UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Adaptive Fail Recovery Control: Uma Abordagem Adaptativa para Recuperação de Falhas em Redes Definidas por *Software*

Alexandre Alves dos Santos de Campos

Orientador

Sidney Cunha de Lucena

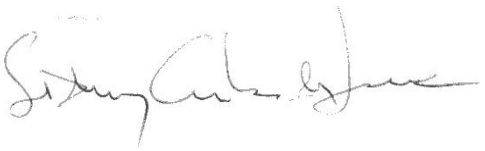
RIO DE JANEIRO, RJ - BRASIL
MAIO de 2020

Adaptive Fault Recovery Control: Uma Abordagem Adaptativa para Recuperação de Falhas em Redes Definidas por *Software*

Alexandre Alves dos Santos de Campos

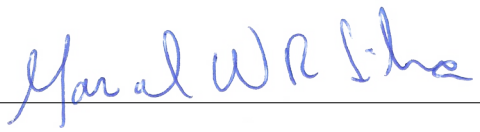
DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:



Sidney Cunha de Lucena, D.Sc.

Carlos Alberto Vieira Campos, D.Sc.



Marcel William Rocha da Silva, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
MAIO de 2020

Campos, Alexandre Alves dos Santos de
C198 Adaptive Fault Recovery Control: Uma Abordagem Adaptativa para Recuperação de Falhas em Redes Definidas por Software /
Alexandre Alves dos Santos de Campos, 2020. - - Rio de Janeiro, 2020.
114 f.

Orientador: Sidney Cunha de Lucena.
Dissertação (Mestrado) - Universidade Federal do Estado do Rio de Janeiro, Programa de Pós-Graduação em Informática, 2020

1. Resiliência em Redes Definidas por software. 2. Plano de Dados. 3. SDN 4. OpenFlow. 5. Fast Failover. I. , Sidney Cunha de Lucena, orient. II . Título.

Não poderia atribuir a apenas uma pessoa. Ao pai,Mauro Eduardo (in memorian), que me deixou inestimável herança: o amor pelo conhecimento, À minha mãe,Herondina (in memorian), que me ensinou a nunca desistir, Ao meu amigo,Roberto, que me incentivou a começar nesta jornada, ´À minha amiga, Danielle, que foi quem me apresentou a UNIRIO, Ao meu amigo,Daniel, que me ajudou a vencer os desafios da programação, À minha filha,Isabelle, que esteve sempre ao meu lado. À minha cunhada,Sônia, que sempre me deu palavras de incentivo. Aos meus irmãos,Charles e Andréa, que sempre me apoiaram nas fases mais difíceis nestes 3 últimos anos.

Agradecimentos

Primeiramente eu gostaria de agradecer a toda a minha família, especialmente à minha cunhada Sônia, que sempre que nos falamos, me da palavras de incentivo, perpetuando os ensinamentos da minha mãe, a nunca desistir de um sonho.

Não posso me esquecer dos amigos da pesquisa! Muito obrigado Eduardo, Nicomar, Elton, Felipe, Rodrigo.

Sou profundamente grato ao meu orientador Sidney! Acho que em toda minha vida, eu nunca conheci uma pessoa como ele, até nos momentos de correção, ele usa palavras educadas. Nunca vou me esquecer da virada que fizemos para finalizar a minha dissertação. E com poucas palavras você demostrou o grande ser humano que você é, e foram elas " Está acabando. Fica firme aí! " Muito obrigado !

Agradeço sinceramente à Universidade Federal do Estado do Rio de Janeiro juntamente com todos os seus funcionários, pela oportunidade de adquirir conhecimento em uma instituição de ensino superior de alta qualidade. Certamente as aulas que recebi, bem como os serviços que foram prestados foram relevantes para a minha formação acadêmica.

Aos membros da banca, professores Carlos Eduardo, Marcel e Guilherme, agradeço por vocês aceitarem o convite para avaliar o meu trabalho. Para mim isso é uma honra e ao mesmo tempo um aprendizado.

Deixo também um agradecimento para aquele que se propõe a ler este trabalho. Espero que ele seja de grande valia e contribua para os seus objetivos. Ademais, estou disponível para quaisquer dúvidas ou comentários sobre a pesquisa.

E finalmente agradeço àquele que embora não visível aos meus olhos mas sempre em meus pensamento. Deus(pai), sou eu extremamente grato pelo auxilio diário e por todos os momentos, que meu deu sinais e me livrou de situações para me

mostrar que eu estava no caminho certo e que era possível chegar lá. Também peço desculpas por, se em alguns momento, tive dúvidas e me perdi, e com seu infinito amor me conduziu de volta ao caminho.

Campos, Alexandre Alves dos Santos de. ***Adaptive Fault Recovery Control*: Uma Abordagem Adaptativa para Recuperação de Falhas em Redes Definidas por *Software***. UNIRIO, 2020. 114 páginas. Dissertação de Mestrado. Programa de Pós-Graduação em Informática, UNIRIO.

RESUMO

As *Software-Defined Networks*, ou SDN, tiveram um enorme crescimento e vêm sendo usadas ativamente em diferentes tipos de redes nos últimos tempos. Em função disso, a pesquisa sobre recuperação de falhas no plano de dados é um aspecto importante que precisa ser considerado em uma arquitetura SDN. Tradicionalmente, as abordagens de recuperação de falhas podem ser divididas em dois tipos: reativa e proativa. Na abordagem reativa, após a detecção da falha o controlador reconhece a nova topologia e calcula os novos caminhos para os fluxos afetados pela falha, inserindo novas regras de encaminhamento nas tabelas dos *switches*. Como a quantidade de mensagens de sinalização é proporcional ao tamanho da rede, essa sobrecarga pode se tornar problemática, assim como o tempo para a nova configuração das tabelas. Já na recuperação proativa, caminhos de *backup* são pré-implantados nas tabelas dos *switches* e o redirecionamento automático em caso de falha pode reduzir bastante o atraso da recuperação, porém isso pode gerar um aumento proibitivo do tamanho dessas tabelas, limitadas pela quantidade de memória nos *switches*. A maioria dos trabalhos que propõem soluções para recuperação de falhas em SDN não considera que diferentes tipos de fluxo têm diferentes requisitos de atrasos de recuperação. Com base nisso, este trabalho apresenta o *Adaptive Fault Recovery Control* - AFRC: uma estratégia adaptativa baseada em SDN na qual caminhos de *backup* são pré-implantados apenas para fluxos ativos com determinados requisitos de atraso e somente nos *switches* selecionados pelo controlador para encaminhar esses fluxos, definindo-se a quantidade e o tempo de permanência dessas regras conforme a ocupação das tabelas nos *switches*. O AFRC usa os grupos *Fast Failover*, nativos do protocolo Openflow, para proteger esses fluxos e o parâmetro *idle timeout* é usado para remover essas regras de *backup* após o término dos fluxos, gerenciando assim o uso de memória nos *switches*. Os estudos de caso deste trabalho demonstram a efetividade desta solução para diferentes cenários em redes de *datacenter*.

Palavras-chave: Recuperação de falhas, OpenFlow, SDN e Resiliência.

ABSTRACT

Software-Defined Networks, or SDN, have seen tremendous growth and have been used actively in different types of networks lately. Because of this, research on data plan failure recovery is an important aspect that needs to be considered in an SDN architecture. Traditionally, disaster recovery approaches can be divided into two types: reactive and proactive. In the reactive approach, after the fault detection, the controller recognizes the new topology and calculates the new paths for the flows affected by the fault, inserting new routing rules in switches' tables. As the quantity of signaling messages is proportional to the size of the network, this overload can become problematic, as well as the time for the new configuration of the tables. In proactive recovery, backup paths pre-implanted in switches' tables and automatic redirection in case of failure can greatly reduce the recovery delay, however this can generate a prohibitive increase in the size of these tables, which are limited by the amount of memory of the switches. Most of the works that propose solutions for recovering from SDN failures do not consider that different types of flows have different requirements for recovery delays. Based on this, this work presents the Adaptive Fault Recovery Control - AFRC: an adaptive strategy based on SDN in which backup paths are pre-implanted only for active flows that present certain delay requirements and only for the switches selected by the controller to route these flows, defining the number of rules and their timeouts according to the usage rate of the tables in the SDN switches. The AFRC uses the Fast Failover groups, native to the Openflow protocol, to protect these flows and the idle timeout parameter is used to remove these backup rules after the ending of flows, thus managing memory usage in the switches. The case studies of this work demonstrate the effectiveness of this solution for different scenarios in datacenter networks.

Keywords: Fault recovery, OpenFlow, SDN and Resilience.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Nomenclaturas

SDN	*Sofware Defined Networdk*
TCAM	*Ternary Content-Addressable Memory*
AFRC	*Adaptive Fault Recovery*
CWB	*Cold Walter Bucket*
API	*Application Programming Interface*
NAT	*Network Address Translation*
OFDP	*OpenFlow Discovery Protocol*
LLDP	*Link-Layer Discovery Protocol*
SRAM	*Static Random Access Me-mory*
TOR	*Top of the Rack*
ACL	*Access Conrol List*
LOS	*Loss of Signal*
BFD	*Bidirectional Forward Detection*
LSM	*Link StateMonitoring*
FLI	*Failure Location Identification*
FRR	*Fast Reroute*
SLF	*Single link failures*
SNF	*Single node failures*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MAC	*Media Access Control*
ONF	*Open Networking Foundation*
QoS	*Quality of Service*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
VM	*virtual machine*
QOE	*Quality of Experience*
QoS	*Quality of Service*
NMS	*Network Management System*

1. Introdução

1.1 Considerações Iniciais e Motivação para a Pesquisa

O rápido desenvolvimento da Internet das coisas, computação em nuvem, inteligência artificial e outros campos provocou um crescimento explosivo do tráfego de dados, o que impôs novas demandas às redes de comunicação. As Redes Definidas por Software, ou SDN, de *Software-Defined Networks* [1], representam uma nova arquitetura de rede separando o plano de controle do plano de dados, para aprimorar a flexibilidade e permitir a programabilidade da rede [2].

Em um SDN, o plano de controle é logicamente centralizado. Isso se dá através do uso de um elemento chamado "controlador SDN", responsável pela interação com os dispositivos do plano de dados, comumente chamados de *datapaths* ou simplesmente *switches*. A comunicação entre controlador e *datapaths* é realizada através de um protocolo específico, sendo o OpenFlow [3] o protocolo mais amplamente usado, configurando-se praticamente como um padrão de mercado. Num plano acima do controlador, chamado de plano de aplicação, encontram-se as aplicações de gerenciamento de rede através das quais se programa o comportamento desejado dos *switches* quanto ao encaminhamento dos dados. O protocolo OpenFlow prevê que as regras de encaminhamento podem utilizar múltiplos campos de um pacote de dados, funcionando de forma quase análoga a regras de *firewall*. Portanto, as regras de encaminhamento são denominadas de regras de fluxos e ficam armazenadas nas tabelas de fluxos. Assim, quando um pacote ingressa no *switch* OpenFlow para ser encaminhado, a tabela é varrida sequencialmente até que os campos do pacote combinem com uma regra. Associada à regra, uma ação específica será tomada e, caso não ocorra combinação com nenhuma regra, o pacote é descartado. Contadores são provisionados para permitir o monitoramento de quantos pacotes fazem *match* nas diversas regras. Dessa forma, o controlador funciona como um *middleware* entre o

plano de controle e o plano de dados, permitindo um gerenciamento programável e centralizado da rede, em oposição à forma tradicional na qual o funcionamento da rede se dá em função de configurações e protocolos distribuídos pelos equipamentos e pré-definidos por seus fabricantes.

Seja em uma rede IP tradicional ou em uma SDN, existe a possibilidade de falha no plano de dados, o que levará à interrupção das transmissões. Portanto, a recuperação eficaz de falhas de enlace no plano de dados é importante para garantir a confiabilidade, disponibilidade e estabilidade da rede, e no caso de uma SDN há peculiaridades relacionadas à interação entre plano de dados e plano de controle que precisam ser consideradas [4] [5]. Uma estratégia seria utilizar a abordagem ativa, onde regras adicionais relacionadas a rotas de *backup* seriam instaladas previamente nos equipamentos de encaminhamento (*switches*), porém há limitações impostas pela quantidade de memória nos *switches* e complexidades ligadas às possíveis mudanças de topologia que dificultam o gerenciamento nessa abordagem [6]. Sharma et al [7] provaram que o mecanismo de proteção usando abordagem ativa funciona bem e seu tempo de recuperação pode atender aos requisitos *carrier grade* (nível de "operadora de telecomunicações"), ou seja, menos de 50ms [8]. No entanto, essa abordagem de recuperação desperdiça recursos nos equipamentos e pode não se adaptar à dinâmica de mudanças usual das redes [9].

Aproveitando o controle centralizado em uma SDN, a abordagem de restauração reativa pode ser usada para fins de recuperação de falhas. Nessa abordagem de restauração, um novo caminho será calculado no plano de controle em tempo real, após falhas no plano de dados serem detectadas pelo controlador. Sharma et al mostraram que esse mecanismo de restauração costuma funcionar sem onerar os recursos dos *switches*, mas o tempo de recuperação costuma ser muito superior aos 50ms de requisito *carrier grade* de desempenho [10]. Isso ocorre porque muitas mensagens de sinalização e controle precisam ser trocadas entre o controlador e *switches*, além da computação de rotas alternativas no plano de aplicação, tudo isso contribuindo para o aumento do tempo de recuperação.

Não é uma tarefa simples obter cobertura total de proteção, o que está intimamente relacionado à topologia de rede e ao tipo de falha. De acordo com o tipo e o número de componentes defeituosos, o tipo de falha pode ser categorizado em falha de enlace ou falha de nó [11]. Para algumas topologias, quando ocorre uma falha de múltiplos componentes, alguns fluxos não têm caminhos de *backup* válidos ou o redirecionamento de fluxos causará *loops* de roteamento. Portanto, esses fluxos não podem ser protegidos e a cobertura total de proteção não pode ser alcançada.

É também difícil garantir que os caminhos de *backup* sempre tenham um bom desempenho. Esse desempenho geralmente é refletido nos valores de atraso, utilização da largura de banda e taxa de perda de pacotes, conforme proposto em [12]. Porém, tratam-se de métricas que costumam ser afetadas por vários fatores. Portanto, ao se planejar caminhos de *backup*, é necessário considerar não somente as medidas de desempenho atuais dos enlaces, mas também as tendências de desempenho relacionadas ao estado do enlace no momento que for utilizado como caminho de *backup*.

Com a expansão contínua das redes e dos serviços por elas suportados, os *switches* precisam armazenar cada vez mais regras nas tabelas de fluxo. Todavia, as memórias do tipo TCAM (*Ternary Content-Addressable Memory*), que armazenam as entradas das tabelas de fluxo, são caras e possuem capacidade limitada [13]. Portanto, não é possível armazenar entradas nas tabelas de fluxo "indefinidamente". Por exemplo, no *hardware* do *switch* HP ProCurve 5406zl, a TCAM pode suportar 1500 regras OpenFlow, sendo que cada *host* exige, em média, dezenas de regras OpenFlow na rede para sua conectividade, o que significa que o 5406zl poderá suportar apenas 150 usuários [14]. Portanto, a capacidade de armazenamento da tabela de fluxo é um fator que afeta a manutenção das regas para o redirecionamento quando alguma falha ocorre.

Para reduzir o atraso da recuperação, a maioria das abordagens de recuperação de falhas determina os caminhos de *backup* e implanta entradas de fluxo de *backup* antes que ocorram as falhas. O *backup* pré-implementado precisa proteger um grande número de fluxos, o que é um grande desafio para o espaço de armazenamento das tabelas de fluxo. E quando o espaço de armazenamento acabar, algumas entradas da tabela de fluxo serão excluídas. Além disso, as regras SDN podem ser mais complexas que as regras de roteamento clássicas. Um *switch* SDN geralmente contém entradas de fluxo inferiores a 5000 [14]. No entanto, para uma distribuição em um provedor de serviços ou até uma grande rede do campus, o número de fluxos simultâneos através de um *switch* pode facilmente exceder dezenas de milhares.

Por exemplo, em um *datacenter* de tamanho moderado, a taxa média de ingresso pode chegar a 22 mil fluxos por segundo para um *rack* composto por 40 servidores hospedando cerca de 15 máquinas virtuais por *host* [15]. Portanto, o consumo de espaço e energia da TCAM com regras que podem não ser usadas, assim como a gerência às mudanças na topologia, desencorajam essa abordagem.

Uma outra abordagem pode ser adotada quando já há fluxos instalados nos

switches e há uma mudança na topologia. Após o novo caminho ser calculado no controlador SDN, mensagens OpenFlow do tipo *FLOWMOD*, que modificam regras previamente instaladas, são enviadas aos *switches* onde há fluxos ativos afetados pela falha, notificando o novo caminho ou porta de saída que será utilizada para o encaminhamento do tráfego. Trata-se, portanto, de uma abordagem do tipo reativa, que igualmente demanda uma notificação ao controlador da falha e o envio de mensagens *FLOWMOD* do controlador aos *switches*, resultando em algum nível de atraso até que a falha consiga ser recuperada. No entanto, estudos recentes mostram que aumentos no atraso podem ser críticos para muitos sistemas de *datacenter*. Por exemplo, atrasos de 100ms causam uma queda de 1% na receita da Amazon e atrasos de 400ms causam uma queda de 5 a 9% na receita do Google [16].

Trabalhos anteriores propostos para utilizar eficientemente os recursos da TCAM podem ser divididos em três categorias: compressão de tabela [17][18], *cache* das regras mais utilizadas [19] e distribuição de regras [20] [21]. A compactação de tabela compacta as entradas de regras na TCAM, substituindo os bits de seus campos de correspondência por bits curinga ("*", representando tanto bit 0 como bit 1). Já a distribuição de regras inclui duas etapas: decomposição da tabela e alocação de sub-tabela. A decomposição de tabela particiona a tabela de regras de um único *switch* em sub-tabelas, enquanto a alocação de sub-tabela distribui as sub-tabelas por toda a rede para reduzir o número de regras armazenadas em um único *switch*.

O protocolo OpenFlow define dois parâmetros para determinar o tempo de permanência de um fluxo na memória do *switch*: um é o *hard timeout*, o outro é o *idle timeout*. O *hard timeout* define o tempo máximo que a entrada de fluxo permanecerá na memória do *switch*, mesmo que ainda há trafego de dados utilizando esta regra. O *idle timeout* define o tempo máximo de uma entrada da tabela de fluxo permanecerá ma memória do *switch*. Se o intervalo de tempo entre dois *matches* seguidos de uma entrada de fluxo for superior ao *idle timeout* ou se o *hard timeout* expirar, a entrada de fluxo será considerada invalidada e será removida do switch SDN.

O tempo do *idle timeout* de um fluxo pode ser afetado por vários fatores e seu impacto na eficiência de utilização da capacidade da TCAM depende da relação entre o *idle timeout* e o intervalo de chegada de pacotes subsequentes em um fluxo. Se o *idle timeout* de uma regra de fluxo for muito maior que o intervalo de chegada de pacotes desse fluxo, depois que o *switch* transferir o último pacote, a regra de fluxo correspondente à entrada será armazenada na TCAM por um longo período antes de ser removido, o que desperdiçará a capacidade da TCAM. Segundo as

estatísticas, 92% do espaço de memória que é desperdiçado pelas entradas de fluxo expiradas pode ser usado para transferir 40% dos fluxos descartados [22]. Portanto, definir um *idle timeout* apropriado, de acordo com o intervalo de chegada de pacotes, pode efetivamente melhorar a eficiência de utilização da capacidade da TCAM sem aumentar a carga no controlador SDN.

1.2 Objetivos do Trabalho

Partindo-se da hipótese de que é possível adotar uma abordagem adaptativa de recuperação de falhas, através da qual técnicas reativas e proativas sejam intercambiadas e aplicadas a fluxos específicos, e que essa abordagem provê um melhor compromisso entre o tempo de recuperação de falhas para fluxos prioritários e a ocupação das memórias TCAM dos *switches* de uma SDN, este trabalho propõe o Adaptive Fault Recovery Control - AFRC, uma estratégia adaptativa em SDN onde caminhos de *backup* são instalados proativamente apenas para fluxos ativos com determinados requisitos de atraso e somente nos *switches* que acomodam esses fluxos. Além do compromisso entre o tempo de recuperação e o uso de memória dos *switches*, essa estratégia também alivia a sobrecarga no controlador decorrente de técnicas reativas de recuperação.

O AFRC faz uso dos grupos *Fast Failover* especificados pelo protocolo OpenFlow [3] e seu processo se inicia quando o controlador recebe um pacote solicitando informações de qual ação tomar para um fluxo que tenha sido especificado como importante pelo administrador da rede. Após instaladas as regras de encaminhamento desse fluxo nos *switches* da rede, o AFRC irá instalar nesses mesmos dispositivos as regras de *backup* utilizando as tabelas *fast failover*. Como essas tabelas não contêm campo para controle de remoção (*idle timeout*), que são utilizados nas tabelas de fluxos para remoção de regras após um tempo de inatividade, o AFRC irá remover esses grupos após detectar que elas não estão sendo mais utilizadas, sendo que o tempo para remoção será o mesmo usado pelo *idle timeout* padrão do controlador. Além disso, o AFRC buscará agregar os benefícios da sumarização de regras ao realizar a mudança do método reativo para o proativo [23], reduzindo assim o tráfego de entrada e saída entre o controlador e esses *switches*.

Adicionalmente, para proteger o controlador SDN de possível gargalo no consumo de recursos, como CPU e memória [24], devido a um alto tráfego agregado de entrada e saída, propõe-se incorporar ao AFRC um método aqui proposto e denominado de

Cold Walter Bucket - CWB. O CWB transforma todas as regras de fluxos de todos *switches* relacionados ao controlador em regras proativas, usando o valor de *idle timeout* padrão do controlador e sumarizando todas as regras proativas instaladas, de forma a reduzir a necessidade de consumo de memória desses *switches*.

1.3 Contribuições deste Trabalho

Um protótipo do AFRC foi implementado e validado experimentalmente através de estudos de caso voltados para cenários de redes de *datacenter*. No caso, foram emuladas topologias de redes do tipo *Spine and Leaf* e *Fat-Tree* baseadas em OpenFlow versão 1.4 e usando o controlador Floodlight [25]. Nesses estudos de caso, a solução baseada no AFRC foi validada a partir das seguintes observações:

- o tráfego de dados não é interrompido para fluxos pré-determinados ao se usar regras de *backup* implementadas para esses fluxos através dos grupos de tabelas *fast failover* nativas do protocolo OpenFlow;
- apesar da instalação dessas regras de *backup*, mantém-se um controle sobre o aumento da ocupação da memória dos *switches*, possibilitando assim a reversão da solução caso haja risco de esgotamento desses espaços de memória;
- a solução apresentou a adaptabilidade pretendida, migrando entre as formas de recuperação proativa e reativa, seja parcialmente ou integralmente, conforme o estado da rede e a carga exercida sobre o controlador SDN.

Dito isso, podem ser relacionadas, de maneira sintética, as seguintes contribuições obtidas a partir deste trabalho:

1. Propõe-se aqui uma abordagem inovadora para a recuperação de falhas em arquiteturas SDN baseadas em OpenFlow chamada *Adaptive Fault Recovery Control* (AFRC). O AFRC se adapta às condições da rede e de ocupação das TCAMs dos *switches*, permitindo uma migração entre os modos reativo e proativo de recuperação de falhas, seja para um grupo específico de fluxos em determinados *switches* ou para a rede como um todo.
2. Foi apresentada uma implementação do AFRC que, por ter fraco acoplamento ao controlador SDN, é flexível o suficiente para ser facilmente adaptada a diversos outros cenário, sendo possível ainda sua instanciação num ambiente de

nuvem. Essa implementação possuí também um conjunto de parâmetros reduzido e de simples definição, que permite uma melhor acomodação do AFRC a diferentes cenários de rede.

3. Além de prover resiliência no plano de dados, o AFRC também se preocupa com o plano de controle da rede OpenFlow, adotando para tal estratégias de recuperação de falhas que evitem sobrecargas ao controlador SDN e migrando entre elas de forma adaptativa, conforme a dinâmica da carga exercida no mesmo.

4. O AFRC teve seu funcionamento e eficácia validados para diferentes cenários de redes de *data center* baseadas em OpenFlow, emulados através da ferramenta *mininet*. Os respectivos experimentos mostraram as vantagens da solução proposta frente às soluções tradicionalmente adotadas em SDN.

1.4 Estrutura da Dissertação

Após este capítulo introdutório, o restante deste trabalho está estruturado conforme descrição a seguir.

O Capítulo 2 apresenta uma fundamentação básica para a compreensão do trabalho desenvolvido, tendo como objetivo evidenciar aspectos teóricos importantes sobre SDN, o protocolo OpenFlow e seus mecanismos nativos capazes de prover resiliência. São também apresentadas e discutidas as abordagens reativa e proativa para a recuperação de falhas no plano de dados, assim como são apresentadas algumas topologias de rede usadas em ambientes de *data center*. O capítulo se encerra com uma seção que faz referência aos trabalhos melhor relacionados com o foco da proposta deste trabalho.

O Capítulo 3 é dedicado à descrição da proposta em uma abordagem *top-down*, indo desde os aspectos gerais da solução empregada, passando pelas fases de funcionamento do mecanismo proposto e descrevendo os detalhes técnicos sobre os processos de monitoramento e mitigação de falhas presentes no AFRC.

O Capítulo 4 esclarece os detalhes da implantação do protótipo usado para fins de validação experimental da proposta. Num aspecto de ordem prática, são abordados o ambiente computacional, os componentes utilizados e os desafios encontrados. São descritos também os cenários experimentais e a metodologia utilizada para a validação da proposta, além de discutidos os resultados obtidos a partir da emulação

de falhas de enlaces nas redes dos cenários experimentais.

Por fim, O Capítulo 5 traz as conclusões desta pesquisa e propõe direcionamentos para nortear trabalhos futuros.

2. Fundamentação Teórica

Neste capítulo serão apresentados alguns fundamentos que servem de base para o entendimento do trabalho desenvolvido. Inicialmente, será abordado o conceito de SDN e explicado o funcionamento do ecossistema relacionado ao protocolo OpenFlow, onde será dado um foco especial no funcionamento dos elementos diretamente relacionados a este trabalho. Da mesma forma, questões relacionadas a descoberta de topologia de camada 2 em SDN também serão abordadas. Em seguida, serão apresentadas algumas topologias usadas em redes de *datacenter*, que representam o cenário de aplicação deste trabalho. Questões relacionadas a detecção e recuperação de falhas em SDN são abordadas e, por fim, apresentam-se os trabalhos relacionados.

2.1 Redes Definidas por Software

Apesar de sua ampla adoção, as redes IP tradicionais são complexas e os protocolos de roteamento e comutação são executados dentro dos roteadores e *switches*. Devido a esse fato, o plano de controle - responsável pelas decisões de como lidar com o tráfego de rede - e o plano de dados - responsável por efetivamente encaminhar o tráfego na rede, de acordo com as decisões tomadas pelo plano de controle - são internamente agrupados nos dispositivos de rede, reduzindo a flexibilidade das ações de encaminhamento e dificultando a inovação e a evolução das infraestruturas de redes. As redes definidas por *software* [26], ou SDN, de *Software-Defined Networks*, empregam um modelo de rede emergente que permite ultrapassar as limitações da infraestrutura de rede atual, através do desacoplamento do plano de dados e do plano de controle. Além disso, os equipamentos de rede responsáveis pelo encaminhamento de dados passam a ser mais simples, uma vez que a lógica de controle passa a ser realizada por elemento externo, chamado de controlador SDN. O controlador SDN pode ser entendido como uma plataforma de *software* que

funciona como um *middleware*, fornecendo a programação dos dispositivos de encaminhamento de rede com base em uma visão abstrata logicamente centralizada. Dessa forma, simplifica-se a configuração e a evolução da rede [27].

Os estudos e primeiras propostas voltadas para SDN, tiveram seu início na academia, em meados de 2008, e ganharam força significativa na indústria ao longo dos últimos anos. A maioria dos fornecedores de *switches* comerciais já oferecem uma linha de produtos para SDN e sua adoção foi forte o suficiente para criar a *Open Networking Foundation* - ONF [2], tendo como mantenedores Google, AT&T e COMCAST, entre outros. Seu objetivo é o da promoção e adoção do SDN através do desenvolvimento de padrões abertos. Uma visão dessa arquitetura é mostrada na Figura 2.1.

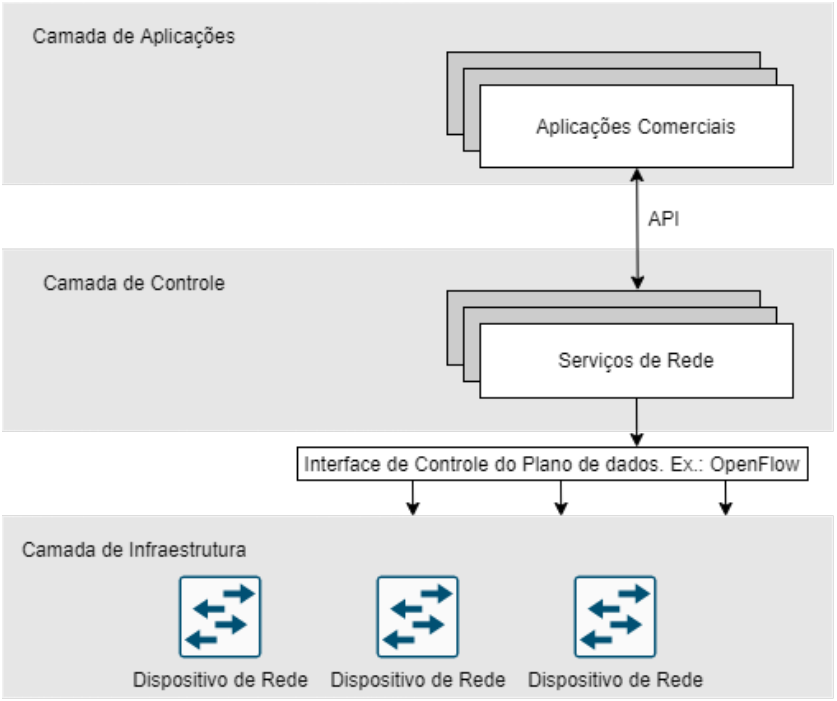


Figura 2.1: Arquitetura SDN
Adaptado de ONF(2014, p. 7).

Devido ao desacoplamento do plano de controle e do plano de dados, a comunicação entre esses planos é realizada por meio de uma interface de programação de aplicação (API). O exemplo mais notável dessa API é o OpenFlow [3, 2]. Um controlador SDN possui duas APIs de comunicação, conforme a camada com a qual vai se comunicar:

- API *southbound*: usada para transmitir e receber informações de/para os *datapaths*, onde o protocolo OpenFlow tornou-se o padrão de "mercado";

- API *northbound*: usada para a comunicação entre o controlador e as múltiplas aplicações SDN, que geralmente instruem como a rede é configurada e gerenciada.

Um dispositivo de encaminhamento OpenFlow, formalmente chamado de *datapath*, possui uma ou mais tabelas de regras de manipulação de pacotes, chamadas de tabelas de fluxo. O termo "fluxo" serve para estabelecer uma associação entre as regras de encaminhamento e a multiplicidade de campos que podem ser usados, conjuntamente ou não, como parâmetros de encaminhamento. Isso difere do que é usualmente praticado nas redes legadas, onde basicamente apenas endereços de destino, sejam de camada dois ou três, são usados como chave para a ação de encaminhamento. Dependendo das regras instaladas, o *datapath* pode ser instruído pelo controlador a funcionar como um *switch*, roteador, *firewall* ou desempenhar outras funções, como por exemplo, a de balanceador de carga. Conforme a especificação[1] da ONF, o processamento de fluxo - ou seja, dos pacotes pertencentes a um fluxo associado a uma determinada regra na tabela do dispositivo - no *pipeline* de um *datapath* é apresentado na Figura 2.2.

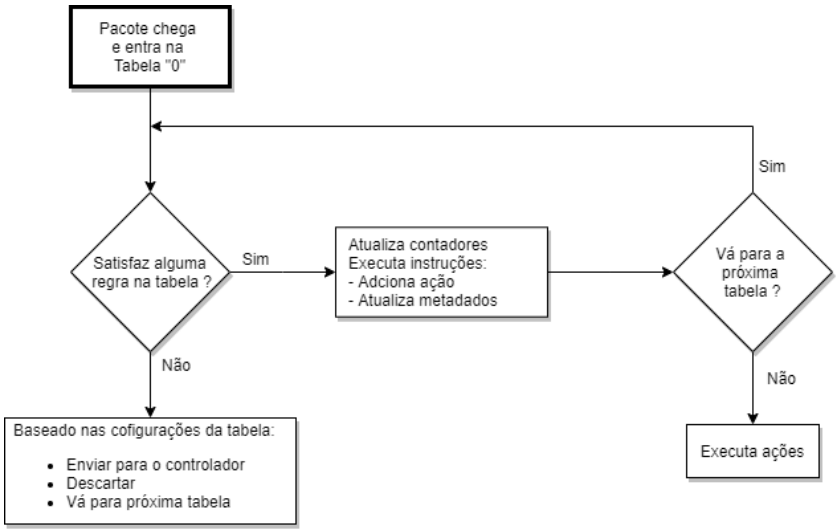


Figura 2.2: Processamento de um fluxo no *pipeline* de um *switch* OpenFlow. *Adaptado de ONF(2014, p. 14).*

Conforme discutido acima, pode-se dizer que uma SDN é caracterizada pela existência de um controlador que gerencia os elementos de rede através de uma interface de programação bem definida (API), usando um protocolo de padrão aberto, tal como o OpenFlow, para essa finalidade. De forma mais específica, os elementos de encaminhamento de rede são orquestrados e as suas tabelas de encaminhamento

[1]https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf

são manipuladas, ou seja, são programadas. Obviamente, o princípio básico de funcionamento das redes definidas por software é a possibilidade da programação dos elementos de rede. Como mencionado, essa programação, ao contrário de uma rede tradicional, se restringe a uma manipulação simples de pacotes baseado no conceito de fluxos, ou seja, sequências de pacotes que compartilham atributos com valores bem definidos. A Figura 2.3 traz um exemplo da inserção de uma regra de fluxo via o comando *curl*[2] na tabela de um *switch* OpenFlow. Descrevendo os campos da regra de fluxo dessa figura, temos:

- ***switch***: ID do *switch* que receberá a regra;
- ***name***: nome que a regra de fluxo receberá para fins de identificação;
- ***priority***: as regras de fluxo são processadas por ondem de prioridade - 0 a 6500 -, onde a com maior valor é processada primeiro;
- ***idle_timeout***: tempo máximo ocioso antes que a regra de fluxo seja removida da memória do *switch*;
- ***in_port***: interface física por onde o fluxo será recebido;
- ***eth_src***: endereço MAC do *host* de origem;
- ***eth_dst***: endereço MAC do *host* de destino;
- ***eth_type***: especifica o tipo de pacote carregado pelo quadro Ethernet (por exemplo, se é um pacote IPV4);
- ***ip_proto***: especifica qual é o protocolo de camada 4 (por exemplo, se TCP ou UDP);
- ***ipv4_src***: endereço IPv4 do *host* de origem;
- ***ipv4_dst***: endereço IPv4 do *host* de destino;
- ***actions***: identifica a ação que será executada com o respectivo pacote sempre que houver *match* com os campos anteriormente especificados (no caso deste exemplo, ser encaminhado pela interface de saída 4).

