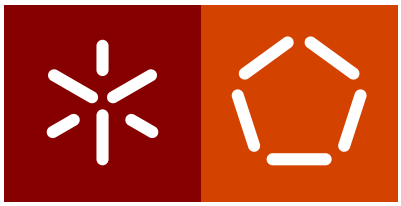
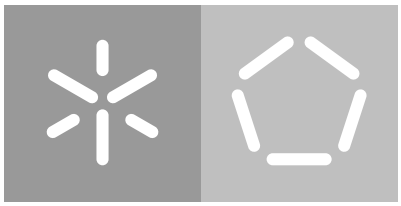


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Gonçalo Epifânio Pereira

Encaminhamento de Tráfego em Redes SDN (Software-Defined Networking)

Novembro 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Gonçalo Epifânio Pereira

Encaminhamento de Tráfego em Redes SDN (Software-Defined Networking)

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação sob a orientação de
Professor Pedro Nuno Miranda de Sousa

Novembro 2019

AGRADECIMENTOS

Esta dissertação de mestrado é o produto de um estudo árduo, intensivo e contínuo, na qual foram depositadas muitas horas de trabalho. Porém, e para que esta se tenha tornado realidade, necessitou dos demais pilares que a suportaram durante toda a sua execução.

Como tal, e em primeiro lugar, sem nunca esquecer a sua disponibilidade, quero agradecer ao professor Doutor Pedro Nuno Miranda de Sousa por toda a orientação dada ao longo deste percurso, assim como por todos os conhecimentos transmitidos que se demonstraram ser cruciais para a realização de todo este trabalho.

Seguidamente, agradeço também a toda a minha família - em particular aos meus pais, António e Isabel, à minha irmã, Filipa, e à minha namorada, Catarina, por todo o apoio, quer emocional quer motivacional, que sempre se revelaram ser a peça fundamental que sempre me guiou por todo este longo percurso, mesmo nos momentos cujo ânimo não estaria no expoente máximo.

Quero agradecer, também, a todos os meus amigos e colegas mais próximos, que me auxiliaram no debate de conhecimentos da área ou em qualquer outro aspeto ínfimo, que sempre se afirmaram disponíveis independentemente da situação.

Por último, agradeço a todos os que conheci durante todo este caminho percorrido e que fizeram de tudo para que fosse possível a conclusão desta etapa.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

Através do crescimento exponencial da utilização dos serviços *internet*, advém a inevitabilidade da criação de uma ferramenta que seja simultaneamente útil e versátil na gestão de todo o volume de tráfego criado.

É desta necessidade que surge o conceito de Software-Defined Networking (SDN), que visa oferecer um conjunto de protocolos e tecnologias, capazes de facilitar a gestão e a eficácia de manutenção das demais infraestruturas de rede que procurem a sua utilização.

Este trabalho visa a comparação inicial das várias capacidades dos demais controladores SDN existentes no mercado, enunciando as suas linguagens de programação e características.

Após tal comparação, será feita a escolha de um dos controladores SDN de forma a que seja desenvolvido um protótipo de uma solução de encaminhamento de tráfego, tirando o máximo de proveito das características do controlador e das abordagens SDN.

A solução desenvolvida deverá permitir uma plenitude de características, destacando-se a possibilidade de poder estar vocacionado para lidar com encaminhamento de tráfego consoante vários parâmetros. Inicialmente deverá incorporar o algoritmo de Dijkstra para cálculo de caminhos mais curtos e injeção de rotas, assim como deverá convergir de imediato após falhas de *links*. Seguidamente, deverá também ser propício a eventos que ocorram em tempo real, permitir a proteção de *links* ou *routers* da infraestrutura, reação eficaz após falhas de *links*, reação consoante diversos níveis de carga na rede, encaminhamento de fluxos de tráfego através de rotas específicas, multiplexagem da topologia por diferentes redes virtuais, entre outros. Deste modo, o protótipo desenvolvido surge, assim, como uma abordagem SDN alternativa a certos protocolos de encaminhamento interno conhecidos, tais como o Routing Information Protocol (RIP) e Open Shortest Path First (OSPF).

Palavras-Chave: SDN, encaminhamento, tráfego, dados, protocolos, *networking*, gestão de fluxo, infraestrutura

ABSTRACT

Through the exponential growth of the internet service usage comes the inevitability of the creation of a tool that can both be useful and versatile in the management of all of the created traffic volume.

From this necessity arises the concept of Software-Defined Networking (SDN), that aims to offer a set of protocols and technologies capable of easing the management and the efficiency of maintenance of the various network infrastructures that require its usage.

This work aims for the initial comparison of the multiple capacities of several SDN controllers existent in the market, specifying their programming languages and characteristics.

After such comparison follows the selection of one of the SDN controllers, in order for a prototype of a traffic forwarding solution to be developed, as to get the most out of the chosen controller's characteristics and of the global SDN approaches.

That same developed solution should allow for multiple characteristics, highlighting the possibility of its capacity to deal with traffic forwarding under specific parameters. Initially, the solution should incorporate the Dijkstra Algorithm to calculate the shortest paths and inject its results into the network, as it also should immediately converge after link failure. Then, it should also be conducive to events that occur in real time, it should allow for link or router protection in the infrastructure, effectively react to link failure, react to different load levels in the network, flow traffic forwarding through specific routes, topology multiplexing through different virtual networks, among others. That way, the developed prototype may come up as an alternative SDN approach to certain well-known interior routing protocols, such as Routing Information Protocol (RIP) and Open Shortest Path First (OSPF).

Keywords: SDN, forwarding, traffic, data, protocols, networking, flow management, infrastructure

CONTEÚDO

LISTA DE FIGURAS

LISTA DE ACRÓNIMOS

5G Fifth Generation (cellular network technology).

API Application Programming Interface.
ARP Address Resolution Protocol.
AS Autonomous System.

BGP Border Gateway Protocol.
BSN Big Switch Networks.

DDoS Distributed Denial of Service.
DoS Denial of Service.
DV Distance-Vector algorithms.

EGP Exterior Gateway Protocol.

IGP Interior Gateway Protocol.
IoT Internet-of-Things.
IP Internet Protocol.
ISP Internet Service Provider.

JAR Java Archive.
JVM Java Virtual Machine.

LLN Low Power and Lossy Network.
LS Link-State.
LTS Long Term Support.

MAC Media Access Control.

ODL OpenDayLight.
OF OpenFlow.
ONOS Open Networking Operating System.
OSPF Open Shortest Path First.

REST Representational State Transfer.
RIP Routing Information Protocol.

SDN Software-Defined Networking.

SSH Secure Shell.

TCP Transmission Control Protocol.
TELNET Teletype Network.
TLS Transport Layer Security.

UDP User Datagram Protocol.

VM Virtual Machine.

WAN Wide Area Network.

1

INTRODUÇÃO

1.1 ENQUADRAMENTO E MOTIVAÇÃO

Desde o início da estruturação de toda a temática que é o *networking* que os dispositivos se configuravam através de interfaces virtuais - quer configurados por Teletype Network (TELNET) ou Secure Shell (SSH) - mas sempre configurados manualmente por um administrador de rede. Esta configuração manual, estando inteiramente dependente do mesmo administrador de rede, pode sempre ser propícia a possíveis erros de configuração que, por sua vez, se podem revelar catastróficos. Sendo todo este processo redundante, repetitivo e, por vezes, duradouro, surge a necessidade de automatizar todo este processo.

Avançando já alguns anos, e ainda tendo em conta a mesma necessidade de automatizar o processo de configuração manual, surge o conceito de Software-Defined Networking (SDN) [14]. Este conceito aparece como globalmente suportado pelos maiores revendedores multinacionais e com extrema facilidade de manuseamento no que toca à sua implementação, característica de bastante relevância visto que outras soluções propostas anteriormente não teriam tido o melhor sucesso.

Com o aparecimento dessas abordagens, é possibilitada a separação clara entre os planos de controlo e de dados das infraestruturas de comunicação. Este procedimento possibilita processos mais flexíveis [24], ágeis e eficientes na configuração de equipamentos de rede (tais como *switches*, *routers*, entre outros), tirando, assim, proveito de uma entidade de controlo centralizada, denominada de controlador SDN [21].

Assim, e aliada ao mesmo contexto prévio, a área que engloba o encaminhamento de tráfego ganha cada vez mais tracção, muito graças às possíveis vantagens da possibilidade de se explorarem soluções SDN na mesma área, já que este tipo de soluções possibilita uma visão global, simples e eficaz sobre toda a rede, facilitando a manipulação dos demais componentes que constituem uma rede.

Deste modo, e sendo toda a temática das SDN uma área com muito por explorar [39], surge então a proposta deste trabalho, com o intuito de, inicialmente, se estudarem as capacidades básicas dos demais controladores SDN. Seguidamente, e destacando-se a potencialidade de encaminhamento de tráfego, este estudo deverá possibilitar o desenvolvi-

mento de um protótipo de uma proposta de encaminhamento de tráfego Internet Protocol (IP), fazendo uso da mesma visão global da rede fornecida pelos controladores SDN. Este protótipo poderá surgir como alternativa aos demais protocolos tradicionais de encaminhamento interno, afirmando-se como potencial solução inovadora nesta área.

1.2 OBJETIVOS

O objetivo deste projeto é o desenvolvimento de um protótipo de uma solução de encaminhamento de tráfego, tirando partido das abordagens SDN. O protótipo deverá permitir a (re)configuração dinâmica das estratégias de encaminhamento de tráfego, mediante diferentes objetivos e estratégias de reação a determinados eventos (e.g. permitir flexibilidade na escolha da métrica de encaminhamento a usar, reagir eficazmente a falhas de *links*, reatividade a determinados níveis de carga na rede, encaminhamento de determinados fluxos de tráfego por rotas com determinados requisitos, entre outros exemplos).

Para que se atinja este objetivo principal, foram definidas as seguintes etapas:

- Investigação e levantamento inicial bibliográfico sobre as várias áreas que são abrangidas neste trabalho, com o intuito de entender os principais conceitos SDN, protocolos e tecnologias relacionadas;
- Realizar um estudo inicial comparando alguns controladores SDN, analisando as suas capacidades de controle e encaminhamento de tráfego;
- Projetar uma abordagem (re)configurável de encaminhamento de tráfego IP usando capacidades SDN, após seleção de um controlador SDN apropriado;
- Implementar e testar o mecanismo concebido usando ambientes emulados (e.g. `Mininet`).

1.3 ESTRUTURA DA DISSERTAÇÃO

Este documento está estruturado em cinco capítulos, descrevendo-se como:

- **Introdução**, no qual foi feito o enquadramento e motivação da dissertação, apresentando também os objetivos do trabalho que será desenvolvido;
- **Estado da Arte**, na qual se abordarão conceitos essenciais à realização da dissertação, começando pela generalidade da definição de Software-Defined Networking e avançando para a especificação de alguns Controladores SDN. Será também feita uma abordagem pelo encaminhamento em redes IP, destancando-se, por fim, alguns trabalhos relacionados com a temática do controlo de tráfego via SDN;

- **Arquitetura e Mecanismos Desenvolvidos**, onde será apresentada a arquitetura global do sistema e os demais mecanismos que a compõem, enunciando todo o trabalho desenvolvido e passando pela lógica da implementação dos mesmos;
- **Testes e Análise de Resultados**, capítulo que terá os vários casos de estudo esboçados e concebidos e onde será demonstrada a solução e mecanismos propostos, bem como uma análise dos resultados correspondentes;
- **Conclusão**, onde se apresentarão considerações finais sobre a dissertação como um todo, tendo em conta todo o trabalho desenvolvido e possível trabalho futuro.

2

ESTADO DA ARTE

Neste capítulo serão abordados vários conceitos fundamentais para a realização da dissertação. Inicialmente será explorado o conceito de Software-Defined Networking (SDN) e a sua arquitetura (secção 2.1), passando para a identificação de alguns controladores SDN (secção 2.2). De seguida, mencionar-se-á o encaminhamento em redes de Internet Protocol (IP), destacando vários algoritmos concebidos para tal (secção 2.3), sendo que, por último, se referirão alguns trabalhos relacionados com a junção das duas áreas mencionadas (secção 2.4). Finalmente, será feito um sumário de todos os conceitos abrangidos (secção 2.5).

2.1 SOFTWARE-DEFINED NETWORKING

O conceito das SDN [23], remete para a possibilidade de moldar e manipular tráfego através de um sistema centralizado, sem existir qualquer necessidade de manutenção individual de cada um dos *switches* da rede em específico. Desta forma é possível redirecionar e controlar o tráfego que circula entre os mesmos *switches* ou *routers*, para ser distribuído consoante os parâmetros definidos pelo administrador da rede. Este conceito visa a evoluir a tradição até então adotada, que pretendia, através de tabelas de *routing*, distribuir o tráfego consoante as decisões pré-planeadas nos *routers*, sendo estes, até lá, responsáveis pela tomada de decisão. Através da introdução deste conceito ser-lhes-ia apenas incumbida a tarefa de encaminhamento de tráfego, deixando a lógica de distribuição e as demais escolhas feitas para o controlador SDN.

A essência deste conceito, porém, surge da sua arquitetura e da organização da mesma, a qual será aprofundada nas secções seguintes.

2.1.1 *Arquitetura das SDN*

A arquitetura de uma SDN, na sua vertente mais básica, é constituída pela separação dos planos de dados, controlo e aplicacionais, centralizando todos os estados da rede no plano de controlo, onde se encontrará o controlador SDN. Através desta separação, o utilizador de um controlador SDN usufrui de uma gestão e visão global da rede, visto que os dados

gerados por qualquer dispositivo ligado ao controlador SDN passam pelo plano de controlo do mesmo, minimizando quaisquer custos de manutenção da estrutura da rede por necessidade de averiguação individual do estado de cada um dos dispositivos.

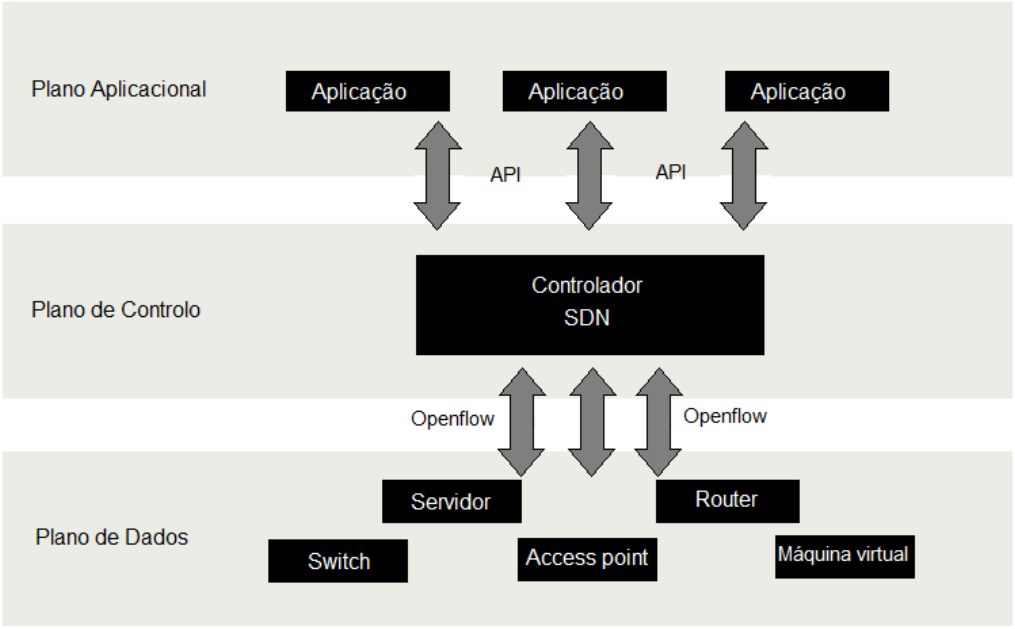


Figura 2.1.: Arquitetura de uma SDN, dividida nos respetivos planos.

Através da Figura 2.1, é possível a constatação de que o plano aplicacional, por ser a camada mais "externa" de todo o processo, se encontra no topo da figura, isto é, a "norte", sendo a comunicação entre o plano aplicacional e o plano de controlo feita, por convenção, através de interfaces *Northbound*. Esta primeira camada é constituída pelas aplicações da rede, comunicando, pela interface *Northbound* e com o plano de controlo, através de Application Programming Interface (API)s, tais como APIs baseadas em Representational State Transfer (REST).

No caso do plano de dados, por outro lado, e encontrando-se no extremo oposto da arquitetura, ou seja, em baixo do plano de controlo (a "sul"), a comunicação entre estes dois planos é feita por interfaces *Southbound*. Este último plano é constituído pela infraestrutura da rede, podendo esta ser baseada em *switches*, *routers*, servidores, *access points* e/ou máquinas virtuais.

As interfaces *Southbound*, regra geral, são baseadas no protocolo OpenFlow (OF), protocolo esse ao qual se dará destaque na secção seguinte.

2.1.2 *Protocolo OpenFlow*

O protocolo OF [12] terá sido criado como um dos primeiros com capacidade de suportar a comunicação entre os planos de controlo e de dados de uma arquitetura SDN, e sendo globalmente aceite (pela sua capacidade simplista de uniformização) como principal meio de comunicação entre os dois planos.

Este mesmo protocolo permite ao administrador da rede a configuração de um canal de comunicação entre a infraestrutura da rede e o controlador SDN, possibilitando a configuração de quaisquer instruções de encaminhamento de cada um dos dispositivos que constitua tal infraestrutura, sendo, deste modo, geridos apenas por um único controlador e protocolo.

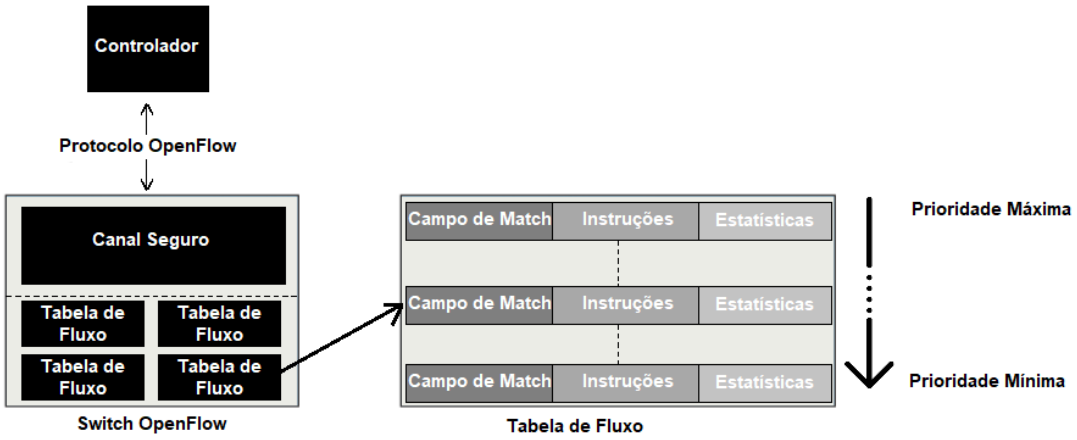


Figura 2.2.: Constituição de uma tabela de fluxo baseada no protocolo *OpenFlow*.

Como é verificável através da Figura 2.2, um equipamento que usufrua do mesmo protocolo despenderá de uma tabela de grupo, sendo esta responsável por agrupar as várias tabelas de fluxo, com o intuito de que seja possível a realização de qualquer tipo de ação sobre estas últimas. Despenderá, também, de pelo menos uma tabela de fluxo, podendo esta ser constituída por entradas. Cada uma destas entradas possui campos que as distinguem entre si, sendo cada entrada única na tabela em que se encontra, nos quais se referem [25]:

- **Campo de "Match"**: neste campo é armazenada informação alusiva aos pacotes que por si passam, tais como a porta de entrada utilizada e os respetivos cabeçalhos;
- **Campo de "Instrução/Ação"**: neste campo é armazenado o intuito da ação a ser tomada consoante o pacote, ou seja, quer seja necessário descartar, enviar ou reencaminhar o pacote;

- **Campo de "Estatística"**: neste campo existem contadores capazes de armazenar informações relativas à quantidade de pacotes e de *bytes* que circulam pelo controlador.

Assim, e através da utilização deste protocolo, é possibilitada uma distribuição adequada e eficaz de qualquer fluxo que circule entre os demais dispositivos que constituam a infraestrutura da rede, facilitando a obtenção de informação dos mesmos ao administrador da rede, sem que seja necessária a averiguação individual de cada um destes. O controlo de toda essa informação é feito pelo controlador SDN existente na rede, descrito na secção seguinte.

2.2 CONTROLADORES SDN

Um controlador SDN, numa *software-defined network*, é o suprassumo de toda a rede. Serve como "centro de operações", sendo a sua posição na arquitetura da rede um ponto estratégico, de forma a que possa comunicar com todos os periféricos. Através da sua centralidade, um controlador SDN apresenta a capacidade de comunicar com o plano de dados através das *Southbound* APIs, e com o plano aplicacional através das *Northbound* APIs, como anteriormente referido, permitindo uma melhor gestão da rede e de todas as aplicações da mesma, redirecionando o tráfego através de políticas de encaminhamento, quer sejam especificadas aquando da execução ou em tempo real.

Na generalidade, uma elevada percentagem de controladores SDN está disponível em licenciamento *open-source*, permitindo que qualquer utilizador possa fazer uso de um controlador que esteja sob tal licença dentro das restrições que lhe são postas, possibilitando a exploração das demais características que, assim, poderão sofrer modificações consoante as contribuições da comunidade, que voluntariamente se disponibiliza a dar seguimento a uma evolução dos controladores SDN. A percentagem que não está inserida no licenciamento não peca pela falta de qualidade, por outro lado, sendo maioritariamente destacada pelo facto de ter suporte contínuo, já que alguns projetos *open-source* se tornam obsoletos consoante a sua falta de manutenção e atualizações, eventualmente levando a uma diminuição no seu uso e, possivelmente, obsolescer.

2.2.1 *Descrição de Controladores SDN*

Estando os controladores SDN em constante evolução, é possível constatar-se que assiduamente surgem vários projetos, quer públicos quer comerciais, tentando oferecer o máximo de características e liberdade de manipulação da rede, nas demais linguagens de programação, com o intuito de se destacarem como o principal controlador SDN do mercado. Porém, e estando numa indústria competitiva e em explosiva evolução, existe uma vasta quantidade de controladores SDN, enunciados pelas suas variadas características [26],

sendo alguns abraçados por fundações e *start-up companies* que se dedicam fundamentalmente a melhorar o controlador que apoiam. Destacam-se os seguintes:

- **NOX/POX** [8] [13], sendo o primeiro, programado em C++, bastante destacado por ser um dos "pais" de todos os controladores SDN pelo facto de ter sido um dos primeiros a implementar o protocolo OF 1.0; o segundo, por outro lado, é uma versão melhorada do NOX, programada em Python, bastante utilizada no ambiente de aprendizagem pela sua simplicidade em comparação com outros controladores;

- **OpenDayLight (ODL)** [2], inicialmente criado para mostrar as demais capacidades de um controlador SDN e destacado por funcionar em ambientes académicos e empresariais, apoiado pela The Linux Foundation, apto para utilizar uma grande variedade de protocolos capazes de funcionar como *Southbound* API;

- **Open Networking Operating System (ONOS)** [15], também adotado pela The Linux Foundation, cujas capacidades se destacam pelo facto de conjugar uma elevada escalabilidade com alta disponibilidade de serviço e desempenho, características especificamente esboçadas para que um *service provider* faça bom uso das mesmas;

- **Faucet** [3], baseado no controlador Ryu e desenvolvido em Python, que funciona com base em duas componentes, Faucet e Gauge, sendo que a primeira é responsável pelo controlo da rede, enquanto que a segunda fica encarregue da monitorização da mesma;

- **Ryu** [34], desenvolvido, também, em Python, capacitado para multi-protocolos. Este controlador permite o controlo de aplicações e eventos, com acesso a uma plenitude de livrarias que podem voltar a ser utilizadas com outros protocolos;

- **IRIS** [16], baseado em Beacon e Floodlight, criando com o propósito de obtenção de horizontalidade nas redes, alta disponibilidade e transparência na eventualidade de falhas, com suporte multi-domínio baseado em OF;

- **Beacon** [6], proposto pela Universidade de Stanford, construído com base num outro controlador SDN, o Floodlight. Criado em JAVA, possui suporte multi-plataforma, funcionando com operações baseadas em eventos e *multithreads*;

- **Floodlight** [9], mantido pela Big Switch Networks (BSN) [28], desenvolvido numa das linguagens de programação atualmente mais utilizadas e sendo o foco principal o do meio empresarial, possui a habilidade de funcionar em multi-plataforma, sendo atualmente licenciado pela licença Apache de *open-source*, permitindo a sua utilização para quase qualquer intuito. Capaz de funcionar com e sem OF, é destacado pela sua versatilidade e fácil manuseamento.

É de salientar que terá sido feito um estudo comparativo mais intensivo e pormenorizado entre vários controladores SDN que estão presentes no mercado, onde são analisados vários detalhes como a versão de OF que incorpora, as áreas de trabalhos relacionados mais fortes do mesmo controlador, a qualidade de documentação e facilidade de aprendizagem dos mesmos, entre outros parâmetros, como encontrado em [26].

Apesar das diferenças e da globalidade no mercado dos controladores SDN, o protocolo OF é a solução de *Southbound* API mais utilizada e standardizada, sendo comum aos demais controladores SDN disponíveis. Porém e apesar de este protocolo ser o protocolo mais utilizado para servir como ponte entre plano de dados e plano de controlo, alguns controladores SDN apresentam soluções capazes de lidar com a totalidade da rede sem a necessidade de implementação do protocolo OF, operando sem que o mesmo esteja presente. O Floodlight, tal como enunciado, é um dos controladores que se destaca por essa característica e que, aliado ao simples manuseamento das demais camadas, facilita a tarefa de manipulação de encaminhamento de tráfego, sendo este explorado na secção seguinte.

2.2.2 *O Controlador Floodlight*

O Floodlight, tal como previamente enunciado, possui a capacidade de funcionar com ou sem protocolo OF, característica que afirma a sua versatilidade. Este funciona à base dos demais módulos implementados, possuindo módulos de controlo (para funcionar em conjunto com o plano de dados) e módulos aplicacionais (para funcionar em conjunto com o plano aplicacional). Entre os primeiros destacam-se a possibilidade de fazer a descoberta da topologia da rede por inteiro, identificando os *links* a que lhe correspondem, a possibilidade de implementação de um servidor com API REST e a manutenção dos demais *switches*. Destaca-se, também, a sua capacidade de lidar com uma plenitude de dispositivos, assegurando a sua escalabilidade.

Este controlador apresenta, também, facilidade de *deployment* e de manipulação das demais camadas. Esta última característica permite ao utilizador comum a possibilidade de modificar, na rede à qual o controlador está atribuído, o comportamento de quaisquer dispositivos ligados ao mesmo, sendo uma das principais razões pela qual é um controlador bastante usado na vertente académica.

Com tal facilidade de manipulação de dados, instruções e dispositivos, é possível a replicação de alguns dos demais algoritmos respetivos às diferentes temáticas, possibilitando a emulação de um ambiente de grande escala simulado, sendo os resultados obtidos pela emulação do mesmo fidedignos aos resultados que seriam obtidos numa situação que se enquadrasse à de sistemas reais, em tempo-real. Apresenta, também, resultados comparáveis [31] a controladores desenvolvidos cujo foco é o empresarial, isto é, controladores não disponíveis em *open-source*.

