Universidade do Minho
Escola de Engenharia

Anthony Jonathan da Silva Leal

Desenvolvimento de Mecanismos de Engenharia de Tráfego em Data Centers através de SDN

abril de 2016

Anthony Jonathan da Silva Leal **Desenvolvimento de Mecanismos de Engenharia de Tráfego em Data Centers através de SDN**

UMinho | 2016

Universidade do Minho
Escola de Engenharia

Anthony Jonathan da Silva Leal

Desenvolvimento de Mecanismos de Engenharia de Tráfego em Data Centers através de SDN

Dissertação de Mestrado
Mestrado Integrado em Engenharia de Telecomunicações e Informática

Trabalho efetuado sob a orientação do
Professor Doutor Pedro Nuno Miranda de Sousa

abril de 2016

AGRADECIMENTOS

Esta dissertação resultou de um longo trabalho e dedicação. Contudo, não seria possível sem a ajuda e conselhos que me foram dados durante todo o trabalho.

Em primeiro lugar, quero agradecer a disponibilidade e orientação académica do Professor Doutor Pedro Nuno Sousa, e também agradecer ao Vítor Sá Pereira pelo apoio e disponibilidade.

Por último, e não menos importante, quero agradecer à minha família, em especial aos meus pais Armindo Leal e Maria Joaquina Leal, ao meu irmão David e à minha namorada, Rita, por todo o apoio que me deram ao longo deste percurso, em especial nos momentos em que se esmoreciam os ânimos.

RESUMO

O crescente uso de aplicações que geram altos volumes de tráfego motivou o desenvolvimento de novas abordagens de Engenharia de Tráfego que pudessem melhorar o desempenho e eficiência das infraestruturas de comunicação, e.g. redes dos ISPs (*Internet Service Providers*), *Data Centers*, etc. Neste contexto, a área denominada por *Software Defined Networking* (SDN) poderá ser útil para a definição de alguns mecanismos inovadores nestes cenários. Este paradigma, que tem sido recentemente explorado, oferece novas tecnologias e protocolos proporcionando novas oportunidades para uma gestão mais expedita e eficiente das infraestruturas de rede.

Este trabalho propõe-se contribuir para o desenvolvimento de mecanismos de Engenharia de Tráfego na área das SDN. Os mecanismos a estudar estarão orientados para tarefas de balanceamento de carga em redes de *Data Centers* e implementados com a ferramenta de emulação *Mininet*. Para tal, será feito inicialmente um estudo das diversas arquiteturas de redes de *Data Centers*, dos conceitos que englobam o paradigma SDN e uma análise das estratégias de balanceamento de carga já existentes. De seguida será desenvolvida uma bancada de testes e implementados alguns mecanismos de balanceamento de carga. Posteriormente, serão efetuados testes de desempenho aos mecanismos desenvolvidos.

Palavras-Chave: *Software Defined Networking*, *Data Centers*, Engenharia de Tráfego, Balanceameto de Carga, *Mininet*

ABSTRACT

The increasing use of applications that generate high traffic volumes prompted the development of new approaches to Traffic Engineering field that could improve the performance and efficiency of communication infrastructures, e.g. ISPs (Internet Service Providers) networks, Data Centers, etc. In this context, arises the area of Software Defined Networking (SDN) that may be helpful to define some innovative mechanisms in such scenarios. This paradigm, which has recently been explored, offers new technologies and protocols providing new opportunities for a more expeditious and efficient traffic management strategies in network infrastructures.

This work aims to contribute to the development of Traffic Engineering mechanisms in the area of SDN. The mechanisms to develop will be oriented to load balancing tasks in Data Centers networks and implemented with the Mininet emulation tool. It will be made initially a study of the various Data Center networks architectures, concepts that encompass the SDN paradigm and an analysis of existing load balancing strategies. Then it will developed a test bench and implemented some load balancing mechanisms. Subsequently, performance tests will be made to the developed mechanisms.

KEYWORDS: Software Defined Networking, Data Centers, Traffic Engineering, Load Balancing, Mininet

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÓNIMOS

ACL	Access Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
CPU	Central Processing Unit
ECMP	Equal Cost Multipath
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MPLS	Multi Protocol Label Switching
NIB	Network Information Base
NIC	Network Interface Card
NVP	Network Virtualization Platform
REST	Representational State Transfer
SDN	Software Defined Networking
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
WLAN	Wireless Local Area Network

1. INTRODUÇÃO

1.1 Enquadramento e Motivação

Nos últimos anos, a computação intensiva de dados e de alto desempenho tem vindo a aumentar de forma abrupta, resultando na pesquisa e exploração de novas arquiteturas de rede bem como no crescimento dos centros de dados[1]. Com este crescimento tornou-se fundamental para as grandes empresas, aumentar o desempenho, a escalabilidade e a eficiência, bem como limitar o custo destas infraestruturas. O desempenho de um centro de dados pode ser caracterizado por um conjunto de métricas nomeadamente a largura de banda, latência, confiabilidade, produtividade, custo e consumo de energia, estando algumas delas relacionadas [1].

Os mecanismos de Engenharia de Tráfego são utilizados para otimizar o desempenho da rede através da análise dinâmica e preventiva do comportamento do tráfego transmitido. Alguns destes mecanismos têm sido explorados ao longo dos anos através das redes ATM (*Asynchronous Transfer Mode)* e posteriormente pelas redes IP/MPLS (*Internet Protocol/Multi Protocol Label Switching*). No entanto as suas implementações não se apresentam adequadas às redes futuras devido ao crescimento de conceitos tais como a computação na nuvem, e portanto, da necessidade de uma gestão mais apropriada da rede capaz de aperfeiçoar a utilização dos recursos para um melhor desempenho. De igual forma muitas das aplicações *online* requerem uma arquitetura de rede com capacidade de reagir em tempo real e ser escalável ao aumento de tráfego [2].

Face a este avanço surgiu nos últimos anos a abordagem SDN (*Software Defined Network*), cujo conceito tem evoluído desde 1996, tendo como principal objetivo, fornecer uma gestão controlada do encaminhamento nos nós da rede. Grupos de investigação tais como *Ipsilon, The Tempest*, IETF (*Internet Engineering Task Force*) e mais recentemente a arquitetura de gestão *Ethane* e o protocolo de comunicação *OpenFlow* [3] contribuíram para o desenvolvimento deste paradigma [4]. A abordagem SDN separa o plano de controlo do plano de encaminhamento de dados e é constituído por dois componentes: o controlador SDN e pelo dispositivo de encaminhamento SDN. O primeiro caracteriza-se como uma função lógica centralizada sendo

[1] A partir deste ponto o documento adota este termo para tradução do termo técnico Data Centers.

responsável por exemplo pela seleção das rotas por onde os fluxos de tráfego serão enviados, ou seja, toda a política de balanceamento de carga poderá residir neste componente. O segundo constitui o plano de dados da rede e aborda toda a transmissão dos pacotes. Estes dois componentes comunicam através da interface normalizada denominada por *OpenFlow* [5]. O *OpenFlow* permite, entre outras funcionalidades, ao administrador da rede dividir e/ou controlar o fluxo de dados na rede através da gestão centralizada das tabelas de *switching*/*routing* dos *switches* Ethernet [6].

Estes cenários podem ser emulados e simulados com a ferramenta *Mininet* [7], pois esta permite elaborar protótipos de redes de larga escala num único computador. O *Mininet* foi projetado tendo em consideração um conjunto de características: flexibilidade, ou seja, novas topologias e funcionalidades podem ser definidas no *software* utilizando linguagens de programação e sistemas operativos comuns; aplicabilidade, isto é, as implementações feitas em protótipos podem ser realizadas em redes reais baseadas em *hardware* sem alteração do código fonte; interatividade, permitindo a gestão e o funcionamento da rede simulada ocorrer em tempo real; escalabilidade, ou seja, é possível simular redes com centenas ou milhares de *switches* num único computador; compartilhável, permitindo a partilha e consequente alteração dos protótipos elaborados [8].

O estudo detalhado de todas as tecnologias anteriormente mencionadas, bem como dos diversos cenários da sua utilização, contribuirá para o desenvolvimento, implementação e teste de novos mecanismos de balanceamento de tráfego que visam melhorar o desempenho, a escalabilidade e a eficiência de redes de centros de dados.

1.2 Objetivos

Este trabalho tem como principal objetivo o estudo, desenvolvimento, e a análise de desempenho de estratégias de Engenharia de Tráfego vocacionadas para tarefas de balanceamento de carga em redes de comunicações. Os cenários em estudo serão centrados em infraestruturas de rede de centros de dados tirando também partido das recentes tecnologias e protocolos da área das SDN.

Neste sentido, para alcançar o objetivo anteriormente mencionado, definem-se como objetivos parcelares deste trabalho os seguintes pontos:

- Investigação e levantamento inicial bibliográfico sobre as várias áreas que são abrangidas neste trabalho (e.g. arquiteturas e funcionamento de centro de dados, tecnologia SDN, estratégias de balanceamento de carga, etc.).
- Análise e seleção de mecanismo(s) específico(s) de balanceamento de carga que serão implementadas e sujeitos posteriormente a testes de análise de desempenho em ambientes emulados.
- Familiarização com ferramentas de emulação de ambientes SDN (e.g. *Mininet*) e estudo de distintos controladores existentes (e.g. *POX*, *Pyretic*, *floodlight*).
- Implementação e teste de desempenho do(s) mecanismo(s) selecionados em ambiente experimental. Parametrização do(s) mecanismo(s), recolha e análise de resultados sobre o comportamento dos mesmos.

1.3 Sumário das Principais Contribuições

A principal contribuição deste trabalho é a implementação e estudo de três mecanismos de engenharia de tráfego que permitem realizar o balanceamento de carga em redes SDN. Para tal efeito, foi desenvolvida uma bancada de testes com o objetivo de simular/emular uma rede de centro de dados por forma a serem testados esses mecanismos. No final, foi feita a avaliação dos mecanismos utilizando como métrica a largura de banda de bissecção utilizada nas ligações.

1.4 Estrutura da Dissertação

Este documento está organizado em 5 capítulos. A descrição de cada um dos capítulos é a seguinte:

- Introdução: este capítulo fez o enquadramento e a contextualização do trabalho, ao apresentar o tema geral e os objetivos do trabalho a desenvolver.
- Estado da Arte: aborda a base teórica que irá permitir a especificação e a implementação de mecanismos de balanceamento de tráfego em redes SDN. É apresentada a arquitetura SDN, alguns mecanismos de engenharia de tráfego relacionados com esta área, bem como algumas arquiteturas e topologias de rede de centros de dados e as métricas que as distinguem.

- Especificação e Desenvolvimento da Bancada de Testes: aborda todo o trabalho desenvolvido, nomeadamente a bancada de testes, a topologia de rede e os mecanismos de balanceamento de carga desenvolvidos.
- Testes e Resultados: demonstra o ambiente de testes que foi usado, a metodologia de testes usada e a avaliação dos resultados obtidos. O objetivo é analisar a largura de banda de bissecção obtida através dos mecanismos.
- Conclusões: neste último capítulo são apresentadas as principais conclusões obtidas de todo o trabalho. É feita uma análise de todos os capítulos bem como da principal contribuição desta dissertação. Por último, é feita uma análise de futuros desenvolvimentos para melhorar a solução.

2. ESTADO DA ARTE

Neste capítulo iremos abordar o paradigma das SDN e a sua arquitetura (secção 2.1). Posteriormente serão apresentados alguns mecanismos de engenharia de tráfego relacionados com esta área, nomeadamente, o balanceamento de carga, a tolerância a falhas, a atualização das políticas de encaminhamento e análise de tráfego (secção 2.2). Por fim, as arquiteturas e topologias (secção 2.3), bem como as métricas comparativas dos centros de dados (secção 2.4), serão também abordadas.

2.1 Arquitetura das SDN

O paradigma das SDN refere-se a uma arquitetura de rede que separa o plano de controlo do plano de encaminhamento e que fornece aplicações de utilizador com uma visão centralizada sobre os estados distribuídos da rede. Este paradigma surgiu com o intuito de melhorar a utilização dos recursos da rede, simplificar a sua gestão, reduzir os custos operacionais e promover a inovação das redes atuais e futuras. Como se pode verificar na Figura 2.1 a arquitetura das SDN pode ser dividida em três planos, nomeadamente o de dados, de controlo e de gestão. O primeiro é constituído pela infraestrutura de rede e por uma interface denominada por *Southbound Interface*. O plano de controlo é organizado em três camadas, nomeadamente, o *Hypervisor* da rede, pelo(s) controlador(es) da rede e pela *Northbound Interface*. Por fim o plano de gestão está estruturado em três camada, das quais as linguagens e abordagens utilizadas na virtualização, as linguagens de programação e as aplicações de rede. Com o objetivo de expor mais detalhadamente esta arquitetura, esta será aprofundada segundo as suas camadas nas secções seguintes.

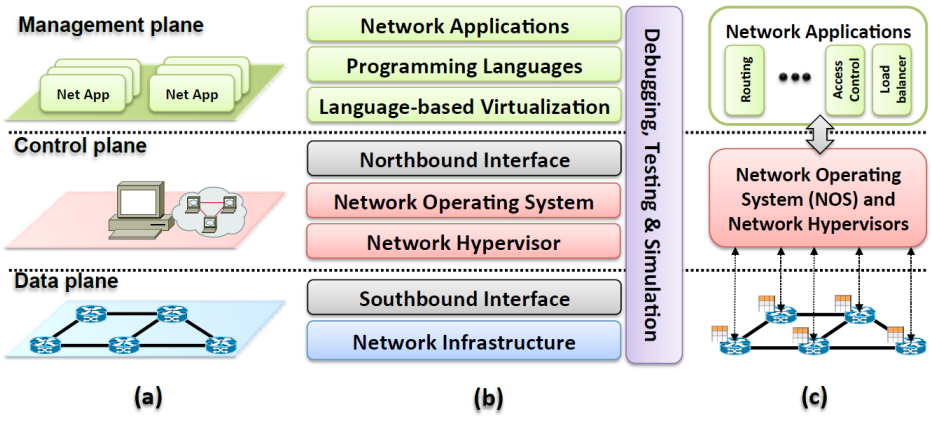

Figura 2.1: Arquitetura SDN dividida em a) planos b) camadas e c) design da arquitetura [9]

2.1.1 Plano de Dados

Estrutura de rede

Numa arquitetura SDN a infraestrutura representa um conjunto de equipamentos, designados por dispositivos de encaminhamento, podendo estes serem *switches*, *routers* e/ou *middleboxes*. Diferente dos tradicionais dispositivos incorporados com sistema de controlo permitindo-os tomarem decisões de forma autónoma, estes apenas procedem ao encaminhamento de pacotes. Um dispositivo de encaminhamento habilitado com o protocolo *OpenFlow* consiste numa ou várias tabelas de fluxo, num canal *OpenFlow* com ligação ao controlador (do plano de controlo) e num protocolo (protocolo *OpenFlow*) que permite ao controlador adicionar, atualizar ou remover entradas de fluxo. Uma tabela de fluxo de um dispositivo de encaminhamento *OpenFlow* é constituída por um conjunto de entradas, estando cada uma constituída por três campos, nomeadamente, um campo *match* definido por um conjunto de informações (ver Figura 2.2), um campo com a ação associada ao pacote e um contador utilizado para estatística. O processo de *matching* é iniciado na primeira tabela e pode continuar pelas tabelas seguintes. Se não existir uma entrada que faça *matching* com o pacote a ser tratado, este pode ser descartado, ser reencaminhado para o controlador ou enviado para a tabela seguinte. Este processo repete-se até o pacote percorrer a última tabela. Caso contrário, ou seja, quando existe *matching* do pacote com uma entrada de uma tabela, o pacote pode ser 1) encaminhado para uma determinada interface de rede de saída, 2) encapsulado e reencaminhado para o controlador, 3) descartado ou 4) enviado para a próxima tabela de fluxos através do processo de *pipeline*.

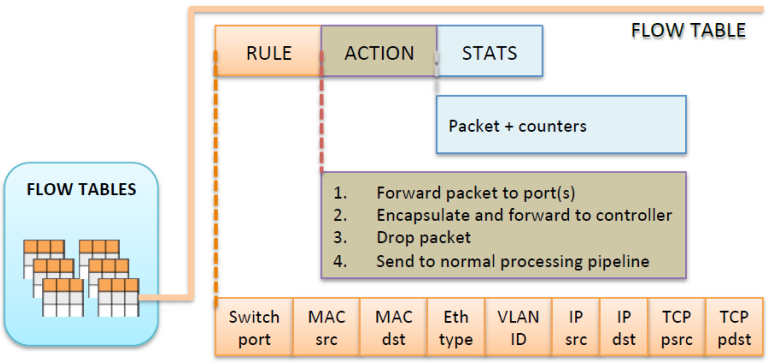

Figura 2.2: Constituição das entradas de fluxo [9]

Interface *Southbound*

Sobrejacente à infraestrutura física ou virtual dos dispositivos de encaminhamento, a interface *Southbound* refere-se ao meio de comunicação entre os elementos de controlo e de

encaminhamento dos planos de controlo e de dados, respetivamente. A interface *Southbound* mais aceite e amplamente implementada nas SDN é a interface/protocolo *OpenFlow*. Através da sua especificação, o protocolo *OpenFlow* permite a configuração dos dispositivos de encaminhamento e a implementação de um canal de comunicação entre esses mesmos dispositivos e o(s) controlador(es) do plano de controlo. Este canal é utilizado para as seguintes situações:

- Quando existem mudanças nas ligações e/ou nas interfaces de rede. Nesta situação o dispositivo de encaminhamento envia para o controlador uma mensagem a indicar esse evento.
- Para o envio de estatísticas, geradas pelos dispositivos de encaminhamento e colecionadas pelo controlador, acerca dos fluxos da rede.
- Quando um novo fluxo é iniciado na rede e este é desconhecido dos dispositivos de encaminhamento. Nesta situação o primeiro pacote desse fluxo é enviado ao controlador.
- Quando um dispositivo de encaminhamento tem como ação o envio de um pacote diretamente para o controlador.

Concluindo, este canal bem como as informações que nele circulam, permite ao controlador obter um certo nível de informação acerca dos fluxos e realizar a distribuição correta do tráfego na rede.

2.1.2 Plano de Controlo

***Hypervisors* da rede**

O *Hypervisor* da rede permite a partilha de recursos de *hardware* de máquinas virtuais distintas. Como é sabido, uma das características das tecnologias de virtualização é a possibilidade de migrar facilmente máquinas virtuais de um servidor físico para outro, permitindo uma gestão mais fácil e flexível dos serviços. No entanto, para que a virtualização seja eficiente, a infraestrutura de rede deve ser capaz de:

- Suportar uma topologia de rede arbitrária e esquemas de endereçamento;
- Permitir a cada *tenant* a habilidade de configurar os nós e a rede em simultâneo;
- A migração de *host* deve acionar automaticamente a migração das interfaces de rede virtual correspondente.

Nas SDN, *hypervisors* de rede como *FlowVisor* [10], *OpenVirteX* [11], *AutoSlice* [12], entre outros, permitem a múltiplas redes virtuais partilharem a mesma estrutura da rede física através de mecanismos de divisão da mesma. O *FlowVisor*, por exemplo, permite a múltiplas redes partilharem a mesma estrutura de rede *OpenFlow*. Para tal este *hypervisor* tem em consideração cinco parâmetros, nomeadamente a largura de banda, topologia, tráfego, *Central Processing Unit* (CPU) do dispositivo e as tabelas de encaminhamento. Uma divisão caracteriza-se por um conjunto de fluxos do plano de dados e possui o seu próprio controlador, sendo que todos os controladores das diversas divisões coexistam na mesma estrutura física da rede. O *FlowVisor* faz a repartição da largura de banda e das tabelas de fluxo de cada *switch* para cada divisão de rede e cada controlador recebe a sua própria tabela de fluxo virtual nos *switches*. Por fim o *FlowVisor* intercepta mensagens *OpenFlow* entre os dispositivos de encaminhamento e os controladores das diversas redes.

Concluindo, o principal objetivo do *hypervisor* numa arquitetura SDN é permitir a execução sequencial ou paralela de aplicações desenvolvidas com linguagens de programação diferentes ou concebidas para diversas plataformas de controlo.

Controladores

Elemento essencial na arquitetura das SDN, o controlador é o principal responsável por toda a lógica do controlo que as aplicações (do plano de gestão) oferecem, pois é ele que gera as configurações da rede baseadas nas políticas definidas pelo administrador. Tal como nos tradicionais sistemas operativos, o controlador proporciona:

- Serviços essenciais, nomeadamente informações acerca da topologia e estados da rede, configurações sobre a distribuição da rede, deteção de novos dispositivos, entre outros;
- Nível de abstração, ou seja, detalhes relativos a um nível mais baixo como conexões e interações entre dispositivos de encaminhamento são abstraídos;
- *Application Programming Interfaces (*APIs) de utilizador.

Numa arquitetura SDN os controladores podem ser implementados de forma centralizada ou distribuída. No primeiro caso, apenas um único controlador é responsável pela gestão de todos os dispositivos de encaminhamento. Porém, nesta abordagem o controlador poderá não ser eficaz na eventualidade da rede apresentar um número considerável de elementos no plano de dados, resultando em limitações na sua escalabilidade. No entanto, controladores como *NOX*-MT [13], *Maestro* [14], *Beacon* [15] e *Floodlight* [16] têm sido adotados em redes empresariais e de centro de dados por atingirem valores de *throughput* necessários nesse tipo de arquiteturas.

A implementação distribuída de controladores, contrariamente à centralizada, permite atingir determinados requisitos num maior número de redes de pequena e grandes dimensões. Essa distribuição pode ser implementada de duas formas, nomeadamente através da utilização de um *cluster* centralizado de nós ou através da distribuição física de vários elementos. Enquanto a primeira oferece um maior valor de *throughput* em redes de centro de dados de grande densidade, a segunda apresenta-se mais resiliente para diferentes tipos de falhas lógicas e físicas.

Interface *Northbound*

Contrariamente à interface *Southband* que apresenta uma proposta amplamente aceite, nomeadamente o *OpenFlow*, a interface *Northbound* ainda é considerada uma questão em aberto, pois esta encontra-se em fase de desenvolvimento não existindo ainda uma solução *standard*. No entanto, controladores como *Floodlight*, *NOX* [17], *OpenDaylight* [18], entre outros, bem como as linguagens de programação *Frenetic* [19], *Procera* [20], *Pyretic* [21], etc, propõem as suas próprias APIs *Northbound* capazes de fazer abstração sobre os detalhes das funções dos controladores e do comportamento do plano de dados face a aplicações de utilizador.

Uma das razões pela ausência de uma interface *Northbound standard* poderá estar no facto de aplicações de rede diferentes requerem requisitos diferentes, por exemplo, APIs para aplicações de segurança são suscetíveis de serem diferentes de aplicações referentes ao *routing*.

Concluindo, as interfaces *Northbound* são primordiais para garantir a portabilidade e interoperabilidade das aplicações entre as diferentes plataformas de controlo.

2.1.3 Plano de Gestão

Linguagens e Abordagens Utilizadas na Virtualização

A capacidade de expressar modularidade e de permitir diferentes níveis de abstração, garantindo no entanto as propriedades desejadas, são duas características de soluções de virtualização. As técnicas de virtualização permitem-nos, por exemplo, obter diferentes pontos de vista de uma única infraestrutura física, simplificando significativamente o desenvolvimento e implementação de aplicações de rede.

