.

Figura 34 – *Standard Fat-tree* - IPDT - Reroteamento Rápido.

um caminho alternativo até seu destino e, graças ao Mecanismo de Reconhecimento e Restauração, ocorre um aumento no número de saltos e do IPDT somente nesta etapa. É importante destacar que, no InFaRR, a etapa $E2$ é executada somente por um pacote, sendo que durante a etapa $E3$, os demais pacotes terão o mesmo número de saltos e IPDT equivalentes à etapa inicial $E0$.

O processo de retorno à rota principal associado à etapa $E5$ acontece imediatamente após o retorno da porta para o estado UP para os algoritmos Estático e Rotor, pois esses algoritmos atuam localmente nos comutadores em que a falha existia. Os algoritmos InFaRR e Plano de Controle promovem o contorno à falha otimizando a tabela de roteamento de forma que os pacotes não necessitam ir até o comutador com falhas. O InFaRR utiliza os mecanismos Pushback, Prevenção de Loop e Reconhecimento e Restauração para prover esta otimização enquanto que o Plano de Controle recalcula as rotas e envia atualizações para as tabelas de roteamento dos comutadores. O algoritmo InFaRR utiliza o Mecanismo de Retorno à Rota Principal para antecipar o uso da tabela de roteamento principal antes do recálculo das rotas pelo Plano de Controle. O Mecanismo de Retorno à Rota Principal, por utilizar checagem em intervalos de tempos predefinidos, não garante o retorno à rota principal imediatamente como os algoritmos Estático e Rotor.

5.5 Resultados - Topologia *AB Fat-Tree*

Esta segunda etapa de experimentação apresenta os resultados obtidos com a topologia *AB Fat-Tree*, a qual se caracteriza pelo remanejamento da conexão de alguns enlaces entre os comutadores de núcleo e os comutadores de agregação, conforme descrito na Seção 2.6. Foi realizada a mesma bateria de testes efetuadas com a topologia *Standard Fat-tree*.

O remanejamento das conexões exige que o algoritmo que simula falhas se adapte para simular falhas sequenciais. No experimento com a topologia *Standard Fat-Tree*, descrito na Figura 31, em que H1P1 se comunica com H1P2, observamos que ambos os

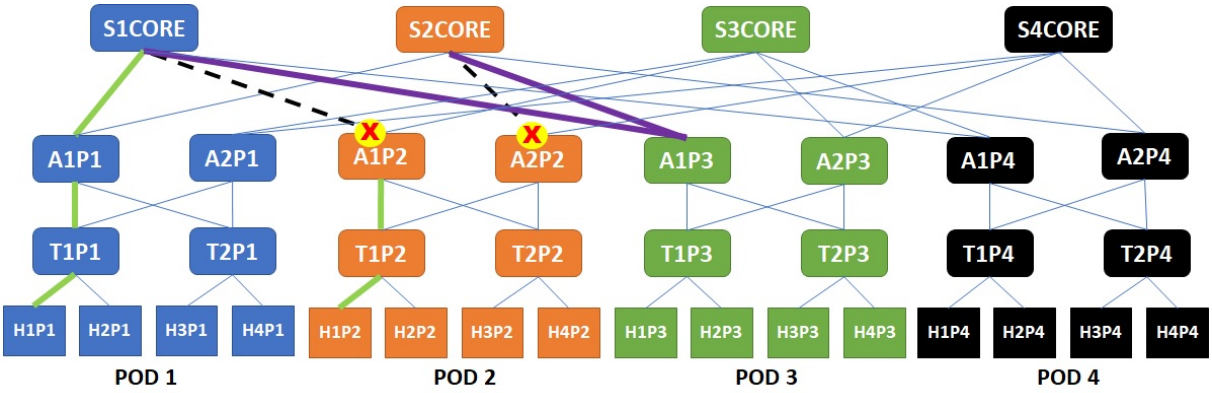


Figura 35 – *AB Fat-Tree* - Cenário 2 falhas - Algoritmos Estático em *loop*.