Em linhas gerais, a regra de fluxo da Figura 2.3 pode ser descrita da seguinte forma: "todo pacote que ingressar no *switch* pela porta **2**, com MAC de origem

[2] O *curl* é uma ferramenta de linha de comando para transferir dados usando vários protocolos (ver https://curl.haxx.se/).

```
curl -X POST -d { switch:00:00:00:00:00:00:00:01, name:flow-mod-1,
priority:1,idle_timeout:5, in_port:2, eth_dst:0a:0c:a5:d0:03:51,
eth_src:72:bd:ae:e8:97:46, eth_type:0x800, ip_proto:0x6, ipv4_src:10.0.0.4,
ipv4_dst:10.0.0.1, actions:output=4} http://IP:8080/wm/staticentrypusher/json
```

Figura 2.3: Exemplo de um comando *curl* para instalação de uma regra de fluxo OpenFlow.

72:BD:AE:E8:97:46 e MAC de destino **0A:0C:A5:D0:03:51**, com IPv4 de origem **10.0.0.4** e IPV4 de destino **10.0.0.1**, será encaminhado pela porta **4**".

Dentre os inúmeros projetos de controladores OpenFlow que surgiram, destacam-se os seguintes: A figura 2.4 destaca as características de cada controlador citado.

- **NOX** - Primeiro controlador SDN lançado pela Nicira Network em 2009. Foi a base de vários projetos de pesquisa na primeira exploração do uso do SDN. [28]
- **POX** - É uma versão do controlador NOX escrito em Python.[29]
- **Floodlight** - É apoiado por uma comunidade de desenvolvedores, incluindo uma série de engenheiros da Big Switch Networks. Este será o controlador usado nos experimentos deste trabalho. [25]
- **RYU** - Além do protocolo OpenFlow, suporta outros protocolos, tais como Netconf e OF-config.[30]
- **OpenDaylight** - É um projeto de código aberto para SDN mantido pela Linux Foundation [3].[31]

	NOX	**POX**	**Ryu**	**Floodloght**	**ODL**
Linguagem	C++	Python	Python	JAVA	JAVA
Performance	Rápido	Lento	Lento	Rápido	Rápido
Distribuído	Não	Não	Sim	Sim	Sim
OpenFlow	1.0/1.3	1.0	1.0 ~1.4	1.0	1.0/1.3
Aprendizado	Moderado	Fácil	Moderado	Crescente	Crescente

Figura 2.4: Controladores SDN

[3]https://en.wikipedia.org/wiki/Linux_Foundation

2.2 *Switches* OpenFlow

Os equipamentos de encaminhamento de redes tradicionais possuem memórias do tipo *Ternary Content Access Memory* - TCAM, onde além do serviço de encaminhamento são também executados serviços como filtragem por regras de acesso, *Network Address Translation* (NAT), priorização de tráfego e balanceamento de carga, entre outros. No paradigma de rede tradicional, como a arquitetura é fechada, as implementações desses serviços são realizadas de forma distinta por cada fabricante. Em equipamentos que suportam OpenFlow, o conjunto de instruções do protocolo pode ser implementado de diferentes formas, já que as regras que definem como cada fluxo será tratado são definidas pelo protocolo. Um *switch* OpenFlow (OF) pode ser decomposto em três partes:

a) **Tabela de Fluxos:** possui uma lista de regras de fluxo onde cada regra possui uma ação associada, de tal forma que, quando um pacote ingressa no *switch*, essas regras são varridas sequencialmente até que uma regra de fluxo case com as características do pacote e a respectiva ação seja processada; com o fluxo ao qual pertencem; Cada entrada na tabela de fluxo contém:

- *match fields*: Campo usado verificar correspondência com fluxos. Eles consistem na porta de entrada e nos cabeçalhos dos fluxos e, opcionalmente, nos metadados especificados por uma tabela anterior;
- *priority*: Utilzado para a precedência do fluxo;
- *counters*: Atualizados quando os fluxos são correspondidos.
- *instructions*: Usandas para modificar o conjunto de ações ou o processamento do pipeline;
- *timeouts*: Quantidade máxima de tempo ou tempo ocioso antes que o fluxo seja removido da tabela de fluxos;
- *cookie*: Valor opaco escolhido pelo controlador. Pode ser usado pelo controlador para filtrar estatísticas de *match*, modificação e exclusão de fluxo. Não usado ao processar fluxos.

b) **Canal Seguro:** serve para conectar o *switch* ao controlador utilizando uma conexão criptografada através dos protocolos TLS (*Transport Layer Security*)[4] ou

[4]https://tools.ietf.org/html/rfc5246

SSL (*Secure Socket Layer*)[5], o que permite que mensagens sejam trocadas entre o controlador e o *switch*;

c) **Protocolo OpenFlow:** é o protocolo de padrão aberto usado para comunicação entre *switch* e controlador, o que, dentre outras funcionalidades, permite a programação da tabela de fluxos de forma independente do fabricante ou das características do *switch*.

Os *switches* OF podem ser físicos ou virtuais. Dentre os *switches* virtuais, tem-se o conhecido e amplamente adotado Open vSwitch [32], que possui código aberto e é composto de um componente residente no *kernel* e outro residente no espaço do usuário. Outros exemplos são as plataformas abertas programáveis desenvolvidas pela UFES (Universidade Federal do Espírito Santo) [33] e pelo CPqD, esse último conhecido como Softswitch13 [34]. Como exemplo de *switch* físico, tem-se o Pica8, que executa um sistema operacional chamado PicOS com capacidade de encaminhamento de camada 2 e camada 3 e suporte ao OpenFlow através do Open Vswitch. O PicOS é baseado no projeto XORP [35].

2.2.1 Componentes do *switch* OpenFlow

Segundo a especificações da ONF [2], um *switch* OpenFlow possui uma ou mais tabelas de fluxo além de uma tabela de grupos, nas quais são realizadas as pesquisas das regras para o encaminhamento de pacotes. A Figura 2.5 apresenta um esquema genérico de um *switch* OF se comunicando com múltiplos controladores.

Usando o protocolo OpenFlow, o controlador SDN pode adicionar, atualizar ou excluir entradas nas tabelas de fluxo. Cada tabela de fluxo no *switch* contém um conjunto de entradas (regras) onde cada entrada consistem em campos de correspondência, contadores equivalências e um conjunto de ações a serem aplicadas aos pacotes correspondentes, conforme Figura 2.6.

2.2.2 Tipos de *switches* SDN

Os *switches* compatíveis com OpenFlow podem ser OpenFlow puro ou OpenFlow híbrido. Os *switches* OpenFlow puro suportam apenas operações OpenFlow, ou seja, todos os fluxos são processados pelo *pipeline* OpenFlow e não podem ser processados de outra forma. Os *switches* híbridos suportam a operação tanto pelo

[5]https://tools.ietf.org/html/rfc6101

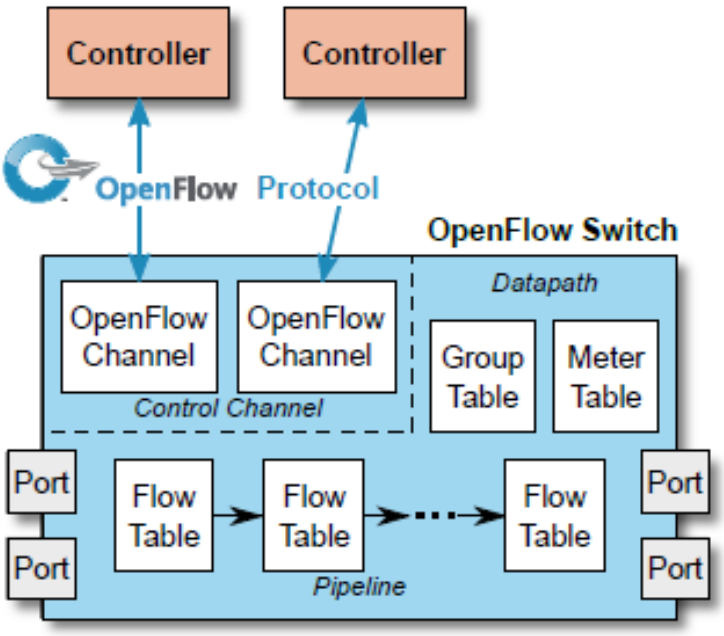


Figura 2.5: Switch OpenFlow
Fonte: Open Networking Foundation (2020)

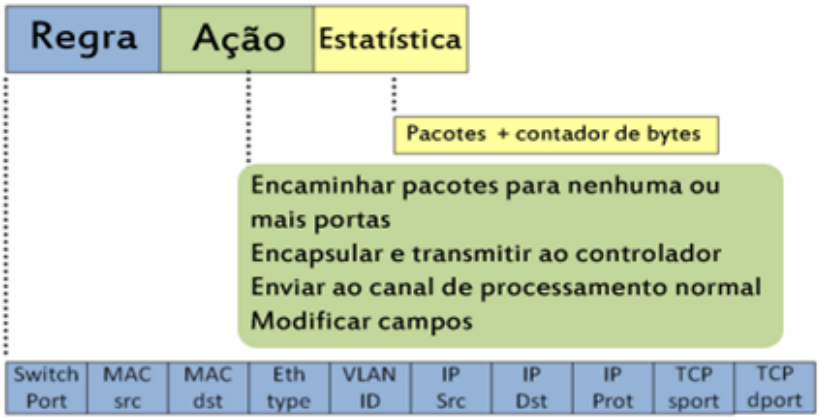


Figura 2.6: Entrada de fluxo em uma tabela
Adaptado de: http://marcial.larces.uece.br/pesquisa/hermesnet

pipeline OpenFlow quanto a operação de comutação Ethernet tradicional (comutação L2, de *Layer* 2), marcação de VLAN, roteamento L3, criação de listas de acesso e marcação de pacotes para priorização de tráfego.

2.2.3 Gerenciamento *in-band* e *out-of-band*

Em SDN, é necessário a troca de mensagens entre controlador e *switches*, que podem ser de controle ou comando. Essas mensagens podem ser trocadas de duas maneiras, *in-band* ou *out-of-band*, conforme mostrado na Figura 2.7.

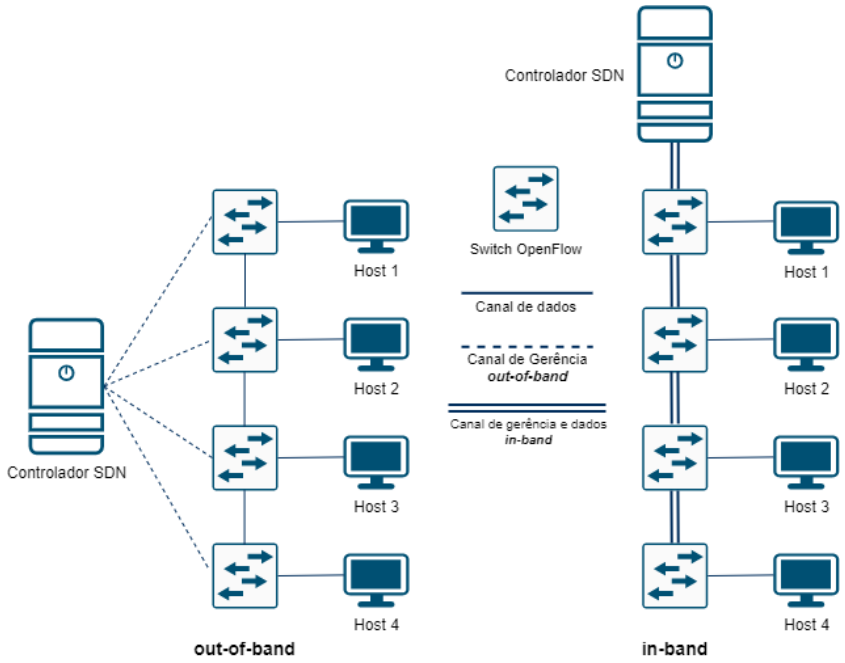


Figura 2.7: Gerenciamento nos modos *out-of-band* e *in-band*, respectivamente

No modo *in-band*, os *switches* compartilham o mesmo canal usado para controle e tráfego de dados. No modo *out-of-band*, os *switches* usam canais diferentes para controle e para tráfego de dados.

No caso do modo *out-of-band*, o controlador está diretamente conectado a uma interface de rede exclusiva de cada um dos *switches* e há completo isolamento entre o tráfego de controle e o tráfego de dados. Embora desejável, esse modo pode se tornar complexo de se implementar em alguns cenários, como no caso de escritórios distribuídos. Pode também se tornar custoso devido ao requisito de uma porta física "extra" em cada *switch*, o que pode ser caro para se obter em uma rede real.

Já no modo *in-band*, não há necessidade de portas físicas "extras" nos *switches* para controlar o tráfego. Nesse caso, uma porta virtual, denominada "local", é definida para permitir que o controlador interaja com o *switch*. Essa tarefa é, no entanto, especialmente desafiadora no processo de inicialização [36], pois os *switches* precisam estabelecer os caminhos de rede até o controlador através dos outros *switches* da rede.

2.3 Protocolo OpenFlow

O OpenFlow [2] é um protocolo padrão para comunicação entre controlador e *switches* SDN através da chamada *interface sul* do plano de controle, ou seja, a

interface entre camada de controle e camada de infraestrutura, conforme apresentado na Figura 2.8. O OpenFlow, promovido pela ONF, tornou-se o padrão adotado pelo mercado. Foi lançado oficialmente em 2009 e já se encontra na versão 1.6.

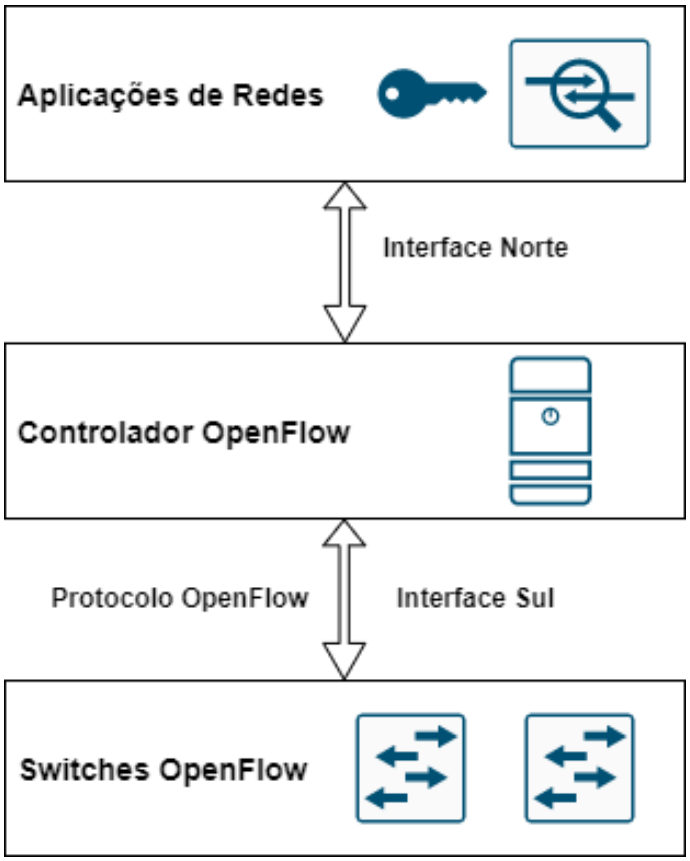


Figura 2.8: Protocolo Openflow

Uma rede SDN que utiliza o protocolo OpenFlow é basicamente composta por um ou mais controladores SDN, aplicações e equipamentos chamados de *datapaths*, que podem se comportar como roteadores ou *switches*, conforme a programação de regras em suas tabelas de fluxo. Sherwood [37] apresentou uma comparação entre uma rede tradicional e uma SDN que pode ser vista na Figura 2.9. Conforme mostrado nessa figura, em uma arquitetura tradicional, tanto o controle quanto a lógica de encaminhamento estão localizados internamente no *switch*. Já em uma SDN, o *switch* executa apenas a lógica de encaminhamento, enquanto que a lógica de controle é executada no controlador e a comunicação entre ambas camadas se dá através do protocolo OpenFlow.

2.3.1 Tipos de mensagens suportadas pelo OpenFlow

No protocolo OpenFlow há três tipos diferentes de mensagens trocadas entre o controlador e o *switch*:

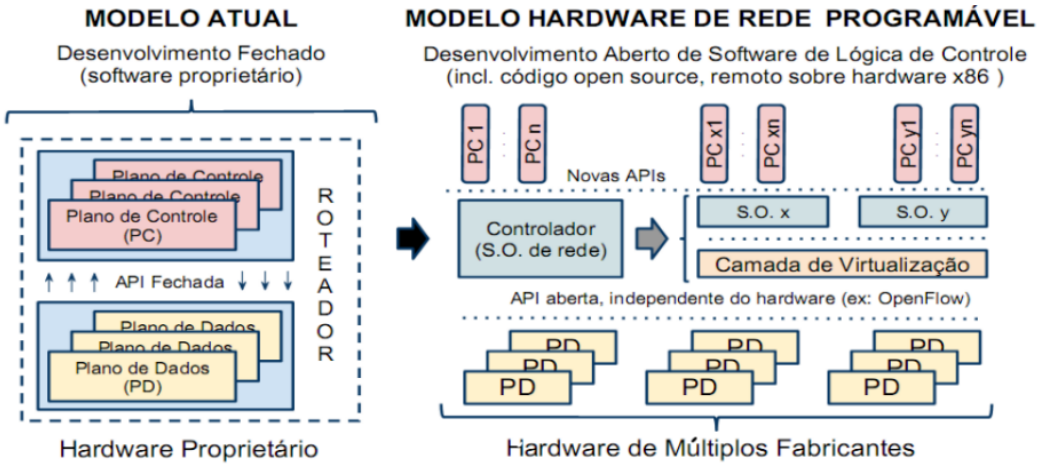


Figura 2.9: Modelo de rede atual versus Modelo de rede programável
Fonte: Rothenberg et al. (2010)

1. Controlador para *switch*, usadas para coletar informações ou enviar configurações, constituindo-se das seguintes mensagens:

   - *Features Request* - O controlador requisita as características do *switch*;
   - *Configuration* - Usada para configurar ou solicitar configurações do *switch*;
   - *Modify-State* - Usada para adicionar, deletar e modificar entradas na tabela de fluxos e também setar propriedades nas portas do *switch*;
   - *Read-State* - Coleta informações estatísticas;
   - *Send-Packet* - Utilizada para enviar pacotes por uma determinada porta do *switch*;
   - *Barrier* - Usada para garantir que as dependências foram atendidas ou para receber notificações de operações finalizadas.

2. Assíncronas, que são geradas pelo *switch* e enviadas ao controlador para notificar sobre eventos na rede e mudanças no estado do *switch*, constituindo-se das seguintes mensagens:

   - *Packet-In* - Utilizada para notificar o controlador de um pacote que ingressou e não corresponde a nenhuma regra de fluxo na tabela do *switch*;
   - *Flow-Removed* - Enviada ao controlador pelo *switch* quando um fluxo é removido da tabela por inatividade, seja por *Idle Timeout* ou *Hard Timeout*, a serem explicados adiante (na Seção 2.3.3);
   - *Port-Status* - Enviada ao controlador sempre que há mudanças nas configurações de uma porta;
   - *Error* - Notificações de erros ao controlador.

3. Simétricas, que podem ser geradas tanto pelo controlador quanto pelo *switch*, constituindo-se das seguintes mensagens:

    - *Hello* - Trocadas entre o controlador e o *switch* durante o processo de estabelecimento da conexão;
    - *Echo* - Usada para identificação de latência, largura de banda e existência de conectividade;
    - *Vendor* - Usada para permitir a implementação de funcionalidades adicionais nos *switches* OpenFlow.

Como parte do processo de inicialização de uma rede OpenFlow, ocorre um processo de *handshake* entre controlador e *switches* durante o qual mensagens do tipo *Features Request* são enviadas pelo controlador solicitando informações de configuração dos *switches* conectados a rede, conforme Figura 2.10. O próximo passo é o controlador conseguir formar uma visão completa da topologia da rede, sendo esse processo tratado pelo mecanismo de descoberta de topologia, apresentado a seguir.

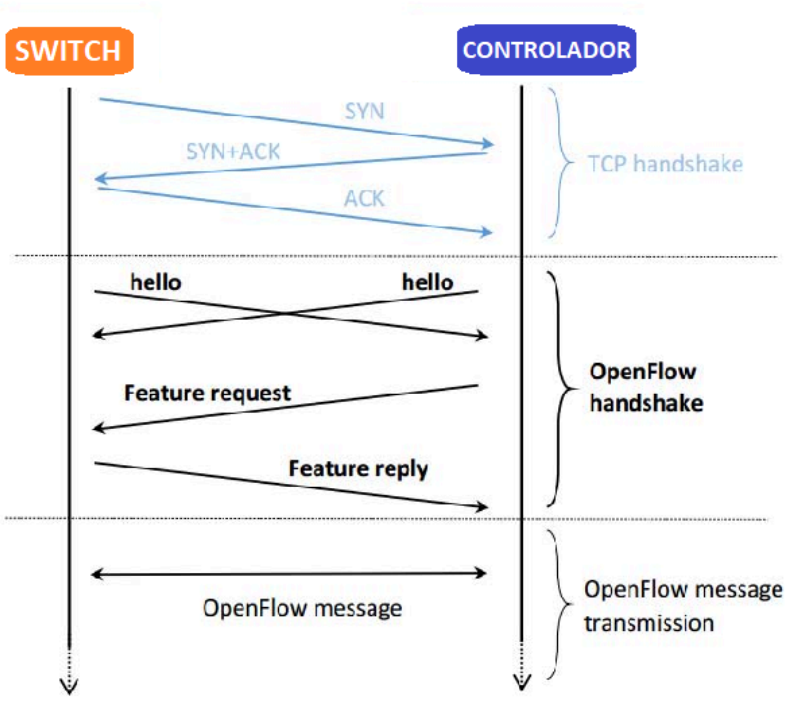


Figura 2.10: Handshake Openflow
Fonte: Open Networking Foundation (2020)

2.3.2 Descoberta de topologia SDN

Para que um controlador SDN possa fornecer serviços de forma confiável é necessário ter informações atualizadas sobre o estado da rede, para isso é necessário

um mecanismo de descoberta de topologia eficiente.

Um controlador SDN não precisa iniciar uma processo de descoberta de quais *switches* compõe a rede, pois esse processo ocorre durante o processo de inicialização do *switch*. Nesse momento, uma conexão com o controlador é estabelecida e os *switches* anunciam a sua existência.

Como não existe um padrão definido para o método de descoberta de topologia em SDNs usando OpenFlow, a maioria das controladores implementa a descoberta de topologia da mesma forma, através do chamado *OpenFlow Discovery Protocol* (OFDP). Esse método utiliza o protocolo *Link-Layer Discovery Protocol* (LLDP) para a descoberta dos enlaces entre os dispositivos [38], fornecendo ao controlador uma visão da topologia da rede. O formato da mensagem LLDP é apresentado na Figura 2.11

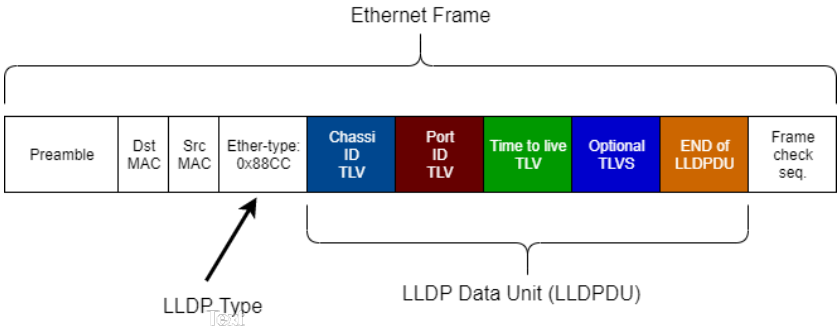


Figura 2.11: Quadro LLDP

O controlador SDN cria pacotes LLDPs e os envia ao *switch* de maneira que cada pacote seja retransmitido por cada porta onde haja dispositivo conectado, conforme Figura 2.12. Nesse exemplo, quando o *switch* S1 recebe os pacotes LLDP via *packet-out* do controlador, ele envia um para cada porta ativa, exceto a porta por onde recebeu a mensagem do controlador. Esses pacotes LLDPs possuem o ID da porta e a identificação do chassi. Por exemplo, o pacote LLDP com Port ID igual a 1 deve ser enviado na Porta 1, o pacote com Port ID igual a 2 na Porta 2 e assim por diante. Como em todos os *switches* há uma regra pré-instalada, informando que qualquer pacote LLDP recebido deve ser encaminhado ao controlador via mensagem *Packet-In*, o mapeamento da topologia é realizado por todos os dispositivos na rede. Esse processo é realizado periodicamente com intervalo padrão típico de 5 segundos.

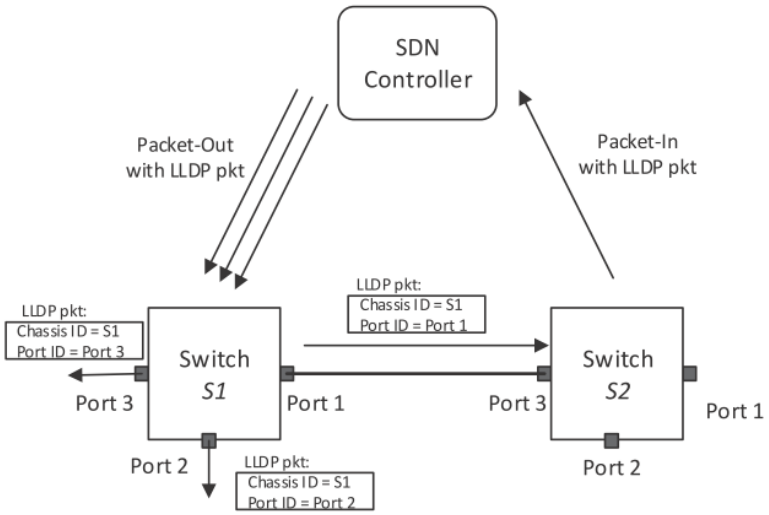


Figura 2.12: descoberta de topologia
Fonte: baseado em [38]

2.3.3 Tempo para expiração de regras de fluxo

Em redes OpenFlow, como todo tráfego é encaminhado conforme as regras de fluxo instaladas nas tabelas dos *switches*, e como os *switches* não podem armazenar um número ilimitado de regras de fluxo, dois mecanismos para controlar o tempo de vida das regras são utilizados: o *hard timeout* e o *idle timeout*. O *hard timeout* é utilizado para remover as regras de encaminhamento da memória do *switch* após a expiração desse contador, mesmo que ainda haja tráfego de dados correspondente a essa entrada na tabela. O *idle timeout* é utilizado para remover as regras de encaminhamento de fluxo que não estão sendo mais utilizadas pelo respectivo tempo nele configurado. Valores padrões de tempo são adotados pelos controladores mais usuais do mercado, conforme apresentado na Tabela 2.13 desabilita a ação desse temporizador.

Controlador SDN	idle_timeout(s)	hard_timeout(s)
ODL	0	0
Floodlight	5	0
POX	10	30
Ryu	0	0
Beacon	5	0

Figura 2.13: Timeout padrão controladores SDN

Diferentes valores de tempos limites foram adotados em trabalhos recentes, porém esses tempos são fixos e não levam em consideração as características de diferen-

tes fluxos. A ausência de um mecanismo inteligente para ajustar adaptativamente o tempo limite atribuído às entradas de fluxo pode levar ao uso ineficiente dos recursos de memória para as tabelas de fluxo e uma sobrecarga desnecessária dos controladores para o atendimento de mensagens destinadas a eles.

Por exemplo, na Figura 2.14, caso o fluxo f_1 seja atribuído com um tempo limite T_1 menor que seu intervalo de pacotes I_1, esse fluxo será removido prematuramente, gerando eventos de consulta ao controlador sobre qual regra de encaminhamento deve ser seguida. Essa sobrecarga de consultas pode degradar o funcionamento do controlador e novos fluxos podem ser descartados. Já para exemplo do fluxo f_2, cujo o tempo de expiração do fluxo T_2 é maior que intervalo de chegada de pacotes, esse fluxo permanecerá na memória consumindo recursos apesar de não haver mais pacotes para serem processados por esta regra.

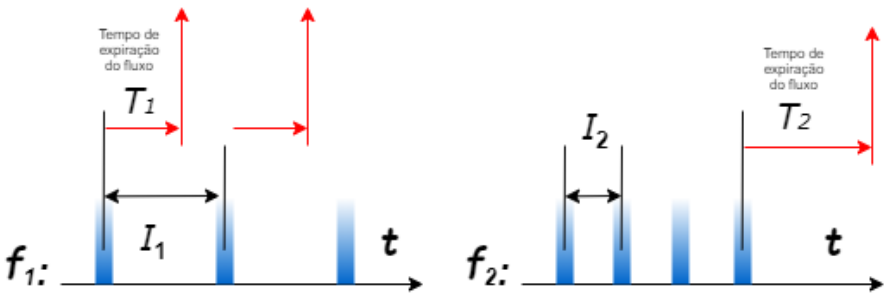


Figura 2.14: Exemplo de tempo limite

Atualmente, para garantir uma pesquisa rápida e um bom desempenho de encaminhamento, as entradas das tabelas de fluxo usam memórias TCAM[6]. Porém, como as TCAMs consomem muita energia, consequentemente as tabelas de fluxo possuem tamanhos limitados. Por exemplo, o *switch* série HP5400 zl habilitados para OpenFlow suportam apenas cerca de 1000 entradas em suas tabelas de fluxo[7]. Intuitivamente, uma carga de tráfego mais pesada ou um tempo limite maior levará a mais ocupação da TCAM. Em casos extremos, a tabela pode exceder seu limite, resultando em perda de pacotes e, por esse motivo, o tamanho da tabela de fluxo deve ser levado em consideração.

Geralmente, uma combinação de TCAM e SRAM[8] *(Static Random Access Memory)* é usada para encaminhar um pacote com base nas ações apropriadas. A TCAM é uma matriz de *bits* totalmente associativa, usada para armazenar entradas de fluxo. Qualquer pacote que chegar ao *switch* passará pela operação de pesquisa

[6]https://en.wikipedia.org/wiki/Content-addressable_memory#Ternary_CAMs
[7]https://h20628.www2.hp.com/km-ext/kmcsdirect/emr_na-c039914891.pdf
[8]https://en.wikipedia.org/wiki/Static_random-access_memory

para encontrar uma correspondência na TCAM. O pacote recebido pode conter entradas curinga no campo de endereço, o que pode levar a várias correspondências na TCAM. Nesses casos, a entrada que corresponde ao prefixo mais longo é selecionada. Adjacente a TCAM, há uma matriz de SRAM que contém a parte de ação para encaminhar o pacote pela porta apropriada no *switch*.

Atualmente os *switches* têm TCAM de tamanho que varia de 1Mbit - 2Mbit. Isso indica que os tamanhos atuais do TCAM são insuficientes para armazenar o número de entradas ativas, conforme será apresentado a seguir. Em geral, os *switches* comerciais empregam dois tipos de células de memória, um chip apenas SRAM para armazenar entradas completas - também conhecidas como **Entradas de Correspondência Exata**, que podem ser consultadas usando um algoritmo de hash direto e outro uma combinação TCAM/SRAM.

A SRAM consome cerca de 50 vezes menos energia com 20 vezes mais latência de acesso em comparação com a TCAM. Como no ambiente SDN (por exemplo, DCs), a pesquisa de baixa latência é preferida devido à natureza e criticidade dos aplicativos, o TCAM vence o SRAM mesmo para entradas de correspondência exata.

2.3.4 Impacto nos switches que usam OpenFlow

O OpenFlow caracteriza um fluxo usando tuplas de 15 campos. Existem dois desafios principais de design para *switches* OpenFlow. Primeiro, o volume de fluxos reconhecíveis pelo *switch* explode. Teoricamente, são possíveis 2356 entradas de fluxo a qualquer momento (356 bits para definir um fluxo). O número real de entradas de fluxo depende dos parâmetros da rede, como número de ações, hosts, fluxos etc. Segundo, a natureza das operações OpenFlow é bem diferente. Para entender o estresse experimentado pelos *switches* OpenFlow em SDN, percorremos uma rede DC e estudamos o volume de entradas de fluxo geradas e a natureza das operações executadas.