Pyretic, *Splendid isolation* [22] e *libNetVirt* [23] são exemplos de linguagens e abordagens utilizadas na virtualização. A linguagem *Pyretic* oferece um alto nível de abstração da topologia de rede através objetos de rede. Estes objetos constituem uma topologia de rede e um conjunto

de políticas que lhes são aplicadas e permitem esconder informações e oferecer os serviços exigidos.

Por sua vez, o *Splendid isolation* e a biblioteca *libNetVirt* utilizam uma abordagem que consiste na divisão estática da rede através de um compilador baseado nas definições da camada de aplicação. O resultado produzido pelo compilador é um programa de controlo com definições de divisão e comandos de configuração da rede. Com esta abordagem o *hypervisor* da rede responsável pela divisão dinâmica da rede deixa de ser necessário. A divisão estática é preferível em implementações com exigências específicas, no entanto, em situações onde são exigidas maiores garantias de desempenho e isolamento simples, a divisão dinâmica é mais apropriada.

Linguagens de Programação

Ao longo dos anos temos verificado uma evolução nas linguagens de programação desde as linguagens de baixo nível, como por exemplo o *Assembly*, até às linguagens de alto nível e mais poderosas como o *Java*, *Python*, entre outras. O facto do uso de linguagens de baixo nível dificultarem a reutilização de *software*, de criar código extenso e modulável e conduzir a um processo de desenvolvimento mais propício a erros levou à migração destas linguagens para linguagens de mais alto nível. Nas redes SDN estas linguagens são concebidas e utilizadas para:

- Permitir um maior nível de abstração para simplificar a programação das tarefas dos dispositivos de encaminhamento;
- Fornecer ambientes mais produtivos e *problem-focused* aos investigadores de redes definidas por *software* e acelerar o desenvolvimento e inovação;
- Promover *software* modulável e a reutilização de código no plano de controlo.
- Fomentar o desenvolvimento da virtualização da rede.

Nas linguagens de baixo nível, como por exemplo o *OpenFlow*, torna-se complicado assegurar que múltiplas tarefas de uma aplicação não interfiram umas com as outras. Isto acontece quando as regras geradas por uma tarefa sobrepõem-se a outras ou quando múltiplas aplicações são executadas num único controlador. Neste último caso, as aplicações geram regras baseadas nas suas necessidades e políticas sem conhecimento acerca das regras geradas por outras, resultando na instalação nos dispositivos de encaminhamento de regras conflituosas, e que por sua vez originam problemas nas operações da rede.

Para além disto, as técnicas de implementação de *software*, como a modularidade e reutilização de código, apresentam-se complicadas de executar em níveis de abstração de baixo nível. Neste

contexto as aplicações são construídas com blocos de código especificamente concebidos e que não podem ser reutilizados noutras aplicações.

Uma outra característica inexistente nas linguagens de baixo nível é a capacidade de implementar *software* para topologias de rede virtuais. Este conceito semelhante à programação orientada a objetos, onde os objetos oferecem abstração sobre os dados e funções, facilita a resolução de problemas particulares sem apreensão acerca das estruturas de dados e a sua gestão. Por exemplo, em vez de implementar e instalar as regras de encaminhamento em cada dispositivo e em toda a rede, topologias de rede virtuais simplificadas podem ser implementadas representando a rede ou um subconjunto da mesma.

Face a isto, linguagens de programação de alto nível foram propostas e com diversos objetivos:

- Evitar o uso do baixo nível e de configurações e dependências na rede como acontece em abordagens tradicionais de configuração de rede;
- Fornecer abstrações que permitem que diferentes tarefas de gestão sejam realizadas correntemente para entender e manter políticas de rede;
- Decompor tarefas múltiplas (por exemplo, *routing*, controlo de acesso, engenharia de tráfego);
- Implementar interfaces de programação de alto nível para evitar conjuntos de instruções de baixo nível;
- Resolver problemas nas regras de encaminhamento, por exemplo, regras incompletas ou que geram conflitos, que podem impedir que um evento relacionado com um *switch* seja acionado, de forma automatizada;
- Endereçar diferentes problemas de *race condition* que são inerentes aos sistemas distribuídos;
- Melhorar técnicas de resolução de conflitos em ambientes distribuídos de tomadas de decisão;
- Fornecer recursos nativos de tolerância a falhas ao longo do processo de criação de rotas no plano de dados;
- Reduzir a latência no processamento de novos fluxos;
- Facilitar a criação de aplicações que armazenam informações de estados, por exemplo, uma *stateful firewall*.

Aplicações de Rede

As aplicações de rede implementam toda a lógica no controlo da rede através dos comandos que vão sendo instalados no plano de dados, orquestrando o comportamento dos dispositivos de encaminhamento. Para além de garantirem tradicionais funcionalidades, tais como *routing*, balanceamento de carga e execução de políticas de segurança, as redes SDN exploram outras abordagens, nomeadamente redução de consumo, *failover*, execução de qualidade de serviço ponto a ponto, virtualização, gestão de mobilidade em redes sem fios, entre outras. Esta variedade de aplicações pode ser dividida em cinco categorias, nomeadamente engenharia de tráfego, mobilidade e redes sem fios, medição e monitorização, segurança e confiabilidade e redes de centro de dados.

- Mobilidade e redes sem fios - As redes SDN apresentam diversas abordagens, como a mobilidade contínua entre *hand-overs*, balanceamento de carga, criação de pontos de acesso virtuais *on-demand*, *downlink* agendável, uso dinâmico do espetro, entre outros, cujo objetivo é facilitar a implementação e a gestão de diferentes redes deste tipo como as *Wireless Local Area Networks* (WLANs) e as redes móveis. Um dos primeiros passos para a realização destas abordagens consiste em proporcionar uma pilha de camadas programáveis e flexíveis. Um exemplo deste conceito é o *OpenRadio* [24]. Construído em torno de uma plataforma de *hardware multi-core*, o componente central do *OpenRadio* é uma camada de abstração de software cujo objetivo é decompor o protocolo das redes sem fios a partir do hardware, através da partilha das camadas *Media Access Control* (MAC) entre os diferentes protocolos.

- Medição e monitorização – Esta categoria abrange duas categorias de aplicações, nomeadamente as que envolvem diferentes tipos de técnicas de amostragem e estimação do tráfego, cujo objetivo é reduzir o encargo do plano de controlo sem influenciar a recolha de estatísticas no plano de dados, e as aplicações que fornecem novas funcionalidades a outros tipos de serviços de rede.

- Segurança e confiabilidade - Os recursos oferecidos pelas redes definidas por *software* possibilitem a capacidade de coletar dados estatísticos acerca da rede e de programar ativamente os dispositivos de encaminhamento. Estas características representam técnicas eficientes para a execução proactiva das políticas de segurança. Nesta perspetiva, a segurança ativa fornece uma interface de programação centralizada, que

simplifica a integração dos mecanismos, para deteção de ataques através da coleção de dados de diferentes fontes, encaminhamento para a configuração apropriada e execução das medidas para bloquear ou minimizar o efeito de possíveis ataques.

- Redes de centro de dados – As SDN podem ajudar os fornecedores de infraestrutura a exporem um maior número de primitivas de rede aos seus clientes através do isolamento de rede virtual, endereçamento personalizado e a colocação de *middleboxes* e aplicações virtuais de *desktop* da nuvem. A funcionalidade que permite explorar totalmente o potencial das redes virtuais na nuvem é a migração da rede virtual. Semelhante à tradicional migração de máquinas virtuais, a migração de uma rede virtual pode ser necessária na eventualidade das suas máquinas virtuais se deslocarem de um lugar para outro. Em plataformas SDN, como por exemplo as *Network Virtualization Platform* (NVP) [25], é possível integrar a migração em tempo real de máquinas virtuais e redes virtuais, no entanto, é necessário reconfigurar dinamicamente todos os dispositivos de rede afetados, quer sejam físicos ou virtuais [9].

O tópico específico de engenharia de tráfego será abordado no subcapítulo seguinte.

2.2 Engenharia de Tráfego nas SDN

A engenharia de tráfego é considerada uma área de trabalho importante, pois é ela que permite otimizar o desempenho de uma rede de dados através da análise, previsão e regulamento do comportamento dos dados transmitidos.

No final da década de 80, as redes ATM foram consideradas na indústria das telecomunicações como *standard*. Nessa altura, a necessidade de satisfazer a diversidade de serviços e o desempenho do tráfego multimédia, originados pelo crescimento dos serviços multimédia como dados, voz e vídeo, tornou o controlo da congestão o principal objetivo da engenharia de tráfego.

No final dos anos 90, as redes IP tornaram-se mais influentes em detrimento das redes ATM, pois permitiram a configuração mais simplificada das redes de dados. Resultado disso, as redes IP chegavam ao mercado mais rapidamente permitindo a popularidade e o aumento drástico de serviços fornecidos através da Internet. Ainda nessa altura surge a tecnologia MPLS. Esta tecnologia subjacente ao protocolo IP foi implementada na tentativa de tornar a engenharia de

tráfego na Internet mais simples, especialmente para os *backbones* da Internet. No entanto, a engenharia de tráfego utilizando o MPLS focaliza-se ainda em demasia no controlo e gestão da Internet sujeita aos mecanismos e elementos de rede atuais. Isto acontece porque muitos protocolos de controlo que residem entre a camada de aplicação e camada de ligação são construídos em cima do conjunto de protocolos da Internet, logo, não têm a capacidade de fornecer mecanismos de engenharia de tráfego suficientes e eficientes para o controlo e gestão do tráfego.

Os mecanismos de engenharia de tráfego nas SDN e a sua implementação como um sistema centralizado apresentam-se mais eficientes comparativamente aos utilizados nas abordagens convencionais, mais concretamente nas redes ATM, IP e MPLS. Isto acontece devido ao elevado número de vantagens que esta arquitetura apresenta, nomeadamente a visão centralizada incluindo informação global da rede; programação dinâmica dos dispositivos de encaminhamento sem necessidade de os programar um por um; interface unificada e aberta para o controlador para a programação do plano de dados e armazenamento dos estados da rede; e *pipelines* de várias tabelas de fluxo nos *switches* para uma gestão de fluxos mais flexível e eficiente. Como mostra a Figura 2.3, alguns dos principais mecanismos de engenharia de tráfego SDN atuais centram-se em quatro vertentes, incluindo a gestão de fluxos, tolerância a falhas, atualização das políticas de encaminhamento e análise de tráfego. Estas vertentes são de seguida analisadas.

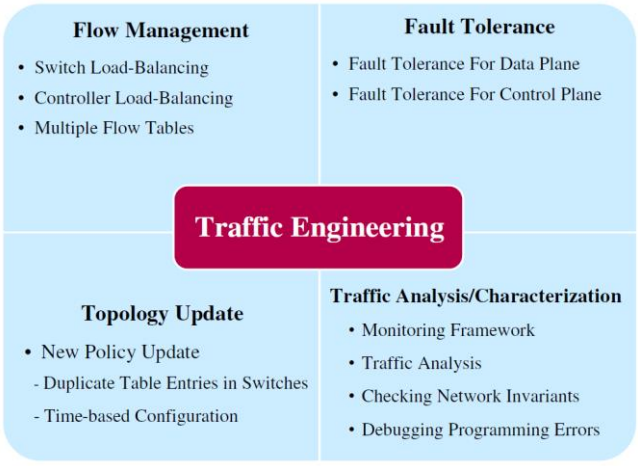

Figura 2.3: Atuais Abordagens de engenharia de tráfego nas SDN [2]

2.2.1 Gestão de Fluxos

Em termos gerais quando um fluxo chega a um *switch*, e este não contém na tabela de fluxos uma entrada correspondente a esse fluxo, o primeiro pacote do mesmo é reencaminhado ao

controlador. De seguida, o controlador decide qual o caminho para o fluxo e instala nos *switches* as entradas correspondentes. Feito isto, os pacotes seguintes do fluxo (ou os pacotes de um outro fluxo com entradas existentes na tabela) são encaminhados no plano de dados ao longo do caminho sem intervenção do controlador. No entanto, a instalação de novas regras de encaminhamento pode apresentar-se como um processo demorado e produzir picos de atraso. Além disso, se um elevado número de novos fluxos chega a um *switch*, e que este não contém os caminhos instalados para eles, uma sobrecarga significativa pode ser produzida tanto no plano de controlo como no plano de dados. As soluções utilizadas nas SDN para evitar a congestão na rede são descritas nas subsecções a seguir.

Balanceamento de Carga nos *Switches*

O mecanismo *Equal Cost Multipath* (ECMP) baseado em *hash* é um mecanismo de balanceamento de carga que utiliza técnicas de *hashing* sobre os fluxos. Este esquema consiste no seguinte: quando um pacote com múltiplos caminhos possíveis chega a um *switch*, este é enviado para o caminho cujo valor corresponde ao valor de *hash* calculado com uma série de campos do cabeçalho do pacote, permitindo assim a divisão da carga para os diversos caminhos da sub-rede.

O *Hedera* [26] foi proposto como forma de resolver as limitações do ECMP baseado em *hash*. Este mecanismo escalável tem uma visão global sobre o *routing* e os requisitos de tráfego, armazena informação nos *switches* acerca dos fluxos permitindo criar caminhos alternativos e não conflituosos e posteriormente instalá-los nos *switches*. Este mecanismo começa por detetar os fluxos de grandes dimensões (*elephant flows*) nos *switches* da camada de acesso quando a ligação atinge um valor limite da sua largura de banda. De seguida, faz uma estimativa da largura de banda requerida pelos fluxos e constrói caminhos alternativos e instala-os nos *switches*. O *Hedera* utiliza ciclos de cinco segundos para armazenar um conjunto de estatísticas dos *switches*, de modo a detetar os *elephant flows* para garantir um equilíbrio entre a utilização da rede global com a sobrecarga mínima no agendamento dos fluxos ativos.

Balanceamento de Carga no Controlador

Como acima mencionado, quando um fluxo é iniciado na rede, o primeiro pacote desse fluxo é reencaminhado ao controlador. Esta característica única do SDN faz do controlador centralizado um dos elementos primordiais na redução da congestão da rede. No entanto, o constante crescimento da rede, nomeadamente o aumento de componentes e o crescente número de fluxos, resulta na redução do desempenho do controlador. Isto acontece, sobretudo em

cenários como centros de dados, pois, o controlador não é capaz de atender todos os diferentes tipos de pedidos referentes às aplicações SDN. No entanto, através de várias implementações de vários controladores, são propostas várias soluções para evitar a congestão entre controladores e *switches*. Estas implementações foram classificadas em quatro categorias: implementação logicamente distribuída (*logically distributed controller deployment*), implementação fisicamente distribuída (*physically distribted controller deployment*), implementação hierárquica (*hierarchical controller deployment*) e implementação híbrida (*hybrid controller deployment*).

Na implementação logicamente distribuída destaca-se o *HyperFlow* [27] e o *DIFANE* [28]. O primeiro utiliza o método *publish/subscribe* com sistema de arquivo *WheelFS* [29] para a comunicação entre controladores e partilha global da rede. O *DIFANE* possibilita a um subconjunto de *authority switches* a distribuição de regras de controladores.

O *Onix* [30] e o *BalanceFlow* [31] são exemplos de implementações distribuídas fisicamente onde o controlo é implementado em plataformas de um ou mais servidores. A plataforma *Onix* utiliza o método *publish/subscribe* com sistema de banco de dados *Network Information Base* (NIB) enquanto o *BalanceFlow* utiliza um "super" controlador responsável pelo balanceamento de carga de todos os outros controladores.

A implementação hierárquica consiste numa hierarquia de dois níveis, uma para controladores locais e outra para um controlador *root* centralizado logicamente. Nesta implementação destaca-se o *Kandoo* [32], onde os controladores locais executam apenas aplicações locais e cada um controla um ou mais *switches*, e um controlador *root* que controla todos os controladores locais e executa aplicações.

Por fim, a implementação híbrida é logicamente centralizada e constituída por *clusters* de controladores fisicamente distribuídos. O *SOX* [33] é um exemplo desta implementação e pode ser composto por: um *cluster* centralizado de controladores (que executam em modo de igualdade) com *failover* e balanceamento de carga automático e direcionado para gerir uma (sub)rede de tamanho considerável; ou por vários *clusters*, distribuídos fisicamente, de controladores, para controlar diferentes (sub)redes com sincronização e consistência necessária.

Múltiplas Tabelas de Fluxo

As tabelas de fluxos existentes nos *switches* são também utilizadas para a gestão dos fluxos que circulam na rede. Na especificação v1.0 do *OpenFlow*, cada *switch* continha uma única tabela de *match* construída na *Ternary Content Addressable Memory* (TCAM). Nesta versão do *OpenFlow* é colocado na tabela de fluxos a agregação de um conjunto de informações relativas

a cada fluxo, nomeadamente, o cabeçalho dos pacotes que o caracterizam, juntamente com uma combinação de uma lista constituída por 10 campos (*10-tuple*), nomeadamente a interface de rede de entrada, o ID (*Identifier*) da VLAN (*Virtual Local Area Network*), o tipo *Ethernet*, o IP e os campos de cabeçalho *Transmission Control Protocol* (TCP). Porém, uma única tabela por *switch* originava um conjunto de regras de tamanho demasiado elevado, resultando na incapacidade de implementações em larga escala devido ao espaço limitado da TCAM e no custo elevado dos recursos.

Na especificação v1.1 foi introduzido o mecanismo de múltiplas tabelas e um conjunto de ações associadas a cada entrada de fluxo. Nesta versão quando um pacote chega a um *switch*, este identifica a entrada de fluxo com maior prioridade e de seguida aplica as instruções com base nos campos do fluxo. Quando existe *match* da informação do pacote com uma entrada de fluxo, o *switch* reencaminha a interface de rede por onde o fluxo entrou, a informação do fluxo e a ação associada, à tabela apropriada, através do processo de *pipeline*. Caso contrário, ou seja, quando o fluxo é desconhecido do *switch*, o fluxo é reencaminhado ao controlador ou simplesmente descartado. A Figura 2.4 expõe o fluxo dos pacotes ao longo das várias tabelas do *switch*, bem como as ações exercidas sobre os mesmos.

Este mecanismo de múltiplas tabelas tornou a gestão dos fluxos mais flexível e eficiente, pois melhora significativamente a utilização da memória TCAM e acelera o processo de *matching*.

Adicionalmente, na versão v1.3+ foi introduzido o *Meter Tabel* (medidor de tabela) para operar os requisitos de qualidade de serviço exigido a cada nível de fluxo para cada pedido de utilizador ou para aplicações de tráfego. Esta versão foi concebida com o objetivo da engenharia de tráfego poder lidar com eficácia os fluxos da rede face ao aumento do desempenho, escalabilidade e flexibilidade do paradigma SDN.

2.2.2 Tolerância a Falhas

O mecanismo de tolerância a falhas é utilizado na eventualidade de ocorrerem falhas na infraestrutura da rede, nomeadamente nos controladores, *switches* e ligações. Este mecanismo deve garantir a confiabilidade da rede de forma eficaz e transparente ao utilizador. Como foi referido anteriormente, controladores e *switches* encontram-se em planos diferentes, sendo por isso necessário a introdução de mecanismos de falhas nesses dois planos.

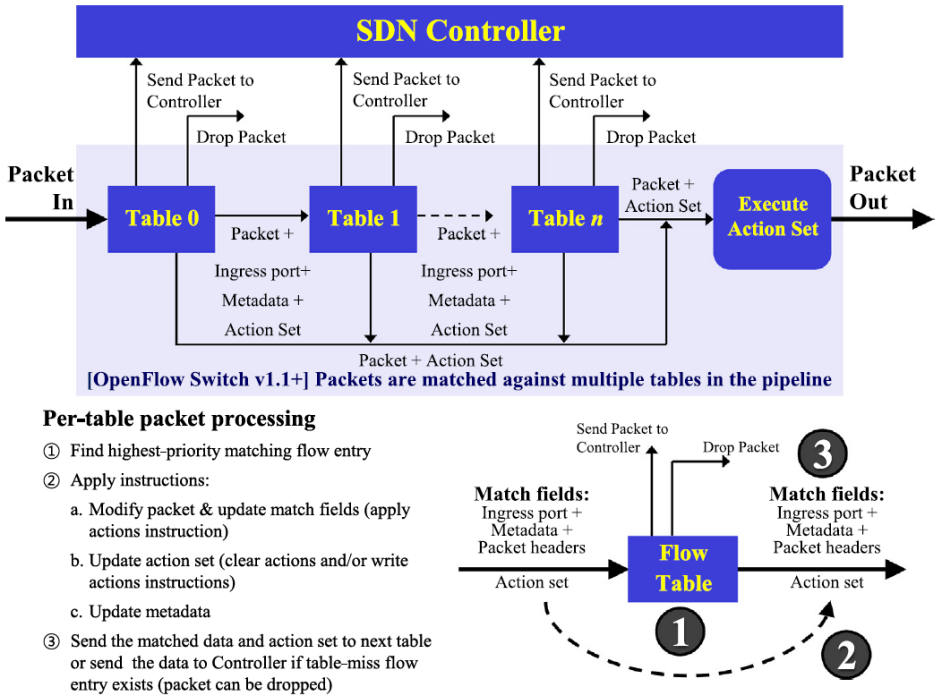

Figura 2.4: Fluxo de pacote através de vários pipelines de múltiplas tabelas [2]

No Plano de Dados

No plano de dados, são dois os mecanismos que podem ser utilizados em caso de falha na rede, nomeadamente designados por restauro do plano de dados (*Data plane restoration*) e a proteção do plano de dados (*Data plane protection*) [34],[35].

O restauro do plano de dados consiste no seguinte: após o controlador ter sido notificado acerca da falha na ligação, uma lista com todas as rotas afetadas pela falha é criada e, para todas elas, é calculada uma rota alternativa usando o algoritmo do caminho mais curto na topologia que foi modificada.

Nos *switches* afetados pela falha de ligação e que se inserem nas rotas alternativas, as entradas na tabela de fluxo são modificadas. Nos *switches* que foram somente afetados pela falha de ligação, as entradas na tabela de fluxo são removidas. Por fim, nos *switches* que apenas se inserem nas rotas alternativas, as entradas na tabela de fluxo são adicionadas.

No mecanismo de proteção do plano de dados, uma rota alternativa é pré calculada e instalada nas entradas de fluxo de todos os *switches*, ficando cada um habilitado com um rota alternativa, utilizada apenas em caso de falha de ligação, e a que é habitualmente utilizada.

No plano de Controlo

Sendo este plano responsável pelo tráfego de dados, através da manipulação das tabelas de fluxos e pelas decisões a tomar sempre que um novo fluxo é iniciado na rede, é crucial que este tenha um mecanismo de tolerância a falhas eficaz de modo a garantir a confiabilidade no restauro da rede. O principal mecanismo utilizado por este plano designa-se por *primary-backup replication* e consiste na utilização de controladores secundários em caso de falha dos controladores em funcionamento. Para tal, deverá existir coordenação entre esses dois tipos de controladores e ter em consideração o número de controladores secundários a implementar.