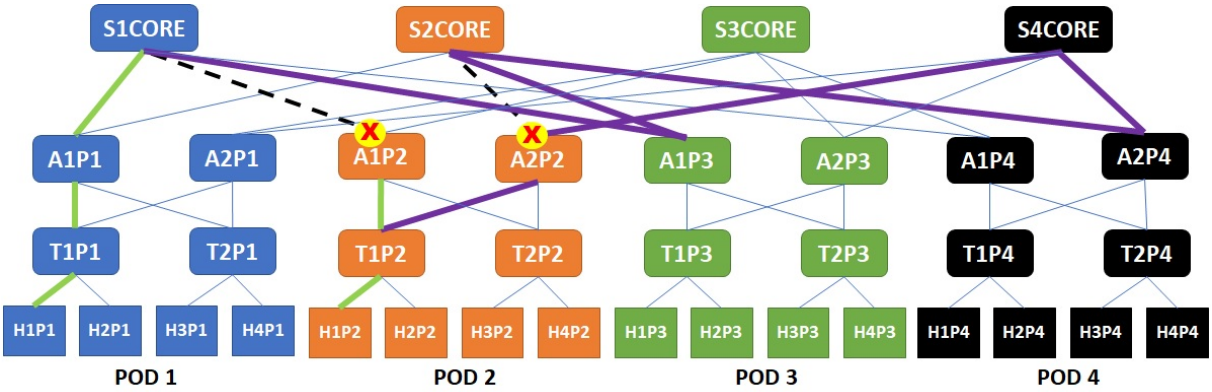


Figura 36 – *AB Fat-Tree* - Cenário com 2 falhas - Algoritmos Rotor.

enlaces com falhas chegam ao comutador A1P2. Para simularmos 2 falhas sequenciais para os mesmos *hosts* na topologia *AB Fat-Tree*, precisamos considerar uma falha no enlace com destino a A1P2, e outro enlace com destino a A2P2, conforme demonstrado na Figura 35.

Na comunicação de H1P1 com H1P2, em um cenário com 2 falhas, o uso do algoritmo Estático descrito na Figura 35, pode-se observar que o comutador S1CORE, impossibilitado de encaminhar para o *POD*2, faz o reroteamento do pacote através do *POD*3, que encaminha para o S2CORE. O comutador S2CORE não pode encaminhar os pacotes para A2P2, e o reroteamento sugere o envio do pacote pela porta de entrada, ocasionando um *loop*.

A topologia *AB FAT-Tree* viabilizou ao algoritmo Rotor condições de recuperação diante de um cenário com duas e três falhas, o que na topologia *Standard Fat-Tree* não existia. A Figura 36 demonstra o processo de recuperação do algoritmo Rotor diante de duas falhas consecutivas; a nova rota proposta, destacada em roxo, aumenta o número de saltos e necessita que os pacotes sejam reroteados em outros 2 *PODs*. A Figura 37 representa a recuperação através do algoritmo Rotor diante de três falhas consecutivas, em que o aumento do número de saltos é ainda maior: para contornar as falhas, ocorreu um aumento de 120%, passando de 5 santos para 11 saltos. A linha destacada em roxo