Deste modo, e através de todas estas características e pelo facto de apresentar resultados bastante positivos, o Floodlight pode ser afirmado como sendo um dos controladores ótimos para desenvolvimento de aplicações nas demais temáticas [20], sendo expansível e flexível a quaisquer melhoramentos, podendo, então, ser usado para recolha de estatísticas, medição de qualidade de serviço e até encaminhamento de quaisquer pacotes na rede em que se enquadra.

Como tal, e devido a todas estas vantagens e características enunciadas, o Floodlight foi o controlador selecionado para desenvolvimento do protótipo proposto neste mesmo trabalho.

2.3 Encaminhamento em Redes IP

O encaminhamento, numa definição mais geral, é definido pelo delineamento do caminho que um pacote de dados terá de percorrer entre uma origem e um destino, possibilitando, assim, a comunicação entre ambos. Para tal, um *router* poderá adaptar dois tipos de encaminhamento, enunciados por características que os diferenciam, sendo estes o encaminhamento estático e o encaminhamento dinâmico.

No caso do primeiro, que é adequado para redes de cariz restrito e pequeno, as regras de encaminhamento são adicionadas manualmente por um administrador de rede, que, à partida, tem conhecimento sobre a totalidade da topologia da rede, visto que esta raramente sofrerá quaisquer alterações (pela sua dimensão e complexidade).

No caso do segundo, adequado às redes de grande escala e complexidade, é caracterizado pelo facto de estar constantemente a ser atualizado pela possibilidade da topologia sofrer alterações constantes. Como tal, as rotas subsequentes são geradas em tempo real, através de algoritmos e protocolos de encaminhamento que reagem a quaisquer modificações na rede. Enquanto que a primeira alternativa requer o conhecimento da topologia, esta segunda alternativa requer apenas o conhecimento e habilidade de manipulação do protocolo usado para lidar com o encaminhamento dinâmico.

No contexto do encaminhamento dinâmico em redes IP, e por convenção, referimo-nos a conjunto de redes IP gerida por uma mesma entidade de "sistema autónomo". Um sistema autónomo é capaz de gerir um ou mais conjuntos de redes IP, sendo, a título de exemplo, um Internet Service Provider (ISP) equivalente a um sistema autónomo.

Através da manutenção desses mesmos sistemas autónomos advêm protocolos de encaminhamento que funcionam de duas formas: externamente (usados no exterior de um sistema autónomo, de modo a que seja possível a obtenção de informação dos caminhos de diferentes sistemas autónomos, tal como a troca de informação de tabelas de *routing*), denominados de Exterior Gateway Protocol (EGP) [30]; e internamente (usados no interior

de um sistema autónomo, de modo a obter informação dos caminhos do próprio sistema), denominados de Interior Gateway Protocol (IGP) [32].

No primeiro caso destaca-se o Border Gateway Protocol (BGP) [10]. O BGP é um protocolo de encaminhamento externo usado na Internet para interligação entre diversos sistemas autónomos. A informação de encaminhamento é atualizada dinamicamente, sendo que as rotas escolhidas são dependentes de diversas regras e políticas de encaminhamento definidas pelos administradores da rede, e que são configuradas nos *routers* BGP.

No segundo caso, existe uma variedade de algoritmos capazes de resolver o problema enunciado de formas diferentes, obtendo o resultado de melhores ou piores formas. Como tal, destacar-se-ão algumas soluções viáveis nas subsecções seguintes.

2.3.1 *Algoritmos Distance-Vector*

Os algoritmos Distance-Vector, quando aplicados ao *routing*, surgem como uma boa alternativa capaz de manter a informação atualizada entre os demais vizinhos, sendo esta periodicamente modificada, nas demais tabelas de *routing* de cada um dos dispositivos pertencentes à rede, consoante haja alguma mudança na topologia da rede.

Estes algoritmos, por exemplo através do algoritmo de Bellman-Ford [37], começam com a criação da tabela de *routing* em cada um dos dispositivos, preenchendo a informação acerca da custo associado ao "salto" entre um *router* e outro. Numa primeira fase, comparável à da Figura 2.3, é apenas garantida a informação relativa aos vizinhos diretos, visto que é a única informação disponível logo após o início do algoritmo.

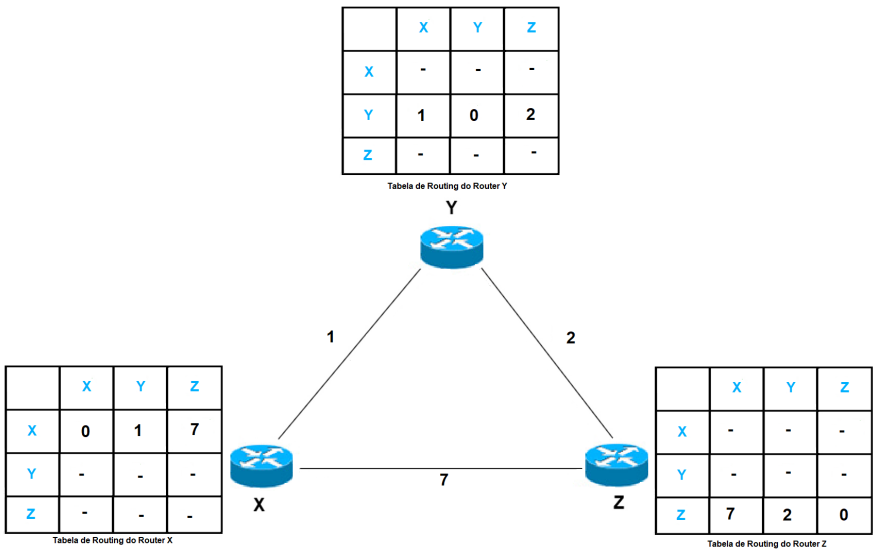


Figura 2.3.: Versão inicial de uma execução do algoritmo Distance Vector Routing.

De seguida, passa-se a uma fase de partilha das tabelas entre cada um dos vizinhos, passando a informação que sabem aos seus vizinhos diretos, e atualizando os campos que lhes faltam consoante a informação que recebem. A título de exemplo, e continuando com a Figura 2.3, atualiza-se a tabela do *router* X com a informação recebida da tabela de *routing* do *router* Y, originando a Figura 2.4:

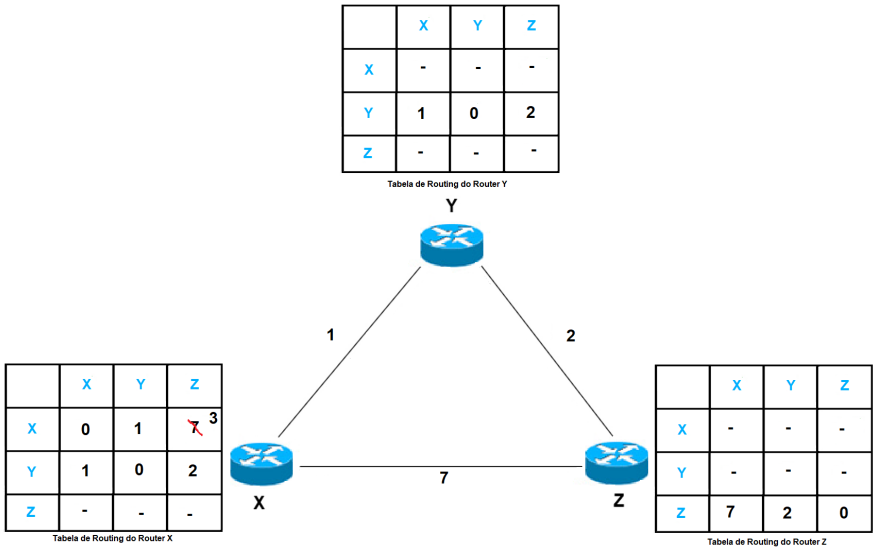


Figura 2.4.: Passo seguinte após a execução inicial do algoritmo Distance Vector Routing.

Finalmente, todos estes passos são repetidos até que as tabelas de todos os *routers* estejam preenchidas, resultando nas respetivas tabelas finais, como verificável na Figura 2.5:

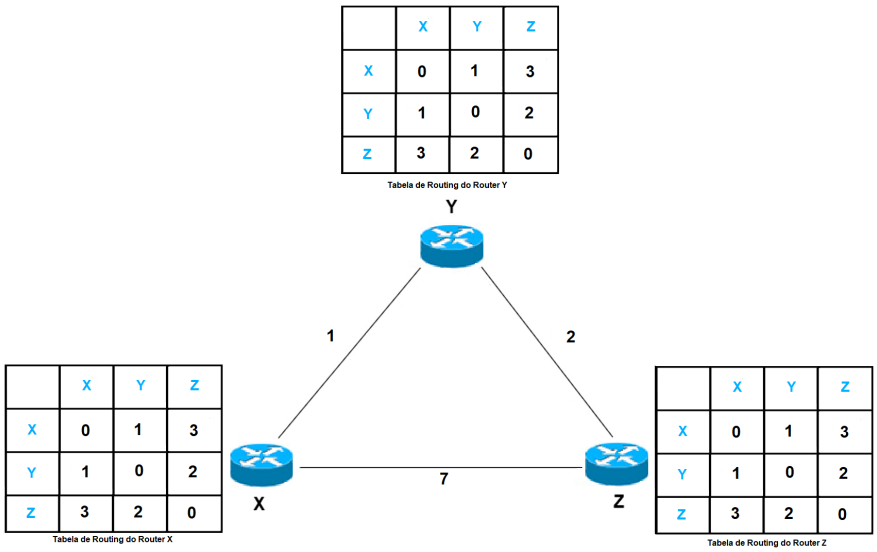


Figura 2.5.: Preenchimento total das tabelas de *routing* no algoritmo de Distance Vector Routing.

Na sua generalidade, este algoritmo é bastante eficaz e capaz de lidar com grandes redes. Possui, porém, algumas desvantagens, tais como:

- O algoritmo pode ficar preso em ciclo, levando a um processo denominado de contagem até ao infinito, na eventualidade de um dos *links* entre dois *routers* sofrer alguma alteração após a execução do algoritmo, pondo em causa a viabilidade das demais tabelas de *routing* [17];
- Em redes de enorme escala, é de se esperar que as tabelas de *routing* geradas sejam extremamente grandes, gerando, assim, um elevadíssimo número de tráfego na eventualidade de uma das entradas da tabela necessitar de ser atualizada.

Existe uma outra alternativa a esta família de algoritmos, capaz de responder a algumas das desvantagens do algoritmo anterior, sendo este enunciado na subsecção seguinte.

2.3.2 *Algoritmos Link-State*

Ao invés dos Distance-Vector algorithms (DV), os algoritmos Link-State (LS) podem por exemplo ter como base o algoritmo do caminho mais curto de Dijkstra [38], sendo este capaz de lidar com os problemas elucidados anteriormente, ou seja, o algoritmo Link-State não peca pelo facto de poder ficar num ciclo, levando a um processo denominado de contagem até ao infinito.

Este algoritmo apresenta várias vantagens quando comparado com o anterior, sendo que uma delas advém desde já do facto de qualquer nodo ter o conhecimento da topologia inteira (enquanto que o algoritmo de DV funciona apenas consoante o conhecimento dos nodos vizinhos).

Inicialmente, este algoritmo começa por escolher um qualquer *router*, despoletando a descoberta dos caminhos para outros nós, como visto na Figura 2.6, para o caso do *router* A:

Passo	Nós percorridos	D(B) p(B)	D(C) p(C)	D(D) p(D)	D(E) p(E)	D(F) p(F)
0	A	6,A	2,A	8,A	∞	∞
1						
2						
3						
4						
5						

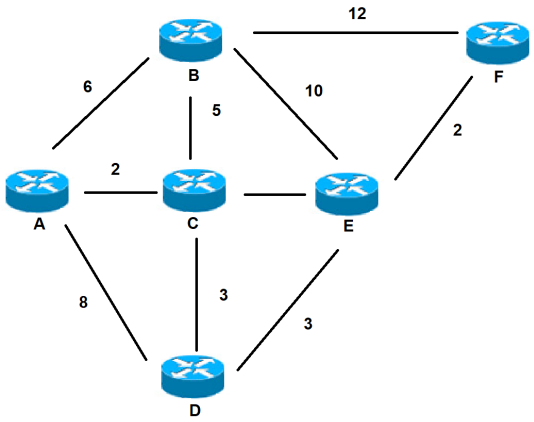


Figura 2.6.: Descoberta inicial dos links vizinhos no algoritmo de Dijkstra aplicado ao *routing*.

Após esta descoberta inicial, o algoritmo funcionará tal e qual o algoritmo de Dijkstra: consoante as arestas e os vértices do grafo a que se lhe corresponde, escolhe a aresta cujo custo é mínimo, tomando-o como caminho seguinte, como verificável na Figura 2.7:

Passo	Nós percorridos	D(B) p(B)	D(C) p(C)	D(D) p(D)	D(E) p(E)	D(F) p(F)
0	A	6,A	2,A	8,A	∞	∞
1	AC					
2						
3						
4						
5						

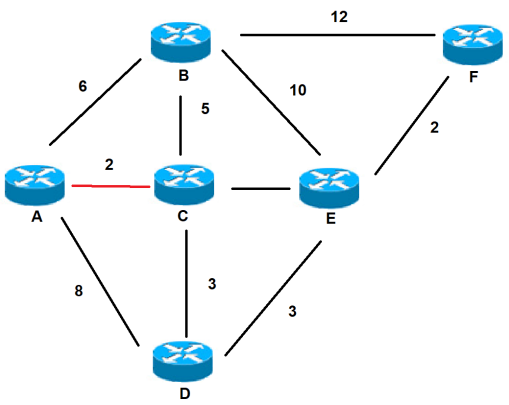


Figura 2.7.: Escolha do caminho mais curto e passagem para o *link* seguinte no algoritmo de Link-State.

Feitos estes dois passos, atualiza-se a tabela do caminho mais curto com o passo seguinte, passando, então, à descoberta do vértice seguinte para o caminho mais curto, tal como na Figura 2.8:

Passo	Nós percorridos	D(B) p(B)	D(C) p(C)	D(D) p(D)	D(E) p(E)	D(F) p(F)
0	A	6,A	2,A	8,A	∞	∞
1	AC	6,A		5,C	8,C	∞
2						
3						
4						
5						

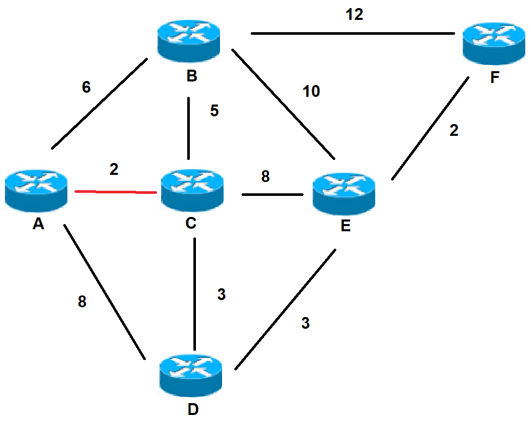


Figura 2.8.: Atualização da tabela de caminho mais curto e passagem para o passo seguinte do algoritmo de Link-State.

Por fim, basta repetir o algoritmo de Dijkstra até que seja descoberto o caminho mais curto que passe por todos os vértices do grafo, levando, finalmente, e terminando o exemplo anterior, ao seguinte grafo com os seguintes caminhos mais curtos, verificável pela Figura 2.9:

Passo	Nós percorridos	D(B) p(B)	D(C) p(C)	D(D) p(D)	D(E) p(E)	D(F) p(F)
0	A	6,A	2,A	8,A	∞	∞
1	AC	6,A		5,C	10,C	∞
2	ACD	6,A			8,D	∞
3	ACDB				8,D	18,B
4	ACDBE					10,E
5	ACDBEF					

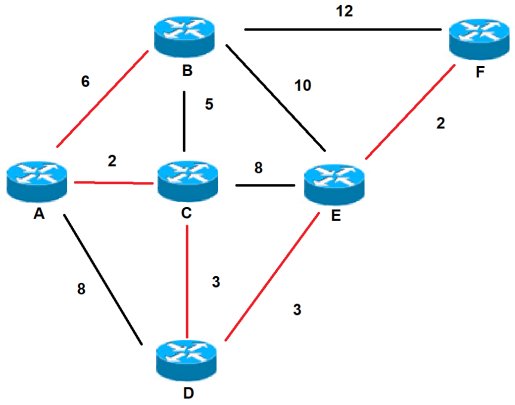


Figura 2.9.: Tabela final após aplicação do algoritmo de Link-State à topologia exemplificada.

A implementação de protocolos baseados neste algoritmo, porém, e mesmo apesar de apresentar algumas vantagens quando comparado com o algoritmo de Distance-Vector, acabam também por ter algumas nuances, sendo estas:

- Requer maior poder de processamento e memória para cálculo considerando a topologia inteira;
- Faz uma mais exigente utilização dos recursos da rede, muito pelo facto de ser necessário o *flooding* das mensagens para todos os *routers*, isto é, é necessário o envio de

uma mensagem para todos os nodos de uma rede para divulgação de toda a topologia, aumentando a carga usada pela globalidade da rede.

2.3.3 *Protocolos OSPF/RIP*

Apesar dos dois algoritmos discutidos nas secções anteriores apresentarem algumas desvantagens, as suas vantagens são vastamente superiores, sendo então utilizados por dois protocolos globalmente conhecidos: o Routing Information Protocol (RIP) [11] em relação ao Distance-Vector, e o Open Shortest Path First (OSPF) [22] no que toca ao Link-State.

O OSPF e o RIP, ambos protocolos de encaminhamento interno, são dois exemplos clássicos de protocolos utilizados na vertente de cálculo das melhores rotas para transmissão de dados intra-Autonomous System (AS).

O primeiro, baseado em princípios de Link-State, recorre ao mesmo método de transmissão de informação: através do *flooding*, os *routers* que perfazem a totalidade da rede enviam o estado das suas interfaces a todos os outros *routers* da rede através de *Link State Advertisements*. Com a finalização da primeira execução do algoritmo de Dijkstra, cada *router* terá uma visão global da topologia, possuindo a capacidade de calcular quaisquer caminhos mais curtos para os demais destinos correspondentes à topologia.

O protocolo OSPF faz uso de uma hierarquia definida localmente, composta por "áreas", isto é, zonas que possuem informação local de *routing*. Existe uma "área" responsável por encaminhar rotas para todas as "áreas" a que lhe estão ligadas, denominada de *backbone area*, permitindo, assim, a comunicação entre os demais *routers* das diversas "áreas". Através desta perspetiva, é possível uma maior escalabilidade e melhor gestão de toda a informação de encaminhamento, como visto na Figura 2.10.

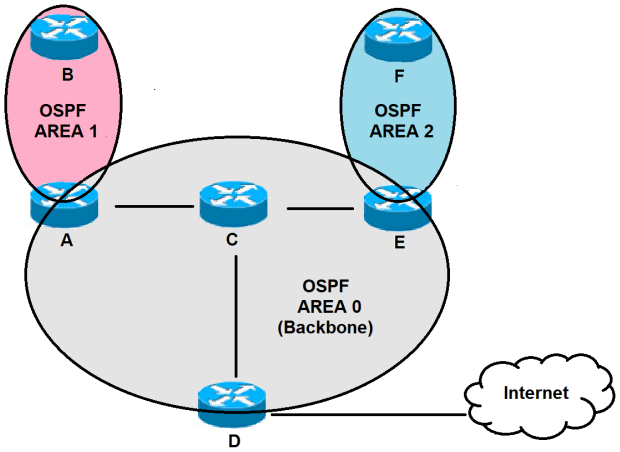


Figura 2.10.: Exemplificação de uma estrutura em áreas com OSPF.

No caso do segundo protocolo, o RIP, baseado em Distance-Vector, as decisões de escolhas dos caminhos são feitas consoante o número de saltos necessários até ao destino. Este protocolo consegue anunciar, juntamente com os pacotes enviados por User Datagram Protocol (UDP), as seguintes informações:

- Endereços IP;
- Distância até a um destino;
- Custo da rota.

O protocolo RIP, porém, sofre das adversidades da utilização do algoritmo baseado em Distance-Vector, tais como a possibilidade de ficar preso num ciclo ou de atualizar uma tabela de *routing* infinitamente. Como tal, o RIP apresenta soluções que pretendem mitigar tais problemas, diminuindo o seu impacto no desempenho da rede. Uma das soluções mais eficazes é a técnica de Split Horizon com Poison Reverse, que, para melhorar a convergência da rede, pretende combinar o conceito de Split Horizon (não anunciar numa interface de rede uma rota que tenha sido aprendida pela mesma interface) com o conceito de Poison Reverse, que ao invés de não se anunciar a rota, anuncia-se com um custo fixo de 16 (correspondente a "infinito").

Este último protocolo, porém, é bastante menos utilizado que o OSPF. Isto deve-se ao facto de as redes estarem em constante crescimento e expansão, sendo que o OSPF se adapta melhor à convergência da rede e a mudanças de rotas, mostrando-se ótimo para a sua utilização em redes de grande escala. Como tal, pode-se afirmar que o protocolo OSPF é um protocolo mais robusto, capaz de levar a cabo tarefas de maior escala em redes de tamanho considerável.

Como nota adicional, convém agora referir o seguinte aspeto. Estando esta dissertação centrada na temática dos ambientes SDN, onde todo o plano de controlo passa a estar sobre a alçada do controlador SDN, fará sentido assumir cenários em que os elementos da rede (e.g., *routers*/*switches*) deixam de ter a necessidade de suportar protocolos de encaminhamento, tais como o RIP ou OSPF. Desta forma a lógica do cálculo do encaminhamento de tráfego passará a estar localizada no controlador SDN. No entanto, é de notar que, devido à visão global e centralizada que o controlador SDN tem sobre a rede, pode fazer sentido que algoritmos tipo os LS abordados anteriormente possam perfeitamente continuar a ser usados pelo controlador para os cálculos dos caminhos entre os diversos elementos da rede.

2.4 Controlo de Tráfego via SDN

Uma SDN surge como possibilidade de implementar os algoritmos enunciados anteriormente da forma que o administrador responsável pela rede preferir, permitindo uma visão

central sobre o tráfego que passa pela totalidade da rede, com o controlador também capaz de moldar o tráfego consoante as demais preferências do utilizador. Esta flexibilidade permite ao administrador da rede a máxima liberdade na implementação e desenvolvimento de quaisquer mecanismos. É através deste conceito que as SDN, aliadas a uma necessidade cada vez mais crescente de escalabilidade e quantidade de *routers*, facilitam todo este processo.

Esta liberdade de manipulação, em junção com a variada panóplia de controladores que existe, assim como as diferentes linguagens de programação em que cada um está implementado, possibilita o desenvolvimento acrescido e constante de todas as tecnologias que circulam as SDN, quer por contributos comunitários para controladores *open-source*, quer por contributos empresariais para controladores com suporte profissional dedicado. Deste modo, os controladores SDN crescem com a evolução do mercado, obedecendo às inúmeras demandas que as redes IP exigem.

Sendo as SDN um conceito em expansão, tem-se, ao logo do tempo, enquadrado facilmente em inúmeras áreas de investigação, capazes de tirar o máximo de proveito das mesmas. Como tal, e na subsecção seguinte, enumerar-se-ão algumas das áreas de utilização das redes SDN, referenciando alguns exemplos.

2.4.1 *Áreas de utilização das SDN*

Em relação ao contexto empresarial, e estando as redes SDN a afirmar-se como alternativa às típicas redes Wide Area Network (WAN) (ou complementando-as através das Software Defined WAN), forma-se um novo paradigma de *networking* no qual é possível ajustar a rede dinamicamente de acordo com as necessidades requeridas. Esta alternativa, complementada com mecanismos de ajuste de banda larga, garante a distribuição correta da mesma consoante os fluxos de dados gerados a qualquer momento [7]. Deste modo, permite à empresa uma poupança significativa de fundos gastos, visto que apenas pagará pela utilização efetiva dos recursos da rede.

Coligando as mesmas redes SDN com *data centers*, um tipo de infraestrutura cada vez mais emergente graças ao crescimento exponencial de dados gerados por dispositivos, conseguimos a escalabilidade na rede das mesmas, escalando a banda larga incrementalmente sem que sejam necessárias despesas adicionais na aquisição de *hardware*. Tal como na menção anterior, esta solução permite a diminuição de custos, fator importante no balanceamento financeiro de qualquer empresa.

Quanto ao âmbito académico, e referindo as redes existentes num Campus universitário, de forma a responder ao tráfego de rede inconstante gerado em tal localização, há a possibilidade de se fazer *slicing* [4] das mesmas redes, ou seja, particionar a rede em múltiplas redes virtuais capazes de se interligar através de *software*. Esta abordagem possibilita o

crescimento livre da mesma rede sem custos acrescidos, permitindo, também, a adaptação consoante as necessidades requeridas por cada rede em específico, quer seja por parte de dispositivos, serviços, aplicações, utilizadores ou até operadores.

Assim, e de modo a apresentar a plenitude de cenários em que uma SDN se enquadra, serão apresentados, na subsecção seguinte, alguns trabalhos desenvolvidos na área de encaminhamento de tráfego.

2.4.2 *Trabalhos Relacionados*

Com o conceito de SDN em constante evolução, surgem, com o decorrer do tempo, soluções capazes de replicar algoritmos de encaminhamento de tráfego já existentes sob esta nova tecnologia, tirando partido das características da mesma.

Com a área de encaminhamento de tráfego maioritariamente presente em qualquer contexto que englobe o próprio conceito das SDN, é possível constatar que, no que toca a contribuições comunitárias e académicas, existe uma panóplia de trabalhos e investigações feitas sobre a mesma.

Inicialmente, e de forma a que se pudesse comprovar a eficiência do paradigma das SDN, foi feito um estudo [29] que visava a avaliação do desempenho do mesmo, em conjunção com o protocolo OF, ambos inseridos numa máquina virtual. Tendo sido concluído que o encaminhamento de tráfego através destas seria o adequado, foi dado azo a novas investigações sobre as demais temáticas, sempre englobando as SDN.

Como tal, e a título de exemplo, entre estes surgiram abordagens que pretendem a maximização da eficiência de envio de pacotes via *broadcast* para uma rede [33], minimizando, também, o tempo estimado para entrega de um pacote e o tráfego criado pela circulação da quantidade exorbitante de pacotes, solução adequada para uma rede de grande escala. Esta solução permite, através do encaminhamento generalizado dos mesmos, minimizar o tempo de espera entre a troca de pacotes de dois *end-points*.

Surge, também, uma solução [18] capaz de lidar com os demais contextos em que uma qualquer topologia se encontra, moldando as suas decisões consoante variáveis que influenciam o ambiente da rede. Esta solução faz uso das capacidades que uma SDN oferece, aliada ao encaminhamento dinâmico de tráfego, capaz de se adaptar ao crescimento inesperado de uma infraestrutura.

Como a infraestrutura de uma rede é dos pontos mais importantes para a implementação de um controlador SDN, e podendo a mesma ser o mais variada possível, é necessário que estas tecnologias sejam capazes de se aperceber do contexto em que se enquadra tal infraestrutura. Assim, surgiu [5] uma metodologia que consegue influenciar o controlador de modo a aperceber-se do contexto em que a rede se encontra, capaz de explorar as demais capacidades e vantagens do paradigma.

