Alexandre Miguel dos Santos Pinote

Nº 36917

Encaminhamento IP com OpenFlow

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : José Legatheaux Martins, Professor,
Universidade Nova de Lisboa

Júri:

Presidente: Doutor Pedro Manuel Corrêa Calvente Barahona

Vogais: Doutor Fernando Manuel Valente Ramos
Doutor José Augusto Legatheaux Martins

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2013

Encaminhamento IP com OpenFlow

Aos meus pais

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador, José Legatheaux Martins, pela orientação dada, assim como os conselhos importantíssimos e também pelo seu entusiasmo e dedicação ao longo da realização desta dissertação. Queria agradecer a Luís Anjos e Samuel Neves da Divisão de Informática da FCT, pelos logs disponibilizados para a realização das avaliações, que se revelaram imprescindíveis para a realização da dissertação. Também queria agradecer a todos os professores que tive neste percurso académico, pelas competências dadas para conseguir realizar este trabalho.

Aos meus pais, Maria Alice e João, pela paciência, amor, compreensão e incentivo, mostrados ao longo de todos estes anos, assim como à minha avó Isaura.

Queria agradecer ao meus amigos, Pedro Tiple, Bruno Ferreira, Filipe Falcão, Jorge Costa e Bernardo Bogarim pelo seu apoio, ideias e comentários, assim como à minha namorada Joana Miranda. Foi um privilégio tê-los ao meu lado. Queria agradecer o apadrinhamento dado pelo David Pinto, que foi importante no início da minha vida académica. A toda a minha família e em particular aos meus primos, Manuel Pinheiro e Pedro Antunes, e padrinhos.

Não posso deixar de agradecer aos demais amigos e colegas deste percurso, pelas noites de estudo, pelas noites de lazer, pelas surfadas e pela boa companhia.

A todos, um sincero obrigado.

Resumo

Nos últimos anos apareceu uma proposta para tornar mais flexível o controlo e evolução das redes, através de uma aproximação que se veio a designar *Software Defined Networking* (SDN). Uma das vantagens advém do facto de existir um servidor logicamente centralizado que controla um conjunto de *switches*.

Esta dissertação apresenta um trabalho que consistiu em introduzir numa rede controlada por um controlador SDN diversas novas funcionalidades com o objectivo de optimizar, tanto quanto possível, o funcionamento de uma rede de *switches* Ethernet, de tal forma que seja mais realista geri-la como uma única *subnet* IP, mesmo numa rede com grande número de utilizadores.

Estas optimizações reduzem a utilização de difusão pelo protocolo ARP, encaminham o tráfego *unicasting* usando o caminho mais curto na rede (funcionalidade já implementada geralmente nos controladores) e encaminham o tráfego *multicasting* usando uma árvore de difusão óptima por grupo IP e por emissor. Para a implementação desta última funcionalidade foi também necessário introduzir uma implementação do protocolo IGMP. O tratamento da segurança, que esteve na origem da aproximação SDN, não foi objecto deste trabalho.

Palavras-chave: Openflow, Software Defined Networks, IP Unicast, IP Multicast

Abstract

Recently, a proposal has appeared to make more flexible the control and the evolution of networks through an approach called Software Defined Networking (SDN). One of the advantages is the fact that there is a logically centralized server that controls a set of switches.

This thesis presents a work that introduce into a network controlled by a controller SDN, several new features with the goal of optimizing, as possible, the operation of a network Ethernet switches, to be more realistic to manage it as a single IP subnet, even in a network with large number of users.

These optimisations reduce the use of broadcasts by the ARP protocol, routing unicasting traffic using the shortest path (functionality already implemented in some controllers) and routing multicasting traffic using an optimal multicasting tree by group IP and by sender. To implement this last feature was also necessary to introduce a implementation of IGMP protocol. The security issues, which led to the SDN approach, was not subject of this work.

Keywords: Openflow, Software Defined Networks, IP Unicast, IP Multicast

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

1.1 Motivação

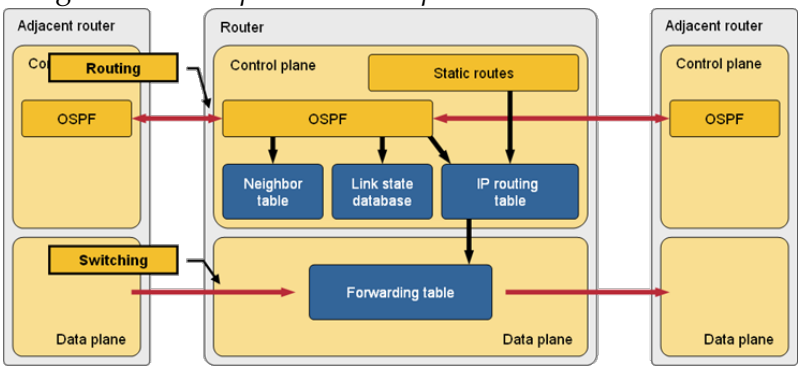
Hoje em dia as redes de computadores estão presentes em todo o lado. As mesmas são compostas por equipamentos e canais de comunicação. Os equipamentos, são normalmente nomeados em português por comutadores e encaminhadores, e em inglês por *switches* ou *routers* [1]. Em redes de média ou grande escala, estes equipamentos têm um elevado custo e a sua parametrização é efectuada por pessoas especializadas. A necessidade de contratar estes profissionais, está no facto de a parametrização dos equipamentos ser uma tarefa bastante complexa e especializada. Uma vez parametrizados, os nós das redes coordenam-se entre si para manter a rede a funcionar correctamente. Existem vários protocolos normalizados para suportar essa coordenação porque os equipamentos podem ter fabricantes distintos. Consequentemente, esses protocolos existem em número reduzido e evoluem muito lentamente. Novas ideias e soluções surgem, mas não conseguem ser testadas facilmente e as que resistem às várias barreiras impostas, levam bastantes anos até serem normalizadas. Em síntese, actualmente nas redes usam-se equipamentos caros, difíceis de parametrizar e com software que não permite facilmente implementar novas ideias de gestão de rede. A aproximação *Software Defined Network* (SDN) é uma tentativa de mudar este estado de coisas.

Nas redes tradicionais os nós estão organizados em dois conjuntos de funcionalidades, designados por *data plane* e *control plane*, como se observa na Figura 1.1. O *data plane* é responsável pela função de *Forwarding*, ou seja, redireccionar os pacotes entre interfaces.

[1] Utilizaremos os termos portugueses "nó" ou "comutador" mas também os termos ingleses tradicionais em itálico por ser muito comum a sua utilização no debate e escrita na língua portuguesa.

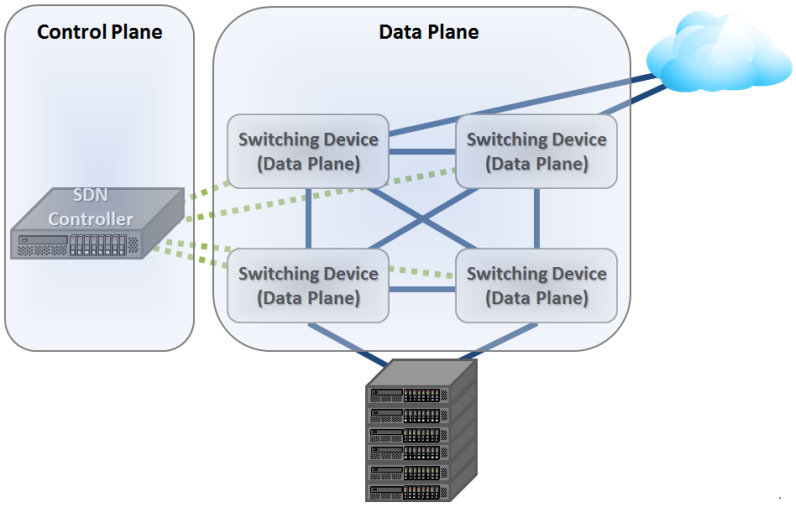
O *data plane* exerce a sua função com base em tabelas de encaminhamento que permitem diferenciar, através dos cabeçalhos dos pacotes, como proceder com cada pacote, ou seja, para que interface deve ser encaminhado, se deve ser destruído, etc. O *control plane* abrange as funcionalidades e os protocolos de coordenação da rede. Este também é responsável por actualizar as tabelas de encaminhamento.

Figura 1.1: *Data plane* e *control plane* numa rede tradicional



Na aproximação SDN surge um novo paradigma em que os equipamentos de comutação apenas lidam com o *data plane* e as funcionalidades do *control plane* são atribuídas a um servidor à parte, como se observa na Figura 1.2. Com um único servidor de controlo é possível controlar mais que um nó.

Figura 1.2: *Data plane* e *control plane* numa aproximação SDN



Através da utilização da aproximação SDN é possível prever um conjunto de melhoramentos face às redes tradicionais:

- equipamentos de rede mais baratos e simples, sem necessidade de executarem um *control plane* complexo;
- gestão da rede mais simples porque é baseada numa visão e controlo logicamente centralizados;

- *control plane* baseado em software a executar em plataformas aplicacionais comuns (Linux / Windows / Mac OS / Hardware X86)

- maior flexibilidade no teste e evolução de novas soluções de *control plane*, pois é mais fácil mudar o software executado pelos controladores do que esperar pela normalização de novos protocolos e que estes sejam implementados pelos fabricantes.

1.2 Objectivos

Analisada a literatura sobre a aproximação SDN verificou-se que o IP Multicasting parece nunca ser disponibilizado, pelo menos de forma optimizada. Assim sendo, o objectivo da dissertação foi explorar mais esta funcionalidade de uma rede TCP/IP convencional, tornando-a utilizável numa aproximação SDN. Este objectivo pretendeu também comprovar que os algoritmos de controlo de rede implementados num controlador, que utiliza a aproximação SDN, são mais simples que a sua versão distribuída. Na primeira fase, foi feito um teste que incidiu sobre perceber a programação e as propriedades do *control plane* logicamente centralizado e do funcionamento e programação de um controlador. Este teste teve como ambiente alvo uma rede empresarial de grande escala e foram utilizados simuladores e controladores do domínio público. Esta fase teve como objectivo perceber o realismo da aproximação SDN para implementar uma rede baseada em TCP/IP sem necessidade de modificações do software dos computadores.

À primeira fase seguiu-se então a fase de optimização do protocolo Address Resolution Protocol (ARP), tendo como objectivo reduzir os *broadcasts* causados pelos ARP Requests. Depois, seguiu-se uma fase de desenho e implementação de IP Multicasting numa aproximação SDN. Esta fase, numa primeira aproximação, baseou-se numa implementação baseada em *broadcasting*. A esta primeira implementação seguiu-se a análise, implementação e teste de uma solução mais optimizada com base em árvores específicas para grupos e para emissores como no protocolo Protocol Independent Multicast (PIM) na sua versão Sparse Mode (PIM-SM).

1.3 Contribuições do trabalho

A maioria dos controladores que irão ser apresentados já oferecem módulos que permitem, através de OpenFlow, construir uma rede simples de nível L2 convencional. Geralmente fornecem igualmente módulos de realização de controlo de acessos (*firewalling*) e de encaminhamento *unicast* usando o caminho mais curto numa configuração arbitrária de *switches* (isto é, em malha e com ciclos). No entanto, tanto quanto se conseguiu apurar, não existem no domínio público implementações de Internet Group Management Protocol (IGMP) nem de encaminhamento IP Multicasting optimizado através de árvores de difusão (como as computadas por PIM por exemplo). Um único artigo faz referência a uma implementação sem apresentar informação concreta sobre a solução [NHS12].

Esta dissertação apresenta um trabalho que consistiu em introduzir no controlador Floodlight um novo módulo e um conjunto de modificações noutros módulos com o objectivo de optimizar, tanto quanto possível, através de OpenFlow, o funcionamento de uma rede de *switches* Ethernet, de tal forma que seja mais realista trabalhar com uma única *subnet* IP, mesmo numa rede de grande dimensão[2].

Estas optimizações reduzem a utilização de difusão pelo protocolo ARP, encaminham o tráfego *unicast* usando o caminho mais curto na rede (funcionalidade já implementada no Floodlight) e implementam o tráfego *multicast* usando uma árvore de difusão óptima por grupo IP e por emissor. Para a implementação desta última funcionalidade foi também necessário introduzir uma implementação do protocolo IGMP. As novas funcionalidades foram desenvolvidas em Java para o Floodlight, e foram testadas usando o simulador de redes *OpenFlow* Mininet [LHM10].

Em síntese, as contribuições desenvolvidas são:

- Análise crítica da gestão de uma rede IP baseada na aproximação SDN;
- Implementação optimizada do protocolo ARP;
- Implementação optimizada de IP Multicasting baseada numa aproximação SDN.

1.4 Organização da dissertação

Nesta dissertação será apresentado o desenvolvimento do estado de arte no capítulo 2. Nesse capítulo é descrita a história da aproximação SDN, seguindo-se a definição do *OpenFlow*, onde também são apresentados alguns controladores, um dos seus simuladores de redes e linguagens de alto nível para programação dos simuladores. No final do capítulo 2 é descrito o funcionamento do IP Unicast e do IP Multicast em redes *switched*. Neste capítulo serão melhor apresentados os objectivos da dissertação. No capítulo seguinte será apresentado o Floodlight, o controlador escolhido para desenvolver o trabalho desta dissertação. Depois, no capítulo 4 é introduzida a implementação onde se apresenta em pormenor a solução desenvolvida. No capítulo 5 é descrita a avaliação feita ao trabalho e, finalmente, nos últimos dois capítulos são apresentados o trabalho futuro e as conclusões.