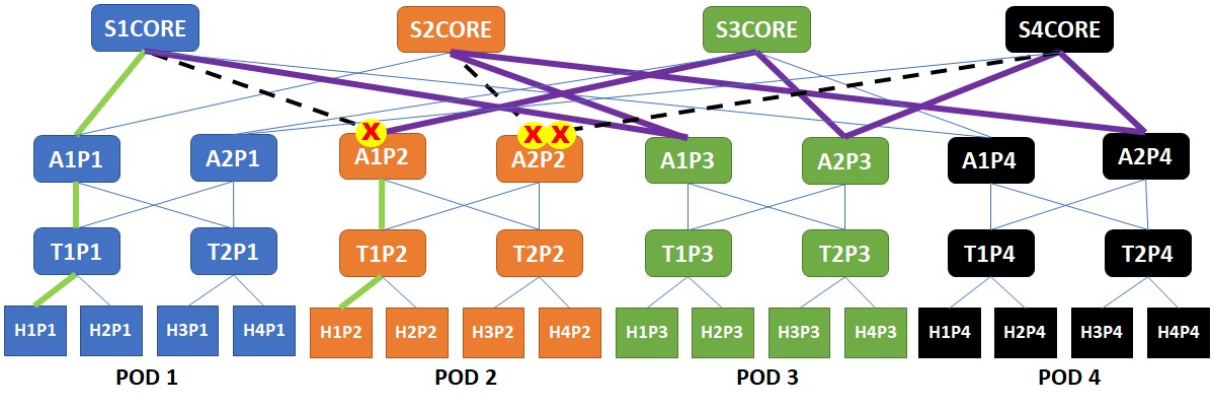


Figura 37 – *AB Fat-Tree* - Cenário com 3 falhas - Algoritmos Rotor.

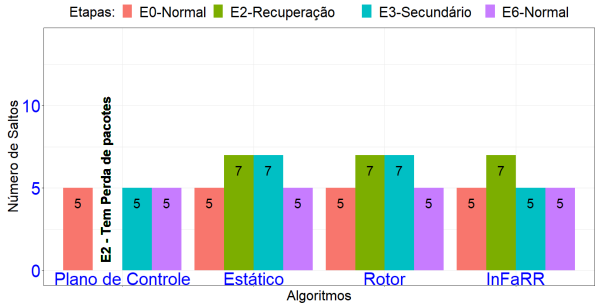
Algoritmos /Cenário de Testes Topologia *Standard Fat-Tree*	**Plano de Controle**	**Estático**	**Rotor**	**InFaRR**
Cenário 1 Falha				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	7	7	7
N.º de saltos - Caminho redundante ($E3$)	5	7	7	5
Atende critérios ITU	Não	Sim	Sim	Sim
Cenário 2 Falhas				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	Não Recupera	9	11
N.º de saltos - Caminho redundante ($E3$)	5	Não Recupera	9	5
Atende critérios ITU	Não	Não	Sim	Sim
Cenário 3 Falhas				
N.º de saltos - Caminho Normal ($E0$)	5	5	5	5
N.º de saltos - Processo de Recuperação ($E2$)	Perde Pacotes	Não Recupera	11	13
N.º de saltos - Caminho redundante ($E3$)	5	Não Recupera	11	5
Atende critérios ITU	Não	Não	Sim	Sim

Tabela 6 – Avaliação consolidada dos algoritmos estudados em Topologias *AB Fat-Tree*.

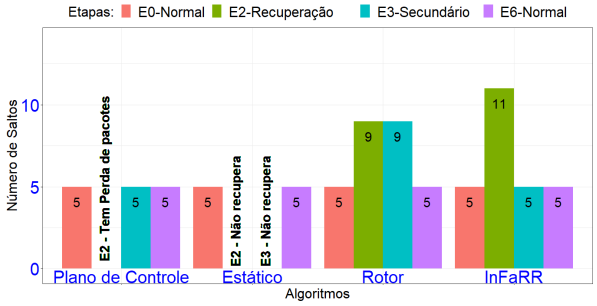
representa os saltos necessários durante o processo de recuperação para que o algoritmo Rotor possa contornar as falhas.

A Tabela 6 consolida os resultados obtidos pelos experimentos realizados sobre a topologia *AB Fat-Tree*. A utilização de um Plano de Controle para orquestrar a política de roteamento é indiferente quanto à organização das conexões entre os equipamentos. O processo de detecção das falhas se manteve o mesmo e apresentou perdas de pacotes durante o processo de recuperação, todavia apresenta uma tabela de roteamento otimizado durante a etapa $E3$. O algoritmo Estático não conseguiu contornar as falhas em cenários com duas e três falhas e manteve as condições já demonstradas na Seção 5.4.

