Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática

2017

Carlos Miguel Tavares da Silva

Balanceamento de carga em controladores de redes definidas por software

Load balancing on software-defined network controllers

Universidade de Aveiro
2017

Departamento de Eletrónica, Telecomunicações e Informática

Carlos Silva

Balanceamento de carga em controladores de redes definidas por software

Load balancing on software-defined network controllers

"Our doubts are traitors, and make us lose the good we oft might win, by fearing to attempt."

— William Shakespeare

Balanceamento de carga em controladores de redes definidas por software

Load balancing on software-defined network controllers

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação do Doutor Daniel Nunes Corujo, Investigador Doutorado do Departamento de Eletrónica, Telecomunicações e Informática e do Doutor Pedro Alexandre de Sousa Gonçalves, Professor Adjunto na Escola Superior de Tecnologia e Gestão de Águeda.

O júri / The juri

Presidente / President

Doutor Rui Luís Andrade Aguiar

Professor Catedrático no Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Vogais / Examiners Committe

Doutor Daniel Nunes Corujo

Investigador Doutorado no Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Doutor Pedro Miguel Neves Naia

Consultor Tecnológico na Altice Labs

Agradecimentos

Esta área serve para agradecer às pessoas que, de certo modo, contribuíram para realização desta dissertação. Primeiro, aos meus orientadores, pela motivação, partilha de conhecimento e suporte dado ao longo da realização da dissertação.

Aos meus pais pela oportunidade e incentivo no decorrer ao longo destes anos todos.

Por fim e não menos importante, quero agradecer às pessoas que todos os dias contribuem para o enriquecimento do conhecimento que a Universidade transmite aos seus alunos.

Palavras Chave *SDN, Distributed Control, OpenFlow*.

Resumo

As redes definidas por software (ou Software Defined Networks - SDN) contemplarão um importante alicerce nas redes futuras, ao providenciar novas formas mais flexíveis de operar a rede. Também no futuro, se prevê que existirão milhares de milhões de dispositivos ligados, o que irá exacerbar todas as condicionantes do controlo da rede. Até agora, o SDN tem-se focado maioritariamente na sua interação com os comutadores de rede, através de protocolos como o *OpenFlow* que precisam de ser endereçados.

Esta dissertação aborda esta problemática considerando o aspeto de balanceamento de carga em controladores de rede, propondo e analisando uma solução.

Keywords *SDN, IoT, Distributed Control, OpenFlow*.

Abstract Software Defined Network (SDN) will contemplate an important foundation for the future networks, by providing new flexible ways to operate them. Also in the future, it is expected that there will be billions of connected devices, which will exacerbate all the constraints of network control. So far, SDN has focused mostly on its interaction with network switches, through protocols such as *OpenFlow* which need to be addressed.

This thesis addresses this problematic considering the aspect of load balancing in Software-Defined Network Controllers, proposing and analysing a solution.

ÍNDICE DE CONTEÚDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE ACRÓNIMOS

ACL	Access Control List
API	Aplication programming interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
CLI	Command Line Interface
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DM	Data Mining
ForCES	Forwarding and Control Element Separation
FPGA	Field-Programmable Gate Array
GUI	Graphical User Interface
HAL	Hardware Abstraction layer
HF	HyperFlow
IoT	Internet of Things
IP	Internet Protocol
MD-SAL	Model-Driven-Service Abstraction Layer
NAT	Network Address translator
NBI	NorthBound Interface
ODL	OpenDayLight
OF	OpenFlow
OF-CONFIG	OpenFlow Configuration and Management Protocol
ONOS	Open Network Operating System
OSGi	Open Services Gateway initiative
OVS	OpenvSwitch
OVSDB	OpenvSwitch Database
PAD	Programmable Abstraction of Data path
POF	Protocol Oblivious Forwarding
QoS	Quality of Service
RAM	Random Access Memory
ROFL	Revised OpenFlow Library
RSPAN	Remote Switched Port Analyzer
SBI	SouthBound Interface
SDN	Software defined Network
SPOF	Single Point Of Failure

SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UPD	User Datagram Protocol
VLAN	Virtual Local Area Network
VM	Virtual Machine
VoIP	Voice over Internet Protocol
VTN	Virtual Tenant Network

Capítulo 1

1 INTRODUÇÃO

Hoje em dia, a Internet, ou as também designadas redes de computação, são uma componente fundamental de apoio a empresas, escolas e lares. A Internet interliga negócios, conhecimento e famílias à escala global. Contudo, ao longo destes últimos 20 anos de tradicionais redes de computadores, assistiu-se ao surgimento de limitações tais como a dependência de protocolos inalterados, o controlo distribuído, o hardware difícil de reprogramar [1], redes complexas e de difícil gestão [2], a necessidade de novos dispositivos com capacidades de rede, e a *Internet of Things* (sensores, câmaras, alarmes, entre outros) que acarreta desafios à gestão de rede.

Com o objetivo de resolver os problemas das redes tradicionais surgiu a tecnologia *Software Defined Network* [3], que traz consigo um novo paradigma de gestão de redes. Este separa o plano de controlo do plano de dados [4], centralizando o primeiro num dispositivo, o controlador. Todas as decisões de encaminhamento de um pacote com SDN passam por esse software centralizado fora dos *switches*. Os *switches* reencaminham, quando assim definido, os pacotes com base em decisões tomadas nesse software. Cada novo requisito ou comportamento com SDN não necessita da alteração de hardware, quer seja a alteração ou a atualização, sendo assim, unicamente necessário a atualização ou modificação do software de controlo. O SDN baixa os custos de operação, uma vez que só necessita que os *switches* tenham capacidade de comunicação usando o protocolo *OpenFlow*.

1.1 MOTIVAÇÃO

O crescente fluxo de dados no dia-a-dia sobre as redes computacionais sugere a necessidade de melhorar a abordagem SDN relativamente à distribuição do controlo de rede. Este crescimento do número de redes e dispositivos ligados conduz-nos a problemas como a sobrecarga dos controladores que, consequentemente, criam congestionamento de tráfego nas redes SDN. Para mitigar este problema, pretende-se implementar um mecanismo de distribuição de tráfego de controlo utilizando um controlador centralizado de modo a provar que é possível melhorar o desempenho e escalar a capacidade, tornando este sistema tolerante a falhas.

1.2 OBJETIVOS E CONTRIBUIÇÕES

Esta dissertação procura implementar um controlador de carga distribuída que vá ao encontro de apresentar respostas conclusivas quanto à viabilidade e performance deste tipo de controladores.

Assim os objetivos categóricos são:

- Implementar na rede:
  - Suporte *OpenFlow*;
  - Configurar rede;
- Implementar um controlador distribuído;
  - Implementação de um controlador;
  - Implementação do balanceador de carga;
- Identificar *uses cases;*
- Analisar implementação do Controlador Distribuído;

1.3 ORGANIZAÇÃO DA DISSERTAÇÃO

A presente dissertação é composta por seis capítulos, sendo o primeiro a introdução onde se faz uma breve relação do tema com o mundo real, e em seguida o estado da arte que descreve o paradigma SDN, bem como as principais tecnologias associadas e a análise dos últimos desenvolvimentos. No terceiro capítulo é descrita a arquitetura da solução e no quarto capítulo é descrita a implementação, as opções tomadas, o hardware e software utlizados. No quinto capítulo apresenta-se uma breve descrição dos testes realizados e são analisados os resultados obtidos. Por fim, no sexto capítulo, são apresentadas as conclusões, assim como as limitações, identificando-se trabalho futuro.

Capítulo 2

2 ESTADO DA ARTE

Na presente secção são apresentadas a definição do SDN, as motivações que levam ao surgimento desse paradigma, a sua arquitetura, o protocolo de comunicação *OpenFlow*, e são ainda descritas a funcionalidades básicas dos elementos que compõem as redes SDN.

O SDN é baseado no conceito de separação entre o plano de dados e de controlo, que comunicam através da interface *SouthBound Interface* com protocolos como o ForCES [5] e *OpenFlow* [3] . O SDN tem como casos de uso maximizar investimentos em virtualização de servidores e *clouds* privadas, disponibilizar programação de rede e incrementar segurança. No futuro, o SDN terá casos de uso como o SD-WAN (Software Defined Wide Area Networking) (Plataforma de gestão para controlar acessos remotos), micro segmentação (separar a rede em redes de acesso, *data centers*, diferentes implementações de segurança) e gestão de equipamentos de IoT (ajudar a priorizar o tráfego e realizar análises sobre o tipo de tráfego na rede) [6].

2.1 SOFTWARE DEFINED NETWORKS

Software defined networks é um paradigma de gestão de redes que efetua a separação entre o controlo de rede e o mecanismo de encaminhamento. O elemento controlador pode ser programado e controlado diretamente [3]. O SDN usa um controlador logicamente centralizado, que tem uma visão global da rede e dos seus equipamentos, que são controlados e configurados através de interfaces como o ForCES [5] e *OpenFlow* [3]. A arquitetura SDN consiste em três componentes principais: a baixo nível a de plano de dados, no nível intermédio a camada de controlo e no nível superior a camada de aplicação. Entre o plano de controlo e o plano de dados a comunicação é feita através do SBI *SouthBound* interface, ou seja, entre o controlador e o *switch*. Entre o controlador e as aplicações a comunicação é feita com o NBI *NorthBound* Interface [3]. Com esta estrutura, os programadores das aplicações adquirem uma maior flexibilidade para alcançar objetivos tais como engenharia de tráfego, QoS, métodos de segurança, métricas e monitorização de rede. Quando uma aplicação determina a alteração do comportamento de rede, o controlador, por usa vez, utiliza tabelas de fluxo dos *switches* para instruir esse novo comportamento. Isto significa que o

comportamento dinâmico da rede se encontra num nível superior em relação às redes tradicionais.

2.2 LIMITAÇÕES DAS REDES TRADICIONAIS

As redes de comunicação atuais têm limitações tais como [7]:

- **A configuração** de rede tradicional é morosa e é propensa a erros;
- Os administradores de rede efetuam muitos passos para gerir equipamentos na rede. Cada equipamento na rede é configurado individualmente. As configurações de ACLs, VLANs e QoS e outros serviços são feitas em cada equipamento isoladamente, sendo que o processo de configurar equipamento a equipamento torna-se lento e pode suscitar incoerência de regras, tornando-o mais tendente ao surgimento de falhas;
- **Requerem** bastante experiência;
- Em média, empresas ou organizações contêm equipamentos de vários fabricantes, o que implica que os administradores de rede tenham um conhecimento amplo sobre os diferentes equipamentos de rede;
- **Arquiteturas** tradicionais complicam a segmentação da rede;
- A evolução da quantidade de equipamentos com capacidade de se ligarem em rede levanta questões de conetividade. Com o crescente número de equipamentos inteligentes surge um novo desafio para as organizações, ou seja, como se irão ligar todos estes equipamentos de forma segura e estruturada. Ao colocar um equipamento vulnerável numa rede que contenha mais dispositivos, os equipamentos na rede podem estar acessíveis por intermédio do equipamento vulnerável;
- **Redes tradicionais** são estáticas e inflexíveis;
- **São *hardware appliances***, isto quer dizer que o hardware é desenhado especificamente para um software;
- **O Plano de controlo** é distribuído, cada equipamento contém o seu controlador embutido no hardware, onde normalmente o controlo é fechado;
- **Usam** ASICs e FPGAs;

Em conclusão, estas são algumas das limitações que têm surgido ao longo da existência das redes de computação tradicionais. Por forma a mitigar as limitações das mesmas, surge então o SDN como uma nova solução na gestão de rede.

2.3 ARQUITETURA DE SOFTWARE DEFINED NETWORK

A arquitetura de *Software Defined Network* consiste em três camadas como é possível observar na Figura 1. No bloco inferior tem-se o plano de dados, que contempla um conjunto de *switches*, e o plano de controlo passa por um sistema externo, que é o controlador. No bloco seguinte, a camada de controlo contém o controlador que comunica com os *switches* através de um protocolo, por exemplo *OpenFlow.* Esta camada, que é logicamente centralizada, é responsável por introduzir as regras nos fluxos dos equipamentos de rede. No último bloco

tem se o plano de aplicação, que engloba a interface para os programadores criarem aplicações para controlar a rede.

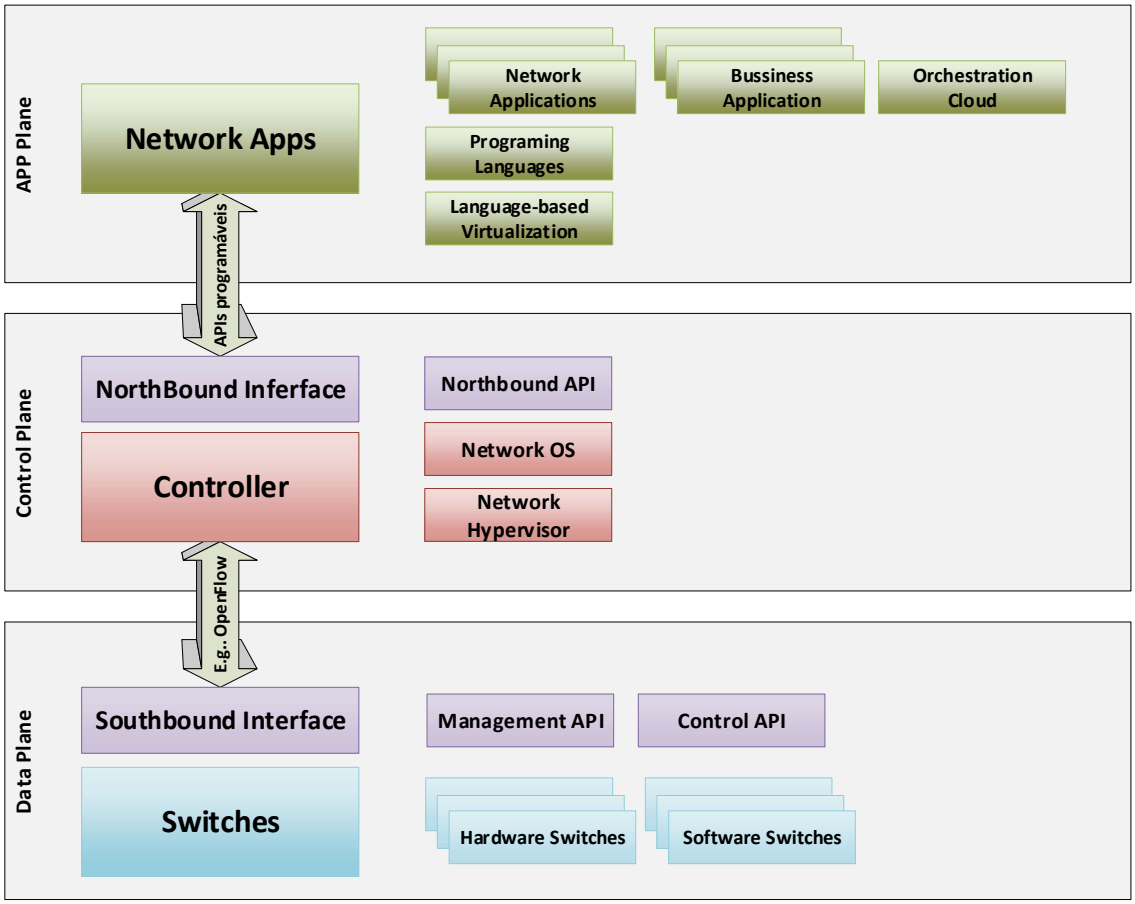


Figura 1 - Arquitetura de SDN [3]

- Com mais algum detalhe o plano de dados é composto por equipamentos de comutação como *switches*, *routers*, interfaces de software (*OpenFlow*, ForCES) e componentes de hardware no equipamento de rede.
- O plano de controlo é responsável pela inteligência da rede, e é operada por um controlador baseado em Software. Os elementos desta camada tomam as decisões para os equipamentos de rede.
- No plano de aplicação é onde se encontram instaladas todas aplicações e serviços capazes de utilizar os recursos da camada de controlo e infraestrutura. Esta camada permite que os programadores criem as suas aplicações tais como *routers, firewall,* balanceadores de trafego, e qualquer política ou regra implementada na rede sai desta camada, sendo o programador responsável por todo o comportamento de encaminhamento, ou até mesmo interligar com outras tecnologias.

2.4 CONTROLADOR *OPENFLOW*

O Controlador *OpenFlow* é o principal componente de SDN. O controlador é o responsável por centralizar a comunicação com todos os elementos programáveis de rede, fornecendo uma visão unificada da rede. O controlador é ainda responsável pela operação dos *switches*, sendo que este comanda a rede e pode influenciar todo o comportamento da mesma. Importa ainda frisar que esta interação com a rede pode ser feita em modo reativo ou em modo proactivo definidos na secção 2.4.1. Os controladores são concetualmente centralizados e apresentam muitos benefícios, tais como a programação centralizada da rede por aplicações sem preocupação com as tecnologias de configuração dos elementos da infraestrutura de rede subjacente [8]. Existem dois tipos de controladores: o controlador fisicamente e logicamente centralizado, em que os alguns dos *softwares* existentes são NOX [9], POX [10], Beacon [11], Floodlight [12], MuL [13], Maestro 14] e Ryu [15] e o controlador logicamente centralizado mas fisicamente distribuído [8] que surge para eliminar problemas de escalabilidade e disponibilidade entre outros motivos como:

- Larga distribuição de *switches: ex. entre regiões*;
- Alta latência entre um controlador e todos os *switches.*

Alguns dos principais controladores logicamente centralizados e fisicamente distribuídos são o ONOS [16] e o ODL [17], os quais serão apresentados mais à frente.