[2]Os requisitos de segurança ultrapassam o âmbito deste trabalho mas poderiam ser implementados com base no módulo de *firewalling* disponibilizado pelo Floodlight.

2

Trabalho Relacionado

2.1 Software Defined Networks

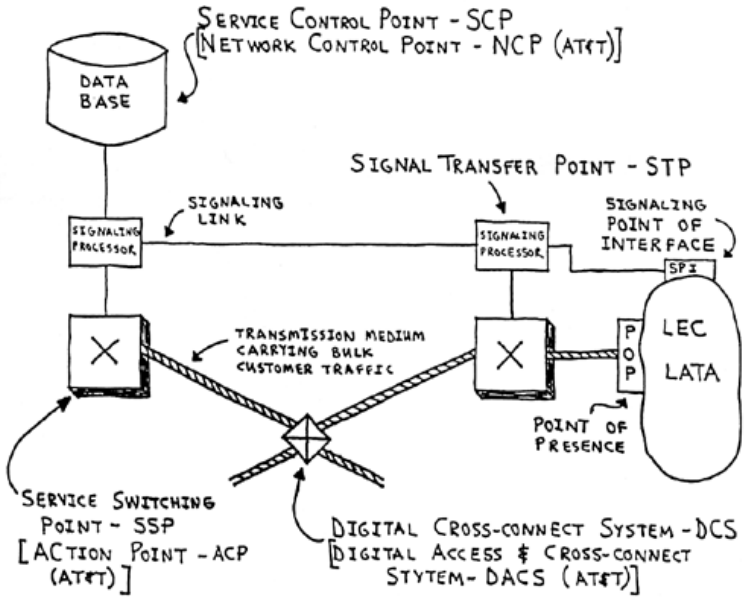
2.1.1 História

A ideia de usar um controlador central foi comum nas redes de telecomunicações. Por exemplo a AT&T, multinacional Americana de Telecomunicações, nas suas redes telefónicas dos anos 80 usava um *Network Control Point*(NCP). O NCP continha uma base de dados com informações de encaminhamento de circuitos de chamadas. Desta forma os *Service Switching Points* existentes na altura tinham de comunicar com os NCP para consultar o *routing* que iam efectuar. Com esta primeira aproximação é possível fazer uma analogia à aproximação SDN, porque já aqui o *control plane* estava contido num componente centralizado. Pode-se observar a arquitectura destas redes na Figura 2.1.

As redes de computadores, de forma geral, e em particular a Internet, adoptaram uma aproximação de controlo distribuído, pelo que os equipamentos das redes têm capacidade autónoma de controlo. Esta aproximação tem tendência a complicar a rede e nos últimos tempos tem sido questionada.

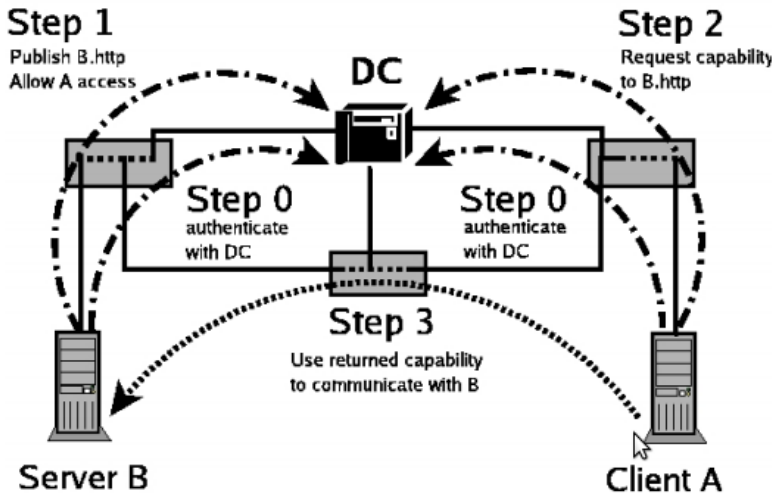
Por exemplo, num projecto de investigação em 2005, apareceu a necessidade de resolver melhor o problema de proteger as redes empresariais. Esta aproximação foi denominada Security Architecture for the Networked Enterprise(SANE)[CGA+06] e queria combater os problemas de ser difícil expressar políticas de segurança nas redes e de as políticas existentes serem fáceis de se violar por acidente ou erro. Para isto os investigadores propuseram criar um *Domain Controller* que: autenticava *end-hosts* e *switches*; continha a topologia da rede; guardava os serviços usados pelos *end-hosts* e continha uma funcionalidade de protecção onde era gerida toda a conectividade da rede. Com a implementação

Figura 2.1: Exemplo do modelo da AT&T



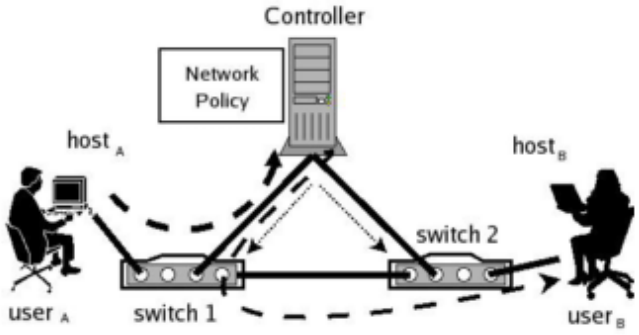
dos *Domain Controller*, os *switches* tinham de comunicar com os controladores para puderem interrogar sobre as políticas da rede. Esta arquitectura está ilustrada na Figura 2.2, retirada de [CGA+06].

Figura 2.2: Exemplo de um modelo com SANE



Ethane[CFP+07] foi uma implementação concreta com base na filosofia do SANE, também criada pelos mesmos investigadores. Para este efeito usavam redes baseadas em *flows* e faziam uso do *Domain Controller*, também usado em SANE, que tratava de autenticações, políticas de segurança globais e também decidia o processamento dos *switches* em relação aos *flows*. Pode-se observar a sua arquitectura na Figura 2.3, retirada de [CFP+07].

Figura 2.3: Exemplo de um modelo com Ethane



A possibilidade de gerir redes com base nesta filosofia exigia um protocolo específico para a comunicação entre os *switches* e os controladores. A definição para este protocolo e os objectivos desta aproximação levaram à criação de uma norma que será apresentada na próxima secção.

2.1.2 *OpenFlow*

OpenFlow [MAB+08] é uma norma pública que aparece no conjunto de propostas para a aproximação SDN, inspirada em Ethane, permitindo usar a arquitectura da aproximação SDN nas redes IP. Com este novo standard são possíveis múltiplas formas de gerir uma rede e permitir novas aplicações. A normalização do *OpenFlow* é fundamental para que a aproximação SDN sejam multi-fabricantes e consequentemente o preço dos equipamentos desça.

Quando foi criado o *OpenFlow*, os quatro objectivos desenhados para este eram:

1. suportar equipamentos com alta performance;
2. descer os custos via a normalização do *OpenFlow*;
3. suportar uma larga escala de investigação nesta área;
4. isolar tráfego experimental do tráfego de produção.

O primeiro e segundo objectivos partem do conceito de que hoje em dia os equipamentos proprietários serem muito fechados em termos de software. Existem alternativas aos equipamentos proprietários tais como Supercharged PlanetLab Platform [TCD+07] e NetFPGA [Neta], em que se pode programar estes equipamentos, mas estes têm as suas desvantagens. No primeiro a plataforma é cara e assim não compensa a substituição. O segundo consiste numa placa Ethernet especial com o limite de quatro portas, que não se integra nas redes actuais, em que se chega a ter algumas centenas de portas. Também uma outra alternativa é usar um PC mas este continua com a mesma limitação de número de portas que NetFPGA, e por isso não é realista. O terceiro e quarto objectivos consistem

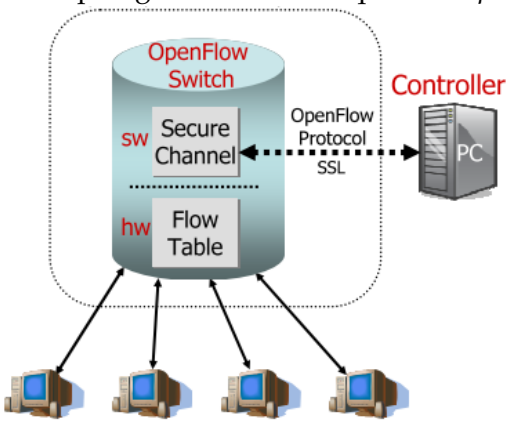
em permitir que uma rede continue a funcionar normalmente com as aplicações existentes ao mesmo tempo que é gerado o tráfego experimental, de forma a não parar estas aplicações.

A base do *OpenFlow* está na descoberta que todos os *switches* actualmente, possuem tabelas de encaminhamento com um certo número de funções comuns. Em consequência, o *OpenFlow* fornece um protocolo que permite programar remotamente esta funcionalidade de encaminhamento. Por exemplo, é possível para um investigador determinar qual a rota que quer que um pacote faça para chegar ao destino. Com esta nova funcionalidade do *OpenFlow* é possível criar novos protocolos numa rede e testá-los, assim como criar novos modelos de segurança, entre outros.

2.1.2.1 *Switches Openflow*

Um *switch Openflow* é constituído por três componentes: a tabela de encaminhamento, um canal seguro para o controlador e o suporte do protocolo *OpenFlow* como se observa na Figura 2.4 (retirada de [MAB+08]). A tabela de encaminhamento contém entradas a especificar o que fazer com cada tipo de pacote. O canal seguro é para assegurar a comunicação entre o *switch* e o controlador. Por fim o protocolo *OpenFlow* permite ao controlador comunicar com o *switch* de uma forma normalizada. Existem dois grupos de *swiches Openflow*, os dedicados e os *OpenFlow-enable*.

Figura 2.4: Topologia de uma rede que usa *OpenFlow*



Os *switches* dedicados são *switches* que não suportam o *layer* 2 e 3. Nestes, cada entrada na tabela de encaminhamento tem associado um cabeçalho que define uma acção e estatísticas. No campo das acções podemos encontrar uma das três seguintes acções: encaminhar os pacotes para uma determinada porta, encapsular e encaminhar os pacotes para o controlador ou descartar os pacotes. Nestes *swiches*, caso seja interceptado um pacote que não faz *matching* com as entradas programas, o mesmo é encaminhado para o controlador para que este o trate.

Os *switches Openflow-enable* são *switches* proprietários que detêm as três acções dos *switches* dedicados, fazendo uso do hardware existente neles. Estes contêm também uma

quarta acção que permitem realizar o encaminhamento normal do *switch*, de acordo com a configuração deste. Em resumo, permite-se separar o tráfego *OpenFlow* do tráfego normal, cumprindo-se assim um dos objectivos iniciais do *OpenFlow*.

2.1.2.2 Controladores

Os controladores adicionam e removem entradas na tabela de encaminhamento de acordo com os interesses de quem os programa. Por conseguinte, pode-se imaginar todo o tipo de controladores, tanto o mais simples que recebe um pacote e adiciona uma entrada na tabela para fazer encaminhamento para uma certa porta, como os mais sofisticados que podem fazer gestão de banda numa rede, adicionando e removendo dinamicamente entradas de acordo com medições de banda efectuadas na rede.

Devido à grande popularidade que o *OpenFlow* teve, é possível actualmente encontrar um grande leque de controladores. Uma simples pesquisa na Internet permite encontrar controladores *open-source* nas mais diversas linguagens de programação.

2.1.2.3 NOX

NOX[GKP+08] foi o primeiro controlador para o *OpenFlow*. É programado em Python ou C++ e actualmente os seus programadores criaram uma versão melhorada denominada POX, que é o controlador recomendado no tutorial oficial do *OpenFlow*.

2.1.2.4 Beacon

O Beacon[Eri] é um controlador programado em Java, suportado por David Erickson da Universidade de Stanford, uma das pessoas envolvidas no desenvolvimento do *OpenFlow*.

2.1.2.5 Floodlight

O Floodlight[Netb] é um controlador derivado do Beacon, programado em Java, suportado pela empresa Big Switch Networks.

O Floodlight divide-se em módulos para ser mais fácil de expandir as suas funcionalidades. Na sua implementação são incorporados alguns módulos por defeito como o módulo Hub e o módulo Leaning Switch que, tal como o nome indica, permitirá ao controlador actuar como *hub* e *switch*.