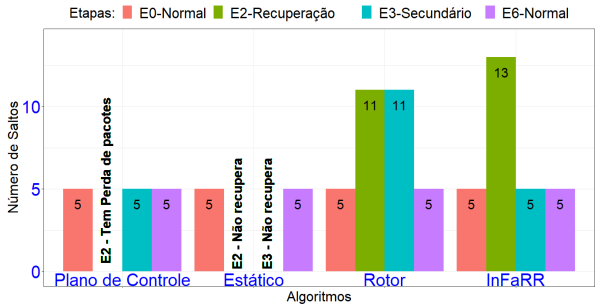
A reestruturação das conexões possibilitou ao algoritmo Rotor a recuperação nos cenários em que antes não era possível utilizando a topologia *Standard Fat-tree*. O número de saltos apresentados durante o processo de recuperação (etapa $E2$) se manteve igual à etapa $E3$ em que o fluxo para contornar as falhas utiliza um caminho redundante. Assim, durante o o processo de reroteamento o número de saltos passa de 5 para 7 no cenário com 1 falha (Figura 38a), para 11 saltos no cenário com 2 falhas (Figura 38b) e chega a 13 saltos no cenário com 3 falhas (Figura 38c). Esse aumento no número de saltos mostra



(a) Número de saltos por etapas. Cenário com 1 falha.



(b) Número de saltos por etapas. Cenário com 2 falha.



(c) Número de saltos por etapas. Cenário com 3 falha.

Figura 38 – *AB Fat-tree* - Resultados Obtidos.

que a tabela de roteamento não é otimizado e que o domínio de recuperação é dependente de outros *PODs*.

Esses resultados obtidos sugerem a hipótese de que se o algoritmo Estático fosse implementado com tabelas de roteamento que utilizassem chave de pesquisa compostas de endereçamento IP origem e endereçamento IP destino, poderia usufruir dos mesmos caminhos explorados pelo algoritmo Rotor para completar o mecanismo de recuperação.

O algoritmo InFaRR não apresentou mudanças nos resultados obtidos sobre a topologia *AB Fat-Tree*; assim, o remanejamento das conexões dos enlaces entre os equipamentos não se faz necessário para o perfeito funcionamento do algoritmo e para a promoção de resiliência da rede. Os Mecanismos de *Pushback*, Reconhecimento e Restauração, Prevenção de *Loop* e Retorno à Rota Principal funcionaram sem restrições.

Durante o processo de recuperação (etapa $E2$) em cenário com 3 falhas, o algoritmo InFaRR submete o pacote com origem em H1P1 a 4 etapas de *Puschbak*, até que o pacote encontre um caminho válido até o destino H1P2. A Figura 39 demonstra o funcionamento do Mecanismo de *Pushback* destacado pelas linhas tracejadas em vermelho e o Mecanismo de Reconhecimento e Restauração atuando nos comutadores T1P1 e A1P1, de forma a propor uma tabela de roteamento otimizado com 5 saltos durante a etapa $E3$, destacado em roxo. Os círculos numerados até 4 demonstram os enlaces que o pacote passou através do reroteamento proporcionado pelo Mecanismo de *Pushback*.