A coordenação é realizada através de um processo designado por *CPRecovery* [36] e consiste no seguinte: o *switch* envia uma sonda de inatividade para o controlador, pelo canal que o interliga, aguardando um intervalo de tempo específico. Se após esse intervalo o controlador não responder, o *switch* considera-o inoperante e pesquisa na sua lista o controlador secundário e envia um pedido de ligação ao mesmo. Recebendo esse pedido, o controlador secundário altera o seu estado para controlador primário e envia mensagens de atualização de estado para o controlador anteriormente primário.

Em [37] foi realizado um estudo, baseado em topologias reais e implementadas utilizando *Internet2 OS3E* [38] e *Rocketfuel* [39], sobre o impacto do número de controladores na latência e confiabilidade da rede. Nesse estudo verificou-se que o número de controladores deveria estar compreendido entre 0,035n e 0,117n sendo n o número de nós da rede, um mínimo de 3 controladores e um máximo de 11. Esse mesmo estudo sugere ainda que a implementação que melhor garante o equilíbrio confiabilidade/latência consiste na utilização de apenas um controlador. Nessa implementação, além de otimizar o valor médio da latência, a métrica confiabilidade atinge o valor mais elevado. No entanto, quando 3 ou 4 controladores são implementados são obtidos melhores resultados na latência, nomeadamente no seu valor médio e pior cenário, mas uma redução na confiabilidade.

2.2.3 Atualização das Políticas de Encaminhamento

Como foi referido anteriormente o(s) controlador(es) da rede fazem a gestão de todos os *switches* através da configuração dinâmica das políticas de *routing*. Uma vez que estas políticas são alteradas ao longo do tempo, torna-se necessário garantir que os pacotes ou fluxos que circulam na rede não sejam afetados prejudicando a qualidade de serviço e no desperdício de recursos.

O mecanismo de atualização das políticas de encaminhamento pode ser configurado utilizando dois métodos distintos: duplicando as entradas de fluxo nos *switches* ou por um método designado por configuração baseada por tempo (*Time-based configuration* [40][41]).

Método da Duplicação de Entradas

A atualização das políticas de encaminhamento através do método da duplicação de entradas nas tabelas dos *switches* pode ser realizada por pacote ou por fluxo e consiste no seguinte:

1. O controlador procede à instalação das novas regras de configuração em todos os *switches* e introduz um número identificando a nova versão;
2. Os *switches* destacam os pacotes que entram com o número da nova versão da política;
3. Enquanto isso outros *switches* processam os pacotes com a versão que está indicada no pacote, podendo esta ser a nova ou a anterior, sendo que cada pacote é processado apenas por uma única política;
4. Após todos os pacotes etiquetados com a política anterior terem saído da rede, o controlador remove em todos os *switches* as entradas associadas à configuração anterior.

Na atualização por fluxo, quando o controlador procede à instalação das novas regras de configuração, estabelece um intervalo de tempo para que os fluxos sejam processados com a configuração anterior. Após esse intervalo, todos os fluxos passam a ser processados com a nova versão. Nesta atualização todos os pacotes que constituem um fluxo são processados pela mesma política.

Porém este método apresenta uma limitação relativamente ao espaço que cada duplicação ocupa, pois a memória utilizada pelos *switches* (TCAM) é limitada tornando impossível o armazenamento de um elevado número de tabelas com duas configurações cada.

Configuração Baseada por Tempo

Este método possibilita a atualização das políticas de encaminhamento através da atualização coordenada dos *switches*. Esta configuração é realizada pelo controlador através do envio de mensagens com intervalos de tempo definidos, o que lhe permite decidir do momento a que os *switches* procedem à sua atualização.

No entanto este método expõe um problema nas redes SDN, pois nestas redes por cada mensagem enviada e para considerar o *switch* atualizado, o controlador espera por um *acknowledgment* antes de enviar a mensagem ao *switch* seguinte, levando a atualização das políticas de encaminhamento um processo demorado.

2.2.4 Análise de Tráfego

Na engenharia de tráfego das arquiteturas SDN, o tópico da análise de tráfego está significativamente dependente da obtenção em tempo real da informação global relacionada com as aplicações, características do tráfego e estados. Esta engloba ferramentas de monitorização, verificação da rede e de depuração de erros de programação. Enquanto que as ferramentas de monitorização da rede orientam-se na redução significativa da sobrecarga aquando o(s) controlador(es) ou dispositivos de monitorização coletam estatísticas acerca da rede, as de verificação e de depuração de erros direcionam-se em questões de segurança, nomeadamente na deteção e prevenção de intrusões.

Ferramentas de monitorização

Nas redes programáveis, tais como as SDN, as aplicações de gestão exigem que a análise estatística nos recursos da rede em diferentes níveis de agregação seja precisa e realizada em tempo real. As ferramentas de monitorização apresentam-se como uma peça fundamental na gestão da rede, pois apesar de requererem instrumentação especial ou de imporem sobrecarga, monitorizam as métricas de desempenho com o objetivo de adaptar as regras de encaminhamento às mudanças na carga de rede. Embora sendo uma ferramenta de monitorização das tradicionais redes IP, o *NetFlow* [42] tem sido a escolha preponderante das arquiteturas SDN. Esta ferramenta utiliza métodos baseados em sondas que são instalados nos *switches* como módulos especiais e procedem à recolha de estatísticas enviando-as posteriormente ao controlador.

Verificação da rede

Como foi referido anteriormente, redes programáveis tais como as SDN simplificam o desenvolvimento de aplicações de rede. No entanto, erros de programação e a possibilidade das aplicações e/ou utilizadores programarem em simultâneo a mesma interface de rede podem originar no conflito de regras resultando na alteração indevida do comportamento das aplicações. A ferramenta de verificação *VeriFlow* [43] é vista como um *proxy* e é implementado entre o controlador e os *switches* com o objetivo de monitorizar as comunicações de ambos os sentidos e verificar violações invariantes em toda a rede de forma dinâmica em cada regra de encaminhamento inserida.

Depuração de erros de programação

As redes SDN fornecem uma variedade de serviços interligados, tais como serviços de *routing*, monitorização de tráfego, balanceamento de carga e controlo de acesso. No entanto, as linguagens de programação utilizadas para implementar esses serviços são definidas com baixo nível de abstração, dependentes do *hardware* que está subjacente e apresentam falhas no suporte de programação modular, resultando em programas conflituosos, propícios a erros e difíceis de manter. *NICE* [44] apresenta-se como uma ferramenta eficiente que combina o modelo de verificação com o de execução, com o objetivo de encontrar por toda a rede, violações nas suas propriedades de correção devido aos *bugs* nos programas do controlador. Com esta ferramenta o programador *OpenFlow* pode ser instruído através das propriedades de correção genéricas ou de uma aplicação em específico [2].

2.3 Arquiteturas e Topologias de Centros de Dados

Um centro de dados é considerado como um repositório físico ou virtualmente centralizado que permite a computação, o armazenamento e a difusão de informação e dados. Este é tipicamente constituído por um conjunto de computadores, *switches* e/ou *routers*, servidores, balanceadores de carga, equipamento de refrigeração, entre outros, orquestrados pela rede do centro de dados com o objetivo de que todos estes equipamentos funcionem como um todo. A rede de centro de dados e a sua arquitetura são projetadas segundo finalidades distintas. A rede é construída de forma a acomodar dinamicamente os servidores virtualizados e os ambientes de armazenamento, enquanto a arquitetura é responsável por obter o melhor desempenho da rede através da resposta dos seus servidores face às exigências dos serviços e deverá ser ágil para a reconfiguração dinâmica da infraestrutura para novas exigências das aplicações [45]. As arquiteturas de centro de dados podem ser divididas em três grupos: arquiteturas implementadas com *switches*, com servidores ou com ambos os dispositivos, designadas estas por arquiteturas híbridas. Arquiteturas implementadas com *switches* podem apresentar uma topologia *multi-tiered*, *fat tree*, *flattened butterfly*, entre outras. A topologia *camcube* é um exemplo de uma arquitetura baseada em servidores e a *bcube* uma arquitetura híbrida [1].

2.3.1 Arquiteturas Baseadas em *Switches*

Multi-tiered

Esta é a arquitetura tradicional adotada por muitas empresas de média/grandes dimensões. É geralmente constituída por duas ou três camadas (ver Figura 2.5), nomeadamente o núcleo, a camada de agregação, implementada se necessário, e a camada de acesso. No topo da topologia, mais concretamente no núcleo, encontram-se os *switches* responsáveis pelo envio/receção do tráfego para fora do centro de dados e pela ligação à camada inferior, a camada de agregação. Esta última contém os *switches* que fazem a ligação com a camada de acesso e que oferecem os principais serviços de rede, tais como, serviço de domínio, balanceamento de carga, etc. Por fim, a camada de acesso é composta pela ligação aos *hosts* [45].

Os *switches* da camada de acesso apresentam um número de interfaces de rede compreendido entre 48 e 288 e com uma largura de banda de 1 Gbit/s nas ligações aos *host* e de 10 Gbit/s nas ligações ascendentes, ou seja, na ligação com a camada de agregação. Nos níveis mais altos da hierarquia são utilizados *switches* com menor densidade de interfaces de rede, tipicamente com 32 a 128 interfaces de rede, porém, estes apresentam maior largura de banda nas suas ligações descendentes com 10 Gbit/s, a mesma que nas ascendentes.

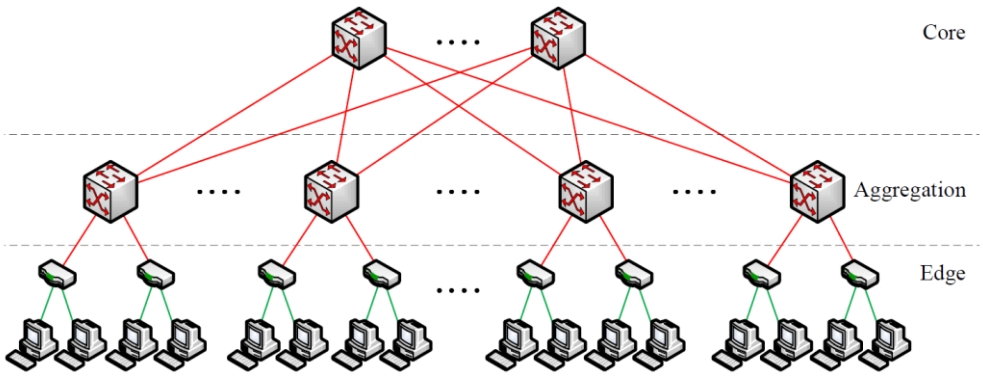

Figura 2.5: Multi-tiered [46]

Fat tree

Esta topologia, cuja arquitetura é apresentada na Figura 2.6, construída sob a forma de uma árvore de múltiplos ramos foi utilizada como topologia de estudo para diversos investigadores em redes de centro de dados, como por exemplo *Portland* [47], *Hedera*, *Elastic Tree* [48], entre outros. A sua constituição depende do número de *pods*, isto é, uma *fat tree* construída com k *pods* contém duas camadas de $k/2$ *switches* cada. Na camada mais baixa os *switches* com k

interfaces de rede encontram-se diretamente ligados com os $k/2$ *hosts*. Cada um dos restantes $k/2$ interfaces de rede estão ligados a $k/2$ dos k interfaces de rede da camada superior, a camada de agregação.

O núcleo é constituído por $(k/2)^2$ *switches*, contendo cada um, uma interface de rede que faz a ligação com cada um dos k *pods*. A interface de rede i de qualquer *switch* do núcleo está conectada ao *pod* i de tal modo que interfaces de rede consecutivas na camada de agregação de cada *pod* estão ligadas aos *switches* do núcleo nos $k/2$ passos. No total, numa arquitetura *fat tree* com k interfaces de rede por *switches*, estão conectados $k^3/4$ *hosts* [46].

Uma vantagem desta topologia em árvore é que todos os dispositivos de encaminhamento são idênticos no que diz respeito à sua densidade de interfaces de rede e largura de banda (1 GigE), o que permite reduzir significativamente o custo da arquitetura. Além disso, as *fat trees* são *rearrangeably non-blocking*, o que significa que para padrões de comunicação arbitrários, existe um conjunto de caminhos cuja largura de banda disponível para os *end hosts* será utilizada na sua totalidade.

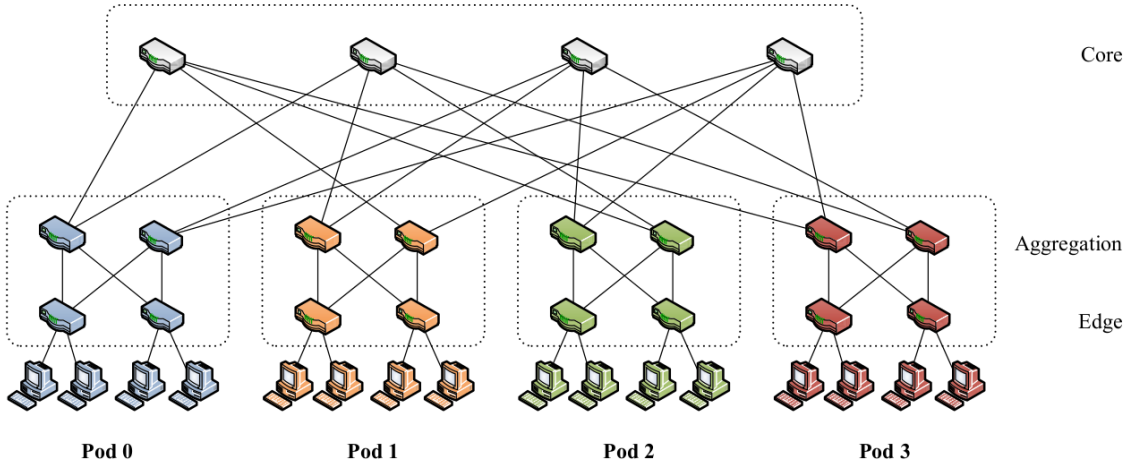

Figura 2.6: Fat tree [46]

Flattened Butterfly

A *flattened butterfly* [49], originalmente proposta para interconexões *on-chip*, consiste numa topologia simétrica multidimensional que tira vantagem do elevado número (milhares) de interfaces de rede por *switches* para originar redes com diâmetros reduzidos. A sua constituição pode ser exposta através do número de dimensões, representado por n, e pelo número de *switches* por dimensão, representado por k. Por exemplo, numa *flattened butterfly* (ver Figura 2.7) de $n = 2$ e $k = 8$, os oito *switches* estão conectados aos outros sete e cada um dos oito

switches conectados a oito *hosts*. Uma *flattened butterfly* de n dimensões e k *switches* por dimensão é construída a partir da junção de uma *flattened butterfly* de n-1 dimensões e k *switches* por dimensão com outra de duas dimensões e k *switch* por dimensão. Concluindo, uma *flattened butterfly* de $n = 3$ e $k = 8$ pode ser implementada copiando outra de $n = 2$ e $k = 8$ oito vezes e conectando de seguida cada um dos *switches* de um grupo aos sete *switches* correspondentes, um em cada um dos outros 7 grupos.

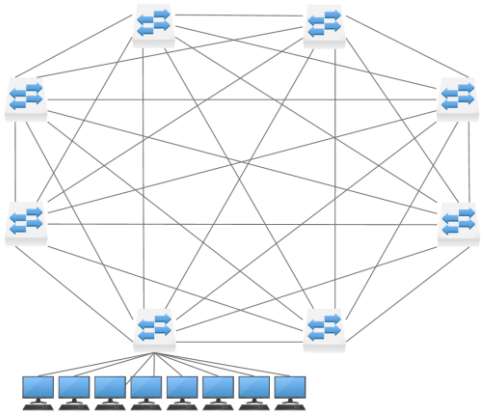

Figura 2.7: Flattened butterfly

2.3.2 Arquitetura Baseadas em Servidores

Camcube

Arquitetura *camcube* é uma arquitetura constituída unicamente por servidores, ou seja, neste tipo de arquiteturas os servidores, para além de representarem *hosts* também procedem ao encaminhamento dos dados na rede [1]. Nesta arquitetura cada servidor está conectado a outros seis e representa uma topologia *3D Torus* (ver Figura 2.8), ou seja, em cada servidor é atribuído um endereço de modo a que um conjunto dos servidores assuma a forma de um sistema de coordenadas tridimensional em torno de um servidor central que representa a origem [50].

2.3.3 Arquiteturas Híbridas

BCube

A arquitetura *bcube* é uma estrutura recursivamente definida por servidores e *switches*. É constituída por um elemento designado por $BCube_k$ construído a partir de n $BCube_{k-1}$, n^{k+1} servidores de $k + 1$ interfaces de rede e n^k *switches* de n interfaces de rede cada. Um $BCube_k$ contém $k + 1$ níveis estando cada servidor conectado a um *switch* de cada nível [1], [51]. Como podemos verificar na Figura 2.9 não existe ligação direta entre servidores e entre *switches*.

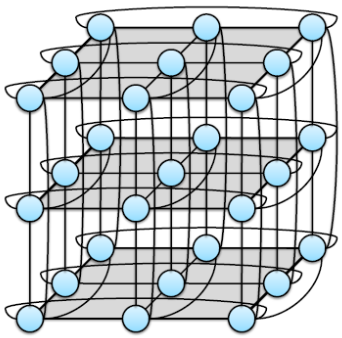

Figura 2.8: 3D Torus [52]

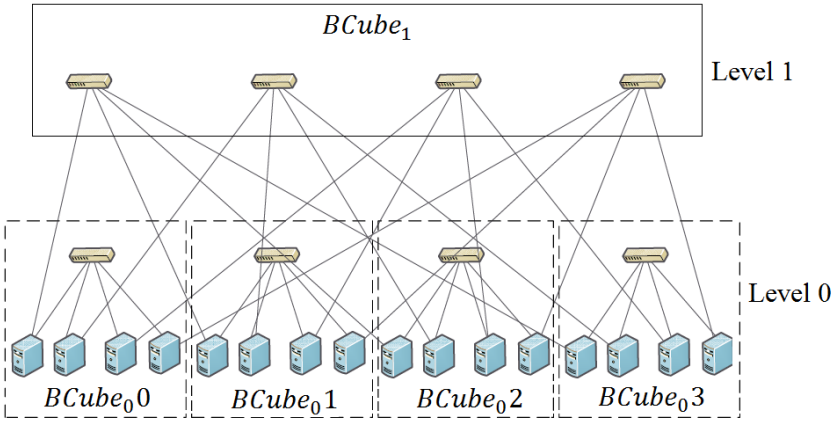

Figura 2.9: BCube [45]

2.4 Métricas Comparativas de Centros de Dados

Como foi referido anteriormente, uma arquitetura de centro de dados pode ser implementada de diversas formas. Esta diversidade de implementações deve-se aos diferentes cenários de utilização traçados pelas empresas responsáveis por esses centros e que requerem a otimização de características diferentes. Se os cenários de utilização forem orientados à computação intensiva, os centros de dados necessitam de ser equipados com nós de maior poder de computação. No entanto, se os cenários de utilização se focalizarem principalmente na transmissão intensiva de dados, as redes de centros de dados devem fornecer o desempenho necessário tendo em consideração os custos que este acarreta. O desempenho de uma rede de centro de dados pode ser tipicamente caracterizado segundo métricas bem conhecidas, tais como a largura de banda, confiabilidade, *throughput*, consumo de energia e custo, podendo estas estar relacionadas. Por exemplo, o custo depende de uma variedade de fatores, como o consumo de energia, os servidores e *switches* utilizados, cabos, etc. A infraestrutura de uma

rede de centro de dados desempenha um papel cada vez mais importante na medida em que é esta que influencia o custo e o desempenho do sistema global, algo que tem sido geralmente subestimado, uma vez que não contribuem diretamente para os lucros das empresas destes centros.

Neste subcapítulo serão abordadas algumas métricas que caracterizam as arquiteturas de centros de dados, como a escalabilidade, a diversidade de caminhos, número de saltos, custo e consumo de energia, e será feita a comparação das mesmas nas arquiteturas abordadas no subcapítulo anterior. Em algumas métricas a arquitetura *camcube* não será comparada com as restantes pelo facto de esta apresentar um desempenho indesejável, comparativamente às outras arquiteturas, devido aos longos caminhos que esta apresenta.

2.4.1 Escalabilidade

Num centro de dados a escalabilidade representa a capacidade da rede poder suportar uma carga significativa de tráfego e acomodar-se às variações que esta sofre ao longo do tempo. A escalabilidade pode ser comparada nos vários tipos de topologia através de um parâmetro designado por *oversubscription*, que corresponde ao rácio entre a largura de banda do tráfego injetado na rede e a capacidade da mesma. Para compararmos as diversas topologias quanto à sua escalabilidade consideramos que em todas as topologias, todos os *hosts* conseguem comunicar entre si com largura de banda máxima, ou seja, com um *oversubscription* de 1:1.

Considerando um estudo realizado em [1] as topologias *flatenned butterfly* e $BCube_2$ apresentam-se mais escaláveis em relação às topologias *multi-tiered* e *fat tree*. Como podemos observar na Figura 2.10, quando o número de interfaces de rede por *switch* é de aproximadamente 64, o número de *hosts* numa arquitetura *flatenned butterfly* é cerca de 8 vezes superior ao número de *hosts* suportados numa arquitetura *fat tree* e 60 vezes superior numa *multi-tiered*.

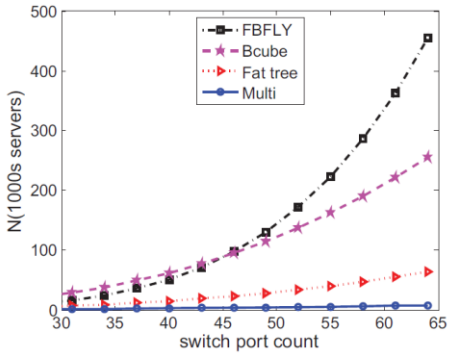

Figura 2.10: Número de hosts vs. número de interfaces de rede nos switches [1]

No entanto, quando analisamos (ver Figura 2.11) o número de interfaces de rede por *switch* por cada *hosts* ou servidor, a arquitetura híbrida apresenta-se mais eficiente, pois apenas necessita de 3 interfaces contra 4,5 e 5 nas restantes.

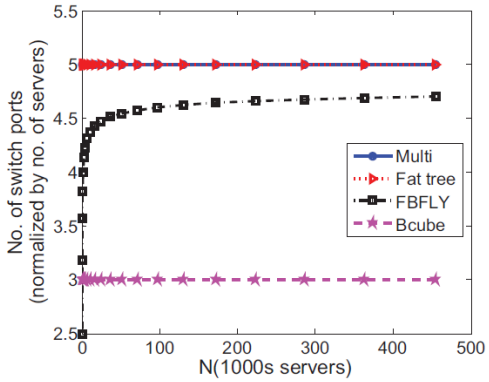

Figura 2.11: Número de interfaces de rede por switch por host vs. número de hosts [1]

2.4.2 Diversidade de Caminhos

A diversidade de caminhos consiste na quantidade de caminhos mais curtos alternativos entre um par de *hosts*. No entanto esta métrica pode ser analisada de duas formas distintas, nomeadamente analisando o número máximo de caminhos mais curtos entre um par de *hosts* e a média dos caminhos mais curtos, considerando todos os pares possíveis nos dois casos.
Nestes dois casos de estudo, e como podemos visualizar na Figura 2.12 e na Figura 2.13, a topologia *fat tree* supera de forma considerável as outras topologias.