2.1.2.6 Onix

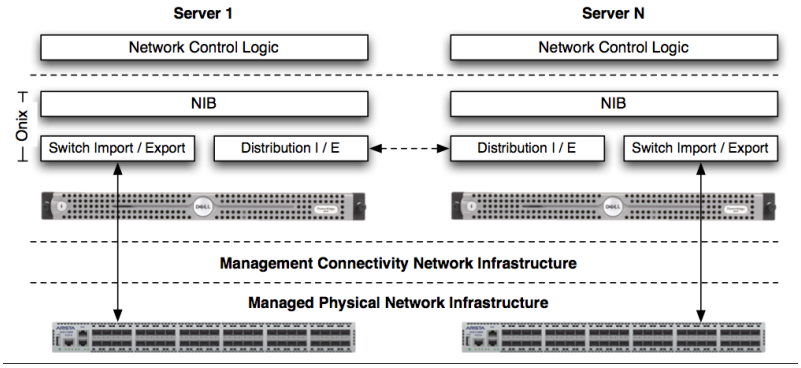
Onix[KCG+10] é um controlador que permite ser programado em C++, Java ou Python. A grande diferença deste para os outros é este ser distribuído, ou seja, a rede fica com vários controladores distribuídos. Na abordagem do Onix dá-se o nome de *Network Information Base*(NIB) ao conjunto de informações sobre a rede que os diferentes controladores procuram. Existem mais dois componentes importantes nesta arquitectura que

são o *Switch Import/Export* e o *Distribution Import/Export* como se pode verificar na Figura 2.5, retirada de [KCG+10]. O primeiro é responsável por tratar da comunicação entre os *switches* e a *Network Information Base*. O *Distribution Import/Export* trata de propagar as alterações de uma *Network Information Base* para as outras, de uma forma assíncrona.

No novo paradigma adoptado pelo Onix, também é possível criar hierarquias de *Network Information Bases* para, por exemplo, ter uma *Network Information Base* responsável por conjuntos de outras *Network Information Bases*.

Como este controlador é distribuído, também vem contradizer as desvantagens apontadas por muitos, sobre o controlador ser um componente centralizado.

Figura 2.5: Arquitectura de Onix



2.1.2.7 Exemplos de uso do *Openflow*

Exemplos relevantes da utilização do *OpenFlow* são dados em [Lim12], [VN11] e [MAB+08]. O exemplo mais relevante do *OpenFlow* é a possibilidade de dar suporte à investigação sem qualquer barreira. Por outro lado, nos centros de dados, o *OpenFlow* também é uma grande aposta. Actualmente já existem grandes empresas a utilizarem-no pois com o *OpenFlow* é mais simples a parametrização das suas políticas.

Um outro exemplo é a possibilidade de contornar um grande problema que tem a ver com a carga não balanceada nas redes. Com o *OpenFlow* é possível verificar a congestão de um canal e caso este esteja com grande carga, utilizar outro caminho para o destino.

Dos exemplos dados por Thomas Limoncelli em [Lim12], um interessante é o de um servidor de jogos. Durante um jogo na LAN, é muito importante que a latência seja baixa, para permitir uma melhor experiência de jogo. Com isto, podemos usar o *OpenFlow* para dar prioridade ao tráfego do jogo. Um outro exemplo ainda dentro deste caso dos jogos online, é que caso um jogador se mova entre *Access Points* diferentes, este mantém o IP e não perde a conexão ao jogo.

2.1.3 Simuladores de Redes com *Openflow*

2.1.3.1 Mininet

O Mininet[LHM10] é um simulador que permite testar redes *OpenFlow*. Este vem substituir por completo simuladores existentes como o ns-2 ou Opnet referidos em [LHM10], que realçam pouco a realidade. O Mininet, através de leves máquinas virtuais, distingue-se dos outros simuladores por ser flexível, interactivo, escalável e mais realista.

Quando se fala em flexibilidade no Mininet, refere-se à forma como quando se quer criar uma nova topologia, definir esta por software através de uma linguagem alto nível. Durante a simulação é possível alterações em tempo real o que faz com que este seja interactivo. O Mininet é escalável por ser possível simular milhares de nós e bastante realista por dar um alto nível de confiança, ou seja, quando se simula algo no Mininet e numa rede real, os resultados do Mininet são semelhantes aos da rede real.

O Mininet corre em ambientes Linux, onde usa várias funcionalidades deste tais como processos a correr em *network namespaces* e pares Ethernet virtuais.

O Mininet tem 4 componentes fundamentais numa rede: *hosts*, *links*, *switches* e controladores. Os hosts são processos da consola a correrem no seu *namespace*. Cada *host* tem as suas interfaces Ethernet virtuais e tem um *pipe* para o processo pai do Mininet. Os *links* são pares Ethernet virtuais que ligam duas interfaces virtuais. Os *switches* virtuais do Mininet vão realizar as mesmas funções de um *switch* físico.

Os controladores são definidos pelo utilizador, ou seja, o utilizador corre o seu controlador e, por parâmetro, informa o Mininet em que IP este está a ser executado. Isto é útil pois o controlador pode estar a correr na mesma máquina ou numa outra na rede.

É possível criar uma rede de várias formas com o Mininet. A primeira forma de criar uma rede é por parâmetro quando se corre o Mininet tal como o exemplo que se segue:

```
mn --switch ovsk --topo tree,depth=2,fanout=8 --controller remote
```

Neste exemplo criou-se uma rede em árvore com profundidade 2, 8 portas em cada *switch* OpenVSwitch e um controlador remoto. Este pode ainda ter o parâmetro *–ip* que permite dar a localização do controlador à rede, sendo assim possível executar o controlador numa outra máquina que não a do Mininet.

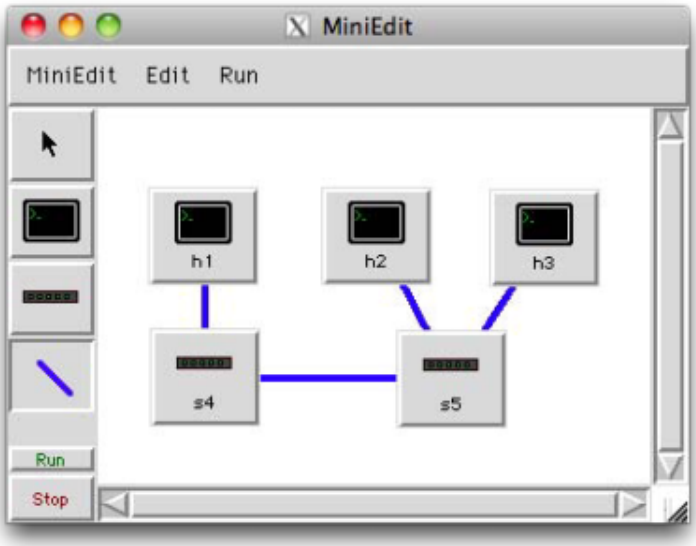
A segunda forma é através da especificação de um ficheiro Python. Segue-se um exemplo da mesma topologia criada acima nesse ficheiro:

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree = TreeTopo(depth=2,fanout=8)
net = Mininet(topo=tree)
net.start()
```

Este ficheiro é preciso incluir em parâmetro quando se corre o Mininet.

A última forma de criar topologias é por uma ferramenta *GUI* que o Mininet incorpora. A Figura 2.6 ilustra essa *GUI*.

Figura 2.6: Ferramenta GUI do Mininet



Depois de criada uma rede é possível interagir com esta, através da consola do Mininet. Através desta é possível correr todas as ferramentas que se utilizam para *debug* das redes tais como ifconfig, tcpdump e todas as outras que estejam instaladas no ambiente Linux onde se está a correr o simulador. Também é possível correr comandos que façam a interacção entre *hosts* tais como:

```
h2 ping 10.0.0.3
```

Que significa que o *host* 2 vai executar o comando ping ao IP 10.0.0.3. Caso não se pretenda usar esta sintaxe de 'nó comando', pode-se abrir consolas para cada *host* em janelas separadas, para simplesmente correr 'ping 10.0.0.3' que se tinha executado anteriormente. Para isso dever-se-á correr:

```
xterm h2
```

A limitação mais significativa do Mininet é a fidelidade do desempenho e do escalonamento de pacotes especialmente com grande carga. Isto deve-se ao facto de que o escalonador do Linux serializa a execução de todos os eventos que têm lugar no sistema, mesmo aqueles que na realidade se processariam em paralelo, e por outro lado não simula correctamente o custo da comunicação nos canais visto que a mesma está dependente do desempenho do *kernel*. Assim, não se dá garantia de dois *switches* existentes no Mininet sejam capazes de encaminhar pacotes como aconteceria na realidade.

2.1.4 Linguagens de programação de alto nível nos controladores

Dos controladores encontrados, a maior parte são programados usando linguagens convencionais como C++, Java, Python, etc. No entanto para simplificar a programação dos programadores, foram criadas algumas linguagens de programação. Estas têm como objectivo fornecer um modelo de mais alto nível ao programador. Dessas linguagens existem duas bastante mais avançadas na sua definição: Procera e Frenetic.

Procera[VKF12] é uma linguagem especificada por Andreas Voellmy e outros da Universidade de Yale, cujo objectivo tal como apresentado nesta secção, é criar uma linguagem de controlo das redes *OpenFlow* mais simplificada. Procera é uma linguagem reactiva, ou seja, reage a eventos da rede e tem como objectivo expressar políticas de segurança.

Frenetic[FHF+11] é uma linguagem desenvolvida pelos investigadores de Princeton. Tal como Procera também é uma linguagem reactiva. Esta linguagem vem combater duas dificuldades apontadas quando se programa um controlador NOX no *OpenFlow*. A primeira dificuldade está na interacção entre módulos, onde o NOX não comunica entre os módulos criados, sendo que trata cada pacote de acordo com as regras definidas nos módulos. A segunda dificuldade está na linguagem de baixo nível necessária para programar o NOX.

Note-se que estas linguagens estão a crescer na aproximação SDN mas que a programação em linguagens convencionais nos controladores ainda é a mais utilizada.

2.2 Funcionamento do IP Unicast numa rede *switched*

Um *host*[1] numa rede IP tem dois endereços, o seu endereço MAC e o seu endereço IP. O endereço MAC é um valor único atribuído à placa de rede na fábrica enquanto o endereço IP é um valor configurado ou atribuído dinamicamente. Para se efectuar comunicações numa rede IP é necessário obter um endereço IP. Quer isto dizer que o computador precisa de configurar manualmente um IP ou então adquirir um IP através do protocolo DHCP - Dynamic Host Configuration Protocol.

O protocolo DHCP tem como objectivo atribuir endereços IP aos computadores. Este serviço é o primeiro passo quando um computador se liga a uma rede IP. Para se iniciar o protocolo, primeiramente o computador difunde uma mensagem para descobrir um servidor DHCP. Esta mensagem é designada de DHCP Discovery. Quando um router (admitindo que não há servidores DHCP dedicados na rede) recebe uma mensagem deste tipo, envia uma mensagem de oferta ao computador com parâmetros de configuração. Esta mensagem tem o nome de DHCP Offer. O computador ao receber a mensagem de oferta, responde aceitando os parâmetros propostos enviando um DHCP Request, recebendo posteriormente uma confirmação (DHCP ACK).

Embora os computadores já obtenham endereços IP através do protocolo DHCP, é

[1] Onde a seguir designaremos pela palavra portuguesa computador.

necessário saberem mais algumas informações para conseguirem comunicar com outros computadores. Quando um computador h1 necessita de comunicar com um computador h2, o primeiro necessita de conhecer o endereço IP do segundo. Conhecendo o endereço IP de destino, h1 cria um pacote IP com endereço IP de destino, o endereço IP do h2, e encapsula numa *frame* Ethernet que necessita de um endereço MAC correspondente. De forma a obter o endereço MAC, h1 consulta a sua tabela ARP que contém endereços IP associados a endereços MAC. Caso h1 contenha uma entrada na tabela com o endereço IP de destino, obtém o endereço MAC correspondente e preenche o campo de destino da *frame* Ethernet com o mesmo. Senão existir uma entrada com esse endereço IP, o computador necessita de obter a associação do endereço IP com o endereço MAC correspondente, usando o protocolo ARP.

O protocolo ARP tem como objectivo descobrir endereços MAC associados a um determinado endereço IP. Quando um computador não conhece um endereço MAC associado a um IP, difunde uma mensagem a perguntar qual o endereço MAC associado a um endereço IP. Esta mensagem é designada de ARP Request. Seguindo o exemplo anterior, o h1 enviaria um ARP Request para descobrir o endereço MAC de h2. Quando h2 recebesse por difusão esta mensagem, iria responder identificando o seu endereço MAC na resposta. Esta mensagem de resposta tem o nome de ARP Reply. Assim, já h1 conseguiria criar uma frame e enviá-la para h2.

Numa rede *switched*, se um *switch* recebe uma *frame* Ethernet, este verifica o seu endereço MAC de destino para determinar para que porta encaminhar a mesma. Para este objectivo, os *switches* contêm uma tabela de encaminhamento que associa os endereços MAC com as suas interfaces. No caso de não encontrarem uma associação, recorrerem à inundação.

Para tornar a inundação realista numa rede em malha, porque esta pode dar origem a *broadcast storms*, criou-se o protocolo Spanning Tree Protocol (STP). Este protocolo cria uma árvore de encaminhamento sem ciclos nos *switches*. Inicialmente quando os *switches* se ligam, precisam de eleger uma raiz da árvore que será o *switch* com o identificador mais baixo entre todos. À medida que os *switches* vão comunicando os seus identificadores, vão também calculando a sua distância até à raiz actual. Neste calculo vão preferir o caminho com menos *hops* até à raiz.

Como se observou, o encaminhamento numa rede Ethernet comutada é baseado em inundação através de árvores calculadas pelo protocolo STP e com aprendizagem pelo caminho inverso para optimização do tráfego ponto a ponto. Nesta abordagem é visível que haverá um grande problema se se utilizar a difusão em redes de grande escala. Assim sendo, de forma a reduzir as difusões, as redes empresariais utilizam VLANs (redes locais virtuais) para que as difusões sejam confinadas por estas e não por toda a rede global. As VLANs utilizam árvores STP individuais e para comunicarem entre elas utilizam o encaminhamento IP. Todo este processo é bastante complexo e pouco optimizado.

Uma abordagem a este problema utilizando a aproximação SDN seria implementar

um controlador que ao conhecer a topologia da rede, evitasse as difusões para aprendizagem de endereços IP e MAC. Este controlador também seria capaz de lidar com todos os equipamentos na rede, ou por outras palavras, reduziria a complexidade da rede à de um *switch* com centenas de portas usando um único prefixo IP. Por último, ainda seria possível desenvolver código no controlador para reproduzir políticas de segurança.

Para implementar o IP Unicast, o controlador escolhido, o Floodlight, já aparenta ter módulos que são capazes de lidar com esta funcionalidade.

2.3 Funcionamento do IP Multicast numa rede *switched*

Numa rede IP a utilização de IP Multicasting é composta pela utilização de um protocolo de gestão de subscrições, o Internet Group Membership Protocol (IGMP), e pelo encaminhamento do tráfego IP através de árvores de difusão. Numa única VLAN, a utilização do IP Multicasting pode ser implementada facilmente. Na abordagem mais comum, os *switches* recorrem à difusão para enviar o tráfego *multicast*. Outras técnicas foram criadas como IGMP Snooping, Cisco's Group Management Protocol (CGMP) e IEEE's Group Address Resolution Protocol (GARP). Estas técnicas restringem a difusão apenas aos ramos da árvore com subscritores do grupo.

O IGMP Snooping é uma optimização do L2 para o IGMP do L3. Esta implementação permite ao *switch* reconhecer as portas que subscreveram tráfego *multicast*, enviado assim, só o tráfego *multicast* para essas portas. O IGMP Snooping tem algumas desvantagens como não poder ser usado em redes que não são IP e ter um *overhead* devido ao *snooping* efectuado em cada pacote IGMP.

Quando um grupo IP Multicast abrange mais do que uma VLAN, é necessário realizar encaminhamento IP Multicasting através de protocolos especiais como por exemplo PIM-SM que é uma das soluções mais comuns e simples. A complexidade da gestão da rede continuará a aumentar e o software que os *switches* convencionais executam terá de suportar: *switches* com várias VLANs, STP, encaminhamento IP Unicasting, IGMP, encaminhamento IP Multicasting e optimização de encaminhamento *multicasting* por exemplo usando IGMP *snooping*. A gestão desta rede, é complexa e passou a requerer muitos conhecimentos que ultrapassam largamente os necessários para gerir uma rede IP simples, com único prefixo IP e baseada num só *switch*, onde a gestão do IP Multicasting é muito simples. Nas redes baseadas num só *switch* este até é mais barato pois não necessita senão de realizar *frame switching* e optimização pelo caminho inverso, dispensa a existência de um *control plane* complexo, quer ao nível 2, quer ao nível 3. O desafio será testar se a implementação do IP Multicasting pelo controlador será possível.

Para a implementação do IP Multicasting é necessário implementar toda a gestão dos grupos existentes (utilizando o IGMP para diálogo com os computadores) e implementar a difusão dos pacotes IP Multicast através de árvores de difusão independentes para cada grupo. Este é um dos primeiros desafios a tentar vencer.

O encaminhamento IP Multicast tem como protocolo de apoio o IGMP. O protocolo

IGMP[2] é um protocolo simples que dá a conhecer aos *routers* os grupos *multicast* existentes e os membros associados a cada grupo. Quando um membro deseja subscrever um grupo, envia um pacote *multicast* para o endereço desse grupo. O *router* no momento em que recebe este pacote, associa o emissor ao grupo pretendido. Estes pacotes têm o nome de IGMP Membership Reports. Para os *routers* saberem se os grupos *multicast* subscritos ainda têm interessados localmente, periodicamente difundem um pacote para o endereço 224.0.0.1. Estes pacotes são designados IGMP Membership Queries, e têm como objectivo desencadearem uma resposta por parte dos computadores que ainda estão a subscrever o seu grupo. De modo a evitar que todos os computadores de um grupo respondam a uma IGMP Membership Query, quando é enviada uma resposta por parte de um computador, os outros recebem-na também e não respondem. Isto é possível porque a resposta é enviada para o endereço do grupo. Se um computador desejar abandonar um grupo, este envia um pacote para o mesmo, para que o *router* actualize a sua informação sobre esse grupo. Estes pacotes têm o nome de IGMP Leave Group.

Conhecidos os grupos *multicast* pelos *routers*, é necessários estes saberem calcular as rotas através dos protocolos de encaminhamento. Existem vários algoritmos de encaminhamento *multicast* como Protocol-Independent Multicast Dense Mode (PIM-DM), PIM-SM, Distance-Vector Multicast Routing Protocol (DVMRP) e Multicast Extensions to Open Shortest-Path First Protocol (MOSPF).

Os protocolos PIM são protocolos que usam a informação já obtida pela tabela de encaminhamento do *unicast*. Desta forma, não trocam qualquer informação para criar tabelas de encaminhamento para o *multicast*. O protocolo PIM-SM tem como base o uso de um *Rendevous Point* (RP). O RP é responsável por ser a raiz da árvore partilhada ou *Rendevous Point Tree* (RPT), que vai crescendo à medida que os *routers* com subscritores enviam *Joins* para o RP. A árvore partilhada é construída com base nas informações da tabela de encaminhamento *unicast*. Desta forma quando um emissor envia tráfego para um grupo, este é enviado para o RP através de um túnel e o RP encaminha para os subscritores através da RPT.

Este protocolo também tem a particularidade de poder construir uma árvore por emissor. Estas árvores também designadas de Shortest Path Trees (SPT) são utilizadas quando o tráfego *multicast* de um emissor excede um limite definido num *last-hop router*. Quando isto acontece, um router constrói um novo ramo de uma árvore SPT com o emissor como raiz e o emissor passa a enviar o seu tráfego *multicast* para um grupo através dessa SPT (para além de continuar a enviar para o RP).

Em resumo, o PIM-SM usa uma RPT para a fase inicial de *Joins* aos grupos e para encaminhar tráfego *multicast* de baixa capacidade. No momento em que um *router* detecta que a capacidade pré-estabelecida para o tráfego foi atingida, este cria uma SPT.

Para implementar o IP Multicasting, o controlador vai passar a ter de receber os IGMP Reports, enviar IGMP Queries, gerir os grupos e a lista de subscritores, detectar a presença de emissores e construir árvores de difusão do tráfego IP Multicasting que terão de

[2]Nesta dissertação a versão do IGMP descrita é a versão 2.

ser parametrizadas nos *switches* através do *OpenFlow*.

3

Floodlight

Neste capítulo é descrito o controlador utilizado, o Floodlight. Na primeira secção do capítulo é introduzido o funcionamento geral do controlador. Em seguida, é descrito o funcionamento dos principais módulos do Floodlight. Por último, apresenta-se o funcionamento do IP Unicast e do IP Multicast no controlador.

3.1 Descrição

O Floodlight é o controlador escolhido para desenvolver a solução apresentada nesta dissertação. A escolha recaiu sobre este controlador, principalmente por ser programado em Java, o que facilita a sua aprendizagem. Outro dos outros aspectos que teve grande peso na escolha foi a grande comunidade que o Floodlight apresenta. Desta forma caso surja alguma questão, pode-se recorrer sempre à mesma para tentar resolver um problema. O Floodlight é repartido em módulos o que torna mais fácil a sua aprendizagem e a sua extensibilidade. No âmbito desta dissertação, os objectivos passarão por criar módulos e modificar os módulos existentes para estender a suas funcionalidades. Destacam-se na Tabela 3.1 os módulos que se entendem como sendo necessários ao funcionamento básico do controlador e uma pequena descrição dos mesmos.

O Floodlight apresenta um fluxo de trabalho bastante complexo e compreender este fluxo necessitou de um ajuste na configuração da classe Logger. Desta forma foi possível compreender melhor o que é efectuado pelo controlador porque a consola apresenta mais informação. Com efeito, posteriormente verificou-se que o controlador, dado pretender ultrapassar o âmbito de uma ferramenta pedagógica e de suporte à investigação, mas pretender também suportar uma rede em produção, veio a revelar-se bastante complexo.

Numa primeira fase, a classe Main é executada e esta irá desencadear dois passos

Módulo	**Descrição**
Controller	Funcionamento do controlador.
StaticFlowEntryPusher	Mantém um conjunto de *flows* estáticos nos *switches*.
ThreadPool	Gestão de *threads* do controlador.
DefaultEntityClassifier	Parâmetros que classificam um computador.
CounterStore	Contadores do controlador.
LinkDiscoveryManager	Gestão dos canais da rede.
PktInProcessingTime	Estatísticas do controlador.
TopologyManagerImpl	Mantém uma noção do grafo da rede.
DeviceManagerImpl	Gestor de computadores da rede.
MemoryStorageSource	Fornece uma estrutura de dados do tipo tabela para os módulos que necessitarem.
FlowReconcileManager	Responsável por substituir os *flows* quando existe um canal que fica indisponível.
Forwarding	Resposável por instalar *flows*.

Tabela 3.1: Módulos do Floodlight

importantes: o carregamento de módulos e a inicialização do servidor REST. Destaca-se o primeiro passo que irá carregar todos os módulos através do ficheiro de configuração *floodlightdefault.properties*. Os módulos que estão presentes nesse ficheiro de configuração são os que estão apresentados na Tabela 3.1. O módulo principal é o Controller e é este que gere os *switches* ligados ao controlador e processa mensagens recebidas dos *switches*. Quando um *switch* se liga inicialmente ao controlador é inicializado um protocolo como se pode observar na Figura 3.1.

Caso o protocolo seja efectuado com sucesso, o controlador e o *switch* ficam a conhecer-se mutuamente e já podem realizar comunicações entre eles. Quando um *switch* envia uma mensagem para o controlador, a classe Controller chama o método *handleMessage()* que irá distribuir a mensagem pelos módulos interessados. Desta forma, quando surge um evento no controlador, este é responsável por perceber de que tipo de evento se trata e distribuir a mensagem pelos módulos interessados. Esta distribuição das mensagens segue um grafo de precedências, ou seja, a classe Controller quando entrega a mensagem aos módulos, segue uma ordem de entrega. Esta ordem é definida nos próprios módulos. Na imagem 3.2 mostra-se o grafo de precedências dos módulos do Floodlight.

Todos os módulos têm uma estrutura definida de acordo com a interface IFloodlightModule. Nesta interface, estão presentes métodos que:

- dão a opção de exportar variáveis de forma a outros módulos poderem aceder a estas;
- inicializam variáveis;
- permitem criar *threads* para outros trabalhos do módulo;

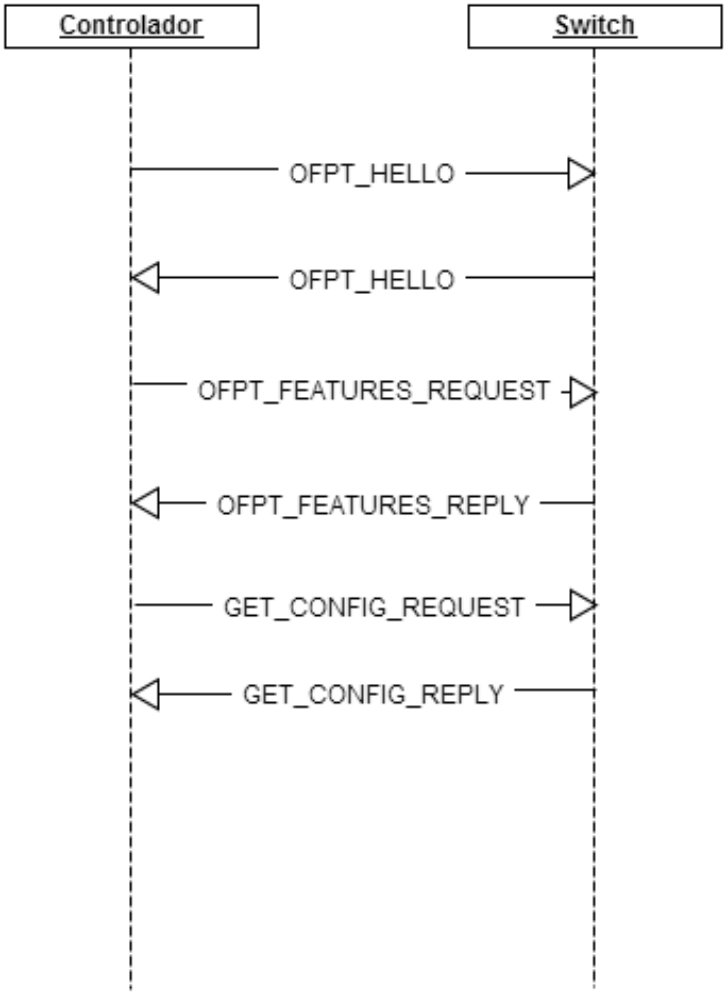


Figura 3.1: Protocolo de inicialização dos switches

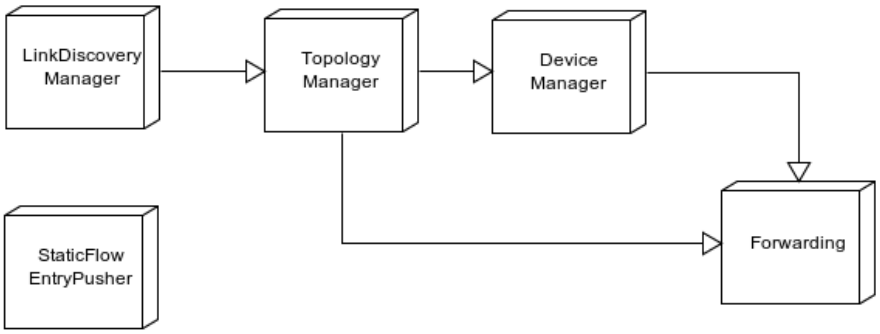


Figura 3.2: Grafo de precedências de módulos

- definem os módulos a que se pretende aceder, de forma a consultar as suas variáveis.

Como se pode observar por esta definição dos módulos, é necessário ainda implementar outras interfaces caso se queira subscrever determinado tipo de eventos. Para os mais comuns basta o módulo implementar a interface IOFMessageListener. Esta interface permite tratar os tipos de mensagens existentes na tabela 3.2.

Actualmente no Floodlight, o módulo Controller é que subscreve maior parte destes tipos de mensagens. Os outros módulos que subscrevem mensagens são:

- StaticFlowEntryPusher, a subscrever mensagens do tipo FLOW_REMOVED;
- LinkDiscoveryManager, a subscrever mensagens do tipo PORT_STATUS;
- TopologyManager, DeviceManager, LinkDiscoveryManager e Forwarding, a subscreverem mensagens do tipo PACKET_IN.

A interface IOFMessageListener tem um método *receive()* de forma a que o módulo quando recebe uma mensagem, entregue pela classe Controller, a trate de forma conveniente. Este método tem de devolver um comando *CONTINUE* ou *STOP* para o controlador saber se distribui o pacote para os próximos módulos interessados ou pelo contrário, parará a sua propagação. Nas próximas secções dá-se a conhecer a forma como os principais módulos lidam com as mensagens que recebem.

3.2 Módulo Controller

O módulo Controller é responsável por um conjunto de funcionalidades básicas para o controlador funcionar. Inicialmente, corre o método *run()*, chamado pela classe *Main*, que cria um servidor para aceitar conexões vindas dos *switches*. Este servidor é criado com um *handler* para o canal de comunicação, o *OFChannelHandler*. É neste *handler* que estão implementadas as funcionalidades de tratamento de mensagens enviadas para o controlador pelos *switches*, mais especificamente no método *messageReceived()*. Este módulo, para além de tratar as mensagens recebidas, também é responsável por identificar novas conexões e desconexões por parte dos *switches*. Quando um *switch* se conecta ao controlador, como já foi retratado em 3.1, é executado o protocolo da figura 3.1 para que o *switch* e o controlador se passem a conhecer mutuamente..

Através do método *messageReceived()*, o controlador trata as mensagens recebidas, isto é, executa os procedimentos locais para aquelas em que tem intervenção directa, e distribuí as mensagens pelos módulos que declararam interesse nas mesmas. Para clarificar a forma como este módulo funciona, apresenta-se de seguida um exemplo de pseudo-código da sua estrutura, ilustrando o estilo de programação utilizado no controlador.

Tipo de mensagem	**Descrição**
HELLO	Mensagem trocada inicialmente entre controlador e *switch* após este se conectar.
ERROR	Mensagem enviada pelo *switch* para notificar o controlador de erros.
ECHO_REQUEST	Mensagem enviada pelo controlador de forma a que o *switch* devolva uma resposta. Usada para medir latências.
ECHO_REPLY	Resposta à mensagem ECHO_REQUEST.
VENDOR	Mensagem específica sobre o fabricante enviada pelo *switch* para o controlador.
FEATURES_REQUEST	Mensagem enviada pelo controlador a pedir as especificações do *switch*.
FEATURES_REPLY	Resposta à mensagem FEATURES_REQUEST.
GET_CONFIG_REQUEST	Mensagem enviada pelo controlador a pedir as configurações do *switch*.
GET_CONFIG_REPLY	Resposta à mensagem GET_CONFIG_REQUEST.
SET_CONFIG	Mensagem enviada pelo controlador para o *switch*, a definir parâmetros de configuração do *switch*.
PACKET_IN	Mensagem da rede recebida pelo *switch* e que este encaminha para o controlador.
FLOW_REMOVED	Mensagem enviada pelo *switch* para o controlador a avisar que um *flow* foi removido.
PORT_STATUS	Mensagem enviada pelo *switch* para o controlador a avisar que uma porta sua foi adicionada, removida ou a sua configuração alterada.
PACKET_OUT	Mensagem enviada pelo controlador para o *switch* para que este a envie para a rede.
FLOW_MOD	Mensagem enviada pelo controlador para o *switch* a instalar, modificar ou a remover um *flow*.
PORT_MOD	Mensagem enviada pelo controlador para o *switch* a alterar o comportamento de uma porta.
STATS_REQUEST	Mensagem enviada pelo controlador para o *switch* a pedir o estado actual do *switch*.
STATS_REPLY	Resposta à mensagem STATS_REQUEST.
BARRIER_REQUEST	Mensagem enviada pelo controlador para garantir que dependências de mensagens são cumpridas.
BARRIER_REPLY	Resposta à mensagem BARRIER_REQUEST.

Tabela 3.2: Mensagens do Floodlight