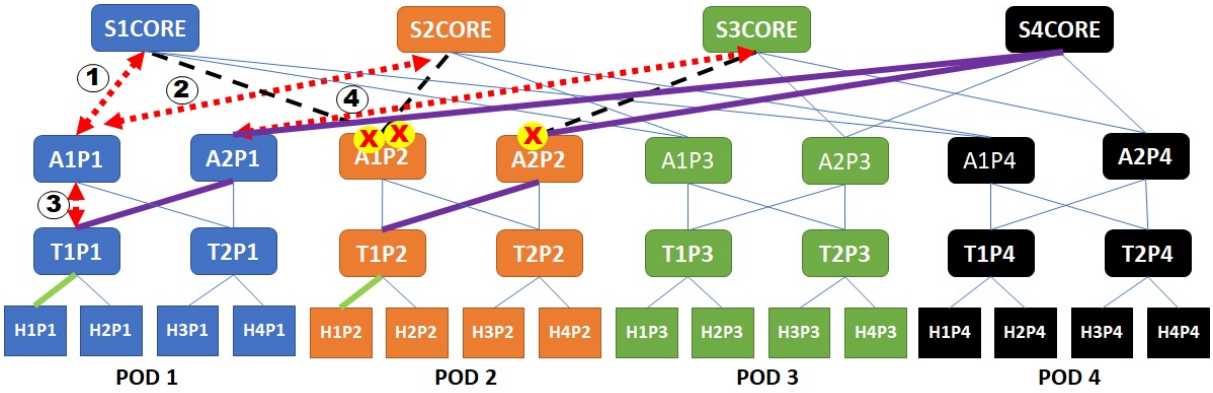


Figura 39 – *AB Fat-Tree* - Cenário com 3 falhas - Algoritmos InFaRR.

Conclusão

O InFaRR é um algoritmo de reroteamento rápido em redes programáveis, desenvolvido na linguagem P4, livre de cabeçalhos adicionais de gerenciamento (*overheads*) e de pacotes de monitoramento do estado da rede (*heartbeats*). Tem como objetivo prover uma solução imediata de recuperação para contornar uma ou mais falhas simultâneas antes da atuação do plano de controle. A base de seu funcionamento explora a característica da linguagem P4 *match-action* sobre uma tabela principal e outra secundária, que podem ser construídas com chaves de pesquisas personalizadas. O algoritmo InFaRR apresenta quatro características essenciais não encontradas, de maneira conjunta, em outros mecanismos de recuperação: Prevenção de *Loop*, *Pushback*, Reconhecimento e Restauração, e Retorno à Rota Principal.

O algoritmo InFaRR demonstrou excelentes resultados, com destaque para o sucesso no processo de recuperação em todos os cenários avaliados, a ausência de perda de pacotes após detectada a falha do enlace (etapa $E2$) e a tabela de roteamento otimizado durante o período de falha (etapa $E3$). A fase de experimentação avaliou diferentes baterias de testes sobre os cenários com uma, duas e três falhas simultâneas nas topologias de redes hierárquicas *Standard Fat-Tree* e *AB Fat-Tree*.

O estudo sobre topologias datacenter com três camadas demonstrou a capacidade de recuperação do algoritmo InFaRR com até três falhas simultâneas na comunicação entre os *PODs*. O mecanismo de recuperação apresentou um plano de encaminhamento otimizado para contornar as falhas; entretanto o aumento do número de *PODs* e do número de enlaces não se refletiu no aumento da capacidade de recuperação, já que o algoritmo proposto trabalha com uma tabela de roteamento principal e outra secundária, impossibilitando o encaminhamento de pacotes a mais de dois caminhos diferentes em cada comutador.

Diante dos resultados obtidos, entende-se que o algoritmo InFaRR ampliaria sua capacidade de recuperação em estruturas hierárquicas com mais camadas, já que se torna possível explorar mais adjacências durante a execução do roteamento, reroteamento e Mecanismo de *Pushback*. O Mecanismo de Retorno à Rota Principal executado no plano de dados viabiliza a recuperação do roteamento diante de falhas remotas antes da intervenção do plano de controle; a antecipação da recuperação é importante para desonerar os enlaces de proteção utilizados para prover o contorno a falhas e trazer o fluxo para o plano de encaminhamento inicialmente idealizado.