2.4.1 MODO REATIVO E PROATIVO

Os modos reativo e proactivo designam tipos de comportamentos que os controladores *OpenFlow* podem assumir através da implementação do plano de aplicação.

No modo reativo, o primeiro pacote pertencente a um fluxo é recebido por um *switch* e despoleta o controlador para inserir uma entrada de fluxo em cada *switch OF* da rede. Esta abordagem usa mais eficientemente a memória na alocação das tabelas de fluxo. No entanto, cada novo fluxo implica tempo de espera adicional para estabelecimento da conetividade [10]. Este modo de configuração do comportamento tem adicionalmente uma desvantagem importante, com uma forte dependência do controlador: se o *switch* perde a ligação, o encaminhamento de pacotes fica comprometido.

No modo proactivo, o controlador preenche previamente as tabelas de fluxos em cada *switch*. Esta abordagem não consome tempo no estabelecimento da conetividade a cada pacote no plano de dados porque as regras são definidas inicialmente. Porém, se o *switch* perde a ligação com o controlador este não quebra o tráfego. Como desvantagem, a operação da rede requer uma gestão pesada de toda a rede. Por exemplo, esta estratégia requer a agregação de regras para todo o tráfego que cubram toda a rede [10].

2.4.2 CONTROLADORES DISTRIBUÍDOS OPENFLOW

Nesta secção são apresentados e descritos diferentes controladores que suportam Openflow.

2.4.2.1 HYPERFLOW

É um controlador *event-based* distribuído[11]. Apesar de ser logicamente centralizado é fisicamente distribuído, fornecendo escalabilidade e resiliência ao particionamento de rede e falha de componentes. No *Hyperflow*, a decisão é feita em controladores de forma individual, para minimizar o tempo de resposta do plano de controlo aos pedidos do plano de dados [11]. O *Hyperflow* é implementado como uma aplicação para o NOX, e o estado de cada controlador é partilhado entre si, que para os *switches* irá fazer parecer que são um único controlador. Cada *switch* liga a um controlador na sua proximidade. O *Hyperflow* é consistente em toda a rede e corre como controlador total de rede. Para alcançar a consistência em toda a rede, cada controlador em *HyperFlow* publica os eventos que modificam o estado do sistema através de um sistema de comunicação *publish/subscribe*, e os outros controladores escutam e reconstroem o estado. Para implementar o sistema *publish/subscribe* é utilizada a *framework* WheelFS [12], que compõe um sistema de armazenamento distribuído de ampla área para ajudar a tolerância a falhas de aplicações *multi-site*.

2.4.2.2 ONIX

O controlador *Onix* é uma plataforma de controlo para redes de larga escala. O projeto *Onix* foi construído em NOX. O NOX denomina um sistema operativo de controlo de rede centralizado desenvolvido pela Nicara e introduzido para a comunidade em 2009, construído em C++. O NOX utiliza o controlador centralizado para gerir e os *switches* implementam a gestão. Apesar do NOX ser um controlador centralizado, o *Onix* permite a distribuição de controlo com NOX [13] .

2.4.2.2.1 Arquitetura

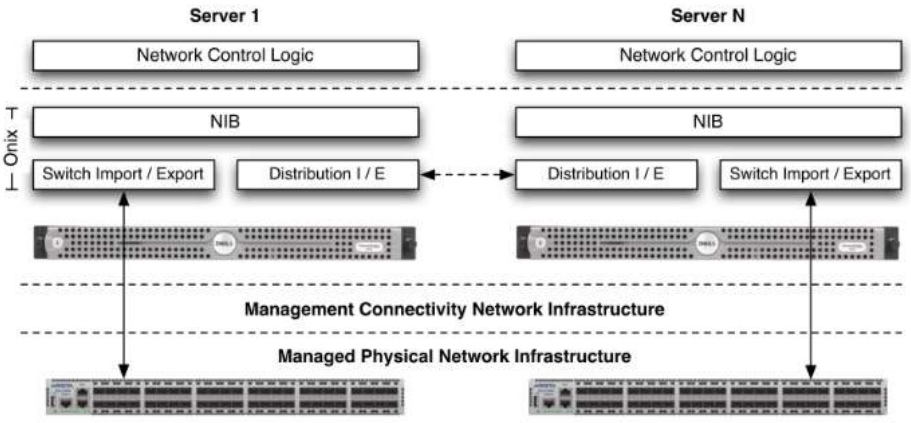


Figura 2 - Arquitetura do Onix [13]

A estrutura de controlo de rede pelo Onix consiste nas seguintes componentes:

- **A infraestrutura física**: *switches*, routers, e todos os elementos de rede que suportem uma interface;
- **A infraestrutura de conetividade:** é a componente de comunicação entre a rede física e a plataforma Onix. Este pode ser implementado *in-band* (tráfego de controlo partilha os mesmos elementos de encaminhamento que o tráfego de dados) ou *out-of-*

band (elementos de encaminhamento separados para o tráfego de controlo). Protocolos de encaminhamento como o IS-IS e OSPF são usados para construção e manutenção de estado de encaminhamento na componente *connectivity infrastructure* [13].

- **A Plataforma ONIX**: sistema que corre em cluster de um ou mais servidores físicos, cada um dos quais pode executar várias instâncias de Onix. O Onix é responsável por fornecer acesso à componente de controlo lógico e acesso à rede. Distribuído para escalar a redes grandes e oferecer a resiliência necessária para implementações de produção. Cada instância de *Onix* é responsável pela propagação do estado da rede para as outras instâncias dentro do cluster.
- **A arquitetura do Onix** consiste numa *network information base* (NIB), e outros dois componentes: *switch import/export* e *distribuition import/export*. Todos os elementos de rede ativos são guardados no NIB em pares chave-valor. O NIB é distribuído por diversas instâncias Onix. O componente *switch import/export* comunica para os *switches* se configurarem de acordo com as instruções provenientes dos nós Onix. A *distribution import/export* faz com que múltiplos NIB´s estejam consistentes entre si assincronamente.
- **A componente controlo de lógica** é implementada no topo da API do Onix. O *Control logic* ou a lógica de controlo determina o comportamento desejado da rede. O Onix permite o acesso ao estado da rede [23].

2.4.2.3 ONOS

O controlador ONOS [14] é um controlador de rede SDN distribuído e tem como pontos fortes a escalabilidade, a alta performance e disponibilidade. É um sistema *open source* hospedado pela *The Linux Foundation*. O controlador ONOS foi lançado em 5 de dezembro de 2014 pelos parceiros AT&T e NTT *Communications*. O ONOS é escrito em Java, oferece uma plataforma de aplicações distribuídas SDN em cima do Apache Karaf assim como o ODL. O sistema é projetado para operar em cluster, possibilitando que, quando exista uma falha num nó o controlo de rede não fique comprometido. Deste modo mantém-se a capacidade de controlo da operação de rede [15].

2.4.2.3.1 Lançamentos

Desde o início que o ONOS conta com vários lançamentos, sendo que nos mais recentes a documentação ainda não está completa. Na seguinte tabela pode-se observar todas as versões de ONOS lançadas até ao dia de hoje [16].

Tabela 1 - Lançamentos do controlador ONOS

Release Name	Data de lançamento
Avocet	Dezembro de 2014
BlackBird	Março de 2015
Cardinal	Junho de 2015
Drake	Setembro de 2015

Emu	Dezembro de 2015
Falcon	Março de 2016
Goldeneye	Junho de 2016
Hummingbird	Setembro de 2016
Ibis	Dezembro de 2016
Junco	Fevereiro de 2017
Kingfisher	Junho de 2017
Loon	Setembro de 2017

2.4.2.3.2 Arquitetura

A arquitetura do ONOS está desenhada com foco no fornecedor de serviços, e primazia a alta disponibilidade, escalabilidade e performance, assim como abstrações da interface *Northbound* e *Southbound* [16]. A Figura 6 esquematiza a arquitetura do ONOS.

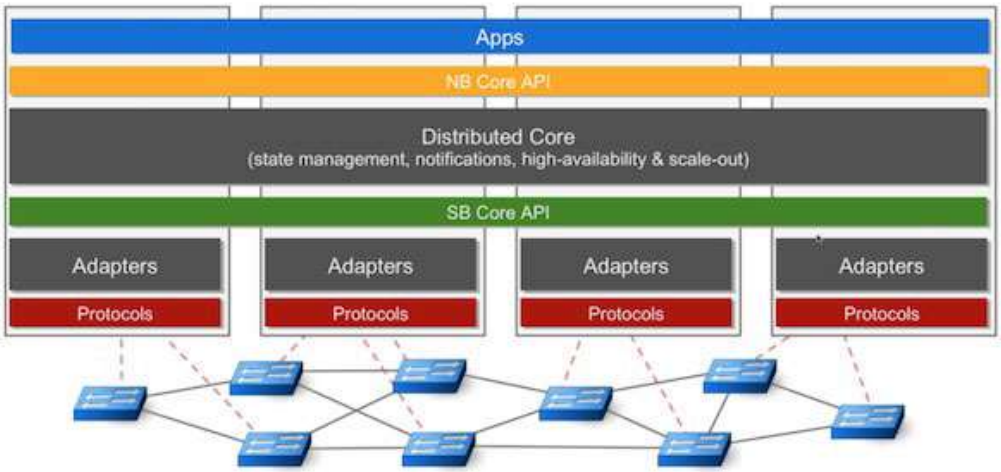


Figura 3 - Arquitetura do ONOS [17]

As principais características estratégicas do projeto ONOS são:

- **A Distribuição do core** que permite escalabilidade, alta performance e disponibilidade. Para que funcione como um controlador centralizado, o core de distribuição fornece serviços de mensagens, gestor de estados e eleição de líderes. No ONOS encontram-se embutidos protocolos de recuperação no caso de falhas de instância [16].
- **A *Northbound* API** disponibiliza grafos de rede, e *Intent Applications* para fácil desenvolvimento do controlo, gestão e configuração de serviços. As intenções de aplicação designam um objeto imutável que descreve a solicitação de uma aplicação ao core do controlador de rede para alterar o comportamento da rede. Tem como *Northbound API's* compatível a REST API;

- **A *Southbound* API** conecta os dispositivos via *OpenFlow*. Tem como *Southbound* API's compatíveis o OF de 1.0 a 1.3 e o NETCONF;
- Software em módulos para simplificar o desenvolvimento, depuração, manutenção, e a atualização do ONOS como um sistema de software por uma comunidade de programadores e fornecedores de serviços.

2.4.2.4 OPENDAYLIGHT

O OpenDaylight (ODL) [18] é outro controlador para a plataforma SDN, é *open source* e hospedado pela *The Linux Foundation*, apresentado em abril de 2013 e introduzido em fevereiro de 2014. Entre os fundadores deste projeto estão empresas como a Brocade, Cisco, Citrix, HP, IBM, Juniper Networks, Microsoft, NEC, Red Hat e VMware. O objetivo deste projeto é acelerar a adoção de SDN e criar um alicerce sólido para *Network Functions Virtualization* (NFV). Desenvolvido em Java, suporta *OpenFlow* 1.0, 1.3. [19] assim como no controlador distribuído ONOS [20].

O projeto *OpenDaylight* é aberto à comunidade, com código fonte aberto que permite aumentar o ciclo de vida do software e diminuir a complexidade do controlador. O ODL tem como ideia: construir aplicações na plataforma para aproveitar as funcionalidades como os *bundles* da mesma, cada qual exporta os serviços através de interfaces em Java. Outros paradigmas e ferramentas do ODL são o Maven para construir e gerir dependências, OSGi um *Backend container framework* que permite dinamicamente carregar *bundles* e o Karaf que define *containers* acima dos *packages* OSGi. A grande parte dos serviços são construídos no modelo fornecedor/consumidor sobre a camada MD-SAL como se pode observar na Figura 4. O ODL usa o padrão de arquitetura *Model-View-Control*, o que significa que a aplicação é dividida em três partes interligadas, a fim de separar representações internas de informação [21].

2.4.2.4.1 Lançamentos

O ODL, desde que surgiu, já apresentou os seguintes lançamentos:

Tabela 2 - Lançamentos do controlador OpenDayLight

Release Name	Data de lançamento
Hydrogen	Fevereiro de 2014
Helium	Outubro de 2014
Lithium	Junho de 2015
Beryllium	Fevereiro de 2016
Boron	Novembro de 2016
Carbon	Julho de 2017
Nitrogen	Setembro de 2017

2.4.2.4.2 Arquitetura

Como é comum no plano de controlo, a arquitetura do ODL consiste em blocos:

- A camada de controlo e coordenação formada por adaptações de serviços e funções de rede;
- A camada de aplicações e serviços *Northbound API*;
- A camada de extensões e protocolos *Southbound API*.

O ODL na camada de controlo tem o grupo *Base Network Service Functions*, como se pode verificar na Figura 4, que tem os serviços de gestão de topologia, estatísticas, switches, regras de encaminhamento, inventário, base de dados de elementos da rede e, por fim, Rastreador de host [21].

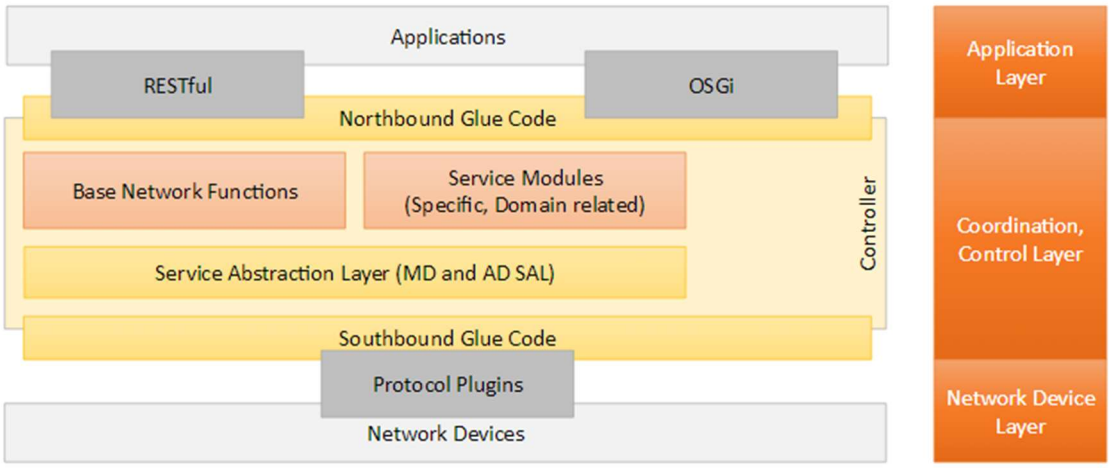


Figura 4 - ODL arquitetura [22]

A camada de aplicações tem serviços como GUI, VTN *coordinator*, *OpenStack Neutron* para interligação com sistemas operativos de *cloud*, SDNI *Wrapper* e Proteção DDoS. Todos os serviços mencionados passam por um filtro de Autenticação classificado AAA (*Authentication, Authorization, Accounting*) [21]. O método de distribuição de ODL é do tipo *cluster* em que existe um elemento que é o coordenador e todos os outros elementos são suplicantes.

2.4.2.5 DISCO

O controlador DISCO [23] é um projeto de controlador distribuído de rede da *"Agence naticionale de la recherche"*. Tem vindo a ser desenvolvido desde janeiro de 2014 e tem como data limite até junho 2017. Este projeto ainda está em fase de desenvolvimento, e por este motivo ainda não está disponível para a comunidade testar e avaliar. É, no entanto, um controlador a considerar pelo facto de ser distribuído. O DISCO pretende alcançar objetivos como flexibilidade, escalabilidade e resiliência em arquiteturas SDN. O DISCO difere em relação às outras propostas porque proporciona um plano de controlo distribuído para rede de larga área (WAN), redes restritas baseando-se num barramento de comunicação orientado para troca de mensagens entre as instâncias de controlo [23], [24].

2.4.2.5.1 Arquitetura

O controlador DISCO é constituído por duas componentes: a componente de intra-domínio, que reúne as principais funcionalidades do controlador, e a componente de interdomínio que

gere a comunicação com outros controladores DISCO (reserva, modificações de estado da topologia, monotorização, etc.). O controlador tem uma interface *Southbound* para enviar aos equipamentos de rede as regras e recuperar estados, e uma interface *Northbound* para receber regras, por exemplo, de uma aplicação de um gestor de rede [24]. A componente intra-domínio permite ainda que o monitor de rede gira a priorização do fluxo de modo que o controlador pode calcular os caminhos da prioridade de fluxo baseada no estado de diferentes parâmetros de redes. Permite ainda dinamicamente reagir a problemas de rede como falhas de ligação, latência alta, limite de largura de banda excedido, redirecionado ou interrompendo o tráfego de acordo com criticidade dos fluxos [24]. Em interdomínio, o DISCO usa dois elementos chave: um módulo de mensagem que descobre controladores vizinhos e a distribuição *publish/subscribe* do canal de comunicação e vários agentes que utilizam este canal para trocar informação da alta largura de banda com outros controladores [24].