```
run(){
  create_server();
  while(true){
    //handle switch updates
    // like new connections and disconnections
    handleNewUpdates();
  }
}


create_server(){
  ControllerServer of = new ControllerServer(OFChannelHandler);
  //channel that accepts switches connections
  ChannelGroup cg = new ChannelGroup;
  cg(of.bind(new InetSocketAddress(openFlowPort));
}

handleNewUpdates(){
  if update is new Switch
    add switch to known switches
  else
    remove switch from known switches
}

messageReceived(MessageEvent e){
  List<OFMessage> msglist = (List<OFMessage>)e.getMessage();
  for each msg : msglist
    processMessage(msg);
}

processMessage(OFMessage msg){
  switch msg
    case HELLO
      send FEATURES_REQUEST;
    case ECHO_REQUEST
      send ECHO_REPLY;
    case ECHO_REPLY
      break;
    case FEATURES REPLY
      save features;
```

```
    send GET_CONFIG_REQUEST;
  case GET_CONFIG_REPLY
    set that switch = ready;
  case VENDOR
    handle many vendor specific messages;
  case ERROR
    handle errors;
  case STATS_REPLY
    //get switch's table informations
    save switch statistics;
  case PORT_STATUS
    trigger event to the main loop
  default
    for each listener : listOfMessageListener
      if listener is interessed
        listener.received(msg)
```

3.3 Módulo LinkDiscoveryManager

O módulo LinkDiscoveryManager é responsável por manter as informações dos canais entre os *switches OpenFlow* da rede. Para esse efeito, este módulo envia periodicamente mensagens, usando o protocolo Link Layer Discovery Protocol (LLDP), para os *switches*. Essas mensagens são enviadas por inundação e como o Floodlight não instala *flows* por defeito para este tráfego, irá recebe-las de volta no controlador quando chegarem a outro *switch*. Este módulo na sua definição tem especificado que quer tratar mensagens do tipo PACKET_IN e PORT_STATUS.

O módulo LinkDiscoveryManager quando recebe um PACKET_IN, irá verificar se este é do tipo LLDP porque é o único tipo de mensagens em que este módulo está interessado. Depois, ao verificar o conteúdo da mensagem, o módulo consegue identificar canais entre *switches*. Esta identificação é conseguida porque a mensagem recebida contém a identificação do *switch* que enviou a mensagem origem para a rede, e o controlador dá a conhecer ao módulo qual o *switch* que recebeu o PACKET_IN. Assim, através desta forma de reconhecimento, o controlador fica a conhecer os canais entre *switches*. De notar que no fim de analisar uma mensagem do tipo LLDP, o módulo envia o comando *STOP* de forma a esta ser descartada.

Por outro lado, o LinkDiscoveryManager quando recebe um PORT_STATUS, vai verificar se a porta é alguma das que comunica com outros *switches* e por sua vez, adicionar ou remover um canal da sua lista de canais.

3.4 Módulo TopologyManager

O módulo TopologyManager é responsável por manter informações sobre a topologia da rede. Este módulo exporta ainda o serviço de encaminhamento, porque é este que faz o cálculo de caminhos na rede usado pelo módulo Forwarding. Através de um objecto, o TopologyInstance, o módulo TopologyManager calcula árvores de melhores caminhos tendo um certo nó como raiz. A árvore de difusão é a árvore que tem raiz no *switch* com o menor identificador, equivalente à árvore calculada pelo STP. O TopologyInstance recorre ao algoritmo de Dijkstra para calcular estas árvores. As mesmas são portanto árvores de caminhos mais curtos.

3.5 Módulo DeviceManager

O módulo DeviceManager é responsável por manter informações sobre os computadores da rede. Este módulo quando recebe um PACKET_IN, vai analisar a mensagem e criar ou actualizar um objecto Device consoante as informações que recolhe sobre o emissor da mensagem. Os objectos Device são objectos que contêm as seguintes informações: um identificador único, o seu endereço MAC, o seu endereço IP, a sua VLAN, a data da última vez que se manifestou na rede e os seus AttatchmentPoints, ou seja, as portas do/s *switch/s* a que está ligado. O módulo DeviceManager também exporta um serviço que permite aceder a todas as informações dos Devices da rede.

3.6 Módulo Forwarding

O módulo Forwarding é responsável por encaminhar pacotes e instalar *flows*. Para este efeito, quando um PACKET_IN chega ao módulo, este vai analisá-lo e escolher entre fazer difusão ou instalar um *flow*. Este módulo também tem outras formas de lidar com as mensagens, como as descartar, mas para isso é necessário ter o módulo Firewall activo.

Quando o módulo recebe um pacote *broadcast*, este vai chamar o método *doFlood()*. Este método verifica se a porta do *switch* pela qual a mensagem chegou faz parte da cobertura da árvore de difusão da rede referida em 3.4. Caso a porta faça parte da árvore de difusão, envia um PACKET_OUT para o *switch* fazer *flood*. Desta forma são evitados os *broadcast storms* porque o tráfego é todo testado antes de ser enviado por inundação.

No caso do módulo receber um pacote *unicast*, é chamada a função *doFlowForward()*. Nesta função o módulo consulta a árvore de melhores caminhos com o *switch* emissor da mensagem como raiz, e instala os *flows* correspondentes ao caminho mais curto da origem ao destinos nos *switches* que pertencem ao caminho.

Todas as estruturas de dados relacionadas com as árvores da rede foram introduzidas em 3.4.

3.6.1 Funcionamento do IP Unicast

Um computador quando inicia uma comunicação por IP Unicast, terá de enviar uma ARP Query através de *broadcast* pois este não tem a tabela ARP preenchida. Na implementação por defeito do Floodlight o trafégo *broadcast* é todo encaminhado para o controlador pois, por defeito, os *switches* não têm *flows* instalados para este tipo de tráfego. Esta acção é premeditada porque se houvessem *flows* instalados, o controlador não saberia da existência destes computadores quando estes se manifestam inicialmente. Esta implementação também permite que o controlador controle a localização dos computadores na rede (cf. 3.5). Entretanto, da mesma forma como o controlador conheceu o primeiro computador também conheceu o segundo, porque este se manifestou na resposta ao ARP. Quando os computadores se manifestam o controlador guarda as suas informações na estrutura de dados Device (cf. 3.5).

3.6.2 Funcionamento do IP Multicast

Na implementação do Floodlight, por defeito, o *multicast L2*, o *multicast L3* e o *broadcast* são tratados da mesma forma, usando a árvore de difusão já introduzida em 3.4. Desta forma compete ao controlador assegurar o encaminhamento por esta árvore, de todo o tráfego deste tipo.

4

Implementação

Neste capítulo é descrita a implementação realizada. Esta compreende duas partes. Por um lado foi introduzido um novo módulo, o módulo IPDiscoveryManager, que é descrito na secção 4.1. Em seguida são descritas as modificações introduzidas no módulo Forwarding que implementam as optimizações do encaminhamento IP Multicast, secção 4.2.

4.1 Descrição do módulo IPDiscoveryManager

Os objectivos do módulo IPDiscoveryManager estão relacionados com a optimização do funcionamento do protocolo IP, nomeadamente tentando evitar o mais que possível a utilização de difusões inúteis na rede (nomeadamente as relacionadas com o funcionamento do protocolo ARP). O módulo também realiza as acções necessárias para recensear os grupos IP Multicast que existem na rede através de uma solução baseada em IGMP.

4.1.1 Optimização do protocolo ARP

Uma das funcionalidades do módulo é optimizar o funcionamento do protocolo ARP minimizando as suas difusões de forma a reduzir o número de *frames* Ethernet que atravessam os diferentes canais.

O módulo usa as informações recolhidas no controlador pelo módulo DeviceManager como uma cache partilhada de ARP. Com efeito, essa cache permite obter por *soft state* a associação entre endereços IP Unicast e os endereços MAC das interfaces com esse endereço IP.

Assim, o módulo IPDiscoveryManager intercepta os pedidos ARP (mensagem ARP

Request enviada em *broadcast*) e, caso a informação solicitada esteja na cache (isto é, tenha sido introduzida pelo módulo DeviceManager na tabela de Devices) há menos que um intervalo de tempo (daqui para a frente designado por TRUST INTERVAL e actualmente com o valor de 60 segundos) responde directamente com um pacote ARP Reply. Caso contrário deixa o *broadcast* prosseguir.

Assim sendo, o controlador quando recebe um ARP Request irá consultar a estrutura de dados exportada pelo DeviceManager. A partir desta, vai verificar se o computador objectivo do ARP Request existe e qual a última vez que foi visto pelo controlador. Caso a última vez que foi visto na rede tenha sido há menos de TRUST INTERVAL, considera-se essa informação válida e cria-se um pacote ARP Reply para responder ao computador que enviou o ARP Request. Em consequência, vai-se evitar possíveis pacotes *broadcast* gerados pelos computadores. Se a última vez que o computador foi visto pelo controlador tiver sido há mais de TRUST INTERVAL segundos, o pacote segue o tratamento tradicional do controlador, sendo enviado por *broadcast* para a rede. Em síntese, nesta implementação o controlador terá de:

- interceptar os pacotes ARP Request, verificando nas suas estruturas de dados se o computador de destino enviou pacotes que chegaram ao controlador há menos de TRUST INTERVAL segundos;
- responder aos pacotes ARP Request caso a condição acima descrita seja satisfeita.

Para se compreender melhor o papel desta cache é necessário ter em atenção que o controlador receberá os pacotes emitidos por um computador IP sempre que estes não são encaminhados directamente pelos *switches*. Isso passa-se com todos os ARP Requests visto que os mesmos são emitidos em difusão e com todos os primeiros pacotes de cada novo fluxo *unicast*, sendo estes caracterizados por aparecer um pacote em que os campos IP origem, IP destino ou VLAN são distintos de outros anteriores.

Sempre que um novo fluxo é iniciado, o controlador instala o seu caminho nos *switches* através de comandos *OpenFlow*. Esses comandos são válidos enquanto o fluxo tiver actividade, e durante esse tempo os pacotes do fluxo não chegam ao controlador. No entanto, qualquer ARP Request assim como algum ARP Reply e ainda qualquer novo fluxo que seja iniciado por qualquer outro par de computadores, permitirá ao controlador enriquecer a sua cache com nova informação.

Para clarificar esta funcionalidade do módulo, apresenta-se o pseudo-código correspondente, que ilustra igualmente o estilo de programação usada no controlador.