O processo de recuperação apresentou resultados que atendem aos requisitos de qualidade de serviços descritos na Tabela 5 e 6. Se comparado aos algoritmos adaptados

da literatura (Plano de controle, Estático e Rotor), o InFaRR obteve melhor resultado por se recuperar corretamente em todos os cenários de falhas avaliados. A programabilidade existente no plano de dados viabilizou a codificação do InFaRR permitindo a detecção do *loop* e a otimização do domínio de recuperação, não se fazendo necessário o roteamento do tráfego em um *POD* externo. Esta característica possibilitou a otimização do número de saltos durante a etapa $E3$ - caminho secundário.

As proposta do remanejamento das conexões da topologia *AB Fat-tree* promove de fato maior resiliência da rede. O algoritmo Rotor, que foi incapaz de se recuperar em cenários com duas e três falhas na topologia *standand Fat-Tree*, conseguiu contornar as falhas na topologia *AB Fat-Tree*. O algoritmo InFaRR não apresentou distinção entre as topologias, e entregou um plano de encaminhamento sempre otimizado graças aos Mecanismo de *Pushback* e Reconhecimento e Restauração.

Os testes realizados em ambiente controlado fundamentam a realização de estudos mais aprofundados. Questões sobre a implementação em ambientes físicos possibilitarão a observação do comportamento do InFaRR diante da velocidade de processamento, consumo de memória em redes com diferentes tipos de classes de tráfegos, flutuações na carga da rede e outras chaves de pesquisa na tabela de roteamento.

A inclusão de funcionalidades atribuídas à etapa $E1$ - mecanismo de Controle que não fizeram parte do escopo deste trabalho, poderão ser avaliadas em trabalhos futuros, de forma a viabilizar o reroteamento rápido diante da avaliação de alta utilização da rede, descartes de pacotes, monitoração da retransmissão de pacotes ou o não atendimento de requisitos de qualidade de serviços como IPTD, IPDV ou perda de pacotes. Todas essas variáveis podem ser controladas por um comutador programável. Por fim, o InFaRR deverá ser avaliado em outras topologias, de forma a verificar sua viabilidade e desempenho em contextos reais de utilização.

Publicações

A Figura 40 é a cópia do certificado de apresentação na Trilha Principal no evento promovido pelo XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos 2022, realizado em Fortaleza - CE.

Figura 40 – Certificado de apresentação no SBRC 2022.

Referências

ALI, J. et al. Software-defined networking approaches for link failure recovery: A survey. *Sustainability (Switzerland)*, v. 12, n. 10, 2020. ISSN 20711050. Citado na página 25.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, IEEE, v. 1, n. 1, p. 11–33, 2004. Citado na página 30.

BOROKHOVICH, M.; SCHIFF, L.; SCHMID, S. Provable data plane connectivity with local fast failover: Introducing OpenFlow graph algorithms. *HotSDN 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Hot Topics in Software Defined Networking*, p. 121–126, 2014. Citado na página 44.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2656877.2656890>. Citado 5 vezes nas páginas 26, 36, 38, 39 e 62.

CHAN, K.-Y. et al. Fast failure recovery for in-band controlled multi-controller openflow networks. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. [S.l.: s.n.], 2018. p. 396–401. Citado na página 83.

CHIESA, M. et al. Fast Recovery Mechanisms in the Data Plane. *Ieee Cmst*, p. 1–46, 2020. Citado 4 vezes nas páginas 30, 32, 43 e 44.

CHIESA, M. et al. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. New York, NY, USA: Association for Computing Machinery, 2019. (CoNEXT '19), p. 1–14. ISBN 9781450369985. Disponível em: <https://doi.org/10.1145/3359989.3365410>. Citado 2 vezes nas páginas 26 e 45.

CHODOREK, R. Qos measurement and evaluation of telecommunications quality of service [book review]. *Communications Magazine, IEEE*, v. 40, p. 30–32, 03 2002. Citado na página 31.