2.4.2.6 ZEROSDN

O *Zero Software Defined Networking* (ZSDN) [25] é um controlador SDN distribuído que consiste em múltiplos módulos independentes que estão conectados por um *middleware* de *messaging* ZMQ. O ZSDN suporta *OpenFlow* 1.0 e 1.3, e foi desenvolvido por alunos durante um projeto na Universidade de Estugarda. Este controlador é totalmente distribuído o que significa que não existem instâncias centrais que controlam toda a rede. A comunicação entre os módulos é feita com recurso ao Google *protobuffers*. A execução de cada módulo é controlada por uma instância do Zero Module Framework [26]. A *Zero Module Framework* controla a execução dos módulos ligando-os ao ecossistema ZSDN via ZMQ e, por sua vez, o ZMQ interliga os controladores e os *switches*.

2.5 PROTOCOLO *OPENFLOW*

O *OpenFlow* é um protocolo de comunicação para a interface *SouthBound* entre o controlador e o *switch*, e o mais amplamente utilizado em SDN. Esta tecnologia normaliza a maneira de comunicar e é uma API para comunicação entre controladores e *switches*.

A primeira versão do projeto *OpenFlow* data de 2008, e a versão em uso é a 1.4, lançada em 2014. Em seguida, a Tabela 3 demonstra as novas características, motivos e casos de uso de cada uma das versões de *OpenFlow*.

Tabela 3 - Características das versões OF [27]

Version	Major Feature	Reason	Use Cases
1.0 - 1.1	Multiple table	Avoid flow entry explosion	
	Group Table	Enable Applying action sets to group of flows	Load balancing, Failover, Link Aggregation
	Full VLAN and MPLS Support		
1.1 - 1.2	OXM Match	Extend matching flexibility	
	Multiple Controller	HA/Load balancing/Scalability	Controller Failover, Controller Load Balancing
1.2 - 1.3	Meter table	Add QoS and DiffServ capability	
	Table miss entry	Provide flexibility	
1.3 - 1.4	Synchronized Table	Enhance table scalability	Mac Learning/Forwarding
	Bundle	Enhance switch synchronization	Multiple switch configuration
1.4 - 1.5	Egress Table	Enabling processing to be done in output port	
	Scheduled bundle	Further enhance switch synchronization	

Como se pode observar na Figura 5, a tabela apresenta a evolução do OF desde 2009 até aos dias de hoje. Neste momento o OF continua a sofrer melhorias como se observa. Ainda se pode verificar que a versão 2.0 de OF ainda não tem data de lançamento, mas de acordo com [28] terá contribuições do protocolo P4.

O protocolo P4 é uma linguagem para expressar como os pacotes são processados pelo plano de dados de um elemento de encaminhamento programável como um *hardware* ou *software switch*, ou *router*. O P4 funciona em conjunto com protocolos de controlo de SDN como o *OpenFlow*. O protocolo *OpenFlow* indica ao *switch* o que deve fazer, mas não a forma como o deve e é isso que o P4 faz. O P4 diz como o *switch* deve processar os pacotes. O P4 permite definir os cabeçalhos que o *switch* deve reconhecer, como combinar em cada cabeçalho, e quais as ações que ocorrem em cada diferente cabeçalho [29].

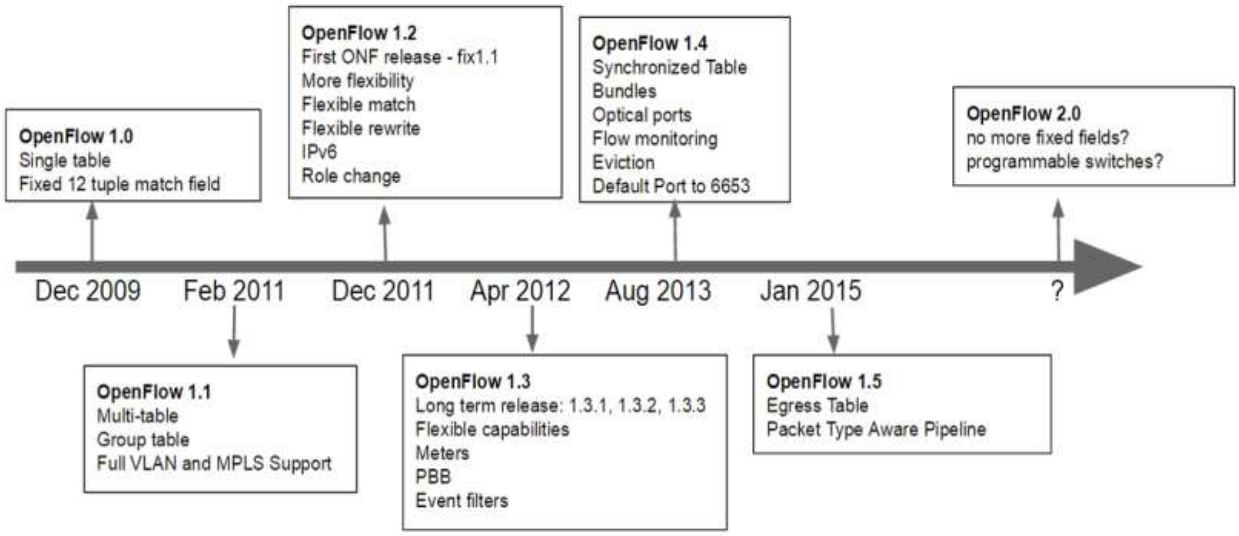


Figura 5 - Cronograma de evolução do OpenFlow [27]

2.5.1 MENSAGENS DO PROTOCOLO *OPENFLOW*

As mensagens do protocolo *OpenFlow* podem ser classificadas de acordo com três tipos de mensagem, trocadas entre o controlador e o *switch* que suporta este tipo de protocolo:

- ***Controller-to-switch*** - são as mensagens iniciadas pelo controlador, para solicitar os recursos comutados ou atualizar a configuração do *switch*;
- ***Asynchronous*** - são as mensagens enviadas pelo *switch* para o controlador de forma a atualizar os erros, tabelas de fluxo e modificação de estados das portas;
- ***Symmetric*** - são as mensagens enviadas (e iniciadas) de qualquer controlador de *switch*, como por exemplo *hello*, *echo,* usadas para confirmar o estado da ligação deste tipo;

Em seguida são apresentadas as mensagens mais relevantes no âmbito do trabalho elaborado (a especificação do *OpenFlow* contém a lista completa [30]) :

- **Hello** – Mensagem do tipo *symetric*, trocada entre o *switch* e o controlador para inicializar a ligação.
- **Echo** – Mensagem do tipo *symetric,* trocada entre o *switch* e o controlador para manter a ligação ativa. Também podem ser usadas para indicar a latência, a largura de banda e o estado de vida da ligação controlador-*switch*.
- **Features** – É uma mensagem do tipo *controller-to-switch,* que após a sessão TLS estabelecida, envia a mensagem de *features request* para o *switch*. O *switch* deve responder com *features reply* que especifica as capacidades suportadas pelo *switch*.
- **Configuration** – É uma mensagem do tipo *controller-to-switch*, onde o controlador pode definir e questionar os parâmetros de configuração no *switch*. O *switch* unicamente responde à questão do controlador.

- **Barrier** – É uma mensagem do tipo *controller-to-switch*, usada pelo controlador para garantir que as dependências de mensagens foram atendidas ou para receber notificações para operações completas.
- **Modify-State ou Flow_mod** – É uma mensagem do tipo *asynchronous* enviada pelo controlador para o *switch*; o objetivo primário desta mensagem é adicionar, apagar, ou modificar os fluxos nas tabelas de fluxos e para definir propriedades da porta de *switch*.
- **Packet-in** –É uma mensagem do tipo *asynchronous*; este tipo de mensagem é enviado pelo *switch* ao controlador quando os pacotes do plano de dados não têm uma entrada de fluxo correspondente.

2.6 SWITCHES OPENFLOW

Tal como os *switches* convencionais, os *switches* SDN comutam tráfego, mas a capacidade de decisão fica num controlador remoto. Para obter este comportamento os *switches* são constituídos por uma ou mais tabelas de fluxos que são preenchidas por um controlador, ou seja, são controlados e programados no plano de controlo.

Nas tabelas de fluxo são implementadas as regras pelo controlador que podem modificar a operabilidade do *switch,* que consoante as regras, se podem assumir como *router*, *NAT*, *firewall*, balanceador de carga, entre outros, de redes tradicionais.

Um *switch OpenFlow* pode ser um *software* ou *hardware* que consiste em três principais componentes:

- **Flow table** – consiste em várias entradas de fluxo, cada uma dividida em seis partes (Figura 6);
- **Secure channel –** normalmente utiliza-se com a finalidade de comunicação com TLS ou SSL entre o *switch* e o controlador;
- **Protocolo *OpenFlow*** – é o protocolo para a comunicação com os *switches*.

Em *switches* SDN existem dois tipos: os que são compatíveis com *OpenFlow,* ou os que são nativos *OpenFlow* [3], como se pode observar na Figura 6. Os *switches* compatíveis com *OpenFlow* podem operar nos dois modos como *switch* de rede tradicional ou como *switch OpenFlow*.

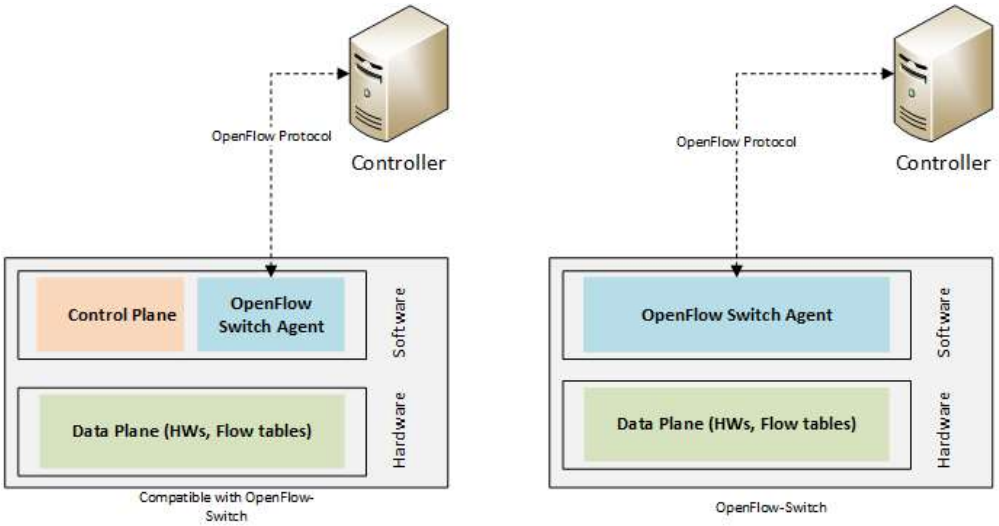


Figura 6 - Diferença entre SW OpenFlow e SW compatível com OpenFlow. Fonte: [3]

Para ambientes em que se usa rede virtualizada, existe o *OpenvSwitch* [31] (OVS) que é um *switch* virtual. O OVS é responsável pelo encaminhamento das mensagens entre VMs. O OVS suporta protocolos como o *NetFlow*, *sflow*, CLI, RSPAN. O protocolo de comunicação é o *OpenFlow*. O OVS contém várias tabelas de fluxo, sendo que estas recebem as instruções de fluxo por meio do driver *OpenFlow*. No modo reativo, caso não exista a instrução de fluxo ou regra na tabela de fluxo, o OVS envia ao controlador a mensagem *Packet_in* e espera pela decisão do mesmo. Caso não existam respostas o OVS rejeita o pacote [31]. *OpenvSwitch* suporta desde a versão 1.0 à versão 1.3 do protocolo *OpenFlow*[3].

2.6.1 COMPOSIÇÃO DA TABELA DE FLUXO.

A tabela de fluxo é atualizada pelo controlador que remove e adiciona entradas recorrendo ao protocolo *OpenFlow*. Uma tabela de fluxo compreende um conjunto de entradas (valores de cabeçalho para combinar pacotes novamente), contadores de atividade, e um conjunto de zero ou mais ações para aplicar a pacotes que cruzem com entradas da tabela de fluxos. Todos os pacotes que entram no *switch* são comparados na tabela de fluxo. Caso se encontre uma entrada correspondente, a ação indicada nessa entrada é aplicada ao pacote (por exemplo, um pacote pode ter uma ação que pode encaminhar para uma porta específica). No caso de não encontrar, o pacote é reencaminhado para o controlador por via do protocolo *OpenFlow*.

As entradas de fluxo podem encaminhar os pacotes para uma ou mais portas *OpenFlow*. Em geral estas portas são físicas, mas o protocolo não impede abstrações como agregações de portas ou tráfego VLAN numa porta que aparece como *OpenFlow*.

Apresenta-se, de seguida, a tabela de fluxo que é constituída pelas componentes:

Tabela 4 - Parâmetros da tabela de fluxo OpenFlow

Parâmetros	Descrição
Header fields	cada cabeçalho de fluxo de entrada é composto por seis elementos, que definem as regras de *matching*. Engloba seis parâmetros: *Match fields*: usado para combinar os pacotes com base na porta de chegada e cabeçalhos dos pacotes. *Priority*: usado para definir prioridades para diferentes entradas de fluxo. *Counters*: usado para estatística, conta sempre que um pacote combina. *Instructions*: indica ao pacote combinado alterações que podem resultar em modificações ao pacote. *Timeouts*: usado para definir o tempo de vida daquela entrada de fluxo até expirar. *Cookies*: usada para filtrar estatísticas de fluxo, é um valor de dados opaco escolhido pelo controlador.
Actions	tem a decisão para aplicar aos pacotes, cada fluxo é associado com zero ou mais ações que indicam como os equipamentos tratam os pacotes da respetiva. Se numa entrada de fluxo não estiver presente uma *action* de encaminhamento o pacote é descartado. Os *switches* não suportam todos os tipos de *action*. Quando o *switch* se conecta ao controlador indica quais os tipos de *action* que suporta. Como anteriormente indicado, existem 2 tipos de *switches*, os *OpenFlow only* que suportam as *required actions* e os *OpenFlow-enabled* que suportam *Required Action* e *Optional Action*. Na Tabela 5 pode-se observar as ações e os tipos que pertencem

Tabela 5 - Lista de ações requeridas e opcionais

Forwarding	Requeridas	All
		Controller
		Local
		Table
		In-Port
	Optional	Normal
		Flood
Drop	Required	
Modify Field	Optional	

2.6.2 COMPONENTE DE MATCHING

Ao receber um pacote o *switch OpenFlow* despoleta uma sequência de processos e decisões a que se dá o nome de *Matching,* como se pode constatar na Figura 7*.*

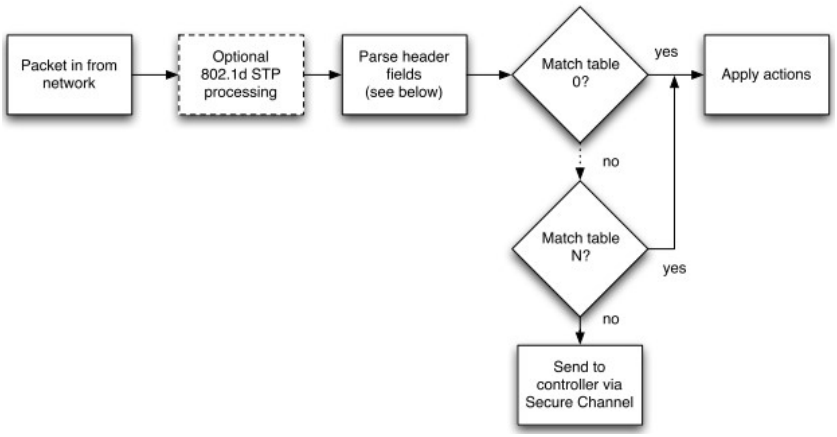


Figura 7 - Fluxograma de Matching OpenFlow [32]

Este processo é iniciado pelo *switch* que recebe um pacote proveniente do plano de dados, e que este envia opcionalmente para o processamento de 802.1D Spanning-tree para validar se existem pacotes duplicados na rede (ou *loops*). Em seguida, são analisados os campos do cabeçalho do pacote, e são tomadas decisões comparando os campos do pacote com as entradas da tabela de fluxo. Esta ação vai de tabela em tabela até encontrar uma correspondência, e assim que encontre aplica a *action*. Caso não exista a correspondência o pacote é enviado para o controlador para processar uma decisão.

2.7 CONCLUSÃO

Ao longo desta secção foi demonstrado o estado de arte que é um trabalho científico em que identifica as últimas tecnologias do tema, e incrementa o conhecimento de conceitos, tecnologias e paradigmas. Mais concretamente os conhecimentos abordados foram a arquitetura SDN que incluem controladores, protocolos de comunicação de *Southbound interface* e os *switches* OP. Dentro do grupo dos controladores apresentou-se os controladores distribuídos ODL, ONOS entre outros. Com esta aprendizagem efetuada foram solidificados os conhecimentos que compõem a base da solução apresentada no capítulo seguinte, que compõe a arquitetura do balanceador de carga para controladores SDN.

Capítulo 3

3 ARQUITETURA

No presente capítulo, é apresentada a arquitetura global, demonstrando como os equipamentos de rede interagem entre si, bem como os algoritmos de distribuição de carga, e como é feita a troca de mensagens entre os *switches* e os controladores.