2.3.5 Explosão da tabela de fluxo

Nosso objetivo é mostrar o número de entradas de fluxo em um *switch* que pode ser gerado por segundo para as características de fluxo descritas em [39] e [40]. Tomamos um exemplo de uma topologia de rede DC simplista. Usamos uma topologia de Fat Tree proposta em [41]. Como exemplo, considere um layout de rede de data center de 11.520 portas. Neste exemplo, o datacenter consiste em 24 linhas, cada

uma com 12 racks. Cada rack possui 40 máquinas interconectadas por um switch *Top of the Rack (ToR)*[9]. Em um ambiente não virtualizado, isso pode significar apenas 11.520 máquinas, mas com a virtualização cada máquina pode hospedar no mínimo 15 VMs[10], aumentando assim o número de hosts finais para aproximadamente 172.000 nós. O número de fluxos que chegam ao *ToR* é dado por: taxa de chegada de fluxo x número de servidores x número de VM (considerando o modo de *bridge* na VM). De acordo com a fórmula acima, para diferentes características de tráfego mostradas em [39], o número de fluxos que chegam ao *ToR* é estimado em cerca de 75.000 - 1,3 milhões de fluxos/min. Considere um TCAM de tamanho 1 Mbit, o número de entradas de 356 bits (um tamanho de entrada de fluxo na rede SDN que consiste em 15 tuplas de campo) que pode ser armazenado é de cerca de 2800. Os TCAMs comerciais são de cerca de 18Mbit e esse número é muito pequeno comparado o número de fluxos ativos chegando a um *switch*.

2.3.6 Agregação de regras de fluxos

Para oferecer suporte ao número grande de aplicativos de rede, as regras do OpenFlow são mais complexas do que as regras de encaminhamento nos roteadores IP tradicionais. Por exemplo, uma ACL *Access Conrol List* requer correspondência nos endereços IP de origem/destino, números de porta e protocolo [42], enquanto um balanceador de carga pode corresponder apenas nos prefixos IP de origem e destino [43]. Essa correspondência complicada pode ser bem suportada usando o TCAM, pois todas as regras podem ser lidas em paralelo para identificar as entradas correspondentes para cada pacote. Conforme já mencionando anteriormente, TCAM é caro e consome muita energia, o tamanho do TCAM no chip geralmente é limitado. Muitos trabalhos existentes na literatura tentaram resolver esse problema limitado de espaço para regras. Por exemplo, começando com o OpenFlow v1.1 e além, em vez de ter uma tabela de fluxo, há um pipeline composto por várias tabelas de fluxo. Embora essa proposta possa lidar com um grande número de regras, ela adiciona complexidade ao padrão [44].

2.3.7 Tabelas de grupo no OpenFlow

As tabelas de grupos OpenFlow foram introduzidos no OpenFlow 1.1 como uma maneira de executar operações mais complexas em pacotes que não podem ser definidos apenas em um fluxo. Existem diferentes tipos de grupos definidos na espe-

[9]http://bradhedlund.com/2009/04/05/top-of-rack-vs-end-of-row-data-center-designs/
[10]https://www.redhat.com/pt-br/topics/virtualization/what-is-a-virtual-machine

cificação do OpenFlow para atender a vários propósitos. Ao ser encaminhado para uma tabela de grupo correspondente, os fluxos já foram devidamente processados, ou seja, já houve *"matching"* desse fluxo com alguma regra existente.

Uma tabela de grupo é composta pelos seguintes campos, conforme especificação do OpenFlow pela ONF[11]:

- *Group Identifier*: um inteiro de 32 bits que identifica unicamente o grupo OpenFlow.
- *Group Type*: determina a finalidade do grupo.
- *Counters*: contabilizam os fluxos processados por um grupo.
- *Action Buckets*: uma lista ordenada de depósitos de ação (*buckets*), em que cada *bucket* da lista contém um conjunto de ações a serem executadas e parâmetros associados. As ações em um *bucket* são sempre aplicadas como um conjunto de ações.

Conforme a especificação da ONF, não é necessário que um *switch* dê suporte a todos os grupos, apenas os designados com *required*, e o controlador também pode consultar o *switch* sobre quais grupos são suportados. Existem quatro tipos de grupos:

- *Required* **Indirect**: Este grupo suporta apenas um único *bucket*. Permite que várias entradas de fluxos apontem para um único grupo em comum. Ele pode ser usado para agrupar fluxos que tenham como destino o próximo salto.
- *Required* **All**: Executa todos os *buckets* no grupo. Este grupo é usado para encaminhamento *broadcast* ou *multicast*. O fluxo é clonado para cada *bucket*, que é processado conforme instrução contida neste *bucket*, ou seja, várias ações podem ser executadas para o mesmo fluxo.
- *Optional* **Select**: Foi criado com propósito de balanceamento de carga. Cada *bucket* no grupo *Select* tem um peso associado e cada fluxo que entra no grupo é enviado para um único *bucket* através de um algoritmo de seleção tipo *round robin*. O peso de um *bucket* é dado como um parâmetro especial e cada *bucket* em um grupo *Select* contém uma lista de ações. Então, quaisquer ações suportadas pelo OpenFlow podem ser usadas em cada *bucket* e, assim como no grupo *All*, os *buckets* não precisam ser iguais.

[11] https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

- *Optional* **Fast failover**: Cada *bucket* está associado a uma porta de interface e/ou grupo específico que controla se esta porta ou grupo está ativo. Os *buckets* são avaliados na ordem definida pelo grupo e o primeiro *bucket* associado a uma porta/grupo ativo é selecionado. Este tipo de grupo permite que o *switch* altere o encaminhamento sem exigir informações para o controlador, caso uma porta ou grupo associado a um *bucket* se torne inativa. Se nenhum *bucket* estiver ativo, os fluxos serão descartados.

2.4 Topologias de um *Data Center*

Os *data centers* dão suporte de infraestrutura para serviços baseados em *cloud* ou *bare metal*. Assim, as características de desempenho e confiabilidade dos *data centers* terão impacto significativo na escalabilidade desses serviços. Em particular, a rede do *data center* precisa ser ágil e reconfigurável para responder rapidamente às demandas de aplicativos e requisitos de serviço em constante mudança.

A topologia *Fat-Tree* [41] preza o uso de *switches* e a garantia da disponibilidade dos serviços de rede, devido aos múltiplos caminhos redundantes oferecidos. Essa arquitetura foca na presença de *switches* e não requer o uso dos servidores em nenhuma atividade de rede, conforme apresentado na Figura 2.15.

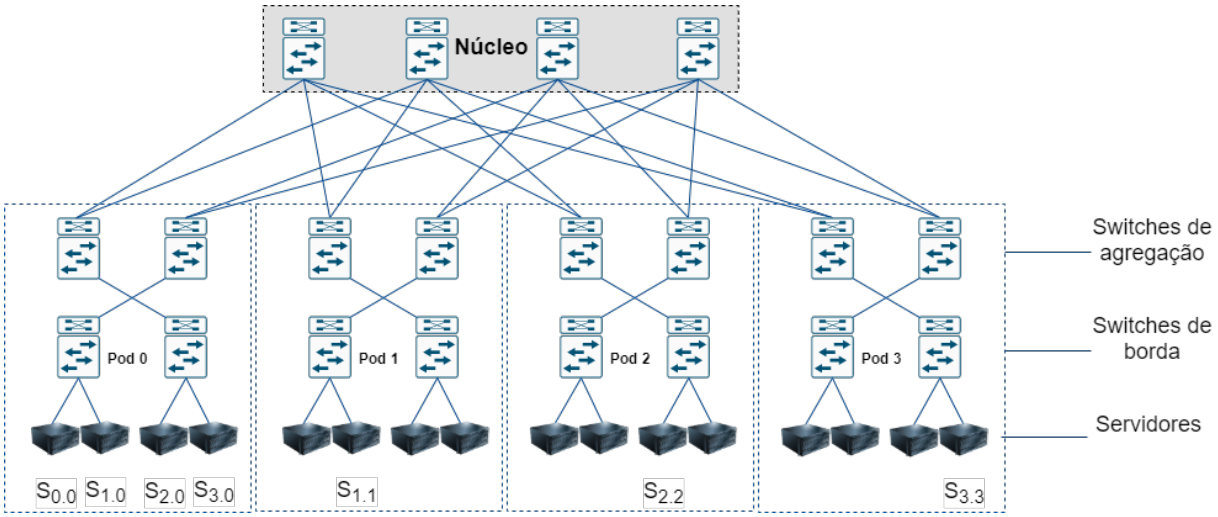


Figura 2.15: Topologia utilizada pela arquitetura Fat-Tree

Já na topologia *DCell* [45], os servidores têm um peso importante, visto que são utilizados para o encaminhamento de dados. Nessa topologia, uma estrutura de células são conectadas através de servidores, que desempenham o papel de garantir

a conectividade intra-célula conforme apresentado na Figura 2.16.

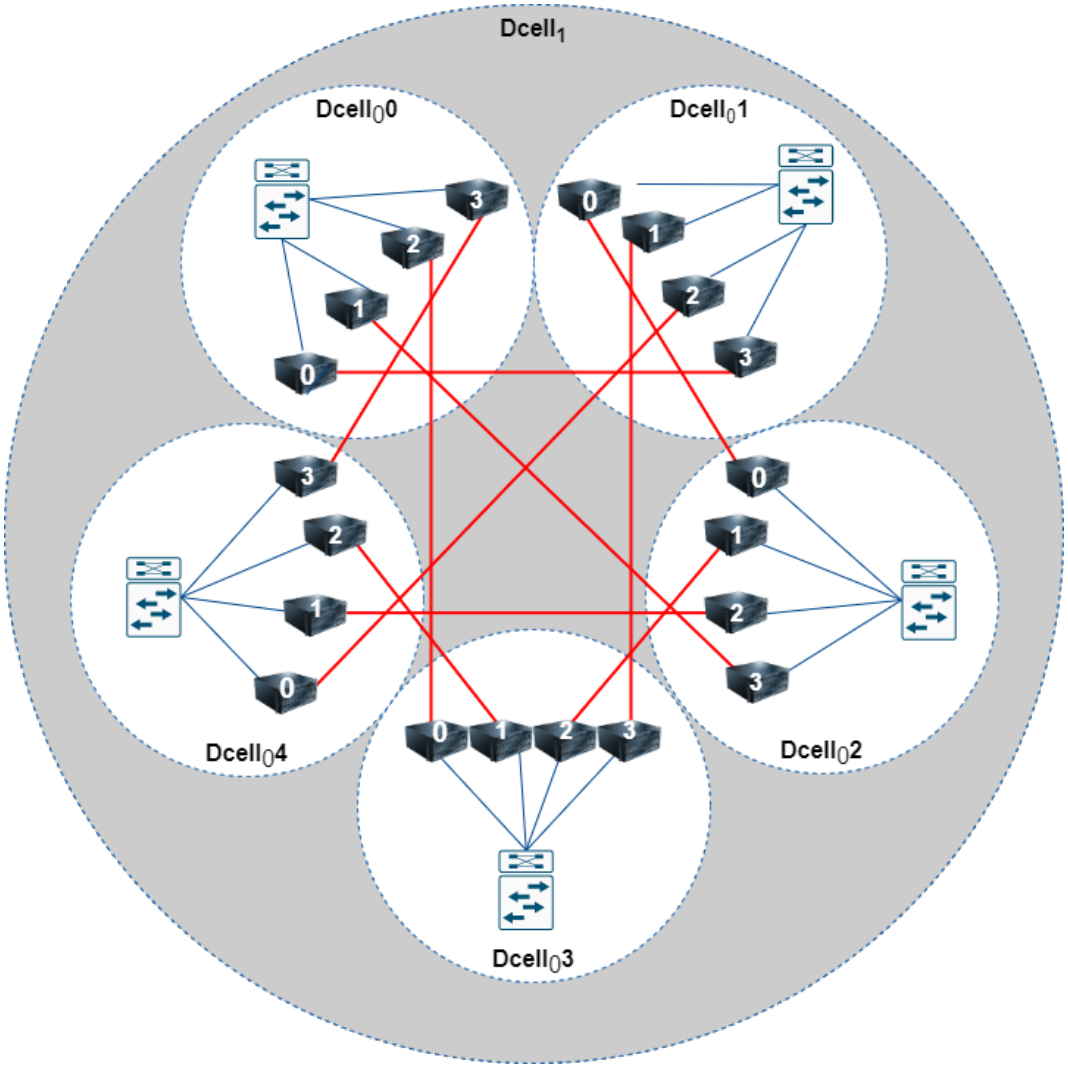


Figura 2.16: Topologia utilizada pela arquitetura DCell.

A topologia BCube [46] é apresentada na Figura 2.17. Trata-se de uma topologia que foi desenvolvida para ser embutida em contêineres selados para facilitar o seu deslocamento. A BCube também possui arquitetura modular na qual, diferente do DCell, seus módulos são interconectados por *switches*.

Já a topologia *Spine and Leaf*, apresentada na Figura 2.18, é uma topologia de rede de *data center* de duas camadas, útil para *data centers* que experimentam mais tráfego de rede leste-oeste,ou seja, o tráfego cliente/servidor. A arquitetura *Spine and Leaf* foi proposta por Charles Clos, um pesquisador da Bell Labs [47]. O novo conceito de Clos divide a rede de dados em duas camadas: a *Spine* e a *Leaf*. A camada *Leaf* consiste nos *switches* de acesso que conectam servidores, *firewalls*, roteadores e demais elementos de rede e computacionais. Não existe comunicação direta entre os *"leafs"* sem passar pela camada *Spine*. A camada *Spine* consiste na

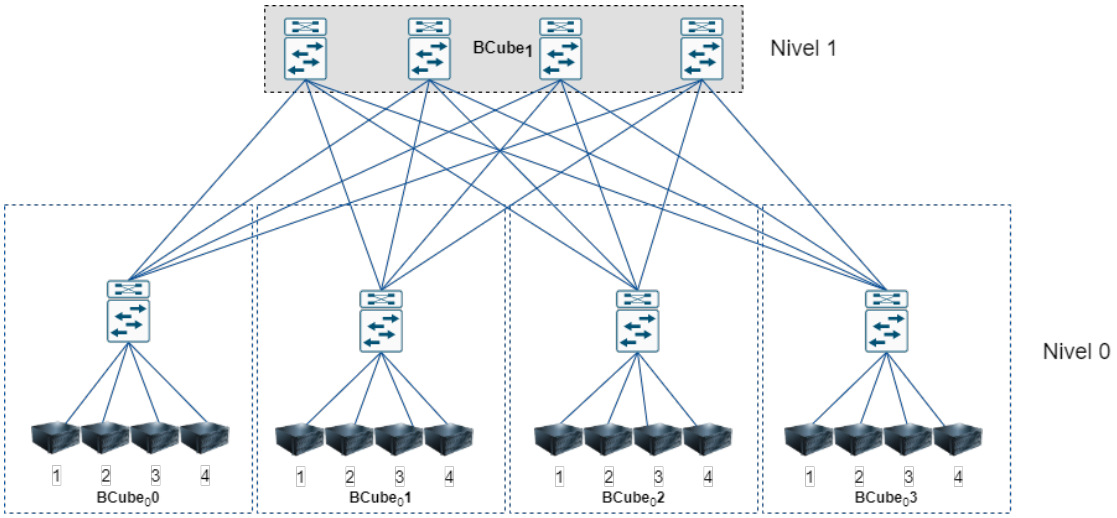


Figura 2.17: Topologia utilizada pela arquitetura BCube.

interconexão dos *"leafs"* e nenhum outro equipamento é conectado diretamente ao *Spine*.

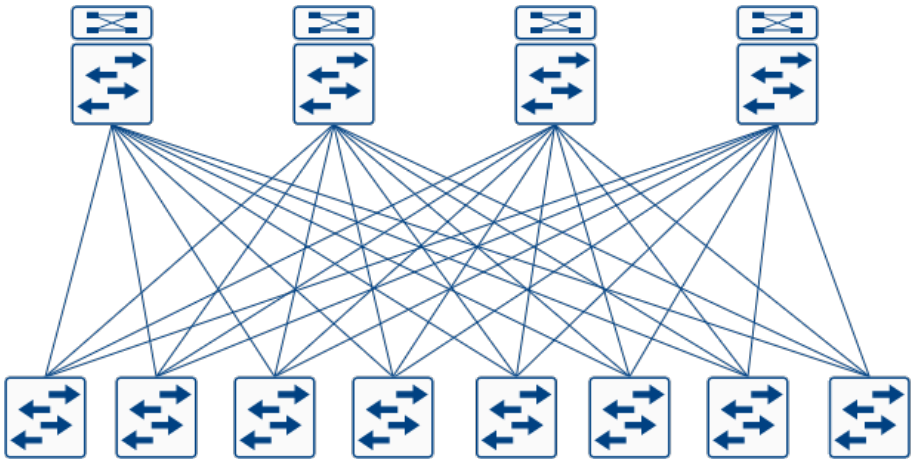


Figura 2.18: Modelo Clos (topologia *Spine and Leaf*)

A topologia *Spine and Leaf* pode fornecer uma conectividade em camada 2 ou em camada 3 (entre as camadas 2), o que traz um *design* favorável ao TRILL (*Transparent Interconnection of Lots of Links*), especificado na RFC 6325, ao SPB (*Shortest Path Bridging*), referente ao padrão IEEE 802.1aq 2012, e ao FabricPath, padronizado pela Cisco em 2012 para fins de balanceamento de tráfego entre os enlaces ativos como opção ao uso do protocolo *Spanning-Tree*. Nesse caso, o *Spine and Leaf* oferece uma topologia livre de *loops* para a interconexão dos servidores.

As redes que se valem da topologia *Spine and Leaf* obtém no máximo dois saltos entre origem e destino com múltiplos caminhos simultâneos. Dessa forma, essa topologia reduz o tempo de atraso na entrega do tráfego interno do *data center*,

reduz os tempos de convergência dos protocolos, reduz a possibilidade de falhas e aumenta o desempenho geral da rede.

2.5 Detecção e Recuperação de Falhas em SDN

2.5.1 Detecção de falhas

O tempo de recuperação de falha de um enlace é composto de duas partes: tempo de detecção da falha e tempo de reação. Portanto, para realizar a recuperação rápida de falhas de enlace é importante detectar e localizar a falha imediatamente após a ocorrência da mesma. Perda de Sinal (LOS, de *Loss of Signal*) e Detecção de Encaminhamento Bidirecional (BFD, de *Bidirectional Forward Detection*) [48] são mecanismos amplamente utilizados para detectar falhas de enlace. No entanto, o tempo de detecção de falhas usando o LOS pode ser de centenas de milissegundos, o que não atende ao requisito de desempenho de rede de nível de operadora. O BFD usa um protocolo simples de *"hello echo"* entre dois terminais finais de um caminho, porém o BFD não consegue localizar o enlace de falha no caminho. Para resolver esse problema, autores de [49] propõem um mecanismo que aplica o BFD entre os terminais de um enlace em vez de em um caminho.

Em [50] e [51], um mecanismo de monitoramento e localização de falhas é proposto. Com um ciclo de monitoramento cobrindo todos os enlaces em uma determinada rede, um pacote de "Monitoramento do Status do Link" (LSM, de *Link State Monitoring*) é usado para supervisionar o status do caminho. Os *switches* que receberem o pacote LSM enviam pacotes FLI (*Failure Location Identification*) para o controlador quando há uma falha de enlace. Assim, o nó de falha pode ser localizado precisamente de acordo com os pacotes FLI. Além disso, os autores de [52] propuseram um mecanismo de detecção de falha de enlace que é aplicado para detectar a falha do canal de controle entre *switches* e o controlador, usando o mecanismo de compartilhamento de informações entre múltiplos controladores.

2.5.2 Restauração de falha em SDN

O termo "recuperação de falha em SDN" é tipicamente relacionado a uma abordagem de caráter reativo cujo método típico é conhecido como *restauração rápida*, proposta em [10], [53] e [54]. Nesse método, o *switch* informa o controlador do evento de alteração de topologia devido a falhas. Em seguida, o controlador calcula

e instala o novo caminho para os fluxos interrompidos. O consumo de tempo desse processo pode ser grande, portanto pode causar perda de pacotes de dados e interrupção de serviço enquanto novas regras de encaminhamento não são instaladas. A criação de regras de fluxo sob demanda para reagir aos eventos de pacotes recebidos decorre da geração de mensagens *PACKET IN* para o controlador. Depois que as regras são instaladas nos *switches*, os pacotes são desenfileirados e encaminhados na rede. As regras recém-instaladas processarão qualquer pacote subsequente do fluxo sem intervenção adicional do controlador. Além disso, há um tempo gasto pelo controlador para configurar um caminho reativamente entre os nós de origem e de destino, definido como o intervalo que começa com a primeira mensagem de solicitação de provisionamento de fluxo recebida pelo(s) controlador(es) em sua interface sul e termina com a última mensagem de resposta de provisionamento de fluxo enviada do(s) controlador(es) em sua interface sul. A Figura 2.19 demostra o funcionamento quando os *switches* detectam que houve alguma queda de enlace. Conforme [55], o tempo de resposta para migração para o novo caminho de *backup* no método reativo costuma estar entre 75 e 130 ms.

Os métodos de restauração têm principalmente duas desvantagens. Primeiro, devido ao envolvimento do controlador, o consumo de tempo da recuperação da falha do enlace pode não atender à demanda, o que é demonstrado através de experimentos em [7] e [55]. Segundo, a computação e a instalação de caminhos para fluxos interrompidos podem introduzir uma carga pesada no controlador e, portanto, degradar o desempenho de toda a rede.

2.5.3 Proteção de falha em SDN

Para superar as desvantagens dos métodos de restauração de falha, os métodos de proteção de falha em SDN têm caráter proativo e usam caminhos de *backup* pré-programados estáticos para redirecionar os fluxos interrompidos sem o envolvimento do controlador. A chave dos métodos de proteção é usar um mecanismo de *handover* que pode desativar as entradas de fluxo de caminhos antigos e depois colocar automaticamente as entradas de fluxo do caminho de *backup* em uso. Nos *switches* habilitados para OpenFlow, não existe esse mecanismo, mas sim os de expiração de entrada de tabela, como o *idle timeout* e o *hard timeout*. Então, para perceber rapidamente a falha no enlace "protegido", [56] propõe um método chamado de *"auto-reject"*, que pode detectar a falha do enlace e desabilitar as entradas de fluxo correspondentes.

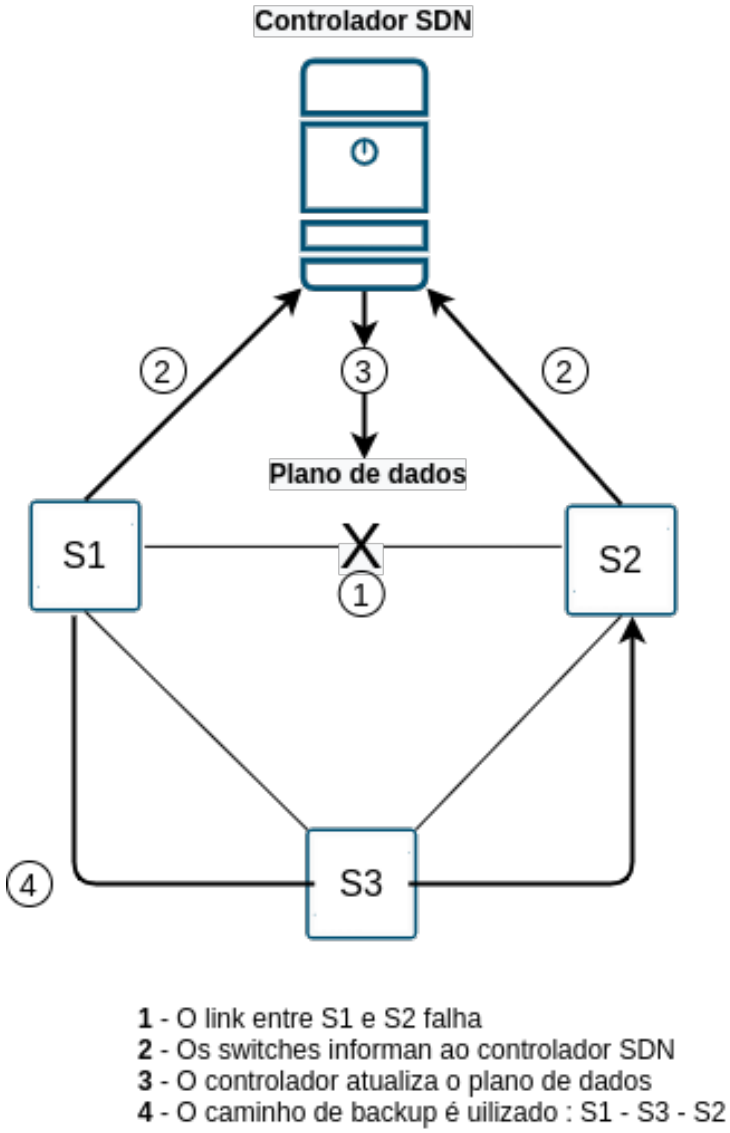


Figura 2.19: Recuperação de falha em SDN (método reativo)

Em [57], os autores propõem um mecanismo de expiração de entrada de fluxo para excluir automaticamente as entradas do caminho antigo quando ocorre falha no enlace. Em [58], os autores usam o OpenState para monitorar o pacote e o estado das portas do *switch* e realocar o fluxo para o novo caminho quando a porta do *switch* está inativa. No entanto, os mecanismos mencionados acima precisam de modificações adicionais no *switch* e são difíceis de serem aplicados na prática.

A Figura 2.20 demostra o funcionamento do método de proteção de falha quando os *switches* detectam que houve alguma queda de enlace. Conforme [55], o tempo de resposta para a migração para o novo caminho de *backup* no método de proteção de falha leva em torno de 45 ms.

O OpenFlow versão 1.1 fornece um mecanismo de "proteção rápida" chamado

tabela de grupo de failover, através da instanciação de listas de ações múltiplas para a mesma entrada na tabela de fluxo e aplicando-as de acordo com o status do enlace. O uso desse mecanismo não exige nenhuma modificação no protocolo do OpenFlow e pode conseguir o *handover* automaticamente, de acordo com o status que habilita ou não o uso das listas de ações.

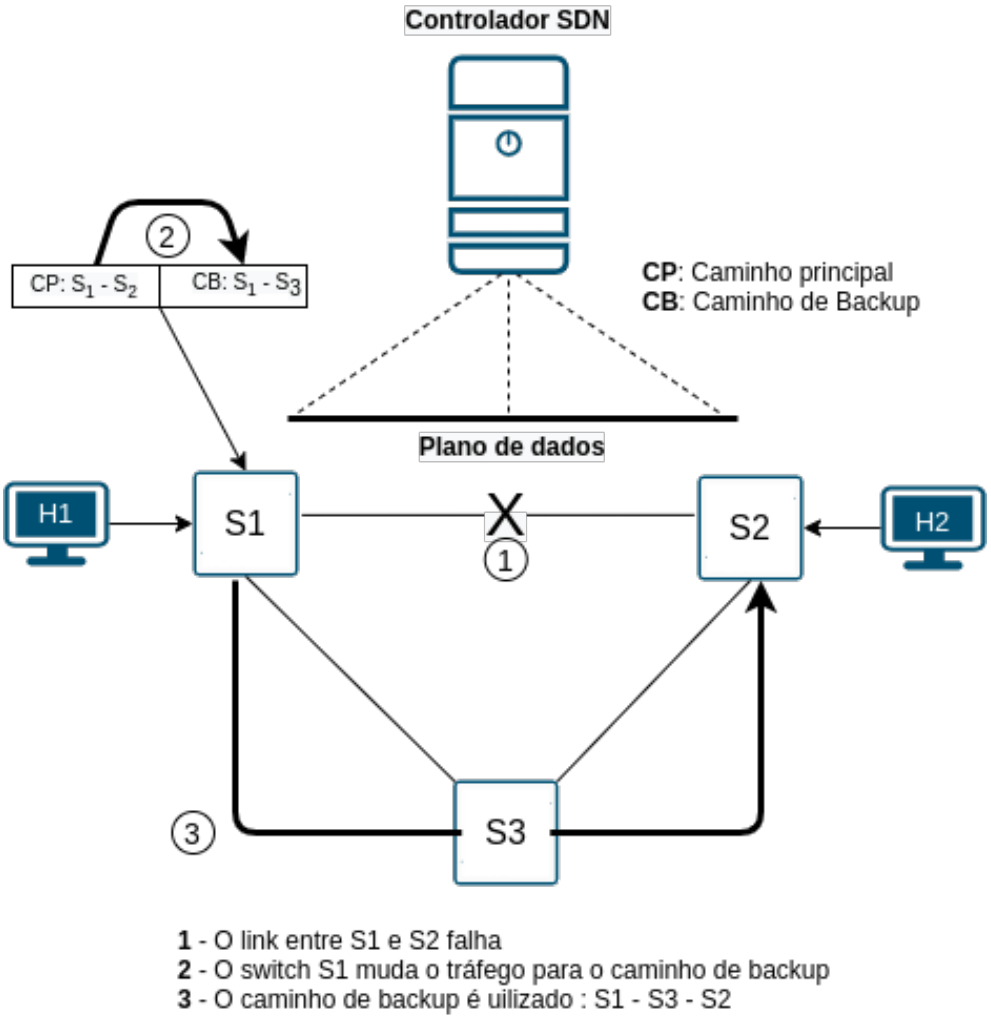


Figura 2.20: Proteção de falha em SDN, (método proativo)

2.6 Criação de Regras de Fluxo no Modo Ativo

A abordagem de criação de regras de fluxo no modo ativo tem semelhanças com a criação de fluxo nos modos reativo e proativo. A diferença entre os modos ativo e reativo é que, no modo ativo, todos os *switches* OpenFlow recebem regras OpenFlow de uma só vez, em vez do modo reativo que requer que cada *switch* OpenFlow se comunique individualmente com o controlador para criar uma nova regra de fluxo. Além disso, é importante destacar a diferença entre modo ativo e modo proativo.

O modo ativo não cria regras antecipadamente e espera que o pacote da origem ao destino chegue a um *switch* OpenFlow. Em seguida, o *switch* envia uma mensagem OpenFlow ao controlador que processa e calcula o caminho da origem ao destino para esse pacote específico. Posteriormente, o controlador envia a todos os *switches* no caminho novas mensagens OpenFlow instruindo-os a criar as respectivas regras de fluxo. Portanto, a abordagem de criação de fluxo ativo é uma "mistura" entre as abordagens reativa e proativa.

Os experimentos deste trabalho não contemplaram o modo ativo, porém isso não interferiu na validação da proposta.

2.7 Trabalhos Relacionados

Além dos trabalhos citados nas subseções 2.5.2 e 2.5.2, a saber [10, 53, 54, 7, 55], relativos a restauração de falhas em SDN, e [56, 57, 58], relativos a proteção de falhas em SDN, outros trabalhos também se relacionam a este.

Com o objetivo de proteger o plano de dados em SDN de possíveis paralisações usando uma abordagem proativa, alguns trabalhos foram direcionados para a instalação prévia de rotas de *backup* nos *switches*. Isso pode gerar alto consumo de TCAM, que tem espaço limitado, e alto consumo de energia. Assim sendo, serão também aqui apresentados trabalhos que buscam a otimização do uso de TCAM, porém sem o foco na resiliência do plano de dados.

A compactação das regras SDN foi discutida em [59], onde os autores propõem algoritmos para reduzir o tamanho das tabelas, mas apenas usando uma regra padrão. São armazenadas em *cache* as regras mais populares da TCAM, direcionando os pacotes com *cache miss* para o controlador. O trabalho [60] considera regras de compactação usando qualquer campo do cabeçalho do pacote, citando também que é possível considerar a agregação de múltiplos campos que compõe o cabeçalho do pacote, o que pode adicionar uma melhoria importante e permitir uma compressão mais eficiente das tabelas de fluxo. Como parte integrante do presente trabalho, propõe-se uma compactação de regras baseada em campos de fonte e destino dos fluxos, tais como IP ou Porta TCP/UDP, mas outros campos Openflow podem ser também considerados. Além das questões relacionadas ao consumo de energia da TCAM, outro campo de pesquisa envolve o tempo ideal para a remoção das regras das memórias dos *switches* [61] [62] [63]. No entanto, o presente trabalho não ataca esse problema, usando o valor padrão adotado pelo controlador para a remoção de

regras.

Os trabalhos em [64] e [23] compararam abordagens reativas e proativas e demonstraram em seus resultado uma melhora no desempenho do controlador quando há, no modo reativo, redução nas mensagens de controle. No entanto, algumas heurísticas poderão automaticamente definir caminhos alternativos com antecedência no modo proativo, melhorando o desempenho e mantendo a configuração mais simples. Nesse sentido, uma opção para criar novas regras de encaminhamento de fluxos dentro de uma rede OpenFlow é a de prover ao controlador uma visão global sempre atualizada da rede (*bird's eye view*) [65] [23], o que tende a minimizar o gasto de processamento para a obtenção desses novos caminhos.

Devido à capacidade restrita do controlador [24], alguns trabalhos são dedicados a minimizar o uso do controlador ou a ampliar sua capacidade. Os autores de [66] dedicaram-se a resolver o problema de ampliação da capacidade do controlador usando multi-controladores. O trabalho [67] propôs uma arquitetura de controlador distribuído elástico para melhorar o desempenho de controladores multi distribuídos. Os autores de [68] se concentraram em vários controladores, ajustando dinamicamente o número de controladores ativos e delegando a cada controlador um subconjunto de *switches* OpenFlow, de acordo com a dinâmica da rede. Os autores de [69] estudaram o problema de atribuição dinâmica de controlador como um problema de correspondência estável com transferências para minimizar o tempo de resposta do controlador. Todos esses trabalhos estudaram o problema de recursos limitados em SDNs de modo a obter a maximização da taxa de transferência de dados. Neste trabalho, propõe-se um método que migrará regras que são instaladas de forma reativa para o modo proativo, dessa forma minimizando o consumo de recursos do controlador SDN.