CHOLDA, P. et al. A survey of resilience differentiation frameworks in communication networks. *IEEE Communications Surveys and Tutorials*, v. 9, n. 4, p. 32–55, 2007. ISSN 1553877X. Citado 2 vezes nas páginas 32 e 35.

CHOLDA, P. et al. Quality of resilience as a network reliability characterization tool. *IEEE Network*, v. 23, n. 2, p. 11–19, 2009. ISSN 08908044. Citado na página 30.

GIBB, G. et al. Design Principles for Packet Parsers. p. 13–24, 2013. Citado na página 37.

GILL, P.; JAIN, N.; NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 41, n. 4, p. 350–361, aug 2011. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/2043164.2018477>. Citado na página 39.

GOMEZ, J. et al. A survey on TCP enhancements using P4-programmable devices. *Computer Networks*, Elsevier B.V., v. 212, n. May, p. 109030, 2022. ISSN 13891286. Disponível em: <https://doi.org/10.1016/j.comnet.2022.109030>. Citado na página 38.

HAUSER, F. et al. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. 2021. Disponível em: <http://arxiv.org/abs/2101.10632>. Citado 3 vezes nas páginas 36, 38 e 54.

ITU-T. Recommendation Y.1541: network performance objectives for IP-based services. *ITU-T*, p. 7–9, 2002. Citado na página 31.

KAMISINSKI, A. Evolution of IP fast-reroute strategies. In: *Proceedings of 2018 10th International Workshop on Resilient Networks Design and Modeling, RNDM 2018*. [S.l.: s.n.], 2018. ISBN 9781538670309. Citado na página 26.

KFOURY, E. F.; CRICHIGNO, J.; BOU-HARB, E. An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access*, v. 9, p. 87094–87155, 2021. ISSN 21693536. Citado na página 37.

KIRANMAI, L. et al. IP Fast Rerouting framework with Backup Topology. *International Journal of Computer Engineering In Research Trends*, v. 1, n. 2, p. 96–103, 2014. ISSN 2349-7084. Citado na página 25.

KRISHNA, V.; SURI, N. N. R. R.; ATHITHAN, G. A comparative survey of algorithms for frequent subgraph discovery. In: . [S.l.: s.n.], 2011. Citado na página 44.

LEISERSON, C. E. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34, n. 10, p. 892–901, 1985. ISSN 00189340. Citado 3 vezes nas páginas 39, 41 e 47.

LI, Z. et al. P4Resilience: Scalable Resilience for Multi-failure Recovery in SDN with Programmable Data Plane. *Computer Networks*, Elsevier B.V., v. 208, n. March, p. 108896, 2022. ISSN 13891286. Disponível em: <https://doi.org/10.1016/j.comnet.2022.108896>. Citado na página 45.

LINDNER, S. et al. P4-Protect: 1+1 Path Protection for P4. *EuroP4 2020 - Proceedings of the 3rd P4 Workshop in Europe, Part of CoNEXT 2020*, p. 21–27, 2020. Disponível em: <http://arxiv.org/abs/2001.11370>. Citado na página 44.

LIU, J. et al. Ensuring connectivity via data plane mechanisms. In: *10th USENIX Symposium on Networked Systems Design and Implementation*. [S.l.: s.n.], 2013. p. 113. Citado 2 vezes nas páginas 26 e 45.

LIU, V. et al. F10: A fault-tolerant engineered network. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. USA: USENIX Association, 2013. (nsdi'13), p. 399–412. Citado 3 vezes nas páginas 42, 44 e 73.

Mas MacHuca, C. et al. Technology-related disasters: A survey towards disaster-resilient Software Defined Networks. *Proceedings of 2016 8th International Workshop on Resilient Networks Design and Modeling, RNDM 2016*, IEEE, n. November, p. 35–42, 2016. Citado na página 30.