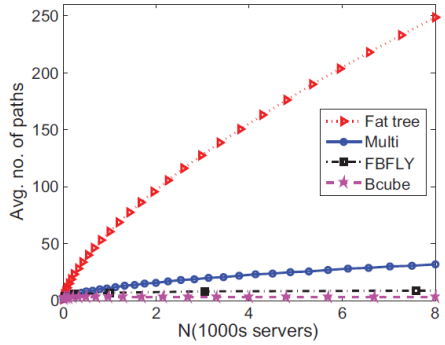

Figura 2.12: Número médio de caminhos mais curtos entre um par de hosts [1]

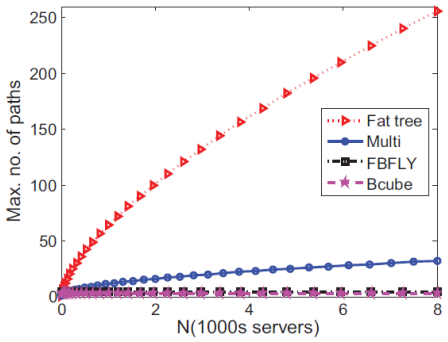

Figura 2.13: Número máximo de caminhos mais curtos entre um par de hosts [1]

2.4.3 Número de Saltos

Esta métrica analisa a média de saltos no caminho mais curto entre todos os pares de *hosts*. Juntamente com a métrica anterior, os *hosts* comunicam entre si utilizando toda capacidade das ligações para que o desempenho seja normalizado. Observando a Figura 2.14 verificamos que quando uma topologia atinge aproximadamente os mil *hosts* e/ou servidores, o número médio de saltos começa a estabilizar. A *flattened butterfly* apresenta uma média de 5 saltos, 5,5 saltos para a $BCube_2$ e uma média de saltos próxima de 6 para a *fat tree* e *multi-tiered*.

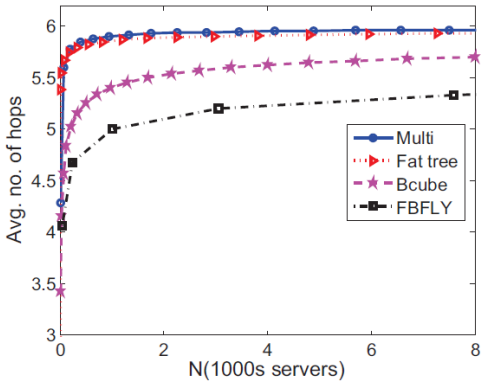

Figura 2.14: Média de saltos por host [1]

2.4.4 Custo

O custo de uma topologia baseia-se no preço dos componentes que constituem a sua infraestrutura. Numa arquitetura baseada em *switch* incluem-se os *Network Interface Card* (NIC), os *switches* e os cabos, enquanto nas topologias baseadas em servidores e híbridas os componentes abrangidos são igualmente os NIC, os servidores e os *switches*, no caso das arquiteturas híbridas. O preço dos componentes utilizados para a comparação estão referidos

em [53] e foram utilizados aproximadamente 4 mil *hosts* em todas as topologias para que a comparação seja coerente.

Como a Tabela 2.1 o comprova a implementação de uma arquitetura por *switches* implica um maior custo em componentes sobretudo quando a topologia é uma *multi-tiered* onde esta apresenta-se bastante mais dispendiosa quando comparada com as restantes duas. A topologia *camcube* chega em quarto lugar com um custo muito próximo da terceira, a *flattened butterfly* e finalmente, no último lugar da tabela encontramos a arquitetura híbrida *bcube*.

Tabela 2.1: Comparativo sobre os custos das topologias [1]

	Cost(k$)			
	Switch	NIC	CPU core	Total
Multi-tiered	13360	690	0	14050
Fat tree	8806	690	0	9496
FBFLY	5939	690	0	6629
Camcube	0	3686	3276	6962
BCube	614	1843	1638	4095

2.4.5 Consumo de Energia

Aqui também são assumidas as características descritas em [53] e a conexão de 4 mil *hosts* para compararmos os consumos de energia das diferentes topologias. Como se observa na Tabela 2.2, e contrariamente à métrica anterior, a topologia *multi-tiered* apresenta o menor consumo de energia. As topologias *bcube* e *flattened butterfly*, com consumos muito próximos, ocupam a segunda e terceira posição, respetivamente, das topologias com menor consumo de energia. Finalmente, as topologias que apresentam um maior consumo de energia são a *camcube* e *fat tree* [1].

Tabela 2.2: Comparativo sobre o consumo de energia das topologias [1]

	Power(kW)			
	Switch	NIC	CPU core	Total
Multi-tiered	110	46	0	156
Fat tree	264	44	0	308
FBFLY	178	45	0	223
Camcube	0	245	162	407
BCube	18	122	81	221

2.5 Sumário

Neste capítulo foram abordados os diversos conceitos essenciais ao trabalho desenvolvido, nos quais se inserem as camadas que constituem a arquitetura das SDN bem como os diferentes mecanismos de engenharia de tráfego utilizados neste tipo de redes. Destes mecanismos deu-se maior relevância à gestão de fluxos, nomeadamente no balanceamento de carga nos dispositivos de encaminhamento e controladores de rede e no conceito das múltiplas tabelas que os primeiros fazem uso. Também foram abordados diferentes tipos arquiteturas de redes de centros de dados e as topologias que lhes são associadas. Por último, apresentou-se um conjunto de métricas e procedeu-se à comparação dos resultados dessas mesmas métricas nesses mesmos centros de dados.

3. Especificação e Desenvolvimento da Bancada de Testes

Este capítulo descreve o trabalho de investigação e de desenvolvimento realizado em torno da conceção de uma bancada de testes que permite o estudo de mecanismos de balanceamento de carga em redes de centros de dados. Nesse sentido a secção 3.1 descreve a plataforma de emulação e o controlador SDN utilizado, bem como a topologia de rede sob estudo. De seguida, na secção 3.2, será dado relevo ao desenvolvimento de três mecanismos de balanceamento de carga que foram abordados neste trabalho.

3.1 Emulação de Centro de Dados e Controladores SDN

Nesta secção pretende-se especificar os componentes que tornarão possível a simulação de uma arquitetura de rede de centro de dados. Para tal, será apresentado em primeiro lugar o emulador *Mininet* referindo as suas características, componentes que oferece, assim como as vantagens e desvantagens da sua utilização (secção 3.1.1). De seguida será feita a descrição do *Floodlight*, o controlador escolhido para este trabalho, nomeadamente focando a sua arquitetura e aplicações e serviços que o constituem (secção 3.1.2). Por fim será especificada a topologia de rede escolhida e sobre a qual serão testados os mecanismos de balanceamento de carga (secção 3.1.3).

3.1.1 Mininet

Sendo o paradigma SDN ainda recente, o foco da sua investigação têm sido essencialmente através do estudo em detrimento de testes em plataformas reais. No entanto, investigadores deste tipo de redes enfrentam dificuldades quando necessitam de testar os seus novos mecanismos nos controladores, dispositivos de encaminhamento, ou no protocolo *OpenFlow*. Estas dificuldades acontecem essencialmente devido ao escasso número disponível de dispositivos baratos capazes de serem implementados nestas redes. Além disso, em casos mais específicos, quando é necessário simular grandes redes com um elevado número de *hosts*, *switches* e controladores SDN, a utilização da Internet não é uma boa prática, pois, esta contém configurações impróprias que podem causar problemas. Uma das soluções utilizadas face a este problema está na implementação de protótipos e na simulação/emulação dos mesmos através do conceito de virtualização.

Para tal, algumas ferramentas foram criadas entre as quais o *Mininet* [54]. O *Mininet* é um emulador que permite criar protótipos de rede de *hosts* virtuais, *switches*, controladores e ligações utilizando apenas um único computador. O *Mininet* executa código real incluindo aplicações de rede *Unix/Linux standard*, assim como *kernel* Linux e a pilha de rede. Com isto, o código que for desenvolvido e testado no *Mininet*, para um controlador *OpenFlow*, *switch* ou *host,* poderá ser transferido para um sistema real, com o mínimo de alterações, com o objetivo de testar, avaliar o desempenho e implementar num ambiente real. Ou seja, um projeto desenvolvido no *Mininet* geralmente poderá ser replicado diretamente para plataformas reais.
As características associadas ao *Mininet* são as seguintes:

- Flexibilidade: novas topologias e novas funcionalidades podem ser definidas no *software*, utilizando linguagens de programação e sistemas operativos comuns;
- Aplicabilidade: implementações feitas em protótipos devem ser também utilizáveis em redes reais baseadas em *hardware* sem quaisquer alterações no código fonte;
- Interatividade: a gestão e o funcionamento da rede simulada deve ocorrer em tempo real como acontece nas redes reais;
- Escalabilidade: o ambiente que permite criar os protótipos deve ser escalável para grandes redes, nomeadamente com centenas ou milhares de *switches*, numa única máquina;
- Realista: o comportamento do protótipo deve representar o comportamento em tempo real e com elevado grau de confiança, de modo que aplicações e pilhas protocolares possam ser utilizáveis sem qualquer modificação do código;
- Partilhável: os protótipos criados devem ser facilmente partilhados com outros investigadores, permitindo-os de seguida, executá-los e/ou modifica-los.

Tal como grande parte dos sistemas operativos, que virtualizam os recursos de computação utilizando a abstração baseada em processos, o *Mininet* utiliza essa mesma abstração de virtualização para executar *hosts* e *switches* num único *kernel*. Desde a versão 2.2.26, o *Linux* tem apoiado a utilização de *namespaces* de rede, um recurso de virtualização leve que fornece processos individuais com interfaces de rede separadas, tabelas de *routing* e tabelas *Address Resolution Protocol* (ARP). O *Mininet* pode criar *kernel* ou *switches user-space OpenFlow*, controladores para os controlar, e *hosts* para comunicarem através da rede simulada. O *Mininet* liga os *switches* e *hosts* usando pares *Ethernet* virtuais (*Veth*) e utiliza em grande parte o *Python* como linguagem de programação.

Elementos SDN no *Mininet*

Como foi referido anteriormente o *Mininet* pode criar elementos SDN, personalizá-los, compartilhá-los com outras redes e realizar interações. Nestes elementos incluem-se *hosts*, *switches*, controladores e ligações. No *Mininet*, um *host* é um processo simples com o seu próprio ambiente de rede que é executado no sistema operativo. Cada *host* fornece processos com propriedades exclusivas de interface de rede virtual, interfaces de rede, endereços e tabelas de encaminhamento. Os *switches OpenFlow* criados pelo *Mininet* fornecem a mesma semântica de entrega de pacotes que seria fornecida por um *switch* físico. Numa emulação utilizando o *Mininet*, os controladores podem ser executados numa rede real ou simulada, desde que a máquina na qual os *switches* estão em execução tenha conetividade com o controlador. O *Mininet* permite ainda, se for necessário, criar um controlador *standard* dentro do ambiente de simulação local e ligações virtuais entre os elementos através de suas interfaces virtuais.

Interação com a Rede e uso de Ferramentas de Controlo

Após terem sidos criados os elementos e as respetivas conexões, é possível executar comandos nos *hosts* para testar a funcionalidade da rede e verificar o funcionamento dos *switches*. Para tal, o *Mininet* incluiu uma interface de linha de comandos que permite aos utilizadores procederem ao controlo e gestão da rede criada. Os comandos inseridos são interpretados pelo emulador e executados no ambiente de rede simulado.

Controladores SDN e o *Mininet*

Um dos princípios das SDN é a capacidade de controlar o processo de encaminhamento de pacotes através de uma interface única. O controlador pode centralizar todas as comunicações com os elementos programáveis da rede e fornecer uma visão única do seu estado, isolando os detalhes de cada elemento. Uma das vantagens das SDN é exatamente esta visão centralizada da rede que torna possível desenvolver uma análise detalhada e decidir como o sistema deve funcionar.

O emulador *Mininet* implementa a ligação entre *switches* e diferentes controladores, como por exemplo o *NOX* e o *Floodlight*, tornando possível aos investigadores interessados em criar e testar recursos de controlador, a capacidade de usar o *Mininet* para realizar as suas simulações.

Vantagens e Desvantagens do *Mininet*

Comparativamente a várias propostas de emuladores, testes ao *hardware* e a simuladores, o Mininet:

- Permite o arranque mais rápido - na ordem dos segundos em vez de minutos em outras abordagens.
- Apresenta escalabilidade mais eficiente - centenas de servidores e *switches*.
- Permite a realização de testes com maior largura de banda - tipicamente na ordem dos 2 Gbit/s em *hardware* mais simples.
- É mais fácil de instalar – máquina virtual disponível que pode ser executada no *VMware* ou *VirtualBox* em várias plataformas como *Mac*, *Windows* ou *Linux*, com ferramentas *OpenFlow* v1.0 já instaladas.
- É mais barato e está sempre disponível.
- Apresenta maior rapidez na sua reconfiguração e reinício.
- Executa código real e inalterado, incluindo o código da aplicação, o código do *kernel* e o código de plano de controlo (código do controlador *OpenFlow* e do *Open vSwitch*).
- Mais fácil de conectar a redes reais;
- Oferece um desempenho interativo.

No entanto, o *Mininet* apresenta também algumas limitações, pois, as redes criadas não conseguem exceder o CPU ou a largura de banda disponível num só servidor, e não permite executar *switches OpenFlow* que não sejam compatíveis com *Linux* [8][54].

3.1.2 Controlador *Floodlight*

Esta secção descreve o trabalho de familiarização efetuado sobre o controlador SDN selecionado para este projeto.

O *Floodlight* é um controlador *OpenFlow* para redes empresariais baseadas na linguagem de programação *Java* e distribuído pela *Apache license*. Este controlador foi originalmente um projeto do controlador *Beacon*, tendo depois a *Big Switch Network* contribuído para a sua comunidade *open source*.

A sua infraestrutura é constituída por módulos escritos em Java que prestam um conjunto de serviços. Semelhante a outros controladores, quando o *Floodlight* é executado, ambas as operações das interfaces *Northbound* e *Southbound* do controlador são ativadas, ou seja, quando o controlador é executado, o conjunto de aplicações referentes aos módulos são executadas. As APIs *Representational State Transfer* (REST) da interface *Northbound* expostas por todos os módulos em execução tornam-se disponíveis através da interface REST definida. Com isto, qualquer aplicação pode interagir com o controlador através do envio de comandos *Hypertext Transfer Protocol* (HTTP) REST. Por outro lado, na interface *Southbound*, o módulo *Provider*

do *Floodlight* começa por ouvir na interface de rede TCP para as ligações a partir dos dispositivos *OpenFlow*.

A escolha deste controlador deveu-se aos seguintes pontos:

- Oferece um sistema de carregamento de módulos que facilita a implementação e o desenvolvimento dos mesmos.
- Fácil de configurar e com dependências mínimas.
- Suporta uma vasta gama de *switches OpenFlow* virtuais e físicos.
- Permite a junção de redes *OpenFlow* com outros tipos de redes.
- Foi projetado com o objetivo de fornecer alto desempenho.
- Suporte para a plataforma de gestão na nuvem *OpenStack*.

Arquitetura Floodlight

A arquitetura do controlador *Floodlight* descreve-se como uma arquitetura modular como mostra a Figura 3.1. A arquitetura do núcleo inclui vários módulos, tais como gestão de topologia; gestão de dispositivos/*end-station*; computação de caminhos/rotas; infraestrutura para acesso web; contadores *OpenFlow*; e um sistema de armazenamento de estado, estando todos eles geridos pelo sistema de gestão de módulos.

Aplicações

O *Floodlight* disponibiliza um conjunto de aplicações, entre as quais as aplicações *Forwarding*, *Learning Switch*, *Static-Flow Entry*, *Virtual Network Filter* e a aplicação *Firewall*.

A primeira permite o encaminhamento de pacotes entre dois dispositivos que podem ser, ou não, conectados utilizando o protocolo *OpenFlow*. A aplicação *Learning Switch* possibilita a aprendizagem por parte dos *switches* acerca dos nós na rede através dos pacotes que recebem. A aplicação *Hub* apenas distribui qualquer pacote para todas as outras interfaces de rede ativas. A *Static-Flow Entry* consiste na adição de uma entrada de fluxo num *switch* específico. A *Virtual Network Filter* é uma aplicação que permite a implementação de múltiplas redes usando o endereçamento MAC, num único domínio da camada 2. Por fim, a *Firewall* aplica regras *Access Control List* (ACL), que são o conjunto de condições que têm como finalidade o controlo do tráfego baseado num conjunto de políticas.

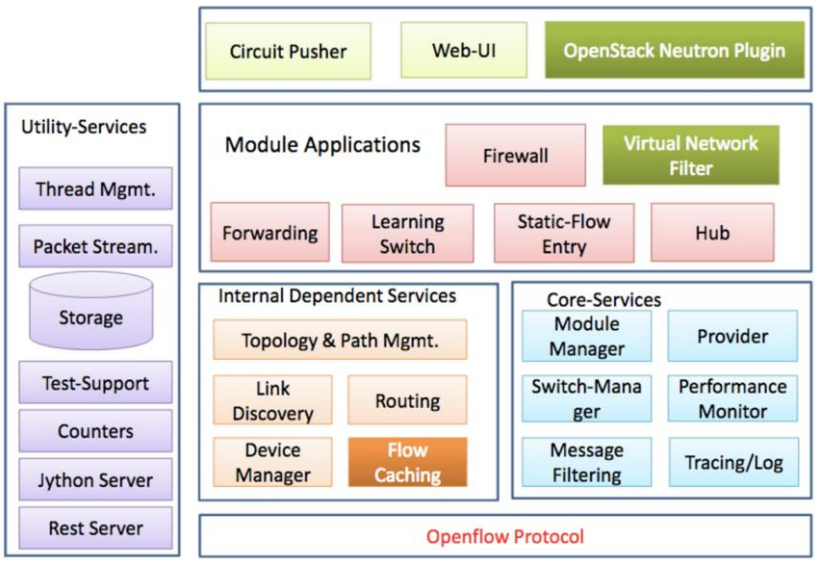

Figura 3.1: Arquitetura Floodlight [55]

Serviços

Os serviços definem a capacidade do controlador e são usados pelas aplicações da interface *Northbound*. Assim sendo, o *Floodlight* inclui variados serviços como a descoberta de estados e eventos da rede para o suporte na comunicação entre *switches*, armazenamento, *threads* e interface de utilizador web. A Tabela 3.1 descreve alguns dos serviços oferecidos pelo controlador.

Tabela 3.1: Alguns serviços do controlador Floodlight

Nome do Serviço	Descrição
Floodlight-Provider	Faz a gestão das conexões seguras com os *switches*. O módulo *FloodlightProvider* é responsável por traduzir as mensagens *OpenFlow* recebidas nos eventos, que podem ser processadas por outros módulos. Além disso, também garante a ordem das mensagens recebidas antes de as transmitir para outros módulos.
Device-Manager	O *Device Manager* mantém o registo dos dispositivos ou *end-stations* através dos pedidos *PacketIn*, em vez da informação (MAC, VLAN, etc.) presente nos mesmos. Com isto, é possível saber qual a interface de rede que está ligado ao *switch* conectado a um determinado dispositivo.
Link Discovery Manager	Semelhante a outros controladores SDN, a descoberta de ligações também utiliza *Link Layer Discovery Protocol* (LLDP). Uma ligação está definida para ser estabelecida entre dois *switches* se um LLDP é enviado para fora de uma interface de rede de um *switch*, e esse mesmo LLDP é recebido numa interface de outro *switch*.
Topology-Service	O serviço de topologia computa topologias com base na informação das ligações do *Link Discovery Manager*. O termo "ilha *OpenFlow*" é designado a um conjunto de *switches*

	interligados e geridos pela mesma instância de um controlador SDN (*Floodlight*). Adicionalmente, as ilhas podem ser interligadas através de *switches* não habilitados com o protocolo *OpenFlow* no mesmo domínio de camada 2.
Flow Cache	Este módulo permite aos investigadores implementarem soluções de acordo com as suas necessidades através da manipulação de uma série de eventos.
Packet Streamer	O *Packet streamer* permite encaminhar pacotes *OpenFlow* para qualquer dispositivo de monitoração conectado na rede. Fornece uma interface para especificar mensagens *OpenFlow* de interesse. É considerado tipicamente como um filtro.
Memory Storage Source	O *Memory Storage Source* consiste numa fonte de armazenamento de dados partilhados. Outros módulos podem criar/excluir/modificar dados nessa fonte. Além disso, os módulos também podem registar as alterações aos dados indicando as respetivas tabelas e linhas.
ThreadPool	O serviço *ThreadPool* é um modulo *Floodlight wrapper* para o *Scheduled Executor Service* do *Java*. Pode ser usado para criar *threads* que podem ser executadas em tempos específicos ou periodicamente.

APIs REST

O *Floodlight* inclui um servidor *RestAPI*, que usa a biblioteca *Restlets*. Com essa biblioteca qualquer módulo desenvolvido pode expor APIs REST suplementares através do serviço *IRestAPI*. Tipicamente, os módulos que dependem do servidor REST expõem APIs através da implementação de *RestletRoutable* numa classe. O controlador oferece um conjunto extenso de APIs REST para obter e definir vários tipos de informação e é recomendada para desenvolver aplicações utilizando funções suportadas pelo *Floodlight*. A Figura 3.2 que se apresenta seguidamente mostra a estrutura API REST do *Floodlight* [16][55].

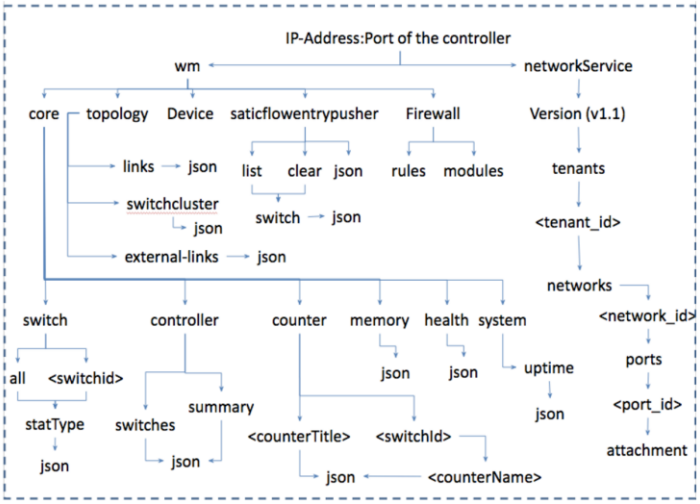

Figura 3.2: Estrutura da API REST do Floodlight [55]

3.1.3 Topologia de Rede

No trabalho realizado para testar os mecanismos de balanceamento de carga desenvolvidos neste projeto foi utilizada uma topologia de rede *fat tree* [46], sendo esta implementada com recurso a uma script em *Python* disponível no *Bitbucket* [56]. A topologia apresenta 4 *pods*, resultando numa topologia constituída por 4 *switches* na camada do núcleo, 8 *switches* na camada de agregação e 8 *switches* na camada de acesso. Ligado à última camada estão conectados 16 *hosts*, 4 em cada *pod* (ver Figura 3.3).