As SDN afirmam-se, constantemente, como meio capaz de centralizar o controlo em várias situações-chave. Porém, implementar uma arquitetura SDN centralizada numa Low Power and Lossy Network (LLN), uma rede de baixa potência e com elevada perda de pacotes, torna-se desafiante, visto que o tráfego que circula no mesmo controlador é propício a *jitter*, devido a links não-confiáveis e contenção da rede, afetando, de certo modo, o desempenho da mesma. Como tal, existe uma proposta de solução [1] cujo intuito é o de propor um mecanismo capaz de responder a este tipo de problemas, de forma leve e eficaz.

No que toca a um dos problemas mais frequentes e existentes no dia-a-dia, foi esboçada também uma alternativa ao típico modelo de *slave/master* [40], com o intuito de minorar o tempo de recuperação após uma falha inesperada num controlador, através da resiliência dos *switches* capazes de comunicar através de uma SDN. Esta possibilidade abarca o mapeamento entre os vários *switches* e o controlador, maximizando a sua confiabilidade e a capacidade de recuperação na eventualidade de falha de mapeamento entre um destes.

Já em relação à temática da latência, e ainda englobando o encaminhamento de tráfego, destaca-se um estudo cujo intuito é o de diminuição do número de entradas de fluxo criadas através de encaminhamento origem-destino baseado em *multipath* [35]. Esta abordagem possibilita uma minimização substancial das entradas de fluxo geradas quando comparada com técnicas existentes, utilizadas por ISPs, através da tentativa de *load balancing*.

Relativo ao plano de controlo de uma SDN, e visando o uso cada vez maior de *switches* virtuais, advém o problema de garantir a máxima segurança na transmissão de dados entre o controlador e os *switches*, de modo a que não seja intersectado qualquer fluxo. Como tal, é concebida uma proposta de solução para este desafio [19], através da criação de novos caminhos e regras de encaminhamento consoante a chegada do fluxo à SDN. Estas condições são adicionadas como alternativa às impostas pelo controlador imediatamente após a inicialização do mesmo, provando ser uma alternativa segura, válida, eficaz e simples.

Para obter informação acerca do número de pacotes de diferentes fluxos, algumas aplicações SDN podem instalar regras de encaminhamento adicionais, com o propósito único de contar pacotes com cabeçalhos específicos. Porém, e para obter estatísticas sobre a totalidade da rede, estas aplicações instalariam uma elevada quantidade de regras de encaminhamento, limitando o espaço disponível na tabela de encaminhamento para outras aplicações. Como tal, surge uma solução capaz de minimizar o número de regras criadas na mesma situação através de um algoritmo desenvolvido [27], capaz de diminuir o número das mesmas em até 50%.

Existem, também regras de encaminhamento mal enunciadas, inseridas ou aplicadas. Deste modo, e bastando uma regra de encaminhamento errada para prejudicar o normal funcionamento da rede, há também uma solução capaz de detetar quaisquer falhas na inserção das mesmas, quer seja em tempo real ou não, permitindo uma correção de tais

entradas na tabela de encaminhamento, assegurando que o fluxo da rede é encaminhado corretamente [36].

Como tal, estes são alguns dos trabalhos que destacam e exemplificam que esta é uma área em constante evolução, com novas soluções feitas regularmente, podendo, assim, ser uma área facilmente explorada, com uma imensidade de benefícios na sua correta utilização. As SDN são ótimas para quaisquer sub-temas que envolvam encaminhamento de tráfego, mostrando-se, assim, capazes de corresponder a quaisquer expectativas.

2.5 Sumário

Neste capítulo foi feita uma descrição de toda a temática referente à dissertação, sendo que nos próximos capítulos será feita toda a especificação, desenvolvimento e análise do foco principal da dissertação, que será o de tentativa de desenvolvimento de uma solução inovadora na área de encaminhamento de tráfego, tendo em conta a utilização de uma SDN.

Inicialmente, foram mencionadas as áreas em que as SDN se enquadram, assim como o seu conceito e as suas características.

Seguidamente, foi introduzida a arquitetura das SDN, enunciando os seus planos de dados, controlo e aplicacional, assim como os componentes que asseguram a sua interligação. Nesta secção, foi também dado destaque ao facto de os controladores *sdn* possibilitarem a comunicação com componentes externos através de APIs baseadas em REST.

Foi feita uma paralelização com o protocolo OF, que é o que possibilita a comunicação entre os variados planos presentes num controlador SDN, protocolo este que é o mais utilizado na data de escrita desta dissertação. Neste, foram referidos os campos das entradas que constituem as suas tabelas de fluxo, cruciais para a realização deste trabalho.

Foi, também, feita uma breve exposição aos vários controladores SDN existentes no mercado, assim como algumas das sua características e linguagem de programação em que estão implementados. Nesta secção, é também justificada a escolha do controlador utilizado no capítulo seguinte, o `Floodlight`.

Foram referidos alguns dos algoritmos e protocolos de encaminhamento em redes IP, passando quer pelos de encaminhamento externo, quer pelos de encaminhamento interno, demonstrando a funcionalidade dos mesmos com exemplos passo-a-passo.

Por fim, é feita uma interseção entre a área de encaminhamento de tráfego e o controlo de tráfego através de SDN, passando pela descrição da utilização do mesmo para esta área, sendo depois dada uma breve menção a alguns trabalhos relacionados com a área de encaminhamento de tráfego e que englobam os SDN, tirando partido das suas características.

A partir deste ponto, será feita uma averiguação e investigação dos desafios que rodeiam o problema introduzido, assim como arquiteturas e mecanismos esboçados para combater os mesmos.

3

ARQUITETURA E MECANISMOS DESENVOLVIDOS

Neste capítulo será descrita toda a arquitetura desenvolvida em detalhe, assim como os mecanismos que a caracterizam. A primeira secção (secção 3.1), identifica todas as entidades que constituem a arquitetura geral, enquanto que a secção seguinte (secção 3.2) menciona os mecanismos desenvolvidos, a sua viabilidade e cenários reais comparáveis. Por fim, será feito um sumário deste mesmo capítulo (secção 3.3).

3.1 ARQUITETURA GERAL E ENTIDADES

Nesta secção é apresentada a arquitetura geral da solução desenvolvida, juntamente com as entidades que a caracterizam. Esta será explorada ao detalhe, enunciando qual o seu papel para o desenvolvimento da solução.

A arquitetura implementada da solução em questão, apresentada na Figura 3.1, terá sido feita com uso de uma *script* Mininet, capaz de emular uma rede virtual em tempo real, assim como os demais *links*, *routers* e *end-user areas*. Esta arquitetura é composta, então, pelos seguintes componentes:

- **Uma rede de Internet Service Provider (ISP)**, que é constituída por *routers* `core` e *links* de interligação entre os mesmos;
- ***End-user areas***, que são compostas por vários *hosts* (que, na arquitetura em específico, foram considerados 3 *hosts* como valor arbitrário das mesmas), que estão ligadas à rede ISP através de um *router* de fronteira entre a topologia e a área;
- **Um controlador Software-Defined Networking (SDN)**, que, como se observa na Figura 3.1, por sua vez tem vários módulos de apoio, isto é, estruturas de dados (para armazenar as rotas calculadas, estatísticas, previsão de falhas, níveis de carga de *links* e estado dos *links* da topologia) e as lógicas de encaminhamento (que serão os mecanismos desenvolvidos na próxima secção, entre os quais está a execução do algoritmo de `Dijkstra`, a configuração de métricas de encaminhamento, a convergência imediata após falhas de *links*, a proteção de tráfego de entidades da infraestrutura, a

reatividade a níveis de congestão, a especificação de rotas individuais para fluxos de tráfego e a multiplexagem da topologia física por diferentes redes virtuais);

- **Um administrador de rede**, que interage com o controlador de forma manual ou através de processos automatizados.

A arquitetura apresenta, também, na Figura 3.1, vários números representados a vermelho, sendo estes identificadores dos custos dos *links* atribuídos pelo administrador da rede para a topologia em questão. A tracejado estão os comandos OpenFlow (OF) com as entradas de encaminhamento para os *switches* e estatísticas para o controlador, sendo esta a ligação que os *switches* fazem com o controlador SDN.

Esta rede é também composta por múltiplos *routers*, *links* e quatro *end-user areas*, cada uma definida em formato `/24`, de endereços únicos `10.0.X.Y` (ou seja, seria atribuída uma gama específica de endereços a cada *end-user area*), tal que:

$$\{X \mid X \in \{1, 2, ..., \text{número total de } \textit{end-user areas}\}\ \}$$
$$\{Y \mid Y \in \{1, 2, ..., \text{número total de } \textit{hosts} \text{ na rede} + 1\}\ \}$$

Para o último intervalo (da letra Y), o número total de *hosts* da rede é incrementado em uma unidade de forma a contabilizar o router que faz a ligação entre uma *end-user area* e a topologia. Este é representado, para qualquer *end-user area* X pela gama de endereços `10.0.X.4/32`.

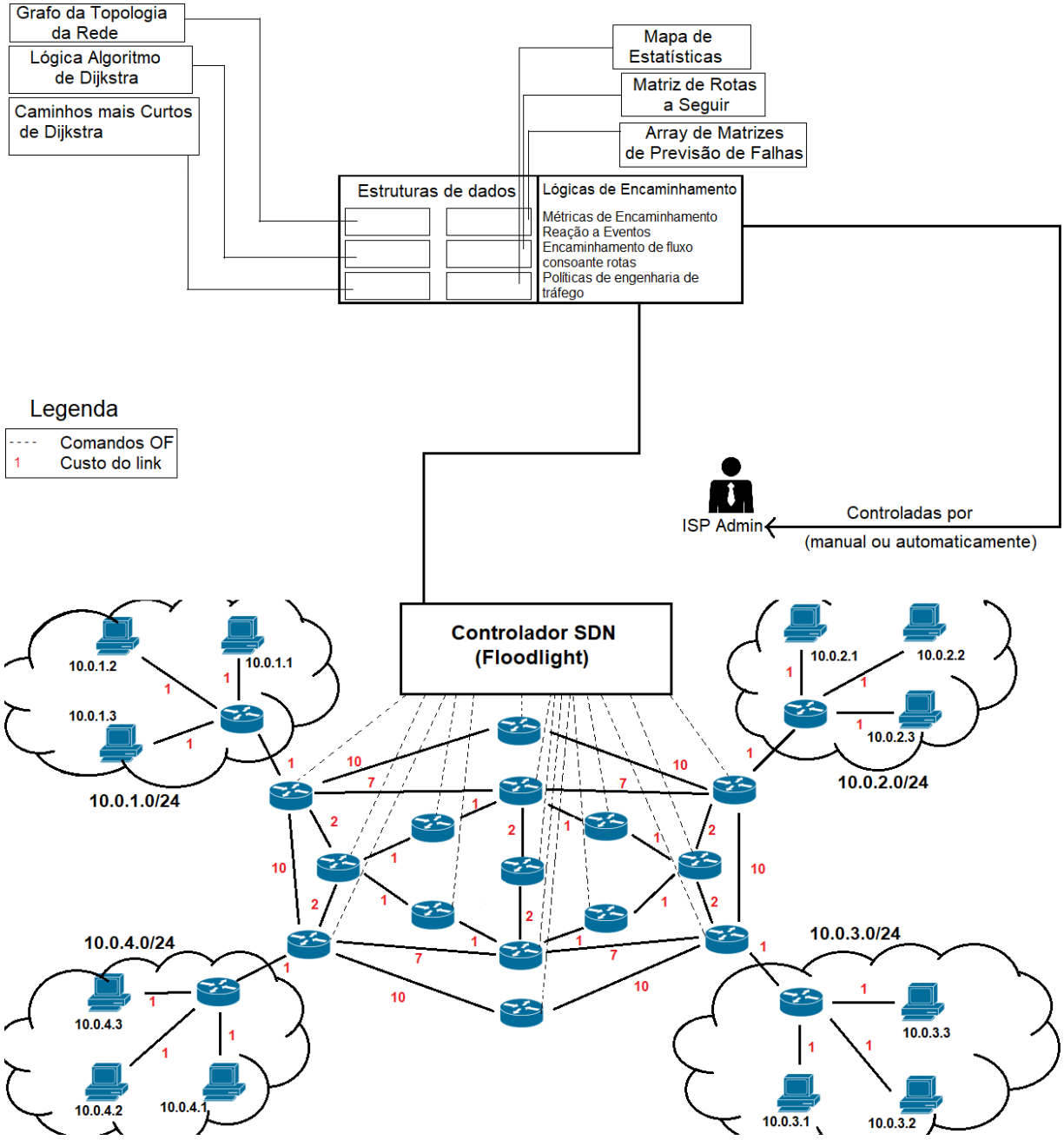


Figura 3.1.: Arquitetura da solução implementada.

3.2 MECANISMOS DESENVOLVIDOS

Nesta secção são apresentados os mecanismos desenvolvidos sobre a solução implementada, tentando estes responder às necessidades requeridas a uma solução de encaminhamento, por parte do mercado. Cada um dos mecanismos apresentará uma descrição in-

dividual e uma porção da sua pseudo-implementação, juntamente com a sua utilidade esperada em relação a cenários reais, comparando-a, também, com os mesmos. Na primeira subsecção (3.2.1) consideram-se os algoritmos existentes como essenciais para que a solução apresentasse funcionalidades básicas mas eficazes, sendo que nas seguintes subsecções se apresentam os demais mecanismos adicionais que complementam a mesma solução.

3.2.1 *Solução Básica de Encaminhamento*

Para que se criasse uma solução de encaminhamento de tráfego completa no que toca a mecanismos essenciais, considerou-se a necessidade de se implementar um algoritmo de cálculo de caminhos mais curtos para obtenção das rotas a seguir, pelos *links* da topologia. Com este mesmo mecanismo, restaria apenas a averiguação de potenciais falhas de *links*, visto que, na eventualidade de estas ocorrerem, os caminhos mais curtos calculados pelo algoritmo anterior seriam invalidados, já que, potencialmente, um dos *links* responsáveis por qualquer um dos caminhos mais curtos poderia ser um dos afetados.

Deste modo, e com estes dois mecanismos, considera-se que a solução de encaminhamento de tráfego já deveria obedecer aos requisitos mínimos para uma solução de encaminhamento completa, restando, assim, a implementação dos demais mecanismos adicionais que enriquecessem a mesma. Como tal, e nas duas próximas subsecções do Algoritmo de Dijkstra e de Notificação de Falhas de *Links* serão enunciados os tais mecanismos-base necessários para que a solução funcionasse corretamente.

Algoritmo de Dijkstra

Inicialmente, e de forma a que fosse possível a obtenção dos caminhos mais curtos para encaminhamento de tráfego entre as demais *end-user areas*, implementou-se uma variante do Algoritmo de Dijkstra [38]. Este algoritmo, também utilizado no protocolo de *routing* Open Shortest Path First (OSPF), prevalece pela sua simplicidade e eficiência de construção de um grafo dos caminhos mais curtos até um nodo.

O algoritmo, adaptado ao caso em questão, permite o cálculo dos caminhos mais curtos entre as demais *end-user areas* presentes na topologia. Estes caminhos mais curtos serão particularmente úteis para os próximos mecanismos implementados, visto que é sob estes que a grande maioria dos mesmos se baseará, de forma a optimizar os caminhos percorridos entre origem e destino.

No que toca à funcionalidade do mesmo, o algoritmo de Dijkstra adaptado recebe a topologia em questão e os custos dos demais *links*, devolvendo, como *output*, uma matriz bidimensional com as rotas entre os variados *hosts* das *end-user areas*. Esta matriz bidimensional é dada por um *array* de duas dimensões, na qual se representam as origens e

os destinos em cada uma das dimensões e as respetivas rotas no conteúdo destas, como visível na Figura 3.2.

Área Origem / Área Destino	1	2	3	4
1		rota (1,2)	rota (1,3)	rota (1,4)
2	rota (2,1)		rota (2,3)	rota (2,4)
3	rota (3,1)	rota (3,2)		rota (3,4)
4	rota (4,1)	rota (4,2)	rota (4,3)	

Figura 3.2.: Estrutura da matriz bidimensional armazenada.

De seguida, as rotas calculadas pelo mesmo algoritmo serão injetadas nos demais *switches* da topologia, de modo a que a tabela de encaminhamento de cada um destes possua a informação calculada e armazenada no controlador.

O algoritmo, adaptado ao caso em questão, para que seja possível a obtenção da matriz bidimensional, é representado pelo seguinte pseudo-código dado pelo Algoritmo 1:

Algoritmo 1: Obtenção dos caminhos mais curtos da topologia através do Algoritmo de Dijkstra.

Input: topologia e custos dos links
Output: matriz bidimensional com rotas correspondentes

```
1 inicialização;
2 for uma qualquer end-user area X do
3 |   executa Dijkstra sobre um host de uma end-user area X origem para um host de
  |     outra qualquer end-user area destino;
4 |   preenche na matriz bidimensional temporária os caminhos calculados;
5 end
6 guardar resultado do preenchimento em matriz bidimensional;
7 injectar rotas calculadas nos switches;
```

A execução deste algoritmo é feita antes do normal funcionamento da rede, isto é, logo a seguir ao arranque da topologia e imediatamente antes da circulação de fluxo de tráfego entre os demais *routers*.

Esta matriz é criada e mantida desde a execução inicial da solução concebida, sendo, posteriormente, atualizada consoante outros mecanismos a requeiram.

Notificação de Falhas de Links

De modo a que a solução básica de encaminhamento de tráfego obedecesse aos requisitos básicos impostos anteriormente, considerou-se que a mesma deveria reportar quaisquer falhas de *links* que deixassem de responder ao controlador. Como tal, o controlador deveria contornar as mesmas falhas através do recálculo das rotas entre as *end-user areas*, seguido da atualização da topologia por injeção das respetivas novas rotas. Como tal, recorre-se a algumas capacidades já presentes no controlador `Floodlight` para que este passo seja possível.

A utilidade deste algoritmo deve-se ao facto de se poder lidar com quaisquer situações inesperadas, podendo, o controlador, agir prontamente sobre quaisquer falhas que estejam fora do seu controlo. Deste modo, e independentemente de o controlador estar presente a uma topologia com elevadíssimo número de *routers* ou não, este deverá saber o que fazer sempre que haja uma falha.

Como tal, o mesmo algoritmo foi implementado tendo em conta a existência da mesma estrutura de dados enunciada na subsecção anterior, a matriz bidimensional com as rotas entre *end-user areas*, sendo que na eventualidade de uma falha de um *link*, esta será reportada ao controlador que, através do módulo de notificações já embebido no mesmo, receberá uma notificação de falha do *link* em questão e, consequentemente, deverá retirá-lo da topologia e efetuar novos cálculos sobre a mesma, guardando o resultado dos cálculos nesta mesma estrutura de dados, tal como apresentado no Algoritmo 2.

Algoritmo 2: Notificação de falhas de links ao controlador SDN.

Input: nenhum
Output: matriz bidimensional com rotas correspondentes

```
1  inicialização;
2  ativação de módulo de notificações;
3  if um qualquer link falha then
4  |  falha é reportada ao controlador;
5  |  atualizar topologia eliminando o link;
6  |  aplicar algoritmo de Dijkstra (Algoritmo 1) sobre topologia nova;
7  |  guardar nova matriz bidimensional de caminhos mais curtos;
8  else
9  |  continua em execução.
10 end
```

A execução deste algoritmo é feita em permanência enquanto que o controlador e a solução estiverem em funcionamento, sendo que reportará quaisquer falhas na topologia ao controlador constantemente, atuando sobre as mesmas imediatamente após a falha do *link* se constatar.

3.2.2 *Convergência Imediata após Falhas de Links*

Estando a solução base já completa, partiu-se, desde este ponto, para a melhoria e aumento da complexidade e do número de características disponibilizadas pela mesma. Como tal, e sendo o mecanismo anterior útil para lidar com quaisquer falhas, peca pelo seu custo e tempo elevado gasto nos cálculos necessários efetuados em paralelo, sendo que a resposta dada às mesmas falhas não é em tempo real (isto é, os cálculos para o mesmo são feitos imediatamente após a falha), característica esta que se torna pouco apelativa para um administrador de rede. Assim, e de modo a responder a esta problemática, partiu-se para a melhoria do mesmo mecanismo, tentando, deste modo, fazer com que o mesmo mecanismo pudesse responder imediatamente após a falha.

Desta forma, e sendo que os cálculos necessários para a nova topologia têm de ser feitos de qualquer modo, seguiu-se a metodologia de, *offline* e antecipadamente calcular as demais novas rotas para a falha de qualquer *link* da topologia, armazenando o resultado da mesma numa outra estrutura de dados. Deste modo, mal o controlador se aperceba que o *link* falhou, pode imediatamente injetar a nova configuração, previamente calculada, e responder de forma mais eficaz.

A aplicabilidade deste mesmo algoritmo remete para uma melhoria no tempo de convergência após essas mesmas falhas, sendo de responsabilidade do controlador responder

às mesmas em qualquer altura. Desta forma não se perde tempo durante o normal funcionamento da topologia para o cálculo das mesmas rotas, podendo-se, assim, fazer o cálculo das mesmas inicialmente, após a execução da solução.

O algoritmo em questão remete para o Algoritmo 1, o algoritmo de Dijkstra adaptado, sendo que a execução do mesmo é feita tendo em conta o número de *links* existentes, ou seja, é executado o algoritmo de Dijkstra adaptado por cada vez que se simula a falha de cada um dos links existentes. Estas execuções serão armazenadas num *array* de matrizes bidimensionais, estruturalmente semelhantes à matriz bidimensional do primeiro algoritmo (ou seja, possuem as demais rotas entre *end-user area* origem e *end-user area* destino, mas sem o *link* que terá falhado), dado pela figura 3.3:

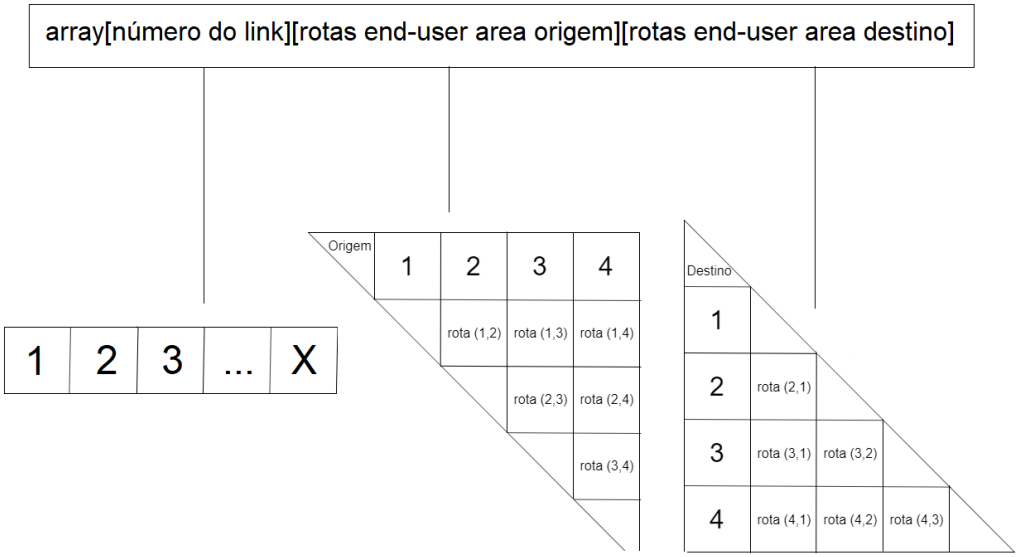


Figura 3.3.: Estrutura global do *array* de matrizes dimensionais.

Para cada uma das posições do mesmo *array*, simula-se a falha do *link* respetivo (por exemplo, na posição 1 está armazenada uma matriz bidimensional com as mesmas rotas, assumindo que o *link* 1 não está funcional). O *array* resultante pode ser observado na Figura 3.4:

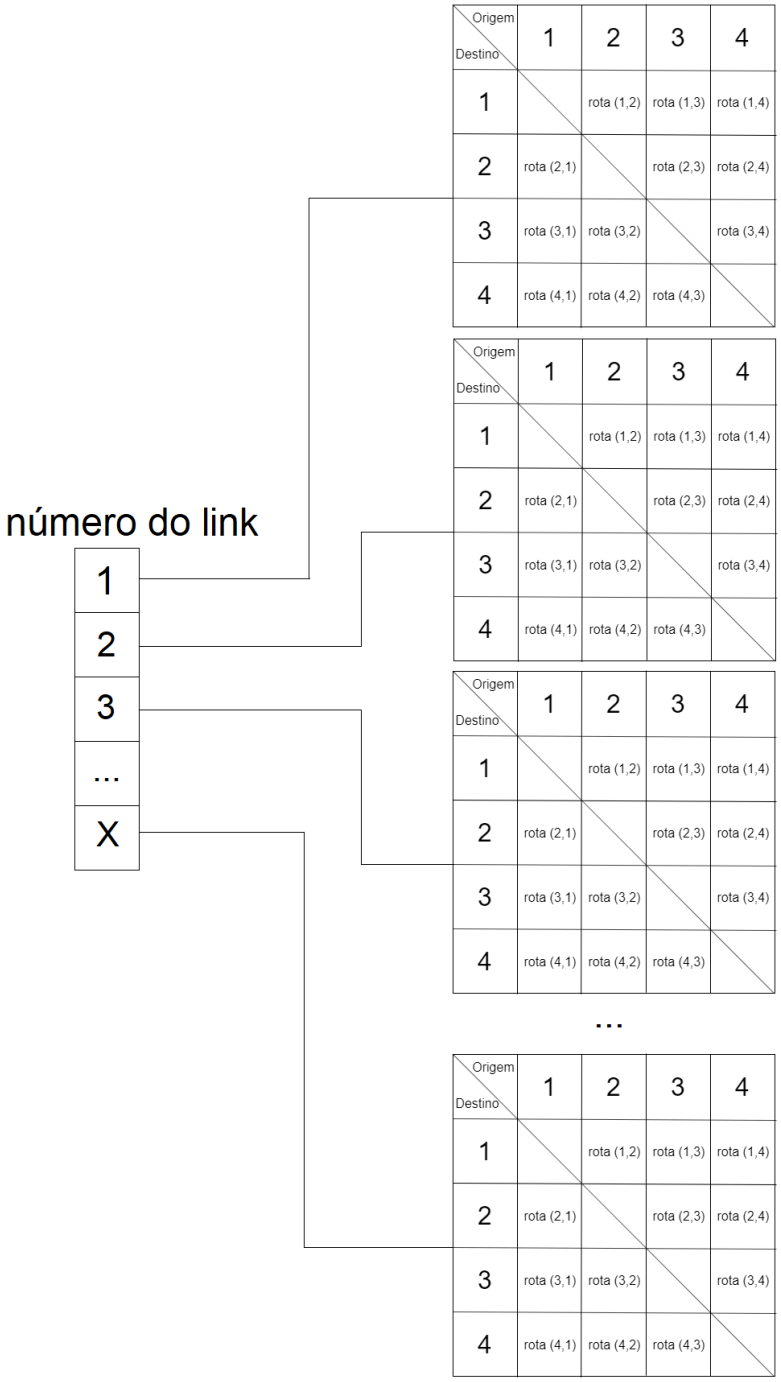


Figura 3.4.: Estrutura pormenorizada do *array* de matrizes dimensionais, relativa à simulação de falhas nos *links*.