3.1 VISÃO GERAL

A Figura 8 apresenta a arquitetura que consiste num controlador distribuído, composto por um módulo de balanceamento e várias instâncias de controlo, sendo que cada *switch* está ligado ao módulo de balanceamento.

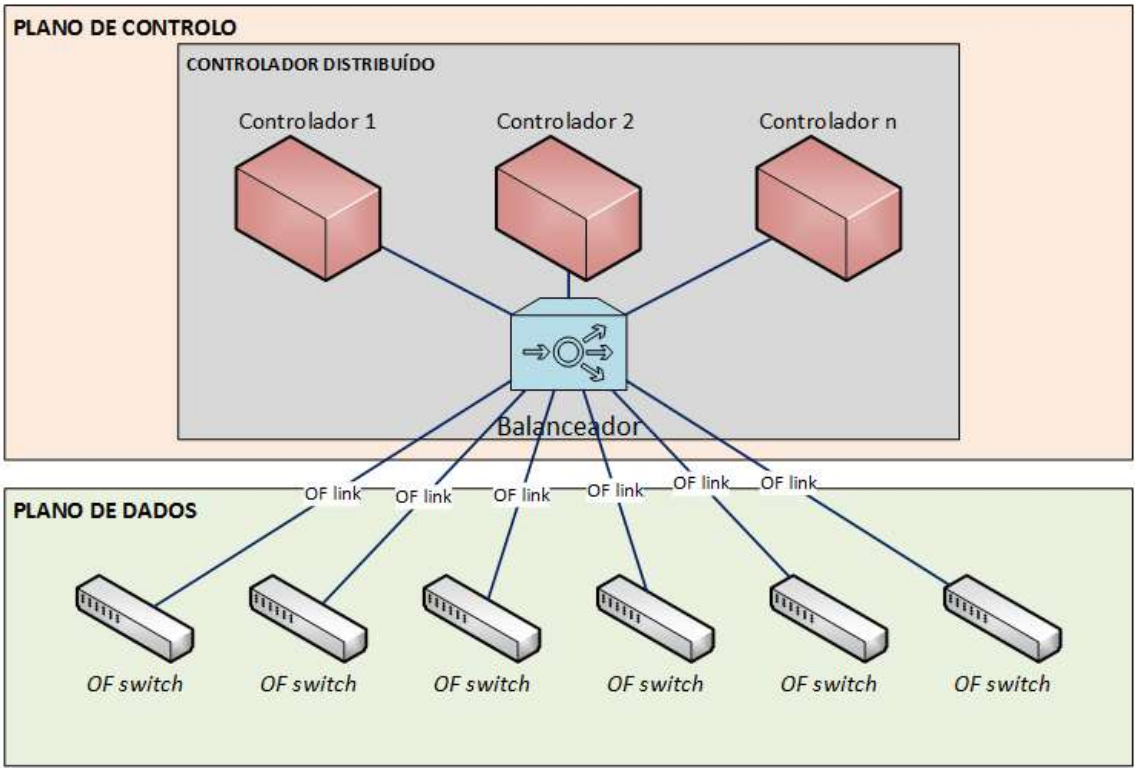


Figura 8 - Topologia de rede.

No plano de controlo existe o controlador distribuído, que pode ter múltiplas instâncias ligadas a um balanceador. O balanceador serve a distribuição do controlo neste controlador.

3.2 ARQUITETURA DA DISTRIBUIÇÃO

Neste subcapítulo, abordam-se as possíveis estratégias para a distribuição, algoritmo de distribuição e como é processada a troca de mensagens entre os *switches* e as instâncias de controlo. Com o propósito de obter distribuição existem estratégias que podem ser adotadas, como as seguintes:

- ***Master-Master*** ou Replicação Ativa [33];
- ***Master-Slave*** ou Replicação Passiva [33];

Na estratégia de distribuição *Master-Master* [33], cada nó é chefe das suas decisões e em nenhum momento são escravos ou submissos a outro nó (Figura 9).

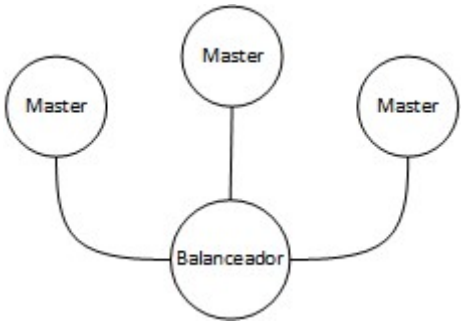


Figura 9 - Distribuição Master-Master.

A estratégia *Master-Slave* [33], como descrito na Figura 10, tem como abordagem um *Master* e todos os outros elementos como *Slave*, sendo utilizada em soluções como o ONOS e ODL.

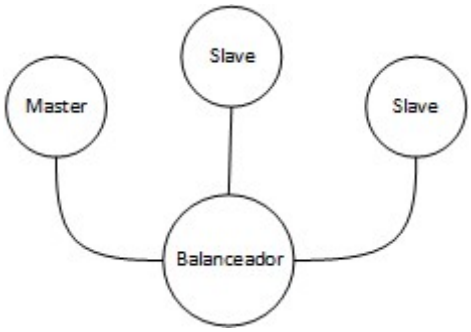


Figura 10 - Distribuição Maste-Slave.

Para distribuir o tráfego dos *switches* pelos controladores é necessária uma estratégia como se pode observar na Figura 10. Os *switches* enviam o tráfego para o balanceador, que por sua vez, escolhe o controlador e distribui o tráfego. Para escolha do controlador é utilizado um algoritmo de balanceamento, dos quais alguns deles estão representados na Tabela 6.

Tabela 6 - Algoritmos de distribuição

Algoritmo	Comportamento
Round robin	A cada novo fluxo vai para o seguinte servidor no ciclo
Weighted round robin	Igual ao *Round robin* mas cada servidor tem um peso atribuído
Least connected	O Servidor com o menor número de fluxos recebe o novo pedido
Weighted least connected	Igual ao *Least Connected* mas cada servidor tem um peso atribuído

O algoritmo de balanceamento *weighted round robin* e o *weighted least connection* atribuem um peso a cada servidor. De modo a tornar a distribuição do tráfego mais eficiente, o valor do peso pode ser baseado num algoritmo de monitorização composto pelas variáveis de consumo de RAM, consumo de CPU e largura de banda do servidor. O peso teria a seguinte fórmula:

$$Peso = (percentagem\ de\ CPU * 0{,}4) + (percentagem\ de\ RAM * 0{,}1) + (percentagem\ de\ numero\ de\ ligacões * 0{,}2) + (percentagem\ do\ tempo\ de\ resposta * 0{,}3)$$

E a normalização do resultado em percentagem, por exemplo, para um intervalo de 0 a 10 utilizando a fórmula seguinte:

$$Resultado\ normalizado = (m * 100)/10$$

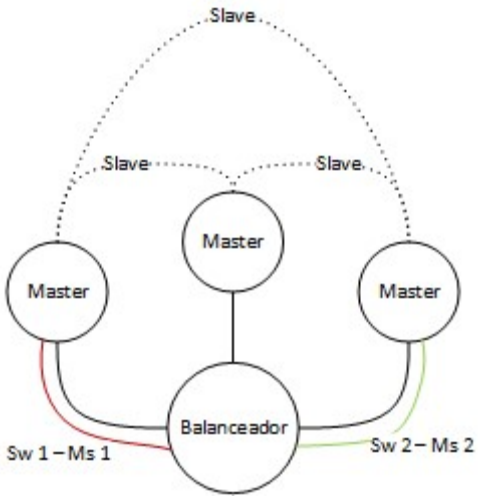


Figura 11 - Distribuição Balanceador-Maste-Master.

Para esta implementação propõe-se a distribuição de carga representada na Figura 11 com o paradigma *Master-Master* e *Round-Robin* como algoritmo de distribuição.

3.2.1 TROCA DE MENSAGENS ENTRE CONTROLADOR E SWITCH

Num processo natural de *OpenFlow*, o *switch* e o controlador trocam mensagem entre si sem interferência na ligação dos seus *drivers*, mas nesta arquitetura, entre o controlador e o *switch,* existe um balanceador que gere a distribuição das comunicações entre os controladores e os *switches*. Esta arquitetura não modifica estruturalmente o protocolo *OpenFlow* e não cria limitações ao seu funcionamento. O balanceador deve para cada *switch* criar um canal de comunicação que se mantém aberto durante todo o tempo de execução do *switch.* Na Figura 12 apresenta-se a mensagem de *hello* entre o controlador e o *switch* quando estabelecem a ligação.

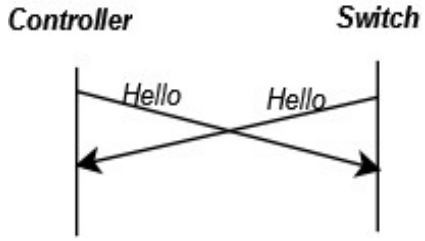


Figura 12 - Mensagem Hello OpenFlow [34]

Com balanceador, todas a mensagens de *Openflow* passam pelo balanceador, como se pode observar na Figura 13.

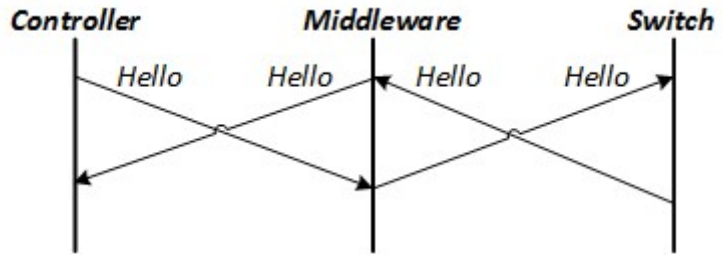


Figura 13 - Mensagen Hello OpenFlow com Middleware.

O mesmo comportamento do balanceador repete-se para todas as mensagens de *OpenFlow* e tal como se pode constatar na Figura 14, que mostra a sequência de todas as mensagens trocadas entre os vários *switches* e os controladores. A Figura apresenta múltiplos elementos do plano de controlo e do plano de dados, para demonstrar quais são os elementos que podem sofrer modificação a nível quantitativo na arquitetura.

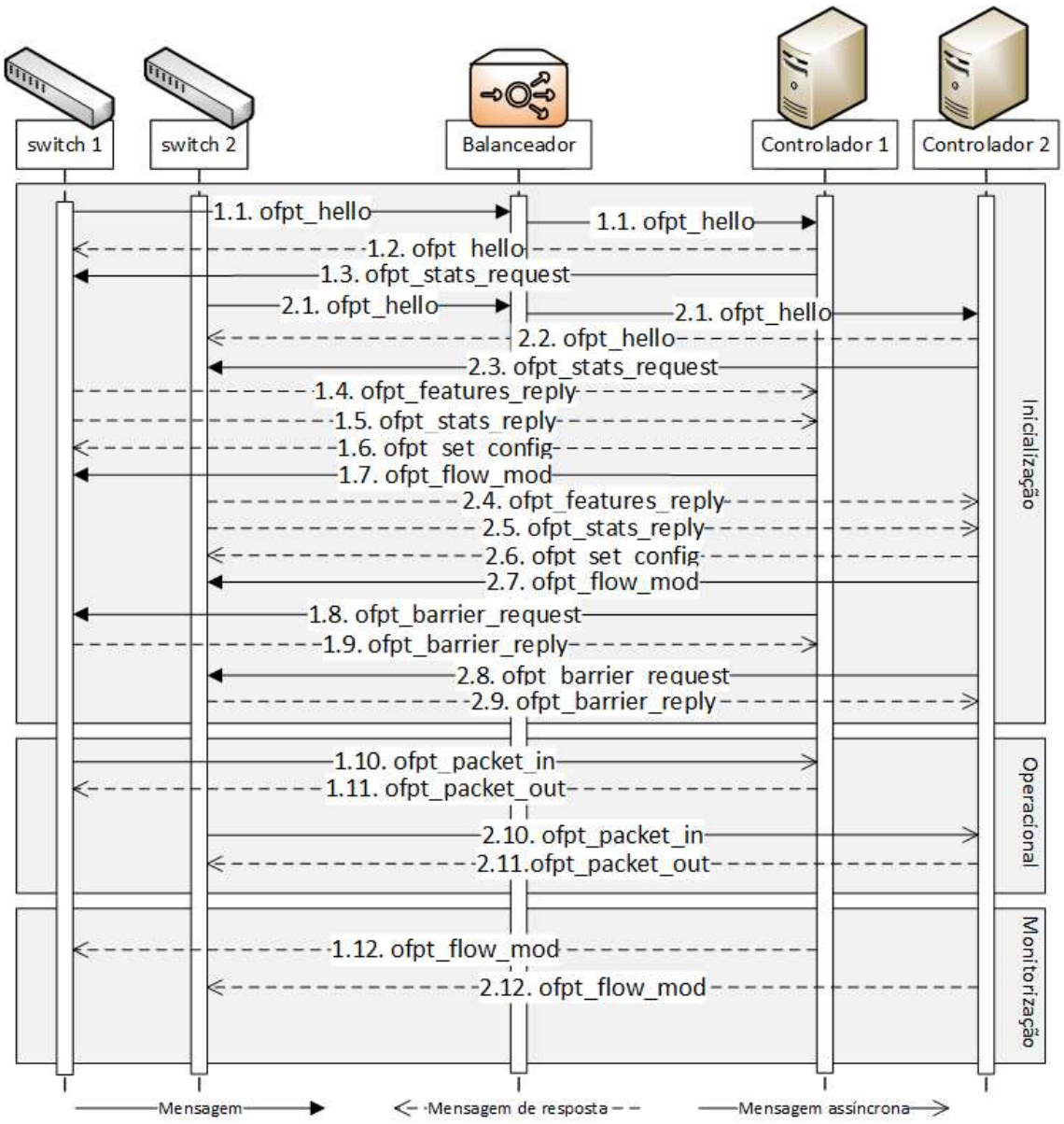


Figura 14 - Sequência de mensagens entre os switches e os controladores.

A sequência de mensagens trocada entre os *switches*, o balanceador e os controladores, que vai do intervalo de 1.1. a 1.9. e de 2.1. a 2.9., são mensagens de inicialização da comunicação entre os *switches* e os controladores de *OpenFlow*. A primeira mensagem enviada pelo *switch* 1 é a 1.1.1. ofpt_hello, que é composta pelo campo *version* onde se encontra indicada a versão mais alta que o *switch* suporta de *OpenFlow*, e tem como destino o endereço do balanceador que por sua vez reencaminha a mensagem 1.1.2. opft_hello para um controlador. O controlador recebe a mensagem, faz interpretação da mesma e em seguida responde com a mensagem 1.2. ofpt_hello, prosseguido pela mensagem 1.3. ofpt_stats_request. A mensagem 1.3. pergunta sobre o estado corrente do *datapath* (identificador da instância *OpenFlow* no *switch*). A mensagem 1.5. ofpt_stats_reply é a resposta à mensagem 1.3. sobre o *datapath* e em ambas existe a variável *type* que indica o tipo de informação, que pode ser uma descrição do *switch*, estatísticas individuais de fluxo, aglomerados de fluxo, de tabelas de fluxo, de portas físicas, de *Queue*, ou do fabricante. A mensagem 1.5 contém a variável *flag* que indica se existem mais mensagens para além desta. Na mensagem 1.4. ofpt_features_reply o *switch*

indica ao controlador informações sobre o *datapath ID*, o número de *buffers* e o número de tabelas. A mensagem 1.6., proveniente do controlador 1 e em direção ao *switch* 1, é uma mensagem de inicialização que permite ao controlador modificar a configuração dos *switches*. A mensagem 1.7. é uma mensagem do controlador para o *switch* que dá instrução para modificar a tabela de fluxo do *switch*. A mensagem 1.8., do controlador para o *switch*, serve para o controlador definir um ponto de sincronização, garantindo que todas as mensagens de estado são completadas antes do *switch* responder ao controlador. A mensagem 1.9 é a resposta à mensagem 1.8 composta pelo *xid* do pedido original e é enviada ao controlador quando conclui o ponto de sincronização. As mensagens no intervalo de 1.10. a 1.11. e 2.10. a 2.11. são mensagens de Operação do *OpenFlow*. A mensagem ofpt_packet_in (1.10.) é enviada ao controlador quando surge uma entrada de um pacote no plano de dados que, por sua vez, responde com uma mensagem do tipo ofpt_packet_out (1.11.). Para a mensagem 1.10. não existe uma resposta linear na especificação *OpenFlow* mas o controlador pode responder com mensagens do tipo ofpt_flow_mod ou ofpt_packet_out. Este comportamento depende da implementação da aplicação do controlador. Relativamente às mensagens com numeração iniciada pelo número dois são mensagens da segunda ligação entre o *switch 2* e controlador 2 e têm o mesmo comportamento demonstrado nas mensagens 1, pois a sua representação no diagrama tem como objetivo demonstrar que todas as mensagens passam por um balanceador de carga e que a distribuição é feita para **N** controladores.

3.3 CONCLUSÃO

Neste capítulo foi apresentada a arquitetura geral da solução desenvolvida no âmbito deste trabalho. Da mesma, o balanceador de carga é o elemento responsável pela distribuição do plano de controlo *OpenFlow* entre os diversos controladores e os comutadores da rede, tendo sido optado por realizar a distribuição sob forma de replicação ativa (Master-Master), onde cada novo fluxo é canalizado para o próximo controlador disponível segundo um princípio de *round-robin*. Para terminar, foi também apresentada a sequência de mensagens da sinalização *OpenFlow* com recurso ao uso de um balanceador operando seguindo estes princípios. O capítulo seguinte irá detalhar a implementação desta arquitetura.