Endereçamento do Equipamento

Para o endereçamento dos *switches* e *hosts* foi adotado o método utilizado em [46], ou seja, utilizando as seguintes condições: Aos *switches* dos *pods* são dados endereços com o formato 10.*pod*.*switch*.1, onde *pod* corresponde ao número do *pod* (que varia de 0 a 3) e *switch* indica a posição que ocupa o *switch* no *pod* (varia de 0 a 3 a partir da esquerda para a direita e de baixo para cima). O endereçamento dos *switches* da camada do núcleo utilizam o formato 10.4.j.i, onde j e i indicam as coordenadas do *switch* (variando entre 1 a 2, a partir do canto superior esquerdo). Os *hosts* têm o endereço seguinte em relação ao *switch* a que está ligado e têm o formato 10.*pod*.*switch*.ID, onde ID é a posição do *host* daquela sub-rede (e varia de 2 a 3 partindo da esquerda para a direita). A Figura 3.4 apresenta o pseudocódigo utilizado para a criação da topologia *fat tree* com o endereçamento acima descrito.

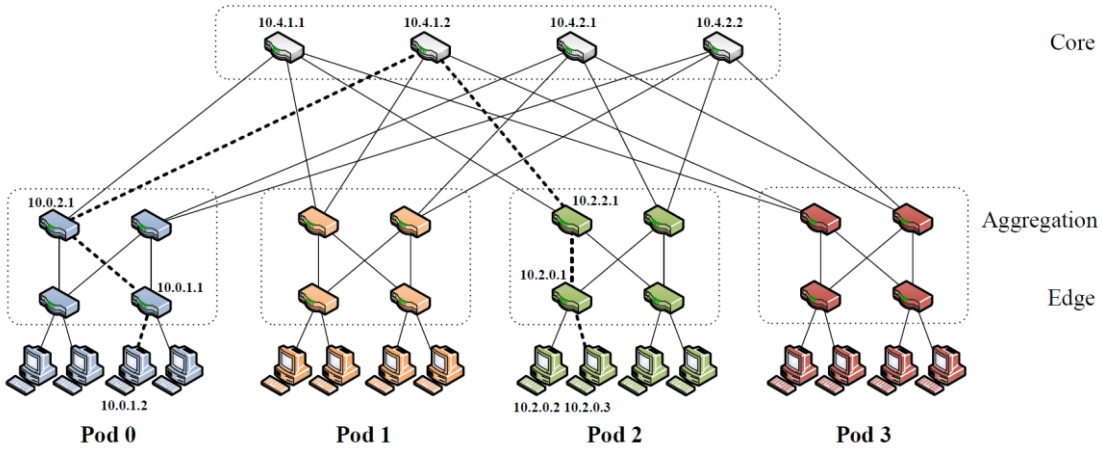

Figura 3.3: Topologia Fat Tree que foi utilizada para desenvolvimento/teste dos mecanismos desenvolvidos

```
pods = [0,1,2,3]
edge_sw = [0,1]
agg_sw = [2,3]
core_sw = [1,2]
hosts = [2,3]
foreach p in pods  do
        foreach e in edge_sw do
                edge = 10.p.e.1
                ADICIONAR SWITCH edge
                foreach h in hosts  do
                                host = 10.p.e.h
                                ADICIONAR HOST host
                                ADICIONAR LIGAÇÃO edge,host
                end
                foreach a in agg_sw  do
                                agg = 10.p.a.1
                        ADICIONAR SWITCH agg
                                ADICIONAR LIGAÇÃO agg,edge
                end
        end
        foreach a in agg_sw  do
                foreach c in core_sw  do
                                core = 10.4.(a-2)+1.1
                                ADICIONAR SWITCH core
                                ADICIONAR LIGAÇÃO agg,core
                end
        end
end
```

Figura 3.4: Pseudocódigo do algoritmo de criação da topologia fat tree

Na Figura 3.5 é apresentada uma outra representação da topologia usada. A largura de banda das 48 ligações assume o valor de 10 Mbit/s cada. Como se pode verificar na mesma figura, a atribuição dos endereços MAC segue o mesmo formato utilizado nos endereços IP, ou seja, 00:00:00:00:00: *pod*:*switch*:1 para os *switches* das camadas de agregação e de acesso; 00:00:00:00:00:4:j:i nos *switches* da camada do núcleo; e 00:00:00:*pod*:*switch*:ID para os *hosts*.

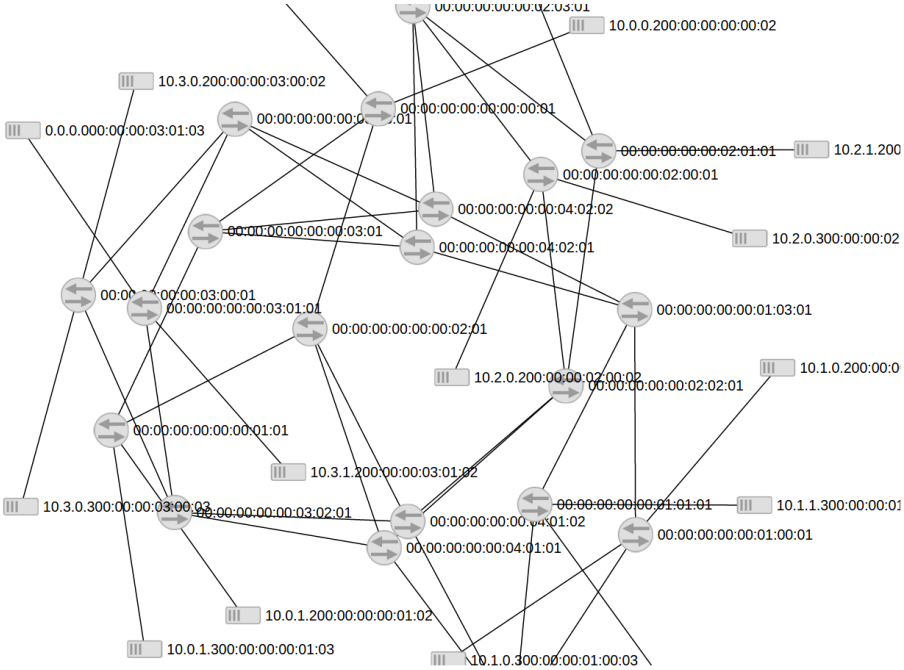

Figura 3.5: Topologia de rede

3.2 Mecanismos Implementados

Os mecanismos de balanceamento de carga implementados neste trabalho foram o *Round Robin*[2], o ECMP (versão [57]) e o *Hedera*. Estes foram estruturados em dois módulos, o primeiro para os mecanismos *Round Robin* e ECMP e o segundo para o *Hedera* e foram implementados em *Java* e instalados no controlador *Floodlight*. Esta estruturação deve-se ao facto do método responsável pelo encaminhamento no *Hedera* ser diferente do método utilizado nos outros dois mecanismos.

Após terem sido implementados todos os elementos da topologia de rede, nomeadamente o controlador, os dispositivos de encaminhamento e os *hosts*, bem como as respetivas ligações, surge a necessidade de definir qual será o comportamento do tráfego a circular na rede. Este comportamento é definido por um conjunto de classes e métodos que foram instalados nos *switches*, estando os principais identificados na Tabela 3.2.

Tabela 3.2: Alguns métodos implementados para o encaminhamento do tráfego

Módulo	Classe	Método	Descrição

[2] No documento designamos por *Round Robin* um mecanismo de escolha sequencial de interfaces de saída de entre aqueles que tem o mesmo custo para um determinado destino.

net.floodlightcontroller.multipathrouting	MultiPathForwarding	processPacketInMessage()	Processa a mensagem do pacote de chegada e reencaminha para o método adequado.
net.floodlightcontroller.multipathrouting	MultiPathForwarding	doDropFlow()	Descarta o pacote e o fluxo.
net.floodlightcontroller.multipathrouting	MultiPathForwarding	doForwardFlow()	Encaminha o pacote e o fluxo.
net.floodlightcontroller.multipathrouting	MultiPathForwarding	doFlood()	Encaminha o pacote para todas as suas interfaces.
net.floodlightcontroller.hedera	Hedera	doForwardFlow()	Encaminha o pacote e o fluxo.

O método *processPacketInMessage()*, cujo pseudocódigo é apresentado na Figura 3.6, consiste em processar o pacote de entrada, ou seja, em verificar se existe uma decisão a tomar sobre o *routing* dos pacotes e, em caso afirmativo, na decisão da ação a executar sobre esses pacotes. Como podemos verificar na mesma figura, a variável *decision* pode apresentar diferentes resultados, nomeadamente:

- *NONE* – nenhuma ação é executada, prosseguindo o processamento dos pacotes de entrada.
- *FORWARD_OR_FLOOD* – o pacote e o fluxo são encaminhados, no entanto, se o destino não é conhecido, é feito o *flood* do pacote para o *switch* de origem.
- *FORWARD* - o pacote e o fluxo são encaminhados, no entanto, se o destino não é conhecido, é iniciado uma *discovery action*, como por exemplo o protocolo ARP.
- *MULTICAST* – é realizado o *multicast* para todas as interfaces.
- *DROP* – o pacote e o fluxo são descartados.

```
processPacketInMessage(){
    if decision ≠ null then
        switch(decision)
        case NONE
            PROCESSAMENTO DO PACOTE DE CHEGADA
        case FORWARD_OR_FLOOD
        case FORWARD
            doForwardFlow()
            PROCESSAMENTO DO PACOTE DE CHEGADA
        case MULTICAST
            doFlood()
            PROCESSAMENTO DO PACOTE DE CHEGADA
        case DROP
```

```
                    doDropFlow()
                    PROCESSAMENTO DO PACOTE DE CHEGADA
        else
            if ethernet frame = BROADCAST or MULTICAST then
                doFlood()
            else
                doForwardFlow()
        PROCESSAMENTO DO PACOTE DE CHEGADA
}
```

Figura 3.6: Pseudocódigo do algoritmo do método processPacketInMessage()

Na Figura 3.7 é representado a forma como o pacote é encaminhado para o *switch* seguinte (método *doForwardFlow()*). Em primeiro lugar é verificado, através do cabeçalho do pacote, se o endereço destino é conhecido. Em caso negativo, é realizado o *flood*, ou seja, o pacote é encaminhado para todas as suas interfaces. Caso contrário, são testadas quatro condições, nomeadamente se o endereço da fonte é nulo, se a ilha onde se encontra a fonte é desconhecida, se o *switch* destino está numa ilha diferente ou na mesma interface. Em caso afirmativo nas duas primeiras condições e na última, o método retorna para o método anterior (processamento de pacotes de entrada). Em caso afirmativo na terceira condição, é realizado o *flood* antes de retornar para o método anterior. No entanto, se nenhuma das condições se verificarem: 1) são instaladas as rotas; 2) é aplicada a função de *hash*, se o mecanismo selecionado for o *Hedera*; 3) é aplicado o mecanismo e 4) o fluxo é enviado.

```
doForwardFlow(){
        if Endereço do destino != null then
            if Endereço do emissor == null then
                return
            if srcIsland == null then
                return
            if DESTINO ESTÁ NUMA ILHA DIFERENTE then
                doFlood()
                return
            if MESMA INTERFACE
                return
            INSTALAR ROTAS
            APLICAR HASH SOBRE fluxo (SE HEDERA SELECIONADO)
            APLICAR MECANISMO
            ENVIAR FLUXO
        else
            doFlood()
}
```

Figura 3.7: Pseudocódigo do algoritmo do método doForwardFlow()

3.2.1 ECMP

O mecanismo ECMP utilizado neste trabalho aplica uma função de *hash* sobre os fluxos da rede. O funcionamento deste mecanismo é apresentado na Figura 3.8 e consiste no cálculo de um valor de *hash* obtido através de um conjunto de 5 campos existentes no cabeçalho do pacote, nomeadamente o endereço de origem e destino, a interface de rede de origem e destino e o tipo de pacote. De seguida, utilizando a operação de resto da divisão inteira, é calculado o módulo desse *hash* com o número de rotas possíveis. O resultado desta operação permite mapear cada fluxo para uma rota específica.

Para além de permitir uma distribuição mais eficiente do tráfego na rede, este mecanismo também contribui para que os pacotes de um fluxo cheguem ordenadamente ao destino, pois, todos os pacotes desse fluxo irão ser transmitidos pela mesma interface de rede.

```
getRouteHashThreshold(fluxo){
        INICIALIZAR rotas
        hash = fluxo.hashCode()
        índice_da_rota = hash % número_total_de_rotas
        return rotas.get(índice_da_rota)
}
```

Figura 3.8: Pseudocódigo do algoritmo do método getRouteHashThreshold()

Porém, este método apresenta-se pouco eficiente quando dois ou mais fluxos de grandes dimensões apresentam o mesmo valor de *hash*. Nesta situação os dois fluxos são enviados pela mesma interface de rede, podendo originar a congestão na rede. Isto acontece, pois o ECMP baseado em *hash* não tem conhecimento acerca do estado da rede e do tamanho dos fluxos, podendo resultar em colisões, sobrecarga nos *buffers* e na degradação das ligações. Este mecanismo pode ser melhorado complementando-o com um mecanismo designado por *Hedera*.

3.2.2 Hedera

O *Hedera* foi implementado de forma a resolver as limitações do ECMP, nomeadamente no que concerne aos *elephant flows* referidos anteriormente. Como foi referido no estado da arte, através dos métodos que o constitui, este mecanismo permite obter uma visão global sobre a rede. Um *switch* que esteja habilitado com este mecanismo executa o ECMP, com maior eficiência, para o encaminhamento dos fluxos. Enquanto o encaminhamento dos fluxos é realizado pelo ECMP, o *Hedera* executa em ciclo os seguintes passos: o primeiro consiste em detetar os fluxos de grandes dimensões nos *switches* da camada de acesso. Este passo é

conseguido através da realização da estimativa dos requisitos de tráfego dos fluxos, ou seja, é calculada a largura de banda requerida pelo fluxo. Se esta for igual ou superior a 10% da largura de banda da ligação será considerado *elephant flow*. Uma vez identificados os fluxos de grandes dimensões, o próximo passo consiste na construção de caminhos alternativos para esses fluxos através do algoritmo *Global Best Fit*. Por último, os novos caminhos são instalados nos respetivos *switches*. O valor limite (*threshold*) da largura de banda da ligação utilizado para considerar um fluxo como *elephant flow* é calculado através da utilização de um ciclo executado de cinco em cinco segundos que consiste em armazenar um conjunto de estatísticas nos dispositivos de encaminhamento. A Tabela 3.3 identifica as principais classes e métodos que foram implementados no módulo referente ao mecanismo *Hedera*.

Tabela 3.3: Métodos implementados para deteção de elephant flows e atribuição de novas rotas

Classe	Método	Descrição
StatsRequest	flowStatRequest()	Permite a análise estatística dos fluxos
TrafficEstimation	estimateDemands()	Calcula os requisitos de tráfego dos fluxos e realiza a convergência dos mesmos.
TrafficEstimation	estSrc()	procede ao aumento das capacidades dos fluxos no emissor.
TrafficEstimation	estDst()	procede ao aumento das capacidades dos fluxos no recetor.
StatsRequest	bestFit()	Escolhe a rota indicada para o *elephant flow*.

O método da Figura 3.9 apresenta a *thread* responsável pela sondagem dos *switches* e é constituída por 2 ciclos. O primeiro consiste simplesmente em realizar um compasso de espera que tem como objetivo garantir que o número de *hosts* estabilize. O segundo ciclo vai ser executado de 5 em 5 segundos e é responsável por obter uma lista de todos os *switches* da camada de acesso, assim como um contador correspondente ao número de *switches* da lista, e passá-los um por um para o método da Figura 3.10.

```
run(){
        do
                nhosts = knownHosts
                Thread.sleep(4000);
        while nhosts != knownHosts || nhosts == 0
        while(true)
                INICIALIZAR edgeSwitches
                if EXISTE SWITCH EDGE then
                        numero_de_switches = edgeSwitches.size()
```

```
                    while(EXISTE SWITCHES EDGE)
                        numero_de_switches --
                        flowStatRequest(edgeSwitches, numero_de_switches)
                Thread.sleep(5000)
}
```

Figura 3.9: Pseudocódigo do algoritmo da thread executada de 5 em 5 segundos

A Figura 3.10 representa o método que possibilita a coleção de um conjunto de estatísticas nos *switches* da camada de acesso e que possibilita a deteção de *elephant flows*. Inicialmente é verificado se os *switches* estão conectados ao controlador, pois, apenas estes serão sondados. Feito isto, são coletadas, e guardadas numa lista, as informações existentes nas entradas de fluxos, nomeadamente, os endereços MAC e IP, assim como as interfaces de rede, dos fluxos de origem e destino, o tipo Ethernet e o requisito de tráfego do fluxo. Após todos os *switches* terem sido sondados, é analisada a lista correspondente a todos os fluxos e são testadas três condições: 1) se existem fluxos desconhecidos, ou seja, novos fluxos e, sendo o caso, o requisito de tráfego do (novo) fluxo é adicionado à lista; 2) se o requisito de tráfego de um fluxo (conhecido) aumentou e, em caso afirmativo, é atualizado ou 3) se o fluxo já não existe na rede e, se se verificar, o fluxo é removido da lista. De seguida, a lista dos fluxos e dos *hosts* são passados para o método da Figura 3.12 para o cálculo da estimativa dos requisitos de tráfego dos fluxos. Por último são analisados os requisitos de tráfego de cada fluxo e se se tratar de um *elephant flow*, uma nova rota é implementada e instalada pelo método da Figura 3.15.

```
flowStatRequest(edgeSwitches, numero_de_switches){
        if SWITCH CONETADO then
                ADICIONAR À LISTA DE SWITCHES A ANALISAR
        INICIALIZAR LISTA DE ESTATÍSTICAS
        while LISTA DE ESTATÍSTICAS NÃO CHEGAR AO FIM
                INICIALIZAR LISTA DE ENTRADAS DE FLUXOS
                foreach entrada in entradas do
                        OBTER INFOS DE entrada
                        INICIALIZAR fluxo
                        INTRODUZIR INFOS EM fluxo
                end
        if numero_de_switches == 0 then
                INICIALIZAR LISTA DE FLUXOS
                foreach fluxo in fluxos do
                        if fluxo DESCONHECIDO then
                                PEDIDO DE fluxo ADICIONADO
                        else
                                if PEDIDO DE fluxo AUMENTOU  then
                                        ATUALIZAR PEDIDO
                end
```

```
            foreach fluxos in fluxos do
                if fluxo JÁ NÃO EXISTE then
                    REMOVER fluxo DA LISTA DE FLUXOS
            end
            estimateDemands(fluxos, hosts)
            foreach fluxo in fluxos do
                OBTER PEDIDO DE fluxo
                if É ELEPHANT FLOW then
                    nova_rota = bestFit(fluxo, pedido)
                    if nova_rota ≠ null then
                        ATUALIZAR LARGURA DE BANDA
                        INSTALAR ROTA
            end
}
```

Figura 3.10: Pseudocódigo do algoritmo do método flowStatRequest()

Estimativa dos requisitos de tráfego dos fluxos

Com o objetivo de detetar *elephant flows* e posteriormente tomar decisões adequadas acerca da atribuição dos novos caminhos para estes fluxos, é necessário fazer uma estimativa sobre os requisitos de tráfego dos fluxos. Ou seja, calcular adequadamente a largura de banda máxima e mínima de cada fluxo para posteriores colocações, e se estes estão limitados apenas pela NIC do emissor ou do recetor.

Essa estimativa é realizada através de uma matriz M $N \times N$ em que N corresponde ao número de *hosts*. Cada elemento da matriz contém 3 valores, nomeadamente o número de fluxos do *host* i para o *host* j (i e j correspondem ao número da linha e coluna, respetivamente); o requisito de tráfego de cada fluxo do *host* i para o *host* j; e uma *flag* destacando os fluxos cujos requisitos de tráfego convergiram. Sobre essa matriz é realizado um conjunto repetitivo de iterações que procedem ao aumento das capacidades dos fluxos no emissor e diminuição nos recetores até todos os fluxos terem convergido. A Figura 3.11 mostra o processo do cálculo estimativo dos requisitos de tráfego dos fluxos considerando o exemplo em [26]: 4 *hosts* (H0, H1, H2, H3) interligados através de uma topologia *non-blocking*. H0 envia 1 fluxo para os restantes; H1 envia 2 fluxos para H0 e 1 fluxo para H2; H2 envia 1 fluxo para H0 e H3; e H3 envia 2 fluxos para H1. As matrizes indicam os requisitos de tráfego dos fluxos durante as várias iterações do algoritmo. Os parêntesis indicam se o fluxo ainda está a convergir (parêntesis curvos) ou se já convergiu (parêntesis retos). O valor que se encontra fora dos parêntesis indica a quantidade de fluxos do *host* (da linha) para o *host* (da coluna) e o requisito de tráfego por fluxo é representado como uma fração da largura de banda do NIC. A última matriz indica os requisitos de tráfego finais dos fluxos.