MAUTHE, A. et al. Disaster-resilient communication networks: Principles and best practices. *Proceedings of 2016 8th International Workshop on Resilient Networks Design and Modeling, RNDM 2016*, p. 1–10, 2016. Citado 3 vezes nas páginas 29, 30 e 31.

MENTH, M. et al. Implementation and evaluation of activity-based congestion management using P4 (P4-ABC). *Future Internet*, v. 11, n. 7, p. 1–12, 2019. ISSN 19995903. Citado na página 39.

NIKOLAEVSKIY, I. *Scalability and Resiliency of Static Routing*. 74 + app. 54 p. Tese (Doctoral thesis) — School of Science, 2016. Disponível em: <http://urn.fi/URN:ISBN:978-952-60-7194-7>. Citado na página 46.

ONF. OpenFlow Switch Specification 1.4.0. *Current*, v. 0, p. 1–3205, 2013. ISSN 09226389. Citado na página 48.

QU, T. et al. SQR: In-network packet loss recovery from link failures for highly reliable datacenter networks. *Proceedings - International Conference on Network Protocols, ICNP*, IEEE, v. 2019-Octob, 2019. ISSN 10921648. Citado na página 39.

RAK, J. *Resilient Routing in Communication Networks (Springer, 11.2015)*. [S.l.]: Springer Publishing Company, Incorporated, 2015. ISBN 978-3-319-22332-2. Citado na página 33.

SEDAR, R. et al. Supporting emerging applications with low-latency failover in P4. *NEAT 2018 - Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, Part of SIGCOMM 2018*, p. 52–57, 2018. Citado 2 vezes nas páginas 26 e 47.

SERVICE, T.; OPERATION, S. *Definitions of terms related to quality of service*. [S.l.], 2008. Citado na página 29.

SGAMBELLURI, A. et al. OpenFlow-based segment protection in Ethernet networks. *Journal of Optical Communications and Networking*, v. 5, n. 9, p. 1066–1075, 2013. ISSN 19430620. Citado 2 vezes nas páginas 25 e 48.

SHAN, Y. et al. Singapore towards a fully disaggregated and programmable data center. ACM, v. 22, p. 11, 2022. Disponível em: <https://doi.org/10.1145/3546591.3547527>. Citado na página 75.

SHARMA, S. et al. OpenFlow: Meeting carrier-grade recovery requirements. *Computer Communications*, v. 36, n. 6, p. 656–665, 2013. ISSN 01403664. Disponível em: <http://dx.doi.org/10.1016/j.comcom.2012.09.011>. Citado na página 32.

SMITH, P. et al. Network resilience: A systematic approach. *IEEE Communications Magazine*, v. 49, n. 7, p. 88–97, 2011. ISSN 01636804. Citado na página 29.

STANKIEWICZ, R.; CHOLDA, P.; JAJSZCZYK, A. QoX: What is it really? *IEEE Communications Magazine*, IEEE, v. 49, n. 4, p. 148–158, 2011. ISSN 01636804. Citado na página 31.

STERBENZ, J. P. et al. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Computer Networks*, v. 54, n. 8, p. 1245–1265, jun 2010. ISSN 13891286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128610000824>. Citado 3 vezes nas páginas 29, 30 e 33.

SUBRAMANIAN, K. et al. D2r: Policy-compliant fast reroute. In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. New York, NY, USA: Association for Computing Machinery, 2021. (SOSR '21), p. 148–161. ISBN 9781450390842. Disponível em: <https://doi.org/10.1145/3482898.3483360>. Citado na página 26.

XU, J.; XIE, S.; ZHAO, J. P4Neighbor: Efficient Link Failure Recovery with Programmable Switches. *IEEE Transactions on Network and Service Management*, v. 18, n. 1, p. 388–401, 2021. ISSN 19324537. Citado na página 45.

YAN, Y. et al. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology*, IEEE, v. 38, n. 9, p. 2688–2694, 2020. Citado na página 75.