O cálculo da nova matriz bidimensional é feito, também, imediatamente após a execução inicial da topologia, guardando-a numa estrutura de dados que é constantemente atua-

lizada até que o módulo seja forçosamente parado. Este algoritmo pode ser dado pelo seguinte pseudo-código, representado pelo Algoritmo 3:

Algoritmo 3: Algoritmo de cálculo de novas rotas.

Input: nenhum
Output: *array* de matrizes bidimensionais com rotas pré-calculadas

```
1  inicialização;
2  for cada link na topologia do
3      seleciona link atual;
4      for cada end-user area X na topologia do
5          executa algoritmo de dijkstra adaptado desde end-user area X até qualquer
             outra end-user area, com a inexistência do link dado por input na topologia;
6          injeta em matriz bidimensional o resultado das novas rotas calculadas;
7          escolhe outra end-user area destino;
8      end
9      injeta resultado no array de matrizes bidimensionais, na posição do número do
         link escolhido;
10     simula falha de outro link;
11 end
12 guardar resultado do preenchimento em array de matrizes bidimensionais.
```

Depois, e consoante haja uma qualquer falha de um link, o controlador pode remeter para este *array* de matrizes bidimensionais para recuperar as novas rotas, injetando-a imediatamente nas tabelas de encaminhamento da topologia, diminuindo, assim, o impacto que a falha do mesmo link causa no normal funcionamento de toda a rede.

3.2.3 *Proteção de Tráfego de Entidades da Infraestrutura*

Com o desenvolvimento deste mecanismo pretende-se a possibilidade de um administrador de rede conseguir proteger determinados elementos da rede, tais como *routers* e *links*, do tráfego proveniente das *end-user areas*, por um determinado período de tempo especificado. Para que tal funcione de forma correta, deverão ser calculadas as rotas que não passem nos *links* especificados no *input* do mesmo mecanismo, injetando regras de encaminhamento que tentem desviar o tráfego de modo a que se evite o mesmo *link*. Estas regras, porém, só devem ser injetadas se não causarem partições no grafo. Caso causem partições, opta-se pela mínima passagem de tráfego pelo mesmo, tentando, deste modo, evitar o *link* ao máximo.

A utilidade deste mecanismo reside no facto de possibilitar, por exemplo, a reserva de rotas para tráfego especial requisitado por clientes e/ou empresas, para eventuais transferências de dados em grande escala entre plataformas de *cloud*, para operações de manutenção de infraestruturas, entre outros exemplos. O mecanismo também permite isolar *links* em específico sem quebrar o normal funcionamento da rede, característica que se demonstra útil em cenários equiparáveis ao de um ISP.

Este mecanismo inicia toda a sua lógica algorítmica através do *input* que recebe, que contém quer um *link* quer um *router* a proteger. Caso o *input* seja um *link*, este mecanismo tratará de verificar se a sua remoção causa partição na topologia, através da simulação de remoção do mesmo com recurso ao *array* de matrizes bidimensionais do Algoritmo 3. Caso este cause alguma partição na topologia, o mecanismo ignora a remoção temporária do mesmo e tenta evitar o *link* durante o período de tempo especificado. Caso não cause nenhuma partição na topologia, o controlador injetará rotas correspondentes à falha do *link* durante o período de tempo enunciado.

Para o caso de o *input* se tratar de um *router*, o controlador verificará se a remoção temporária do *router* causa alguma partição. Em caso positivo, e tal como para com os *links*, o controlador ignorará a remoção do mesmo e tentará utilizá-lo o mínimo possível, durante o período de tempo enunciado. Em caso negativo, o controlador removerá o *router* das tabelas de encaminhamento dos *routers* que lhe são adjacentes durante o período de tempo enunciado, guardando as rotas removidas em memória para depois reinjetá-las.

Após expirar o período de tempo durante o qual se pretendia proteger o recurso da rede, e caso se trate de um *link* como *input*, o controlador removerá as rotas injetadas. Caso se trate de um *router*, o controlador injetará as demais rotas removidas dos *routers* que lhe são adjacentes, reestabelecendo a topologia. Este algoritmo poderá ser verificado no seguinte Algoritmo 4:

Algoritmo 4: Algoritmo de proteção de tráfego de certos pontos da infraestrutura.

```
   Input: link ou router a proteger, período de tempo
   Output: nenhum
 1 inicialização;
 2 switch input do
 3     case input é link do
 4         for router que contenha o link do
 5             verifica array de matrizes bidimensionais para simular a falha;
 6             if remoção do link causa partição no grafo then
 7                 ignora pois causa partição do grafo;
 8             else
 9                 injeta rotas correspondentes à falha do link com prioridade maior
                    durante período de tempo especificado;
10             end
11         end
12     end
13     case input é router do
14         if remoção do router causa partição then
15             ignora a remoção do router;
16         else
17             remove o router das tabelas de encaminhamento dos routers adjacentes
                durante o período de tempo especificado;
18             simula eliminação dos links que estão ligados ao router;
19             executa novamente o Algoritmo 1 com os links anteriores removidos e
                injeta novas rotas;
20         end
21     end
22 end
23 adormece durante período de tempo especificado no input;
24 remove rotas correspondentes à falha do link com prioridade maior.
```

Estas rotas protegidas serão instaladas imediatamente após receção de instruções para proteção desses mesmos recursos da rede, perdurando até ao período de tempo que lhes foi atribuído. De cada vez que se queira proteger um *link*, deverá ser feita uma chamada à execução deste mesmo mecanismo com o respetivo *input*.

3.2.4 *Reatividade a Níveis de Congestão*

Como as rotas geradas pelo algoritmo de Dijkstra são, e na eventualidade de nenhum *link* falhar, as mais utilizadas na topologia, estas poderão ser propícias a elevados volumes de tráfego, levando à possibilidade de uma eventual congestão da rede ou dos *links*. Estando nesta situação, a topologia pode apresentar degradação do seu serviço. Desta forma, o mecanismo implementado para lidar com este problema passa pela reatividade do controlador a diferentes níveis de congestão, inicialmente estipulados pelo administrador da rede. Como tal, deverá alterar algumas rotas de forma a evitar o *link* congestionado, até que o mesmo deixe de ficar congestionado. O mecanismo receberá, então, duas *end-user areas* origem e destino, respetivamente dadas como *input*. Através destas, o controlador deve evitar o caminho mais curto inicialmente calculado pelo Algoritmo 1 com o intuito de não utilizar o *link* congestionado, seguindo, caso seja possível, a nova rota temporariamente calculada para diminuição da congestão desse mesmo *link*. Caso não seja possível, este deverá tentar utilizar esse mesmo *link* o mínimo de vezes possível, tentando minimizar a circulação de tráfego pelo mesmo.

A utilidade deste mesmo mecanismo está na melhoria de qualidade do serviço na topologia, evitando que se remeta para a passagem de tráfego por *links* congestionados. Através deste, pretende-se evitar que os *links* que o algoritmo de Dijkstra gerou, na matriz bidimensional das rotas, sejam bombardeados por tráfego que provenha das demais *end-user* areas, usando-se outros *links* menos utilizados.

Para que este mecanismo funcione de forma correta, são fornecidas duas *end-user areas*, sendo uma a origem e a outra o destino a seguir, na gama de endereços /24. É também fornecido um valor base, dado pelo administrador de rede, com o qual se considerará que é o valor mínimo para que um *link* seja assinalado como congestionado. De seguida, o mecanismo constatará se, na rota entre as *end-user areas* do *input*, há algum *link* com congestão, verificando se qualquer um dos *links*, um a um, apresenta uma taxa de congestão superior à fornecida. Na eventualidade de isto acontecer, como visto no Algoritmo 5, são recalculadas novas rotas entre as mesmas *end-user areas* através do algoritmo de falhas de *links*, isto é, assumindo que o *link* em questão falha. Estas novas rotas serão injetadas no novo caminho a seguir, com prioridade sobre as anteriores. Estas mesmas rotas serão apenas utilizadas caso haja obrigatoriamente um *match* entre a origem e o destino fornecidos como *input* na tabela de encaminhamento que cada um dos *routers* apresenta. Se, por ventura, apenas exista um *match* na origem ou no destino (e não nos dois em simultâneo), as rotas seguidas serão as mesmas que as calculadas pelo Algoritmo 1, como expectável.

Algoritmo 5: Algoritmo de reatividade a níveis de congestão.

Input: *end-user areas* origem e destino /24, percentagem de congestão base considerada pelo administrador de rede;
Output: nenhum

```
1  inicialização;
2  for rota entre end-user area origem e destino especificada no input do
3      if próximo link apresenta congestão inferior ao valor do input then
4          continua;
5      else
6          if remoção do link quebra a topologia then
7              ignora link e segue para o próximo;
8          else
9              re-calcula rotas entre end-user areas origem e destino dadas no input,
                assumindo a falha do link em causa;
10             injeta como prioritárias novas rotas calculadas nos routers que fazem
                parte do novo caminho, apenas caso a origem seja da end-user area
                origem para a end-user area destino;
11         end
12     end
13     próximo salto da rota.
14 end
```

A utilidade deste mesmo mecanismo reside no facto de, na eventualidade de existir qualquer diminuição na qualidade do serviço prestado, possa responder de forma pronta e adequada a quaisquer taxas elevadas de congestão, levando à não-degradação contínua do mesmo. Deste modo, pretende-se que os *links* mais utilizados para encaminhamento de tráfego, calculados pelo controlador, não sejam tão saturadas quanto seriam sem qualquer intervenção do controlador.

Este mecanismo, para funcionar de modo correto, fez uso do módulo de estatísticas presente no controlador utilizado, que funciona do modo enunciado na seguinte subsecção.

Estatísticas de Tráfego

Este módulo, já embebido no controlador `Floodlight`, dado pelo Algoritmo 6, possibilita não só a averiguação de quaisquer estatísticas relativas à componente física da rede (ligações, estado dos demais *switches*, estado dos demais *links*), mas também a averiguação de estatísticas de tráfego, tais como o estado dos *links*, *uptime*, o estado do controlador, entre outros.

Algoritmo 6: Recolha de estatísticas de tráfego através do módulo embebido no Floodlight.

Input: nenhum
Output: hashmap com estatísticas

```
1 inicialização;
2 while em execução do
3     percorre todos os routers;
4     recolhe dados acerca dos mesmos routers;
5     armazena resultados em variável local;
6     adormece durante tempo definido até à próxima recolha de dados.
7 end
```

É de salientar que, apesar de a impressão das estatísticas ser em tempo real, as estatísticas apresentadas não são, porém, em tempo real. Estas estatísticas são meros *snapshots* que apresentam informação verdadeira em relação ao momento em que foram computados, indicando informações relativas à métrica, fluxo, portas, tabelas, entre outros.

3.2.5 *Rotas Individuais para Fluxos de Tráfego*

O mecanismo presente nesta subsecção funciona para a instalação de rotas específicas para determinados fluxos de tráfego. Assumindo que, por ventura, o administrador de rede recebe um pedido de instalação de rotas particulares para IP origem/destino da gama /32 durante um período de tempo especificado no *input* fornecido, estas deverão, por si só, ter prioridade sobre qualquer outras regras de encaminhamento estipuladas nas respetivas tabelas de encaminhamento dos demais *routers*. Este mecanismo, desta forma, permite a criação dessas mesmas regras e consequente injeção nos *routers* envolvidos.

O benefício da utilização deste mecanismo será o de tratamento, quer para empresas ou particulares, diferenciado (e melhor) na rede, tendo uma rota específica entre dois pontos da mesma. Este mecanismo possibilita, também, transferências especiais requisitadas com antecedência, característica útil para, por exemplo, transferência de dados entre *data centers*, que requeiram que o tráfego entre dois pontos seja o ideal.

Para implementação deste mecanismo, é necessária a especificação do IP origem e IP destino no *input*, assim como a rota a seguir, estipulada pelo administrador da rede. Este, através da rota enunciada, seguirá, *router* a *router*, e injetará em cada um destes as regras preferenciais de encaminhamento de tráfego, tendo em conta o IP origem e IP destino. Estas rotas injetadas terão, consequentemente, prioridade sobre as rotas das subsecções anteriores, na gama de endereços /24, sendo, deste modo, as rotas com maior preferência no sistema implementado. Após injeção das mesmas, o mecanismo fica em espera até que

o período de tempo acabe, removendo as mesmas regras injetadas. Este algoritmo pode ser verificado através do pseudo-código enunciado no Algoritmo 7:

Algoritmo 7: Algoritmo de especificação de rotas individuais para fluxos de tráfego.

Input: IP_Origem, IP_Destino, rota e período de tempo
Output: nenhum
1 inicialização;
2 analisar rota dada como *input* e injetar regras de encaminhamento durante o período de tempo;
3 **for** *qualquer tráfego que provenha de IP_Origem para IP_Destino* **do**
4 | seguir rota injetada ao invés de rota *default*;
5 **end**
6 adormece até que período de tempo acabe;
7 remove rotas injetadas anteriormente.

As rotas deste mesmo algoritmo serão, assim, as primeiras a ser vistas na tabela de encaminhamento, verificando, primeiro, a existência de *match* entre o IP origem e IP destino. Após término do período de tempo especificado pelo *input*, estas serão, como anteriormente referido, removidas, voltando a topologia ao estado normal de funcionamento, com as rotas previamente instaladas. Caso o administrador de rede queira adicionar novas rotas protegidas, deverá executar este mecanismo múltiplas vezes, com uma execução por rota a adicionar.

Existe a possibilidade de completar este mecanismo com critérios específicos de averiguação, tal como a latência baixa entre os *routers* estipulados. Porém, e à data de escrita, o controlador SDN utilizado não permite o acesso a essas variáveis, dificultando, deste modo, o enriquecimento da implementação do mesmo.

3.2.6 *Multiplexagem da Topologia Física por diferentes Redes Virtuais*

Caso um administrador de rede seja confrontado com cenários de alguma escassez de recursos, ou simplesmente queira optimizar os seus recursos da rede distribuindo os mesmos por várias redes virtuais independentes, o mecanismo representado nesta subsecção permite a partição de uma topologia em múltiplas redes, cada uma destas operando independente das outras. O controlador receberá, por *input*, os *switches* que constituem várias partições, sendo que estas diferentes partições não deverão comunicar entre si. Cada uma das partições funciona tal como se tratasse de uma sub-rede presente na mesma topologia, mas que não tem acesso às outras sub-redes particionadas da topologia física. Como tal, e aplicando o mesmo conceito aos conjuntos numéricos, a união das demais sub-redes deverá

resultar num subconjunto menor ou igual à totalidade da topologia (visto que podem ser deixados *switches* de fora de qualquer sub-rede).

A utilidade deste mecanismo remete para a possibilidade de alocação específica de recursos da rede a cada uma das partições, como por exemplo, se uma das partições for dedicada a uma infra-estrutura de suporte a um ambiente de Internet-of-Things (IoT), é necessária uma alta disponibilidade da rede, assim como níveis de latência e taxas de transferência de dados específicas. A partição da topologia permite serviços facilmente configuráveis pelo administrador da rede, de forma a corresponder a diferentes requisitos impostos pelos demais utilizadores da rede administrada pelo mesmo.

Para que este mecanismo fosse implementado, remeteu-se à topologia inicial, ainda antes do cálculo dos caminhos mais curtos pelo algoritmo de Dijkstra. Após receção da topologia o controlador verifica as *slices* (ou partições), através de um *array* que contenha as mesmas partições dadas por *input*, e procede para a separação da rede consoante o que terá sido fornecido como *input*. De seguida, e após partição da mesma rede, executa-se o algoritmo de Dijkstra sob essa partição e são injetados os caminhos mais curtos específicos a essa partição, prosseguindo com o mesmo processo por cada partição até que não hajam mais partições dadas pelo *input*. Na Figura 3.5 observa-se uma topologia exemplo, enquanto que na Figura 3.6 se verifica a partição da mesma em duas sub-redes. Nesta última Figura, a partição A da rede está rodeada por uma linha vermelha, e a partição B da rede está rodeada por uma linha azul:

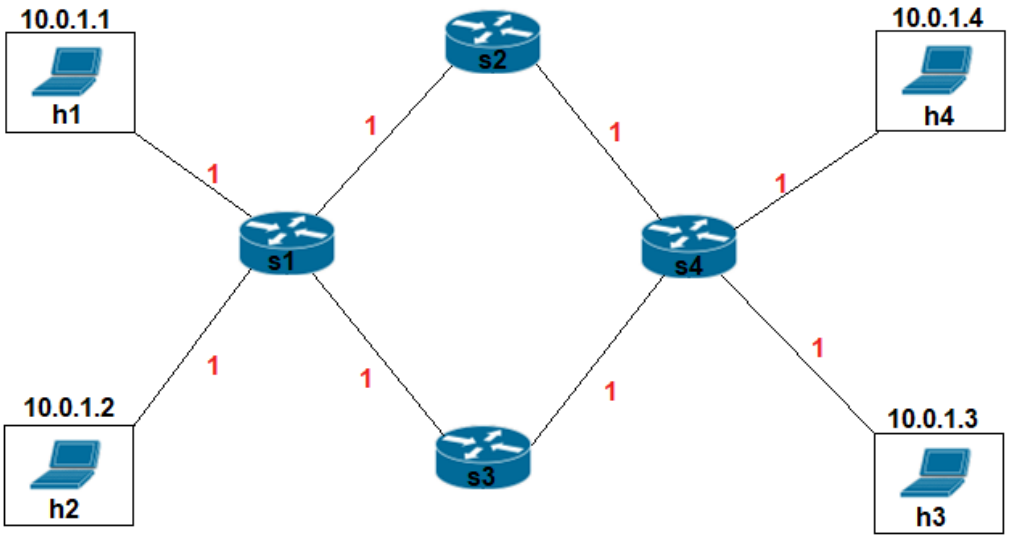


Figura 3.5.: Topologia exemplo para particionar em várias partições.

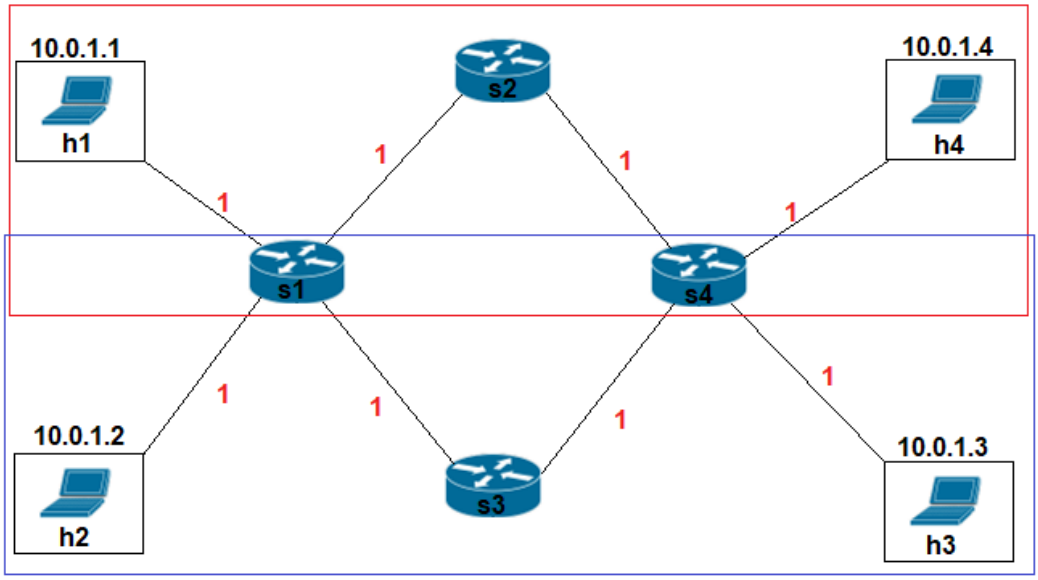


Figura 3.6.: Topologia com partições escolhidas pelo administrador de rede.

Com as partições já escolhidas pelo administrador de rede, o mecanismo procederá para a partição das mesmas, separando-as em duas sub-redes, dadas pelas Figuras 3.7 e 3.8, que representam as partições superior e inferior, respetivamente, da mesma topologia:

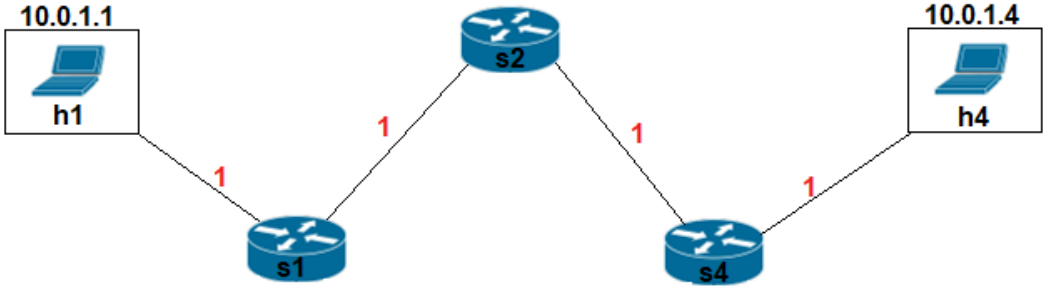


Figura 3.7.: Partição superior da topologia anteriormente enunciada.

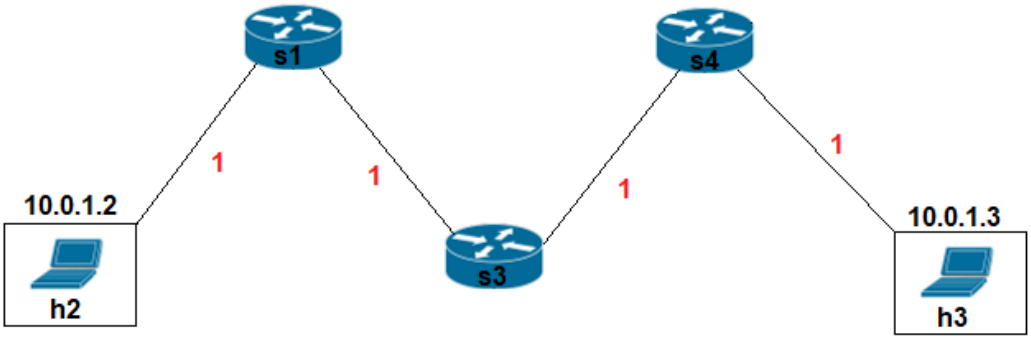


Figura 3.8.: Partição inferior da topologia anteriormente enunciada.

Estas, deste modo, funcionam como duas sub-redes completamente diferentes, sendo que cada uma destas não sabe da existência da outra, mesmo apesar de estarem inseridas na mesma topologia física e partilharem alguns *switches* (s1 e s4). As regras de encaminhamento injetadas nas tabelas de encaminhamento dos *switches* são a base do correto funcionamento deste mecanismo, sendo que estas são as que efetivamente particionam a topologia física. Após partição das mesmas, é executado o algoritmo de Dijkstra sobre cada uma das partições, sendo obtidos e injetados os caminhos mais curtos respetivos a cada sub-rede. O *input* dado é sob forma de um *array*, sendo que cada elemento do mesmo possui o caminho, através de *switches*, a ser percorrido para criação de cada partição. Neste incluem-se também, no mesmo elemento do *array*, as várias *end-user areas* que pertençam à respetiva partição. A adição da *end-user area* ao *array* é particularmente necessária para os casos em que há partilha do mesmo *switch* entre partições diferentes (por exemplo, na Figura 3.6 a partição A e a partição B ambas partilham os *switches* s1 e s4), sendo que, sempre que haja passagem de tráfego pelos *switches*, será feito o *match* entre origem e destino das *end-user areas* com a sua gama de endereços /24, para que haja distinção do tráfego e os *switches* façam a respetiva separação correta do mesmo. O pseudo-código resultante deste mecanismo pode ser dado pelo Algoritmo 8:

Algoritmo 8: Algoritmo de seguimento de rotas individuais para fluxos de tráfego.

Input: topologia, custos dos *links* e *array* com *switches* que constituem cada partição
Output: nenhum

```
1 inicialização;
2 for cada elemento do array com partições da topologia do
3 |   executar algoritmo de Dijkstra sob os elementos que constituem uma partição
  |    dada;
4 |   injetar caminhos mais curtos gerados pelo algoritmo de Dijkstra em cada
  |    partição, com regras de match incluindo as end-user areas origem e destino;
5 end
```

Este algoritmo apenas poderá ser executado se qualquer outro mecanismo não tiver sido executado antes da inicialização de toda a solução, ou seja, não é possível a partição das redes durante a execução da solução. Como tal, as partições da topologia deverão ser previamente enunciadas pelo administrador da rede no *array* anteriormente mencionado, antes do início da mesma solução. Caso isto se verifique, podem-se depois, porventura, executar os mecanismos anteriormente descritos nas secções anteriores a qualquer partição definida por este mecanismo.

3.3 SUMÁRIO

Neste capítulo foi enunciada a arquitetura a utilizar para desenvolvimento dos demais mecanismos implementados. Na arquitetura, foram referidas as demais entidades que a caracterizam, assim como a topologia que a incorpora. Foram também referidos os intervenientes da mesma, constatando o administrador de rede responsável pela manutenção e execução dos demais mecanismos.

Em relação aos mecanismos implementados, foi feita uma exposição do seu propósito e porque terá surgido a ideia da sua implementação. Para cada um destes, foi referida, também, a respetiva estrutura de dados que é construída com o mesmo. Mencionou-se, em cada um, a sua utilidade e a sua implementação e lógica algorítmica, quer por extenso quer através de pseudo-código.

4

TESTES E ANÁLISE DE RESULTADOS

Neste capítulo será demonstrada toda a bateria de testes utilizada para validação da solução implementada, assim como a análise feita sobre estes. Inicialmente mencionar-se-ão as tecnologias utilizadas ao longo de todo este projeto (secção 4.1), passando, de seguida, pela exemplificação do caso de estudo em que a totalidade da solução se enquadra (secção 4.2). Passar-se-á para a amostragem e análise de resultados (secção 4.3), finalizando com um sumário de todo o capítulo (secção 4.4).