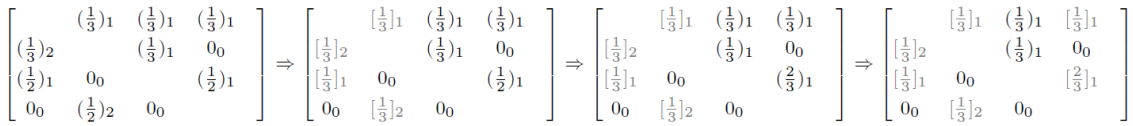

Figura 3.11: Processo do cálculo estimativo dos requisitos de tráfego dos fluxos

O cálculo da estimativa dos requisitos de tráfego dos fluxos é realizado pelo método da Figura 3.12. Este começa por inicializar a matriz M referida anteriormente e reencaminha para os dois métodos das Figuras 3.13 e 3.14 a matriz, os fluxos e os *hosts* emissor e recetor. No final, a matriz é atualizada se os requisitos de tráfego dos fluxos sofreram alterações.

```
estimateDemands(fluxos, hosts)
        INICIALIZAR MATRIZ M
        while true do
                foreach emissor in hosts do
                        estSrc(M, fluxos, emissor);
                end
                foreach recetor in hosts do
                        estDst(M, fluxos, recetor);
                end
                foreach hosti in hosts do
                        foreach hostj in hosts do
                                if (hosti, antigo_pedido_hostj) ≠ (hosti, pedido_hostj) then
                                        flag = false
                                        ATUALIZAR pedido_hostj
                        end
                end
                if flag = true then
                        break
        return M
```

Figura 3.12: Pseudocódigo do algoritmo do método estimateDemands()

Como foi referido anteriormente, a estimativa dos requisitos de tráfego dos fluxos é conseguida através de um conjunto de iterações realizadas sobre a matriz M e que consistem no aumento das capacidades dos fluxos no emissor e diminuição no recetor. A Figura 3.13 apresenta o pseudocódigo do método que realiza o aumento das capacidades dos fluxos no emissor. O método começa por verificar se os fluxos convergiram e, em caso afirmativo, o pedido de convergência é atualizado. Caso contrário, o número de fluxos não convergidos é incrementado. Após todos os fluxos terem sido analisados e se nenhum fluxo não convergiu, o método retorna para o método anterior. Caso contrário, a variável correspondente à largura de banda a partilhar

é atualizada. No final os fluxos são novamente analisados e a matriz é atualizada e o requisito de tráfego do fluxo alterado para o valor da largura de banda calculado se o fluxo não convergiu.

```
estSrc (M, fluxos, emissor)
        pedido_de_convergência = 0.0
        número_fluxos_não_convergidos = 0
        partilha_igual = 0.0
        foreach fluxo in fluxos do
                if ENDEREÇO EMISSOR DE fluxo = emissor then
                        if fluxo ESTÁ CONVERGIDO then
                                pedido_de_convergência += pedido_do_fluxo
                        else
                                número_fluxos_não_convergidos += 1
        end
        if número_fluxos_não_convergidos = 0 then
                return
        partilha_igual = (1 - pedido_de_convergência) / número_fluxos_não_convergidos
        foreach fluxo in fluxos do
                if ENDEREÇO EMISSOR DE fluxo = emissor then
                        if fluxo NÃO ESTÁ CONVERGIDO then
                                M É ATUALIZADO
                                pedido_do_fluxo É ALTERADO PARA partilha_igual
        end
```

Figura 3.13: Pseudocódigo do algoritmo do método estSrc()

A Figura 3.14 apresenta o pseudocódigo do método que realiza a diminuição das capacidades dos fluxos no recetor. Este começa por analisar fluxo a fluxo, com o mesmo endereço destino, se convergiram e, em caso afirmativo, adiciona o requisito de tráfego de cada fluxo ao requisito de tráfego total dos fluxos. No final do ciclo se o requisito de tráfego total dos fluxos for inferior à capacidade total da ligação, o método retorna para o método anterior. Caso contrário, é calculado o valor da largura de banda que será partilhada pelo fluxos. Feito isto, é verificado se os fluxos estão limitados no recetor. Em caso afirmativo, são comparados os valores dos requisitos de tráfego de cada fluxo e o valor da largura de banda a partilhar. Se o primeiro for inferior, o requisito de tráfego do fluxo é adicionado ao valor correspondente ao requisito de tráfego limite no recetor. Senão, o número de fluxos limitados no recetor é incrementado. No final a matriz e os fluxos são atualizados com o novo valor da largura de banda a ser partilhada.

```
estDst(M, fluxos, recetor)
        pedido_total = 0.0
        pedido_limite_emissor = 0.0
        número_fluxos_limitados_recetor = 0
        partilha_igual = 0
```

```
        foreach fluxo in fluxos do
                if ENDEREÇO RECETOR DE fluxo = recetor then
                        if fluxo ESTÁ CONVERGIDO then
                                fluxo LIMITADO NO RECETOR
                                pedido_total += pedido_do_fluxo
                                número_fluxos_limitados_recetor += 1
        end
        if pedido_total <= 1 then
                return
        partilha_igual = 1 / número_fluxos_limitados_recetor
        flag = true
        while flag do
                flag = false
                número_fluxos_limitados_recetor = 0
                foreach fluxo in fluxos do
                        if ENDEREÇO RECETOR DE fluxo ≠ recetor then
                                continue
                        if fluxo LIMITADO NO RECETOR then
                                if pedido_do_fluxo < partilha_igual then
                                        pedido_limite_emissor += pedido_do_fluxo
                                        fluxo NÃO LIMITADO NO RECETOR
                                        flag = true
                                else
                                        número_fluxos_limitados_recetor += 1
                end
                partilha_igual = (1 - pedido_limite_emissor) /
número_fluxos_limitados_recetor
        foreach fluxo in fluxos do
                if ENDEREÇO RECETOR DE fluxo ≠ recetor then
                        continue
                if fluxo LIMITADO NO RECETOR then
                        M É ATUALIZADA COM partilha_igual
                        M CONVERGIU
                        fluxo CONVERGIU
                        fluxo É ATUALIZADO COM partilha_igual
        end
```

Figura 3.14: Pseudocódigo do algoritmo do método estDst()

Algoritmo Global Best Fit

Após obter-se conhecimento sobre os requisitos de tráfego finais dos fluxos e terem sido identificados os *elephants flows*, é necessário a construção de rotas, com adequada largura de banda, para os alocar. O algoritmo que foi desenvolvido é o *Global Best Fit* e consiste em atribuir aos fluxos os caminhos que têm maior capacidade disponível considerando as reservas existentes. O algoritmo é apresentado na Figura 3.15 e começa por analisar um a um os nós que constituem o caminho. Nesse ciclo são testadas 4 condições: se a ligação é desconhecida, se faz parte do caminho já utilizado, se não tem espaço para o fluxo ou se a ligação está disponível

para alocar o fluxo. Caso a primeira condição se verifique, a ligação será adicionada. Na segunda condição, o requisito de tráfego do fluxo será retirado da ligação, uma vez que o fluxo será colocado noutra. O ciclo testa o próximo nó se a terceira condição se verificar. Na última condição é calculada a capacidade restante se o fluxo for alocado e é atualizado o valor correspondente à capacidade disponível. No final será atribuído ao fluxo o caminho que têm maior capacidade disponível.

```
bestFit(fluxo, pedido_fluxo)
        melhor_rota = null
        capacidade_máx = 0.0
        INICIALIZAR ROTAS
        foreach rota in rotas do
                caminho = CAMINHO DA rota
                capacidade_disp = 1.0
                nó_anterior = null
                cabe = true
                INICIALIZAR CAMINHOS
                foreach nó in caminho do
                        if nó_anterior ≠ null then
                                if LIGAÇÃO É DESCONHECIDA then
                                        LIGAÇÃO É ADICIONADA COM reserva = 0
                                uso_ligação É ATUALIZADO COM reserva
                                if LIGAÇÃO FAZ PARTE DO caminho_antigo then
                                        uso_ligação = uso_ligação - pedido_fluxo
                                if LIGAÇÃO NÃO TEM CAPACIDADE then
                                        cabe = false
                                        break
                                else
                                        d = 1 - uso_ligação + pedido_fluxo
                                        if d < capacidade_disp then
                                                capacidade_disp = d
                        nó_anterior = nó
                end
                if cabe = true && capacidade_disp >= capacidade_máx then
                        melhor_rota = rota
                        capacidade_máx = capacidade_disp
        end
        return melhor_rota
```

Figura 3.15: Pseudocódigo do algoritmo do método bestFit()

3.2.3 Round Robin

O mecanismo *Round Robin* utilizado neste trabalho consiste no mesmo princípio que o algoritmo de escalonamento *Round Robin* utilizado na execução de processos. Desta vez, o seu uso reside na escolha das rotas por onde os pacotes serão encaminhados.

O funcionamento deste mecanismo (ver Figura 3.16) apresenta-se relativamente simples, pois, a rota por onde o pacote será enviado é escolhida seguindo uma ordem. Quando um pacote chega a *switch*, este último consulta nas suas tabelas de entradas de fluxo as possíveis rotas por onde esse pacote pode ser encaminhado. Feito isto, o pacote é reencaminhado para a rota cujo índice corresponde ao índice exatamente a seguir ao da rota utilizada anteriormente, ou seja, a escolha da rota é feita sequencialmente. Semelhante ao mecanismo ECMP, o *Round Robin* não tem conhecimento sobre o estado das ligações. Com isto, um *switch* poderá eventualmente reencaminhar tráfego nas ligações que já se encontrem sobrecarregadas, resultando na degradação das mesmas.

```
getRouteRR()
        INICIALIZAR rotas
        índice_da_rota = (índice_da_rota + 1) % número_total_de_rotas
   return rotas.get(índice_da_rota)
```

Figura 3.16: Pseudocódigo do algoritmo do método getRouteRR()

3.3 Sumário

Este capítulo centrou-se na topologia de rede e nos mecanismos de balanceamento de carga que irão ser testados e avaliados. Inicialmente, foi feita uma contextualização do ambiente de emulação *Mininet* que irá servir de base para o estudo anteriormente referido, e do controlador *Floodlight*, nomeadamente a sua arquitetura, as aplicações e os serviços que oferece. De seguida, foi feita uma especificação dos mecanismos de balanceamento de carga, explicitando como foram implementados e o modo de funcionamento dos diversos métodos que os constituem.

No próximo capítulo irá ser abordado o cenário de testes, as metodologias usadas e a avaliação dos resultados obtidos.

4. TESTES E RESULTADOS

De modo a ser possível analisar os mecanismos de balanceamento de carga implementados, a topologia de rede foi submetida a vários testes, com o objetivo de obter resultados que possibilitem a avaliação dos mecanismos. O ambiente de testes usado consiste na injeção de tráfego com diferentes padrões de comunicação. Neste contexto, a secção 4.1 descreve os cenários de estudo que caracterizam os padrões de comunicação utilizados nos testes. De seguida, na secção 4.2, será dado relevo à metodologia de testes. Por último, na secção 4.3 serão apresentados os resultados obtidos, assim como a interpretação dos mesmos.

4.1 Cenários de Estudo

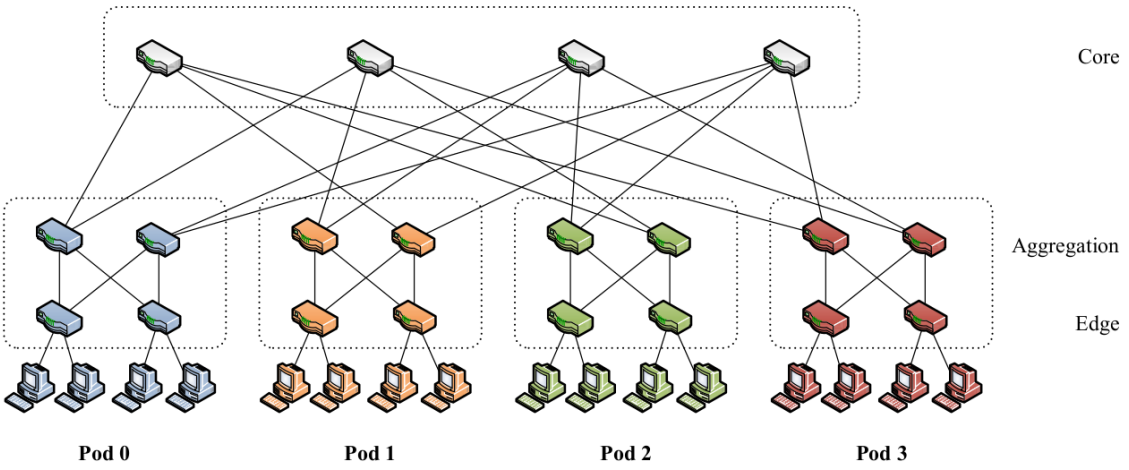

Figura 4.1: Fat tree

Esta secção tem como objetivo descrever os casos de estudos definidos para a comparação dos mecanismos de balanceamento de carga. Cada cenário retrata o envio de tráfego (maioritariamente *elephants flows*) entre os 16 *hosts* da topologia da Figura 4.1 e diferenciam-se pelos pares emissor/recetor. Neste trabalho são utilizados 20 cenários de teste, estando eles inseridos nos três padrões também utilizados em [26], nomeadamente:

- Stride(i) – Um *host* com índice x envia tráfego para o *host* com índice $(x + i) mod (número\ de\ hosts)$. Utilizaram-se quatro cenários distintos dentro deste padrão.

- Staggered Prob $(EdgeP, PodP)$ – Um *host* envia para outro *host* cujo *switch* da camada de acesso é partilhado por ambos com probabilidade $EdgeP$, do mesmo *pod* com probabilidade $PodP$, e para o resto da rede com probabilidade $1 - EdgeP - PodP$. Utilizaram-se seis cenários distintos dentro deste padrão.
- Random – Um *host* envia tráfego para qualquer outro *host* na rede com probabilidade uniforme. Utilizaram-se dez cenários distintos dentro deste padrão.

As secções seguintes detalham estas três categorias de testes que foram efetuados.

4.1.1 Stride(i)

Stride1

Neste cenário cada *host* envia tráfego para o *host* que se encontra ao lado na topologia, resultando no seguinte: oito *hosts* enviam tráfego para outros oito sendo que os *hosts* que formam o par emissor/recetor partilham o mesmo *switch*; quatro enviam tráfego para outros quatro onde os *hosts* de cada par pertencem ao mesmo *pod*; e os restantes quatro *hosts* enviam tráfego para o resto da rede.

Stride2

Cada *host* envia tráfego para o *host* que se encontra a seguir ao seu vizinho na topologia. Não existe o envio de tráfego entre pares que partilham o mesmo *switch* da camada de acesso; oito *hosts* enviam tráfego para outros oito onde os *hosts* de cada par pertencem ao mesmo *pod*; e os restantes oito enviam tráfego para o resto da rede.

Stride4

No *stride4* todos os dezasseis *hosts* enviam tráfego para outros dezasseis sendo que todos os pares encontram-se em *pods* diferentes. Isto acontece porque cada recetor encontra-se no *pod* a seguir ao do emissor, ou seja, o primeiro *host* do *pod* 0 envia tráfego para o primeiro do *pod* 1, este último envia tráfego para o primeiro do *pod* 2 e assim sucessivamente.

Stride8

Semelhante ao cenário anterior, ou seja, aqui também todos os *hosts* enviam tráfego para o resto da rede. No entanto, a diferença está no recetor se encontrar no *pod* a seguir ao *pod* vizinho do *pod* onde se encontra o emissor, isto é, o primeiro *host* do *pod* 0 envia tráfego para o primeiro do *pod* 2, o primeiro do *pod* 1 para o primeiro do pod 3 e assim sucessivamente.

4.1.2 Staggered Prob (0.2,0.3)

Para este padrão de comunicação, a probabilidade escolhida para que um *host* envie tráfego para outro cujo *switch* da camada de acesso é partilhado é de 0.2 e de 0.3 para um *host* do mesmo *pod*. A escolha destes valores resulta em $16 \times 0.2 = 3$ *hosts* para o primeiro caso e $16 \times 0.3 = 5$ *hosts* para o segundo. Isto significa que os $16 \times (1 - 0.2 - 0.3) = 8$ *hosts* restantes irão enviar tráfego para o resto da rede. Foram utilizados 3 cenários deste tipo, com diferenças na atribuição dos pares emissor/recetor.

4.1.3 Staggered Prob (0.5,0.3)

O princípio deste padrão é idêntico ao anterior, no entanto a diferença está nas suas probabilidades. Nos três cenários utilizados para este tipo de comunicação são $16 \times 0.5 = 8$ *hosts* que enviam tráfego para outros tantos com o mesmo *switch* partilhado e $16 \times 0.3 = 5$ *hosts* que enviam tráfego para *hosts* que se encontram no mesmo *pod* (par emssor/recetor).

4.1.4 Random

Como foi referido anteriormente, neste grupo a atribuição dos pares emissor/recetor foi realizada de forma aleatória e com probabilidade uniforme. No entanto, utilizaram-se quatro tipos diferentes de cenários. O primeiro contém três cenários (*rand0*, *rand1* e *rand3*), em que a diferença mais uma vez está na atribuição dos pares, porém, um recetor tem a possibilidade de receber tráfego de dois ou mais emissores. O segundo contém três cenários (*randbij0*, *randbij1* e *randbij2*) e utiliza a mesma característica das funções injetivas, ou seja, um emissor envia a apenas um recetor e este recebe tráfego de apenas um só emissor. O terceiro contém igualmente três cenários: no *randx2* um *host* envia duas vezes para outros dois, com possibilidade destes serem os mesmos ou não; no *randx3* o emissor envia três vezes; e no *randx4* quatro vezes. Por fim, o quarto tipo de cenários com atribuição aleatória é representado apenas por um cenário. Este designa-se por *hotspot* e consiste na utilização de dois *hotspots* que podem receber tráfego de vários emissores.

A atribuição dos pares emissor/recetor destes cenários, assim como de outros que não foram utilizados neste trabalho, pode ser consultada através dos ficheiros que se encontram no *Bitbucket* [56] (ver

https://bitbucket.org/msharif/hedera/src/569360b9490f77207035f8c92851fdec6375943f/inputs/?at=master).

4.2 Metodologia de Testes

De forma a se poder comparar e avaliar os mecanismos de balanceamento de carga, foram feitos vários testes aos diferentes cenários anteriormente referidos e obtidos os resultados para análise. Os testes consistem na injeção de tráfego por parte dos *hosts* e com isto calcular a largura de banda de bissecção obtida através da largura de banda que é utilizada nas interfaces da topologia. A largura de banda de bissecção é a soma da largura de banda utilizada em todas as ligações que atravessam uma bissecção mínima (um corte que divide o grafo da topologia pela sua metade). Para tal recorreu-se ao gerador de tráfego *cluster_loadgen* [58] e ao monitor de tráfego *bwm-ng* [59] que nos permite listar os fluxos da rede através de um conjunto de métricas, nomeadamente:

- Unix Timestamp (*unix_timestamp*)
- Nome da interface (*iface_name*).
- Número de bytes de saída (*bytes_out*).
- Número de bytes de entrada (*bytes_in*).
- Número de bytes total (*bytes_total*).
- Número de pacotes de saída (*packets_out*).
- Número de pacotes de entrada (*packets_in*).
- Número de pacotes total (*packets_total*).
- Número de pacotes de saída com erros (*errors_out*).
- Número de pacotes de entrada com erros (*errors_in*).

Os testes são realizados com recurso a um script *Shell* que tem como execução o algoritmo apresentado na Figura 4.2.

```
INICIALIZAR FICHEIROS DE ENTRADA
foreach ficheiro in ficheiros_de_entrada do
        input_file = inputs/ficheiro
        mecanismo = fattree-roundrobin
        output_file = results/mecanismo/ficheiro
        EXECUTAR hedera.py
end
foreach ficheiro in ficheiros_de_entrada do
```

```
        input_file = inputs/ficheiro
        mecanismo = fattree-ecmp
        output_file = results/mecanismo/ficheiro
        EXECUTAR hedera.py
end
foreach ficheiro in ficheiros_de_entrada do
        input_file = inputs/ficheiro
        mecanismo = fattree-hedera
        output_file = results/mecanismo/ficheiro
        EXECUTAR hedera.py
end
```

Figura 4.2: Pseudocódigo do algoritmo do script run_experiment.sh

Como podemos visualizar no algoritmo da Figura 4.2 este começa por inicializar os ficheiros de entrada (ver Tabela 4.1). Estes ficheiros correspondem aos vinte cenários que serão testados. Estes cenários, que apenas diferem nos pares emissor/recetor, apresentam os seguintes campos:

- IP do emissor (*src_ip*).
- IP do recetor (*dst_ip*).
- Porta destino (*dst_port*).
- Tipo de tráfego em bytes (*type*).
- Número do fluxo (*seed*)
- Início da injeção do tráfego em segundos (*start_time*).
- Final da injeção do tráfego (*stop_time*).
- Tamanho do fluxo (*flow_size*) – quantidade de tráfego gerado pelo fluxo ao longo do período de simulação.

Tabela 4.1: Ficheiro exemplo de um cenário

src_ip	dst_ip	dst_port	type	seed	start_time	stop_time	flow_size
10.0.0.2	10.0.0.3	12345	TCP	1	0.0	60.0	100000000000
10.0.0.3	10.0.1.2	12345	TCP	2	0.0	60.0	100000000000
10.0.1.2	10.0.1.3	12345	TCP	3	0.0	60.0	100000000000
10.0.1.3	10.1.0.2	12345	TCP	4	0.0	60.0	100000000000
10.1.0.2	10.1.0.3	12345	TCP	5	0.0	60.0	100000000000
10.1.0.3	10.1.1.2	12345	TCP	6	0.0	60.0	100000000000
10.1.1.2	10.1.1.3	12345	TCP	7	0.0	60.0	100000000000
10.1.1.3	10.2.0.2	12345	TCP	8	0.0	60.0	100000000000
10.2.0.2	10.2.0.3	12345	TCP	9	0.0	60.0	100000000000
10.2.0.3	10.2.1.2	12345	TCP	10	0.0	60.0	100000000000
10.2.1.2	10.2.1.3	12345	TCP	11	0.0	60.0	100000000000
10.2.1.3	10.3.0.2	12345	TCP	12	0.0	60.0	100000000000
10.3.0.2	10.3.0.3	12345	TCP	13	0.0	60.0	100000000000
10.3.0.3	10.3.1.2	12345	TCP	14	0.0	60.0	100000000000

10.3.1.2	10.3.1.3	12345	TCP	15	0.0	60.0	100000000000
10.3.1.3	10.0.0.2	12345	TCP	16	0.0	60.0	100000000000

Como se pode verificar na Tabela 4.1 o tipo de tráfego escolhido neste trabalho foi o TCP devido à grande parte do tráfego que circula num centro de dados ser deste tipo. Nessa mesma tabela é ainda possível visualizar que o tamanho de cada fluxo está fixado nos 100 Gbits. No entanto, este valor apenas representa o valor limite para o tráfego total que é gerado pelos fluxos TCP, pois, uma vez que a largura de banda das ligações está fixada nos 10 Mbit/s, os fluxos não chegam a atingir esse valor. Este foi o método adotado para garantir que os fluxos representam *elephants flows*.

De seguida serão executados em ciclo os vinte cenários nos três mecanismos implementados. Em cada ciclo são criadas as diretorias dos ficheiros de entrada e de saída e será executada a script *hedera.py* passando-lhe os seguintes argumentos:

- O ficheiro correspondente ao cenário.
- A diretoria do ficheiro de saída.
- A fração de CPU necessária para alocar cada *host*.
- O tempo de execução do teste.
- O nome do mecanismo a utilizar.

A script *hedera.py* é responsável pela realização do teste num determinado cenário. Inicialmente começa por executar o controlador *Floodlight* com o mecanismo correspondente ao parâmetro recebido. De seguida é inicializada a topologia através da execução do script *DCTopo.py* e é atribuída a largura de banda nas ligações (10 Mbit/s) e a fração de CPU a cada *host* (que também foi recebido como parâmetro). Depois é realizada a ligação da topologia com o controlador e é então injetado o tráfego segundo as características acima mencionadas. O gerador de tráfego tem uma duração de execução de 50 segundos e gera um ficheiro de texto que é atualizado todos os segundos, com as informações referentes a todas as interfaces de rede existentes na topologia. Ao fim dos 50 segundos, o controlador é desligado e desconetado da topologia. A topologia e os processos relativos aos *switches* e *hosts* são terminados.
Como mostra a Figura 4.3 todo este processo repete-se 60 vezes (representando uma amostra), pois cada mecanismo é testado 20 vezes, uma para cada cenário. Neste trabalho foram realizadas 30 amostras, sendo depois calculada a média da largura de banda de bisseção destas 30 amostras.

Uma vez realizados os testes surge a necessidade de calcular a largura de banda de bissecção. No cálculo da largura de banda de bissecção são unicamente consideradas as taxas de transmissão recolhidas nas interfaces dos *switches* da camada de acesso. A topologia utiliza é uma *fat tree* contendo 48 ligações com 10 Mbit/s cada, em que 32 correspondem a ligações com os *switches* da camada de acesso. Isto resulta numa largura de banda de bissecção máxima de 160 Mbit/s, pois se se fizesse um corte sobre as ligações da camada de acesso apenas 16 ligações seriam abrangidas por esse corte.

A largura de banda da bissecção é calculada por uma script designado por *gen_plot.sh* que começa por inicializar os ficheiros referentes aos resultados obtidos pela injeção de tráfego nos vários cenários, declara a variável *rate* correspondente ao *output* que irá ser gerado no final da execução da script e executa a script *plot_rate.py*, escrita em *Python*, passando-lhe esses dois argumentos. A script *plot_rate.py* é responsável por analisar os resultados das interfaces que foram gravados nos ficheiros de texto, e no final originar um ficheiro de imagem (*rate.png*) mostrando um gráfico com a largura de banda de bissecção normalizada em todos os cenários nos três mecanismos. A script percorre linha a linha os ficheiros de texto e quando uma linha corresponde a uma interface de um *switch* da camada de acesso esta é guardada. Após terem sido percorridas todas as linhas do ficheiro e guardadas as que se referem a interfaces de interesse é calculada a taxa de transferência média de cada interface e posteriormente a largura de banda de bissecção normalizada. Esta última corresponde à divisão da média de todas as taxas de transferência média de todas as interfaces com a largura de banda de bissecção máxima (160 Mbit/s).