Em [70] foi realizado um estudo propondo o método *Fast Reroute* (FRR). Os mecanismos FRR existentes para redes MPLS ou IP podem ser portados para SDN, mas essa abordagem sofre de um grande problema. Os mecanismos FRR existentes exigem muitas entradas de encaminhamento adicionais para proteger todo o tráfego contra, pelo menos, *single link failures* (SLF) e *single node failures* (SNF) sem criar *loops*, sendo que os *switches* OpenFlow possuem tabelas de fluxo de tamanho moderado e têm limitações quanto ao número máximo de entradas suportado nessas tabelas, o que pode variar de fabricante para fabricante.

Até onde foi possível investigar, nenhum trabalho anterior propôs uma mudança automática e adaptativa entre os modos de proteção e recuperação em SDN, visando

diminuir a carga de processamento do controlador e otimizar o uso de TCAM nos *switches*. A proposta deste trabalho segue essa abordagem adaptativa.

3. Solução Proposta

Este capítulo explica a proposta de solução iniciando por uma descrição abrangente do funcionamento do *Adaptive Fault Recovery Control* (AFRC) para, em seguida, detalhar suas fases de funcionamento e seus mecanismos de detecção e implementação da resiliência no plano de dados através das tabelas *fast failover* especificadas no protocolo OpenFlow.

3.1 Visão Geral

De um modo geral, o AFRC objetiva uma resiliência do plano de dados através do uso de recursos nativos do OpenFlow, aplicando uma abordagem proativa de forma adaptativa, onde fluxos específicos, de maior criticidade e tipicamente mais sensíveis a atrasos durante um processo de recuperação de falha, como costuma ser o caso de fluxos de vídeo e voz, são privilegiados. Essa recuperação no modo proativo fará uso do grupo de tabelas *fast failover* para instalação de regras de fluxo de *backup*.

Conforme mostrado na Seção 2.7, outros trabalhos não se preocuparam com a remoção das regras de fluxos de *backup* instaladas proativamente, o que pode contribuir para gerar ou agravar um problema de alto consumo de memória nos *switches*. Neste trabalho, esse problema é levado em consideração no contexto geral da solução, o que inclui também a configuração dos temporizadores de *idle* e *hard timeout* para remoção das regras de *backup*.

A adaptabilidade desta solução se reflete na programação e no intercambiamento entre os modos reativo e proativo para fins de recuperação de falhas em função do estado de sobrecarga do controlador e de ocupação de memória dos *datapaths*, respectivamente. Conforme esses estados, a abordagem usada - reativa ou proativa

- pode ser trocada para regras de fluxo específicas, todas as regras de um mesmo *switch* ou de todos os *switches* da rede gerenciada.

3.2 *Adaptive Fail Recovery Control* AFRC

Para implementar a visão geral apresentada acima, este trabalho propõe o *Adaptive Fault Recovery Control*, aqui referenciado pela sigla AFRC. O AFRC apoia-se na arquitetura de controle centralizado de rede típica do SDN, o que confere ao controlador uma visão holística de toda a rede por ele gerenciada. Ou seja, o controlador SDN conhece todos os *switches* e as ligações entre eles, conseguindo dessa forma criar e manter atualizada uma estrutura contendo a topologia da rede e, assim, calcular os menores caminhos entre todos os pares de *end points*, ou seja, para todos os pares origem-destino da rede. Na camada 2 de redes legadas, esses cálculos e consequentes decisões de encaminhamento costumam ser obtidos por intermédio de protocolos que implementam algoritmos de *spanning tree*. Já em camada 3, esses caminhos costumam ser obtidos por protocolos de roteamento, como o OSPF. É importante, portanto, garantir que todos os *switches* que façam parte da rede sejam gerenciados pelo controlador SDN para assim permitir o uso do AFRC.

A Figura 3.1 apresenta um fluxograma que resume o funcionamento do AFRC. Trata-se de um mecanismo que foi idealizado para funcionar de forma autônoma, interagindo com as estruturas mantidas pelo controlador SDN e fazendo uso de sua *northbound* API para fins de instalação de regras nos *switches*. Por esse motivo, há toda uma fase inicial do AFRC na qual é verificado se há controlador SDN na rede e se há *switches* controlados por ele, de maneira que se possa adentrar a etapa de monitoramento do AFRC. Esse monitoramento objetiva identificar as situações que irão demandar uma intervenção do AFRC, aqui chamada de fase de "mitigação". São elas:

1. Número excessivo de mensagens PACKET-IN provenientes de um mesmo *switch*, o que pode tender a um consumo exagerado de recursos do controlador (uso de CPU ou banda na interface de rede entre controlador e *switch*), indicando assim a necessidade de migração para o modo de recuperação proativo, o que pode exigir também um procedimento de sumarização de regras OpenFlow para que se evite um alto consumo de espaço de memória no *switch*;

2. Número excessivo de regras OpenFlow instaladas num mesmo *switch*, aproximando-se assim do consumo total de memória desse dispositivo, o que aponta para a

necessidade de sumarizar regras de encaminhamento OpenFlow e de remover regras instaladas para fins de recuperação proativa de falhas, o que implica em uma migração para o modo de recuperação reativo;

3. Presença de fluxos sensíveis ou críticos na rede, cujas características foram previamente definidas na programação do AFRC, para os quais devem ser criadas regras de recuperação proativa através dos grupos *fast failover* do OpenFlow;

4. Consumo excessivo de CPU no controlador, o que pode indicar uma iminente degradação do plano de controle da rede como um todo em virtude, dentre outras possibilidades, de um elevado fluxo agregado de mensagens com destino ao controlador, em especial durante processos de recuperação reativa de falhas, indicando a necessidade de adoção de um procedimento generalizado para aliviar a carga no controlador, aqui chamado de *Cold Water Bucket* (ver Subseção 3.5.5.

Como se pode notar pelas situações acima descritas, em especial nos casos (1) e (4), o AFRC busca também prevenir o mau funcionamento do controlador em função de uma sobrecarga de consumo originada por mensagens de PACKET-IN provenientes dos *switches*. Dessa forma, pode-se dizer que o AFRC contribui para a resiliência do plano de controle da rede, o que se faz necessário para que também se atinja a resiliência do plano de dados da SDN.

A sumarização de regras de encaminhamento OpenFlow tem duas consequências positivas quanto às questões de resiliência endereçadas pelo AFRC. A primeira, mais óbvia e já mencionada, relaciona-se à economia de espaço de memória nos *switches*. A segunda consequência relaciona-se ao fato de que sumarizar regras equivale a obter um número menor de regras mais genéricas, onde cada regra seja capaz de representar um subconjunto exclusivo com o maior número de regras possível. Em função disso, diminui-se a probabilidade de um *miss* na tabela de regras OpenFlow, ou seja, de um pacote ingressado no *switch* não possuir uma regra de encaminhamento associado a ele, o que geraria uma mensagem de PACKET-IN para o controlador. Ou seja, a sumarização de regras também tem como efeito a diminuição da taxa média de envio de mensagens PACKET-IN ao controlador.

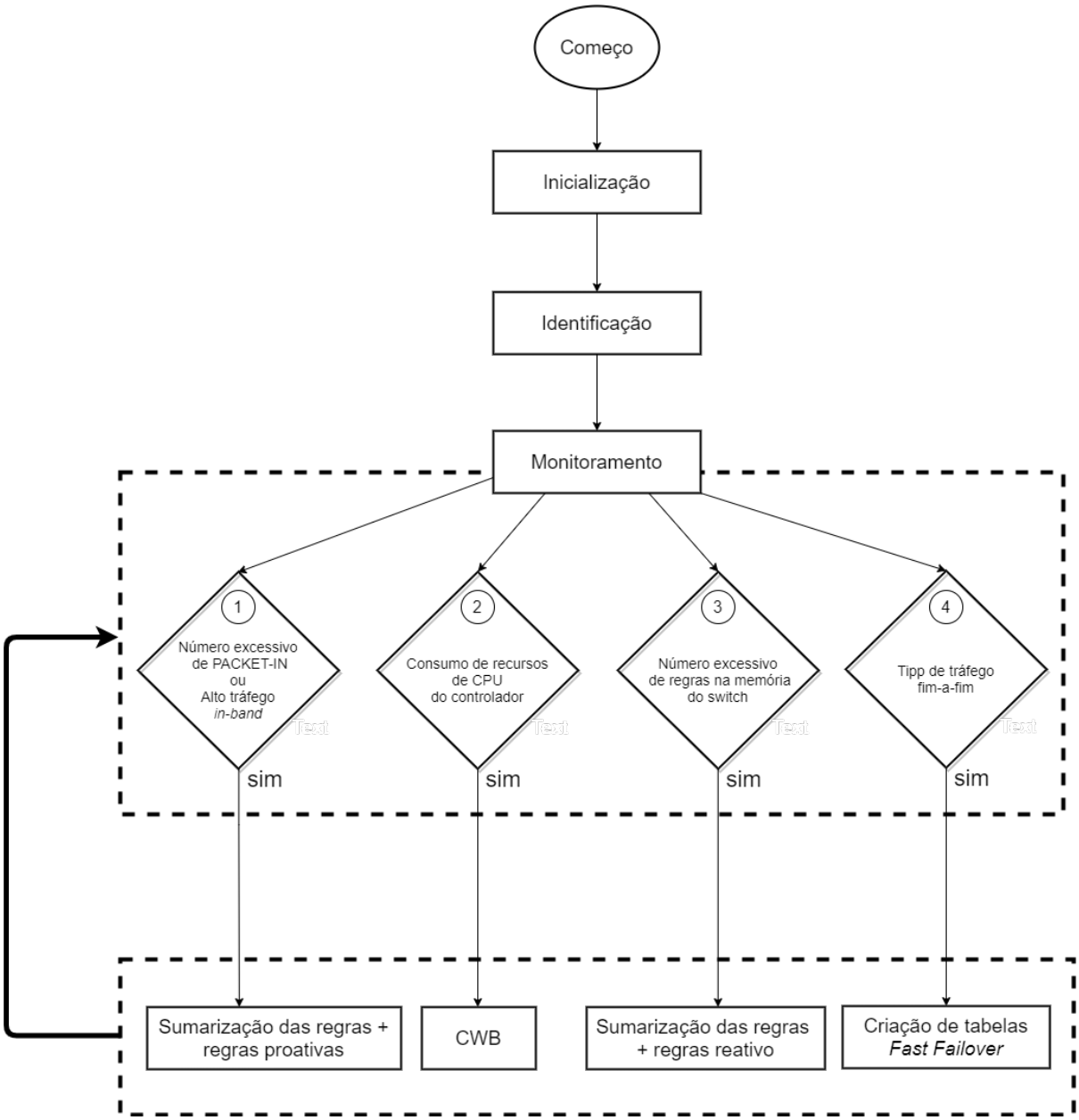


Figura 3.1: Fluxograma do AFRC.

3.3 Protótipo Implementado

O protótipo é totalmente escrito em *Shell Script* e o controlador utilizado neste trabalho foi o *Floodlight*.

Conforme pode ser notado pela Figura 3.1, o funcionamento do aplicativo se dá em 4 fases: inicialização, identificação, monitoramento e mitigação. Durante o processo de inicialização da rede, o AFRC será iniciado em segundo plano como uma aplicação Linux. A partir desse momento, utilizando ferramentas nativas, o

AFRC irá monitorar os *sockets* no servidor Linux onde o controlador SDN está sendo executado. Uma vez que foi identificado que há um controlador SDN em operação, inicia-se a fase de identificação, na qual serão coletadas informações dos *switches* que estão sendo gerenciados pelo controlador SDN.

Após a fase de identificação, começa a fase de monitoramento dos seguintes recursos da rede:

- utilização de CPU pelo controlador SDN;
- número de mensagen PACKET-IN que chegam ao controlador SDN (destinados à porta 6653);
- *throughput* da interface de rede que o controlador SDN usa para se comunicar com os *switches*, o que se torna importante quando o controlador se comunica no modo *in-band* com os *switches*;
- número de regras de fluxo instaladas em cada *switch* da rede;
- por fim, se há fluxo relacionado a alguma politica de "QoS de recuperação de falha" configurada no AFRC, para privilegiar determinados fluxos com o modo proativo de recuperação.

Portanto, a operação do AFRC se dá através de sondagens periódicas para coletar informações sobre o estado de cada item listado na fase de monitoramento. Caso um desses itens atinja valores que possam comprometer o funcionamento da rede, adentra-se na fase de mitigação. Nessa fase, conforme já mencionado, será realizada a mudança entre os modos de recuperação reativo e proativo em todos ou apenas nos *switches* cujas medições estejam ultrapassando os limites pré-definidos.

Este protótipo do AFRC não é o que se pode chamar de uma aplicação SDN, uma vez que ele não foi implementado como um *plugin* integrado ao controlador. Ele é executado como uma aplicação em segundo plano e não precisa necessariamente estar sendo executado na mesma maquina que roda o controlador SDN, porque na fase de inicialização o endereço IP do controlador é especificado. Isso permite, inclusive, que o AFRC rode na nuvem. O AFRC usa a *northbound* API do controlador, através da qual é possível aplicar as configurações nos *switches*. O AFRC instrui ao controlador quais novas regras devem ser escritas nos *switches*, que as aplica através do canal OpenFlow (interface *southbound* do controlador). A Figura 3.2 mostra uma topologia *fat-tree* empregando o AFRC onde um dos sistemas finais é um servidor

HTTP e FTP. No caso, os fluxos de resposta a requisições *web* ou de transferências de arquivo poderiam ser tratados de forma proativa durante uma recuperação de falha, conforme a situação número 3 ilustrada na Figura 3.1.

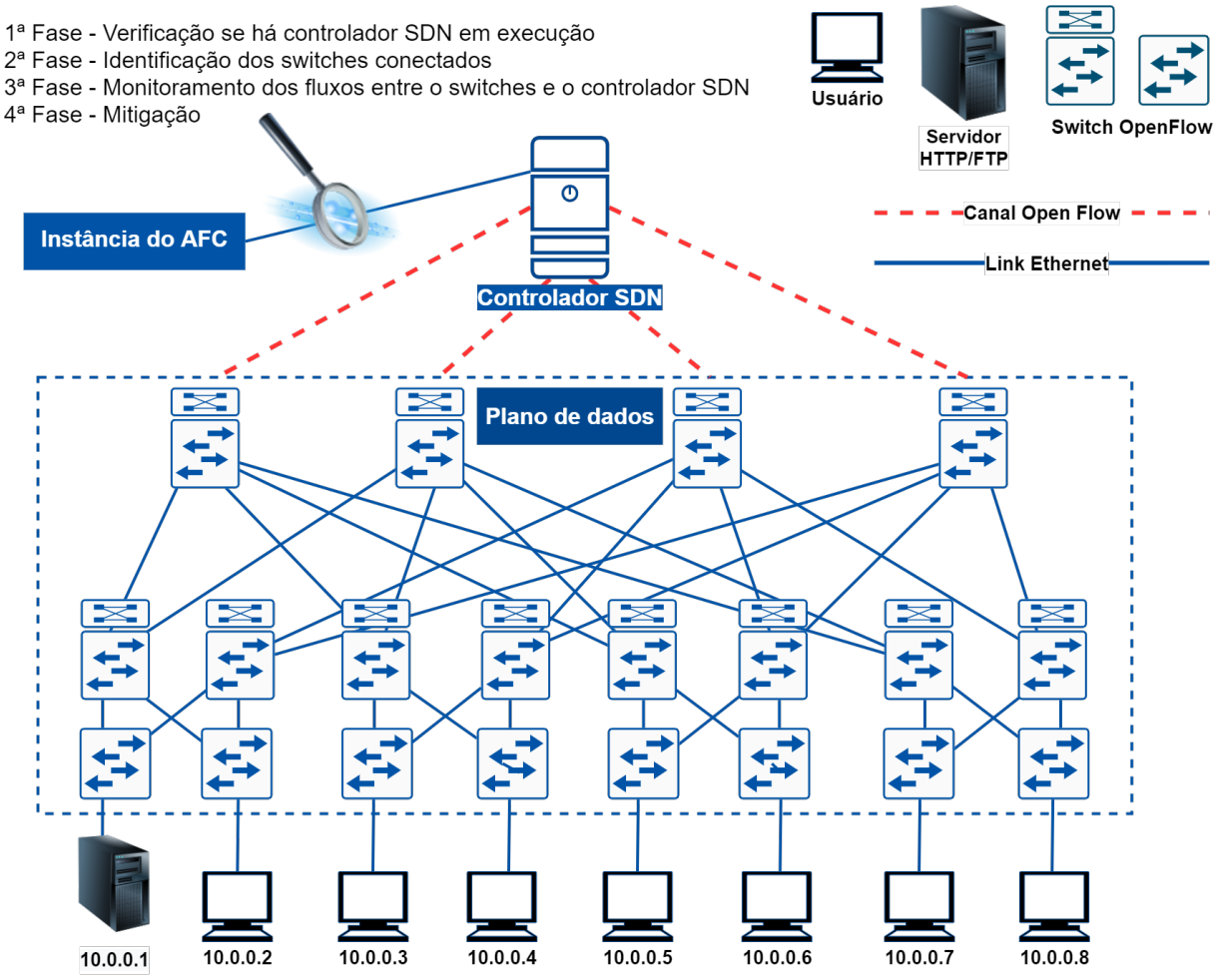


Figura 3.2: Topologia *fat-tree* com AFRC.

3.3.1 Inicialização

Esta fase consiste em apenas iniciar o AFRC, informando o IP do servidor onde o controlador SDN está sendo executado. A partir desse momento, o AFRC passa a monitorar as conexões TCP destinadas às portas 6633 e 6653, a fim de identificar se há um controlador instanciado. Após identificar que há um controlador, o AFRC prossegue para a fase de identificação. A Figura 3.3 mostra o comando para inicialização do AFRC.

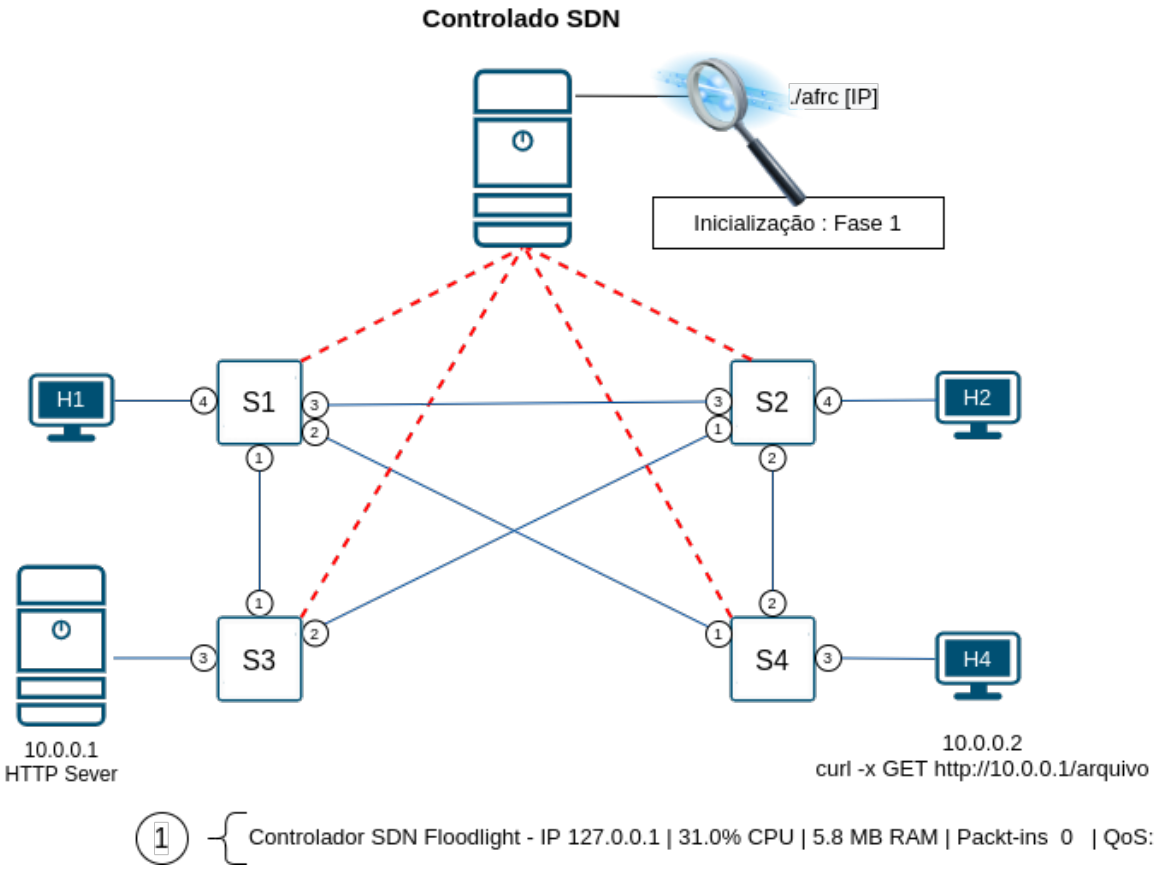


Figura 3.3: Inicialização do AFRC.

3.3.2 Identificação

Esta fase é usada pelo AFRC para identificar quais são os *switches* que estão se comunicando com o controlador SDN. Esta identificação é realizada através de uma sondagem periódica via *curl* no controlador SDN, onde são identificados os IDs dos *switches* a ele conectados no formato 'XX:XX:XX:XX:XX:XX:XX:XX XX'. A Figura 3.4 mostra os comandos que são usados para a coleta dessas informações, bem como o resultado dos dados coletados, referentes a todos os *switches* conectados ao controlador SDN.

3.4 Fase de Monitoramento

Conforme informado anteriormente, a fase de monitoramento ocorre após a fase de identificação e é subdivida em cinco processos de sondagens periódicas que são executados paralelamente. São eles:

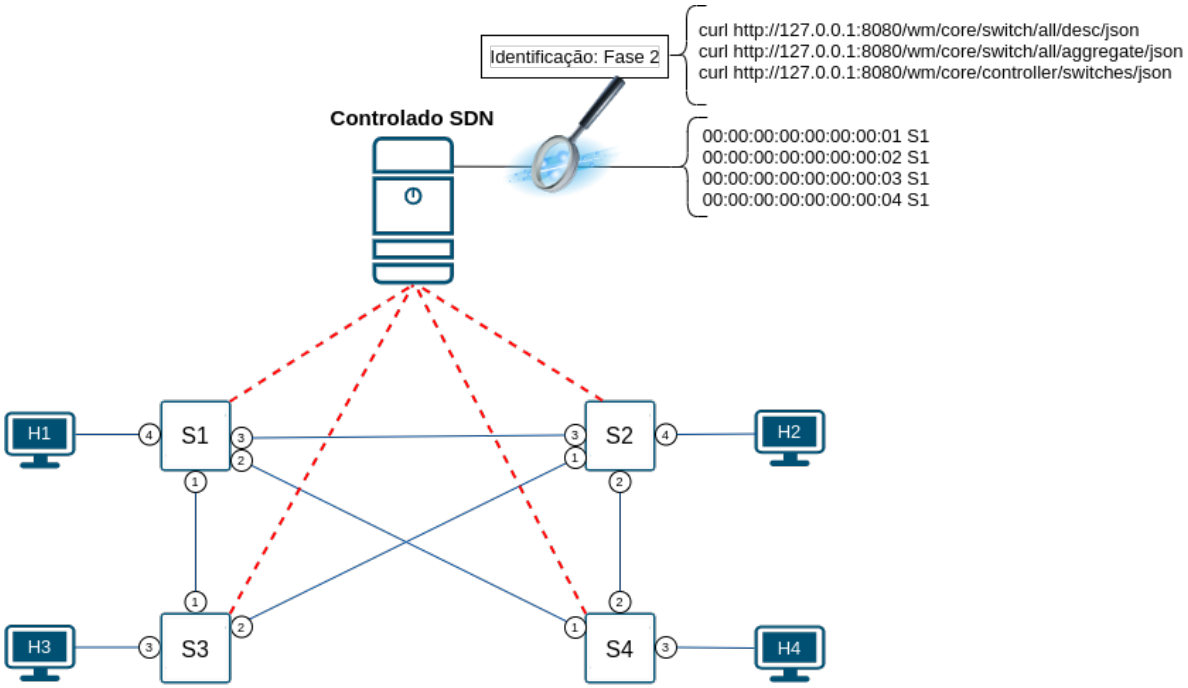


Figura 3.4: Identificação dos *swithes* pelo AFRC.

1. **Monitoramento do número de requisições destinadas ao controlador SDN via PACKET-IN**: Sempre que um fluxo chega ao *switch* e nele não há uma regra definida para este fluxo, o *switch* envia uma requisição ao controlador SDN via mensagem PACKET-IN. Essa quantidade de mensagens não deve ser superior a um valor limite pré-definido, dentro de um intervalo de tempo de observação também pré-definido, para que não ocorra um problema de sobrecarga do controlador [71]. A Figura 3.5 demonstra o processo de monitoramento do tráfego PACKET-IN destinado ao controlador SDN, onde são realizadas leituras, a cada intervalo de tempo de observação, de um contador que registra o número de mensagens PACKET-IN que chegam ao controlador. Esse contador pode ser obtido através de filtros instalados na ferramenta *tshark*, por exemplo. A diferença entre duas leituras subsequentes desse contador determina o número de mensagens PACKET-IN que chegou no respectivo intervalo de tempo de observação. Tanto o intervalo de tempo de observação quanto o valor limite para o número de mensagens PACKET-IN nesse intervalos de tempos são parâmetros definidos pelo administrador da rede.

2. **Monitoramento do *throughput* da interface de rede do controlador SDN para comunicação *in-band* com os *switches*:** Caso o controlador SDN se comunique com os *switches* em modo *in-band*, o congestionamento

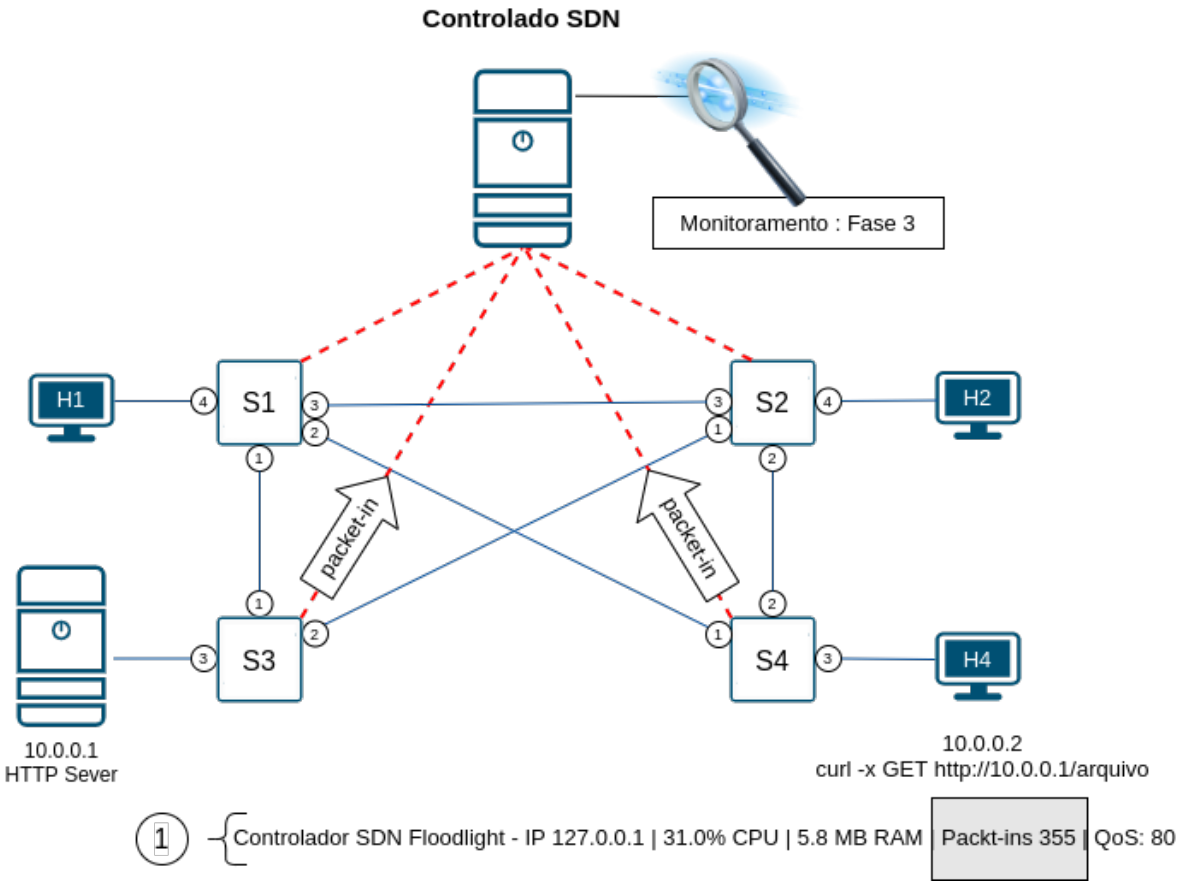


Figura 3.5: Monitoramento do tráfego de mensagens PACKET-IN para o controlador.

dessa respectiva interface de rede afetará o gerenciamento da SDN pelo controlador, ou seja, requisições referentes a mensagens PACKET-IN poderão se perder por descarte pacotes, assim como as respostas do controlador a essas requisições. Em função disso, o AFRC adotará, neste caso específico de canal de controle *in-band*, as mesmas ações decorrentes de um número excessivo de mensagens de PACKET-IN ao controlador. Esse *throughput* é medido da mesma forma que descrita acima para o tráfego de mensagens PACKET-IN destinadas ao controlador, sendo que para tal são usados os contadores disponibilizados pelo próprio sistema operacional Linux.

3. **Monitoramento do número de regras em cada *switch* da topologia:** Podem haver muitas regras nos *switches* e isso pode comprometer seu funcionamento caso isso acarrete o esgotamento de seu espaço de memória. Nesse caso, o número de regras precisará ser diminuído através de uma migração para o modo de recuperação reativo. A Figura 3.6 demonstra o funcionamento desse processo de monitoramento, que faz uso do *curl* para extrair essas informações

do próprio controlador SDN.

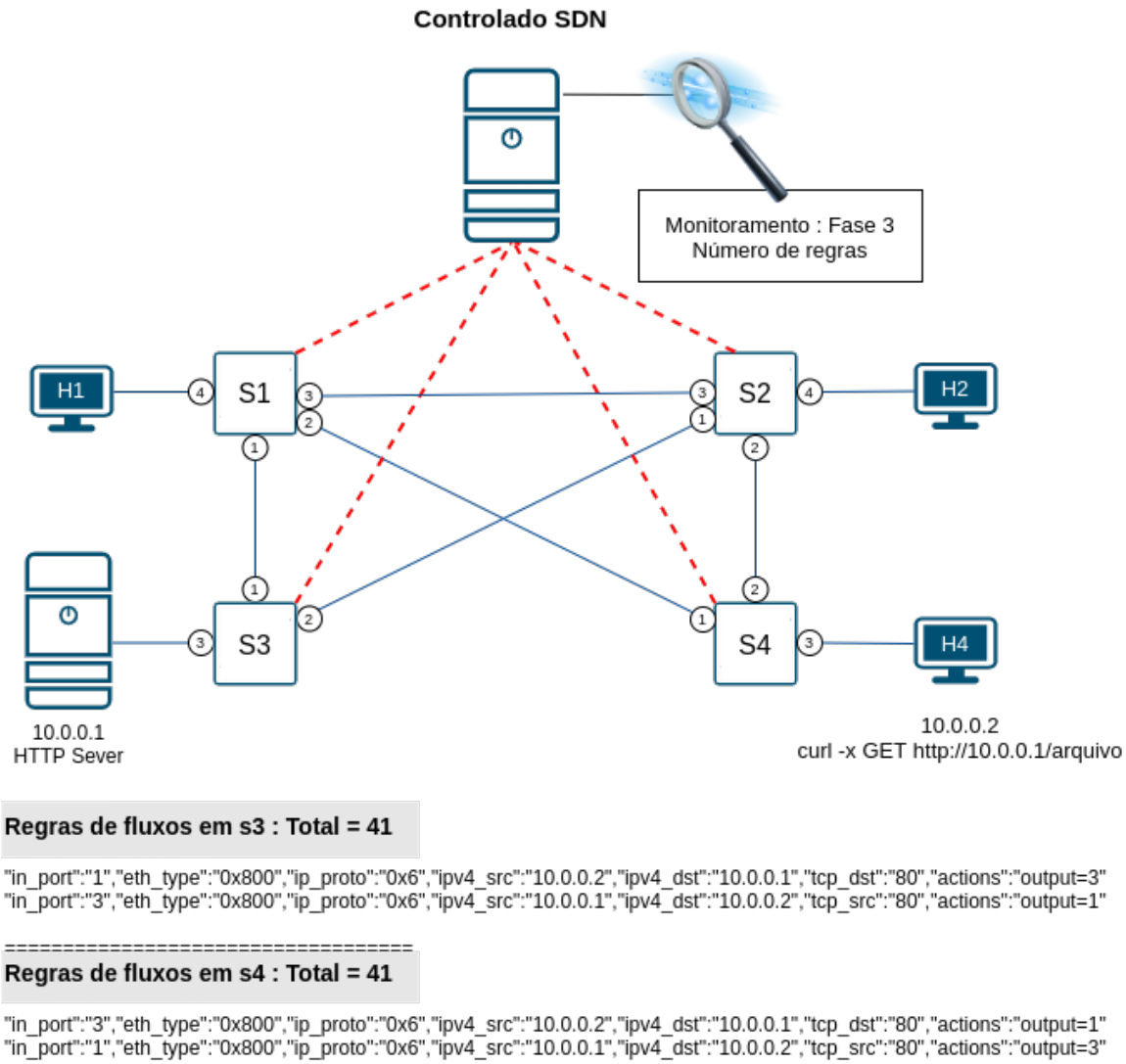


Figura 3.6: Monitoramento do número de regras nos *switches*

4. **Monitoramento do tipo de tráfego entre os sistemas fim-a-fim:** Através desse monitoramento é possível verificar se há algum tipo de tráfego que deve ser tratado de forma diferente, conforme previamente configurado no AFRC. Havendo esse tipo de tráfego, o AFRC instalará regras de *backup* para esses fluxos nos *switches* por onde esses fluxos passam, através de tabelas *fast failover*. Assim, caso haja alguma falha de enlace, esses fluxos estarão protegidos e os tráfegos fim-a-fim não serão comprometidos devido a um tempo excessivo para recuperação da falha. A Figura 3.7 demonstra o funcionamento deste processo para fluxos relacionados à porta 80, onde se pode verificar que:

   - em (1) aguarda-se identificar algum tráfego destinado ao controlador SDN

requisitando informações sobre fluxos originados ou destinados à porta 80;

- em (2) o processo de monitoramento identifica que o controlador está recebendo requisições sobre qual ação tomar para fluxos com porta de origem ou destino 80 e, com isso, na fase de mitigação, tabelas *fast failover* serão instaladas na memória dos *switches* que estão sendo utilizados por esses fluxos.