Capítulo 4

4 IMPLEMENTAÇÃO

Este capítulo detalha a implementação da distribuição do controlador e da rede para testes. A secção de implementação descreve o controlador utilizado, a instalação, implementação e distribuição do controlo. Na implementação da rede fala-se sobre a instalação do software para emular a rede para fazer testes.

4.1 CONTROLADOR

Para a implementação do controlador foi escolhido o POX [35] da Nicira. O POX é o sucessor do NOX [9]. O POX requer o *Python* [36] versão 2.7 para ser executado, o que permite ser executado em ambientes Windows, Linux e Mac OS. Para visualizar a topologia do POX existem ferramentas de visualização como o Gephi. O POX é um controlador *single-thread*. O POX é unicamente compatível com a versão 1.0 de *OpenFlow*.

O POX funciona no paradigma *publish/subscribe* e é um sistema do tipo *event-driven.* No padrão *publish/subscribe* existem três principais componentes: o editor, o assinante, e o evento. O editor, ou Remetente, pode publicar um evento sem qualquer conhecimento do assinante ou destinatário. O assinante ou destinatário pode assinar ou escutar um editor. Um editor pode ser assinado por múltiplos assinantes.

Por norma, para escrever uma aplicação com padrão *publish/subscribe*, são necessárias 3 classes, uma classe evento, uma classe publicadora, e uma classe subescritora. A imagem seguinte mostra a orientação do padrão.

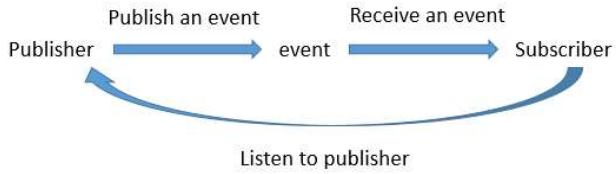


Figura 15 - Paradigma de manipulação de eventos do POX Fonte: [37]

A título de exemplo, considerando a Figura 15, pode-se considerar a classe de IO *socket* do POX como sendo um editor, a classe *OpenFlow* como sendo a classe evento, e a classe

l2_learning como sendo a classe assinante. O IO socket recebe as mensagens do protocolo *OpenFlow*, e em seguida a classe publicadora lança o evento. Logo depois o subescritor que tem a função para definir o tipo de evento processa a mensagem [35]. O paradigma *publish/subsecribe* é uma framework útil para escalabilidade de distribuição de sistemas [38].

4.1.1 INSTALAÇÃO DO CONTROLADOR

Para instalar o POX é necessário fazer a transferência do projeto do repositório e escolher a versão.

```
$ git clone https://github.com/noxrepo/pox.git
$ cd pox
$ git checkout dart
```

Para executar e testar o POX indica-se o seguinte comando:

```
$ python pox.py log.level --DEBUG
```

No comando o termo *Python* é o interpretador, seguido do nome do *script* e os argumentos que indicam ao módulo de *log* para funcionar em nível *DEBUG*.

4.1.2 UTILIZAÇÃO DO CONTROLADOR

O POX disponibiliza um conjunto de componentes que podem ser ativados através de argumentos de comando na inicialização do controlador. Na lista seguinte apresentam-se as componentes do POX.

A lista das aplicações do POX são:

Hub – comporta-se como simples Hub de rede, instala regras de *flood* em todos os *switches*;
L2_learning – comporta-se como um *Ethernet learning switch*. Aprende o endereço MAC, e combina todos os campos no cabeçalho do pacote instalando os fluxos necessários na rede para cada par de endereços MAC;
L2_multi – semelhante ao *l2_learning*. Utiliza o módulo *OpenFlow.discovery* para conhecer a topologia completa da rede e usa o caminho mais curto entre dois pontos;
L2_pairs – *OpenFlow learning switch* que instala regras para cada par de endereços L2;
L2_flowvisor – É um l2_pairs que funciona como *Flowvisor* para topologias com *loop*;
L2_nx – é uma aplicação *learning switch* que requer extensões Nicira como as encontradas no Open vSwitch;
L3_learning – é como um *switch* de layer 3;
Topo_proactive – instala as regras de encaminhamento em endereços MAC. Esta aplicação tem um comportamento proactivo pré-instalando as regras em cada *switch*;
Spanning_tree – este módulo usa o componente *discovery* para construir uma visão da topologia de rede, e desabilita o *flooding* em portas de *switches* que não estão na árvore pelo seu bit *NO_FLOOD*;

Topology – este módulo armazena todas as entidades de rede *OpenFlow*. Este módulo funciona em conjunto com o módulo **Discovery;**

Discovery – coloca o controlador a enviar mensagens LLDP para a rede com o objetivo de descobrir a conectividade entre os *switches OpenFlow*;

Host_tracker – mantém o rastreamento dos *hosts* na rede, onde eles estão e como estão configurados.

Por exemplo o comando para correr o POX com o módulo de descoberta de topologia ou **Topology** e a aplicação **L2_learning**:

```
$ python pox.py log.level --DEBUG OpenFlow.topology forwarding.l2_le
arning
```

4.1.3 BALANCEAMENTO DE CARGA DO CONTROLADOR

A implementação da distribuição do controlo é feita por software e como linguagem de programação foi utilizado o *Python*. Na Figura 16 apresenta-se o diagrama de atividade do balanceador que demonstra todas as atividades que foram implementadas.

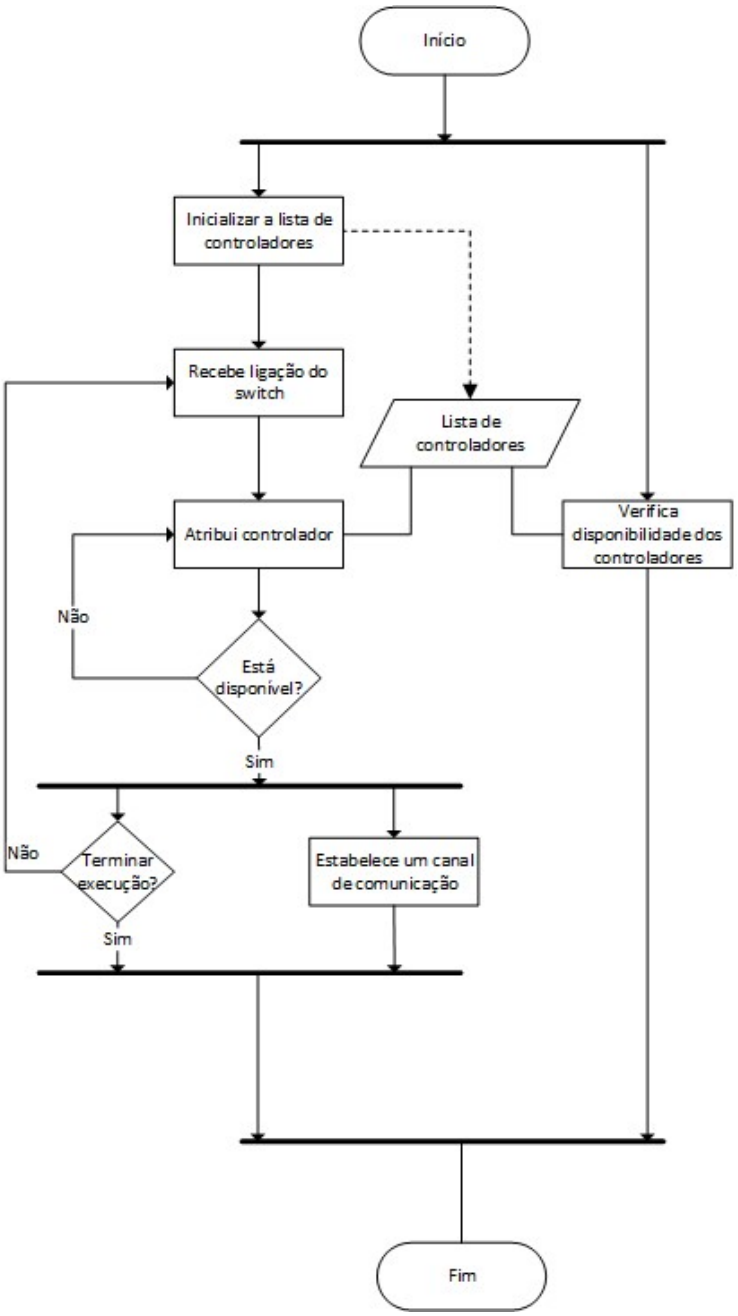


Figura 16 - Diagrama de atividade do balanceador de carga.

O balanceador, quando inicia, recebe como argumentos os endereços IP e os portos de escuta dos controladores colocando-os numa lista. A lista de controladores é uma memória partilhada entre o módulo que verifica a disponibilidade e o módulo que recebe as ligações do

switches. Durante a execução, quando o balanceador recebe uma ligação de um *switch,* este atribui-lhe um controlador disponível. O processo de atribuição do controlador é feito através

```
1.  def check(self, target, status):
2.    incr = 0
3.    while True:
4.      if len(target) <= incr:
5.        sleep(5)
6.        incr = 0
7.      result = check_server(target[incr][0][0], target[incr][0][1])
8.      status[target[incr]]= result
9.      incr+=1
```

do algoritmo *Round-Robin.* Em seguida, o balanceador cria um *pipe* ou canal de comunicação entre o *switch* e o controlador, que se mantém aberto enquanto nenhum dos nós quebrar a ligação. Para paralelizar a execução e maximizar a execução no balanceador é atribuída uma *thread* gestora a cada canal de comunicação. Existe também uma *thread* independente que é lançada na inicialização para verificar o estado dos controladores ciclicamente (intervalos de cinco segundos).

Para finalizar o subcapítulo importa ainda mencionar como funciona o *stream socket,* que é como uma chamada de telefone, ou seja, de um lado está quem invoca a chamada, do outro está quem escuta a chamada, sendo que quem inicia estabelece a conexão através do envio de uma mensagem SYN para quem fica à escuta que, por sua vez envia uma resposta através da mensagem do tipo SYN-ACK, e o inicio da conversação é finalizado com o envio de ACK para quem recebeu a chamada. Posteriormente, encontra-se estabelecida a chamada e quem a invocou inicia a troca de informação, que no caso do balanceador a informação consiste em mensagens do protocolo *OpenFlow*. Após a troca de informação para finalizar, quem iniciou a chamada envia a mensagem do tipo FIN e recebe com resposta um FIN-ACK, a qual devolve novamente uma mensagem do FIN.

4.2 TOPOLOGIA DE REDE

A rede foi implementada no simulador de rede *Mininet* [39]. A implementação de rede descrita em seguida serve de apoio para depurar e produzir testes descritos no Capítulo 5.2 - Testes de Controlador.

4.2.1 INSTALAÇÃO DO *MININET*

O *Mininet* é uma ferramenta *Open-source*, e como tal transfere-se o *source code* do repositório *GitHub* com a ferramenta *git* e comando "git clone *<link>"*, em seguida corre-se o comando "git checkout <versão>" e por fim, executa-se o *bash script* na pasta de raiz

```
$ git clone git://github.com/mininet/mininet
$ cd mininet
$ git tag
$ git checkout -b 2.2.0
$ ~/mininet/util/install.sh -a
```

"*Mininet*/útil/install.sh -a". O argumento "-a" instala todo o software disponibilizado pelo *Mininet*. Para a instalação correta do *Mininet* deverá ser seguida a seguinte sequência de comandos:

Com a instalação concluída, o *Mininet* fica apto para ser usado.

4.2.2 SIMULADOR DE REDE *MININET*

O *Mininet* tem vantagens como o desenvolvimento rápido, baixo custo em equipamento de rede e criação de diversificados cenários de rede. Para criar múltiplos *switches,* controladores e ligações num único *kernel*, o *Mininet* utiliza processos baseados em virtualização através de *Linux namespaces* [40] para conceber virtualização de rede. De modo a tornar o desenvolvimento mais dinâmico o *Mininet* conta ainda com uma API. Como limitações o *Mininet* não pode exceder o limite de CPU e largura de banda num único servidor e não corre em plataformas que não sejam baseadas em Linux.

4.2.3 IMPLEMENTAÇÃO DA REDE

A implementação foi feita por meio da API do *Mininet* onde foi criada a topologia de rede que inclui os elementos tais como os *switches,* os *hosts* e os *links*. Permite ao programador criar topologias com comportamento personalizado.

Para instanciar o objeto que será a rede o código é:

```
1.  net = Mininet( topo=None, build=False)
```

Os parâmetros *topo* indica que não vai passar nenhum objeto e *build* pergunta se pretende construir a rede a partir do argumento *topo*. Depois do objeto de rede instanciado, adicionam-se os *switches*:

```
1.  s1 = net.addSwitch( 's1', listenPort=6634, mac='00:00:00:00:00:01')
2.  s2 = net.addSwitch( 's2', listenPort=6634, mac='00:00:00:00:00:02')
```

E em seguida adicionam-se os *hosts:*

```
1.  hc1 = net.addHost( 'h1', mac='10:00:00:00:00:01', ip='10.1.0.1/24' )
2.  hc2 = net.addHost(' h2', mac='10:00:00:00:00:02', ip='10.1.0.2/24' )
```

Para criar a ligação entre os *switches* e os *hosts*, precisamos de implementar *links*:

```
1.  net.addLink(h1, s1 )
2.  net.addLink(h2, s2 )
3.  net.addLink(s1, s2 )
```

Até este momento apresentou-se o modo como se configurar o plano de dados, e de seguida iremos verificar como se configuram as ligações no plano de controlo. O seguinte código mostra como adicionar o controlador e ligar aos *switches* existentes:

```
1. controller_ip = '192.168.1.12'
2. ctrl = net.addController( 'c0', controller=RemoteController, ip=controller
   _ip, port=6633)
3. s1.start( [ctrl] )
```

4.3 CASOS DE USO

Neste subcapítulo serão apresentados os casos de uso, com o objetivo de auxiliar a identificar em que pontos o projeto focaliza as capacidades para o qual está idealizado. Os casos de usos são uma forma ilustrativa e simples de compreender como um sistema pode interagir com o mundo real. Em seguida, são identificados os casos de uso e consequentemente é detalhado cada um deles:

- Capacidade de resposta a grande quantidade fluxos.
- Tolerância a falhas.

4.3.1 CAPACIDADE DE RESPOSTA A GRANDE QUANTIDADE PEDIDOS

O caso de uso decorre quando existe uma quantidade elevada de pedidos na interface *Southbound* API, o que pode provocar as perdas de desempenho no controlador. No entanto, com recurso a uma análise constante de RAM, CPU e latência, pode-se inferir qual a capacidade e consumo do controlador em prol de adicionar mais instâncias de controlo.

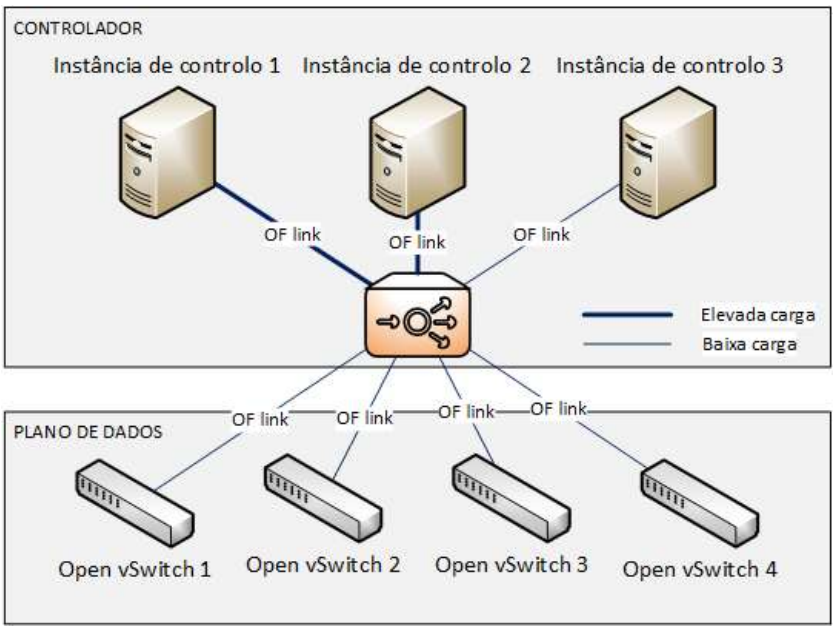


Figura 17 - Caso de uso em carga elevada.

O cenário do caso de uso surge quando existe um número elevado de *switches OpenFlow*, ou quando existe uma procura elevada de pacotes, quer seja de *streams* de vídeo, pequenos equipamentos, ou serviços sobre a rede, e no qual existe a necessidade de manter os níveis de QoS num patamar em que o efeito provocado pela entrada elevada de pacotes seja mínimo.

Um outro caso de uso deste sistema é a capacidade de resposta a uma grande quantidade de fluxos e em que este não seja o *bottleneck* de toda a rede.

4.3.2 TOLERÂNCIA A FALHAS

Este caso de uso decorre quando uma instância de controlo deixa de comunicar com os seus *datapaths* por motivos tais como falha na ligação, ou falha do servidor. Como requisito de caso de uso o controlador distribuído tem de ter no mínimo duas instâncias de controlo a executar.