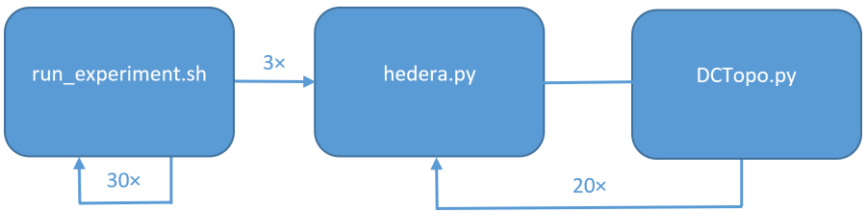

Figura 4.3: Processo de execução dos testes

Especificações do Ambiente de Emulação

A topologia de rede e os mecanismos de balanceamento de carga, assim como os testes realizados sobre eles foram realizados numa máquina virtual, disponível em [60], instalada com a ferramenta *VirtualBox Oracle* versão 5.0.6 numa máquina Toshiba Kira-102 com as seguintes

especificações: Processador *Intel ® Core ™* i7-4510U 2.00GHz, 8GB de RAM com o sistema operativo *Windows* 8.1 64bits.

A máquina virtual pré-configurada com o emulador *Mininet* versão 2.1, o *Integrated Development Environment* (IDE) Eclipse Standard/SDK versão *Kepler Service Release 2* e o controlador *Floodlight* versão 1.0 no sistema operativo *Linux Ubuntu* 14.04 64bits foi executada com 4GB de RAM e dois processadores.

4.3 Análise de Resultados

Esta secção tem como objetivo apresentar os resultados obtidos nos testes aos mecanismos de balanceamento de carga. Consequentemente será feita a comparação desses resultados nos três mecanismos e nos diferentes cenários utilizados.

4.3.1 ECMP

A Figura 4.4 apresenta os resultados obtidos através da utilização do mecanismo ECMP. Como podemos verificar a largura de banda de bissecção obtida (normalizada no intervalo [0,1]) encontra-se aproximadamente no intervalo [0.32, 0.57]. Começando pelos cenários com o padrão *Stride*, podemos afirmar que este mecanismo obteve melhores resultados nos cenários *stride1* (0.55) e *stride2* (0.55) quando comparados com o *stride4* e *stride8* (0.38 e 0.33 respetivamente). Isto acontece pois contrariamente ao *stride4* e *stride8*, nos dois primeiros cenários a comunicação é feita na maior parte das vezes entre *hosts* vizinhos, ou seja com *hosts* que partilham o mesmo *switch edge*, e/ou entre *hosts* do mesmo *pod*, e considerando que com o mecanismo ECMP os fluxos, com diferentes pares emissor/recetor, são distribuídos pelos vários caminhos mais curtos entre estes, e logo encaminhados para as respetivas interfaces de saída, as ligações não serão lotadas com um elevado número de fluxos diferentes evitando assim a partilha da largura de banda.

O mesmo acontece nos padrões de comunicação *Staggered Prob* (0.2,0.3) e *Staggered Prob* (0.5,0.3), ou seja, nos três cenários *Staggered Prob* (0.5,0.3) verificamos que apresentam melhores resultados relativamente aos outros três do *Staggered Prob* (0.2,0.3) devido ao maior número de trocas se efetuar entre *hosts* que partilham o mesmo *switch edge* e/ou entre *hosts* do mesmo *pod* do que com o resto da rede.

Relativamente aos cenários do padrão *Random* podemos destacar dois casos. O primeiro refere-se aos cenários em que um emissor envia tráfego para mais do que um recetor (*randx2*, *randx3*

e *randx4*). Nestes três cenários a largura de banda de bissecção apresenta valores satisfatórios tendo em conta as suas características, isto é, tendo em conta o elevado tráfego que circula para fora dos *pods*. Apesar de existir um maior número de fluxos diferentes, o facto de o tráfego ser distribuído por toda a rede, ou seja, utilizando um maior número de *switches* e ligações, o ECMP consegue fazer o encaminhamento dos fluxos de forma eficaz. O segundo refere-se aos cenários *randbij1* e *randbij2*. Estes dois cenários assemelham-se aos cenários *randbij0*, *rand0*, *rand1*, *rand2* e *hotspot*, pelo que se esperava resultados relativamente parecidos.

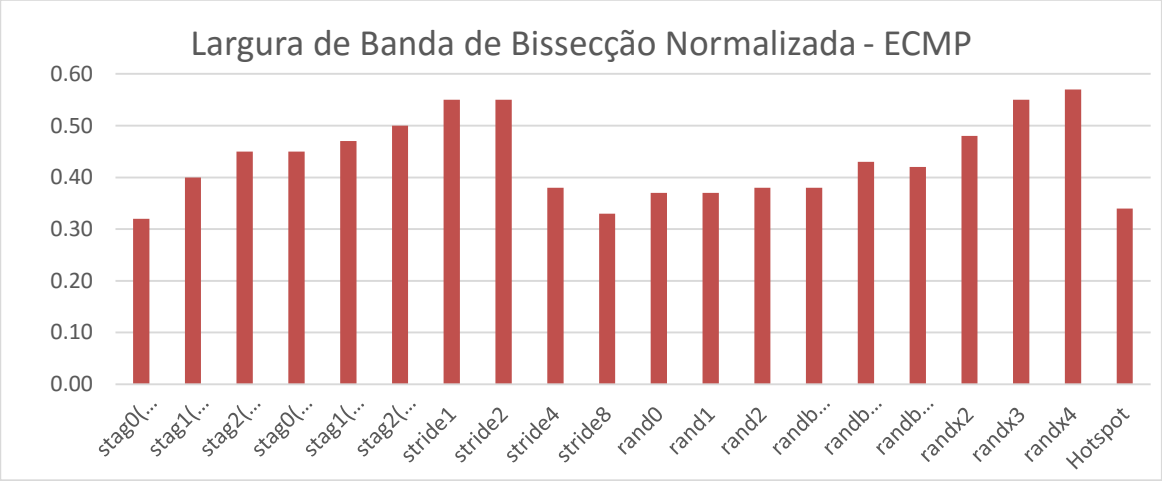

Figura 4.4: Largura de banda de bisseção normalizada obtida pelo mecanismo ECMP

4.3.2 Hedera

Como podemos verificar na Figura 4.5, os resultados obtidos nos diversos cenários situam-se no intervalo [0.34, 0.55]. Analisando os cenários correspondentes ao padrão de comunicação *Random* podemos concluir que a largura de banda de bissecção apresenta resultados aproximadamente constantes nos cenários onde cada *host* envia tráfego a apenas um só outro *host*. Nos cenários restantes obteve-se 0.47 no randx2 e valores superiores a 50% no *randx3* (0.52) e *randx4* (0.54).

Apesar dos três cenários do padrão *Staggered Prob* (0.2,0.3), bem como os três cenários do *Staggered Prob* (0.5,0.3), apresentarem as mesmas características, os resultados apresentaram valores ligeiramente diferentes. Enquanto no primeiro padrão obteve-se um intervalo de 0.07 entre o valor mais baixo (cenário *stag0*(0.2,0.3)) e o mais elevado (cenário *stag2*(0.2,0.3)), no segundo o intervalo obtido foi de 0.05.

Finalmente foi no padrão *Stride*, mais concretamente no cenário *stride1*, que se registou o valor mais elevado da largura de banda de bissecção normalizada (0.55). O *stride2* apresenta um

resultado ligeiramente inferior (0.53). Finalmente, o *stride4* e *stride8* apresentaram valores muito inferiores comparativamente aos dois anteriores com 0.37 e 0.34, respetivamente. Esta diferença de valores resulta na diferença no número de *hosts* que enviam tráfego para *hosts* do mesmo *pod* (*stride1* e *stride2*) e no número de *host* que envia tráfego para *hosts* de *pods* diferentes (*stride4* e *stride8*).

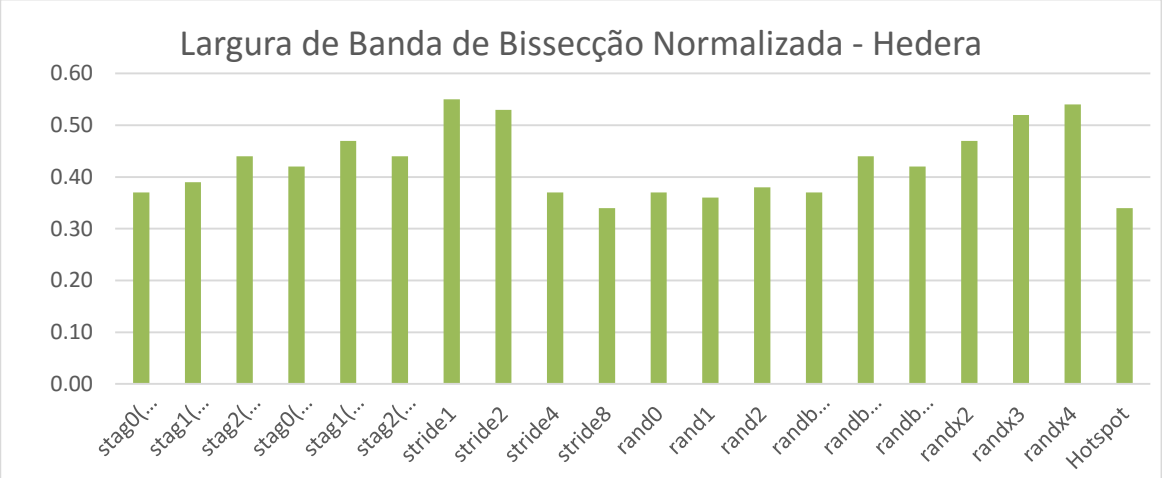

Figura 4.5: Largura de banda de bisseção normalizada obtida pelo mecanismo Hedera

4.3.3 Round Robin

Tal como nos mecanismos anteriores, e como podemos visualizar no Figura 4.6, o *Round Robin* apresenta pior largura de banda de bissecção nos cenários do padrão *Random* (exceto *randx2*, *randx3* e *randx4*) e nos cenários *stride4* e *stride8*, ou seja, onde o envio de tráfego é realizado em grande parte entre *hosts* de diferentes *pods* e onde cada emissor envia apenas uma única vez. Quase metade destes cenários apresentam valores abaixo dos 30% o que comprova que a escolha sequencial das interfaces de rede para o encaminhamento dos fluxos faz do *Round Robin* um mecanismo de balanceamento de carga pouco eficiente.

No entanto, este mecanismo superou as expectativas relativamente aos restantes cenários, com destaque para o *stride2* cujo valor atinge 72% da largura de banda de bisseção máxima.

4.3.4 Comparação de Resultados

Como podemos visualizar nas Figuras 4.7 e 4.8 os mecanismos apresentam resultados situados entre os 0.25 e os 0.75 e relativamente semelhantes em alguns cenários. Comparando os mecanismos podemos afirmar que o ECMP foi o mecanismo mais eficaz, pois obteve melhores

resultados em 14 cenários, seguido do *Round Robin* e o *Hedera* com 7 cenários cada, se incluirmos os mecanismos que registaram os mesmos valores.

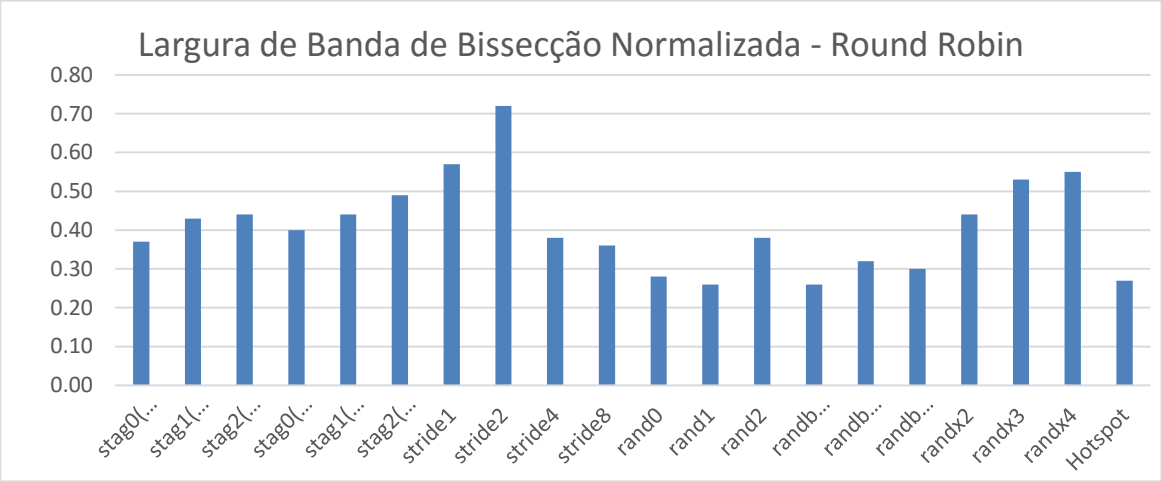

Figura 4.6: Largura de banda de bisseção normalizada obtida pelo mecanismo Round Robin

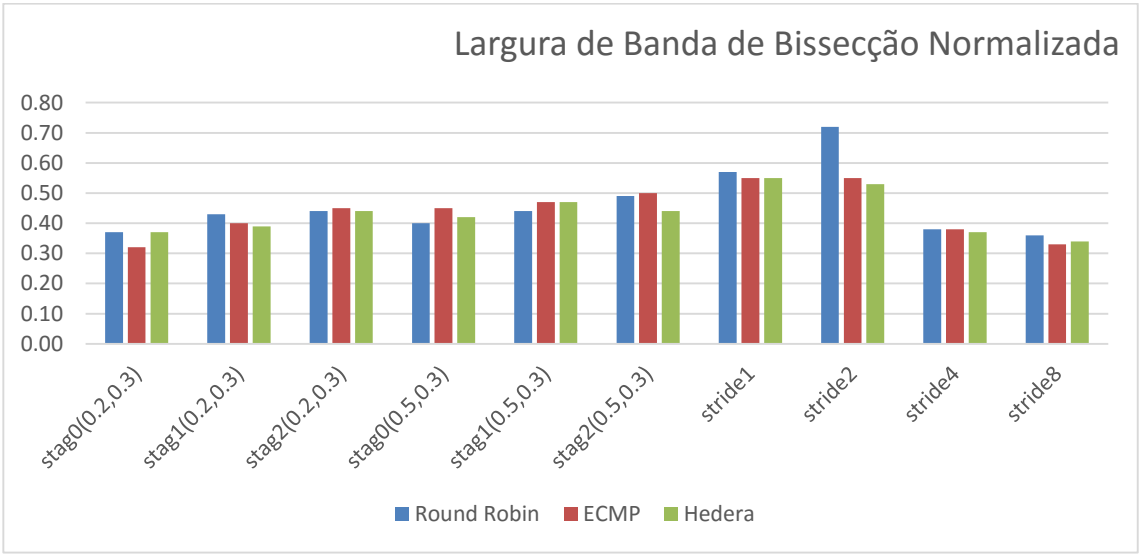

Figura 4.7: Largura de banda de bisseção normalizada obtida em todos os mecanismos

Relativamente aos cenários do padrão *Staggered Prob* (0.2,0.3), podemos verificar que o *Hedera*, em igualdade com o *Round Robin*, obteve melhor desempenho no *stag0(0.2,0.3)*, o *Round Robin* foi mais eficiente no *stag1(0.2,0.3)* e o ECMP no *stag2(0.2,0.3)*. A conclusão que se pode retirar acerca deste padrão é que a largura de banda de bissecção normalizada será aproximadamente de 0.4 seja qual for o método escolhido.

Quanto aos cenários do *Staggered Prob* (0.5,0.3) o ECMP supera os outros dois mecanismos, à exceção do *stag0(0.2,0.3)* onde registou o mesmo resultado que o *Hedera*.

Nos cenários *Stride*, obteve-se melhor largura de banda em todos os cenários quando o método utilizado foi o *Round Robin* (em igualdade com o ECMP no *stride4*). Foi ainda neste padrão, mais concretamente no cenário *stride2*, que se registou o resultado mais elevado com 0.72.

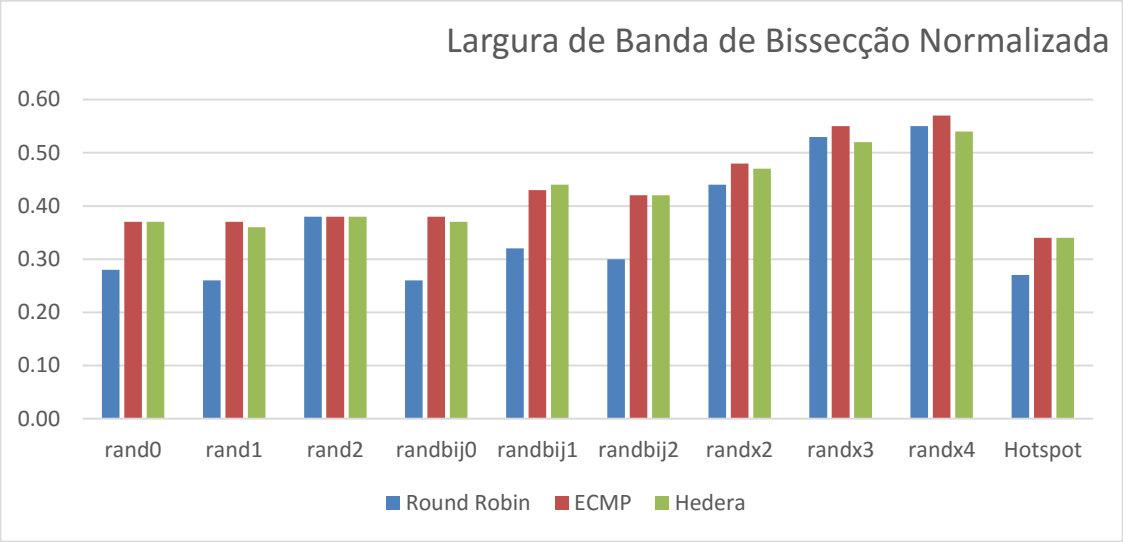

Figura 4.8: Largura de banda de bisseção normalizada obtida em todos os mecanismos

Nos cenários relativos ao padrão *Random* o ECMP apresenta resultados superiores na maior parte dos cenários, estando em alguns cenários em igualdade com o *Hedera* (no *rand0*, *randbij2* e *hotspot*) ou com o *Round Robin* (no cenário *rand2*).
Sendo o mecanismo *Hedera* um mecanismo que utiliza o ECMP adicionando inteligência no encaminhamento do tráfego, e uma vez que todos os fluxos da rede correspondiam a *elephants flows*, o resultado expectado seria obter valores superiores aos que foram obtidos no ECMP. Curiosamente obteve-se resultados ligeiramente inferiores (uma média de 0.02) em onze cenários, igual em seis cenários e uma melhoria, também de 0.02 nos restantes três.

Como forma de conclusão, segundo os resultados que foram obtidos nestes vinte cenários particulares podemos afirmar o seguinte:

- O ECMP aparenta ser o mecanismo de balanceamento de carga que melhor se adapta aos diferentes tipos de envio de tráfego.
- Para grandes quantidades de tráfego que circula entre diferentes *pods*, o ECMP apresenta ser o mecanismo mais indicado.
- O *Round Robin* é mais eficiente quando o tráfego da rede apresenta pouca densidade e quando a comunicação é feita entre *hosts* do mesmo *pod*. Em cenários do tipo *Random* a escolha deste mecanismo poderá não ser a mais adequada, pois, dos dez cenários deste

tipo, este mecanismo apresentou resultados inferiores aos do *Hedera* e ECMP em sete cenários.

- Embora implementado com esse objetivo, o *Hedera* não apresentou melhorias notórias face ao mecanismo ECMP. O facto da maior parte dos fluxos que foram injetados na rede apresentarem-se como *elephants flows* e sendo o *Hedera* um mecanismo com capacidade para tratar de forma diferencial fluxos particulares, poderá justificar os resultados obtidos.

Os resultados obtidos pelo *Hedera* foram um pouco surpreendentes, até porque os cenários em estudo são semelhantes aos usados pelos autores desse mecanismo. Nesse estudo o *Hedera* superava os mecanismos concorrentes [26]. Nesse sentido, foi efetuado uma pesquisa que pudesse corroborar os resultados obtidos neste trabalho.

A Figura 4.9 apresenta os resultados de um estudo realizado por um grupo de investigadores e que se encontra disponível em [61]. Nesse estudo, foram comparados os mecanismos ECMP e *Hedera* (com o algoritmo *Global First Fit* para a seleção de novas rotas). Os cenários testados foram os mesmos dos que foram utilizados neste trabalho, no entanto, o controlador escolhido foi o *POX* (controlador implementado em *Python*) e a largura de banda das ligações de 1Gbit/s cada. Analisando os resultados podemos verificar que também nesse estudo o mecanismo ECMP apresentou melhores resultados do que o *Hedera* na maior parte dos cenários. Comparando os resultados desse estudo com os que foram obtidos neste trabalho, podemos reconfirmar que nos cenários *stag0(0.2,0.3)*, *stag0(0.5,0.3)* e *stride8*, o *Hedera* obteve um desempenho superior ao ECMP.

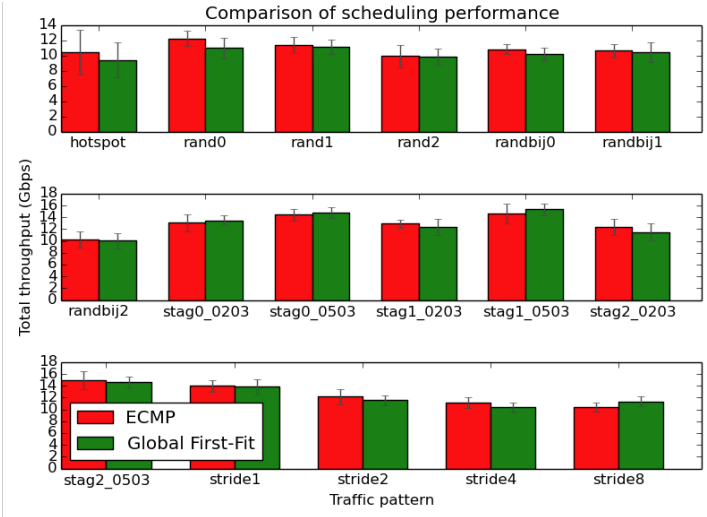

Figura 4.9: Largura de banda de bisseção normalizada obtida no estudo [61].

Como o cenário adicional pretende-se estudar agora o desempenho dos mecanismos quando na presença de fluxos bastante heterogéneos. Com o objetivo de testar os mecanismos de balanceamento de carga na presença dos dois tipos de fluxos, isto é, com fluxos considerados não *elephant flows* e com *elephant flows*, foi implementado o cenário *randx5* com as mesmas características dos cenários anteriores, porém com as seguintes diferenças:

- Cada *host* envia tráfego (TCP) para outros cinco;
- 60 fluxos têm um tamanho de 900 Kbits;
- 20 *elephant flows* têm um tamanho de 100 Gbits. Mais uma vez, este valor apenas representa o valor limite para o tamanho máximo dos fluxos, não sendo atingido no período de simulação.
- A atribuição dos pares emissor/recetor foi atribuída aleatoriamente e o ficheiro correspondente a este cenário pode ser consultado em anexo (ver Anexo 1).