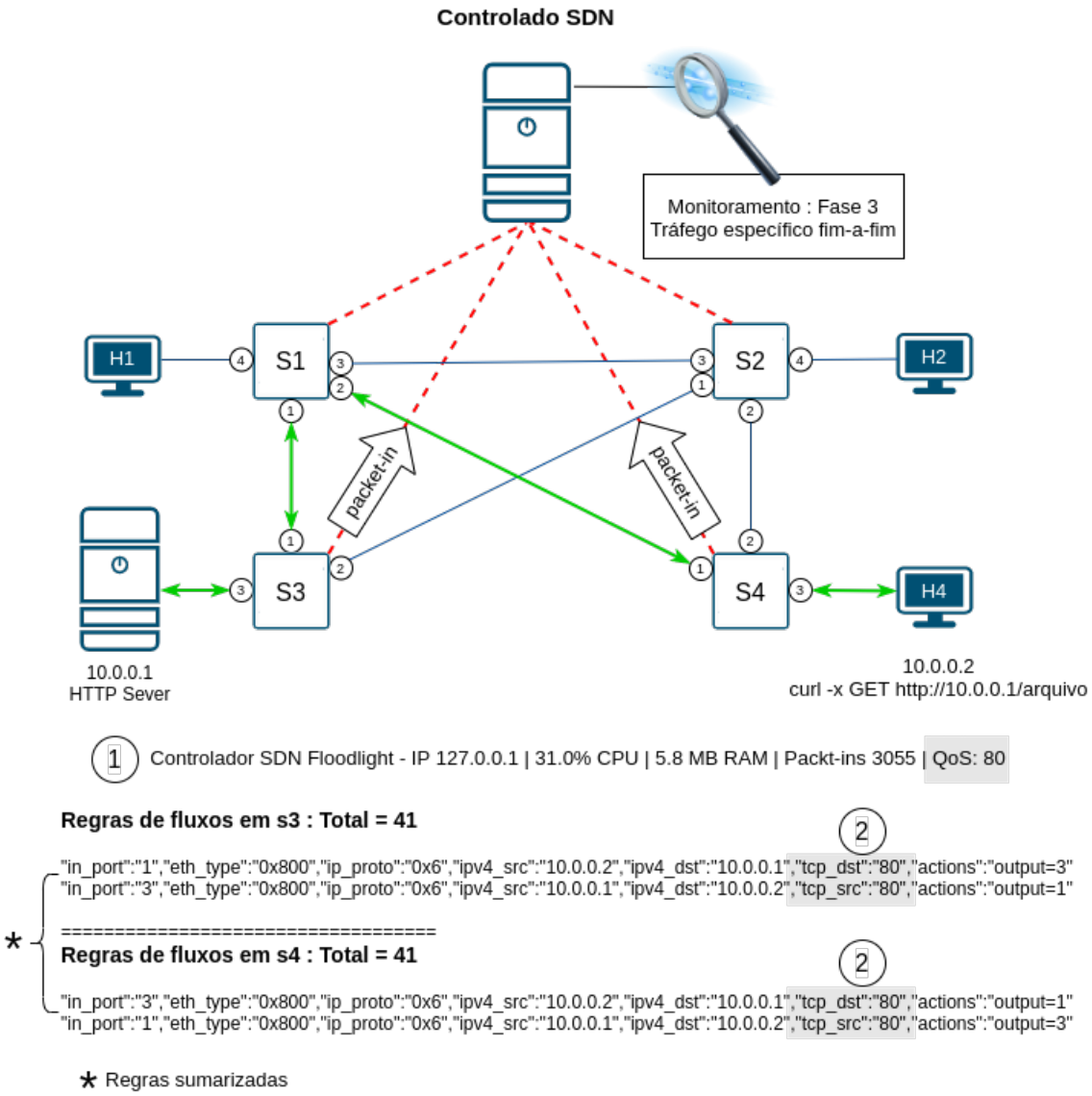


Figura 3.7: Monitoramento do tipo de tráfego fim-a-fim na rede.

5. **Monitoramento do consumo de CPU gerado pelo controlador SDN:** Sempre que uma mensagem chega ao controlador SDN, ele precisa processar essa mensagem. Geralmente, em consequência desse processamento é criada

uma nova mensagem de resposta com instruções ao *switch*. Tudo isso onera ainda mais o uso de CPU pelo controlador e esse uso é monitorado, no caso do AFRC, através do comando de linha *ps* do Linux, aplicado via *shell script*. Uma vez que seja identificado que o controlador SDN está gerando um elevado consumo de CPU, conforme apresentado na Figura 3.8, um processo conservador de redução de demandas ao controlador, aqui chamado de CWB, será disparado pelo AFRC.

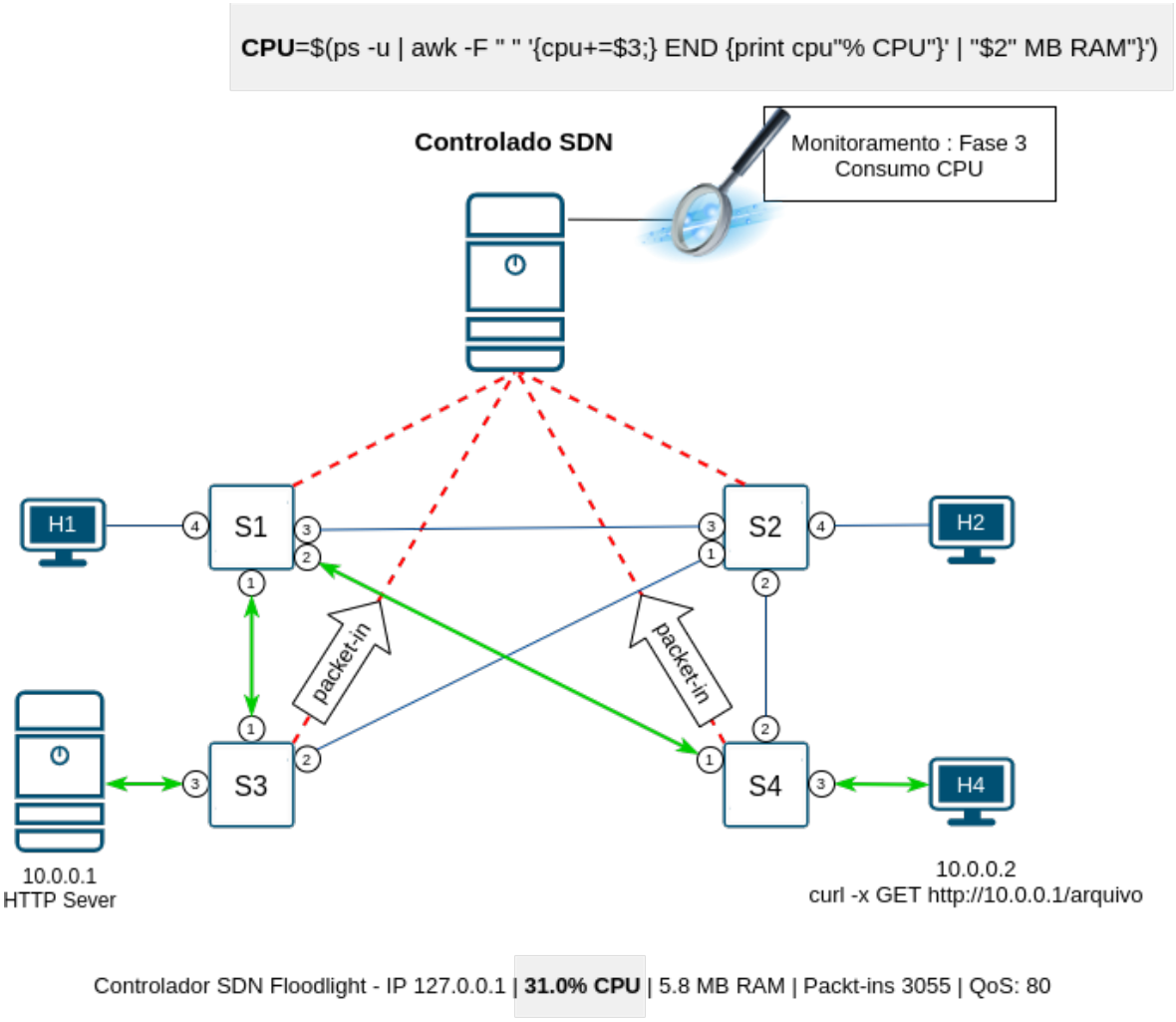


Figura 3.8: Monitoramento do consumo de CPU do controlador.

3.5 Fase de Mitigação

Após o AFRC detectar, na fase de monitoramento, uma das cinco situações listadas acima, inicia-se a fase de mitigação. Este processo pode ser diferente em decorrência de cada item da fase de monitoramento, conforme a seguir:

1. Itens 1 e 2: será realizada uma migração para o modo de recuperação proativo nas regras de fluxo dos *switches* que estão gerando um número excessivo de mensagens de PACKET-IN para o controlador, com a sumarização de regras de fluxos conforme apresentado na subseção 3.5.1. Para descobrir quais *switches* estão originando um número excessivo de mensagens de PACKET-IN para o controlador, o que, num canal de controle *in-band*, pode também estar congestionando o esse canal, faz-se necessário monitorar o tráfego destinado a porta do controlador SDN, neste caso TCP 6653.

   Item 3: será realizada uma sumarização das regras de fluxo e subsequente migração para o modo de recuperação proativo nos *switches* com um número excessivo de regras de fluxo instaladas.

2. Item 4: Para os *switches* que possuem regras de fluxo que se enquadram nesta categoria, tabelas *fast failover* serão criadas para proteger esses fluxos em caso de falha de enlace.

3. Item 5: Todos os *switches* serão submetidos ao procedimento denominado de *Cold Water Bucket*, ou CWB e, nesse caso, todas as regras de fluxo serão protegidas em modo proativo, e regras de backup através de tabelas *Fast Failover*.

3.5.1 Sumarização de regras de encaminhamento OpenFlow

Vários trabalhos tentam resolver o desafio de reduzir o espaço consumido pelo número de regras OpenFLow nas tabelas de fluxo, investigando várias configurações e características de possíveis implementações em *hardware* para as tabelas de fluxo (por exemplo, com relação aos tipos de memória que podem ser usados). No caso do AFRC, conforme mencionado anteriormente, essa redução será buscada através da sumarização das regras de fluxo instaladas na memória dos *switches*, usando os campos com maior número de correspondências para sumarizar as regras ao máximo, porém sem perder a semântica de encaminhamento.

Conforme mencionado anteriormente, sumarizar um conjunto de regras significa buscar um conjunto menor de regras que consiga representar inequivocamente o conjunto original. Um exemplo simples ocorre no caso onde todas as regras de encaminhamento de uma tabela apontam para o mesmo próximo salto. Nesse caso, todas elas poderiam ser substituídas por uma única regra indicando uma única saída para os pacotes e um único MAC de destino a ser inserido nos quadros Ethernet de saída - o que frequentemente é chamado de " rota *default*" ou "*gateway* padrão".

Em roteamento (camada três), outro caso muito comum é quando há múltiplas rotas para sub-redes de destino apontando para um mesmo *next-hop*, sendo que essas sub-redes fazem parte de uma faixa IP mais abrangente para a qual não existe nenhum "sub-bloco" com rota apontando para outro *next-hop*. Portanto, nesse caso todas as rotas para essas sub-redes podem ser substituídas - ou seja, sumarizadas - por uma única rota para a faixa de IP mais abrangente.

No caso de SDN, o processo de sumarização deve contemplar a correspondência dos diversos campos das regras de fluxo, além das ações a serem executadas com os pacotes ingressantes. Em outras palavras, o processo de sumarização deverá verificar quais regras estão relacionadas a uma mesma ação e buscar a quantidade mínima de agrupamentos que podem ser criados considerando todos os campos usados pelas regras a serem agrupadas, de maneira que nenhuma regra original desses novos agrupamentos seja violada. Para tal, o procedimento de sumarização adotado no AFRC percorre uma sequência de passos, mostrados a seguir.

O primeiro passo é coletar as regras de fluxo instaladas na memória dos *switches* via controlador SDN. Para realizar esta ação, será usado o **curl** (ou cURL)[1]. O curl foi projetado para funcionar sem interação do usuário e é uma ferramenta de linha de comando utilizada para obter ou enviar dados, incluindo arquivos, usando a sintaxe *URL*. A sintaxe padrão do curl é *"curl http://localhost/"*. Por padrão, uma requisição simples no cURL é feita implicitamente através do método *GET*, mas é possível alterar o método HTTP da requisição através da opção *-X*. Há vários métodos suportados pelo HTTP, porém os mais comumente usados são *GET, POST, PUT e DELETE*. Neste projeto fazemos uso da sintaxe **"curl http://127.0.0.1:8080/wm/core/switch/00:00:00:00:00:00:00:01/flow/json"**, para coletar todas as regras de fluxo instaladas nos *switches* conectados ao controlador SDN.

O próximo passo é tratar as informações de tabulação e marcação dos campos que compõem as regras de fluxo OpenFlow, inseridas pelo controlador SDN na memória dos *switches* que estão conectados a ele. Esses *switches* estão sendo usados para encaminhar pacotes e foram identificados pelo AFRC que precisam ter as suas regras manipuladas, pois fazem fazem parte do grupo de *switches* que dão *"match"* com alguma das situações apresentadas no fluxograma 3.1. Esta etapa é realizada pelo comando **sed**, que é uma ferramenta usada para analisar e transformar textos. Por exemplo, o comando *sed 's/,/ /g'* substitui o caracter **",**" por um espaço em

[1]https://curl.haxx.se/

branco.

Após o processo de organização das informações pelo comando *sed*, na etapa anterior, será realizada a seleção dos campos das regras de fluxos que serão usadas na etapa de sumarização. Esta etapa será executada pelo comando **awk**. O AWK é definido como uma linguagem de programação de correspondência de padrões (expressões regulares). Um exemplo da utilização do AWK é o comando **awk '/in_port/ {print $12,$13$14$15,$16,$17,$18,$19,$20$21$22}'**, cuja sintaxe determina a seleção das linhas que contêm campos da regra de fluxo OpenFlow com a informação **"in_port"**, ou seja, linhas que correspondem a uma regra de fluxo que usa como parâmetro a *interface* pela qual um pacote está ingressando no *pipeline* de processamento do *switch* OpenFlow. Poderíamos usar qualquer campo da regra de fluxo OpenFlow, porém esta é a primeira informação útil para o o processo de sumarização encontrada após os campos de controle do controlador SDN, que também fazem parte das regras de fluxo.

A próxima etapa para realizar a sumarização envolve filtrar e excluir linhas extras que contêm todos os campos da regra de fluxo OpenFlow em comum. Nesta etapa também será usado o AWK, e para esta tarefa usaremos a *sintaxe* **awk '!x[$0]++'**. A matriz **[$0]** contém todo as linhas geradas após o processo de seleção de campos realizado anteriormente pelo AWK. A *sintaxe* **x[$0]** significa "procure as linhas armazenadas em **$0** na matriz associativa chamada de **"x"**". Se essa matriz ainda não existir, ela será criada. Deseja-se imprimir a linha somente quando esta for a primeira vez que ela é encontrada e, para isso, usa-se o parâmetro **"!"**. Portanto, para cada linha, incrementa-se um nó da matriz e imprimi-se a linha se o conteúdo desse nó não foi encontrado anteriormente. Além disso, incrementa-se a entrada para indicar mudança de linha através do operador **"++"**.

Vale destacar que está fora do escopo deste trabalho a proposta de um método generalizado para sumarização de regras OpenFlow, uma vez que se trata de uma área mais abrangente de pesquisa [72]. Essa área de pesquisa costuma abranger também a redução do número de regras em *firewalls* [73], [74].

Para exemplificar o processo de sumarização das regras de fluxo, será utilizada a topologia da Figura 3.9.

Inicialmente, vê-se nas regras de fluxo OpenFlow exibidas no código 3.1 que todos os *switches* estão com a regra padrão usada em SDN quando não há pacotes a serem processados, e informa que, para qualquer pacote ingressante, esse deverá ser

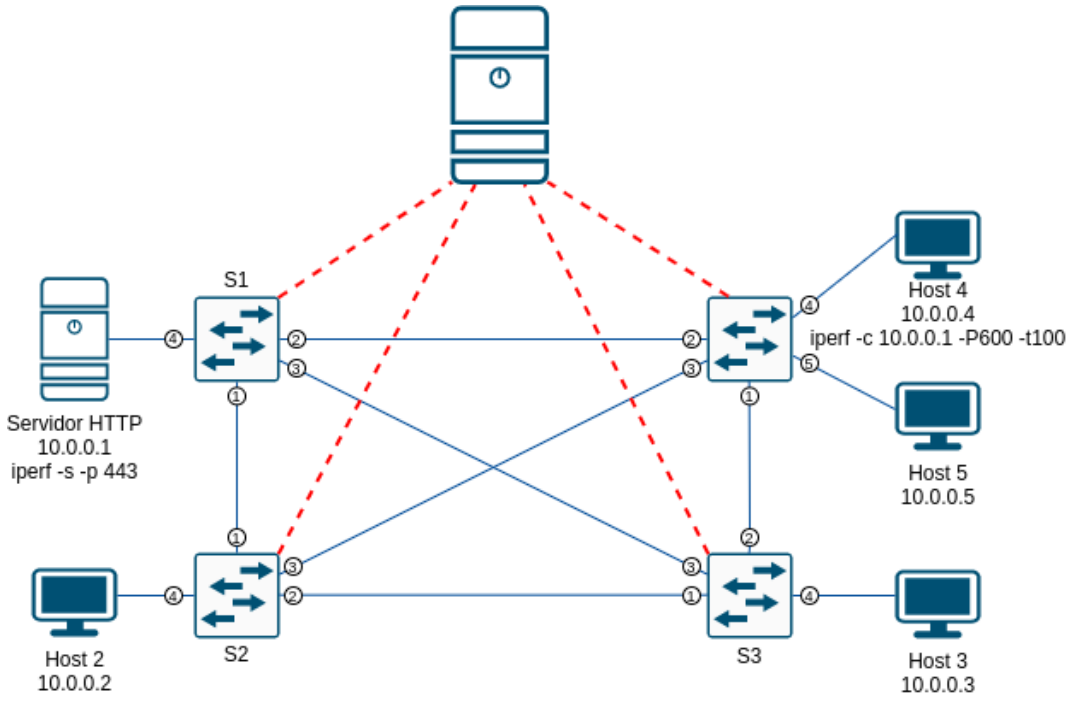


Figura 3.9: Topologia de exemplo para a sumarização de regras OpenFlow.

encaminhado diretamente ao controlador. Essa regra pressupõe que o controlador tomará uma decisão de encaminhamento com base nos diversos campos contemplados pelo OpenFlow e escreverá essa regra no respectivo *switch*, instruindo sobre o que fazer com novos pacotes que façam parte desse mesmo fluxo.

Código 3.1: Regra de fluxos dos *switches* da rede do exemplo.


```
"00:00:00:00:00:00:00:04":{flows:[{version:"OF_14",cookie:"0",table_id:"0
    x0",packet_count:"29",byte_count:"2183",duration_sec:"146",
    duration_nsec:"571000000",priority:"0",idle_timeout_s:"0",
    hard_timeout_s:"0",flags:[],match:{},instructions:{
    instruction_apply_actions:{actions:"output=controller"}}}]},
"00:00:00:00:00:00:00:02":{flows:[{version:"OF_14",cookie:"0",table_id:"0
    x0",packet_count:"27",byte_count:"2033",duration_sec:"146",
    duration_nsec:"575000000",priority:"0",idle_timeout_s:"0",
    hard_timeout_s:"0",flags:[],match:{},instructions:{
    instruction_apply_actions:{actions:"output=controller"}}}]},
"00:00:00:00:00:00:00:03":{flows:[{version:"OF_14",cookie:"0",table_id:"0
    x0",packet_count:"29",byte_count:"2191",duration_sec:"146",
    duration_nsec:"553000000",priority:"0",idle_timeout_s:"0",
    hard_timeout_s:"0",flags:[],match:{},instructions:{
    instruction_apply_actions:{actions:"output=controller"}}}]},
"00:00:00:00:00:00:00:01":{flows:[{version:"OF_14",cookie:"0",table_id:"0
```

```
x0",packet_count:"26",byte_count:"1950",duration_sec:"146",
duration_nsec:"573000000",priority:"0",idle_timeout_s:"0",
hard_timeout_s:"0",flags:[],match:{},instructions:{
instruction_apply_actions:{actions:"output=controller"}}}]}}
```


Para gerar fluxos de pacotes na rede da Figura 3.10, vamos utilizar o Iperf [2] para simular conexões HTTPS [3] simultâneas. Usamos a sintaxe *iperf -s -p 443*, no lado do servidor, e, no lado do cliente, *iperf -c 10.0.0.1 -p 443 -P50 -t100*. Todos os fluxos foram gerados por conexões iniciadas no *host 4* em direção ao servidor conectado pelo *switch S1*. As regras de fluxo geradas são apresentadas no código 3.2, instaladas nos *switches* após os fluxos HTTPS serem gerados. O fato de se ter um total de cinco regras instaladas apenas nos *switches* S1 e S4 é porque o controlador determinou que o caminho usando o enlace entre esses *switches* é o menor dentre os possíveis (portanto, o melhor), conforme representado na Figura 3.11.

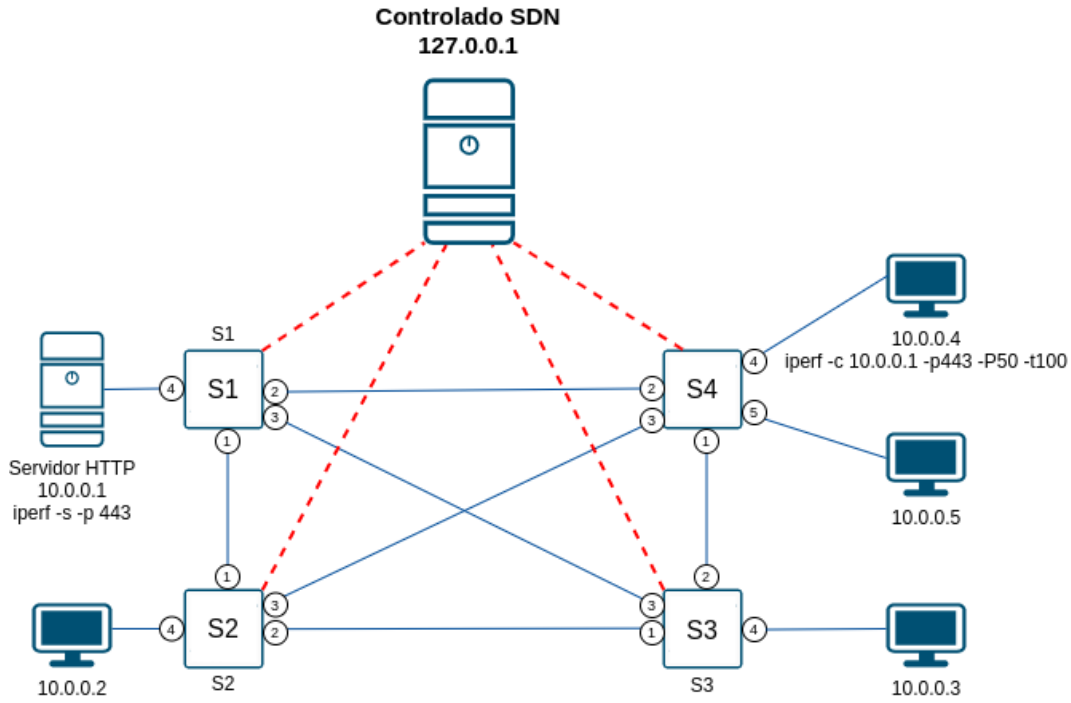


Figura 3.10: Topologia do exemplo com destaque para o caminho do tráfego HTTPs.

Código 3.2: Regras de fluxo instaladas pelo controlador SDN em S1.


```
{flows:[{version:"OF_14",cookie:"9017209766543360",table_id:"0x0",
    packet_count:"450535",byte_count:"29739078",duration_sec:"8",
    duration_nsec:"66000000",priority:"1",idle_timeout_s:"5",
```


[2] https://iperf.fr/
[3] https://pt.wikipedia.org/wiki/Hyper_Text_Transfer_Protocol_Secure

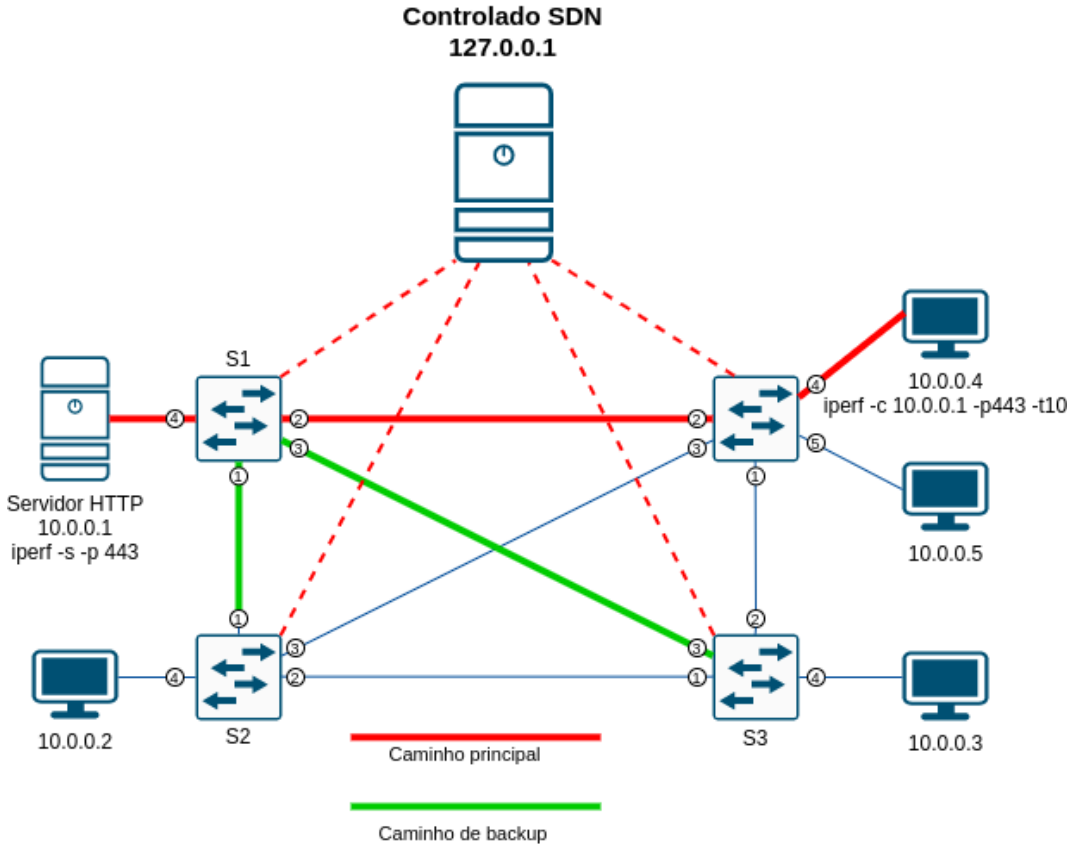


Figura 3.11: Caminho determinado pelo controlador *floodlight*

.


```
    hard_timeout_s:"0",flags:[],match:{in_port:"4",eth_dst:"ba:ea:3a:0b:6b
    :bb",eth_src:"8a:60:f5:56:2f:9d",eth_type:"0x800",ip_proto:"0x6",
    ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.4",tcp_src:"443",tcp_dst:"45066",
    ovs_tcp_flags:"0x0010"},instructions:{instruction_apply_actions:{
    actions:"output=2"}}},
{version:"OF_14",cookie:"9017210521518080",table_id:"0x0",packet_count:"
    586601",byte_count:"35977245234",duration_sec:"8",duration_nsec:"
    55000000",priority:"1",idle_timeout_s:"5",hard_timeout_s:"0",flags:[],
    match:{in_port:"2",eth_dst:"8a:60:f5:56:2f:9d",eth_src:"ba:ea:3a:0b:6b
    :bb",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.4",ipv4_dst:"
    10.0.0.1",tcp_src:"45066",tcp_dst:"443",ovs_tcp_flags:"0x0010"},
    instructions:{instruction_apply_actions:{actions:"output=4"}}},
{version:"OF_14",cookie:"9017210638958592",table_id:"0x0",packet_count:"
    148418",byte_count:"225412676",duration_sec:"8",duration_nsec:"
    54000000",priority:"1",idle_timeout_s:"5",hard_timeout_s:"0",flags:[],
    match:{in_port:"2",eth_dst:"8a:60:f5:56:2f:9d",eth_src:"ba:ea:3a:0b:6b
    :bb",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.4",ipv4_dst:"
    10.0.0.1",tcp_src:"45066",tcp_dst:"443",ovs_tcp_flags:"0x0018"},
```

```
    instructions:{instruction_apply_actions:{actions:"output=4"}}},
```

Para o processo de sumarização deste exemplo, foram considerados apenas alguns campos das regras de fluxos instaladas nos *switches* ”S1”e ”S4”, conforme já mencionamos anteriormente. A Regras de Fluxo 3.3 apresenta o resultantes da sumarização realizada pelo AFRC para este exemplo em todos os *switches*.

Código 3.3: Sumarização das regras de fluxo da rede exemplo.

```
Fluxos s1 : 4 Fluxos | 2 Sumarizadas
in_port:"2",eth_dst:"5e:6a:91:ad:35:22",eth_src:"62:c8:ce:15:85:a0",
    eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.4",ipv4_dst:"10.0.0.1
    ",tcp_dst:"443",actions:"output=4"
in_port:"4",eth_dst:"62:c8:ce:15:85:a0",eth_src:"5e:6a:91:ad:35:22",
    eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.4
    ",tcp_src:"443",actions:"output=2"
==================================
Fluxos s2 : 1 Fluxos - Regra de fluxos padrão
"00:00:00:00:00:00:00:02":{flows:[{version:"OF_14",cookie:"0",table_id:"0
    x0",packet_count:"27",byte_count:"2033",duration_sec:"146",
    duration_nsec:"575000000",priority:"0",idle_timeout_s:"0",
    hard_timeout_s:"0",flags:[],match:{},instructions:{
    instruction_apply_actions:{actions:"output=controller"}}}]},
==================================
Fluxos s3 : 1 Fluxos - Regra de Fluxo padrão
"00:00:00:00:00:00:00:03":{flows:[{version:"OF_14",cookie:"0",table_id:"0
    x0",packet_count:"29",byte_count:"2191",duration_sec:"146",
    duration_nsec:"553000000",priority:"0",idle_timeout_s:"0",
    hard_timeout_s:"0",flags:[],match:{},instructions:{
    instruction_apply_actions:{actions:"output=controller"}}}]},
==================================
Fluxos s4 : 4 Fluxos | 2 Sumarizadas
in_port:"4",eth_dst:"5e:6a:91:ad:35:22",eth_src:"62:c8:ce:15:85:a0",
    eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.4",ipv4_dst:"10.0.0.1
    ",tcp_dst:"443",actions:"output=2"
in_port:"2",eth_dst:"62:c8:ce:15:85:a0",eth_src:"5e:6a:91:ad:35:22",
    eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.4
    ",tcp_src:"443",actions:"output=4"
```

3.5.2 Criação de grupos de tabela *fast failover*

O grupo do tipo *Fast Failover* foi criado a partir do OpenFlow 1.1 como uma opção de gerar resiliência para o plano de dados. O processo de criação de tabelas de grupo é muito parecido com o de criar regras de fluxo. É requerido o campo ***"entry_type"*** para informar ao *switch* que a regra que está sendo instalada representa um grupo, e não um fluxo, pois caso isso não seja feito, será interpretado pelo *switch* como uma regra de fluxo padrão. Embora o OpenFlow não defina identificadores de *buckets*, uma entrada de grupo deve ter um **ID** único por *bucket* para definir a ordem dos *buckets*. Os *buckets* são ordenados sequencialmente do ID mais baixo para o ID mais alto. Um *bucket* é uma lista de ações separadas que, neste caso, fazem parte de uma tabela de grupo do tipo *Fast Failover*, conforme apresentado na Figura 3.12.