```
static final int TRUSTINTERVAL;
//exported by deviceManager module
Set<Device> devices;


received(PacketIn pi, Switch sw){
```

```
  if (pi is an ARP packet)
    return handleArp(pi, sw)
}

handleArp(PacketIn pi, Switch sw){
  device = device such that device.IP = pi.targetIP
  if (device.lastSeen < TRUSTINTERVAL){
    send arp reply
    return stop
  }else{
    return continues
  }
}
```

Em suma, o módulo tenta transformar um protocolo pedido/resposta implementado por defeito por difusão de pedidos, num protocolo em que o servidor serve a totalidade da rede através de um protocolo pedido/resposta ponto a ponto.

A discussão desta solução levanta várias questões. A primeira é a da sua eficácia, que será analisada no capítulo 5. A segunda questão é a da sua correcção, que será analisada na secção 4.1.3.

4.1.2 Gestão do protocolo IGMP - Recenseamento dos grupos IP Multicast e da sua filiação

Para gerir os grupos *multicast* e os computadores subscritores dos mesmos, é utilizado o protocolo IGMP de acordo com a sua norma, assumindo o controlador o papel de um *router multicast*. Através do envio periódico de IGMP Queries e da análise dos IGMP Reports, o controlador pode recensear os grupos IP Multicast existentes e os seus subscritores. Adicionamente, esse recenseamento também tem lugar sempre que os computadores emitem por sua iniciativa mensagens IGMP Join e IGMP Leave. Se um subscritor se deixa de manifestar durante um certo intervalo de tempo, isto é, não responde mais às IGMP Queries, o controlador considera que o mesmo deixou de subscrever o grupo.

Em suma, o controlador terá de:

- enviar periodicamente IGMP Queries;
- interceptar os IGMP Membership Report para saber os membros associados aos grupos e actualizar as estruturas de dados sobre grupos;
- interceptar IGMP Join e IGMP Leave e actualizar as estruturas de dados sobre grupos;

- limpar periodicamente as estruturas de dados sobre grupos que detenham informações não actualizadas.

Para clarificar a gestão do IGMP feita pelo módulo, apresenta-se o pseudo-código correspondente, que ilustra igualmente o estilo de programação usada no controlador. Um *thread* (não apresentado) assegura a quarta funcionalidade do módulo.

```
HashMap <group ip, list of devices>  groups;
HashMap <subscriptors ip, timestamp> subscriberLastSeen;

received(PacketIn pi, Switch sw){
  if (pi is an IGMP packet)
    return handleIgmp(pi, sw);
}

handleIgmp(packetIn pi, switch sw){
  if (pi.srcIp != controllerIp){
    device = device such that device.IP = pi.srcIP
    //we will handle this packet
    //since it is a join/leave/report message
    if (pi.type equals report or join){
      groups.add(pi.dstIp, device)
      update subscriberLastSeen
    }else if (pi.type equals leave){
      groups.remove(pi.dstIp, device)
    }
  }
  // allow other modules to process the packet
  return continues;
}

//this method is called by an endless thread
sendQuery(){
each 90 seconds
        create packetOut like a IGMP Report
        with packetOut.src = controllerIp
        and packetOut.dst = allGroupsIP
        write to switch with lower id
}
```

4.1.3 Discussão da correcção da solução

De modo a testar a correcção da solução foi utilizado o simulador Mininet, assim como o *packet sniffer* Wireshark. Através destes dois programas foi possível perceber se o comportamento efectuado pelo controlador era o correcto. Testaram-se ambas as situações, a optimização do ARP e o recenseamento de grupos. As duas soluções apresentaram um comportamento correcto, verificando-se na rede simulada as respostas ARP criadas pelo controlador e o protocolo IGMP em funcionamento. A versão do protocolo IGMP implementada foi, por simplicidade, a versão 2. Optou-se por começar por esta versão, como primeira fase da abordagem à complexidade de uma implementação com o Floodlight. A versão 3, sendo bastante semelhante com a versão 2, introduziria apenas novas estruturas de dados auxiliares.

4.2 IP Multicasting - alteração ao módulo Forwarding

Para optimizar a implementação do IP Multicasting usada pelo Floodlight é necessário evitar tráfego inútil e fazer chegar cada mensagem difundida a cada subscritor pelo caminho mais curto. Dispondo de informação sobre os subscritores de cada grupo e, com base nos pacotes enviados para os mesmos, o controlador, sempre que recebe um pacote IP Multicast com origem no emissor E e dirigido ao grupo G deveria difundir um pacote através de uma árvore de caminhos mais curtos com raiz em E e cobrindo apenas os subscritores de G.

O Floodlight já dispõe de uma forma de calcular uma árvore de cobertura da rede com raiz num dado nó, são as árvore de melhores caminhos (no Floodlight chamadas de BroadcastTrees) já introduzidas no capítulo 3. Para as adaptar ao novo objectivo é necessário podá-las para evitar mensagens inúteis. O algoritmo que realiza esta computação é apresentado em pseudo código a seguir.

```
Algoritmo computeFloodSwitches

Dados
  Seja groups o dicionário exportado pelo módulo
    IPDiscoveryManager, para o recenseamento de membros
  Seja rootSw a raiz da árvore, ou seja, o switch do emissor
  Seja bt a árvore de caminhos mais curtos com raiz em rootSW
  Seja group o endereço IP do grupo
    para onde o emissor quer enviar
  Seja swIn a lista de switches com membros no grupo group

Resultado
  A árvore A podada
```

```
Início
  adiciona-se rootSw a A
  adicionam-se todos os switches com membros de group a swIn

  para cada switch em swIn
      adiciona-se switch a A

      enquanto o switch não é a raiz de bt
          adiciona-se o próximo switch para chegar à raiz a A

Fim
```

Como se observa, este algoritmo calcula um conjunto de *switches* inicializado com a raiz e todos os *switches* com subscritores do grupo. Em seguida acrescenta a esse conjunto todos os *switches* extra que são necessários para chegar de cada *switch* no conjunto inicial até à raiz. Existem três alternativas na implementação do IP Multicast que se irão apresentar nas secções seguintes.

4.2.1 Implementação do IP Multicast através de difusão sem instalação de *flows*

Na primeira implementação o controlador, analogamente à forma como procede com os pacotes de *broadcast*, recebe todos os pacotes IP Multicast. Assim, nesta implementação não são instalados *flows* propositadamente.

O controlador ao receber um pacote IP Multicast no módulo Forwarding, chama a função *doMulticast()*. Nesta função o controlador identifica o emissor, e com base nessa informação, obtém a BroadcastTree com raiz no *switch* desse emissor. Depois, é chamada a função para "podar"a BroadcastTree que se obteve e assim, obter os *switches* que devem fazer inundações e as portas por onde esses pacotes devem chegar. No final desta função, é efectuada uma condição a testar a porta e o *switch* por onde o pacote actual chegou de modo a chamar a função *doFlood()*, caso a condição seja cumprida. A função doFlood é a função que o Floodlight chama por defeito quando se pretende ordenar ao *switch* para fazer uma inundação. É apresentado a seguir o pseudo código da função *doMulticast()* para este caso, ilustrando mais uma vez o estilo de programação do controlador:

```
received(PacketIn pi, Switch sw){
  if(pi is a multicast packet){
    doMulticast(pi, sw)
  }
}
```

```
doMulticast(PacketIn pi, Switch sw){
  Long srcSw = pi.src.srcSw;
  BroadcastTree bt = TopologyManager.getBroadcastTree(srcSw);
  //long is the switch ID and the integer is the inPort
  HashMap<Long, Integer> tree = computeFloodSwitches(bt,
                                          srcSw, pi.Dst);

  if(tree.contains(sw)){
    if(tree.get(sw) == pi.inPort){
      //this is the default flood function
      //used by Floodlight
      doFlood();
    }
  }
}
```

4.2.2 Implementação do IP Multicast através de difusão com instalação de *flows*

Enquanto na primeira implementação não se instala *flows* nos *switches*, a segunda instala *flows* nos *switches* onde deve ser feita a inundação. É utilizada a função de podar a árvore tal como na primeira implementação para se saber quais os *switches* onde se deve fazer a inundação. Assim sendo, esta implementação segue a mesma ordem de execução até ao uso do algoritmo *computeFloodSwitches*. Nesta fase, existe uma constante de controlo chamada *useFlows*, que altera o fluxo da execução. Esta serve para determinar se se pretende utilizar *flows*. Esta variável é definida antes de se executar o Floodlight, no ficheiro de propriedades do mesmo. Caso *useFlows* seja verdadeira, é chamada a função *installDoFloodFlow()*. Com efeito, o pseudo código mostrado anteriormente tem a seguinte condição inserida:

```
doMulticast(PacketIn pi, Switch sw){
        ...
        if(tree.contains(sw)){
            if(useFlows){
                        installDoFloodFlow(tree, pi, sw);
                }else{
                        if(tree.get(sw) == ptk.inPort){
                                //this is the default
                                // flood function
                                // used by Floodlight
```

```
                                    doFlood();
                    }
                }
        }
}
```

A função *installDoFloodFlow()* que é chamada a seguir à condição de *useFlows* ser verdadeira, vai instalar os *flows* nos *switches*. Para este efeito, esta função instancia os *flows* e instala-os de acordo com o pseudo-código que se segue.