A Figura 4.10 apresenta os resultados referentes ao cenário *randx5* acima descrito. Como podemos observar, o *Hedera* supera os outros mecanismos, pois, uma vez que existem agora fluxos particulares (*elephant flows*), o *Hedera* consegue superar o desempenho do mecanismo ECMP.
Em todo o caso, este estudo permitiu também concluir que serão necessários testes adicionais (e provavelmente a definição de novos cenários) para ser possível, com um grau de confiança elevado, diferenciar o desempenho dos mecanismos estudados. De igual forma, será importante investigar se possíveis limitações nas plataformas computacionais usadas poderão também justificar os resultados obtidos.

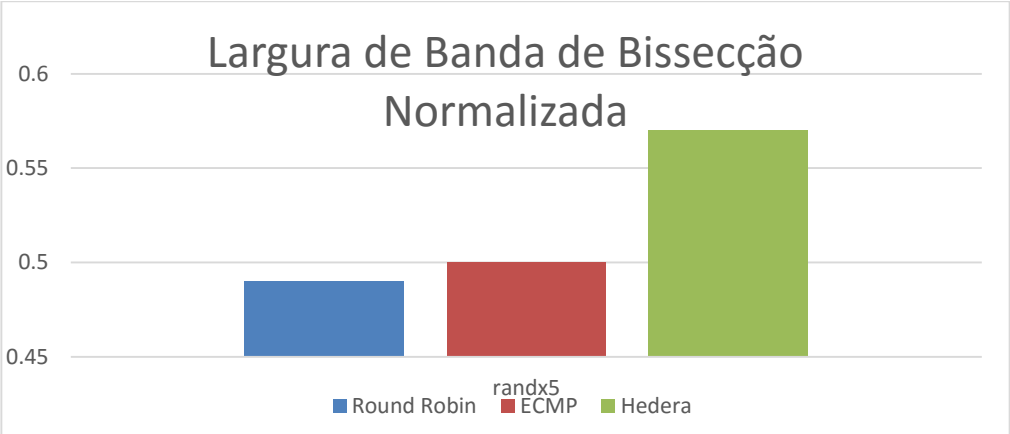

Figura 4.10: Largura de banda de bisseção normalizada obtida no cenário randx5

4.4 Sumário

Neste capítulo foram apresentados os cenários de estudo, abordada a metodologia usada nos testes e avaliados os resultados obtidos. De uma forma global, os três mecanismos de balanceamento de carga que foram testados neste trabalho apresentaram um desempenho abaixo do que seria esperado, pois a largura de banda de bisseção obtida encontra-se abaixo dos 50% na maior parte dos cenários. Esperava-se também que o *Hedera* apresentasse resultados bastantes superiores comparativamente aos outros dois mecanismos devido aos métodos que o caracteriza e que possibilita o encaminhamento mais inteligente do tráfego.

5. CONCLUSÕES

Ao longo da dissertação foi apresentado e explicado todo o trabalho feito, e é neste capítulo que irão ser abordadas as conclusões do trabalho desenvolvido. Será feita uma síntese de cada uma das etapas do trabalho, seguindo-se de uma conclusão dos resultados obtidos.

5.1 Resumo do Trabalho Desenvolvido

Por forma a ter uma base de conhecimento para poder implementar os mecanismos de balanceamento de carga selecionados, foi estudada a arquitetura das redes SDN, mecanismos de engenharia de tráfego, arquiteturas e topologias de redes de centros de dados e algumas métricas que as distinguem. Na arquitetura SDN foram abordados os planos que a integram, nomeadamente o de dados, de controlo e de gestão. Posteriormente, foram abordados tópicos de engenharia de tráfego em redes SDN, bem como mecanismos que otimizam o funcionamento dessas redes. Por último, foram analisadas algumas arquiteturas e topologias de rede, e um conjunto de métricas que as distinguem, utilizadas em centros de dados, inclusive a que foi adotada neste trabalho.

O desenvolvimento e especificação da bancada de trabalho envolveram as fases de emulação, bem como a especificação de mecanismos de balanceamento de carga. Na fase de emulação foi selecionado o emulador *Mininet*, que possibilitou a implementação de um ambiente de centro de dados através da virtualização dos componentes da rede. O controlador *Floodlight* foi selecionado, onde os mecanismos foram então implementados e testados na topologia de rede *fat tree*. Foi também explicado o funcionamento dos mecanismos *Hedera*, ECMP e *Round Robin* e as principais classes e métodos que estes utilizam.

Com o objetivo de analisar o desempenho dos mecanismos de balanceamento de carga, foram efetuados vários testes para obter e avaliar os resultados da largura de banda de bissecção que é utilizada nas ligações. Foi possível concluir que em alguns dos cenários os mecanismos oferecem resultados semelhantes, mas o ECMP foi o mecanismo que obteve melhores resultados em grande parte dos casos de estudo. Relativamente ao *Hedera* esperava-se, devido ao seu modo de funcionamento mais inteligente e aos seus métodos mais elaborados, que obtivesse melhores resultados do que os outros dois. Este acontecimento pode ser justificado através de dois aspetos: pelo facto da totalidade do tráfego da rede nos cenários considerados ser representado na maior parte das vezes por *elephant flows* e pelo *Hedera* ser um mecanismo eficaz na presença de fluxos heterogéneos. Concluindo, uma das possíveis justificações é como

na maior parte das experiências realizadas estamos na presença de fluxos de grandes dimensões e com tamanhos semelhantes, o *Hedera* não consegue otimizar o encaminhamento desses mesmos fluxos. No entanto, e como vimos na Figura 4.9, quando existe diversidade de fluxos, o *Hedera* apresenta-se mais eficiente relativamente aos mecanismos de balanceamento de carga ECMP e *Round Robin*.
Como foi referido anteriormente, a existência de limitações computacionais poderão também justificar os resultados obtidos.

5.2 Principais Contribuições

As contribuições desta dissertação são a especificação e desenvolvimento de uma bancada de testes, a realizar no âmbito das SDN, e a implementação e demonstração de três mecanismos de balanceamento de carga que podem ser utilizados nesse tipo de redes. Relativamente à bancada de testes, foi utilizado o emulador *Mininet*, pois, este permite-nos simular a rede de centro de dados; o controlador *Floodlight*, cujo objetivo é o de orquestrar os *switches* no encaminhamento dos fluxos na rede; e a topologia *fat tree* que caracteriza o centro de dados a simular e por onde esses fluxos circulam.
Quanto aos mecanismos de balanceamento de carga, uma vez que o crescente uso de aplicações que utilizam serviços na nuvem tem vindo a aumentar, torna-se importante a implementação de novos mecanismos deste tipo que melhor se adequam às arquiteturas de rede de centro de dados.

5.3 Trabalho Futuro

Relativamente aos mecanismos desenvolvidos, poderão ser efetuadas melhorias no ECMP e no *Hedera*, nomeadamente na função de *hash* no caso dos dois mecanismos, e na recolha de informação sobre os fluxos no *Hedera*. Em relação à primeira, a melhoria seria na adição de mais campos do cabeçalho do pacote para o cálculo do valor de *hash*. Esta melhoria diminuiria a probabilidade de dois pacotes de fluxos diferentes retornarem o mesmo valor de *hash*, o que tornaria os mecanismos mais robustos. Quanto à recolha de informação sobre os fluxos, uma possível melhoria seria uma metodologia diferente da que foi utilizada. A diminuição do tempo de *pooling* utilizado na recolha de informações pode também ser uma abordagem viável. Por fim, poderão novamente serem testados os mecanismos de balanceamento de carga elaborados nesta dissertação, desta vez em mais cenários em que existem diferentes fluxos, nomeadamente no seu tamanho, e/ou de diferentes tipos (por exemplo, fluxos UDP (*User Datagram Protocol*)).

BIBLIOGRAFIA

[1] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A comparative analysis of data center network architectures" *IEEE International Conference on Communication (ICC)*, pp. 3106–3111, 2014.

[2] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in software defined networks" *Computer Networks*, vol. 71, pp. 1–30, Oct. 2014.

[3] "OpenFlow." [Online]. Available: http://archive.openflow.org/wp/learnmore/.

[4] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks" *IEEE Communication Magazine*, vol. 51, no. 7, pp. 36–43, Jul. 2013.

[5] S. Agarwal, M. Kodialam, and T. V. Lakshman, "Traffic engineering in software defined networks" *Proceedings IEEE INFOCOM*, pp. 2211–2219, Apr. 2013.

[6] 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, p. 69, Mar. 2008.

[7] Mininet, "No Title." [Online]. Available: http://www.mininet.org.

[8] R. L. S. de Oliveira, C. M. Schweitzer, and A. A. Shinoda, "Using Mininet for emulation and prototyping Software-Defined Networks" *IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–6, 2014.

[9] D. Kreutz, F. M. V Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A comprehensive survey" *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015.

[10] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar, "FlowVisor: a network virtualization layer" *Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks, Tech. Rep.*, 2009.

[11] A. Al-Shabibi and M. De Leenheer, "OpenVirteX: A Network Hypervisor" *Open Network Summit*, 2014.

[12] Z. Bozakov and P. Papadimitriou, "AutoSlice: automated and scalable slicing for Software-Defined Networks" *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, ser. CoNEXT Student '12. New York, NY, USA: ACM, pp. 3–4, 2012.

[13] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, “On controller performance in software-defined networks” *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE’12. Berkeley, CA, USA: USENIX Association, pp. 10–10, 2012.

[14] Z. Cai, A. Cox, and E. T. S. Ng, “Maestro: A System for Scalable OpenFlow Control” Rice University, Tech. Rep., 2011.

[15] D. Erickson, “The beacon openflow controller” *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ser. HotSDN ’13. New York, NY, USA: ACM, pp.13-18, 2013.

[16] Floodlight, “Floodlight.” [Online]. Available: http://www.projectfloodlight.org/.

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

[18] OpenDayLight, “OpenDayLight” [Online]. Available: https://www.opendaylight.org/.

[19] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: a network programming language” in *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP ’11*, vol. 46, no. 9, p. 279, 2011.

[20] A. Voellmy, H. Kim, and N. Feamster, “Procera: a language for high-level reactive network control” in *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN ’12*, p. 43, 2012.

[21] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular SDN Programming with Pyretic” *USENIX ;login*, vol. 38, no. 5, October 2013.

[22] S. Gutz, A. Story, C. Schlesinger, and N. Foster, “Splendid isolation: a slice abstraction for software-defined networks” *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN ’12. New York, NY, USA: ACM, pp. 79–84, 2012.

[23] D. Turull, M. Hidell, and P. Sjodin, “Using libNetVirt to control the virtual network” in *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*, pp. 148–152, 2012.

[24] M. Bansal, J. Mehlman, S. Katti, and P. Levis, “OpenRadio: A programmable wireless dataplane” *HotSDN*, pp. 109–114, 2012.

[25] T. Koponen, K. Amidon, and P. Balland, “Network virtualization in multi-tenant data

centers” *Download3.Vmware.Com*, pp. 1–22, 2013.

[26] M. Al-Fares, S. Radhakrishnan, and B. Raghavan, “Hedera: dynamic flow scheduling for data center networks” *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI’10. Berkeley, CA, USA: USENIX Association, pp. 19–19, 2010.

[27] A. Tootoonchian, “HyperFlow : A Distributed Control Plane for OpenFlow” *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, ser. INM/WREN’10. Berkeley, CA, USA: USENIX Association, pp. 3–3, 2010.

[28] 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, p. 351, Aug. 2010.

[29] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, “Flexible , Wide-Area Storage for Distributed Systems with WheelFS” *Proceedings of the 6th USENIX symposium on Networked Systems Design and Implementation*, NSDI’09, vol. 9, pp. 43–58, April 2009.

[30] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Others, and S. Shenker, “Onix: A distributed control platform for large-scale production networks” *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI’10. Berkeley, CA, USA: USENIX Association, pp. 1–6, 2010.

[31] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, “BalanceFlow: Controller load balancing for OpenFlow networks” *IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, pp. 780–785, 2012.

[32] S. H. Yeganeh and Y. Ganjali, “Kandoo: a framework for efficient and scalable offloading of control applications” *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN ’12. New York, NY, USA: ACM, pp. 19–24, 2012.

[33] M. Luo, M. Luo, Y. Tian, Q. Li, J. Wang, and W. Chou, “SOX - A Generalized and Extensible S mart Network O penflow Controller (X)” *Proceedings of the First SDN World Congress*, Damsdadt, Germany, October 2012.

[34] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, “Enabling Fast Failure Recovery in OpenFlow Networks” *Proceedings of 8th International Workshop*

on the Design of Reliable Communication Networks, DRCN'11, pp. 164–171, October 2011.

[35] D. Staessens, S. Sharma, D. Colle, M. Pickavet, and P. Demeester, "Software defined networking: Meeting carrier grade requirements" *Proceedings of the 18th IEEE Workshop on Local & Metropolitan Area Networks*, LANMAN'11, pp. 1–6, October 2011.

[36] P. Fonseca, R. Bennesby, E. Mota, and A. Passito, "A replication component for resilient OpenFlow-based networking" *IEEE Network Operations and Management Symposium,* pp. 933–939, 2012.

[37] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "On reliability-optimized controller placement for Software-Defined Networks" *China Communications,* vol. 11, no. 2, pp. 38–54, February 2014.

[38] "Internet2 OS3E." [Online]. Available: https://www.internet2.edu/news/detail/4865/.

[39] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.

[40] T. Mizrahi and Y. Moses, "Time-based Updates in OpenFlow : A Proposed Extension to the OpenFlow Protocol" *Israel Institute of Technology, Technical Report*, CCIT Report, vol. 835, July 2013.

[41] T. Mizrahi and Y. Moses, "Time-based Updates in Software Defined Networks" *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN'3, pp. 163–164, August 2013.

[42] "NetFlow." [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[43] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: verifying network-wide invariants in real time" *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, p. 49, 2012.

[44] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE Way to Test OpenFlow Applications" *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, pp.127–140, April 2012.

[45] T. Wang, Z. Su, Y. Xia, and M. Hamdi, "Rethinking the Data Center Networking: Architecture, Network Protocols, and Resource Sharing" *IEEE Access*, vol. 2, pp. 1481–1496, 2014.

[46] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network

architecture" *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, p. 63, October 2008.

[47] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. (University of California. Vahdat, and R. N. Mysore, "PortLand: a scalable fault-tolerant layer 2 data center network fabric" *SIGCOMM '09 Proc. ACM SIGCOMM 2009 Conference Data Communication*, pp. 39–50, 2009.

[48] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree : Saving Energy in Data Center Networks" *Proc. 7th USENIX Conference Networked Systems Design and Implementation*, pp. 17–17, 2010.

[49] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: a cost-efficient topology for high-radix networks" *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 126, June 2007.

[50] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly, "Symbiotic routing in future data centers" *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, p. 51, 2010.

[51] C. Guo, G. Lu, D. Li, H. Wu, and X. Zhang, "BCube: a high performance, server-centric network architecture for modular data centers" *ACM SIGCOMM 2009 Conference Data Communication.*, pp. 63–74, 2009.

[52] P. Costa, A. Donnelly, G. O'Shea, and A. Rowstron, "CamCubeOS: A key-based network stack for 3D torus cluster topologies" *Proceedings 22nd International Symposium High-Performance Parallel Distributed Computing HPDC*, pp. 73–84, 2013.

[53] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnaswamy, and I. Stoica, "A Cost Comparison of Data Center Network Architectures" *Proceedings. 6th ACM Conext*, vol. 6, no. 1, 2010.

[54] "Mininet." [Online]. Available: http://mininet.org/overview/.

[55] "The New Stack." [Online]. Available: http://thenewstack.io/sdn-series-part-v-floodlight/.

[56] "Bitbucket." [Online]. Available: https://bitbucket.org/msharif/hedera.

[57] C. E. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm.", IETF RFC 2992, DOI 10.17487/RFC2992, November 2000, [Online]. Available: https://tools.ietf.org/html/rfc2992.

[58] S. Radhakrishnan, "cluster_loadgen." [Online]. Available: https://bitbucket.org/nikhilh/mininet_tests/src/9f051450a32a8411f03b7f6bbb99cb436c

5a4a73/hedera/hedera/.

[59] “bwm-ng (Bandwidth Monitor NG).” [Online]. Available: https://www.gropp.org/?id=projects&sub=bwm-ng.

[60] “SDN Hub.” [Online]. Available: http://sdnhub.org/tutorials/sdn-tutorial-vm/.

[61] “CS244 ‘15: Hedera Flow Scheduling.” [Online]. Available: https://reproducingnetworkresearch.wordpress.com/2015/05/31/cs244-15-hedera-flow-scheduling-draft/.

ANEXO I – FICHEIRO DO CENÁRIO RANDX5

src_ip	dst_ip	dst_port	type	seed	start_time	stop_time	flow_size
10.0.0.2	10.0.0.3	12345	TCP	1	0	60	900000
10.0.0.2	10.1.0.2	12345	TCP	2	0	60	100000000000
10.0.0.2	10.1.1.3	12345	TCP	3	0	60	900000
10.0.0.2	10.2.1.2	12345	TCP	4	0	60	100000000000
10.0.0.2	10.3.0.3	12345	TCP	5	0	60	900000
10.0.0.3	10.0.0.2	12345	TCP	6	0	60	900000
10.0.0.3	10.0.1.3	12345	TCP	7	0	60	900000
10.0.0.3	10.1.1.2	12345	TCP	8	0	60	900000
10.0.0.3	10.2.0.3	12345	TCP	9	0	60	900000
10.0.0.3	10.3.0.2	12345	TCP	10	0	60	100000000000
10.0.1.2	10.3.1.3	12345	TCP	11	0	60	900000
10.0.1.2	10.2.0.2	12345	TCP	12	0	60	900000
10.0.1.2	10.2.1.3	12345	TCP	13	0	60	100000000000
10.0.1.2	10.3.1.2	12345	TCP	14	0	60	900000
10.0.1.2	10.0.0.3	12345	TCP	15	0	60	900000
10.0.1.3	10.1.0.2	12345	TCP	16	0	60	900000
10.0.1.3	10.1.1.2	12345	TCP	17	0	60	900000
10.0.1.3	10.2.0.3	12345	TCP	18	0	60	900000
10.0.1.3	10.3.0.2	12345	TCP	19	0	60	100000000000
10.0.1.3	10.3.1.3	12345	TCP	20	0	60	900000
10.1.0.2	10.0.1.2	12345	TCP	21	0	60	100000000000
10.1.0.2	10.1.0.3	12345	TCP	22	0	60	900000
10.1.0.2	10.2.0.2	12345	TCP	23	0	60	900000
10.1.0.2	10.2.1.3	12345	TCP	24	0	60	900000
10.1.0.2	10.3.1.2	12345	TCP	25	0	60	100000000000
10.1.0.3	10.0.0.3	12345	TCP	26	0	60	900000
10.1.0.3	10.1.0.2	12345	TCP	27	0	60	900000
10.1.0.3	10.1.1.3	12345	TCP	28	0	60	900000
10.1.0.3	10.2.0.3	12345	TCP	29	0	60	100000000000
10.1.0.3	10.3.0.2	12345	TCP	30	0	60	900000
10.1.1.2	10.3.1.3	12345	TCP	31	0	60	900000
10.1.1.2	10.0.1.2	12345	TCP	32	0	60	900000
10.1.1.2	10.1.0.3	12345	TCP	33	0	60	900000
10.1.1.2	10.2.0.2	12345	TCP	34	0	60	900000
10.1.1.2	10.2.1.3	12345	TCP	35	0	60	100000000000
10.1.1.3	10.3.1.2	12345	TCP	36	0	60	900000
10.1.1.3	10.0.0.3	12345	TCP	37	0	60	100000000000
10.1.1.3	10.1.0.2	12345	TCP	38	0	60	900000
10.1.1.3	10.3.0.3	12345	TCP	39	0	60	900000
10.1.1.3	10.0.0.2	12345	TCP	40	0	60	900000
10.2.0.2	10.0.1.3	12345	TCP	41	0	60	900000
10.2.0.2	10.1.1.2	12345	TCP	42	0	60	100000000000

10.2.0.2	10.2.0.3	12345	TCP	43	0	60	900000
10.2.0.2	10.3.0.2	12345	TCP	44	0	60	900000
10.2.0.2	10.3.1.2	12345	TCP	45	0	60	900000
10.2.0.3	10.0.0.3	12345	TCP	46	0	60	900000
10.2.0.3	10.1.0.2	12345	TCP	47	0	60	100000000000
10.2.0.3	10.1.1.3	12345	TCP	48	0	60	900000
10.2.0.3	10.2.1.2	12345	TCP	49	0	60	900000
10.2.0.3	10.3.0.3	12345	TCP	50	0	60	900000
10.2.1.2	10.0.0.2	12345	TCP	51	0	60	900000
10.2.1.2	10.0.1.3	12345	TCP	52	0	60	100000000000
10.2.1.2	10.1.1.2	12345	TCP	53	0	60	900000
10.2.1.2	10.2.0.3	12345	TCP	54	0	60	900000
10.2.1.2	10.3.0.2	12345	TCP	55	0	60	100000000000
10.2.1.3	10.3.1.3	12345	TCP	56	0	60	100000000000
10.2.1.3	10.0.1.2	12345	TCP	57	0	60	900000
10.2.1.3	10.1.0.2	12345	TCP	58	0	60	900000
10.2.1.3	10.1.1.3	12345	TCP	59	0	60	900000
10.2.1.3	10.2.1.2	12345	TCP	60	0	60	900000
10.3.0.2	10.3.0.3	12345	TCP	61	0	60	900000
10.3.0.2	10.0.0.2	12345	TCP	62	0	60	900000
10.3.0.2	10.0.1.3	12345	TCP	63	0	60	900000
10.3.0.2	10.1.1.2	12345	TCP	64	0	60	900000
10.3.0.2	10.2.0.3	12345	TCP	65	0	60	100000000000
10.3.0.3	10.3.0.2	12345	TCP	66	0	60	900000
10.3.0.3	10.3.1.3	12345	TCP	67	0	60	900000
10.3.0.3	10.0.1.2	12345	TCP	68	0	60	100000000000
10.3.0.3	10.1.0.3	12345	TCP	69	0	60	100000000000
10.3.0.3	10.1.1.3	12345	TCP	70	0	60	900000
10.3.1.2	10.2.1.2	12345	TCP	71	0	60	900000
10.3.1.2	10.3.0.3	12345	TCP	72	0	60	900000
10.3.1.2	10.0.0.2	12345	TCP	73	0	60	900000
10.3.1.2	10.0.1.3	12345	TCP	74	0	60	100000000000
10.3.1.2	10.1.1.2	12345	TCP	75	0	60	900000
10.3.1.3	10.1.1.2	12345	TCP	76	0	60	100000000000
10.3.1.3	10.2.0.3	12345	TCP	77	0	60	900000
10.3.1.3	10.2.1.3	12345	TCP	78	0	60	900000
10.3.1.3	10.3.1.2	12345	TCP	79	0	60	900000
10.3.1.3	10.2.0.2	12345	TCP	80	0	60	900000