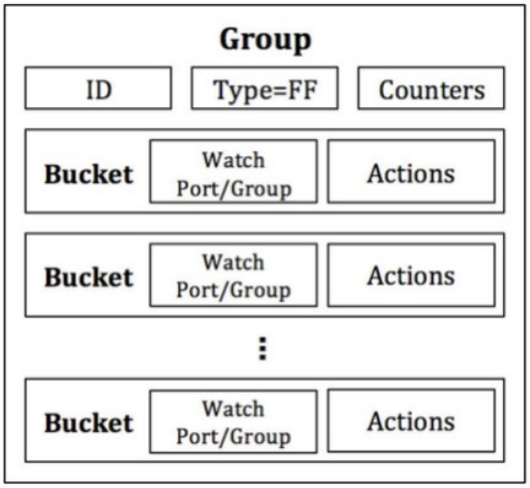


Figura 3.12: Exemplo de grupo do tipo *Fast Failover*.
Fonte: Izard,2018

Para que uma tabela de grupo seja criada em *switches* OpenFlow, é necessário informar as propriedades deste grupo, conforme apresentado na Seção 2.3.7. A linha de comando mostrada no código 3.4 instrui o controlador Floodlight a criar um grupo do tipo *Fast Failover*, que monitora através dos *buckets* "1" e "2" as interfaces "1" e "2", respectivamente. Como os *buckets* são processados conforme a ordem crescente do *bucket_id*, o processamento dos pacotes que são direcionados para esse grupo, conforme especificado em uma regra de fluxo instalada no *switch*, será da seguinte forma:

1. Os pacotes do respectivo fluxo serão encaminhados inicialmente para o *bucket* "1", que tem como ação encaminhar o pacote para a interface de saída "1",

enquanto durar o tráfego.

2. Caso durante a ocorrência deste tráfego a interface "1" for detectada pelo *switch* como inativa, os pacotes serão encaminhados automaticamente para o *bucket* "2", que tem como ação encaminhar os pacotes para a interface "2".

3. Caso a interface "2" fique inativa também, o *switch* terá que realizar o procedimento *default*, que geralmente é o de encaminhar o pacote para o controlador, que por sua vez poderá instruir o *switch* com novas regras relacionadas a este fluxo.

Além da criação do grupo *Fast Failover*, é necessário também criar regras de fluxo para encaminhar os pacotes para o grupo que será usado. Na regra de fluxo mostrada no código 3.4 verifica-se como um grupo do tipo *Fast Failover* é criado em *switches* OpenFlow a partir do controlador Floodlight. O processo de utilização de grupos OpenFlow respeita uma ordem lógica, pois as regras são aplicadas *on the fly*. Ou seja, primeiro cria-se o grupo OpenFlow para depois se criar as regras de fluxo de encaminhamento dos pacotes para este grupo. Caso essa ordem seja invertida, os pacotes entrantes poderão ser descartados por não haver um grupo válido para o seu devido processamento.

Código 3.4: Comandos para criar grupos *fast failover* no *Floodlight*.

```
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"FF1",entry_type:"
    group",active:"true",group_id:"1",group_type:"fast_failover","
    group_buckets": [{bucket_id:"1",bucket_watch_port:"2",
    bucket_watch_port:"output=2"},{bucket_id:"2",bucket_watch_port:"3",
    bucket_watch_port:"output=3"}]}' http://127.0.0.1:8080/wm/
    staticentrypusher/json
```

É possível verificar através da Figura 3.13 como fica o cenário de proteção do plano de dados usando o grupo *Fast Failover*, caso ocorra algum problema entre o enlace de S1 a S4, usando as regras dos códigos 3.4 e 3.5.

Código 3.5: Comandos para encaminhar os pacotes para o grupo "1" em S1.

```
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-1",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
    :"1",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-2",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
```

```
    :"2",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-3",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
    :"3",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-4",
    eth_type:"0x0800",priority:"12",idle_timeout:"5",active:"true",in_port
    :"4",ipv4_dst:"10.0.0.4",actions:"group=1"}' http://127.0.0.1:8080/wm/
    staticentrypusher/json
```

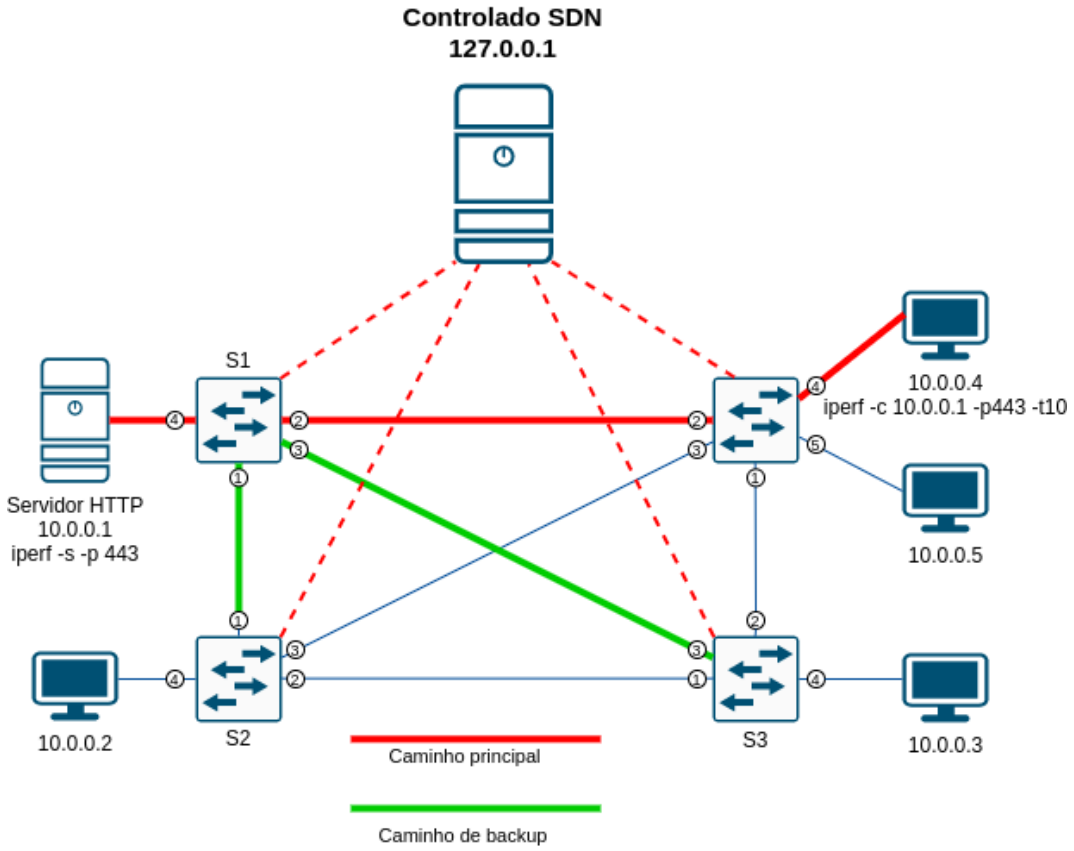


Figura 3.13: *C*enário após a criação das regras e tabelas

3.5.3 Migração para o modo de recuperação proativo

No modelo proativo, o controlador instalar proativamente nos *switches* as regras necessárias para o encaminhamento dos pacotes, evitando que os *switches* façam consultas frequentes ao controlador SDN durante a entrada de novos pacotes, para os quais não haveria regras instaladas. Como consequência, a forma proativa elimina qualquer latência induzida pela consulta a um controlador em cada "novo" fluxo.

Porém, dois pontos importantes devem ser considerados: primeiro, as regras de fluxo devem ser sumarizadas ao ponto de atender o maior número de fluxos possíveis, e segundo, o temporizador *idle timeout* deverá ser desativado ou configurado com um valor muito alto, para que a regra não expire prematuramente. Para mitigar esses dois problemas, o AFRC fará do uso do *idle timeout* padrão adotado pelo *Floodlight* (ver Seção 2.3.3) em conjunto com o grupo *Fast Failover* para que, em caso de falha de enlaces, os pacotes não sejam descartados e não sejam geradas novas consultas ao controlador. Além disso, a sumarização de regras também será usada neste processo, uma vez que as regras resultantes consomem menos espaço na memória dos *swicthes*.

No código 3.6 são apresentadas as linhas de comando utilizadas pelo AFRC para inserir as regras que mudarão o modo de funcionamento de **reativo** para **proativo** no *switch* S1 da topologia do exemplo anterior. Como se pode verificar, trata-se do mesmo conjunto de regras 3.5, acrescidas ao final de uma regra padrão instruindo o *switch* a consultar controlador sobre qual ação deve ser executada caso nenhuma das regras anteriores possa ser executada, o que inclui a regra de encaminhamento ao grupo *fast failover*. Caso não houvesse essa regra padrão, o fluxo seria naturalmente descartado em caso de falhas no enlace primário e nos de *backup*, porém sem a ciência do controlador. Em uma situação genérica de rede, isso impediria qualquer ação alternativa do plano de controle.

Código 3.6: Comandos para induzir recuperação de falha em modo proativo.

```
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-1",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
    :"1",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-2",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
    :"2",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-3",
    eth_type:"0x0800",priority:"10",idle_timeout:"5",active:"true",in_port
    :"3",eth_type:"0x800",ipv4_dst:"10.0.0.1",tcp_dst:"443",actions:"
    output=4"}' http://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",name:"flow-mod-4",
    eth_type:"0x0800",priority:"12",idle_timeout:"5",active:"true",in_port
    :"4",ipv4_dst:"10.0.0.4",tcp_src:"443",actions:"group=1"}' http
    ://11.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",actions:"output=
```

```
    controller"}' http://11.0.0.1:8080/wm/staticentrypusher/json
```

3.5.4 Migração para o modo de recuperação reativo

O modo reativo é nativo do OpenFlow, ou seja, sempre que chegar um pacote para o qual não houver correspondência em nenhuma das regras de fluxo instaladas na memória do *switch*, o controlador SDN deve ser consultado. Quando o AFRC instala nos *switches* regras de migração para o modo proativo, o campo de *idle timeout* (ver 2.3.3) será usado para remover as regras de fluxo da memória após um tempo de inatividade. Dessa forma, volta-se a consultar o controlador SDN no modo reativo, conforme instrução da regra de fluxo mostrada no código 3.7.

Código 3.7: Comandos para induz o modo reativo de recuperação de falha.

```
curl -X POST -d '{switch:"00:00:00:00:00:00:00:01",actions:"output=
    controller"}' http://127.0.0.1:8080/wm/staticentrypusher/json
```

Especificamente no caso de uso do grupo *Fast Failover*, quem efetivamente fará a remoção do grupo da memória do *switch* será o próprio AFRC, após monitorar a regra de fluxo que encaminha pacotes para este grupo. Isso torna-se possível justamente porque nesta regra há também um campo de *ilde timeout*.

Quando um grupo é criado, o campo ***name:"group_ff"***, é usado para identificação do nome do grupo, além do campo ***group_id:"1"***, usado para identificar o grupo na memória do *switch*. Além disso, quando uma regra de fluxo de encaminhamento para esse grupo é criada, é necessário indicar esse grupo através do campo ***actions:"group=1"***. De posse dessas duas informações, é possível através da linha de comando apresentada na Regar de Fluxos 3.8 determinar qual grupo será removido. No código 3.4 é possível verificar a correspondência desse exemplo.

Código 3.8: Comandos para a remoção de regras de recuperação proativa.

```
curl -X DELETE -d '{name:"GROUP_FF1"}' http://127.0.0.1:8080/wm/
    staticentrypusher/json
```

3.5.5 *Cold Water Bucket* (CWB)

Devido ao comportamento padrão do OpenFlow, quando não há uma regra no *switch* para um fluxo, o *switch* envia uma mensagem PACKET-IN ao controlador

perguntando qual ação deve tomar para o fluxo relacionado ao respectivo pacote entrante. O controlador SDN então responde ao *switch* via mensagem PACKET-OUT. Devido ao número de *switches* que podem compor uma rede, ou a um alto tráfego entre os dispositivos, essas solicitações podem gerar alto consumo de CPU no servidor que hospeda o controlador. Além disso, podem haver outras aplicações sendo executadas em paralelo no controlador SDN que poderão ser impactadas ou que poderão impactar a capacidade de resposta do controlador às mensagens PACKET-IN. Como para o funcionamento de uma SDN há uma dependência nativa do controlador, o método *Cold Water Bucket* (CWB) visa proteger o plano de dados sempre que os módulos que compõem o monitoramento do AFRC não sejam capazes de obter as informações necessárias para disparar as demais ações da fase de mitigação. Conforme dito, isso pode ocorrer justamente no caso de haver um alto consumo de CPU no controlador.

Portanto, o CWB é o método que o AFRC usa para migrar todas as regras de fluxo para o modo de recuperação proativo nos *switches* que estão conectados ao controlador SDN. Durante esse processo, todas as regras são sumarizadas [75] para minimizar a utilização da memória dos *switches*. Além disso, o CWB utilizará o *idle timeout* para controlar quando as regras serão removidas da memória dos *switches*.

O CWB não cria grupos *Fast Failover*, pois o módulo que faz o monitoramento da CPU do controlador não faz discriminação de tipo de tráfego, ou se há número de regras excessivas no *switch* ou alto tráfego *in-band* para o controlador. Apenas a "saúde" do servidor onde o controlador SDN está sendo executado é avaliada através do uso de sua CPU. Através de uma conexão via SSH (*Secure Socket Shell*) ao controlador, o AFRC faz esse monitoramento.

A Figura 3.14 apresenta a saída de um monitoramento do estado do controlador, onde é possível verificar qual controlador está sendo usado (Floodligh), o IP do controlador (127.0.0.1), o consumo de CPU (31%), o consumo de RAM (5.8 MB), a quantidade de mensagens de PACKET-IN (3055), os *switches* que estão sendo usados e suas respectivas regras de fluxo após sumarização.

3.6 Aspectos Operacionais do AFRC

O AFRC foi desenvolvido em *shell script*, por isso não é necessário ser compilado. Quando executado em linha de comando, o endereço IP do servidor onde está sendo executado o controlador SDN pode ser passado como parâmetro. Caso o AFRC

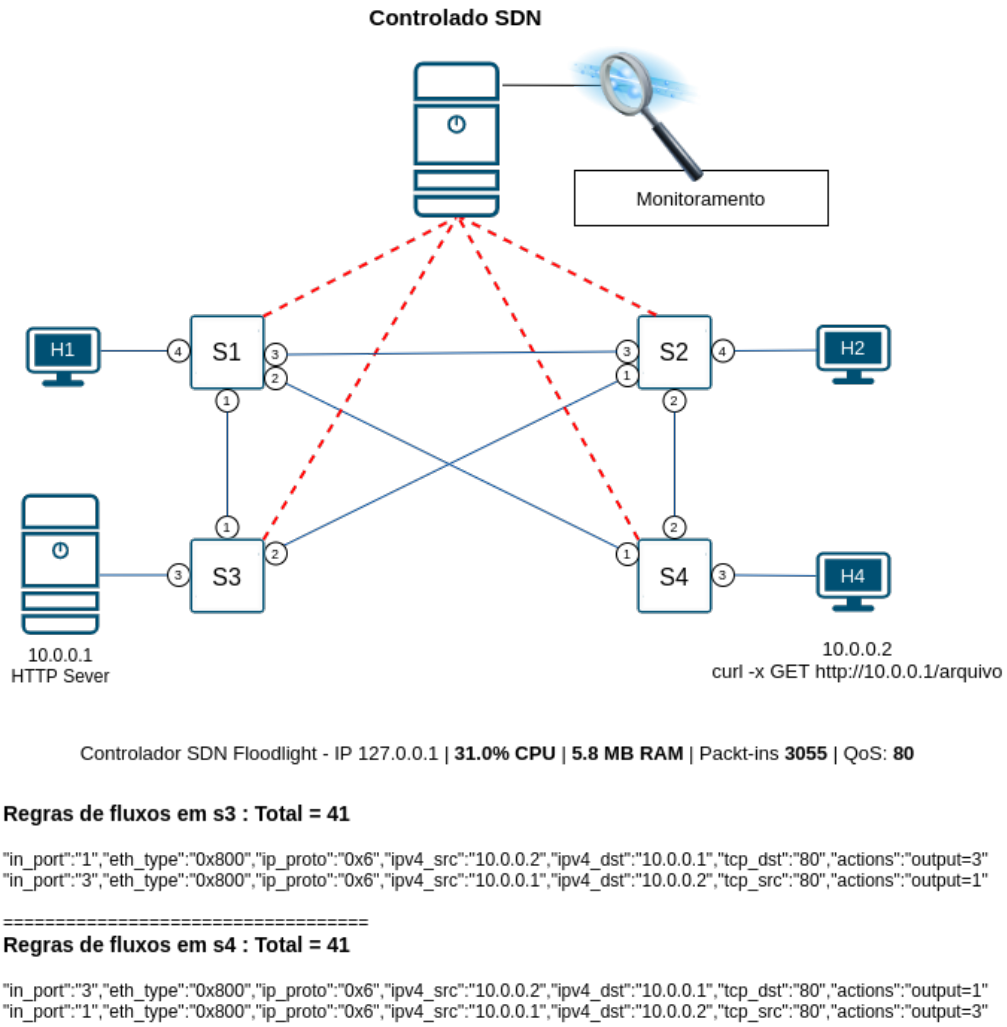


Figura 3.14: *S*aída do monitoramento do estado do controlador, visto pelo AFRC.

esteja sendo executado no mesmo hospedeiro do controlador SDN, esse controlador poderá ser identificado se estiver usando as portas TCP padrão do OpenFlow. Esse objetivo é alcançado porque, internamente, o AFRC realiza o seguinte comando caso não seja passado o IP: **“ip=$(netstat -atunp | grep 6[1-9][1-9]3 | sed ’1d’ | awk ’print $5)’ | cut -d ”:-f1 | uniq | head -n1”**. Portanto, para executar o AFRC basta digitar na linha de comando **afrc <IP do controlador SDN>**.

Caso haja um controlador SDN em execução, o AFRC irá coletar informações da “saúde” deste controlador. Essa etapa é executada internamente através do comando **flood=$(ps -aux | grep java | grep -v grep | awk ’print $3,$4,$13’ | sed ’s// /g’ | awk ”{print $1”% CPU |”,$2”MB RAM”})”**, o que pode também ser usado como painel de monitoramento do AFRC.

Na fase de investigação, através de *pollings* a cada 1s são coletadas as informações

de quais são os *switches* que estão conectados a este controlador, e para que este processo seja executado é usada a linha de comando **curl -s http://$ip:8080/wm/core/switch/all/desc/json**. Quando é detectado que um *switch* está conectado ao controlador na etapa anterior, o "ID" dele é identificado e essa informação será usada por todos os módulos do AFRC.

Alguns *scripts* externos podem ser invocados para executarem uma ação específica e, depois, são finalizados ou ficam rodando em *backgroud*. No código 3.9 é apresentado o *script* que monitora a quantidade de PACKET-INs que são enviados ao controlador.

Para que o AFRC possa executar a criação dos grupos *fast failover*, é executado dentro do *pooling* de monitoramento a porção do código apresentado no código 3.10, que tem a tarefa de executar a leitura em um arquivo denominado "qos.txt", caso exista. Se houver esse arquivo, ele conterá uma informação em formato textual contendo o número da porta de camada de transporte, origem ou destino, que deve ser vasculhada nas mensagens PACKET-INs enviadas ao controlador. Para proteger esses pacotes via grupos *fast failover*, é apresentado um exemplo de como criar esses grupos no código 3.13. Trata-se, portanto, de um arquivo de nome *"gff"* que deve ser criado pelos operadores da rede, conforme a configuração desejada para os grupos *fast failover*.

O módulo responsável por ler regras de fluxo dos *switches* e sumarizá-las é apresentado no código 3.12.

Quando o AFRC detecta a necessidade de migração para o modo proativo, o módulo responsável executa a ação apresentada no código 3.11, que consiste em ler os diretórios onde o AFRC gravou o resultado da sumarização das regras e gerar um aquivo que será lido na execução do envio de regras para os *switches* via controlador SDN. O código 3.14 apresenta um exemplo de um aquivo com regras sumarizadas e de proteção, usando os grupos *fast failover*.

Código 3.9: Monitoramento do tráfego de mensagens PACKET-IN.

```
xterm -e "tshark -d tcp.port==6653,openflow -O openflow_v5 -n -i any -Y '
    openflow_v5.type == 10'
```

Código 3.10: Porção de código que lê o arquivo *"qos.txt"*.

```
..
qos=$(cat qos.txt)
       if [ -z "$qos" == "" ]; then
            .\criargrupos
       fi
..
```

Código 3.11: Geração de arquivo com regras que serão enviadas aos *switches*.

```
#!/bin/bash
lista=$(ls /scripts/logs/flows/S*)
cat $lista | sed 's/"tcp_src":"\*",//g' | sed 's/tcp_dst:"\*",//g' | sed '
    1s/^/#!\/bin\/bash\n/' | sed '2s/^/i=1\n/' | tee flows
chmod +x addflows
```

Código 3.12: Leitura e sumarização das regras de fluxo dos *switches*.

```
 curl -s http://$ip:8080/wm/core/switch/$f1/flow/json | sed 's/.}}},/&\'$'
    \n/g' | sed 's/,/ /g' | awk '/in_port/ {print $12,$13,$14,$15,$16,$17
    ,$18,$19,$20,$21,$22}' | sed 's/:{/ /g' | sed 's/}}}/ /g' | awk '{
    print $2,$3,$4,$5,$6,$7,$8,$9,$10,$14}' | sed 's/ /,/g' |sed 's/"/ /g
    ' | awk '!x[$0]++' | awk '{if(($35) <= 1023)sub(($31),"*"); else $35
    ="*"; print $0}' | awk '!x[$0]++' | sed 's/^ //g' | sed 's/^/ /g' |
    sed 's/ /"/g' | sed 's/$/"/' | sed 's/""/"/g' | sed 's/"tcp_src
    ":"\*",//g' | sed 's/tcp_dst:"\*",//g' | sed "s/^/curl -X POST -d '{\
    "switch\":\"$f1\",\"name\":\"$flow\",\"priority\":\"1\",\"
    idle_timeout\":\"5\",\"active\":\"true\",/" | sed "s/$/}' http:\/\/
    $ip:8080\/wm\/staticentrypusher\/json ; i=$\(\(i+1\)\)/" | tee /
    scripts/logs/flows/$f2
find /scripts/logs/flows/ -type f -size +1c -exec cp {} /scripts/logs/
    flows/fluxos/ \; 2> /dev/null
echo -e "\n"
```

Código 3.13: Arquivo usado para criar grupos *fast failover*.

```
curl -X POST -d '{:"00:00:00:00:00:00:00:01",name:"FF1",entry_type:"group
    ",active:"true",group_id:"1",group_type:"fast_failover","group_buckets
    ": [{bucket_id:"1",bucket_watch_port:"2",bucket_actions:"output=2"},{
    bucket_id:"2",bucket_watch_port:"3",bucket_actions:"output=3"}]}' http
    ://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{:"00:00:00:00:00:00:00:02",name:"FF2",entry_type:"group
    ",active:"true",group_id:"2",group_type:"fast_failover","group_buckets
    ": [{bucket_id:"1",bucket_watch_port:"1",bucket_actions:"output=1"},{
    bucket_id:"2",bucket_watch_port:"2",bucket_actions:"output=2"}]}' http
    ://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{:"00:00:00:00:00:00:00:03",name:"FF3",entry_type:"group
    ",active:"true",group_id:"3",group_type:"fast_failover","group_buckets
    ": [{bucket_id:"1",bucket_watch_port:"1",bucket_actions:"output=1"},{
    bucket_id:"2",bucket_watch_port:"2",bucket_actions:"output=2"}]}' http
    ://127.0.0.1:8080/wm/staticentrypusher/json
curl -X POST -d '{:"00:00:00:00:00:00:00:04",name:"FF4",entry_type:"group
    ",active:"true",group_id:"4",group_type:"fast_failover","group_buckets
    ": [{bucket_id:"1",bucket_watch_port:"1",bucket_actions:"output=1"},{
    bucket_id:"2",bucket_watch_port:"2",bucket_actions:"output=2"}]}' http
    ://127.0.0.1:8080/wm/staticentrypusher/json
```

Código 3.14: Exemplo de criação de regras sumarizadas e grupos *fast failover*.

```
curl -X POST -d '{:"00:00:00:00:00:00:00:01",name:"flow-mod-'$i'",name
    :"1",name:"2",eth_dst:"00:00:00:00:00:01",eth_src
    :"00:00:00:00:00:02","eth_type":"0x800","ip_proto":"0x6","ipv4_src
    ":"10.0.0.2","ipv4_dst":"10.0.0.1",tcp_dst:"80",actions:"output=1"}'
    http://127.0.0.1:8080/wm/staticentrypusher/json ; i=$((i+1))
curl -X POST -d '{:"00:00:00:00:00:00:00:01",name:"flow-mod-'$i'",name
    :"1",name:"3",eth_dst:"00:00:00:00:00:01",eth_src
    :"00:00:00:00:00:02","eth_type":"0x800","ip_proto":"0x6","ipv4_src
    ":"10.0.0.2","ipv4_dst":"10.0.0.1",tcp_dst:"80",actions:"output=1"}'
    http://127.0.0.1:8080/wm/staticentrypusher/json ; i=$((i+1))
curl -X POST -d '{:"00:00:00:00:00:00:00:01",name:"flow-mod-'$i'",name
    :"100",name:"1",eth_dst:"00:00:00:00:00:02",eth_src
    :"00:00:00:00:00:01","eth_type":"0x800","ip_proto":"0x6","ipv4_src
    ":"10.0.0.1","ipv4_dst":"10.0.0.2","tcp_src":"80",actions:"group=1"}'
    http://127.0.0.1:8080/wm/staticentrypusher/json ; i=$((i+1))
```

4. Validação e Análise da Proposta

Este capítulo descreve a metodologia experimental e alguns aspectos técnicos sobre a implantação de um protótipo da solução AFRC para fins de validação e análise. Serão abordadas as tecnologias utilizadas para a implementação da solução e do cenário experimental, assim como algumas decisões que foram tomadas para assegurar o cumprimento dos objetivos do projeto. Além disso, a solução proposta será analisada quanto a seu uso em casos particulares e em cenários mais amplos, onde serão abordadas limitações e alguns aspectos de ordem prática.

4.1 Ambiente computacional

Por uma questão de flexibilidade e redução de custo na implementação do protótipo, optou-se pela adoção da ferramenta *mininet* [76] como forma de emular o plano de dados dos cenários experimentais. Através do mininet, torna-se possível criar *switches* OpenFlow e *hosts*, assim como os enlaces de rede que os interligam. Da mesma forma, torna-se possível criar um canal de comunicação dedicado entre cada *switch* e a maquina que abrigará o controlador adotado, neste caso o Floodlight [25].

Podemos ressaltar também que nenhum parâmetro adicional, tais como largura de banda e atraso, foi configurado no mininet, de forma que todo recurso computacional disponível da maquina hospedeira seja utilizado.

Este ambiente experimental foi implementado através de um *laptop* destinado a executar o controlador SDN e emular uma rede física com *switches* OpenFlow virtualizados pelo Open vSwitch[1], com sistema operacional Linux Ubuntu 18.04 e processador Intel i3 de 64 bits com 8 (oito) GB de RAM.

[1]Disponível em `http://openvswitch.org/`.

4.2 Cenário Experimental

O cenário experimental adotado corresponde a uma topologia reduzida representando uma rede de *datacenter* baseada na arquitetura *spine and leaf* com sua configuração mínima possível, conforme ilustrado na Figura 4.1. O objetivo de se usar essa topologia reduzida é para simplificar o número de caminhos alternativos entre os sistemas finais, a serem usados para prover uma abordagem proativa de proteção. Foi utilizado um servidor HTTP escrito em Python[2] aguardando conexões do cliente na porta TCP 80. No lado do cliente, foi executado um *script* próprio, conforme apresentado na Listagem 4.1. Para gerar fluxos contínuos na rede da Figura 4.1, o cliente se conectou ao servidor remoto a cada 1 segundo para fazer o *download* de um aquivo de 10GB, utilizando o curl como ferramenta de transferência de arquivo via HTTP.

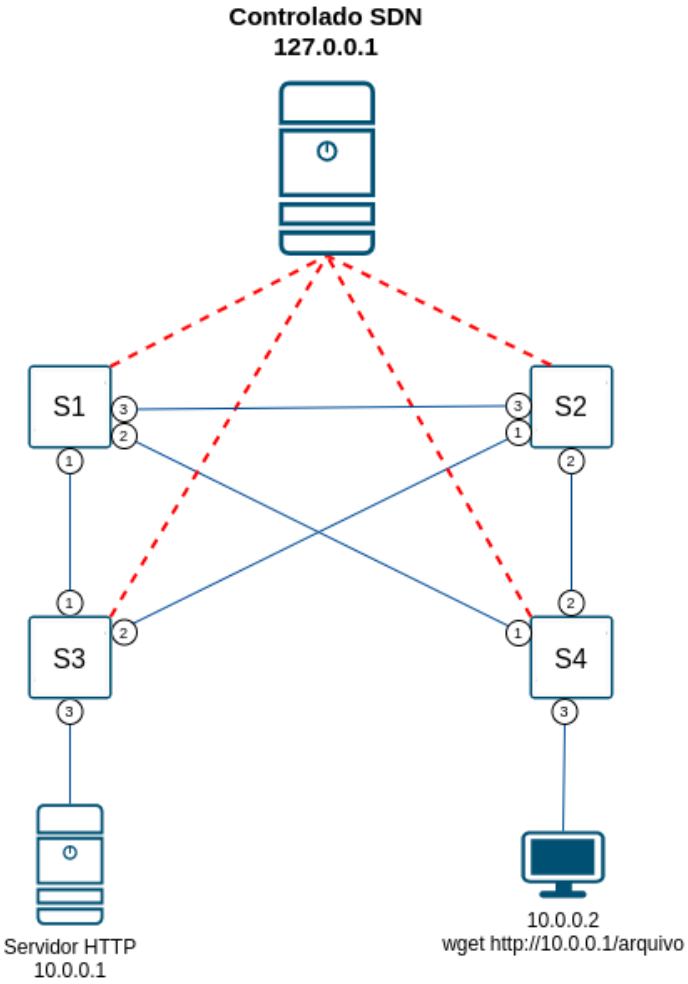


Figura 4.1: Cenário experimental para validação.

[2]https://www.python.org/

Código 4.1: Script para gerar tráfego HTTP.

```
#!/bin/bash
While :
do
    curl -so /dev/null -w '%{time_total} segs\n' http://10.0.0.1/arquivo |
        ts >> downloads
    sleep 1; clear
done
```

No cenário de validação, para que o *host* 10.0.0.2 realize a transferência de arquivo, ele precisa se conectar ao servidor e, para isso, o *switch* no qual ele está conectado precisa ter regras de fluxos instaladas. Da mesma forma, para o servidor HTTP atender às solicitações do cliente, é necessário também que o *switch* no qual ele está conectado tenha regras de fluxo instaladas. Essas regras de fluxo são apresentadas na Listagem 4.2, instaladas reativamente pelo controlador nos *switches* após os fluxos HTTP de ida e volta serem gerados. O caminho de ida e volta que reflete as regras de encaminhamento instaladas pelo controlador, obtidas a partir da execução do algoritmo de *spanning tree* nativo do Floodlight, está representado pela linha verde na Figura 4.2.

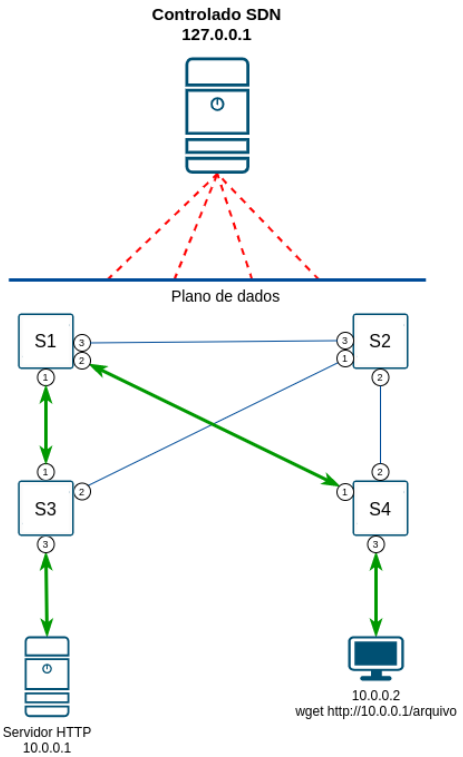


Figura 4.2: Caminho selecionado pelo controlador Floodlight.