Para manter o sistema tolerante a falhas é preciso lidar com o estado de funcionamento dos servidores e dos controladores, nomeadamente verificar se estão *online*, verificar o consumo de recursos de *hardware* e conter sistemas redundantes. Como se pode observar na Figura 18, o controlador distribuído usa **n** instâncias, onde ambos os controladores estão ativos e ligados ao balanceador de carga/distribuidor. O caso de uso pode ocorrer, quando por exemplo um controlador deixa de responder, onde todos os *switches* ligados a este controlador terão de ser alocados a outros controladores.

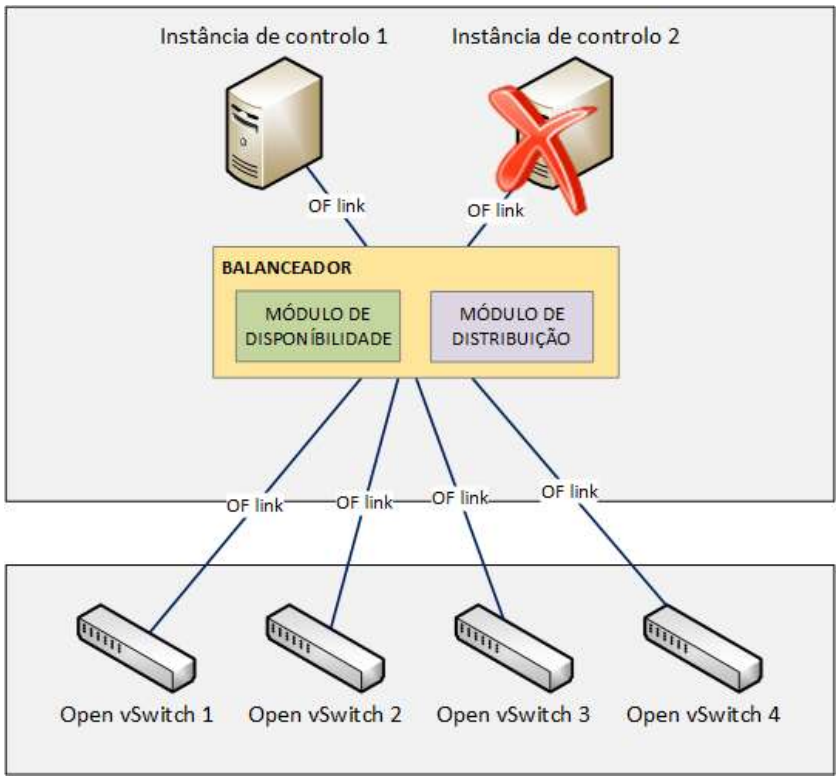


Figura 18 - Caso de uso Tolerância a falhas.

Para tal comportamento, existe uma função que verifica a disponibilidade dos controladores que está incorporada no balanceador, como se pode observar na Figura 18. A avaliação de

estado dos controladores foca-se na disponibilidade relativamente ao *hardware*, avaliando o consumo de CPU e de RAM e na validação do estado da aplicação, este verifica se a aplicação se mantém disponível através da validação de porto do escuta do controlador.

4.4 CONCLUSÃO

Durante esta secção foi apresentada a implementação do balanceador de carga de controladores de redes definidas por *software*, desde a configuração da rede em que foi utilizado o simulador virtualizado de rede *Mininet*. A mesma configuração de rede foi programada através da API que o *Mininet* disponibiliza. Esta API usa o *Python* como linguagem de programação. Quanto ao controlador para cada instância de controlo foram utilizados o POX e o Ryu. Ambos os controladores estão programados em *Python*. A utilização do POX deve-se ao facto de ser uma boa plataforma para os programadores fazerem investigação, e a utilização do controlador centralizado Ryu deve-se à carência de versões de *OpenFlow* do controlador POX que é unicamente compatível com *OpenFlow* versão 1.0. Relativamente ao balanceador de carga, é programado em *Python* 3.6. Ainda nesta secção, também foram identificados os casos de uso do balanceamento de carga dos controladores em redes definidas por *software,* nomeadamente a capacidade de respostas a grande quantidade de pedidos e a tolerância a falhas.

O próximo capítulo dará inicio à secção de análise de resultados, que irá demonstrar testes sobre a implementação descrita na presente secção.

Capítulo 5

5 ANÁLISE DE RESULTADOS

A presente secção serve para recolher e avaliar os resultados do sistema desenvolvido em termos de métricas de avaliação do desempenho de rede, da avaliação do desempenho do controlador distribuído comparativamente a um controlador centralizado, da avaliação do desempenho de um modo geral, em ordem para ter uma ideia da quantidade de fluxos que os controladores podem gerir.

5.1 TESTES DE CONTROLADOR

Os testes separam-se por duas fases, onde cada uma caracteriza-se pela utilização de *hardware* e soluções de controlo de redes definidas por *software* diferentes.

Na primeira fase o hardware utilizado em cada controlador e simulador de rede foi um Raspberry Pi 1 modelo B com um processador ARM11 ARM1176JZF-S core a 700mhz, 512mb de memória RAM, e placa de rede Ethernet de 10/100 megabits. O balanceador de carga usa o *hardware* com o processador Intel (R) i7-2630qm @ 2.0GHz - 2.9GHz, 6GB de memória RAM, e placa de rede cablada de 10/100/1000 megabits.

Na segunda fase, o *hardware* utilizado em cada controlador é composto por um computador com um processador Intel (R) Atom (TM) CPU D2550 @1.86GHz, 4GB de memória RAM, armazenamento SSD com o Samsung SSD 850 120GB e placa de rede Ethernet de 10/100/1000 megabits. Para o balanceamento e simulação de rede foi utilizado o um *hardware* com o processador Intel (R) i7-2630qm @ 2.0GHz - 2.9GHz, 6GB de memória RAM, e placa de rede Ethernet de 10/100/1000 megabits.

Os testes realizados com o *cbench* são de dois tipos: o primeiro corresponde à taxa de transferência do estabelecimento de fluxos; o outro tipo é a latência de estabelecimento de fluxos. No teste de medição da taxa de transferência do estabelecimento de fluxos, os *switches* enviam várias mensagens de *packet_in* para o controlador, assim garantindo que o controlador tem sempre mensagens para processar. No modo de medição de latência do estabelecimento de fluxos, os *switches* enviam uma mensagem *packet_in* para o controlador, esperam pela resposta, e em seguida repetem o processo novamente tão rápido quanto o possível. No fim, o número total de respostas recebidas do controlador pode ser usado para

calcular a média de tempo que o controlador levou para processar a mensagem *packet_in* iniciada no *switch* [35].

Na Tabela 7 identifica todos os parâmetros que a ferramenta *cbench* disponibiliza para realização dos testes.

Tabela 7 - Parâmetro da ferramenta cbench

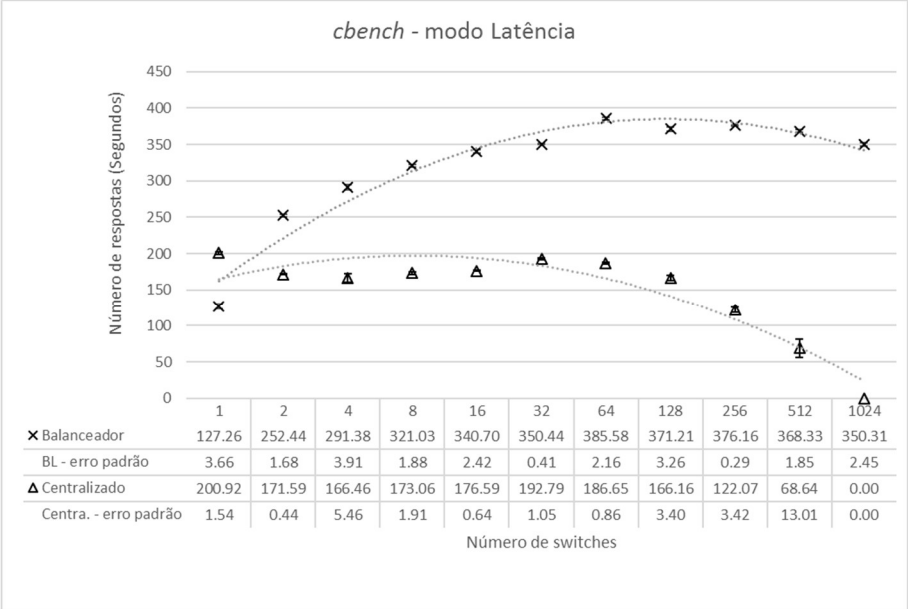
Parâmetro	Descrição	Valor por defeito
-c/--controller	*<str>* hostname do controlador a qual se conecta	*("localhost")*
-d/--debug	ativar depuração	*(off)*
-h/--help	imprimir mensagem	
-l/--loops	*<int>* ciclos por teste	16
-M/--mac-addresses	*<int>* endereço MAC único por switch	100000
-m/--ms-per-test	*<int>* duração do teste em ms	1000
-p/--port	*<int>* porta do controlador	6633
-r/--ranged-test	intervalo de teste de 1 a $n switches	*(off)*
-s/--switches	*<int>*falsos $n switches	16
-t/--throughput	teste de taxa de transferência em alternativa ao teste de latência	
-w/--warmup	*<int>*ciclos de teste a ser desconsiderado no início (warmup)	1
-C/--cooldown	*<int>*ciclos de teste a ser desconsiderado no fim (cooldown)	0
-D/--delay	*<int>*atraso do início do teste depois da mensagem features_reply recebida (em ms)	0
-i/--connect-delay	*<int>*atraso na ligação entre grupo de switches (em ms)	0
-I/--connect-group-size	*<int>*número de switches no grupo de atraso de ligação	1
-L/--learn-dst-macs	envia respostas ARP gratuitas para aprender os MAC de destino antes de testar	*(on)*
-o/--dpid-offset	<int>offset de DPID dos switches	1

5.1.1 TESTE COM O BALANCEADOR DE CARGA E UM CONTROLADOR CENTRALIZADO (1ª FASE)

Este teste é referente à primeira fase no qual foram testados o balanceador de carga de controlo e o controlador centralizado POX [29]. Para desempenhar o teste foi utilizada a ferramenta *cbench*. O comando para executar o *cbench* com um número especifico de *switches* e o número de vezes que repete o teste é o seguinte:

```
$./cbench -c 192.168.1.11 -s 16 -l 8
```

A Figura 19 demostra os resultados do controlador distribuído e o controlador centralizado, o controlador distribuído a partir do grupo 2 *switches* é superior ao número de respostas por segundo comparativamente ao controlador centralizado. O controlador centralizado apresenta um decréscimo de performance a partir do grupo de 32 *switches*, sendo que no grupo de 128 *switches* a sua capacidade de resposta degrade até ao grupo de 1024 *switches* e nesse grupo deixa de responder por completo. Enquanto isso, o controlador distribuído mantém um crescimento ligeiro até ao grupo de 256 *switches*, mostrando ser capaz de responder o dobro em relação a um controlador centralizado. No entanto a partir do grupo 512 *switches* o controlador distribuído inicia a degradação da capacidade de resposta.



	1	2	4	8	16	32	64	128	256	512	1024
× Balanceador	127.26	252.44	291.38	321.03	340.70	350.44	385.58	371.21	376.16	368.33	350.31
BL - erro padrão	3.66	1.68	3.91	1.88	2.42	0.41	2.16	3.26	0.29	1.85	2.45
Δ Centralizado	200.92	171.59	166.46	173.06	176.59	192.79	186.65	166.16	122.07	68.64	0.00
Centra. - erro padrão	1.54	0.44	5.46	1.91	0.64	1.05	0.86	3.40	3.42	13.01	0.00

Figura 19 - Latência dos controladores.

5.1.2 TESTE NUM CENÁRIO DE REDE IMPLEMENTADA (1ª FASE)

O objetivo deste teste é medir a latência entre o controlador centralizado como no controlador distribuído, como se pode ver na Figura 20, no plano de controlo usou-se um controlador distribuído com duas instâncias de controlo. No plano de dados ou topologia de rede temos dois *switches* ligados entre si e um cliente em cada *switch*. Foram colocados dois *switches* para que o balanceador distribuía a carga pelo número de *switches.* Como se pode observar na Figura 20 adicionou-se um *host* a cada *switch.* Estes *hosts* serviram para criar tráfego na rede. A ferramenta para simular tráfego e desempenhar este teste é o *iperf* [41] .

O *Iperf* é uma ferramenta de rede que serve para medir o desempenho da largura de banda dos protocolos TCP e UDP e permite obter resultados para largura de banda disponível, *delay*, *jitter, packet loss* [42] no plano de dados, mas no entanto espera-se gerar uma grande quantidade da pacotes que devem passar pelo plano de controlo.

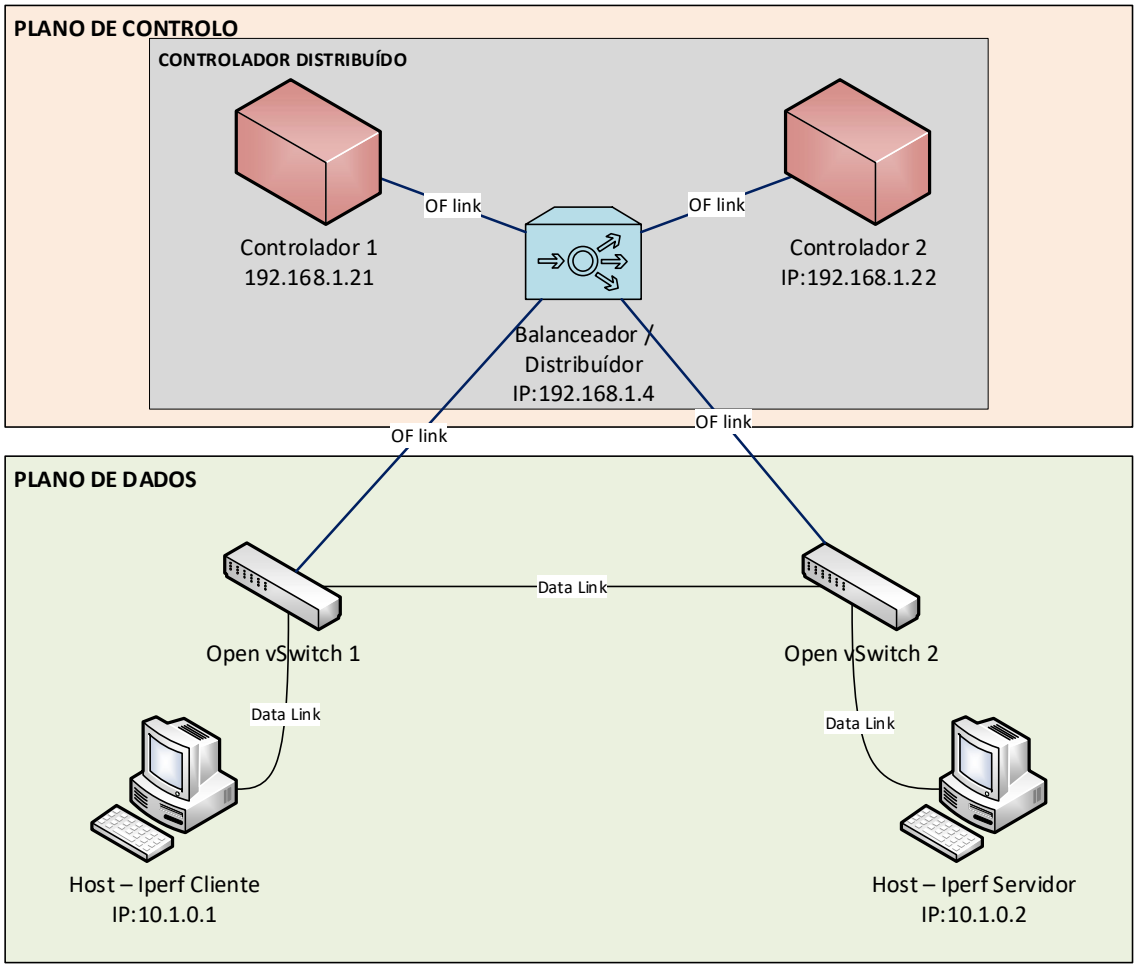


Figura 20 - Cenário de testes com ferramenta iperf.