4.1 TECNOLOGIAS UTILIZADAS

Para implementação de toda a solução e dos mecanismos que a constituem, foram usadas as seguintes tecnologias, obrigatórias para garantir o correto funcionamento da mesma:

- **Sistema Operativo Windows 10**, capaz de executar os programas requeridos para o normal funcionamento de toda a aplicação, assim como a simulação de toda a rede emulada;
- **Oracle Virtual Machine (VM) Virtualbox**, componente necessária para virtualizar uma imagem de formato .iso, com a utilidade de permitir a utilização do sistema operativo `Ubuntu`. Este programa possibilita a junção do mesmo com o utilitário Mininet, possibilitando, deste modo, que seja criada uma rede fictícia capaz de recriar uma infraestrutura fidedigna;
- **Sistema Operativo Ubuntu 16.04 Long Term Support (LTS)**, necessário para execução em paralelo com o utilitário `Mininet`, assim como as diversas componentes requeridas para desenvolvimento da solução;
- **Mininet**, que possibilita a criação de uma rede virtual de forma instantânea de forma a que seja facilitado o desenvolvimento de topologias prontas a testar, através de uma linguagem de programação que se adeque a tal;
- **Python**, linguagem de programação utilizada para especificar algumas topologias-teste de menor escala, assim como a topologia final utilizada para validação da

solução. Esta linguagem de programação apresenta, também, bibliotecas pré-feitas que conseguem abarcar os demais conceitos e possibilitam a adição de *links*, *switches* e várias características;

- **JAVA**, linguagem de programação utilizada na implementação de toda a solução, sendo esta a linguagem de programação utilizada pelo controlador `Floodlight`. Esta linguagem tem um atraso inicial no arranque da sua Java Virtual Machine (JVM) para preparar as demais componentes, mas apresenta vantagens na sua robustez;
- **Floodlight**, controlador Software-Defined Networking (SDN) comunitário e apoiado pela Big Switch Networks (BSN), com inúmeras funcionalidades implementadas pela mesma fundação. Este controlador, tal como descrito anteriormente, demonstra a sua aplicabilidade pelo facto de, nos presentes dias, estar em constante desenvolvimento e manutenção, sendo novas características adicionadas ao longo do tempo;
- **Eclipse**, ambiente de desenvolvimento integrado que melhor se integra com o controlador `Floodlight`, também usado para desenvolvimento de toda a solução. Este ambiente de desenvolvimento foi escolhido única e simplesmente pela razão de que vem embebido com a instalação do mesmo Floodlight, facilitando, assim, a sua execução na criação dos arquivos de Java (Java Archive (JAR));

4.2 CASO DE ESTUDO

Esta secção tem como objetivo a descrição dos demais casos de estudo definidos para o teste e validação dos mecanismos implementados e que foram descritos no capítulo anterior. Cada um destes casos de estudo identifica a funcionalidade de cada um dos mecanismos, assim como o conjunto de testes utilizados para comprová-los. Estes mecanismos serão de fácil manipulação e edição, garantindo, também, em cenários futuros, que seja facilmente adaptável a qualquer circunstância.

De forma a que os resultados obtidos apresentassem o máximo de fidedignidade, recorreu-se à utilização do utilitário Mininet, cujas capacidades foram enunciadas anteriormente. Este utilitário permitiu a criação da seguinte topologia, descrita na Figura 4.1:

```
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h21 h22 h23 h31 h32 h33 h41 h42 h43
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19
*** Adding links:
(s1, h11) (s1, h12) (s1, h13) (s1, s5) (s2, h21) (s2, h22) (s2, h23) (s2, s18) (s3, h31)
 (s3, h32) (s3, h33) (s3, s19) (s4, h41) (s4, h42) (s4, h43) (s4, s6) (s5, s6) (s5, s7)
(s5, s10) (s5, s11) (s6, s7) (s6, s13) (s6, s14) (s7, s8) (s7, s9) (s8, s11) (s9, s13) (
s10, s18) (s11, s12) (s11, s15) (s11, s18) (s12, s13) (s13, s16) (s13, s19) (s14, s19) (
s15, s17) (s16, s17) (s17, s18) (s17, s19) (s18, s19)
*** Configuring hosts
h11 h12 h13 h21 h22 h23 h31 h32 h33 h41 h42 h43
*** Starting controller
c0
*** Starting 19 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 ...
*** Starting CLI:
mininet>
```

Figura 4.1.: Arquitetura textual da rede implementada em Mininet.

Nesta topologia, os *switches* são enunciados pela letra `S` e os *hosts* pela letra `H`, sendo que no caso dos *hosts*, a sua nomenclatura tem em conta a sua *end-user area*, tal que o `X` em `hXY` corresponde ao número da sua *end-user area*. Por exemplo, e um *host* que esteja representado pela *string* `h13` corresponderá ao terceiro *host* da *end-user area* `1`. Esta topologia, quando visualizada graficamente, apresenta-se como ilustrado na Figura 4.2:

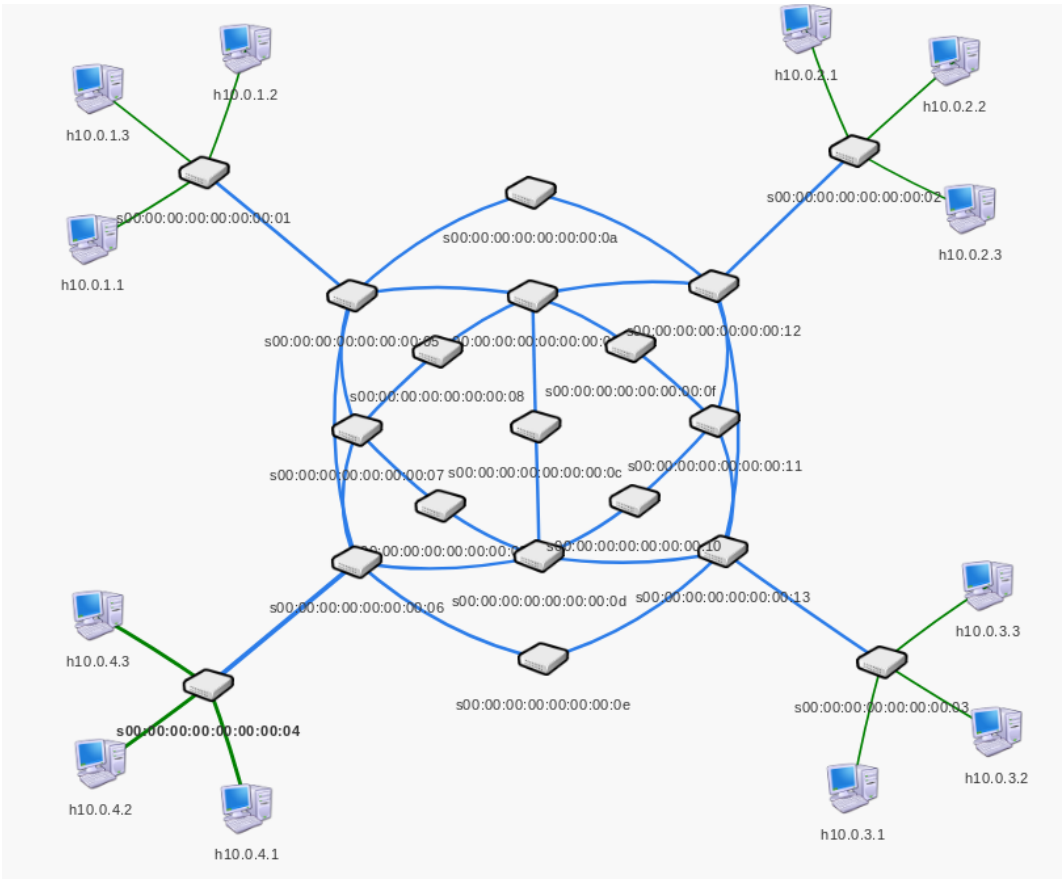


Figura 4.2.: Arquitetura gráfica da rede implementada em Mininet.

Nos casos de estudo apresentados nas secções seguintes, verificar-se-ão os resultados obtidos após execução de um conjunto de testes. Estes serão especificados por cada mecanismo proposto, dividindo a próxima secção em várias subsecções.

4.3 Análise de Resultados

Para cada um dos mecanismos implementados, foram feitos testes relevantes às suas capacidades, sendo cada um dos testes único a cada um dos mecanismos. Deste modo, os resultados terão o intuito de comprovar a aplicabilidade de cada uma das funcionalidades, com uma subsecção por mecanismo proposto. Como tal, serão enunciados, nas subsecções seguintes.

4.3.1 *Algoritmo de Dijkstra*

Para que fosse possível comprovar a funcionalidade geral de toda a solução de encaminhamento de tráfego, seria obrigatório, como primeiro passo, verificar se as rotas geradas pelo algoritmo de Dijkstra seriam, de facto, as corretas. Como tal, e após a configuração inicial e a execução do algoritmo, seguido do preenchimento da variável local armazenada em memória com o grafo da topologia, foram feitos os seguintes testes através do comando `ping`, para verificar que o tráfego enviado seria através dos caminhos mais curtos de nível 3, por IP destino dos pacotes enviados.

Testando, para uma *end-user area* inicial, a *end-user area* 10.0.3.X, até às outras *end-user areas*, observa-se, delineado a verde, a rota seguida pelo tráfego na Figura 4.3 que corresponde ao caminho mais curto calculado pelo Algoritmo 1, tendo em conta os custos dos *links* e para o caso em que a *end-user area* destino era a 10.0.1.X:

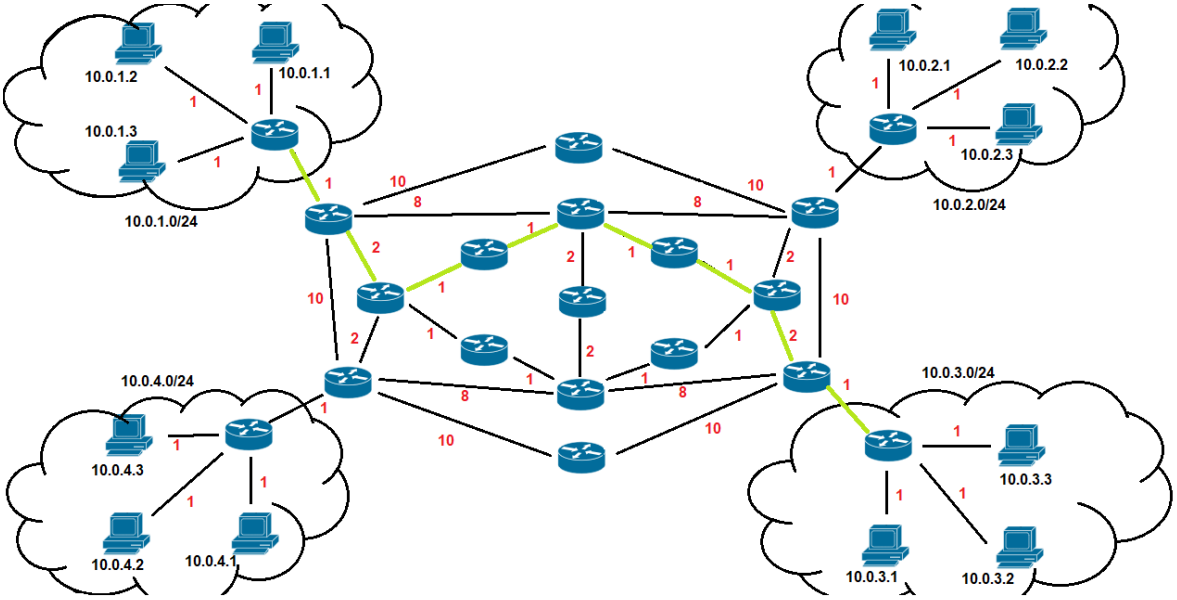


Figura 4.3.: Teste de `ping` entre *hosts* das *end-user areas* 10.0.3.X e 10.0.1.X.

Sendo este comprovado pelo seguinte `traceroute` da Figura 4.4:

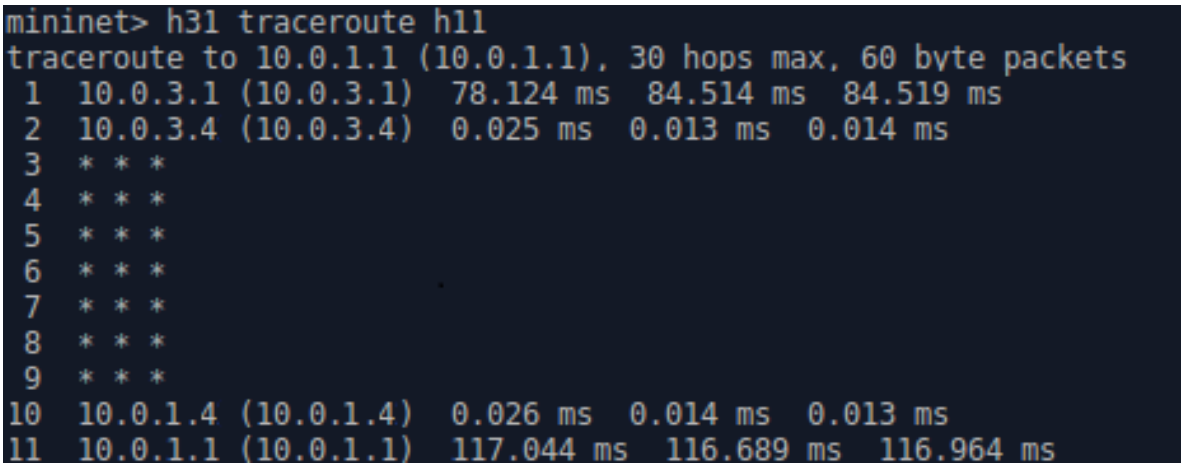


Figura 4.4.: Teste de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.1.X.

Este *traceroute*, feito no Mininet, apresenta, porém, alguns problemas. Como é possível verificar pela Figura 4.4, o mesmo apresenta apenas os *routers* das *end-user areas*, sendo que as restantes interfaces são representadas com múltiplos asteriscos. Constata-se, por outro lado, que o número de saltos é o correto, sendo a informação revelada por esta figura ainda relevante. Desta forma, e para se verificar o caminho correto percorrido, remete-se para o controlador, que imprime os endereços dos *switches* por onde passa o tráfego, apresentado pela Figura 4.5:

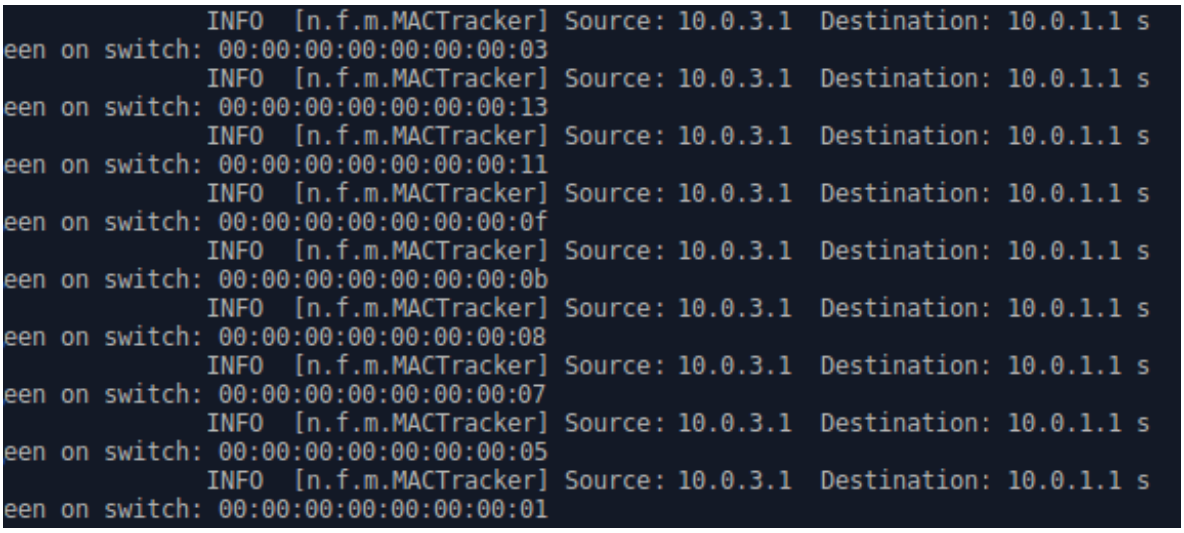


Figura 4.5.: Verificação de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.1.X.

Os endereços Media Access Control (MAC) que são aqui amostrados remetem para os mesmos dos respetivos *switches* da Figura 4.2, que correspondem ao caminho mais curto calculado pelo algoritmo de Dijkstra. Deste modo, e tal como para a Figura 4.3, no caso das figuras seguintes que apresentem uma topologia, será amostrado o caminho percorrido a verde, demonstrando os endereços dos *switches* pelos quais o tráfego terá passado.

Para o caso em que o tráfego enviado provinha da *end-user area* com gama de endereços 10.0.3.X para a *end-user area* destino com gama de endereços 10.0.2.X, verificou-se a passagem pela seguinte rota, elucidada pela Figura 4.6:

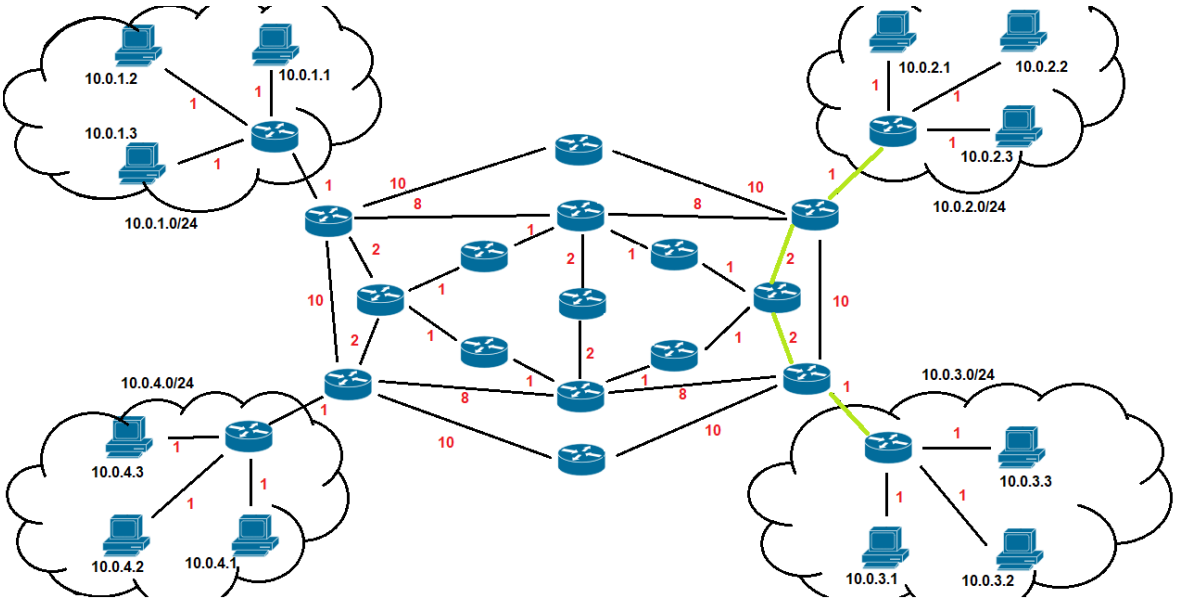


Figura 4.6.: Teste de `ping` entre as *end-user areas* 10.0.3.X e 10.0.2.X.

De forma a comprovar que este terá sido o caminho mais curto calculado pelo algoritmo de Dijkstra e seguido na topologia, remeteu-se para um novo teste de *traceroute*, o qual é possível ser verificado pela Figura 4.7:

```
mininet> h31 traceroute h21
traceroute to 10.0.2.1 (10.0.2.1), 30 hops max, 60 byte packets
 1  10.0.3.1 (10.0.3.1)  30.134 ms  34.808 ms  96.742 ms
 2  10.0.3.4 (10.0.3.4)  0.030 ms  0.017 ms  0.017 ms
 3  * * *
 4  * * *
 5  * * *
 6  10.0.2.4 (10.0.2.4)  0.129 ms  0.214 ms  0.014 ms
 7  10.0.2.1 (10.0.2.1)  21.761 ms  50.192 ms  50.362 ms
```

Figura 4.7.: Teste de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.2.X.

Estando este teste sob os mesmos problemas que o teste de *traceroute* anterior, procedeu-se a uma nova verificação da passagem do tráfego através do controlador, de forma a obter os endereços MAC dos *switches* pelos quais teria passado. Como tal, obtém-se o seguinte, visto na Figura 4.8:

```
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.2.1 s
een on switch: 00:00:00:00:00:00:00:03
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.2.1 s
een on switch: 00:00:00:00:00:00:00:13
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.2.1 s
een on switch: 00:00:00:00:00:00:00:11
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.2.1 s
een on switch: 00:00:00:00:00:00:00:12
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.2.1 s
een on switch: 00:00:00:00:00:00:00:02
```

Figura 4.8.: Verificação de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.2.X.

Estando já duas das três *end-user areas* testadas, resta a execução do teste para a última *end-user area* ainda não verificada, a de gama de endereços 10.0.4.X. Assim, o caminho percorrido desde a *end-user area* origem 10.0.3.X até à *end-user area* destino 10.0.4.X terá sido o seguinte, visto na Figura 4.9:

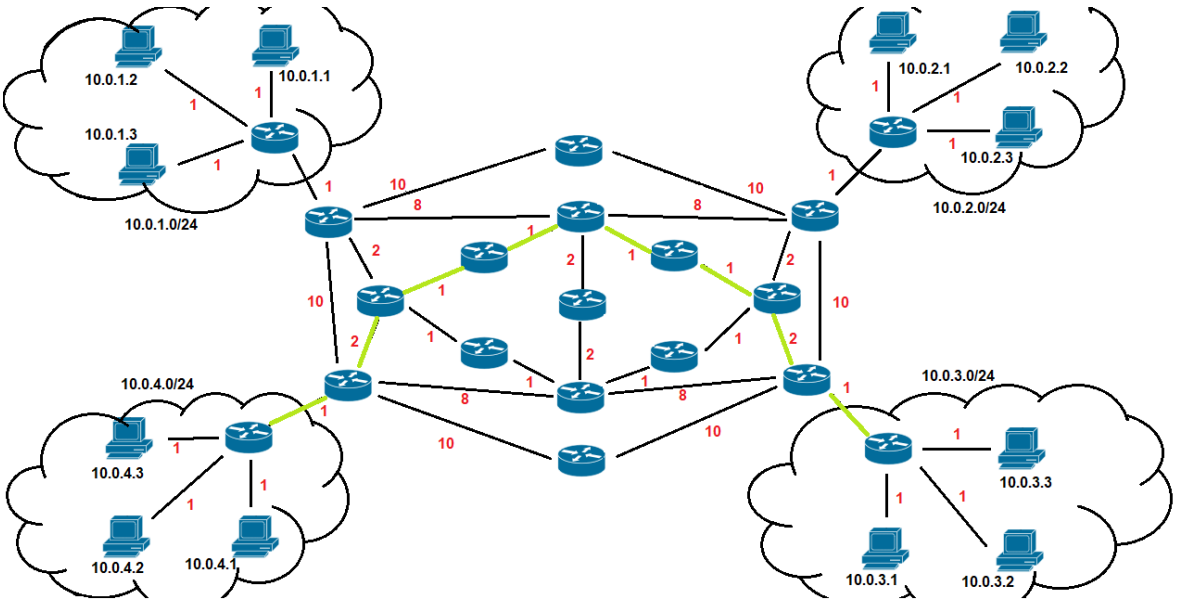


Figura 4.9.: Teste de `ping` entre as *end-user areas* 10.0.3.X e 10.0.4.X.

Para se verificar que este caminho mais curto terá sido o correto, procedeu-se a um novo *traceroute* correspondente, na Figura 4.10:

```
mininet> h31 traceroute h41
traceroute to 10.0.4.1 (10.0.4.1), 30 hops max, 60 byte packets
 1  10.0.3.1 (10.0.3.1)  82.403 ms  91.534 ms  92.958 ms
 2  10.0.3.4 (10.0.3.4)  0.027 ms  0.014 ms  0.013 ms
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  10.0.4.4 (10.0.4.4)  0.025 ms  0.014 ms  0.014 ms
11  10.0.4.1 (10.0.4.1)  103.706 ms  94.565 ms  81.668 ms
```

Figura 4.10.: Teste de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.4.X.

Não fugindo à exceção, este *traceroute* também apresenta os mesmos problemas que os dois *traceroutes* anteriores, demonstrando os *switches* que não pertencem a *end-user areas* com asteriscos. Assim, remete-se uma vez mais para o controlador, sendo que se verificou a seguinte rota, dada pela Figura 4.11:

```
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:03
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:13
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:11
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:0f
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:0b
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:08
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:07
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:06
              INFO  [n.f.m.MACTracker] Source: 10.0.3.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:04
```

Figura 4.11.: Teste de `traceroute` entre um *host* da *end-user area* 10.0.3.X e outro da *end-user area* 10.0.4.X.

Como tal, e estando verificado o correto funcionamento do algoritmo de Dijkstra aplicado à topologia, através das Figuras 4.5, 4.8 e 4.11, prossegue-se para os testes das restantes funcionalidades.

É de se notar que, de forma a evitar que sejam colocados capturas de imagem das demais rotas, *traceroutes* e provas de *traceroute*, e como este mecanismo se verifica como válido e será a base dos demais mecanismos das secções seguintes, optar-se-á por apenas se colocar uma imagem da topologia em questão e uma imagem do respetivo *traceroute*. Estas capturas de imagem serão apenas adicionadas em casos que requeiram a verificação de alteração de rota ou quando tenha havido mudança da rota óptima calculada pelo algoritmo de Dijkstra.

4.3.2 *Falhas de Links*

De forma a verificar que a topologia identificava de forma adequada as falhas de links existentes na própria, foi executado um comando no terminal onde a topologia `Mininet` teria sido inicialmente executada, de forma a desligar um dos links, como na Figura 4.12.

```
tul@tul-VirtualBox:~/Desktop/topos$ sudo mn --controller=remote,ip=127.0.0.1,por
t=6653 --switch ovsk --custom bigtopo.py --topo mytopo
*** Creating network
*** Adding controller
*** Adding hosts:
h11 h12 h13 h21 h22 h23 h31 h32 h33 h41 h42 h43
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19
*** Adding links:
(h11, s1) (h12, s1) (h13, s1) (h21, s2) (h22, s2) (h23, s2) (h31, s3) (h32, s3)
(h33, s3) (h41, s4) (h42, s4) (h43, s4) (s1, s5) (s2, s18) (s3, s19) (s4, s6) (s
5, s6) (s5, s7) (s5, s10) (s5, s11) (s6, s7) (s6, s13) (s6, s14) (s7, s8) (s7, s
9) (s8, s11) (s9, s13) (s10, s18) (s11, s12) (s11, s15) (s11, s18) (s12, s13) (s
13, s16) (s13, s19) (s14, s19) (s15, s17) (s16, s17) (s17, s18) (s17, s19) (s18,
 s19)
*** Configuring hosts
h11 h12 h13 h21 h22 h23 h31 h32 h33 h41 h42 h43
*** Starting controller
c0
*** Starting 19 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 ...
*** Starting CLI:
```

Figura 4.12.: Execução da topologia no utilitário `Mininet`.

Através deste mesmo terminal, e explicitado na Figura 4.13, escolhendo arbitrariamente um link entre dois *switches* quaisquer, fez-se uso do `link A B down`, em que A e B são os identificadores dos mesmos *switches*.

```
mininet> link s5 s7 down
```

Figura 4.13.: Encerramento de um dos *links* entre os *switches* s5 e s7.

Deste modo, e após encerramento imediato do mesmo link, verificou-se que, no `Eclipse`, a ferramenta utilizada para desenvolvimento e execução da solução, e na Figura 4.14, teria aparecido uma mesma notificação de que o link teria falhado.

```
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:07 outPort=1, dst=00:00:00:00:00:00:00:05
, inPort=4, latency=11]
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:05 outPort=4, dst=00:00:00:00:00:00:00:07
, inPort=1, latency=17]
```

Figura 4.14.: Notificação de falha do *link* respetivo.