```
installDoFloodFlow(HashMap<Long,Integer> switches,
                   PacketIn packetIn, Long sw){
  initiate OFFlowMod
  if(packetIn!=null){
    set match from packetIn
      save match to firstPacketList
  }else{
    set match from firstPacketList
  }
  set idle and hard timeouts
  set command to add flow
  set action to flood
  set wildcards to inPort, dataLinkVLAN, dataLinkSrc,
      dataLinkDst, networkMaskSrc, networkMaskDst
  for each switch in switches{
    if(switch.Id equals sw.Id){
      set match inPort to packetIn inPort
    }else{
      set match inPort to switches.get(switch)
      send flow to switch
        }
  send packetOut to sender switch
  }
}
```

Como se observa no pseudo-código, existe uma lista de *packetIn* que representa os pacotes recebidos pelo controlador dos emissores. Esta lista é importante para tratar a problemática dos *flows* estarem activos e existirem alterações nos grupos. Ou seja, caso um membro entre ou saia no grupo, é importante que este comece a receber o tráfego ou que sejam desinstalados *flows* inúteis. Esta situação ocorre, porque os *timeouts* dos *flows* no Floodlight são *idle timeouts*, ou por outras palavras, enquanto passa tráfego nestes

flows, eles não expiram. Esta situação dá origem a que se um emissor estiver sempre a enviar tráfego, só o grupo que começou a receber o tráfego o irá receber e mais nenhum membro que entre. Por outro lado, se um membro sair do grupo estará um *flow* instalado desnecessariamente (a não ser que nesse *switch* exista outro membro).

Para resolver esta problemática, para além de se guardar o primeiro pacote de um emissor, guarda-se as alterações de um grupo quando chega um pacote IGMP. As alterações são exportadas pelo módulo IPDiscoveryManager de modo a que o módulo Forwarding tenha acesso a elas. Assim, quando chega um pacote IGMP ao módulo Forwarding e este reconhece que existiram mudanças num grupo, é chamada a função *modifyFlows()*. Esta função é responsável por calcular a diferença das árvores entre a árvore com a alteração e a árvore sem a alteração do grupo. Através deste calculo de diferenças de árvores é possível descobrir quais os *flows* que se devem instalar e quais os que se devem remover. É apresentado em seguida o pseudo-código para a problemática das alterações de um grupo:

```
received(PacketIn pi, Switch sw){
  //if getChanges is 0, no changes
  //if getChanges is 1, new member
  //if getChanges is 2, member removed
  if(pi is an IGMP packet AND IPDiscoveryManager.getChanges>0)
    modifyFlows(pi, sw);
}

modifyFlows(PacketIn pkt, Switch sw){
  //we iterate over all senders of member group
  for each sender
    Long senderSw = sender.getSw;
    BroadcastTree bt =
      TopologyManager.getBroadcastTree(senderSw);
    //added parameter to computeFloodSwitches
    //to know the switch that must be added or removed
    HashMap<Long, Integer> List1=
      computeFloodSwitches(bt, senderSw, pkt.Dst, sw);
    HashMap<Long, Integer> List2=
      computeFloodSwitches(bt, senderSw, pkt.Dst, 0);

    //this function do the diff between trees and return
    //the switches to remove flows
    HashMap<Long,Integer> toRemove =
      getUnnecessarySw(ListSw1, ListSw2);
    //this functions do the diff between trees and
```

```
    //return the switchs to install flows
    HashMap<Long,Integer> toInstall =
      getNewSw(ListSw1, ListSw2);

    pkt = firstPacketList.get(sender);
    OFMatch match = match from pkt;

    for each entry in toRemove
      clearFlows(toRemove, match);

    for each entry in toInstall
      installDoFloodFlow(toInstall, null, srcSw);

}
```

A função *modifyFlows()* também poderia ser utilizada em tratamento de falhas como será discutido mais à frente.

4.2.3 Implementação do IP Multicast por encaminhamento multi-porta

Finalmente, existe uma possível terceira implementação do IP Multicast encaminhando um pacote para várias portas. Nesta abordagem, o controlador indica no pacote a ser encaminhado, mais que uma porta de envio. Desta forma, substitui-se a inundação efectuada pelas duas primeiras implementações, por um encaminhamento multi-porta.

Para desenvolver esta solução, o Floodlight já contém uma função no módulo Forwarding que efectua este encaminhamento multi-porta. Esta função recebe por parâmetro um pacote, um identificador de um *switch* e um array de portas de forma a possibilitar o encaminhamento multi-porta.

Todavia, tanto os *switches* como os simuladores não garantem esta funcionalidade de encaminhamento multi-porta, daí ser uma implementação para trabalho futuro como referido no capítulo 6.

4.2.4 Discussão da correcção da solução

De modo a testar a correcção da solução foi utilizado o simulador Mininet, o *packet sniffer* Wireshark e o programa sock. Através do programa sock foi possível instanciar *hosts* como subscritores e emissores *multicast*. Assim, utilizou-se o Mininet para simular a rede OpenFlow e executar nos *hosts* o programa sock. O Wireshark foi utilizado para fins de *debug* de forma a compreender o comportamento dos computadores e do controlador.

Através deste conjunto de programas foi possível confirmar-se que a solução era correcta. Observou-se sempre os pacotes *multicast* a serem encaminhados devidamente pela árvore podada (através do Wireshark).

4.2.5 Discussão - Concorrência e Tratamento de falhas

O Floodlight trata cada mensagem recebida por um *switch* de um modo sequencial por uma única thread. Depois de ser processada por cada *listener* dos módulos, a próxima mensagem recebida é processada e assim sucessivamente. Por outro lado, quando são recebidas mensagens no controlador vindas de *switches* diferentes, é feito um tratamento potencialmente paralelo, através de várias *threads*. Assim sendo, os serviços exportados pelos módulos, e as variáveis partilhadas, são executados em concorrência.

Um aspecto a ter em consideração é que os canais do Floodlight são assíncronos, e, assim sendo, o envio das mensagens do controlador para os *switches* não é bloqueante. Na segunda implementação apresentada com *flows*, na secção 4.2.2, podem ocorrer *race conditions* devido à escrita nos canais. Depois do controlador escrever os *flows* para os *switches* nesta implementação, o pacote é enviado para o *switch* que enviou a mensagem para o controlador. Esta situação não introduz necessariamente incorrecções, mas pode introduzir processamento extra e inútil no controlador e atrasos no encaminhamento. Este problema é delicado e necessita de um tratamento mais aprofundado. Uma possível forma de o minorar, corresponde a impor uma ordem estrita de instalação dos *flows* pelos diferentes *switches*, começando sempre a instalá-los nos *switches* mais próximos da raiz. Outra alternativa é utilizando as mensagens BARRIER, introduzidas na tabela 3.2, que permitem assegurar a ordem de entrega das mensagens nos *switches*.

Se existirem problemas com os *switches* ou com os canais da rede, o Floodlight dispõe de mecanismos para os detectar. Adicionalmente, o controlador permite que os módulos subscrevam eventos para que seja possível recuperar das falhas. Na implementação do IP Multicasting, é necessário acrescentar o tratamento das falhas, subscrevendo os correspondentes eventos de notificação e reconfigurando todas as árvores afectadas.

Tal como para a detecção de falhas, o Floodlight tem mecanismos para identificar se um computador se moveu na rede. O processamento desses eventos passaria por duas situações: 1) a de um computador subscritor de um grupo se mover e 2) a de um computador emissor se mover. Para a primeira, é necessário, analogamente ao que acontece na mudança da *membership* de um grupo, calcular a diferença entre a árvore antiga e a nova e proceder de forma idêntica. Na segunda situação, o tratamento mais simples consistiria em desinstalar a antiga árvore e instalar a nova.

5

Avaliação

5.1 Optimização do ARP com IP packet snooping

A optimização do ARP com IP packet snooping evita algumas difusões numa rede que utiliza uma aproximação SDN em comparação com uma rede IP tradicional. Esta optimização utiliza uma cache partilhada no controlador de modo a que quando o controlador recebe um pacote ARP Request, avalie se pode responder ou se precisa de difundi-lo. O controlador para responder ao ARP Request, tem de ter identificado actividade há menos de 60 segundos (ou há menos de TRUST INTERVAL, constante introduzida na sub secção 4.1.1) no computador de que se pretende saber o IP. Caso esta condição seja satisfeita, o controlador cria um pacote ARP Reply, responde directamente pela porta do *switch* por onde chegou o pacote ARP Request e finalmente faz *drop* deste.

De modo a avaliar a optimização do ARP com IP packet snooping criou-se um *script* em AWK. A linguagem AWK é uma linguagem que permite desenvolver facilmente programas de extracção de dados de logs. O objectivo deste *script* é ler um trace tcpdump de uma rede tradicional e calcular quantos pacotes ARP difundidos são evitados.

Assim, analisaram-se os cabeçalhos dos pacotes, identificando exclusivamente 2 tipos de tráfego: IP e ARP. No caso de ser tráfego IP ou ARP Reply, o simulador guarda o IP que passou a conhecer e o *timestamp* actual. Caso o tráfego seja ARP Requests, o simulador vai verificar se existe o IP no dicionário e caso não exista, incrementa o número de difusões pois é essa a acção que o controlador faz. Caso o IP que um computador pretenda saber exista no dicionário, faz-se a diferença do tempo actual para o *timestamp* guardado e se for superior a TRUST INTERVAL incrementa-se o número de difusões. Faz-se a operação inversa que o controlador faz porque o simulador guarda o número de difusões. Assim, quando a diferença é inferior a TRUST INTERVAL o simulador não incrementa o número

Hora	TI = 0	TI = 60	TI = 1200	Ganho TI = 60	Ganho TI = 1200
1	4872	1137	584	76,66%	88,01%
2	5802	1495	560	74,23%	90,35%
3	5575	2323	1409	58,33%	74,73&
4	6534	2443	1030	62,61%	84,24%
5	7205	1845	451	74,39%	93,74%
6	5142	992	395	80,71&	92,32%
7	228	56	5	75,44%	97,81%
Total	35358	10291	4434	70,89%	87,46%

Tabela 5.1: Número de ARP Requests por hora (variando TI) e % de redução dos mesmos pela cache do controlador

de difusões.

O *trace* analisado foi recolhido pela Divisão de Informática da FCT-UNL. Na tabela 5.1 apresenta-se o número de ARP Requests existentes na rede e o número de ARP Requests que existiriam na mesma rede se se utilizasse o controlador com a optimização desenvolvida. A tabela tem uma linha por cada hora de tráfego recolhido e apresenta três hipóteses. No primeiro caso é apresentado o número de ARP Requests total capturados durante essa hora. No segundo caso é apresentado o número de ARP Requests que existiriam na mesma rede durante essa hora com TRUST INTERVAL (TI) = 60 segundos, e no terceiro caso com TRUST INTERVAL (TI) = 1200. Em cada caso está também indicado o ganho conseguido através da indicação da percentagem de difusões evitadas.

Como se pode observar existe redução significativa das difusões de ARP Requests. O ensaio com TI = 1200 adveio da hipótese de na rede existirem computadores que usem Gratuitous ARP. Os pacotes Gratuitous ARP utilizados em alguns sistemas operativos, nomeadamente no Mac OS, têm o objectivo de informar a rede dos seus endereços MAC e IP, sendo enviados quando a sua interface fica activa. Com efeito, o uso de Gratuitous ARP numa rede que utiliza a aproximação SDN, poderia efectivamente dar a conhecer ao controlador a visão completa e actualizada dessa rede. Ou seja, cada vez que um computador entrasse na rede, o controlador ficaria a conhecer a sua localização na rede não tendo de esperar que este começasse a enviar tráfego. Desta forma, o ensaio TI = 1200 utiliza um valor mais elevado do TI para simular uma cache de ARP partilhada com uma visão mais completa e actualizada da rede.

5.2 Optimização do IP Multicast com topologia centralizada e gestão de grupos

A optimização do IP Multicast com topologia centralizada e gestão de grupos foi a parte do trabalho implementada para adicionar a funcionalidade do IP Multicast ao controlador. Para a gestão dos grupos usou-se o protocolo IGMP como base para esta gestão, fazendo com que os computadores e controlador comunicassem. Esta gestão é semelhante à gestão utilizada pelos *routers* com IP Multicast activo.

A optimização do IP Multicast baseia-se na informação obtida pela gestão de grupos do controlador e em árvores de caminhos mais curtos. O controlador já dispõe de árvores de caminhos mais curtos de um determinado *switch* para todos os outros e são estas árvores que se utilizam. A diferença está em que estas árvore são podadas com um algoritmo que informa o controlador quais são os caminhos necessários para não existir tráfego inútil.

Realizaram-se vários testes usando um conjunto de *scripts* escritos em Python. Estes *scripts* controlavam a execução de programas nos computadores ligados à rede simulada. Deste modo, foi possível criar *scripts* que simulavam redes OpenFlow usando o Mininet, e controlavam o fluxo de comandos que os computadores no simulador executavam. Dois comandos foram usados intensivamente, o programa sock e o programa tcpdump. O programa sock é um programa da autoria de Richard Stevens, que contém um grande leque de opções para demonstrar e testar as funcionalidades do TCP/IP. Através deste programa é possível instanciar emissores e receptores *multicast* muito facilmente. Deste modo, criaram-se *scripts* para simular uma rede *OpenFlow* e ordenar que os computadores dessa rede executem o programa sock. Assim, foi possível simular uma rede com grupos IP Multicast em funcionamento e testar se o comportamento da mesma era o esperado, analizando os *traces* de pacotes recolhidos em diversos pontos da rede através de tcpdump.

Como nesta fase, não se pretendia avaliar o desempenho da solução, mas se a mesma era correcta, os testes realizados foram sobretudo usados para verificar a correcção da implementação realizada no Floodlight. No entanto, para submeter a implementação a um teste mais significativo, foi também realizado um teste com 7 *switches* e 8 computadores que aleatoriamente entravam e saiam de diversos grupos IP Multicast e, também aleatoriamente, iniciavam a transmissão de fluxos de pacotes para os grupos. Este teste, mais abrangente, permitiu, através de análise detalhada, verificar o comportamento do controlador e do controlo por este exercido sobre a rede. A topologia da rede de testes é ilustrada na figura 5.1.

Apresenta-se em seguida o script utilizado para os testes, ilustrando o estilo de programação utilizado para configurar a rede com o Mininet, programado em Python.

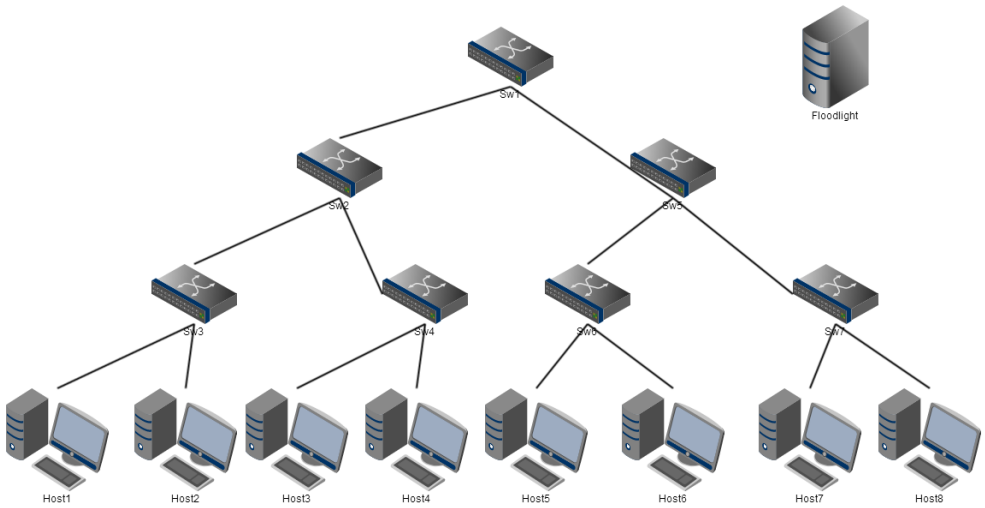


Figura 5.1: Topologia da rede de testes