Como se pode observar na Figura 20, para correr os testes ecolocou-se um *iperf* em modo cliente e outro em modo servidor. Para a executar em modo cliente como se pode constatar com a seguinte comando de invocação da ferramenta:

```
$ iperf -c 10.1.0.1 -t 5 -P 2
```

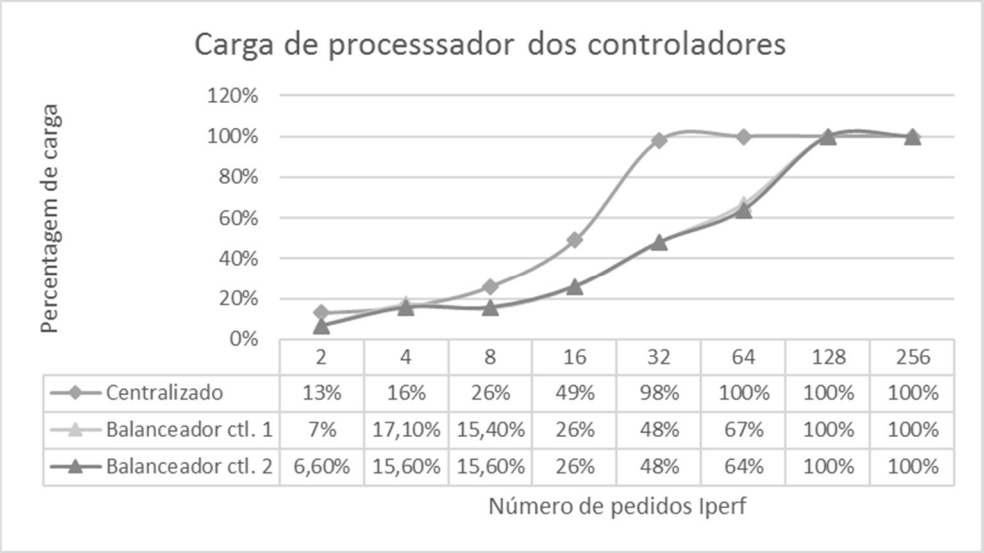
E em modo servidor:

```
$ iperf -c 10.1.0.1 -t 5 -P 2
```

Quando é iniciado o teste, os pacotes começam a ser injetados na rede a partir do cliente *iperf* e deverão seguir para o servidor. Durante o percurso passam por dois switches, onde estes, sempre que recebem um pacote, perguntam ao controlador qual ação devem tomar, (isto porque o controlador está a funcionar em reativo com aplicação "L2_learning"). Cada pacote vai ao controlador e retorna, sendo que a diferença de tempo entre a pergunta e a resposta indica-nos a latência *Round trip* do ciclo da mensagem.

Para efetuar uma leitura de latência passivamente durante a execução da ferramenta *iperf,* efetuou-se a captura dos pacotes na interface que o *Mininet* usa para comunicar com o controlador com a ferramenta *tshark* [43] e em simultâneo monitorizou-se o consumo de RAM e CPU nos controladores. Mais tarde os dados recolhidos com o *tshark* foram analisados com a ferramenta *wireshark* [44]. Os testes foram realizados com o controlador distribuído e o

centralizado, tendo sido usado o POX. A Figura 21 mostra os resultados de consumo de CPU dos controladores, e a Figura 22 o tempo de resposta médio a pedidos simultâneos.



	2	4	8	16	32	64	128	256
Centralizado	13%	16%	26%	49%	98%	100%	100%	100%
Balanceador ctl. 1	7%	17,10%	15,40%	26%	48%	67%	100%	100%
Balanceador ctl. 2	6,60%	15,60%	15,60%	26%	48%	64%	100%	100%

Figura 21 - Carga de CPU.

Observando a Figura 21, o controlador centralizado no grupo 2 tem um consumo superior aos controladores na implementação do balanceador de carga de controlo, mas no grupo 4 o controlador centralizado tem um consumo menor em relação a cada controlador do balanceador de carga. Nos grupos 4, 8 e 16 o controlador centralizado apresenta um consumo superior comparativamente a cada controlador no balanceador de carga. Mas se contabilizarmos os controladores com um único controlador nos grupos 2, 4, 8 e 16 o consumo do balanceador será sempre maior que o controlador centralizado. No grupo dos 64, 128 e 256 pedidos o controlador centralizado regista o consumo de 100%. Com o balanceador os controladores atingem o consumo de 100% a partir do grupo 128 e 256. Os valores anteriores apresentados são meramente representativos uma vez que foi realizada uma a recolha do valor máximo ao longo de execução de cada grupo de número de pedidos, servindo, no entanto, para evidenciar a tendência do aumento da percentagem de carga com o aumento do número de pedidos iperf.

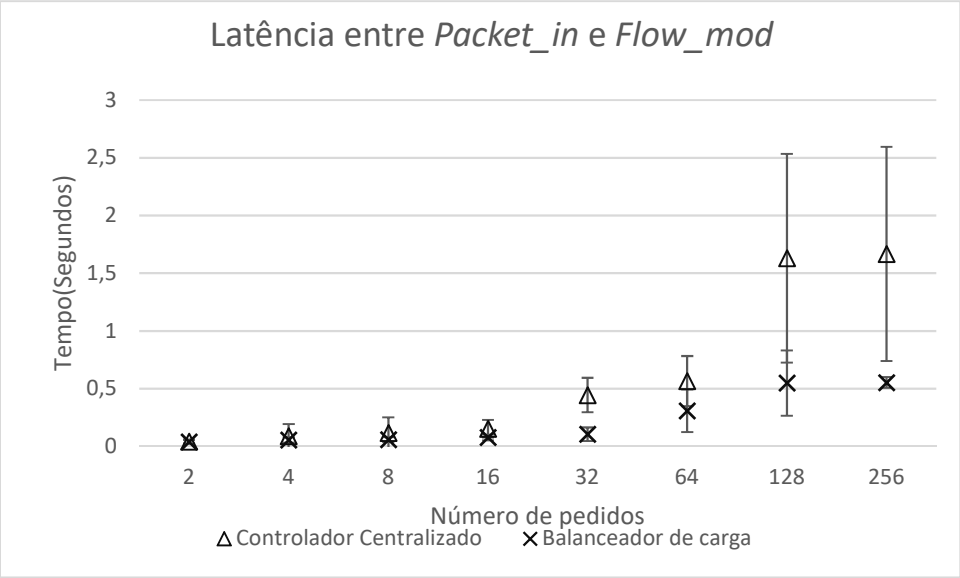


Figura 22 - Latência entre Packet_in e Flow_mod no controlador centralizado.

Quanto aos resultados de latência entre Packet_in e Flow_mod (Figura 22), o controlador centralizado e o balanceador de carga têm um tempo de resposta muito próximo até aos oito pedidos em simultâneo. Como se pode observar na Figura 22 , a partir do grupo 16 pedidos dá-se o crescimento do tempo de resposta do controlador centralizado, e a partir do grupo 64 pedidos simultâneos que o crescimento é mais acentuado, e até ao grupo de 256 pedidos o tempo de resposta por pedido chega a um valor aproximadamente de 3,5 segundos. Quanto balanceador de carga, o comportamento é semelhante ao controlador centralizado, mas mostrando ser diferente no intervalo de tempo entre a mensagem Packet_in e a mensagem Flow_mod.

5.1.3 TESTE COM O BALANCEADOR DE CARGA E UM CONTROLADOR DITRIBUÍDO (2ª FASE)

Nesta secção os testes são referentes à segunda fase, em que se compara o balanceador de carga de controlo implementado e um controlador distribuído ONOS. Neste teste foi realizada a medição da taxa de transferência e da latência com a ferramenta *cbench*.

Em cada controlador, os parâmetros para realizar o teste foram o número de *switches* entre 2 e 256, e 8 ciclos por simulação.

Para executar a pilha de testes foi necessária a instalação do controlador ONOS que tem como requisito um CPU com 2 cores e 2 GB de RAM para executar [36]. Para instalar o ONOS tem de se fazer a transferência do mesmo através da página oficial que neste caso foi a versão 1.11.1. Após a transferência é necessária descompactar para aceder aos ficheiros que permitem executar o controlador. Então para executar o controlador é necessário indicar o seguinte comando:

```
$ onos/bin/onos-service start
```

Agora, o controlador encontra-se em funcionamento, mas, no entanto, este não tem aplicações ativas, o que quer dizer que qualquer tentativa de comunicação por parte dos *switches* será ignorada. Para este teste é necessário ter o controlador em modo reativo e a fazer o encaminhamento de todos os pacotes, sendo que os comandos necessários para tal são:

```
onos> app activate org.onosproject.openflow
onos> app activate org.onosproject.fwd
```

O primeiro comando instrui a ativação da *SouthBound Interface* com o protocolo *OpenFlow,* que por omissão neste controlador é a versão 1.3.1., e o segundo comando instrui o controlador a funcionar como um simples encaminhador de pacotes. Dá-se assim por concluída a tarefa de instalação e iniciação da execução do controlador ONOS. Por outro lado, para executar o balanceador de carga deve-se indicar os endereços IP e portos de escuta de cada instância de controlo no balanceador. A instância de controlo irá executar em modo reativo e fará o encaminhamento de todos os pacotes. O balanceador de carga e controlador corre através da invocação dos seguintes comandos:

```
$ sudo python3.6 loadbalance.py
$ sudo python2.7 pox.py log.level --DEBUG forwarding.l2_learning
```

Para desempenhar o teste de taxa de transferência com o *cbench* executa-se com o seguinte comando:

```
$ cbench -c <ip ONOS> -s <n. º de switches (2 a 256)> -l <n. º de ciclos
por teste (8)> -t
```

Como se pode observar na Figura 23, as observações de *switches* são agrupados de 2, 4, 8 até 256 *switches*. No caso do controlador ONOS verifica-se que em grupos 2, 4 e 8 a média da taxa de transferência é relativamente superior em relação à média dos agrupamentos seguintes (16, 32, 64, 128 e 256).

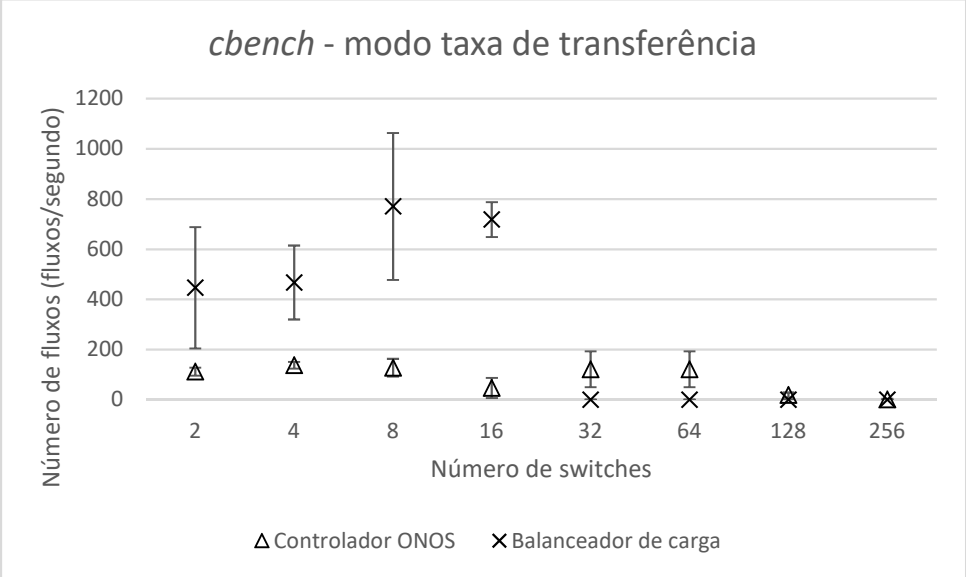


Figura 23 - Taxa de transferência entre o balanceador de carga e o controlador ONOS.

A taxa de transferência do balanceador de carga implementado, nos grupos 2, 4, 8 e 16 manifesta resultados. Por outro lado nos grupos 32, 64, 128 e 256 os resultados são nulos. A falta de fluxos nos grupos 32, 64, 128 deve-se possivelmente pelo facto de o *cbench* por defeito coloca a duração dos testes com 10 milésimos de segundo.

Relativamente à leitura da latência, foi utilizado o *cbench* em modo leitura de latência, com tamanho de teste de 10 ciclos e número de *switches* entre 2 e 256 *switches* por teste. O comando para leitura de latência é o seguinte:

```
$ cbench -c <ip ONOS> -s <n. º de switches (2 a 256)> -l <n. º de ciclos
por teste (8)>
```

Conforme a Figura 24 o controlador ONOS apresenta nos grupos 2, 4 valores superiores aos grupos de 8, 16, 32. Nos grupos 64 e 128 o número de fluxos volta a ser superior a 8, 16 e 32, e no grupo 256 a valor é próximo de zero, ou seja, a resposta do controlador neste grupo é muito baixa.

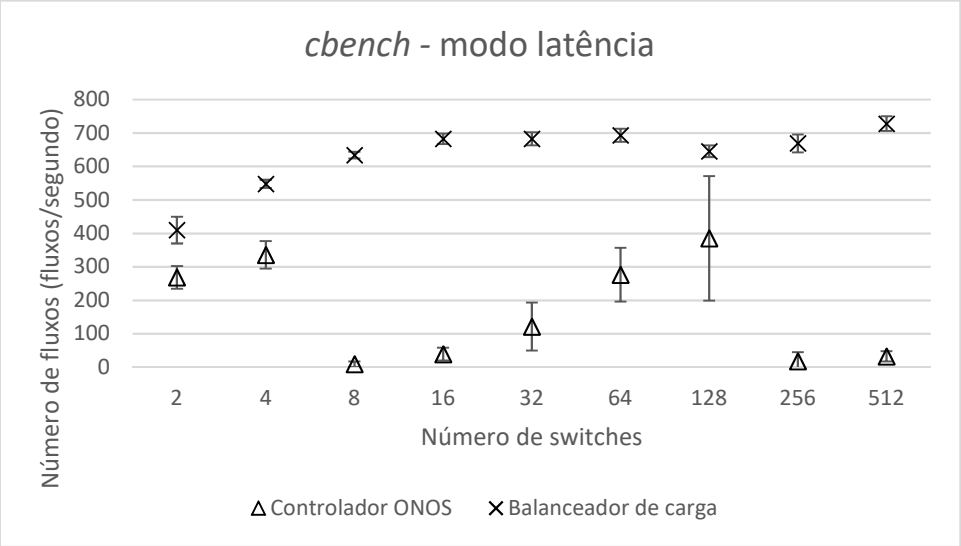


Figura 24 - Latência do controlador ONOS.

O Balanceador de carga de controlo apresenta um crescimento do número de fluxos entre os grupos 2, 4, 8 e 16, nos grupos 32, 64 o crescimento é mais suave caminhado em direção ao valor máximo. No grupo 128 passa a apresentar um decréscimo em relação ao grupo 64. Por outro lado, entre o grupo 256 e 512, é manifestado um crescimento novamente.

5.1.4 TESTE NUM CENÁRIO DE REDE IMPLEMENTADA (2ª FASE)

Este teste usa o mesmo cenário da secção 5.1.2, com a diferença que o plano de controlo ao invés de usar um controlador centralizado para comparação, usa o controlador distribuído ONOS [20].

Para configurar o controlador ONOS é necessária, além dos parâmetros de configuração já demonstrada na secção 5.1.3 o controlador ONOS, a configuração para executar com múltiplas instâncias de controlo, o que se faz através do comando:

```
$ onos/bin/onos-form-cluster <IP da máquina 1> <IP da máquina N>
```

Para obter os pacotes foi utilizada a ferramenta *tshark e* o valor de latência foi medido através da diferença de *timestamp* entre a mensagem packet_in e a mensagem flow_mod. Como se pode observar na Figura 25, demonstra a latência entre o envio da mensagem do tipo packet_in do *switch* para o controlador ONOS e a mensagem do tipo flow_mod do controlador para o *switch*. Como se pode observar o balanceador de carga nos grupos 2, 4, 8, 16 e 64 apresenta valores inferiores a 2 segundos de latência. No grupo 32 apresenta um valor ligeiramente superior a 2 segundos. No grupo 128 apresenta um valor superior à média dos restantes e como um erro padrão elevado.

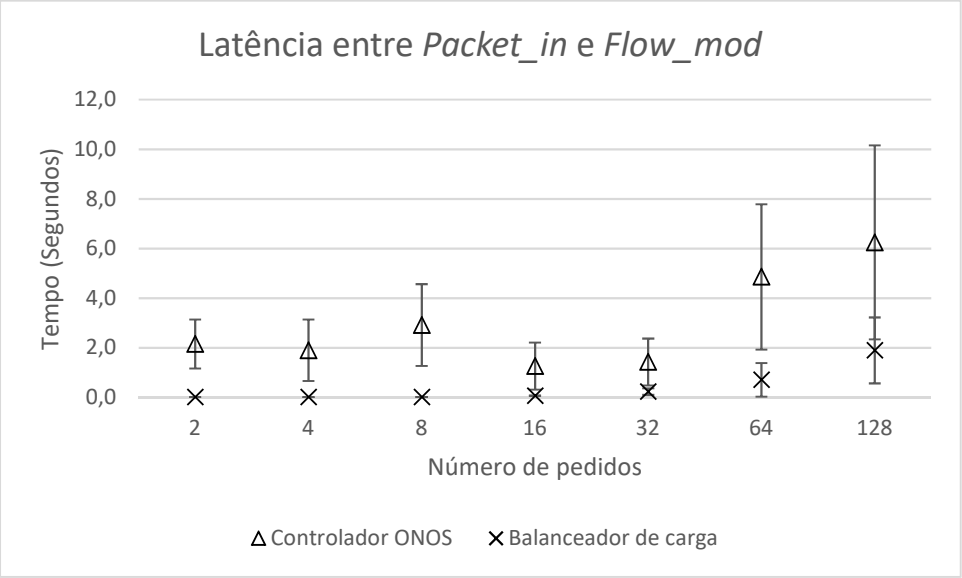


Figura 25 - Latência entre as mensagens Packet_in e Flow_mod (ONOS).

Por outro lado, o controlador ONOS apresenta valores em todos os grupos superiores aos grupos de *switches* demonstrados no teste ao balanceador de carga. Também pode-se observar que no grupo 32, o valor é inferior aos grupos vizinhos.