Seguindo-se um novo teste, agora com um outro link qualquer diferente do inicial, verificaram-se as Figuras 4.15 e 4.16:

```
mininet> link s5 s7 down
mininet> link s11 s18 down
```

Figura 4.15.: Encerramento de um segundo *link* entre os *switches* s11 e s18.

```
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:12 outPort=3, dst=00:00:00:00:00:00:00:0b
, inPort=3, latency=14]
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:0b outPort=3, dst=00:00:00:00:00:00:00:12
, inPort=3, latency=20]
```

Figura 4.16.: Notificação de falha de outro *link* associado à topologia.

Comprovando-se, desta forma, que o controlador receberia notificações de falha dos *links* da topologia, estando, assim, e em conjunção com os resultados da subsecção anterior, a solução de encaminhamento completa no que toca às suas funções básicas, pronta a abarcar novas funcionalidades, mais complexas.

4.3.3 *Convergência Imediata*

Para que fosse possível a verificação do módulo de convergência imediata para falhas de *links*, juntou-se a execução do teste do algoritmo de Dijkstra com o teste de falhas de links, tendo sido feito um `traceroute` antes e após de desligar um *link*. Os resultados verificados podem ser constatados pelo seguinte, iniciando-se com o estado da rede aquando do momento da execução do primeiro `traceroute` que, por impossibilidade de verificação no Mininet se constata na seguinte topologia, com o caminho a verde, dado pela Figura 4.17:

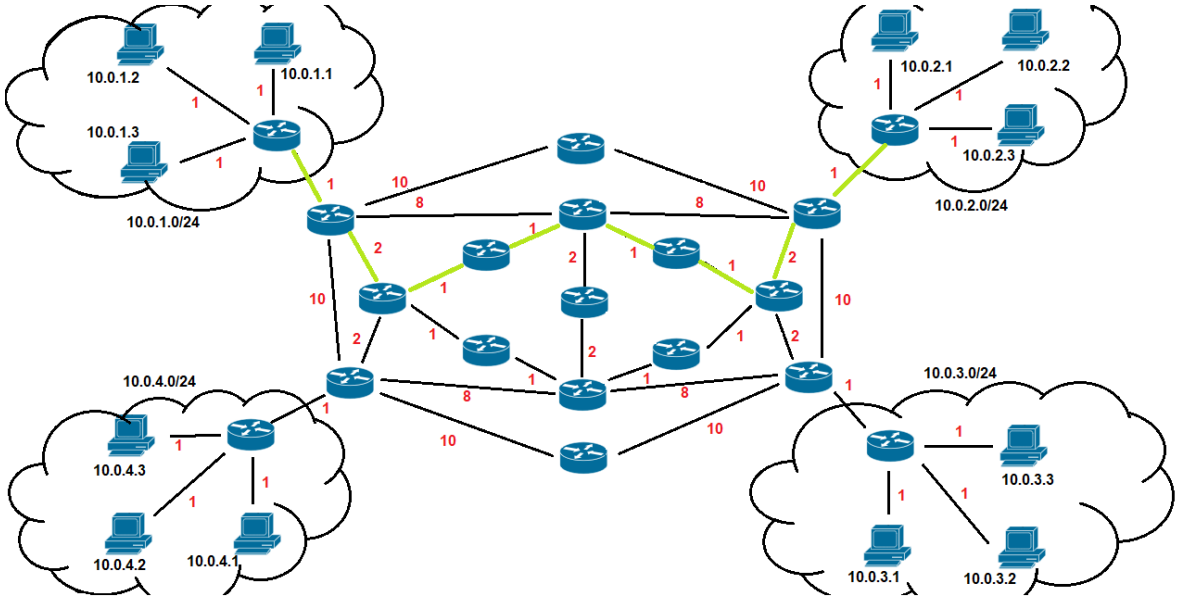


Figura 4.17.: Rota seguida entre *hosts* das *end-user areas* 10.0.1.X e 10.0.2.X. antes da remoção do *link*.

Depois desligou-se um dos *links* entre os *switches* s5 e s7, através do comando representado na Figura 4.18:

```
mininet> link s5 s7 down
```

Figura 4.18.: Remoção de um *link* entre os *switches* s5 e s7.

O link desligado pela execução do comando da figura anterior corresponde, na topologia, ao *link* assinalado na Figura 4.19, com uma cruz vermelha:

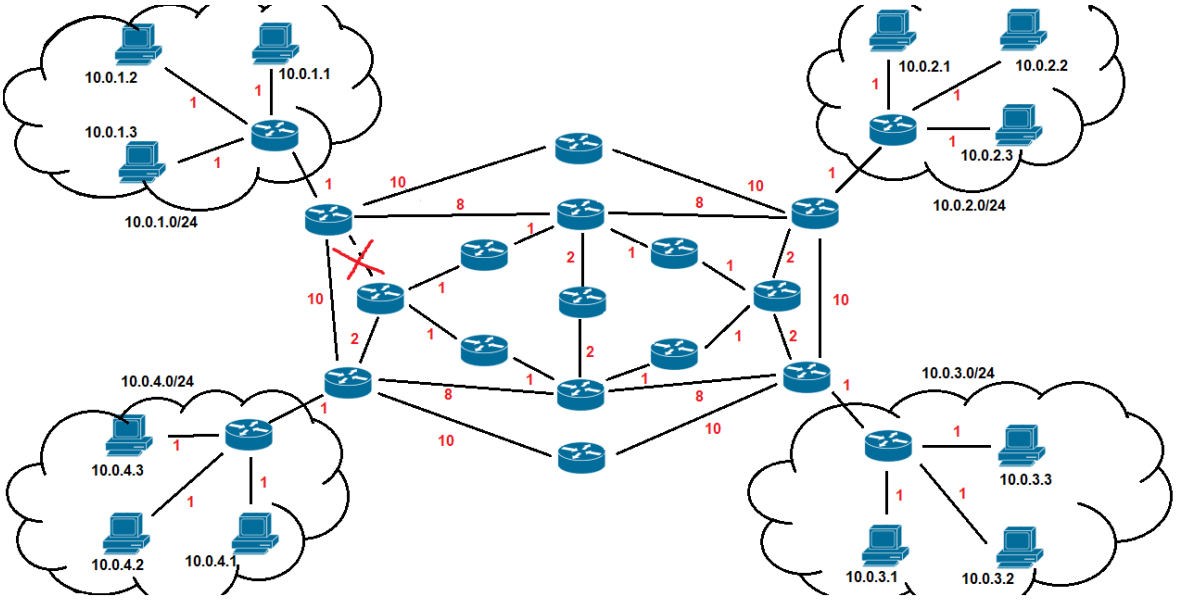


Figura 4.19.: Remoção de *link* na topologia.

Verificou-se, assim, que o controlador detetou o seu encerramento e procedeu à reposição da topologia. A Figura 4.20 apresenta a informação dada pelo controlador relativa à notificação de falha do mesmo *link*, após ter sido removido pelo comando anterior:

```
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:07 outPort=1, dst=00:00:00:00:00:00:00:05
, inPort=4, latency=11]
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:05 outPort=4, dst=00:00:00:00:00:00:00:07
, inPort=1, latency=17]
                    INFO  [n.f.t.TopologyManager] Reestablishing topology du
e to: link-failure
```

Figura 4.20.: Notificação de falha do *link* removido e reposição da topologia.

Após remoção deste mesmo *link*, procedeu-se à execução de um novo *traceroute*, visto que houve alteração do caminho mais curto. O resultado deste *traceroute* apresenta o novo caminho mais curto, especificado a verde na Figura 4.21:

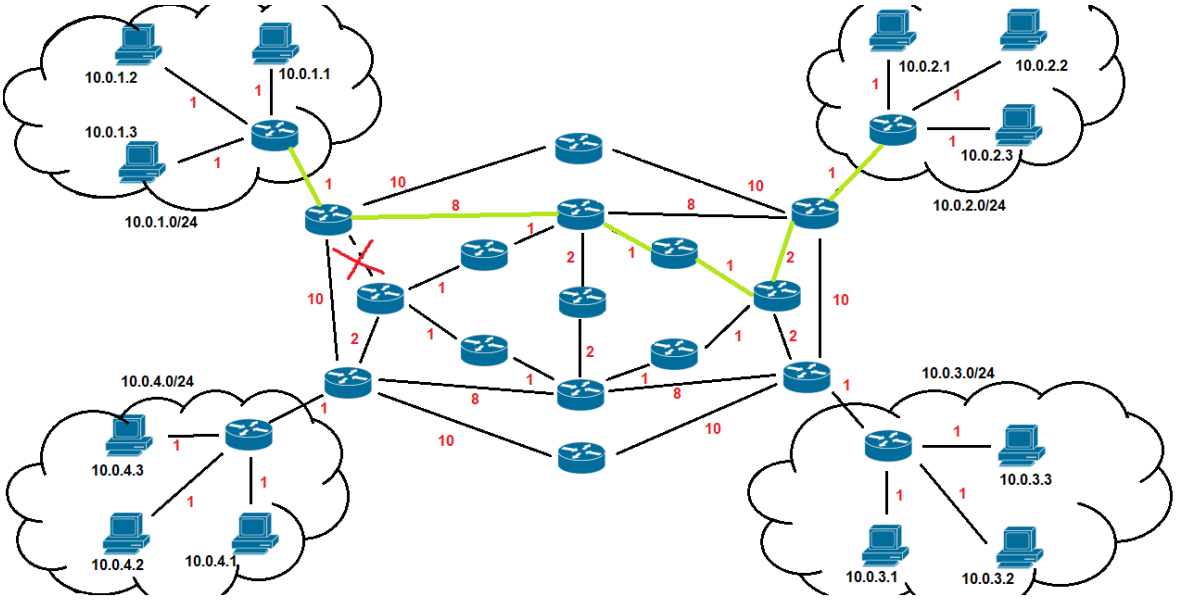


Figura 4.21.: Rota seguida entre *hosts* das *end-user areas* 10.0.1.X e 10.0.2.X após remoção do *link*.

Desta forma verificou-se que a solução respondeu, de modo eficaz, à falha do mesmo link. Repetindo os testes para um outro *link*, após re-execução da topologia e da solução, iniciou-se o novo teste com um novo *traceroute*, antes da falha do *link*. Este novo *traceroute* é enunciado pela rota percorrida na Figura 4.22, também a verde:

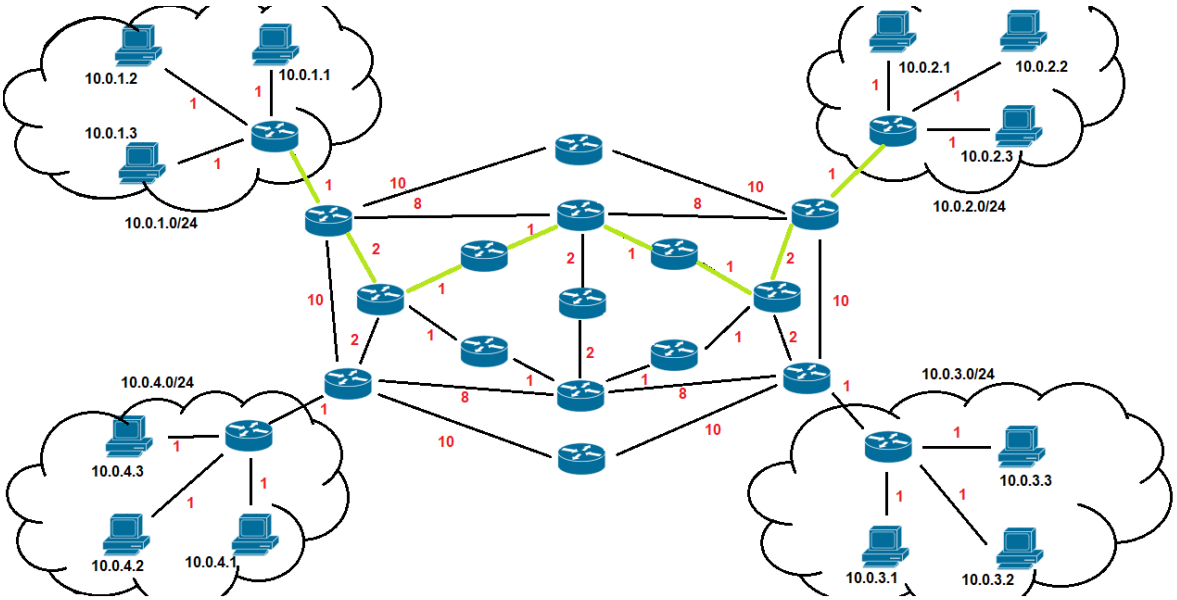


Figura 4.22.: Rota seguida entre *hosts* das *end-user areas* 10.0.1.X e 10.0.2.X antes da remoção do segundo *link*.

Efetuaram-se novos testes sobre um outro *link*, diferente do anterior, de modo a verificar se o controlador responderia ao mesmo. Neste caso foi escolhido o *link* entre os *switches* s11 e s18, através do seguinte comando, dado pela Figura 4.23:

```
mininet> link s11 s18 down
```

Figura 4.23.: Remoção de um outro *link* entre os *switches* s11 e s18.

Na mesma topologia, o *link* entre os *switches* s11 e s18 corresponde ao seguinte *link*, enunciado por uma cruz vermelha na Figura 4.24:

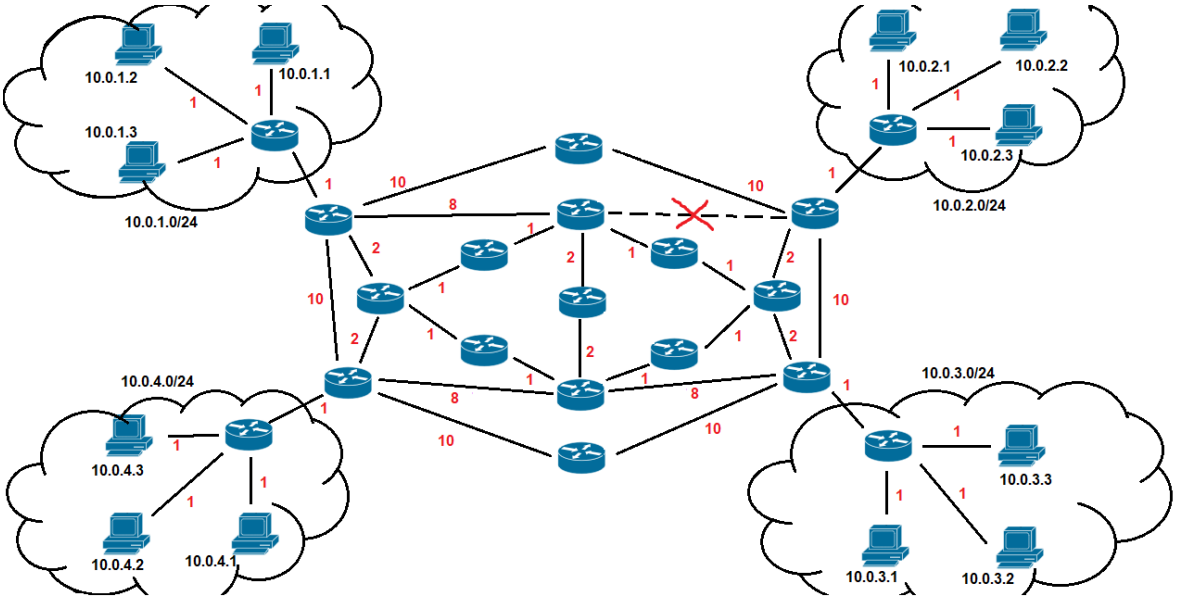


Figura 4.24.: Remoção de um outro *link* na topologia.

Como tal, e tendo sido este *link* desligado da topologia através do Mininet, o controlador respondeu de forma imediata, notificando o administrador da rede da falha do mesmo. A resposta gerada no controlador é explicitada pela Figura 4.25:

```
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:12 outPort=3, dst=00:00:00:00:00:00:00:0b
, inPort=3, latency=14]
                    INFO  [n.f.l.i.LinkDiscoveryManager] Inter-switch link r
emoved: Link [src=00:00:00:00:00:00:00:0b outPort=3, dst=00:00:00:00:00:00:00:12
, inPort=3, latency=20]
                    INFO  [n.f.t.TopologyManager] Reestablishing topology du
e to: link-failure
```

Figura 4.25.: Notificação de falha de um outro *link* removido e reposição da topologia.

Feita uma nova execução de um *traceroute* entre as *end-user areas* inicialmente dadas, de forma a que se observasse o novo caminho mais curto, foi possível a verificação da seguinte rota percorrida, dada na Figura 4.26 por um caminho verde:

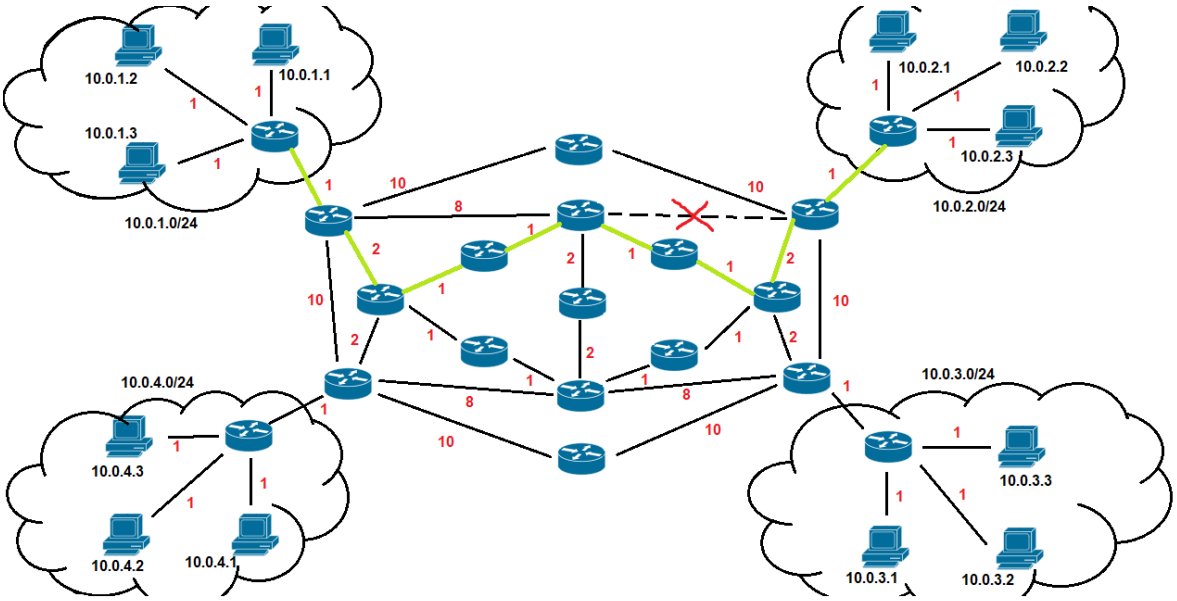


Figura 4.26.: Rota seguida entre *hosts* das *end-user areas* 10.0.1.X e 10.0.2.X após remoção do segundo *link*.

Como, para o caso deste *link* se constatou que a rota não foi alterada (visto que não interfere com a rota prévia), conclui-se que a solução converge de modo imediato para as falhas de quaisquer links da topologia, adaptando as suas rotas consoante afectem os caminhos mais curtos previamente calculados.

4.3.4 *Proteção de Tráfego*

De modo a simular um módulo capaz de proteger certos *links* da topologia do tráfego gerado pelas *end-user areas*, foi necessária a criação de um *array* que contivesse os *links* e os períodos de tempo durante os quais os mesmos devem ficar protegidos do tráfego gerado.

Inicialmente iniciaram-se os testes com o envio de um pacote entre as *end-user areas* 1 e 2, como visto na Figura 4.27 e as *end-user areas* 1 e 4, como visto na Figura 4.28, sendo que os caminhos de interligação entre estes pares de áreas partilham alguns *links*, como visível após execução do Dijkstra.

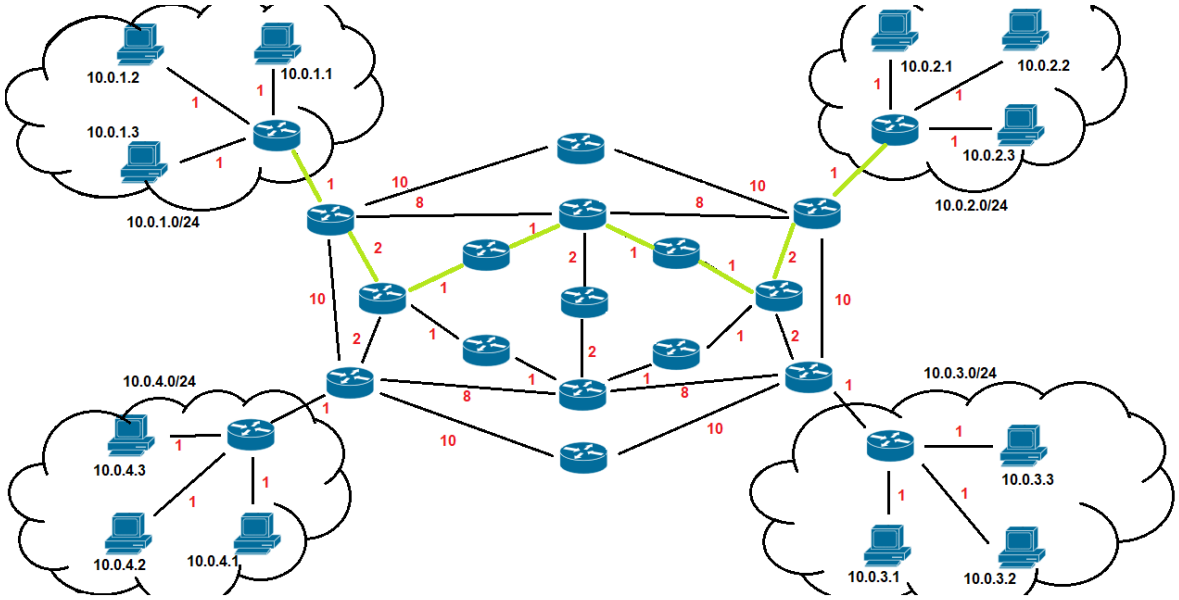


Figura 4.27.: Caminho percorrido por um pacote entre as *end-user areas* 10.0.1.X e 10.0.2.X.

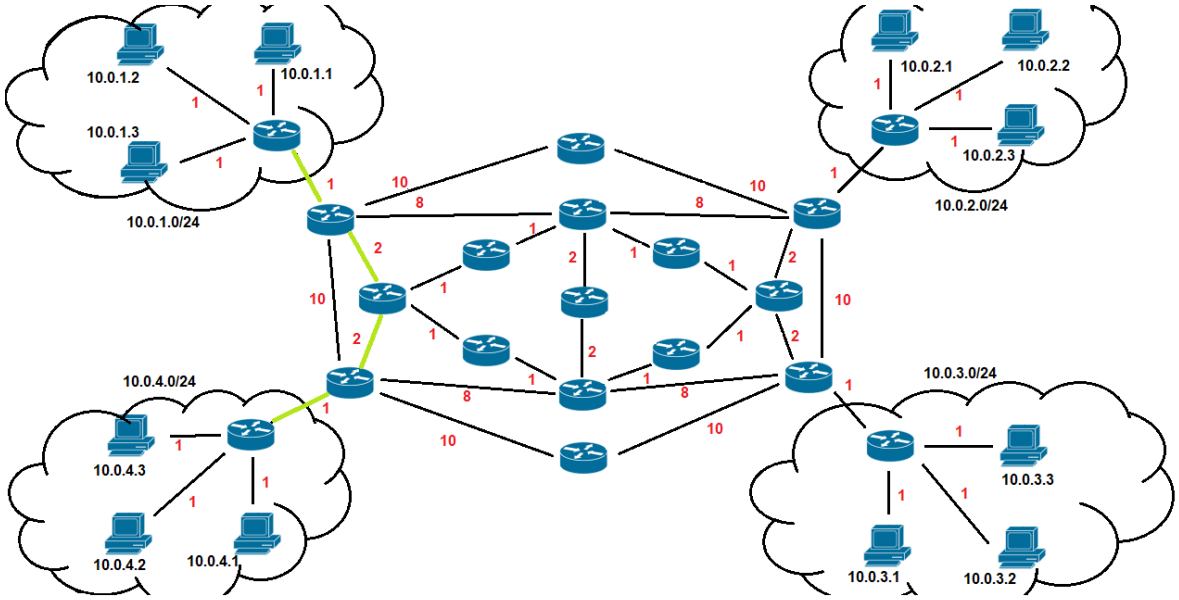


Figura 4.28.: Caminho percorrido por um pacote entre as *end-user areas* 10.0.1.X e 10.0.4.X.

Os *links* dados pelo seguinte *array* correspondem ao *input* fornecido pelo administrador de rede, que representam os *links* que o controlador deverá proteger. Para exemplificar uma qualquer proteção, foram utilizados os seguintes *links*:

```
protected_links =
   {["00:00:00:00:00:00:00:05","00:00:00:00:00:00:00:07"],
```

["00:00:00:00:00:00:00:0b","00:00:00:00:00:00:00:12"]}

Estes *links*, na topologia, correspondem aos seguintes *links*, representados por uma cruz vermelha na Figura 4.29:

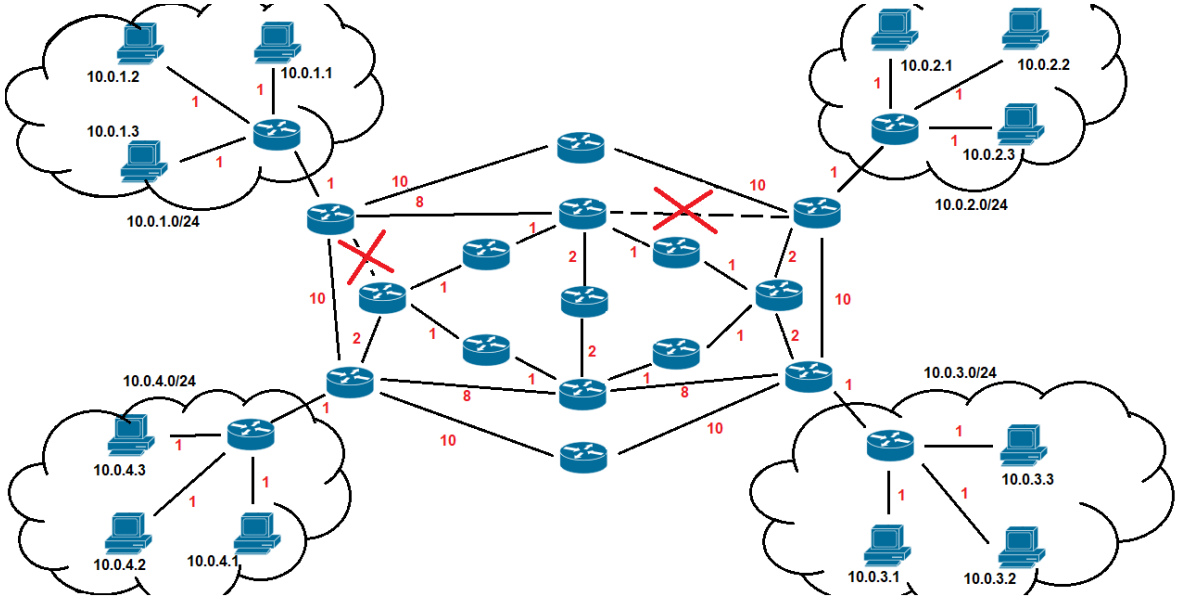


Figura 4.29.: *Links* a serem protegidos pelo mecanismo proposto.