Código 4.2: Regras de fluxo inicialmente instaladas nos *switches*.

```
Regras de fluxos em S1 : Total de fluxos = 9
{flows:[{version:"OF_14" cookie:"9007230191927296" table_id:"0x0"
    packet_count:"1" byte_count:"74" duration_sec:"4" duration_nsec:"
    192000000" priority:"1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[]
     match in_port:"2" eth_dst:"00:00:00:00:00:01" eth_src:"
    00:00:00:00:00:02" eth_type:"0x800" ip_proto:"0x6" ipv4_src:"10.0.0.2"
     ipv4_dst:"10.0.0.1" tcp_src:"35116" tcp_dst:"80" ovs_tcp_flags:"0
    x0002"} instructions "instruction_apply_actions" actions:"output=1"
{version:"OF_14" cookie:"0" table_id:"0x0" packet_count:"15030" byte_count
    :"2980257" duration_sec:"2856" duration_nsec:"445000000" priority:"0"
    idle_timeout_s:"0" hard_timeout_s:"0" flags:[] match } instructions "
    instruction_apply_actions" actions:"output=controller" ]}

{version:"OF_14" cookie:"9007230208704512" table_id:"0x0" packet_count:"1"
     byte_count:"66" duration_sec:"4" duration_nsec:"191000000" priority:"
    1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[] match in_port:"1"
    eth_dst:"00:00:00:00:00:02" eth_src:"00:00:00:00:00:01" eth_type:"0
    x800" ip_proto:"0x6" ipv4_src:"10.0.0.1" ipv4_dst:"10.0.0.2" tcp_src:"
    80" tcp_dst:"35116" ovs_tcp_flags:"0x0012"} instructions "
    instruction_apply_actions" actions:"output=2"

{as demais regras foram omitidas, pois representam o mesmo contexto}
==================================
Regras de fluxos em S2 : Total de fluxos = 1
{"flows":[{version:"OF_14" cookie:"0" table_id:"0x0" packet_count:"668"
    byte_count:"272520" duration_sec:"2856" duration_nsec:"470000000"
    priority:"0" idle_timeout_s:"0" hard_timeout_s:"0" flags:[] match }
    instructions "instruction_apply_actions" actions:"output=controller"
    ]}

==================================
Regras de fluxos em S3 : Total de fluxos = 9
{"flows":[{version:"OF_14" cookie:"9007230191927296" table_id:"0x0"
    packet_count:"1" byte_count:"74" duration_sec:"4" duration_nsec:"
    254000000" priority:"1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[]
     match in_port:"1" eth_dst:"00:00:00:00:00:01" eth_src:"
    00:00:00:00:00:02" eth_type:"0x800" ip_proto:"0x6" ipv4_src:"10.0.0.2"
     ipv4_dst:"10.0.0.1" tcp_src:"35116" tcp_dst:"80" ovs_tcp_flags:"0
    x0002"} instructions "instruction_apply_actions" actions:"output=3"
```

{version:"OF_14" cookie:"0" table_id:"0x0" packet_count:"749" byte_count:"
2093263" duration_sec:"2856" duration_nsec:"503000000" priority:"0"
idle_timeout_s:"0" hard_timeout_s:"0" flags:[] match } instructions "
instruction_apply_actions" actions:"output=controller"]}

{version:"OF_14" cookie:"9007230208704512" table_id:"0x0" packet_count:"4"
byte_count:"13313" duration_sec:"4" duration_nsec:"253000000"
priority:"1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[] match
in_port:"3" eth_dst:"00:00:00:00:00:02" eth_src:"00:00:00:00:00:01"
eth_type:"0x800" ip_proto:"0x6" ipv4_src:"10.0.0.1" ipv4_dst:"10.0.0.2
" tcp_src:"80" tcp_dst:"35116" ovs_tcp_flags:"0x0012"} instructions "
instruction_apply_actions" actions:"output=1"

{as demais regras foram omitidas, pois representam o mesmo contexto}
==================================
Regras de fluxos em S4 : Total de fluxos = 9
{"flows":[{version:"OF_14" cookie:"9007230191927296" table_id:"0x0"
packet_count:"0" byte_count:"0" duration_sec:"4" duration_nsec:"
284000000" priority:"1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[]
match in_port:"3" eth_dst:"00:00:00:00:00:01" eth_src:"
00:00:00:00:00:02" eth_type:"0x800" ip_proto:"0x6" ipv4_src:"10.0.0.2"
ipv4_dst:"10.0.0.1" tcp_src:"35116" tcp_dst:"80" ovs_tcp_flags:"0
x0002"} instructions "instruction_apply_actions" actions:"output=1"
{version:"OF_14" cookie:"0" table_id:"0x0" packet_count:"788" byte_count:"
2043124" duration_sec:"2856" duration_nsec:"538000000" priority:"0"
idle_timeout_s:"0" hard_timeout_s:"0" flags:[] match } instructions "
instruction_apply_actions" actions:"output=controller"]}

{version:"OF_14" cookie:"9007230208704512" table_id:"0x0" packet_count:"1"
byte_count:"74" duration_sec:"4" duration_nsec:"283000000" priority:"
1" idle_timeout_s:"5" hard_timeout_s:"0" flags:[] match in_port:"1"
eth_dst:"00:00:00:00:00:02" eth_src:"00:00:00:00:00:01" eth_type:"0
x800" ip_proto:"0x6" ipv4_src:"10.0.0.1" ipv4_dst:"10.0.0.2" tcp_src:"
80" tcp_dst:"35116" ovs_tcp_flags:"0x0012"} instructions "
instruction_apply_actions" actions:"output=3"
{as demais regras foram omitidas, pois representam o mesmo contexto}

Com base na rota "primária" gerada pelo controlador, foi instalado um caminho de *backup*, tanto de ida quanto de volta, para o respectivo fluxo HTTP usando os grupos *fast failover*. Esse caminho é representado pela linha vermelha na Figura 4.3. O código 4.3 mostra como ficam as regras nos *switches* ao se incorporar esses caminhos de proteção.

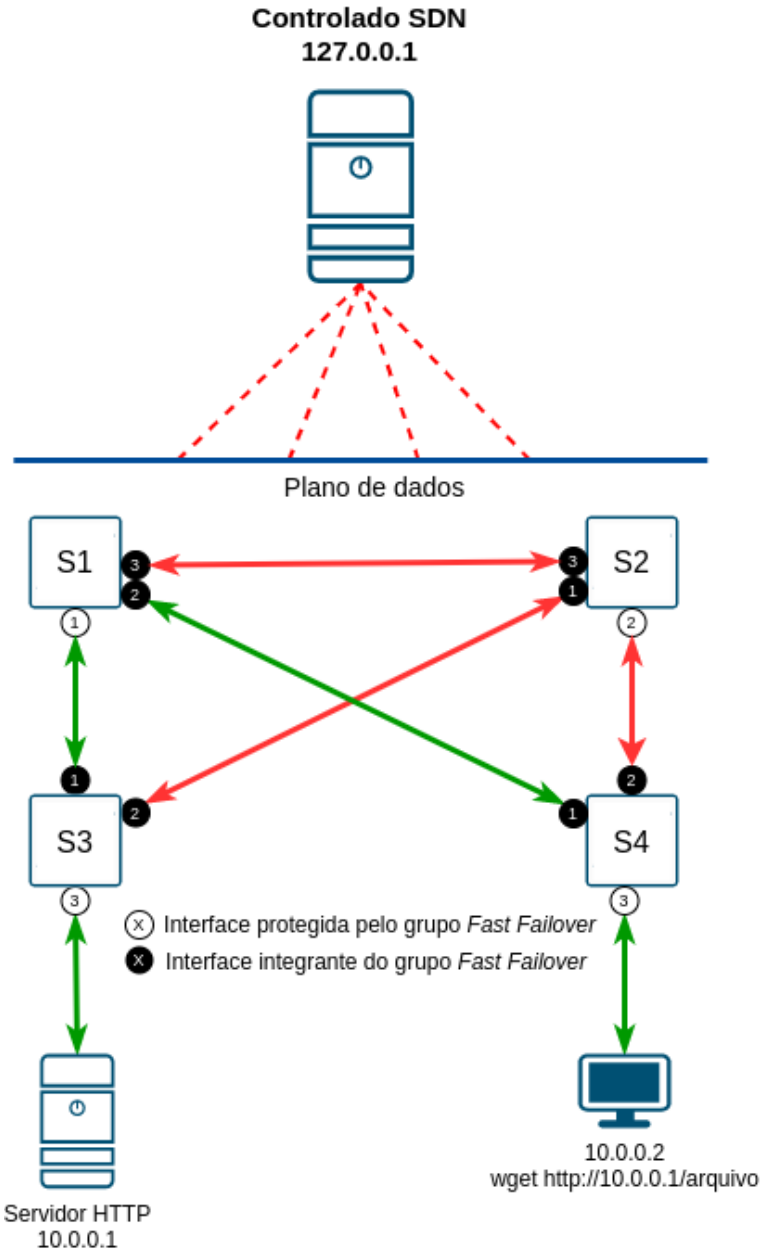


Figura 4.3: Caminho de *backup* com grupos *fast failover*.

Código 4.3: Regras de fluxo instaladas nos *switches* para fins de proteção.

```
00:00:00:00:00:00:00:01
[{"FF1":{version:"OF_14",command:"ADD",group_number:"1",group_type:
    fast_failover,version:"OF_14",xid:"53621",group_buckets:[{bucket_id:"1
    ",bucket_watch_group:"all",bucket_watch_port:"2",bucket_weight:"0","
    bucket_actions":"output=2"},{bucket_id:"2",bucket_watch_group:"all",
    bucket_watch_port:"3",bucket_weight:"0",bucket_actions:"output=3"}]}},

{flow-mod-1:{version:"OF_14",command:"ADD",cookie:"45035997351236006",
    priority:"1",idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",flags
    :"1",cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"2",
    in_port:"00:00:00:00:00:01",in_port:"00:00:00:00:00:02",eth_type:"0
    x800",ip_proto:"0x6",ipv4_src:"10.0.0.2",ipv4_dst:"10.0.0.1",tcp_dst:"
    80"},instructions:{"instruction_apply_actions":{actions:"output=1"
    }}}},{"flow-mod-3":{version:"OF_14",command:"ADD",cookie:"
    45035997351236008",priority:"100",idleTimeoutSec:"0",hardTimeoutSec:"0
    ",outPort:"any",flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",
    match:{in_port:"1",in_port:"00:00:00:00:00:02",in_port:"
    00:00:00:00:00:01",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.1"
    ,ipv4_dst:"10.0.0.2","tcp_src":"80"},instructions:{"
    instruction_apply_actions":{actions:"group=1"}}}},{"flow-mod-2":{
    version:"OF_14",command:"ADD",cookie:"45035997351236007",priority:"1",
    idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",flags:"1",
    cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"3",in_port
    :"00:00:00:00:00:01",in_port:"00:00:00:00:00:02",eth_type:"0x800",
    ip_proto:"0x6",ipv4_src:"10.0.0.2",ipv4_dst:"10.0.0.1",tcp_dst:"80"},
    instructions:{"instruction_apply_actions":{actions:"output=1"}}}}]}

00:00:00:00:00:00:00:02
[{"FF2":{version:"OF_14",command:"ADD",group_number:"2",group_type:
    fast_failover,version:"OF_14",xid:"53623",group_buckets:[{bucket_id:"1
    ",bucket_watch_group:"all",bucket_watch_port:"1",bucket_weight:"0","
    bucket_actions":"output=1"},{bucket_id:"2",bucket_watch_group:"all",
    bucket_watch_port:"2",bucket_weight:"0","bucket_actions":"output=2"
    }]}},

{"flow-mod-6":{version:"OF_14",command:"ADD",cookie:"45035997351236011",
    priority:"100",idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",
    flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"
```

```
    2",in_port:"00:00:00:00:00:01",in_port:"00:00:00:00:00:02",eth_type:"0
    x800",ip_proto:"0x6",ipv4_src:"10.0.0.2",ipv4_dst:"10.0.0.1",tcp_dst:"
    80"},instructions:{"instruction_apply_actions":{actions:"group=2"
    }}}},{"flow-mod-5":{version:"OF_14",command:"ADD",cookie:"
    45035997351236010",priority:"1",idleTimeoutSec:"0",hardTimeoutSec:"0",
    outPort:"any",flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",
    match:{in_port:"1",in_port:"00:00:00:00:00:02",in_port:"
    00:00:00:00:00:01",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.1"
    ,ipv4_dst:"10.0.0.2","tcp_src":"80"},instructions:{"
    instruction_apply_actions":{actions:"output=2"}}}},{"flow-mod-4":{
    version:"OF_14",command:"ADD",cookie:"45035997351236009",priority:"1",
    idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",flags:"1",
    cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"3",in_port
    :"00:00:00:00:00:02",in_port:"00:00:00:00:00:01",eth_type:"0x800",
    ip_proto:"0x6",ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.2","tcp_src":"80"
    },instructions:{"instruction_apply_actions":{actions:"output=2"}}}}],

00:00:00:00:00:00:00:03
[{"FF3":{version:"OF_14",command:"ADD",group_number:"3",group_type:
    fast_failover,version:"OF_14",xid:"53625",group_buckets:[{bucket_id:"1
    ",bucket_watch_group:"all",bucket_watch_port:"1",bucket_weight:"0","
    bucket_actions":"output=1"},{bucket_id:"2",bucket_watch_group:"all",
    bucket_watch_port:"2",bucket_weight:"0","bucket_actions":"output=2"
    }]}},

{"flow-mod-9":{version:"OF_14",command:"ADD",cookie:"45035997351236014",
    priority:"100",idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",
    flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"
    3",in_port:"00:00:00:00:00:02",in_port:"00:00:00:00:00:01",eth_type:"0
    x800",ip_proto:"0x6",ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.2","tcp_src"
    :"80"},instructions:{"instruction_apply_actions":{actions:"group=3"
    }}}},{"flow-mod-8":{version:"OF_14",command:"ADD",cookie:"
    45035997351236013",priority:"1",idleTimeoutSec:"0",hardTimeoutSec:"0",
    outPort:"any",flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",
    match:{in_port:"1",in_port:"00:00:00:00:00:01",in_port:"
    00:00:00:00:00:02",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.2"
    ,ipv4_dst:"10.0.0.1",tcp_dst:"80"},instructions:{"
    instruction_apply_actions":{actions:"output=3"}}}},{"flow-mod-7":{
    version:"OF_14",command:"ADD",cookie:"45035997351236012",priority:"1",
    idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",flags:"1",
```

```
    cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"2",in_port
    :"00:00:00:00:00:01",in_port:"00:00:00:00:00:02",eth_type:"0x800",
    ip_proto:"0x6",ipv4_src:"10.0.0.2",ipv4_dst:"10.0.0.1",tcp_dst:"80"},
    instructions:{"instruction_apply_actions":{actions:"output=3"}}}}],

00:00:00:00:00:00:00:04
[{"FF4":{version:"OF_14",command:"ADD",group_number:"4",group_type:
    fast_failover,version:"OF_14",xid:"53627",group_buckets:[{bucket_id:"1
    ",bucket_watch_group:"all",bucket_watch_port:"1",bucket_weight:"0","
    bucket_actions":"output=1"},{bucket_id:"2",bucket_watch_group:"all",
    bucket_watch_port:"2",bucket_weight:"0","bucket_actions":"output=2"
    }]}},

{"flow-mod-10":{version:"OF_14",command:"ADD",cookie:"49539595626859522",
    priority:"100",idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",
    flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"
    3",in_port:"00:00:00:00:00:01",in_port:"00:00:00:00:00:02",eth_type:"0
    x800",ip_proto:"0x6",ipv4_src:"10.0.0.2",ipv4_dst:"10.0.0.1",tcp_dst:"
    80"},instructions:{"instruction_apply_actions":{actions:"group=4"
    }}}},{"flow-mod-12":{version:"OF_14",command:"ADD",cookie:"
    49539595626859524",priority:"1",idleTimeoutSec:"0",hardTimeoutSec:"0",
    outPort:"any",flags:"1",cookieMask:"0",outGroup:"any",tableId:"0x0",
    match:{in_port:"2",in_port:"00:00:00:00:00:02",in_port:"
    00:00:00:00:00:01",eth_type:"0x800",ip_proto:"0x6",ipv4_src:"10.0.0.1"
    ,ipv4_dst:"10.0.0.2","tcp_src":"80"},instructions:{"
    instruction_apply_actions":{actions:"output=3"}}}},{"flow-mod-11":{
    version:"OF_14",command:"ADD",cookie:"49539595626859523",priority:"1",
    idleTimeoutSec:"0",hardTimeoutSec:"0",outPort:"any",flags:"1",
    cookieMask:"0",outGroup:"any",tableId:"0x0",match:{in_port:"1",in_port
    :"00:00:00:00:00:02",in_port:"00:00:00:00:00:01",eth_type:"0x800",
    ip_proto:"0x6",ipv4_src:"10.0.0.1",ipv4_dst:"10.0.0.2","tcp_src":"80"
    },instructions:{"instruction_apply_actions":{actions:"output=3"}}}}],
```


Com base nesse arranjo, criou-se um cenário para se avaliar o comportamento da recuperação de falhas proativas do AFRC quando aplicadas apenas ao fluxo HTTP aqui definido, envolvendo os *hosts* citados. Para quaisquer outros fluxos, o modo de recuperação será reativo.

A forma de avaliar a eficácia da recuperação de falha adaptativa do AFRC foi

comparar a rapidez de reação a uma interrupção de enlace no modo reativo padrão do OpenFlow com quando se usa os grupos *fast failover* para os fluxos HTTP. No caso, esse efeito foi medido através da observação do tempo necessário para transferência de aquivo utilizando o HTTP, pois há um acréscimo nesse tempo logo após uma falha em função da necessidade do controlador instalar novas regras de encaminhamento quando se usa o modo reativo. Todavia, espera-se que no modo "adaptativo", ou seja, quando há grupos *fast failover* para atender o tráfego HTTP, isso não ocorra.

O CWB e a migração do modo proativo para o modo reativo também foram ensaiados. Para isso, foi usada a mesma topologia apresentada na Figura 4.3. Foram gerados fluxos HTTP entre cliente e servidor, com a rede no modo reativo, de maneira a popular as tabelas de fluxo dos *switches*. Feito isso, a mesma falha no enlace S1 — S4 foi emulada e seu efeito sobre o controlador foi registrado. em uma segunda etapa, aplicou-se o CWB, gerando a aplicação de regras de *backup* com sumarização nos *switches*. Novamente, a mesma falha foi emulada e seu efeito sobre o controlador registrado. Por fim, aplicou-se uma migração de modo proativo para reativo nos *switches* da rede, mostrando que a rede voltou a ter o comportamento esperado.

4.3 Resultados

De maneira a se verificar os efeitos do AFRC em função de uma ação de recuperação de falha, o processo de transferência de arquivo utilizando o HTTP, foi iniciado juntos com um único fluxo de *pings*. A adição desse fluxo de *pings* é para se ter um fluxo que não faça uso dos grupos *fast failover*.

A partir da verificação do uso do caminho primário por esses fluxos (S3 — S1 — S4), forçou-se então a desativação da interface 2 do *switch* S1, ligada à interface 1 de *S4*. Quando esse enlace foi desativado, o tráfego HTTP, tanto de ida quanto de volta, passou a ser direcionado para o caminho S3 — S1 — S2 — S4 quando se tem as regras de proteção já instaladas. Quando não se tem as regras de proteção, o novo caminho para o tráfego HTTP será definido reativamente pelo controlador. Já para o tráfego de ping, independente de haver ou não regras de proteção, o novo caminho após uma falha é sempre definido reativamente pelo controlador. Sendo assim, a reação à inativação da referida interface foi observada tanto no modo reativo quanto no modo proativo, para fins de comparação. A variação na latência

do ping, assim como no tempo de *download* do HTTP, foram medidos em ambos os casos, observando-se os respectivos tempos antes, imediatamente após e algum tempo depois da inativação (ou "falha") da interface. Os eventos de falha foram repetidos três vezes para minimamente caracterizar a variabilidade desses tempos e certificar o comportamento observado.

A Tabela 4.1 apresenta o resultado dos pings com a rede em modo reativo e sem que ocorra qualquer falha. É possível notar que os três primeiros pacotes têm um tempo de ida e volta mais alto, aproximadamente entre 0,5 e 0,7 ms, e posteriormente esse tempo decai cerca de uma ordem de grandeza, indo para um valor de 0,04, aproximadamente. Essa variação é explicada pela ausência de regras nos *switches* da rede a medida que os primeiros pacotes são enviados, o que exige comunicação com o controlador para que as regras de encaminhamento sejam instaladas. Esse processo só será satisfatoriamente concluído após o terceiro pacote, quando então se terá um caminho através de três *switches* com as devidas regras já instaladas, não mais sendo necessário uma consulta ao controlador.

20:12:04:	1408 bytes from 10.0.0.1	icmp_seq=3 ttl=64 time=0.767 ms
20:12:05:	1408 bytes from 10.0.0.1	icmp_seq=4 ttl=64 time=0.766 ms
20:12:06:	1408 bytes from 10.0.0.1	icmp_seq=5 ttl=64 time=0.523 ms
20:12:07:	1408 bytes from 10.0.0.1	icmp_seq=6 ttl=64 time=0.043 ms
20:12:08:	1408 bytes from 10.0.0.1	icmp_seq=7 ttl=64 time=0.044 ms

Tabela 4.1: Atrasos dos pings com rede iniciada em modo reativo

As tabelas 4.2, 4.3 e 4.4 apresentam os atrasos de ida e volta dos pings em três repetições de um mesmo experimento com a rede apenas em modo reativo. Uma falha no enlace S1 — S4 foi introduzida nos *timestamps* 04:06:44, 04:07:35 e 04:08:05, causando elevação significativa do tempo de resposta do ping logo após as falhas, em função da necessidade de consulta ao controlador para a instalação de novas regras de encaminhamento, conforme esperado.

04:06:39:	1408 bytes from 10.0.0.1:	icmp_seq=12 ttl=64 time=0.031 ms
04:06:40:	1408 bytes from 10.0.0.1:	icmp_seq=13 ttl=64 time=0.061 ms
04:06:41:	1408 bytes from 10.0.0.1:	icmp_seq=14 ttl=64 time=0.063 ms
04:06:42:	1408 bytes from 10.0.0.1:	icmp_seq=15 ttl=64 time=0.045 ms
04:06:44:	**1408 bytes from 10.0.0.1:**	**icmp_seq=17 ttl=64 time=4.26 ms**
04:06:45:	1408 bytes from 10.0.0.1:	icmp_seq=18 ttl=64 time=0.055 ms
04:06:46:	1408 bytes from 10.0.0.1:	icmp_seq=19 ttl=64 time=0.054 ms
04:06:47:	1408 bytes from 10.0.0.1:	icmp_seq=20 ttl=64 time=0.066 ms
04:06:48:	1408 bytes from 10.0.0.1:	icmp_seq=21 ttl=64 time=0.067 ms

Tabela 4.2: Atrasos dos pings em modo reativo (rodada 1).

04:07:31:	1408 bytes from 10.0.0.1:	icmp_seq=63 ttl=64 time=0.063 ms
04:07:32:	1408 bytes from 10.0.0.1:	icmp_seq=64 ttl=64 time=0.038 ms
04:07:33:	1408 bytes from 10.0.0.1:	icmp_seq=65 ttl=64 time=0.060 ms
04:07:34:	1408 bytes from 10.0.0.1:	icmp_seq=66 ttl=64 time=0.063 ms
04:07:35:	**1408 bytes from 10.0.0.1:**	**icmp_seq=67 ttl=64 time=5.18 ms**
04:07:36:	1408 bytes from 10.0.0.1:	icmp_seq=68 ttl=64 time=0.032 ms
04:07:37:	1408 bytes from 10.0.0.1:	icmp_seq=69 ttl=64 time=0.061 ms
04:07:38:	1408 bytes from 10.0.0.1:	icmp_seq=70 ttl=64 time=0.056 ms

Tabela 4.3: Atrasos dos pings em modo reativo (rodada 2).

04:08:01:	1408 bytes from 10.0.0.1:	icmp_seq=92 ttl=64 time=0.065 ms
04:08:02:	1408 bytes from 10.0.0.1:	icmp_seq=93 ttl=64 time=0.061 ms
04:08:03:	1408 bytes from 10.0.0.1:	icmp_seq=94 ttl=64 time=0.060 ms
04:08:04:	1408 bytes from 10.0.0.1:	icmp_seq=95 ttl=64 time=0.065 ms
04:08:05:	**1408 bytes from 10.0.0.1:**	**icmp_seq=96 ttl=64 time=4.58 ms**
04:08:06:	1408 bytes from 10.0.0.1:	icmp_seq=97 ttl=64 time=0.557 ms
04:08:07:	1408 bytes from 10.0.0.1:	icmp_seq=98 ttl=64 time=0.057 ms
04:08:08:	1408 bytes from 10.0.0.1:	icmp_seq=99 ttl=64 time=0.068 ms
04:08:09:	1408 bytes from 10.0.0.1:	icmp_seq=100 ttl=64 time=0.065 ms

Tabela 4.4: Atrasos dos pings em modo reativo (rodada 3).

As tabelas 4.5, 4.6 e 4.7 apresentam novos atrasos de ida e volta dos pings em três repetições de um mesmo experimento, mas desta vez com a rede contendo as regras aprendidas reativamente e também incorporando as regras de proteção para o tráfego HTTP. Vale destacar que, durante este experimento, o tráfego HTTP estava ocorrendo em paralelo. Dessa vez, uma falha no enlace S1 — S4 foi introduzida nos *timestamps* 03:37:03, 03:49:11 e 03:53:46, causando elevação significativa do tempo de resposta do ping logo após as falhas, em função da necessidade de consulta ao controlador para a instalação de novas regras de encaminhamento uma vez que os grupos *fast failover* atendem apenas ao tráfego HTTP.

03:36:59:	1408 bytes from 10.0.0.1:	icmp_seq=12 ttl=64 time=0.051 ms
03:37:00:	1408 bytes from 10.0.0.1:	icmp_seq=13 ttl=64 time=0.065 ms
03:37:01:	1408 bytes from 10.0.0.1:	icmp_seq=14 ttl=64 time=0.060 ms
03:37:02:	1408 bytes from 10.0.0.1:	icmp_seq=15 ttl=64 time=0.060 ms
03:37:03:	**1408 bytes from 10.0.0.1:**	**icmp_seq=16 ttl=64 time=7.22 ms**
03:37:04:	1408 bytes from 10.0.0.1:	icmp_seq=17 ttl=64 time=0.064 ms
03:37:05:	1408 bytes from 10.0.0.1:	icmp_seq=18 ttl=64 time=0.048 ms
03:37:06:	1408 bytes from 10.0.0.1:	icmp_seq=19 ttl=64 time=0.071 ms
03:37:07:	1408 bytes from 10.0.0.1:	icmp_seq=20 ttl=64 time=0.057 ms

Tabela 4.5: Atrasos dos pings em modo adaptativo (rodada 1).

03:49:07:	1408 bytes from 10.0.0.1:	icmp_seq=3 ttl=64 time=0.063 ms
03:49:08:	1408 bytes from 10.0.0.1:	icmp_seq=4 ttl=64 time=0.080 ms
03:49:09:	1408 bytes from 10.0.0.1:	icmp_seq=5 ttl=64 time=0.071 ms
03:49:10:	1408 bytes from 10.0.0.1:	icmp_seq=6 ttl=64 time=0.079 ms
03:49:11:	**1408 bytes from 10.0.0.1:**	**icmp_seq=7 ttl=64 time=5.51 ms**
03:49:12:	1408 bytes from 10.0.0.1:	icmp_seq=8 ttl=64 time=0.087 ms
03:49:13:	1408 bytes from 10.0.0.1:	icmp_seq=9 ttl=64 time=0.082 ms
03:49:14:	1408 bytes from 10.0.0.1:	icmp_seq=10 ttl=64 time=0.057 ms

Tabela 4.6: Atrasos dos pings em modo adaptativo (rodada 2).

03:53:42:	1408 bytes from 10.0.0.1:	icmp_seq=5 ttl=64 time=0.043 ms
03:53:43:	1408 bytes from 10.0.0.1:	icmp_seq=6 ttl=64 time=0.053 ms
03:53:44:	1408 bytes from 10.0.0.1:	icmp_seq=7 ttl=64 time=0.055 ms
03:53:45:	1408 bytes from 10.0.0.1:	icmp_seq=8 ttl=64 time=0.061 ms
03:53:46:	**1408 bytes from 10.0.0.1:**	**icmp_seq=9 ttl=64 time=8.89 ms**
03:53:47:	1408 bytes from 10.0.0.1:	icmp_seq=10 ttl=64 time=0.060 ms
03:53:48:	1408 bytes from 10.0.0.1:	icmp_seq=11 ttl=64 time=0.039 ms
03:53:49:	1408 bytes from 10.0.0.1:	icmp_seq=12 ttl=64 time=0.059 ms
03:53:50:	1408 bytes from 10.0.0.1:	icmp_seq=13 ttl=64 time=0.090 ms

Tabela 4.7: Atrasos dos pings em modo adaptativo (rodada 3).

As tabelas 4.8, 4.9 e 4.10 exibem os tempos de *download* HTTP para as mesmas três repetições de um experimento, quando a rede se encontra apenas em modo reativo. Pode ser verificado que nos *timestamps* 03:16:24, 03:20:31 e 04:17:21, quando foi introduzida a falha no enlace S4 — S1, houve elevação considerável nos respectivos tempos de *download* durante o instante de consulta e resposta ao controlador, com instalação de novas regras de encaminhamento, de forma análoga ao efeito observado com os tempos de ping.

03:14:41	0,025185 segs
03:14:42	0,037738 segs
03:14:43	0,028440 segs
03:14:44	0,037185 segs
03:16:24	**2,201961 segs**
03:16:27	1,200846 segs
03:16:30	1,201022 segs

Tabela 4.8: Tempos de *download* no modo reativo (rodada 1).

03:18:10	0,047244 segs
03:18:11	0,032549 segs
03:18:12	0,028841 segs
03:18:13	0,023237 segs
03:20:31	**2,201803 segs**
03:20:34	1,201418 segs
03:20:37	1,201399 segs
03:20:40	1,201299 segs

Tabela 4.9: Tempos de *download* no modo reativo (rodada 2).

04:15:03	0,028275 segs
04:17:16	0,348012 segs
04:17:17	0,053784 segs
04:17:19	0,061084 segs
04:17:21	**1,277433 segs**
04:17:22	0,055114 segs
04:17:23	0,258169 segs
04:17:24	0,039922 segs
04:17:25	0,036477 segs

Tabela 4.10: Tempos de *download* no modo reativo (rodada 3).

As tabelas 4.11, 4.12 e 4.13 exibem os tempos de *download* HTTP para as mesmas três repetições do segundo experimento com ping, quando a rede se encontra com as regras de proteção instaladas. Pode ser verificado que nos *timestamps* 03:37:03, 03:49:11 e 03:53:46, quando foi introduzida a falha no enlace S4 — S1, diferente do que ocorreu no caso anterior, os tempos de *download* não se degradaram em função do redirecionamento para os grupos *fast failover*.

03:36:59	0,021392 segs
03:37:00	0,014033 segs
03:37:01	0,014698 segs
03:37:02	0,012352 segs
03:37:03	**0,013249 segs**
03:37:04	0,018218 segs
03:37:05	0,022530 segs
03:37:06	0,019368 segs
03:37:07	0,017569 segs

Tabela 4.11: Tempos de *download* no modo adaptativo (rodada 1).

03:49:08	0,022649 segs
03:49:09	0,022456 segs
03:49:10	0,023302 segs
03:49:11	**0,021731 segs**
03:49:12	0,022135 segs
03:49:13	0,017626 segs
03:49:14	0,018875 segs
03:49:15	0,023355 segs

Tabela 4.12: Tempos de *download* no modo adaptativo (rodada 2).

03:53:42	0,017334 segs
03:53:43	0,019036 segs
03:53:44	0,020245 segs
03:53:45	0,016043 segs
03:53:46	**0,016750 segs**
03:53:47	0,020015 segs
03:53:48	0,022274 segs
03:53:49	0,022458 segs
03:53:50	0,021096 segs

Tabela 4.13: Tempos de *download* no modo adaptativo (rodada 3).

Para verificação das ações do CWB e da migração de modo proativo para modo reativo, foi utilizado o cenário experimental da Figura 4.3 para capturar, via ferramenta Wireshark[3], a quantidade de pacotes do tipo PACKET-IN enviados pelos *switches* ao controlador, solicitando informações sobre qual regra de fluxo aplicar a um pacote entrante. Isso é possível através do filtro *"tcp.port == 6653 && openflow_v5.type == 10"*, aplicado ao Wireshark. Vale notar que os pacotes PACKET-IN também são usados pelos *switches* para encapsular mensagens do tipo LLDP[4], que são usadas pelo controlador para obter informações da topologia da rede.

Iniciada a rede, quando uma requisição HTTP é enviada pelo *host* 2, conectado no *switch* S4, por não haver nenhum regra de fluxo com uma ação a ser tomada nesse *switch*, uma requisição PACKET-IN é envida ao controlador solicitando uma instalação de regra para o encaminhamento do pacote contendo a requisição HTTP. Esse processo é repetido para cada nova transferência de arquivo via CURL caso a respectiva regra tenha sido removida por *timeout*.

A Figura 4.4 apresenta a taxa de pacotes PACKET-IN, ao longo do tempo, que chegam ao controlador, quantificados a cada 1 s. Esse processo de medição se iniciou aproximadamente às 14:45:20, justamente no instante que foi disparado

[3]https://www.wireshark.org/
[4]https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol

o processo de requisição HTTP no *host* 2. No momento 14:46:15, desativou-se a interface entre os *switches* S1 e S4 e, desta forma, a regra que indicava ao *switch* S4 encaminhar tráfego em direção ao servidor HTTP por esta interface se torna ineficaz. Nesse momento, o contador de *idle timeout* é iniciado. Para que este fluxo não seja descartado, o *switch* S4 envia uma solicitação ao controlador via PACKET-IN sobre qual novo caminho o pacote deve ser encaminhado. Por esse motivo, verifica-se o número de PACKET-INs aumentar no instante entre 14:46:15 e 14:46:20, pois o *idle timeout* padrão é de 5s.

Pouco antes do instante 14:47:00, reativou-se o enlace entre os *switches* S1 e S4, por isso verifica-se também um aumento de tráfego devido ao controlador trocar mensagens LLDPs com os *switches*, encapsuladas em pacotes PACKET-IN, para obter uma visão da nova topologia. Próximo ao instante 14:48:00, realizou-se a aplicação do CWB na rede e pode-se verificar que o número de PACKET-INs praticamente chegou a zero. Durante o período 14:48:00 e 14:49:20, realizou-se o mesmo processo de desativação do enlace entre os *switches* S1 e S4 e percebe-se que apenas uma pequena quantidade de mensagens LLDP são enviadas ao controlador.