```
from mininet.cli import CLI
from mininet.log import lg, info
from mininet.net import Mininet
from mininet.node import OVSKernelSwitch, RemoteController
from mininet.topolib import TreeTopo
from random import choice
from time import sleep
from functools import partial

def multicastTest( net ):
    option = ['join','leave','send']
    hosts = net.hosts
    dic = {'h1':0, 'h2':0, 'h3':0, 'h4':0, 'h5':0, 'h6':0, 'h7':0, 'h8':0}
    info( "*** Initializing\n" )
    sleep(10)
    for x in range(0,3):
            for host in hosts:
                op = choice(option)
                if op == 'join':
                        host.cmd('sock -u -s -j 224.2.2.2 1500 &')
                        pid = host.cmd('echo $!')
                        dic[host.name] = pid
                        info("*** " + host.name + " joining group - pid "
                                                + pid)
                elif op == 'leave':
                        if dic[host.name] != 0:
                                host.cmd('killall sock &')
#                                host.cmd('kill -9 ' + dic[host.name] + ' &')
                                info("*** " + host.name + " leaving group\n")
                elif op == 'send':
```

```
                        host.cmd('sock -u -i -t 32 -w 80
                                        -n 1 224.2.2.2 1500 &')
                        info("*** " + host.name + " sending\n")
                    sleep(10)
            sleep(15)
if __name__ == '__main__':
    lg.setLogLevel( 'info' )
    info( "*** Initializing Mininet and kernel modules\n" )
    OVSKernelSwitch.setup()
    info( "*** Creating network\n" )
    network = Mininet( TreeTopo( depth=3, fanout=2 ),
                       switch=OVSKernelSwitch,
                       controller=partial(RemoteController,
                       defaultIP='192.168.56.1'))
    info( "*** Starting network\n" )
    network.start()
    info( "*** Running multicast test\n" )
    multicastTest( network )
    info( "*** Starting CLI (type 'exit' to exit)\n" )
    CLI( network )
    info( "*** Stopping network\n" )
    network.stop()
```

Apesar de não ter sido desenvolvida uma avaliação rigorosa do desempenho, o trabalho desenvolvido e os testes realizados, permitiram verificar as hipóteses iniciais, isto é, uma vez vencida a barreira de penetrar na arquitectura do controlador, do seu ambiente de programação e testes, e dos detalhes de implementação do OpenFlow e da aproximação SDN, os algoritmos de controlo da rede são certamente mais simples do que a sua versão completamente distribuída.

6

Trabalho Futuro

O trabalho futuro mais imediato decorre directamente dos pontos discutidos na sub secção 4.2.5 nomeadamente os problemas de concorrência e de tratamento de falhas e da implementação multi-porta discutida na sub secção 4.2.3. No entanto, outro ponto que não foi apresentado anteriormente na discussão, porque se prevê que uma rede empresarial não apresente grande número de grupos mas é importante referir: o número de *flows* nos *switches*. É necessário ter em consideração este ponto quando se desenvolve uma solução baseada na aproximação SDN porque os *switches* têm número limitado de entradas para *flows*. Para a implementação apresentada de IP Multicast, a redução de instalações de *flows* passaria por utilizar em determinados grupos árvores partilhadas. Assim é possível reduzir os *flows* instalados de um por emissor por grupo, para um por grupo. Imagine-se numa grande rede existirem 100 emissores em 100 grupos. Existiriam 10000 árvores instaladas na rede. Este número é comportável por algum *switch* actual e provavelmente pela maioria no futuro. O grande problema levantado relaciona-se com o dinamismo destes 100 grupos.

O trabalho futuro passa também por passar a implementação para produção e teste numa rede real. Este só é possível com *switches OpenFlow*. Neste contexto será também necessário proceder à investigação e desenvolvimentos suplementares de algumas das desvantagens introduzidas pela aproximação SDN, das quais:

- os problemas de escalabilidade apresentados pela aproximação SDN[YTG13];
- os problemas relacionados com avarias do controlador;
- os problemas de segurança, particularmente relacionados com ataques ao controlador.

Todos estes problemas são objectos de investigação da aproximação SDN. Espera-se que os controladores no futuro fiquem mais robustos depois de introduzirem as respectivas soluções a estes problemas.

7

Conclusões

Nesta dissertação foram desenvolvidas várias soluções para o encaminhamento numa rede IP. Depois de efectuados os testes e feita a avaliação apresentada no Capítulo 5, concluí-se que os objectivos principais foram cumpridos. Esses objectivos eram:

- Análise crítica da gestão de uma rede IP baseada na aproximação SDN;
- Implementação optimizada de IP Multicasting baseada numa aproximação SDN.

Para além destes objectivos, foram realizados vários trabalhos para optimizar o funcionamento da rede, nomeadamente através da implementação da optimização do protocolo ARP.

No início dos trabalhos para se elaborar esta dissertação começou-se por ler a literatura sobre a aproximação SDN e fazer uma análise crítica desta. De realçar que nesta literatura não eram referidas as implementações de IP Multicast. Assim, iniciou-se a dissertação com o objectivo de criar uma implementação que abrangesse esta funcionalidade das redes IP.

Em paralelo a este estudo geral da aproximação SDN e do protocolo *OpenFlow* , foi necessário ter uma aproximação às ferramentas que se iriam usar, de modo a tentar escolher as mais adequadas à realização da dissertação. Nesta escolha houve uma grande indecisão porque não se conheciam bem as ferramentas nem se iria ter tempo para as testar todas. O Mininet foi a escolha mais fácil por ser o simulador de redes baseadas na aproximação SDN mais popular. Por outro lado, a escolha do controlador foi bastante difícil porque não existia uma comparação entre os controladores existentes, nomeadamente em relação à sua curva de aprendizagem. Foi escolhido o Floodlight mas esta escolha revelou-se arriscada porque se concluiu que este teve uma curva de aprendizagem bastante elevada.

O Floodlight apresentou uma curva de aprendizagem elevada mas afirmou-se como sendo talvez o controlador mais completo da aproximação SDN. Depois de um grande estudo a este, concluiu-se que este teria mecanismos que poderiam ajudar ao desenvolvimento da solução final. Destes mecanismos destacam-se as BroadcastTrees que o Floodlight já processava através do algoritmo de Dijkstra.

Seguiu-se a implementação da optimização do ARP com várias versões. Acabou-se por confirmar que a solução adoptada foi a mais acertada. Esta baseou-se numa cache partilhada que respondia a ARP Requests caso as informações detidas fossem válidas.

Em paralelo com a optimização do ARP, foi implementado o protocolo IGMP no controlador para futuramente servir de base ao IP Multicast. O IGMP serve como o motor para a gestão de grupos multicast e assim determinou-se que seria necessário este, para dar suporte ao IP Multicast. Ambos os desenvolvimentos vieram a integrar um novo módulo, o IPDiscoveryManager.

Concluído o módulo IPDiscoveryManager passou-se para as alterações ao módulo Forwarding onde iriam ser computadas as árvores de cobertura do IP Multicast e instalados os *flows*. Iniciou-se a implementação, seguindo uma solução sem instalação de *flows* análoga à presente no Floodlight para o *broadcast*, mas sem tráfego inútil. Assim, esta implementação deu origem à base da implementação final com *flows*, onde a grande diferença estava no tratamento de alterações na *membership* dos grupos. Este era um aspecto muito importante, porque era necessário não existir tráfego inútil quando saiam membros e por outro lado, era necessário enviar imediatamente tráfego a novos membros. O melhor modo de contornar este aspecto foi fazendo alterações aos *flows* com a recepção de tráfego IGMP com alterações da *membership*. Finalmente foi necessário bastante *debug* para testar estas soluções, usando a API do Mininet, o Wireshark e o próprio Floodlight para verificar se o fluxo de execução era o pretendido.

No final, foram realizados simulações para avaliar o trabalho realizado. A primeira serviu para determinar os ganhos do uso do controlador com a optimização do ARP em comparação a uma rede tradicional. Esta foi desenvolvida em AWK e usou-se traces tcpdump de uma rede real para se conseguir avaliar a eficácia da solução proposta. O teste da implementação IP Multicast foi mais difícil tendo havido necessidade de recorrer ao Mininet, a traces recolhidos por tcpdump e ao programa sock para gerar tráfego. Os testes foram automatizados utilizando *scripts* desenvolvidos em Python.

O trabalho realizado permite tirar várias conclusões. As primeiras estão relacionadas com a extensibilidade e flexibilidade da aproximação SDN. Primeiro, o próprio controlador Floodlight (assim como todos os outros) é o testemunho vivo de que através desta aproximação é fácil modificar o encaminhamento *unicast* passando a usar encaminhamento pelo melhor caminho numa rede de *switches* com configuração arbitrária.

O trabalho realizado mostra adicionalmente como foi possível de forma relativamente simples e com base em algoritmos centralizados: 1) substituir a utilização de difusão pelo uso de um directório centralizado no protocolo ARP, 2) introduzir uma gestão centralizada da filiação de grupos IP Multicasting com base também num directório de grupos

e 3) introduzir encaminhamento IP Multicasting com base em árvores de cobertura optimizadas para cada emissor.

As duas últimas implementações constituem uma ilustração particularmente feliz das vantagens potenciais do paradigma SDN na medida em que realizam encaminhamento com qualidade equivalente à da utilização simultânea de IGMP e PIM-DM com uma implementação baseada em algoritmos centralizados e sem todo o *overhead* de controlo que o PIM-DM introduz (inundações periódicas particularmente ineficazes com uma distribuição pouco densa de subscritores). A complexidade do novo *control plane* é relativamente baixa, ao mesmo tempo que os *switches* apenas mantêm entradas de encaminhamento em todo equivalentes a às entradas banalizadas que já utilizam para o encaminhamento IP Unicasting. Os actuais *switches* que suportam *OpenFlow* utilizam circuitos VLSI verticais (*merchant silicon*) que implementam tabelas de *flows* com várias dezenas de milhar de entradas e o seu preço deverá baixar com a adopção generalizada desta aproximação.

É no entanto necessário ter presente que as vantagens apresentadas são potenciais se não forem encontradas soluções para os novos problemas introduzidos pela aproximação SDN discutidos no capítulo 6. Finalmente, é necessário ter presente que a aproximação SDN introduz um novo ponto a necessitar de normalização e coordenação para evitar a dependência de soluções proprietárias. Tal dependência irá deslocar-se para o software dos controladores, o qual é ainda objecto de desenvolvimento e investigação.

Bibliografia

[CFP+07] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, e Scott Shenker. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, Agosto 2007.

[CGA+06] M. Casado, T. Garfinkel, A. Akella, M.J. Freedman, D. Boneh, N. McKeown, e S. Shenker. Sane: A protection architecture for enterprise networks. In *USENIX Security Symposium*, 2006.

[Eri] David Erickson. Beacon a java-based openflow controller - https://openflow.stanford.edu/display/beacon/home.

[FHF+11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, e David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pág. 279–291, New York, NY, USA, 2011. ACM.

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, e Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, Julho 2008.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, e Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pág. 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[LHM10] Bob Lantz, Brandon Heller, e Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pág. 19:1–19:6, New York, NY, USA, 2010. ACM.

[Lim12] Thomas A. Limoncelli. Openflow: A radical new idea in networking. *Queue*, 10(6):40:40–40:46, Junho 2012.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, e Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Março 2008.

[Neta] Netfpga: Programmable networking hardware - http://netfpga.org.

[Netb] Big Switch Networks. Floodlight an open sdn controller - http://floodlight.openflowhub.org.

[NHS12] Yukihiro Nakagawa, Kazuki Hyoudou, e Takeshi Shimizu. A management method of IP multicast in overlay networks using openflow. In *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12*, pág. 91, New York, New York, USA, 2012. ACM Press.

[TCD+07] Jonathan S. Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson, Charles Wiseman, e David Zar. Supercharging planetlab: a high performance, multi-application, overlay network platform. *SIGCOMM Comput. Commun. Rev.*, 37(4):85–96, Agosto 2007.

[VKF12] Andreas Voellmy, Hyojoon Kim, e Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pág. 43–48, New York, NY, USA, 2012. ACM.

[VN11] S.J. Vaughan-Nichols. Openflow: The next generation of the network? *Computer*, 44(8):13 –15, aug. 2011.

[YTG13] Soheil Hassas Yeganeh, Amin Tootoonchian, e Yashar Ganjali. On scalability of software-defined networking. *Communications Magazine, IEEE*, 51(2):136–141, 2013.