Após injeção dos *links* a serem protegidos pelo controlador, foi feito um novo envio de um pacote teste, capaz de percorrer a rede e seguir as regras de encaminhamento injetadas. Este seguiu com os mesmos remetentes e destinatários anteriormente enunciados, isto é, desde a *end-user area* 1 para a *end-user area* 2, tal como descrito na Figura 4.30, e desde a *end-user area* 1 para a *end-user area* 4, tal como descrito na Figura 4.31:

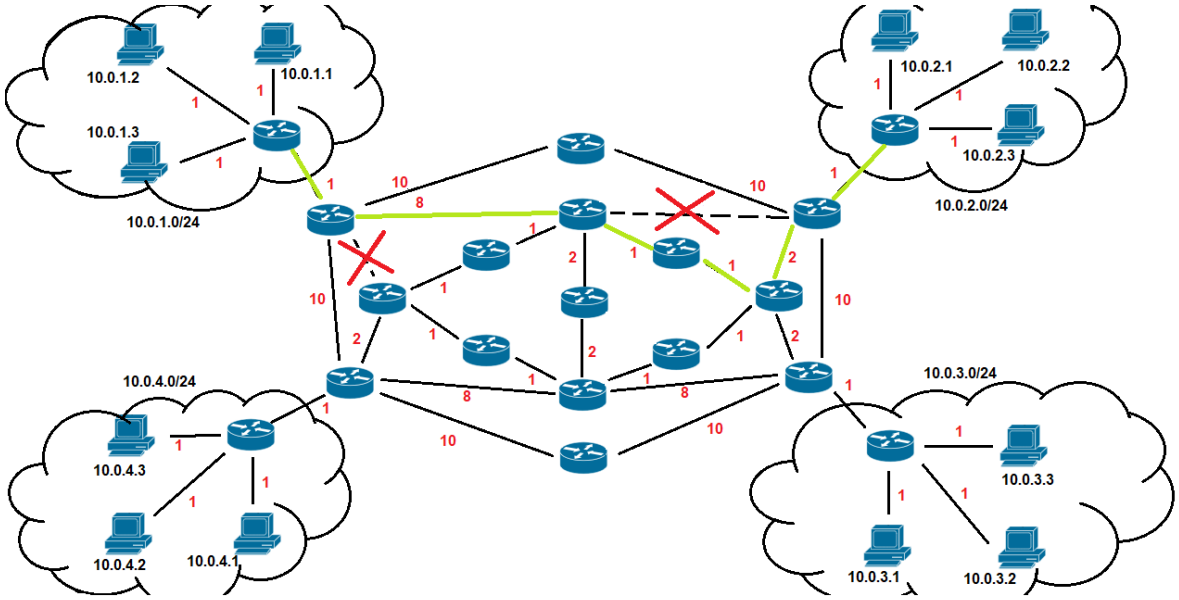


Figura 4.30.: Envio de pacote entre as *end-user areas* 10.0.1.X e 10.0.2.X, após proteção do *link*.

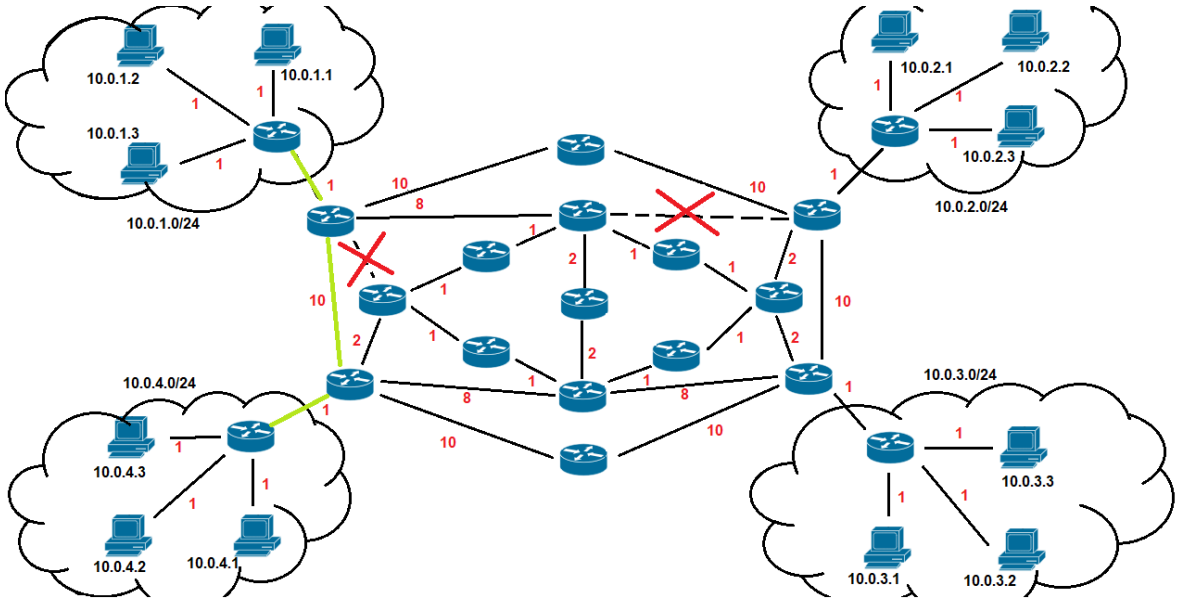


Figura 4.31.: Envio de pacote entre as *end-user areas* 10.0.1.X e 10.0.4.X, após proteção do *link*.

Como tal, é possível a verificação de que as rotas iniciais, calculadas pelo algoritmo de Dijkstra, foram efetivamente alteradas para novos caminhos mais curtos que não integrem os *links* protegidos, constatando-se que o tráfego é, então, desviado com sucesso, e os *links* protegidos de qualquer tráfego não-diferenciado.

4.3.5 *Estatísticas de Tráfego*

O módulo de Estatísticas de Tráfego apresenta a sua utilidade para extração de informação necessária para o mecanismo de reatividade a níveis de congestão, apresentado na secção seguinte, visto que é este mesmo que permite aceder a certos dados sobre qualquer *switch* existente na topologia e ligado ao controlador.

Para ativação deste mesmo módulo bastou a utilização de um módulo pré-desenvolvido pelos desenvolvedores do controlador `Floodlight`. Este mesmo possibilita a sua ativação através da edição do ficheiro de configuração com o nome `floodlightdefault.properties`. Através da alínea `net.floodlightcontroller.statistics.enable=X`, em que X é um booleano.

Substituíndo, deste modo, o mesmo valor X por `TRUE`, adiciona-se um *overhead* que é tanto maior consoante a frequência de recolha de estatísticas, isto é, inversamente proporcional com o número de segundos no qual é feita a recolha das mesmas.

Estando possível, assim, a recolha de estatísticas, e sendo que este mesmo módulo abre uma ligação num endereço local para consulta em *browser*, procedeu-se à mesma consulta, sendo a Figura 4.32 obtida:

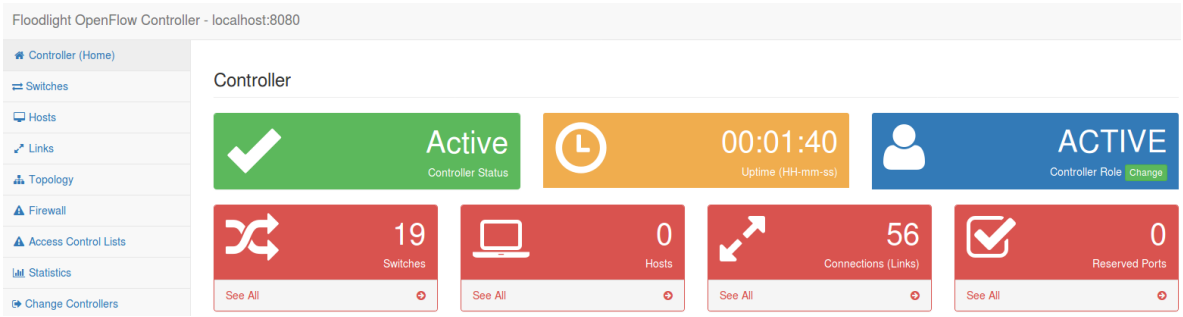


Figura 4.32.: *Homepage* de página de recolha de estatísticas.

Estas estatísticas permitem a monitorização de tráfego dos demais interfaces assim como a constatação de várias métricas presentes na topologia.

4.3.6 *Reatividade a Congestão*

Em relação a este módulo, e sendo que o teste de congestão de quaisquer *links* implica a existência de tráfego em média-grande escala em qualquer um dos mesmos *links*, foi necessária a geração de um alto volume de tráfego constante, para simular, assim, um cenário de congestão de um *link*. Assim sendo, foi escolhido o seguinte *link* da Figura 4.33, representado a vermelho com uma linha tracejada:

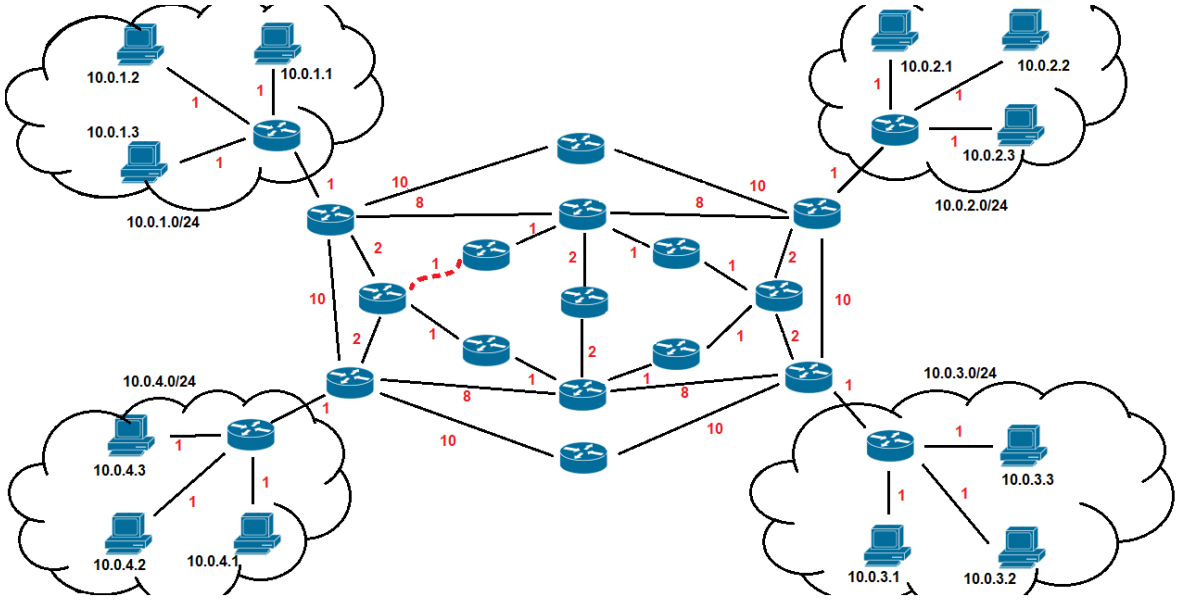


Figura 4.33.: Topologia com *link* congestionado selecionado.

De seguida, foi executada a topologia e a solução de encaminhamento, ainda sem congestionamento através da geração de tráfego no *link* em específico, enviando-se pacotes entre *hosts* das *end-user areas* 10.0.2.X e 10.0.4.X, tal como descrito nas Figuras 4.34 e 4.35:

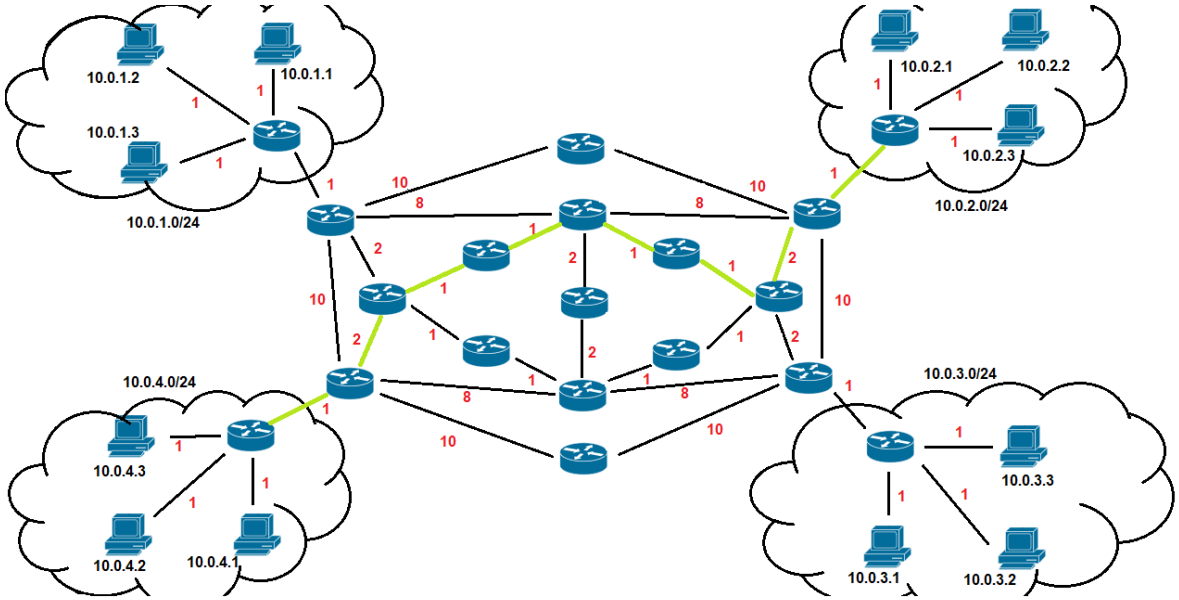


Figura 4.34.: Envio de pacote entre as *end-user areas* 10.0.2.X e 10.0.4.X, antes da congestão do *link* selecionado.

```
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:06
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:07
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:08
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:0b
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:0f
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:11
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:12
             INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:02
```

Figura 4.35.: Rota seguida entre *hosts* das *end-user areas* 10.0.2.X e 10.0.4.X, antes de desvio de tráfego por congestão.

Estando a topologia estável e pronta a ser testada, foi gerado, assim, tráfego entre os dois *switches* selecionados, pelo *link* dos mesmos *switches*, representado nas Figuras 4.36 e 4.37:

```
mininet> s7 ping -s 50000 s8
PING 127.0.0.1 (127.0.0.1) 50000(50028) bytes of data.
50008 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.039 ms
50008 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.089 ms
50008 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.037 ms
50008 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.062 ms
50008 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.101 ms
50008 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.100 ms
50008 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.762 ms
50008 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=2.101 ms
50008 bytes from 127.0.0.1: icmp_seq=9 ttl=64 time=30.100 ms
50008 bytes from 127.0.0.1: icmp_seq=10 ttl=64 time=100.037 ms
```

Figura 4.36.: Congestão do *link* selecionado anteriormente.

```
                    WARNING  [n.f.t.TopologyManager] Link between switch: 00:00:00:00:00:00:00:0
7 and switch: 00:00:00:00:00:00:00:08 under high load (above specified value: 50%)
```

Figura 4.37.: Aviso de congestão nos *links* selecionados.

O mecanismo desenvolvido de reatividade a congestão, imediatamente após constatar que o *link* se encontra congestionado, altera as tabelas de encaminhamento e desvia o tráfego consoante hajam alternativas melhores à atual, visto pela Figura 4.38:

```
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:06
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:07
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:09
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:0d
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:10
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:11
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:12
              INFO  [n.f.m.MACTracker] Source: 10.0.2.1  Destination: 10.0.4.1 s
een on switch: 00:00:00:00:00:00:00:02
```

Figura 4.38.: Rota seguida entre *hosts* das *end-user areas* 10.0.2.X e 10.0.4.X, após desvio de tráfego por congestão.

Este caminho demonstrado pelo controlador é dado pela Figura 4.39 a verde, que representa a mesma rota percorrida pela Figura 4.38:

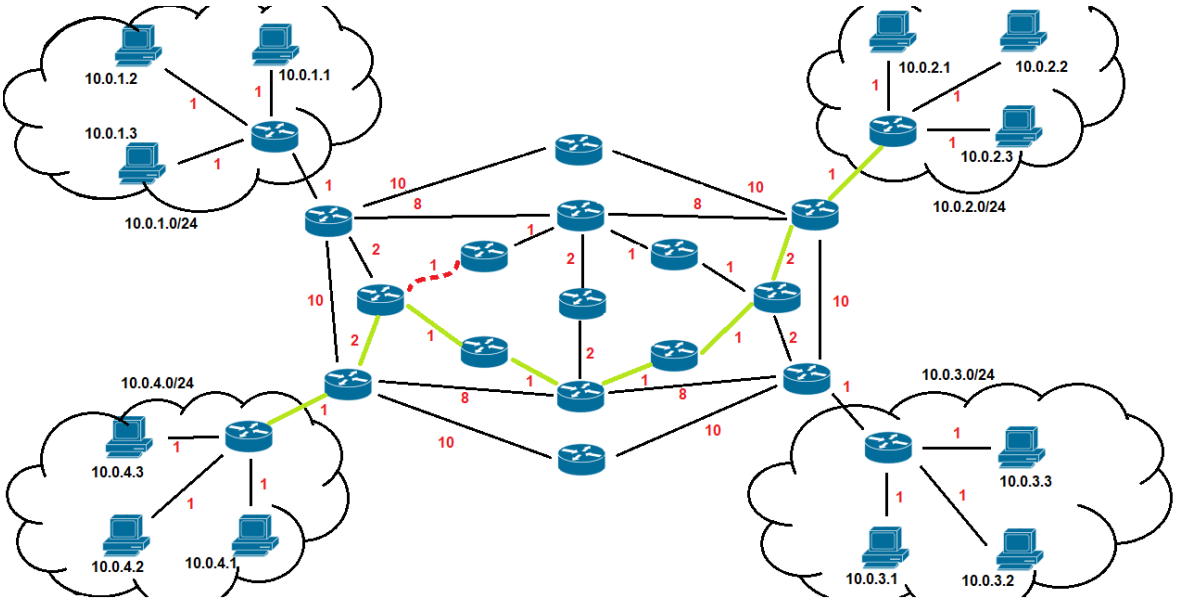


Figura 4.39.: Rota seguida após congestão do *link* selecionado.

Como se vê na Figura 4.39, foi calculado um novo caminho mais curto pelo algoritmo de Dijkstra, mas sem passar no *link* congestionado. Deste modo, o mecanismo desenvolvido de reatividade a congestão consegue, efetivamente, reagir de forma eficaz, evitando a utilização do *link* congestionado.

4.3.7 *Rotas Individuais para Fluxos*

No que toca a este módulo de especificação de rotas individuais para fluxos, é necessária a injeção de rotas individuais por parte do administrador da rede. Como tal, é feita uma adição inicial das mesmas rotas a ser percorridas na topologia, que pode ser dada pelo seguinte:

```
individual_routes =
   {["10.0.4.1","10.0.3.1",["00:00:00:00:00:00:00:06",
   "00:00:00:00:00:00:00:0d","00:00:00:00:00:00:00:13"]]}
```

Este módulo aceita a definição de rotas para fluxos individuais, como verificável pelo *array* anterior, que representa uma rota entre os *hosts* 10.0.4.1 e 10.0.3.1, que deverá passar pelos *switches* especificados. Deste modo, e como é visível na Figura 4.40, o tráfego gerado entre estes mesmos *hosts* seguirá a rota a verde na topologia:

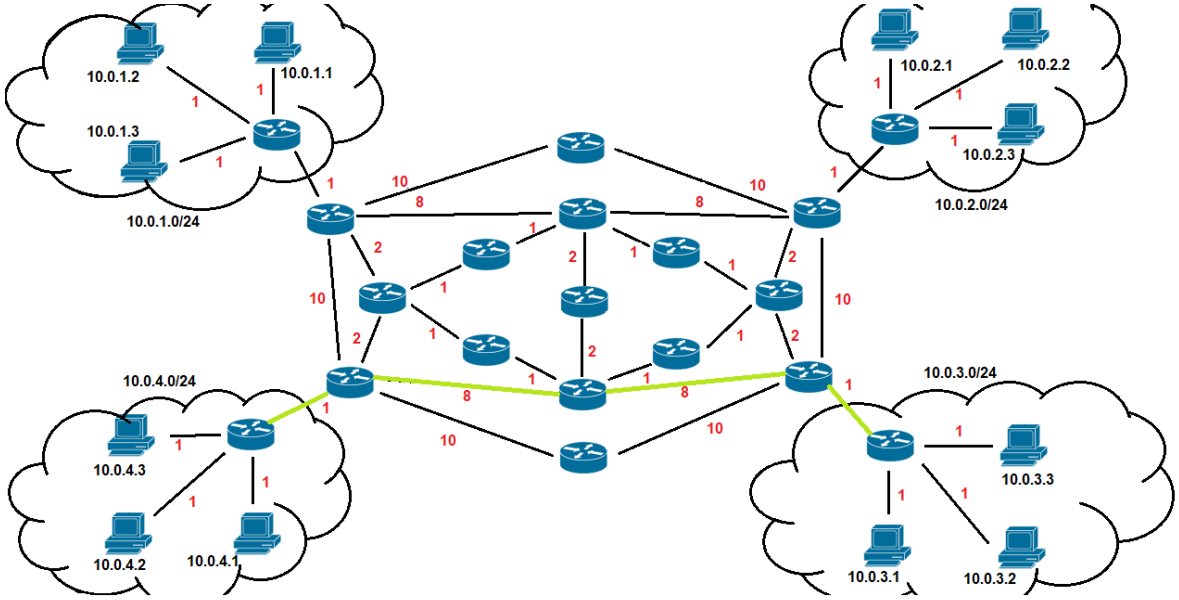


Figura 4.40.: Rota seguida entre *hosts* com os endereços 10.0.4.1 e 10.0.3.1.

De seguida, testou-se se, de facto, pela Figura 4.41, tal rota estaria a ser seguida, sendo esta verificada pela informação dada no controlador, que remete para o seguinte:

```
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.1 s
een on switch: 00:00:00:00:00:00:00:06
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.1 s
een on switch: 00:00:00:00:00:00:00:0d
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.1 s
een on switch: 00:00:00:00:00:00:00:13
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.1 s
een on switch: 00:00:00:00:00:00:00:03
```

Figura 4.41.: Notificação do controlador sobre a rota seguida entre *hosts* com os endereços 10.0.4.1 e 10.0.3.1.

Porém, e estando a figura anterior a representar que a rota terá sido corretamente injetada e percorrida, resta verificar se a mesma não é seguida para um outro *host* origem ou *host* destino, isto é, quando os endereços não fazem *match* com os inicialmente injetados pela rota fornecida. Para se constatar que, por exemplo, mantendo-se o *host* origem e alterando-se o *host* destino, a rota seguida é diferente da rota injetada, procedeu-se a um novo teste, cuja rota terá sido dada pela Figura 4.42:

```
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:04
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:06
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:07
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:08
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:0b
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:0f
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:11
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:13
              INFO  [n.f.m.MACTracker] Source: 10.0.4.1  Destination: 10.0.3.2 s
een on switch: 00:00:00:00:00:00:00:03
```

Figura 4.42.: Rota seguida entre *hosts* com os endereços 10.0.4.1 e 10.0.3.2.

Assim, e tendo sido a rota percorrida completamente diferente da rota injetada, como é visível pela diferença nos *switches* percorridos, é possível verificar-se que, nos casos em que não há *match* entre *host* origem e *host* destino, é seguida a rota *default* do caminho mais curto, calculada pelo algoritmo de Dijkstra. Deste modo, afirma-se, então, o correto funcionamento do mesmo mecanismo desenvolvido.

4.3.8 *Partição de Topologia*

Para este módulo de partição da topologia em diferentes sub-redes, foi necessária a escolha, por parte do administrador de rede, de múltiplos *switches*, de forma a criar uma ou várias sub-redes. Deste modo, e para efeitos de testes, é feita uma adição inicial de duas sub-redes. Esta é feita através dos *switches* e respetivas *end-user areas* que pertençam a cada partição. Assim, o seguinte é adicionado:

```
slice1 = {["10.0.1.0", "10.0.2.0", ["00:00:00:00:00:00:00:01",
"00:00:00:00:00:00:00:05", "00:00:00:00:00:00:00:07",
"00:00:00:00:00:00:00:08", "00:00:00:00:00:00:00:0b",
"00:00:00:00:00:00:00:12", "00:00:00:00:00:00:00:02"]]};
slice2 = {["10.0.4.0", "10.0.3.0", ["00:00:00:00:00:00:00:04",
"00:00:00:00:00:00:00:06", "00:00:00:00:00:00:00:07",
"00:00:00:00:00:00:00:09", "00:00:00:00:00:00:00:0d",
"00:00:00:00:00:00:00:13", "00:00:00:00:00:00:00:03"]]}
```

Estas duas partições correspondem ao seguinte na Figura 4.43, sendo que a primeira partição, que contém as *end-user areas* 10.0.1.0/24 e 10.0.2.0/24, está representada pelos *switches* e *links*, rodeados a vermelho, e a segunda partição, que contém as *end-user areas* 10.0.3.0/24 e 10.0.4.0/24 está representada pelos *switches* e *links*, rodeados a azul:

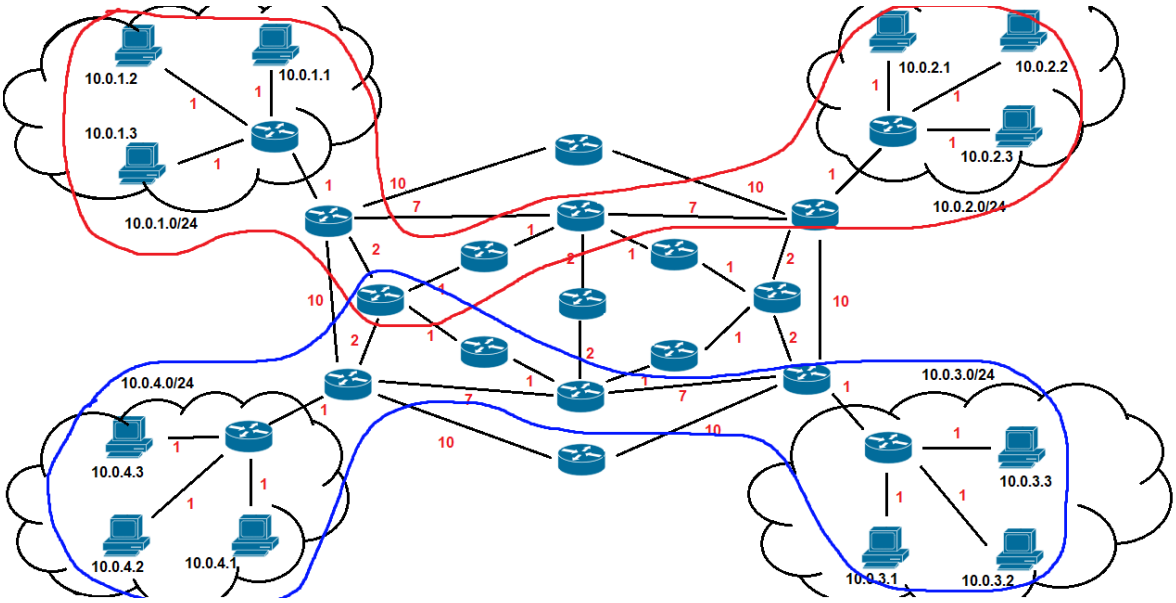


Figura 4.43.: Partição da topologia em sub-redes, dada por *input* ao administrador de rede.