Próximo ao instante 14:49:30, aplicou-se a mudança para o método reativo na rede, ou seja, informações sobre o encaminhamento de "novos" pacotes devem ser enviadas ao controlador. Pode-se então perceber que os *switches* voltam a consultar o controlador sobre regras de encaminhamento.

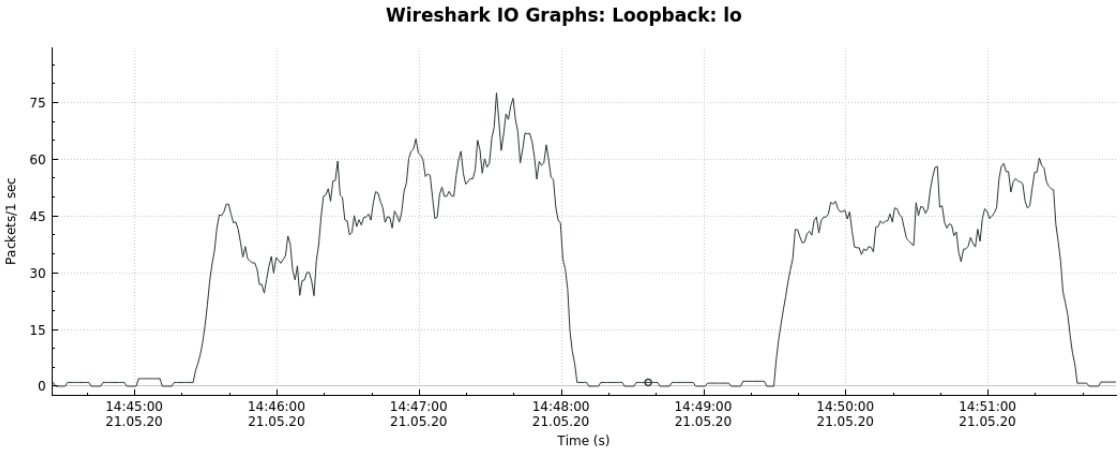


Figura 4.4: Validação da ação do CWB através do tráfego de PACKET-INs ao controlador.

Cabe esclarecer que, por uma questão de síntese de texto, optou-se pela não inclusão de experimentos ligados às ações de monitoramento, dado que elas têm caráter mais trivial. Também o disparo das ações decorrentes das medidas mo-

nitoradas ficaram ausentes dos experimentos, uma vez que ocorrem pelo simples cruzamento de limiares. Já os componentes que integram as ações de mitigação foram todos contemplados nos experimentos aqui realizados.

Para validar o processo de sumarização de regras do AFRC utilizaremos a topologia apresentada na Figura 4.5. Para gerar fluxos de pacotes, vamos utilizar o Iperf [5] para simular 10 conexões TCPs simultâneas. Usamos a sintaxe *iperf -s*, no lado do servidor, e, no lado do cliente, *iperf -c 10.0.0.1 -i1 -P10 -t100*. Todos os fluxos foram gerados por conexões iniciadas no *host 4* em direção ao servidor conectado pelo *switch S3*. Podemos observar que, para este tráfego de dados, foram geradas em cada um dos *switches* 41 regras de fluxos (11 Kbytes) com 9 campos, conforme exemplo apresentado no código 4.4 . Após aplicarmos o processo de sumarização do AFRC apresentado na subseção 3.5.1, reduzimos este número para apenas 2 regras de fluxo (380 bytes) em cada um dos *swicthes*, para este tráfego, conforme apresentado no código 4.5.

Código 4.4: Exemplo de regra de fluxo originalmente instalada.

```
cookie=0x20006d44000000, duration=5.037s, table=0, n_packets=0, n_bytes
    =0, idle_timeout=5, idle_age=5, priority=1,tcp,in_port=2,dl_src
    =00:00:00:00:00:02,dl_dst=00:00:00:00:00:01,nw_src=10.0.0.2,nw_dst
    =10.0.0.1,tp_src=37426,tp_dst=80,tcp_flags=syn actions=output:1
```

Código 4.5: Regras de fluxo sumarizadas.

```
in_port:3,eth_dst:00:00:00:00:00:01,eth_src:00:00:00:00:00:02,eth_type:0
    x800,ip_proto:0x6,ipv4_src:10.0.0.2,ipv4_dst:10.0.0.1,tcp_dst:80,
    actions:output=1
in_port:1,eth_dst:00:00:00:00:00:02,eth_src:00:00:00:00:00:01,eth_type:0
    x800,ip_proto:0x6,ipv4_src:10.0.0.1,ipv4_dst:10.0.0.2,tcp_src:80,
    actions:output=3
```

[5] https://iperf.fr/

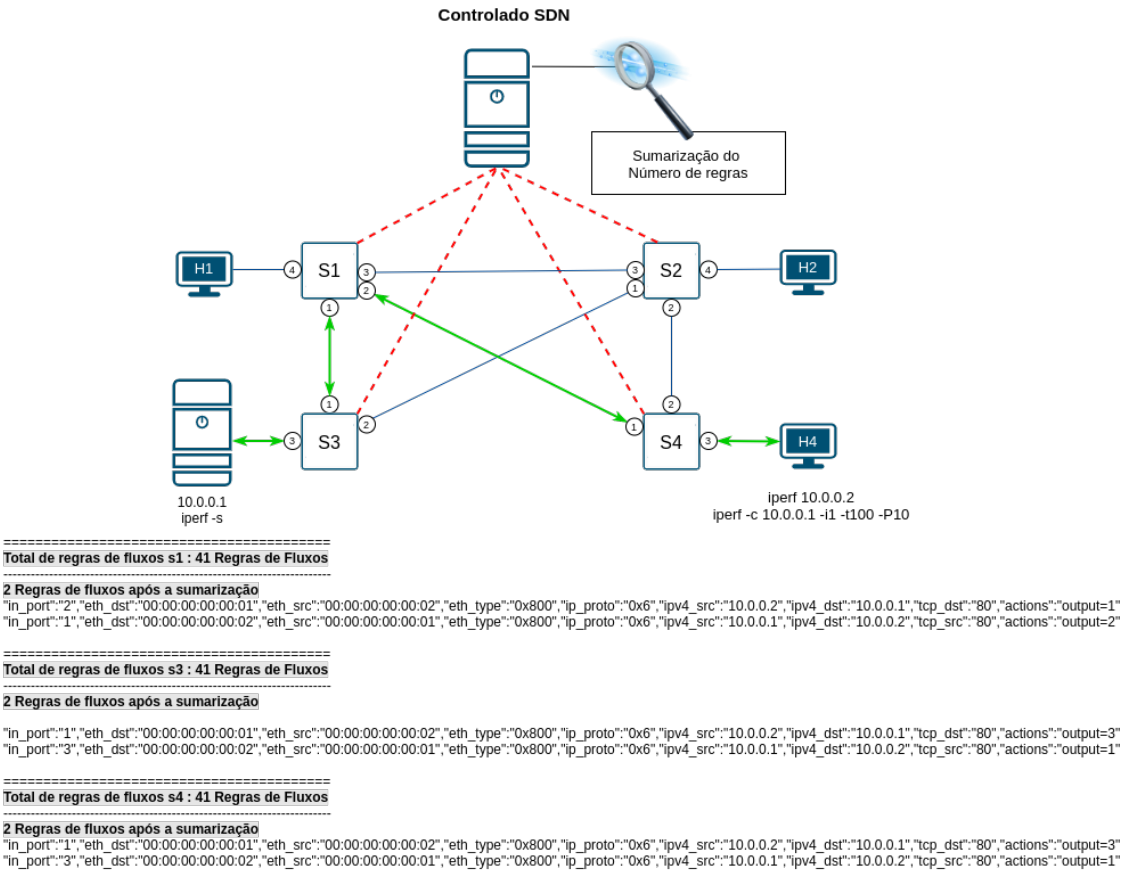


Figura 4.5: Topologia do exemplo para validação da sumarização de regras.

4.4 Discussão e Análises

Vale ressaltar que o cenário montado representa um exemplo reduzido de uso do AFRC para obter resiliência adaptativa do plano de dados, onde as regras instaladas conseguem proteger todos os caminhos primários selecionados pelo controlador. No entanto, para casos diferentes o plano de controle precisará computar todas as possibilidades de falha e instalar regras de caminhos primários e de proteção que evitem *loops* em caso de falha.

A Figura 4.6 mostra uma pequena variação da topologia do cenário experimental que pode facilmente resultar em *loops* a partir da instalação de caminhos primários e de proteção que tenham sido planejados de forma independente. A diferença entre as topologias está na retirada do enlace entre S1 e S2, que serve para fornecer redundância sempre que as interfaces 1 e 2 desses *switches* falhem. No caso de falha no enlace entre S1 e S3, por exemplo, a inexistência de enlace entre S1 e S2 pode levar à seguinte situação (abstraindo-se as trocas de mensagem para abertura de

conexão TCP): requisição HTTPS sai de S4 para S1 (caminho primário aprendido via controlador); em S1, pacote segue rota de *backup* para S4 e entra em *loop*. A única forma de prevenir essa situação, especificamente protegendo a conectividade entre cliente e servidor de uma falha no enlace entre S1 e S3, é forçando o caminho primário a partir de S4 ser sempre via S2, o que reflete um caso muito restrito.

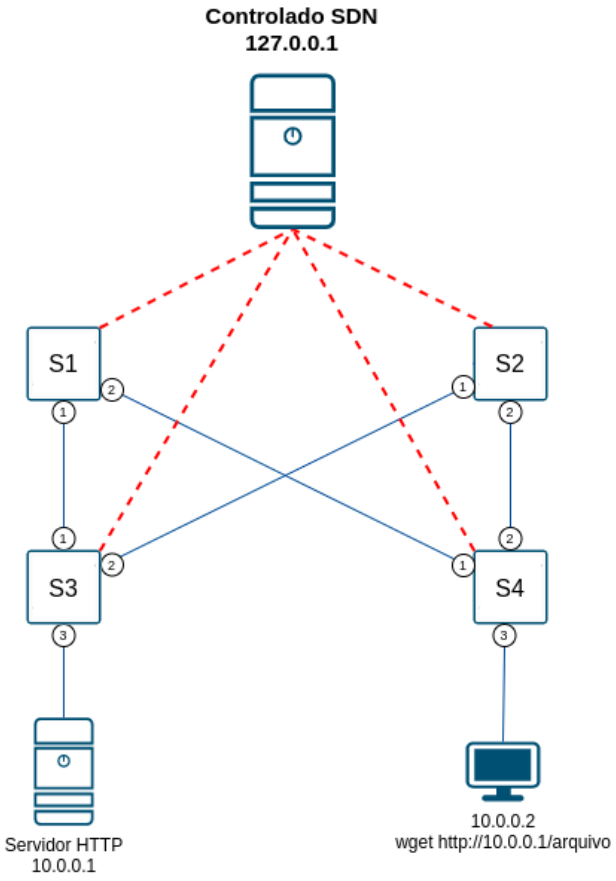


Figura 4.6: Variação do cenário experimental, passível de *loop* em caso de falha.

Ainda com relação a esse exemplo, uma estratégia mais genérica de recuperação proativa não seria possível, obrigando a adoção de uma abordagem reativa. Isso porque a queda de interface em um *switch* - por exemplo, interface 1 em S1 - obrigaria uma decisão de encaminhamento específica em outro *switch* - nesse caso, em S4, encaminhando primariamente para S2 - que não está interligado ao primeiro e, portanto, não consegue detectar a falha. Nesse caso, o controlador seria avisado e, reativamente, refaz os caminhos para a respectiva rede. Torna-se óbvio, portanto, que a abordagem proativa depende da topologia de rede usada.

Outro ponto que vale ser observado é quanto à quantidade de regras de *backup* que precisam ser usadas para proteger um único fluxo conforme o tamanho da rede. Trata-se do caso da abordagem proativa para fluxos sensíveis ou críticos que faz parte do AFRC. Tomando-se como exemplo a topologia *fat-tree* apresentada na Figura 4.7,

composta por 20 *switches* e oito *hosts*, incluindo o servidor HTTPS. É relativamente fácil verificar que a adoção de uma abordagem proativa para proteger apenas o acesso do cliente 10.0.0.8 ao servidor HTTP, em caso de falha única de qualquer enlace da rede demanda uma programação relativamente complexa a ser desempenhada pelo controlador. Conforme já mencionado no exemplo anterior, os caminhos primários que conduzem a esse *host* e ao servidor - que geralmente são computados a partir de um algoritmo de *spanning tree* e que, portanto, nem sempre consideram o menor caminho - devem ser considerados no planejamento dos caminhos de *backup* para se evitar *loops*. No caso deste exemplo, esse cuidado se faz importante quando o pacote atinge um dos *switches* de "primeiro nível" (S1, S2, S3 e S4), uma vez que eles possuem mais que duas opções de encaminhamento. Apenas para esse caso, regras de *backup* terão, portanto, que ser inseridas em S20, S12, S11, S19, S4, S3, S2, S1, S6, S5, S13 e S14 para garantir caminhos de ida e volta entre o cliente 10.0.0.8 e o servidor imunes a uma única falha de enlace. Caso se deseje proteger os acessos HTTPS de todos os clientes da rede, esse planejamento poderá requerer um conjunto expressivo de grupos *fast failover* a serem instalados nos *switches*. Portanto, a escalabilidade de qualquer estratégia proativa de recuperação de falha deve ser considerada conforme a rede a ser aplicada.

É importante ressaltar que a proposta e/ou implementação de uma solução algorítmica genérica para computação de rotas primárias e de proteção está fora do escopo deste trabalho. No entanto, problemas relacionados a excesso de regras sobrecarregando as memórias dos *switches*, em decorrência da instalação de regras de proteção, são contempladas no AFRC, que reage a esse problema migrando os *switches* para o modo reativo.

Com relação aos experimentos realizados e às economias de tempo verificadas após um evento de falha ao se usar o modo adaptativo do AFRC, deve-se ressaltar que se trata de um experimento reduzido para ser maior controle do experimento. em uma situação de rede real, em escala maior, com uma quantidade muito maior de fluxos trafegando pela rede, os tempos de recuperação mais elevados em função do modo reativo podem gerar impacto consideravelmente negativo aos serviços oferecidos tanto pelo provedor de conteúdo quanto ao provedor de transporte de dados.

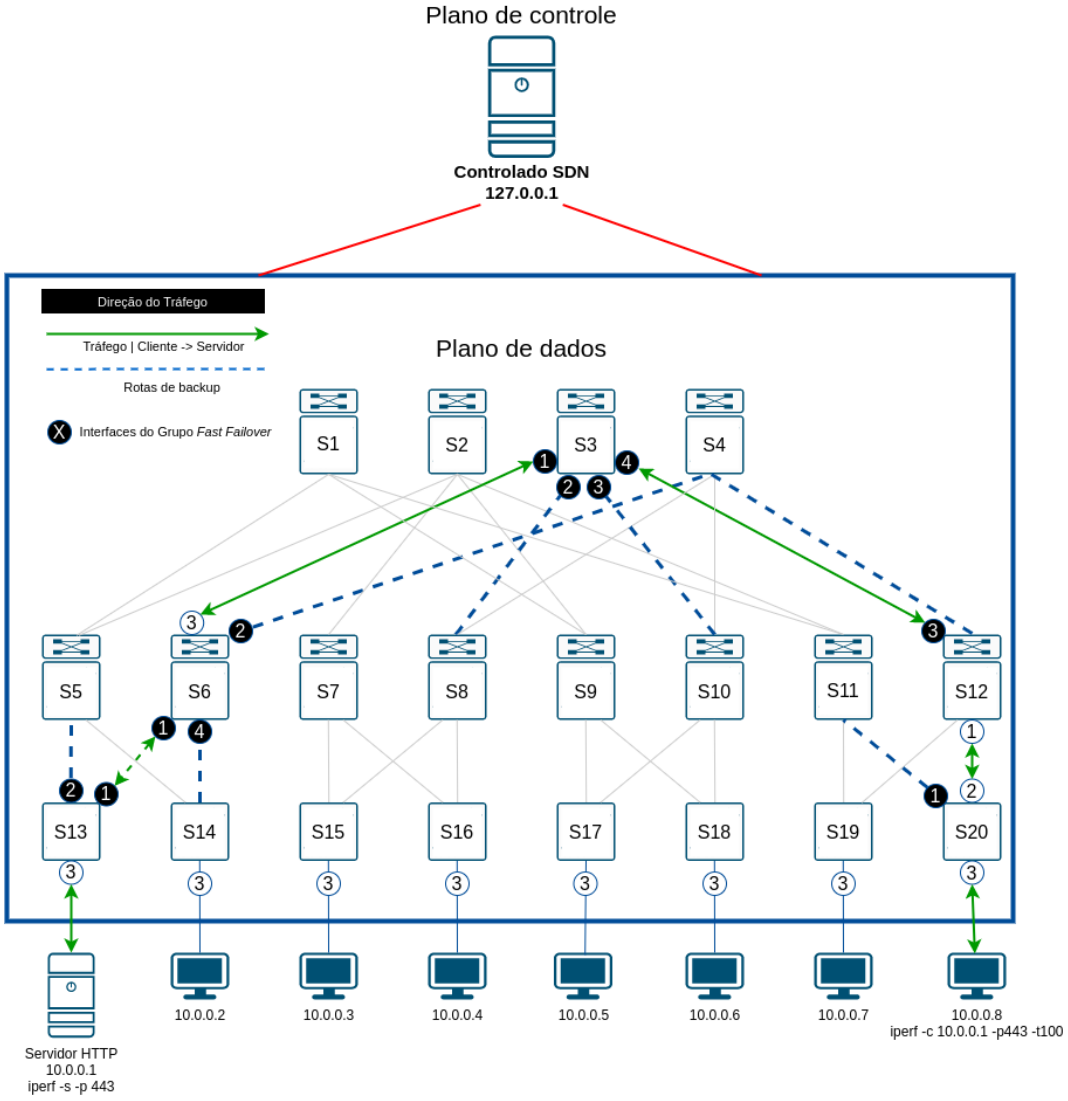


Figura 4.7: Topologia *fat-tree* para discussão sobre escalabilidade do AFRC.

5. Conclusão e Trabalhos Futuros

As redes definidas por *software*, ou SDN, permitem que o comportamento da rede seja mais flexível, bem como adaptável e gerenciável, conforme as necessidades de cada contexto de negócio. Nesse contexto de SDN e do protocolo OpenFlow, por se tratar de um padrão aberto, existem vários grupos propondo soluções diferente para diversos problemas. Com a dinamicidade com que as redes crescem e como as necessidades organizacionais são diferentes, apesar de já existirem vários projetos consolidados, tais como o ONOS[1] e o OpenDaykight[2], ainda há muito campo para pesquisa.

Em menos de uma década, mudamos de uma rede orientada a conexão para um modelo de rede orientado a aplicação. Enlaces redundantes com dupla abordagem e redundância de ativos, como *switches* e servidores, traziam segurança para muitos administradores de redes. Porém, com a mudança das regras de negócios, a forma com que nos divertimos, a forma como nos comunicamos e o acesso à Internet deixam de ser itens de luxo e passam a ser *commodities*. QoS *(Quality of Service)* é substituído pelo QOE *Quality of Experience*, pois a percepção do usuário passa a ser o ponto crucial para que um negocio seja bem sucedido. Em função disso, estudos sobre recuperação de falha em redes OpenFlow têm sido realizados no meio acadêmico, propondo e analisando o tempo de recuperação após falhas de enlaces ou de dispositivo, ou ainda dimensionando o número máximo de *switches* suportados por um único controlador. Também há estudos sobre a forma mais eficaz de sumarização de regras de fluxos, sem perder a semântica de encaminhamento, ou de ferir regras de *firewall*.

Pelas características de uma SDN e pelas conclusões dos diversos estudos na literatura, fica evidente que, do ponto de vista prático, torna-se importante estabe-

[1] https://www.opennetworking.org/onos/
[2] https://www.opendaylight.org/

lecer uma estratégia para recuperação de falha que leve em consideração o tempo de indisponibilidade até a efetiva recuperação. E, de forma paralela, também precisam ser consideradas a taxa de ocupação das memórias dos *switches*, em função das múltiplas regras nas tabelas de fluxo, e a sobrecarga no controlador, por conta do atendimento às diversas demandas da SDN.

Assim sendo, este trabalho propõe o *Adaptive Fault Recovery Control* - AFRC, uma abordagem adaptativa que busca usufruir dos mecanismos oferecidos pelo protocolo OpenFlow, no que tange aos modos de operação reativo e proativo, e a utilização dos grupos *fast failover*, que foram desenvolvidos para a proteção do plano de dados. O objetivo do AFRC é atuar como aliado do controlador SDN. Foi concebido não como uma API integrada ao controlador, mas como uma forma de NMS (*Network Management System*), que monitora o comportamento da rede e atua de forma ativa na tomada de decisões voltadas para a resiliência da rede.

Em termos mais específicos, o AFRC monitora: (i) a taxa de mensagens PACKET-IN ao controlador, pois uma taxa excessiva pode degradar a utilização da CPU do controlador, bem como congestionar a interface utilizada como canal de comunicação do plano de controle com o plano de dados; (ii) a quantidade de fluxos instalados nas memórias dos *switches*, pois caso esta se esgote, fluxos serão descartados; e (iii) a utilização de CPU do próprio controlador SDN, pois como os *switches* não estão habilitados a tomar decisões de encaminhamento, pacotes novos podem ser descartados caso o controlador perca sua capacidade de resposta em virtude de outros fatores.

A solução é composta das etapas de inicialização, investigação, monitoramento e mitigação. Na fase de inicialização é verificado se há um controlador sendo executado no host alvo, ou local, caso sim, a fase de investigação realiza o levantamento dos *switch* conectados no controlador. Feito isto a fase de monitoramento é iniciada. Nessa fase busca-se verificar o comportamento individual de cada *switch* e, caso ocorra algum evento que demande uma ação do AFRC para fins de garantia da resiliência da rede, a fase de mitigação e iniciada. Na fase de mitigação, as seguintes ações podem ser combinadas, conforme a situação: (i) instalação de grupos *fast failover* para uma recuperação proativa de falhas que atenda apenas a fluxos considerados críticos; (ii) sumarização de regras OpenFlow, como forma de diminuir a ocupação das memórias dos *switches*; (iii) migração parcial ou generalizada da rede para o modo de recuperação proativo; (iv) migração parcial ou generalizada da rede para o modo de recuperação reativo.

Para validar esta proposta e analisar sua eficácia, foram realizados experimentos usando tráfego HTTP em uma topologia simplificada de *data center* do tipo *spine and leaf*. Uma rede emulada no Mininet foi implementada de maneira a se verificar a capacidade do AFRC de executar cada uma de suas ações de mitigação. O desempenho da rede e de serviços de transferências de arquivos foram medidos através de tempos de *download* e de reposta a *pings*, antes e após a introdução de falhas de enlace, de maneira a verificar a eficácia do AFRC para as situações onde foi prevista sua atuação. Os resultados e as análises realizadas concluem que a ações de mitigação propostas pelo AFRC atingem seu objetivo principal, que é manter o plano de dados resiliente.

Como trabalhos futuros, tem-se a intenção de desenvolver um módulo que seja capaz de migrar regras de fluxo para outros *switches* da rede, caso através das ações de mitigação não seja possível diminuir significativamente o número de fluxos instalados em um dado *switch*. Além disso, tem-se também a intenção de que, através do conhecimento a respeito dos fluxos trafegados pela rede, seja possível prever quais interfaces são melhor candidatas a serem protegidas por grupos *fast failover*.

Referências Bibliográficas

[1] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[2] A. S. A. Furqan Alam, Iyad Katib, "Open networking foundation. software-defined networking: The new norm for networks. 2013," *Computer Science and Software Engineering*, vol. 3, 2013.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] C.-Y. Chu, K. Xi, M. Luo, and H. J. Chao, "Congestion-aware single link failure recovery in hybrid sdn networks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1086–1094, IEEE, 2015.

[5] R. Odarchenko, D. Serhii, O. Oksiuk, and L. Dakova, "Software-controlled network sdn reliability calculation," in *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)*, pp. 99–103, IEEE, 2018.

[6] R.-H. Hwang and Y.-C. Tang, "Fast failover mechanism for sdn-enabled data centers," in *2016 International Computer Symposium (ICS)*, pp. 171–176, IEEE, 2016.

[7] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[8] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an mpls transport profile," 2009.

[9] D. Adami, S. Giordano, M. Pagano, and N. Santinelli, "Class-based traffic recovery with load balancing in software-defined networks," in *2014 IEEE Globecom Workshops (GC Wkshps)*, pp. 161–165, IEEE, 2014.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Enabling fast failure recovery in openflow networks," in *2011 8th International Workshop on the Design of Reliable Communication Networks (DRCN)*, pp. 164–171, IEEE, 2011.

[11] B. Yamansavascilar, A. C. Baktir, A. Ozgovde, and C. Ersoy, "Fault tolerance in sdn data plane considering network and application based metrics," *ArXiv*, vol. abs/1912.11849, 2019.

[12] W. Xin-gang, "A link performance-based failure recovery approach in sdn data plane," in *Proceedings of the 3rd International Conference on Multimedia and Image Processing*, pp. 46–51, 2018.

[13] F. Mazzola, D. Marcon, M. Neves, and M. Barcellos, "Tá na hora: analisando a latência de modificação de tabelas de fluxo em arquiteturas de switches sdn," in *Anais do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBC, 2018.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, pp. 254–265, 2011.

[15] K. Kannan and S. Banerjee, "Compact tcam: Flow entry compaction in tcam for power aware sdn," in *International conference on distributed computing and networking*, pp. 439–444, Springer, 2013.

[16] C. Wei, R. Buffone, and R. Stata, "System and method for website performance optimization and internet traffic processing," Feb. 7 2012. US Patent 8,112,471.

[17] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 490–500, 2009.

[18] Y. Zhang, S. Natarajan, X. Huang, N. Beheshti, and R. Manghirmalani, "A compressive method for maintaining forwarding states in sdn controller," in

Proceedings of the third workshop on Hot topics in software defined networking, pp. 139–144, 2014.

[19] Y. Liu, S. O. Amin, and L. Wang, "Efficient fib caching using minimal non-overlapping prefixes," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 14–21, 2013.

[20] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data centers," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 157–170, 2013.

[21] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[22] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[23] W. J. A. Silva, "Performance evaluation of flow creation inside an openflow network," *Proceedings of the XXXV Simpósio Brasileiro de Telecomunicações e Processamento de Sinais-SBrT2017, São Pedro, SP, Brazil*, pp. 3–6, 2017.

[24] M. Santos, N. Agoulmine, E. Rachkidy, and S. Fernandes, "Revisitando o problema de alocação de controladores sdn: Uma análise sobre o impacto do custo de recobrimento da rede," in *Anais do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBC, 2018.

[25] "Floodlight controlador sdn." `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343647/Floodlight+Projects`. 20/03/2020.

[26] S. Shenker, M. Casado, T. Koponen, N. McKeown, *et al.*, "The future of networking, and the past of protocols," *Open Networking Summit*, vol. 20, pp. 1–30, 2011.

[27] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[28] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[29] "Pox: A python-based openflow controller." `http://www.noxrepo.org`. 20/03/2020.

[30] "Ryu é uma estrutura de rede definida por software baseada em componentes." `http:/http://osrg.github.io/ryu/`. 20/03/2020.

[31] "Opendaylight controller." `http://www.opendaylight.org/`. 20/03/2020.

[32] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer.," in *Hotnets*, 2009.

[33] A. Liberato, D. Mafioletti, E. Spalla, M. Martinello, and R. Villaca, "Avaliação de desempenho de plataformas para validação de redes definidas por software," in *Wperfomance-XIII Workshop em Desempenho de Sistemas Computacionais e de Comunicaç ao*, pp. 1905–1918, 2014.

[34] Internet, ""cpqd, f. (2014). openflow 1.3 software switch.," *Website. https://github. com/CPqD/ofsoftswitch13.*, vol. 38, no. 3, pp. 105–110, 2008.

[35] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov, "Designing extensible ip router software," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 189–202, USENIX Association, 2005.

[36] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Automatic bootstrapping of openflow networks," in *2013 19th IEEE Workshop on Local & Metropolitan Area Networks (LANMAN)*, pp. 1–6, IEEE, 2013.

[37] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, p. 132, 2009.

[38] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in openflow-based software defined networks," *Computer Communications*, vol. 77, pp. 52–61, 2016.

[39] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, 2010.

[40] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu, "A first look at inter-data center traffic characteristics via yahoo! datasets," in *2011 Proceedings IEEE INFOCOM*, pp. 1620–1628, IEEE, 2011.

[41] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[42] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *Proceedings of the 11th ACM Workshop on Hot Topics in networks*, pp. 61–66, 2012.

[43] R. Wang, D. Butnariu, J. Rexford, *et al.*, "Openflow-based server load balancing gone wild.," *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[44] "Openflow switch specification 1.1." `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.1.0.pdf`. 08/05/2020.

[45] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pp. 75–86, 2008.

[46] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pp. 63–74, 2009.

[47] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[48] J. Chen, J. Chen, F. Xu, M. Yin, and W. Zhang, "When software defined networks meet fault tolerance: A survey," in *International conference on algorithms and architectures for parallel processing*, pp. 351–368, Springer, 2015.

[49] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *2014 Third European Workshop on Software Defined Networks*, pp. 61–66, IEEE, 2014.

[50] S. S. Lee, K.-Y. Li, K. Y. Chan, G.-H. Lai, and Y. C. Chung, "Software-based fast failure recovery for resilient openflow networks," in *2015 7th International Workshop on Reliable Networks Design and Modeling (RNDM)*, pp. 194–200, IEEE, 2015.

[51] S. S. Lee, K.-Y. Li, K.-Y. Chan, G.-H. Lai, and Y.-C. Chung, "Path layout planning and software based fast failure detection in survivable openflow networks," in *2014 10th International Conference on the Design of Reliable Communication Networks (DRCN)*, pp. 1–8, IEEE, 2014.

[52] D. Kotani and Y. Okabe, "Fast failure detection of openflow channels," in *Proceedings of the Asian Internet Engineering Conference*, pp. 32–39, 2015.

[53] R. Kanagavelu, L. N. Mingjie, K. M. Mi, F. B.-S. Lee, *et al.*, "Openflow based control for re-routing with differentiated flows in data center networks," in *2012 18th IEEE International Conference on Networks (ICON)*, pp. 228–233, IEEE, 2012.

[54] S. A. Astaneh and S. S. Heydari, "Optimization of sdn flow operations in multi-failure restoration scenarios," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 421–432, 2016.

[55] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Fast failure recovery for in-band openflow networks," in *2013 9th international conference on the Design of reliable communication networks (drcn)*, pp. 52–59, IEEE, 2013.

[56] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Openflow-based segment protection in ethernet networks," *Journal of Optical Communications and Networking*, vol. 5, no. 9, pp. 1066–1075, 2013.

[57] N. Sahri and K. Okamura, "Fast failover mechanism for software defined networking: Openflow based," in *Proceedings of The Ninth International Conference on Future Internet Technologies*, pp. 1–2, 2014.

[58] A. Capone, C. Cascone, A. Q. Nguyen, and B. Sanso, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *2015 11th international conference on the design of reliable communication networks (DRCN)*, pp. 25–32, IEEE, 2015.

[59] F. Giroire, J. Moulierac, and T. K. Phan, "Optimizing rule placement in software-defined networks for energy-aware routing," in *2014 IEEE Global Communications Conference*, pp. 2523–2529, IEEE, 2014.

[60] M. Rifai, N. Huin, C. Caillouet, F. Giroire, D. Lopez-Pacheco, J. Moulierac, and G. Urvoy-Keller, "Too many sdn rules? compress them with minnie," in *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, 2015.

[61] M. Lu, W. Deng, and Y. Shi, "Tf-idletimeout: Improving efficiency of tcam in sdn by dynamically adjusting flow entry lifecycle," in *2016 IEEE International*

Conference on Systems, Man, and Cybernetics (SMC), pp. 002681–002686, IEEE, 2016.

[62] H. Zhu, H. Fan, X. Luo, and Y. Jin, "Intelligent timeout master: Dynamic timeout for sdn-based data centers," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 734–737, IEEE, 2015.

[63] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in openflow networks," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 177–188, 2014.

[64] M. P. Fernandez, "Comparing openflow controller paradigms scalability: Reactive and proactive," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pp. 1009–1016, IEEE, 2013.

[65] A. Gangwal, M. Conti, and M. S. Gaur, "Panorama: Real-time bird's eye view of an openflow network," in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, pp. 204–209, IEEE, 2017.

[66] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 19–24, 2012.

[67] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 7–12, 2013.

[68] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, "Dynamic controller provisioning in software defined networks," in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pp. 18–25, IEEE, 2013.

[69] T. Wang, F. Liu, J. Guo, and H. Xu, "Dynamic sdn controller assignment in data center networks: Stable matching with transfers," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[70] D. Merling, W. Braun, and M. Menth, "Efficient data plane protection for sdn," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, pp. 10–18, IEEE, 2018.

[71] H. Yu, K. Li, and H. Qi, "An active controller selection scheme for minimizing packet-in processing latency in sdn," *Security and Communication Networks*, vol. 2019, 2019.

[72] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[73] J. H. Yang, "Method and apparatus for reducing firewall rules," Feb. 16 2010. US Patent 7,665,128.

[74] M. Yoon, S. Chen, and Z. Zhang, "Reducing the size of rule set in a firewall," in *2007 IEEE International Conference on Communications*, pp. 1274–1279, IEEE, 2007.

[75] C. B. Martins, T. A. S. Pardo, A. P. Espina, and L. H. M. Rino, "Introdução à sumarização automática," *Relatório Técnico RT-DC*, vol. 2, no. 1, p. 35, 2001.

[76] "Mininet - rede virtual realista." `http://www.mininet.org/`. 20/03/2020.