5.2 CONCLUSÃO

Neste capítulo realizou-se uma revisão do comportamento apresentado na secção anterior.

Na 2ª fase de testes, verifica-se que o balanceador de carga nos testes desempenhados com o *cbench* sobre a taxa de transferência fica limitado a partir do grupo de 32 *switches,* o que não acontece com o controlador ONOS, relativamente aos testes com o *cbench* em modo leitura de latência, o balanceador de carga apresenta mais fluxos por segundo que o controlador ONOS. Nos testes realizados num cenário implementado, o controlador ONOS demonstra-se inferior ao balanceador de carga, o controlador ONOS não foi incapaz de fazer o balanceamento automático dos *switches* pelas instâncias de controlo que tenha disponível. Contudo o lógico seria os resultados demonstrarem um melhor desempenho do controlador ONOS, algo que parece não se suceder e uma das possíveis causas será pelo facto de ONOS não balancear automaticamente os *switches* a si ligados. Desta forma, verifica-se o contributo de uma solução suportada por um balanceador de carga.

Capítulo 6

6 CONCLUSÃO

O SDN surge como alternativa às redes tradicionais que estão em rumo ao limite de uso em grande parte por causa do surgimento de novos medias sociais, equipamentos móveis, *cloud computing* entre outros. A computação e armazenamento tem recebido inovações de virtualização e automação, contudo estão limitados pela rede [41]. As *Software Defined Networks* têm o potencial para revolucionar o modo de fornecer flexibilidade de controlo de rede e já apresenta várias soluções para controladores de rede distribuídos como o ONOS e o *OpenDaylight*.

Esta abordagem distingue-se em relação às outras pela utilização de controladores que são centralizados por hardware, e agora podem estar separados tanto por hardware como por software, e pelo facto que o protocolo *OpenFlow* até à versão 1.2 não suporta que os *switches* se liguem a múltiplos controladores de rede [28] tal facto que nesta abordagem não tem isso como uma limitação e pela utilização de um paradigma Master-Master que apresenta benefícios como no caso de falha de um elemento, os restantes elementos estão disponíveis prontamente.

Como limitação apresenta a falta de consistência entre as instâncias de controlo. Pela utilização de um sistema de replicação à priori (balanceador) que incrementa atraso na comunicação. Continua a ter um ponto único de falha, que é o sistema de balanceamento de carga que não é menos importante, pois se o sistema de balanceamento falha toda a comunicação entre o *switch* e o controlador fica interrompida. Outra limitação é o modelo Master-Master apresenta um consumo maior de recursos.

6.1 TRABALHO FUTURO

Como motivo de trabalho futuro seria interessante reavaliar o controlador a utilizar na solução implementada, pois o POX apresenta limitação na versão de protocolo *OpenFlow*, sendo somente compatível com a versão 1.0 do mesmo. Além disso o POX não tem uma API de *Northbound* nativa. É também possível melhorar a implementação no ponto de recuperação a falhas do controlador, sendo que até ao momento está implementada a componente no balanceador, e em falta está uma componente em cada controlador que envie os dados relativos ao consumo de RAM, CPU e largura de banda disponível. Um outro módulo

para reiniciar, adicionar e remover as máquinas, (e aqui sugeria-se usar a ferramenta Docker e API compatível em *Python* para o módulo de recuperação a falhas presente no balanceador), serviria para melhorar a arquitetura. Também se poderia investigar um sistema de centralização de dados relativos da topologia como os elementos *switch*, *hosts* e ligações, aplicações de rede disponível em cada controlador, de forma a poder aplicar mecanismos de registo da topologia de forma independente dos controladores. Com isto pretende-se reduzir o tempo de recuperação do controlador, a nível da topologia e das aplicações do controlador, em caso de falha. Outro trabalho seria testar amplamente o controlador distribuído com trafego em vários por exemplo simulador de *stream* de vídeo, VoIP, a fim de obter testes ainda mais próximos de cenários reais.

Apêndices

APÊNDICE A. CÓDIGO TOPOLOGIA DE REDE

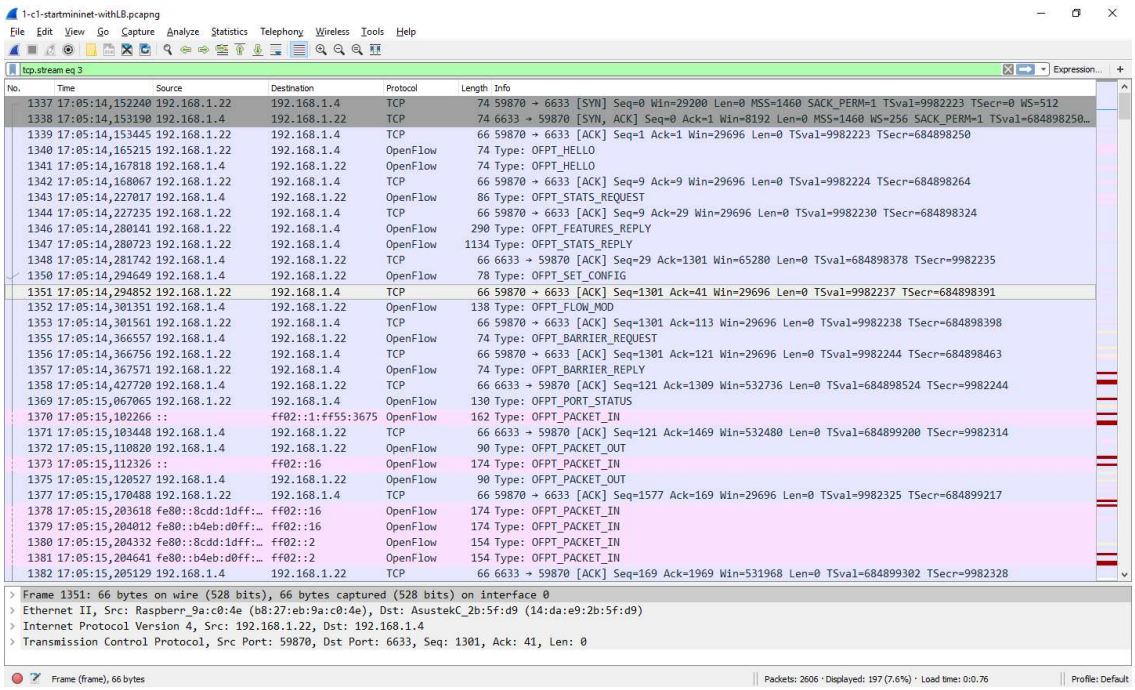
```
1.  #!/usr/bin/Python
2.
3.  from Mininet.net import Mininet
4.  from Mininet.node import Controller, RemoteController
5.  from Mininet.cli import CLI
6.  from Mininet.log import setLogLevel, info
7.  from Mininet.nodelib import LinuxBridge
8.
9.  def myNet():
10.
11.
12.     #IP controller
13.     CT_CONTROLLER_IP='192.168.1.12'
14.     #DST_CONTROLLER_IP='192.168.1.4'
15.
16.     net = Mininet( topo=None, build=False)
17.
18.     # Create nodes
19.     print "*** Creating nodes"
20.     h1 = net.addHost( 'h1', mac='01:00:00:00:01:00', ip='10.1.0.1/24' )
21.     h2 = net.addHost( 'h2', mac='01:00:00:00:02:00', ip='10.1.0.2/24' )
22.
23.     # Create switches
24.     print "*** Creating switches"
25.     s1 = net.addSwitch( 's1', listenPort=6634, mac='00:00:00:00:00:01')
26.     s2 = net.addSwitch( 's2', listenPort=6634, mac='00:00:00:00:00:02')
27.
28.     print "*** Creating links"
29.     #link between controller and switches
30.     net.addLink(h1, s1 )
31.     net.addLink(h2, s2 )
32.     net.addLink(s1, s2 )
33.
34.     # Add Controllers
35.     print "*** Adding controllers"
36.     ctrl = net.addController( 'c0', controller=RemoteController, ip=CT_CONTROL
    LER_IP, port=6633)
37.
38.     net.build()
39.
40.     # Connect each switch to a different controller
41.     s1.start( [ctrl] )
42.     s2.start( [ctrl] )
43.
44.     s1.cmdPrint('ovs-vsctl show')
45.     s2.cmdPrint('ovs-vsctl show')
46.
47.     CLI( net )
48.     net.stop()
49.
50. if __name__ == '__main__':
51.     setLogLevel( 'info' )
52.     myNet()
```

APÊNDICE B. TAXA DE TRANSFERÊNCIA E LATÊNCIA (2ª FASE)

Taxa de transferência								
Controlador ONOS								
	2	4	8	16	32	64	128	256
min	75,71	97,32	0	0	15,75	15,75	0	0
média	112,28	137,75	128,2	47,5	121,66	121,66	18,83	1,85
max	169,57	163,8	207,3	223,89	328,75	328,75	47,94	8,98
stdev	26,15	21,98	56,74	64,61	115,48	115,48	16,84	3,22
Balanceador de controlo								
	2	4	8	16	32	64	128	256
min	0	0	0	444,98	0	0	0	0
média	446,42	467,38	770,04	718,03	0	0	0	0
max	839,55	627,64	1446,21	784,51	0	0	0	0
stdev	390,05	237,54	471,82	112,24	0	0	0	0

Latência									
Controlador ONOS									
	2	4	8	16	32	64	128	256	512
min	170,00	229,00	0,00	0,00	15,75	107,00	23,00	0,00	0,00
média	268,56	336,00	9,14	39,57	121,66	276,57	385,28	17,86	32,71
max	332,00	403,00	32,00	104,00	328,75	512,00	987,99	125,00	74,00
stdev	54,7	66,17	13,3	30,68	115,48	129,45	299,6	43,74	25,02
Balanceador de controlo									
	2	4	8	16	32	64	128	256	512
min	285,00	521,00	601,00	646,00	618,00	627,00	581,99	599,99	673,99
média	410,00	547,86	633,71	682,86	682,86	693,43	645,71	668,85	727,98
max	478,00	585,00	655,00	728,00	731,00	732,00	671,00	704,00	776,98
stdev	64,58	19,87	16,06	25,18	31,35	31,75	28,25	43,29	35,43

APÊNDICE C. SEQUÊNCIA DE TROCAS DE MENSAGENS (WIRESHARK)



BIBLIOGRAFIA

[1] K. S. Sahoo, S. Mohanty, M. Tiwary, B. K. Mishra, and B. Sahoo, "A Comprehensive Tutorial on Software Defined Network," in *Proceedings of the International Conference on Advances in Information Communication Technology & Computing - AICTC '16*, 2016, pp. 1–6.

[2] F. M. V. Ramos, D. Kreutz, and P. Veríssimo, "Software-defined networks: On the road to the softwarization of networking," *Cut. IT J.*, vol. 28, no. 5, pp. 6–13, 2015.

[3] R. Masoudi and A. Ghaffari, "Software defined networks: A survey," *J. Netw. Comput. Appl.*, vol. 67, pp. 1–25, 2016.

[4] "On Scalability of Software-Defined Networking," *IEEE Commun. Mag.*, vol. 51, no. 2, pp. 136–141, 2013.

[5] D. Stanikova *et al.*, "Melanocortin-4 receptor gene mutations in obese slovak children," *Physiol. Res.*, vol. 64, no. 6, pp. 883–890, 2015.

[6] "What SDN is and where it's going | Network World." [Online]. Available: https://www.networkworld.com/article/3209131/lan-wan/what-sdn-is-and-where-its-going.html. [Accessed: 18-Sep-2017].

[7] IPknowledge, *Traditional vs Software Defined Networking*. .

[8] Y. Zhou *et al.*, "A load balancing strategy of SDN controller based on distributed decision," *Proc. - 2014 IEEE 13th Int. Conf. Trust. Secur. Priv. Comput. Commun. Trust. 2014*, no. 978, pp. 851–856, 2015.

[9] "NOX - repo." [Online]. Available: https://github.com/noxrepo/nox. [Accessed: 04-Sep-2017].

[10] M. P. Fernandez, "Comparing OpenFlow controller paradigms scalability: Reactive and proactive," *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, pp. 1009–1016, 2013.

[11] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," *Proc. 2010 internet Netw. …*, pp. 3–3, 2010.

[12] J. Stribling *et al.*, "Flexible , Wide-Area Storage for Distributed Systems with WheelFS," *Proc. 6th {USENIX} {S}ymposium {N}etworked {S}ystems {D}esign {I}mplementation ({NSDI} '09)*, pp. 43–58, 2009.

[13] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," *OSDI, Oct*, pp. 1--6, 2010.

[14] "ON.Lab Delivers Software for New Open Source SDN Network Operating System - ONOS™." [Online]. Available: http://www.prnewswire.com/news-releases/onlab-delivers-software-for-new-open-source-sdn-network-operating-system--onos-300004797.html. [Accessed: 04-Jan-2017].

[15] P. Berde *et al.*, "ONOS," in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, 2014, pp. 1–6.

[16] P. Berde *et al.*, "ONOS," in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, 2014, pp. 1–6.

[17] S. Raynovich, "ONOS Beefs up Interfaces and Scale With Version 1.2," 2015. [Online]. Available: https://www.sdxcentral.com/articles/news/onos-beefs-up-interfaces-and-scale-with-version-1-2/2015/06/.

[18] "OpenDaylight." [Online]. Available: https://www.opendaylight.org/. [Accessed: 07-Sep-2017].

[19] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "SDN controllers: A comparative study," *Proc. 18th Mediterr. Electrotech. Conf. Intell. Effic. Technol. Serv. Citizen, MELECON 2016*, no. April, 2016.

[20] "ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out." [Online]. Available: http://onosproject.org/. [Accessed: 07-Sep-2017].

[21] P. Berde *et al.*, "Facilitation of the OpenDaylight Architecture," *Proc. third Work. Hot Top. Softw. Defin. Netw. - HotSDN '14*, pp. 1–6, 2014.

[22] "SDN Series Part Six: OpenDaylight, the Most Documented Controller - The New Stack." [Online]. Available: http://thenewstack.io/sdn-series-part-vi-opendaylight/. [Accessed: 04-Jan-2017].

[23] "ANR DISCO | Main / HomePage." [Online]. Available: http://anr-disco.ens-lyon.fr/. [Accessed: 05-Jan-2017].

[24] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed multi-domain SDN controllers," in *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014, pp. 1–4.

[25] "ZeroSDN | Distributed SDN Controller." [Online]. Available: http://zerosdn.github.io/. [Accessed: 07-Sep-2017].

[26] F. Dürr, T. Kohler, J. Grunert, and A. Kutzleb, "ZeroSDN: A Message Bus for Flexible and Light-weight Network Control Distribution in SDN," vol. 1, 2016.

[27] C. Hao, Chang, and Y.-D. Lin, "OpenFlow Version Roadmap," 2015. [Online]. Available: http://speed.cis.nctu.edu.tw/~ydlin/miscpub/indep_frank.pdf. [Accessed: 16-Nov-2017].

[28] P. Bosshart, D. Daly, and M. Izzard, "Programming Protocol-Independent Packet Processors," *arXiv Prepr. arXiv*, vol. 44, no. 3, pp. 0–6, 2013.

[29] "Clarifying the differences between P4 and OpenFlow | P4." [Online]. Available: https://p4.org/p4/clarifying-the-differences-between-p4-and-openflow/. [Accessed: 16-Nov-2017].

[30] B. Heller, "OpenFlow Switch Specification 1.0.0," *Current*, vol. 0, pp. 1–36, 2009.

[31] "Open vSwitch." [Online]. Available: http://openvswitch.org/. [Accessed: 01-Sep-2017].

[32] "OpenFlow Switch Specification," 2009.

[33] E. S. Spalla *et al.*, "Resilient Strategies to SDN: An Approach Focused on Actively Replicated Controllers," *Proc. - 33rd Brazilian Symp. Comput. Networks Distrib. Syst. SBRC 2015*, pp. 246–259, 2015.

[34] "Flowgrammable." [Online]. Available: http://flowgrammable.org/sdn/openflow/message-layer/hello/.

[35] "POX Wiki - Open Networking Lab - Confluence." [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki. [Accessed: 30-Aug-2017].

[36] "Welcome to Python.org." [Online]. Available: https://www.python.org/. [Accessed: 26-Sep-2017].

[37] "The Event System in POX." [Online]. Available: http://xuyansen.work/the-event-system-in-pox/. [Accessed: 31-Aug-2017].

[38] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[39] “Mininet: An Instant Virtual Network on your Laptop (or other PC) - Mininet.” [Online]. Available: http://mininet.org/. [Accessed: 05-Sep-2017].

[40] “Introducing Linux Network Namespaces - Scott’s Weblog - The weblog of an IT pro specializing in cloud computing, virtualization, and networking, all with an open source view.” [Online]. Available: https://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/. [Accessed: 13-Sep-2017].

[41] “iPerf - The TCP, UDP and SCTP network bandwidth measurement tool.” [Online]. Available: https://iperf.fr/. [Accessed: 06-Sep-2017].

[42] “What is iPerf and how is it used?” [Online]. Available: http://searchnetworking.techtarget.com/answer/What-is-iPerf-and-how-is-it-used. [Accessed: 06-Sep-2017].

[43] “tshark - The Wireshark Network Analyzer 2.4.1.” [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html. [Accessed: 28-Sep-2017].

[44] Wireshark Foundation, “Wireshark · Go deep.” p. 27.05.2010, 2010.