As duas partições geradas pelo mecanismo e dadas por *input* ao administrador de rede podem ser vistas pelas Figuras 4.44 e 4.45, sendo que a primeira representa a sub-rede que teria um círculo vermelho na Figura 4.43, e a segunda a que teria um círculo azul:

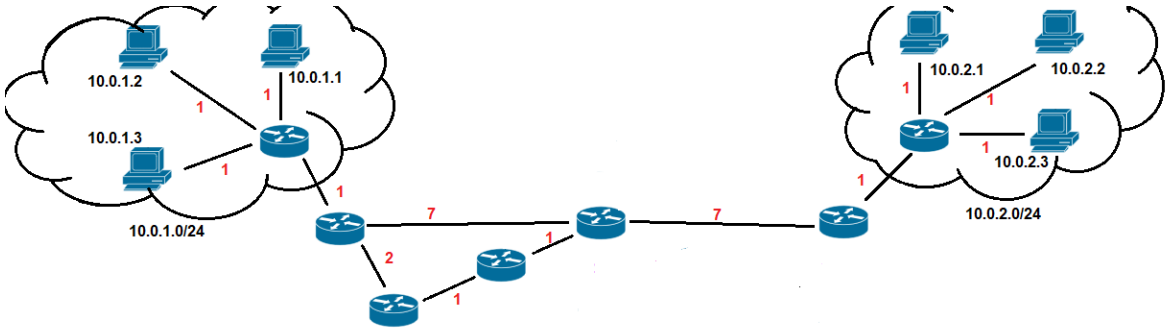


Figura 4.44.: Primeira sub-rede gerada pela partição da topologia.

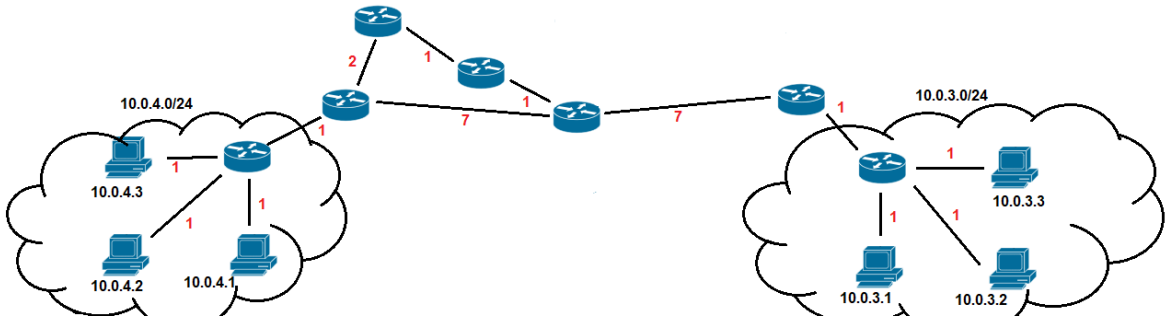


Figura 4.45.: Segunda sub-rede gerada pela partição da topologia.

De seguida, executa-se o Algoritmo 1 sob cada uma destas sub-redes, de forma a que se obtenham os caminhos mais curtos respetivos a cada uma das mesmas e estes sejam respetivamente injetados nos *switches* da topologia. As rotas injetadas nas tabelas de encaminhamento dos vários *switches* serão consoante as *end-user areas* existentes no seu domínio. Deste modo, os *hosts* de qualquer *end-user area* de uma partição A não terão a possibilidade de contactar os *hosts* de uma outra *end-user area* de qualquer outra partição, diferente da partição A.

Estando as rotas injetadas, procede-se a um teste de conectividade, bastando a execução de um *pingall* para testar se cada um dos *hosts* é alcançável através de *pings*. Como tal, e seguindo a legendagem atribuída na Figura 4.1, um *host* representado por **h12** terá o endereço de IP dado por 10.0.1.2 (ou seja, `hXY = 10.0.X.Y`). Como tal, verifica-se pela Figura 4.46 que as sub-áreas estão divididas de forma correta:

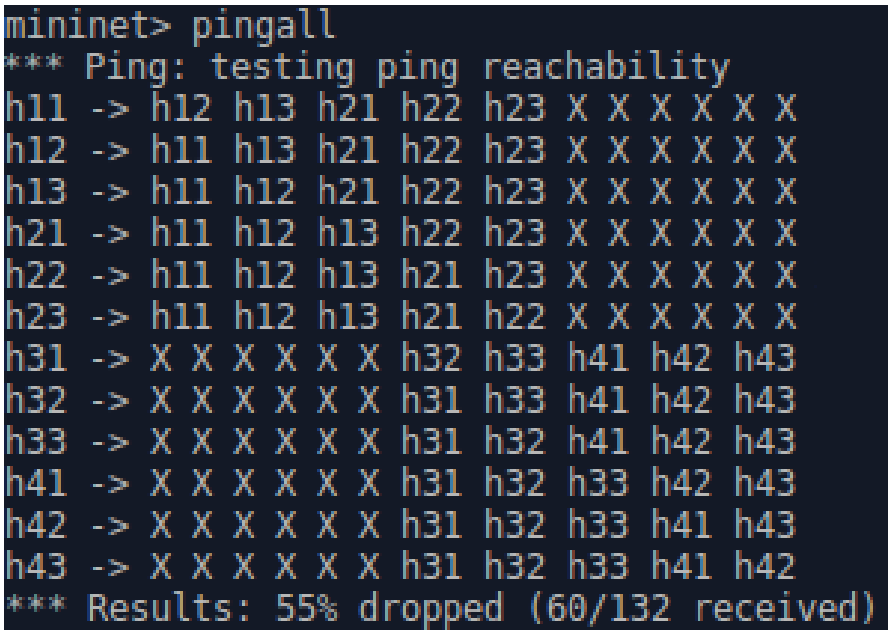


Figura 4.46.: Teste de conectividade entre *hosts* de múltiplas sub-redes.

Assim, e como se constata na 4.46, *hosts* de diferentes sub-redes não se conseguem alcançar e efetuar qualquer conexão, é então possível afirmar que a divisão da topologia em diferentes sub-áreas é feita de forma correta.

4.4 SUMÁRIO

Este capítulo mencionou os testes e respetivos resultados obtidos nos mesmos, apresentando cenários de teste que tentassem abarcar múltiplos casos, mesmo que virtualizados e emulados. Foi apresentada a bateria de testes utilizada, assim como os passos intercalares entre cada uma das etapas. Os resultados obtidos provaram o correto funcionamento dos demais módulos implementados em conformidade com as funcionalidades que o controlador escolhido oferece.

5

CONCLUSÃO

Neste capítulo final serão tecidas as demais considerações sobre o trabalho feito sob forma de síntese, referindo os resultados obtidos ao longo dos múltiplos objetivos delineados. Finalmente, mencionar-se-ão algumas reflexões sobre aspetos relevantes para trabalho futuro.

5.1 RESUMO

De forma a construir todo o compêndio necessário à realização deste documento, assim como a implementação dos demais mecanismos de encaminhamento de tráfego indicados, foi feito um estudo sobre a arquitetura global das Software-Defined Networking (SDN), incidindo sobre os seus conceitos essenciais e áreas que abrange, tal como os protocolos que abarca e as tecnologias que abraça. Mencionou-se o protocolo OpenFlow (OF) e a razão pela qual a sua utilização está praticamente generalizada, sem esquecer o seu funcionamento e os campos que um *switch* com tal tecnologia apresenta nas tabelas de fluxo. Seguidamente, foi feita uma breve comparação sobre os demais controladores SDN existentes no mercado, descrevendo-os inicialmente e incidindo nas suas capacidades. Através desta, foi feita a escolha do controlador utilizado para a implementação dos mecanismos planeados, que, neste caso, seria o Floodlight. Passa-se, assim, para a descrição geral do encaminhamento em redes Internet Protocol (IP), referindo-se alguns dos protocolos existentes. Advindo destes, explicitaram-se os demais algoritmos que os caracterizam, assim como as suas vantagens e desvantagens consoante o tipo de infraestrutura em que estão inseridos. São abrangidos, consequentemente, os protocolos Routing Information Protocol (RIP) e Open Shortest Path First (OSPF), cuja utilização ainda é bastante elevada nos dias de hoje. Por último, foram indicadas algumas áreas de utilização das SDN, mencionando múltiplos trabalhos relacionados com a área em questão em conjunto com a temática de encaminhamento de tráfego.

Posteriormente, e tendo sido feita a escolha do controlador, foi projetada uma arquitetura capaz de emular múltiplos *switches* e *links* que contivesse múltiplas *end-user areas*, de forma a ser uma topologia equiparável a uma real, configurável por um administrador

de um Internet Service Provider (ISP), capaz de suportar variadas lógicas de encaminhamento. Alguns mecanismos foram desenvolvidos e justificados com a sua utilidade. Entre estes, afirmam-se os mecanismos de execução do Algoritmo de Dijkstra sobre a rede para obtenção do grafo da rede emulada; um módulo de notificação de falhas de *links* capaz de identificar falhas inesperadas de *links*; um módulo de recolha de estatísticas de tráfego geradas na infraestrutura; um módulo de convergência imediata em falhas de *links* capaz de responder às mesmas falhas de *links* inesperadas sem que haja qualquer atraso; um módulo de proteção de tráfego de entidades da infraestrutura cujo intuito é o de reserva de rotas para tráfego especial, previamente requisitado, limitando a utilização do mesmo para situações específicas; um módulo de reatividade a níveis de congestão, que lida com a possibilidade de degradação de serviço de *links* congestionados, desviando tráfego temporariamente através da injeção de novas rotas; um módulo de encaminhamento para fluxos individuais de tráfego, que, para tráfego diferenciado de endereços IP no domínio /32, estabelece rotas prioritárias e um módulo de multiplexagem da topologia física por diferentes redes virtuais, que permite a divisão de uma topologia em múltiplas sub-redes.

Para testar todos estes mecanismos, foram definidas baterias de testes específicas a cada um deles, sendo possível a conclusão do seu correto funcionamento, como comprovado pelos resultados enunciados, validando a solução como funcional.

Como tal, conclui-se que, as SDNs afirmam-se, com o passar do tempo, como principal resposta para todo o controlo centralizado de uma rede, independentemente do tamanho, facilitando quaisquer tarefas a um administrador de um ISP. Através das mesmas tecnologias SDN foi criada a solução desenvolvida, que explora as múltiplas características do controlador escolhido. É de se esperar, assim, que com o crescimento acelerado esperado de toda a tecnologia, e com a introdução global do conceito das 5G, as tecnologias SDN também cresçam e, deste modo, apresentem melhores e mais características que enriqueçam quaisquer futuras soluções.

5.2 Trabalho Futuro

Após o desenvolvimento da solução enunciada, segue, de trabalho futuro, esboçar, implementar, testar e executar outros novos mecanismos, capazes de se integrar com a mesma. Daí advirá, também, o aprimoramento das características atualmente implementadas, testando-as com cenários de teste maiores e mais completos, com a possibilidade de abarcar redes físicas, tentando, sempre que possível, dinamizar a solução de modo a que esteja preparada para lidar com o máximo de componentes possíveis. Só assim será possível a obtenção de resultados ainda mais fidedignos, capazes de se adequar a um meio menos simulado e mais adequado à realidade. A título de exemplo, e em relação a um dos mecanismos, o mecanismo de convergência imediata, deverá responder ao máximo de falhas

de *links* possível, possibilitando a deteção de falha do mesmo *link* mais do que uma vez por execução. Passará, também por trabalho futuro, o aumento do grau de facilidade de (re)configuração da solução, podendo esta, como caso análogo, ser extrapolada para uma interface gráfica, não sendo necessário modificar diretamente o próprio código implementado em cada um dos ficheiros de texto. Esta sugestão poderá, também, funcionar em tempo real, possibilitando, deste modo, a injeção de regras em qualquer altura, característica que provará ser útil para o módulo de proteção de tráfego, cuja inserção de *links* reservados é feita antes do início do arranque da solução. Por fim, e por outra perspetiva, estando o paradigma das redes Fifth Generation (cellular network technology) (5G) a crescer exponencialmente, o desenvolvimento desta solução através de controladores SDN, aliada a metodologias de baixa latência, poderá provar a sua utilidade para tal área, visto que a gestão centralizada de redes é, cada vez mais, a chave para o sucesso deste mesmo conceito.

BIBLIOGRAFIA

[1] M. Baddeley, R. Nejabati, G. Oikonomou, S. Gormus, M. Sooriyabandara, and D. Simeonidou. Isolating sdn control traffic with layer-2 slicing in 6tisch industrial iot networks. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 247–251, Nov 2017. doi: 10.1109/NFV-SDN.2017.8169876.

[2] Sumit Badotra and Japinder Singh. Open daylight as a controller for software defined networking. *International Journal of Advanced Computer Research*, 8, 05 2017.

[3] Josh Bailey and Stephen Stuart. Faucet: Deploying sdn in the enterprise. *Communications of the ACM*, 14, 12 2016. doi: 10.1145/3009828.

[4] K. Bakshi. Considerations for software defined networking (sdn): Approaches and use cases. In *2013 IEEE Aerospace Conference*, pages 1–9, March 2013. doi: 10.1109/AERO.2013.6496914.

[5] C. Chuang, Y. Yu, and A. Pang. Flow-aware routing and forwarding for sdn scalability in wireless data centers. *IEEE Transactions on Network and Service Management*, 15(4): 1676–1691, Dec 2018. ISSN 1932-4537. doi: 10.1109/TNSM.2018.2865166.

[6] David Erickson. The beacon openflow controller. pages 13–18, 08 2013. doi: 10.1145/2491185.2491189.

[7] L. Gkatzikis, S. Paris, I. Steiakogiannakis, and S. Chouvardas. Bandwidth calendaring: Dynamic services scheduling over software defined networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2016. doi: 10.1109/ICC.2016.7510888.

[8] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *Computer Communication Review*, 38:105–110, 01 2008.

[9] Vidya B. Harkal and A.a.deshmukh. Article: Software defined networking with floodlight controller. *IJCA Proceedings on International Conference on Internet of Things, Next Generation Networks and Cloud Computing*, ICINC 2016(3):23–27, July 2016.

[10] Vidya B. Harkal and Aaradhana A. Deshmukh. A border gateway protocol (bgp). In *RFC 1105 - Border Gateway Protocol*, 1989.

[11] C. Hedrick. Routing information protocol (rip). In *RFC 1058 - Routing Information Protocol*, 1988.

[12] F. Hu, Q. Hao, and K. Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys Tutorials*, 16(4):2181–2206, Fourthquarter 2014. ISSN 1553-877X. doi: 10.1109/COMST.2014.2326417.

[13] Sukhveer Kaur, Japinder Singh, and Navtej Ghumman. Network programmability using pox controller. 08 2014. doi: 10.13140/RG.2.1.1950.6961.

[14] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015. ISSN 0018-9219. doi: 10.1109/JPROC.2014.2371999.

[15] Open Networking Lab. Introducing onos - a sdn network operating system for service providers, 2014. URL https://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf.

[16] B. Lee, S. H. Park, J. Shin, and S. Yang. Iris: The openflow-based recursive sdn controller. In *16th International Conference on Advanced Communication Technology*, pages 1227–1231, Feb 2014. doi: 10.1109/ICACT.2014.6779154.

[17] J Lindqvist. Counting to infinity. 06 2004. URL https://pdfs.semanticscholar.org/fa54/bc2b006fa093860201a6e398443a1d43e63f.pdf.

[18] S. Luo, J. Wu, J. Li, L. Guo, and B. Pei. Context-aware traffic forwarding service for applications in sdn. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 557–561, Dec 2015. doi: 10.1109/SmartCity.2015.128.

[19] Z. Ma, G. Zhao, Q. Zhang, G. Lin, X. Wang, and H. Yin. An independent forwarding algorithm based on multidimensional spatial superposition model in sdn. In *2017 International Conference on Green Informatics (ICGI)*, pages 49–53, Aug 2017. doi: 10.1109/ICGI.2017.34.

[20] L. V. Morales, A. F. Murillo, and S. J. Rueda. Extending the floodlight controller. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 126–133, Sep. 2015. doi: 10.1109/NCA.2015.11.

[21] M. Mousa, A. M. Bahaa-Eldin, and M. Sobh. Software defined networking concepts and challenges. In *2016 11th International Conference on Computer Engineering Systems (ICCES)*, pages 79–90, Dec 2016. doi: 10.1109/ICCES.2016.7821979.

[22] J. Moy. Ospf version 2. In *RFC 2328 - Open Shortest Path First 2*, 1998.

[23] T. D. Nadeau and K. Gray. *SDN: Software Defined Networks*. O'Reilly, 2013.

[24] Open Networking Foundation. Software-defined networking: The new norm for networks. 04 2012.

[25] A. Nygren, B. Pfaff, B. Lantz, B. Heller, C. Barker, C. Beckmann, D. Cohn, D. Malek, D. Talayco, D. Erickson, D. McDysan, D. Ward, E. Crabbe, F. Schneider, G. Gibb, G. Appenzeller, J. Tourrilhes, J. Tonsing, J. Pettit, K. Yap, L. Poutievski, L. Dunbar, L. Vicisano, M. Casado, M. Takahashi, M. Kobayashi, M. Orr, N. Yadav, N. McKeown, N. dHeureuse, P. Balland, R. Madabushi, R. Ramanathan, R. Price, R. Sherwood, S. Das, S. Gandham, S. Curtis, S. Natarajan, T. Mizrahi, T. Yabe, W. Ding, Y. Yiakoumis, Y. Moses, and Z. Lajos Kis. Openflow switch specification, v1.5.1. pages 22–23, 2015.

[26] G. Pereira, J. Silva, and P. Sousa. Comparative study of software-defined networking (sdn) traffic controllers. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, June 2019. doi: 10.23919/CISTI.2019.8760997.

[27] I. Petrov and O. Morgunova. Forwarding rule minimization for network statistics analysis in sdn. In *2018 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*, pages 1–6, Oct 2018. doi: 10.1109/MoNeTeC.2018.8572182.

[28] J. Poller. Big switch networks: Next-generation data center networking. 2017. URL http://go.bigswitch.com/rs/974-WXR-561/images/ESG%20Lab%20Review%20-%20Big%20Switch%20Networks%20-%20April%202017.pdf.

[29] A. C. Risdianto and E. Mulyana. Implementation and analysis of control and forwarding plane for sdn. In *2012 7th International Conference on Telecommunication Systems, Services, and Applications (TSSA)*, pages 227–231, Oct 2012. doi: 10.1109/TSSA.2012.6366057.

[30] Eric C. Rosen. Exterior gateway protocol (egp). In *RFC 827 - Exterior Gateway Protocol*, 1982.

[31] O. Salman, I. Elhajj, A. Kayssi, and A. Chehab. Sdn controllers: A comparative study. pages 1–6, 04 2016. doi: 10.1109/MELCON.2016.7495430.

[32] N. Shen and H. Smit. Calculating interior gateway protocol (igp) routes over traffic engineering tunnels. In *RFC 3906 - Calculating Interior Gateway Protocol (IGP) Routes Over Traffic Engineering Tunnels*, 2004.

[33] J. Son, D. Kim, H. S. Kang, and C. S. Hong. Forwarding strategy on sdn-based content centric network for efficient content delivery. In *2016 International Conference on Information Networking (ICOIN)*, pages 220–225, Jan 2016. doi: 10.1109/ICOIN.2016.7427118.

[34] Ryu Project Team. Ryu sdn framework, 2013. URL https://osrg.github.io/ryu-book/en/Ryubook.pdf.

[35] Y. Wang, Y. Lin, and G. Chang. Sdn-based dynamic multipath forwarding for inter-data center networking. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–3, June 2017. doi: 10.1109/LANMAN.2017.7972146.

[36] X. Wen, K. Bu, B. Yang, Y. Chen, L. E. Li, X. Chen, J. Yang, and X. Leng. Rulescope: Inspecting forwarding faults for software-defined networking. *IEEE/ACM Transactions on Networking*, 25(4):2347–2360, Aug 2017. ISSN 1063-6692. doi: 10.1109/TNET.2017.2686443.

[37] Wikipedia. Bellman-ford's algorithm, 2018. URL https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[38] Wikipedia. Dijkstra's algorithm, 2018. URL https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[39] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie. A survey on software-defined networking. *IEEE Communications Surveys Tutorials*, 17(1):27–51, Firstquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2014.2330903.

[40] L. Y. Zhi, P. M. Mohan, V. Sridharan, and M. Gurusamy. Secondary controller mapping for reliable control traffic forwarding in sdn. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–2, July 2018. doi: 10.1109/ICCCN.2018.8487383.

A

CONFIGURAÇÃO

Neste anexo serão demonstrados os demais passos seguidos de forma a completar a configuração de todos os componentes presentes na máquina de testes.

A.1 MÁQUINA VIRTUAL

Foi utilizado o seguinte para virtualização do sistema operativo Ubuntu:

https://download.virtualbox.org/virtualbox/6.0.12/VirtualBox-6.0.12-133076-Win.exe

Em conjunção com a seguinte imagem em formato .ISO:

http://releases.ubuntu.com/16.04/ubuntu-16.04.6-desktop-amd64.iso

Seguido de instalação, set-up de conta de utilizador e arranque do mesmo.

A.2 JAVA 8

Antes da instalação do Floodlight, será requerida a instalação da versão 8 do Java. Esta é feita da seguinte forma:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-set-default
```

Seguindo, ao longo da instalação, as instruções da mesma.

A.3 MININET

Para introdução do Mininet na máquina virtual, faz-se uso do seguinte:

```
git clone git://github.com/mininet/mininet
cd mininet
mininet/util/install.sh -a
```

Para se testar a funcionalidade do mesmo, sugere-se a execução do comando:

```
sudo mn --test pingall
```

A.4 FLOODLIGHT

A instalação do Floodlight requer as seguintes *packages*:

```
sudo apt-get install git
sudo apt-get install build-essential ant maven python-dev
```

De seguida, é feito o *download* do mesmo controlador:

```
git clone git://github.com/floodlight/floodlight.git
cd floodlight
git submodule init
git submodule update
ant

sudo mkdir /var/lib/floodlight
sudo chmod 777 /var/lib/floodlight
```

A.5 ECLIPSE

A instalação do Eclipse é feita através do seguinte:

```
wget https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2019-06/
R/eclipse-inst-linux64.tar.gz
tar -xvf eclipse-inst-linux64.tar.gz
```

Seguidamente, procede-se à configuração do Eclipse com o Floodlight:

- Abrir Eclipse -> "New Workspace";
- "File-> "Import-> "General-> "Existing Projects into Workspace-> "Next";
- Carregar em "Browse"do "Select Root Directory"e selecionar a pasta do Floodlight;
- Marcar a *checkbox* para "Floodlight", sendo que mais nenhum projeto deverá estar presente ou selecionado;
- Carregar em "Finish".

Para executar o Floodlight no Eclipse:

- "Run-> "Run Configurations";
- Carregar com a tecla direita do rato em "Java Application-> "New";
- Selecionar FloodlightLaunch em "Name";
- Selecionar Floodlight em "Project";
- Selecionar net.floodlightcontroller.core.Main em "Main";
- Carregar em Apply.

Assim, e carregando no botão ao lado de ”Play”e selecionando o ficheiro a correr, deverá arrancar o Floodlight.

B

TOPOLOGIA

Neste anexo será explicitada a topologia utilizada, por completo, para realização dos demais testes enunciados ao longo da dissertação.

B.1 MININET

A topologia inserida no mininet é representada por:

```
from mininet.topo import Topo
from mininet.link import Link

class MyTopo( Topo ):
 "Simple Topology example."

 def __init__( self ):
 "Create custom topo."

 #Initialize topology
 Topo.__init__( self )

 #Add hosts
 h11 = self.addHost('h11',ip='10.0.1.1')
 h12 = self.addHost('h12',ip='10.0.1.2')
 h13 = self.addHost('h13',ip='10.0.1.3')
 h21 = self.addHost('h21',ip='10.0.2.1')
 h22 = self.addHost('h22',ip='10.0.2.2')
 h23 = self.addHost('h23',ip='10.0.2.3')
 h31 = self.addHost('h31',ip='10.0.3.1')
 h32 = self.addHost('h32',ip='10.0.3.2')
 h33 = self.addHost('h33',ip='10.0.3.3')
```

```
h41 = self.addHost('h41',ip='10.0.4.1')
h42 = self.addHost('h42',ip='10.0.4.2')
h43 = self.addHost('h43',ip='10.0.4.3')

#Add switches
s1 = self.addSwitch('s1',ip='10.0.1.4')
s2 = self.addSwitch('s2',ip='10.0.2.4')
s3 = self.addSwitch('s3',ip='10.0.3.4')
s4 = self.addSwitch('s4',ip='10.0.4.4')
s5 = self.addSwitch('s5')
s6 = self.addSwitch('s6')
s7 = self.addSwitch('s7')
s8 = self.addSwitch('s8')
s9 = self.addSwitch('s9')
s10 = self.addSwitch('s10')
s11 = self.addSwitch('s11')
s12 = self.addSwitch('s12')
s13 = self.addSwitch('s13')
s14 = self.addSwitch('s14')
s15 = self.addSwitch('s15')
s16 = self.addSwitch('s16')
s17 = self.addSwitch('s17')
s18 = self.addSwitch('s18')
s19 = self.addSwitch('s19')

#Host links
self.addLink(s1,h11)
self.addLink(s1,h12)
self.addLink(s1,h13)

self.addLink(s2,h21)
self.addLink(s2,h22)
self.addLink(s2,h23)

self.addLink(s3,h31)
self.addLink(s3,h32)
self.addLink(s3,h33)
```

```
self.addLink(s4,h41)
self.addLink(s4,h42)
self.addLink(s4,h43)

#Switch s1
self.addLink(s1,s5)

#Switch s2
self.addLink(s2,s18)

#Switch s3
self.addLink(s3,s19)

#Switch s4
self.addLink(s4,s6)

#Switch s5
self.addLink(s5,s10)
self.addLink(s5,s11)
self.addLink(s5,s7)
self.addLink(s5,s6)

#Switch s6
self.addLink(s6,s7)
self.addLink(s6,s13)
self.addLink(s6,s14)

#Switch s7
self.addLink(s7,s8)
self.addLink(s7,s9)

#Switch s8
self.addLink(s8,s11)

#Switch s9
self.addLink(s9,s13)

#Switch s10
```

```
        self.addLink(s10,s18)


        #Switch s11
        self.addLink(s11,s18)
        self.addLink(s11,s15)
        self.addLink(s11,s12)

        #Switch s12
        self.addLink(s12,s13)

        #Switch s13
        self.addLink(s13,s16)
        self.addLink(s13,s19)

        #Switch s14
        self.addLink(s14,s19)

        #Switch s15
        self.addLink(s15,s17)

        #Switch s16
        self.addLink(s16,s17)

        #Switch s17
        self.addLink(s17,s18)
        self.addLink(s17,s19)

        #Switch s18
        self.addLink(s18,s19)

topos = { 'mytopo' : (lambda : MyTopo() ) }
```