TRABAJO FIN DE GRADO

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Evaluación del protocolo multicast PIM-SSM en entornos SDN

Autor
Pablo Góngora Luque

Directores
Juan Manuel López Soler
Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Septiembre de 2017

Evaluación del protocolo multicast PIM-SSM en entornos SDN

Pablo Góngora Luque

Palabras clave: SDN, Multicast, evaluación

Resumen

El aumento del tráfico en la red y complejidad de las aplicaciones en los últimos años ha tenido como consecuencia el estudio de nuevas tecnologías de redes. Este conjunto de implementaciones nuevas se engloba en el término 5G. En el ámbito de las telecomunicaciones una de las características más importantes de este nuevo desarrollo es el concepto de Software-Defined Network (SDN) que consiste en separar en diferentes capas las distintas funcionalidades de una red.

SDN consiste en redes más flexibles, escalables y programables que las actuales gracias a la separación del plano de datos del plano de control. Mientras que el plano de datos está formado por hardware simple con implementaciones software básicas, por otro lado se tiene el plano de control constituido por un controlador que se encarga de la monitorización, control y configuración del plano de datos. Esto permite realizar modificaciones sobre la topología de red o sobre los servicios implementados en el sistema con costes muy bajos en comparación con las redes actuales.

Aunque existen varias tecnologías de *5G* el siguiente proyecto se centra en SDN gracias a su arquitectura centralizada y a las ventajas que presenta a la hora de implementar servicios que necesitan transmitir tramas multicast. Se va a profundizar en el concepto de multicast, ya que debido a los servicios de *streaming* cada vez más utilizados actualmente, cada vez se está utilizando más este tipo de transmisiones. Estos servicios cada vez más demandados y con necesidades más exigentes día a día generan gran cantidad de tráfico en la red de tal forma que ya se están produciendo congestionamientos. Uno de los objetivos principales de la tecnología SDN es reducir el tráfico generado por multicast y este proyecto se va a centrar en las ventajas de utilizar SDN para transmisiones multicast frente a las tecnologías actuales utilizadas.

Para realizar un estudio objetivo sobre los beneficios que presenta estas nuevas arquitecturas se va a realizar una comparación de diferentes entornos con implementaciones de servicios multicast. Una de las desventajas que presenta el modelo SDN es su vulnerabilidad al ser una red centralizada por lo que también se observa la robustez ante fallos en la red, es decir, el tiempo de respuesta que presenta el controlador ante un imprevisto comparado con la eficiencia de los mecanismos actuales de encaminamiento dinámico.

El presente proyecto toma como punto de partida el Trabajo Fin de Grado de Carlos Santamaría Espinosa, realizado en el curso 2015/16 [47] y

parte de dos implementaciones realizadas con las herramientas *Quagga* [13] y *OpenDaylight* [19]. El objetivo es evaluar el impacto que puede tener la adopción de la tecnología de red SDN en la distribución de información en entornos multicast. Para ello se ha desarrollado una metodología de evaluación, incluyendo las herramientas necesarias para realizar las medidas necesarias. Tras los resultados obtenidos se puede concluir que la utilización de SDN para la transmisión de tramas multicast ofrece unas mejoras muy significativas en cuanto a escalabilidad y tráfico de señalización, ya que gracias a la centralización que ofrece SDN se consigue disminuir considerablemente el tráfico de señalización y, a su vez, para redes cada vez más grandes el tráfico generado disminuye también, comparando un diseño de red actual con SDN.

Evaluation of the PIM-SSM multicast protocol in SDN environments

Pablo Góngora Luque

Keywords: SDN, Multicast, evaluation

Abstract

The increase in network traffic and application complexity in recent years has resulted in the study of new network technologies. This set of new implementations is encompassed within the 5G term. In the field of telecommunications one of the most important features of this new development is the concept of SDN that consists of separating the different functionalities of a network into different layers.

SDN consists of more flexible, scalable and programmable networks than the present ones thanks to the separation of the plane of data of the control plane. While the data plane consists of simple hardware with basic software implementations, on the other hand the control plane is constituted by a controller that is in charge of the monitoring, control and configuration of the data plane. This allows modifications to the network topology or services implemented in the system with very low costs compared to the current networks.

Although there are several technologies of *5G* the following project focuses on SDN thanks to its centralized architecture and the advantages that it presents in the implementation of services that need to transmit multicast frames. The concept of multicast is going to be deepened, since with the services of *streaming* increasingly used at the moment, this type of transmissions is being used more and more. These services increasingly demanded and with more demanding necessities day by day generate great amount of traffic in the network of such form that already are being congested. One of the main goals of the SDN technology is to reduce the traffic generated by multicast and this project will focus on the advantages of using SDN for multicast transmissions versus the current technologies used.

To perform an objective study on the benefits of these new architectures, a comparison of different environments with multicast service implementations will be made. One of the disadvantages presented by the SDN model is its vulnerability to being a centralized network. Therefore, it is also possible to observe robustness to network failures, that is to say, the response time presented by the controller to an unforeseen event compared to the efficiency of The current mechanisms of dynamic routing.

The present project takes as its starting point the Work End of Degree of Carlos Santamaría Espinosa, realized in the course 2015/16 [47] and part

of two implementations realized with the tools *Quagga* [13] and *OpenDaylight* [19]. The objective is to evaluate the impact that the adoption of the technology of network SDN in the distribution of information in multicast environments can have. For this, an evaluation methodology has been developed, including the necessary tools to carry out the necessary measures. After the results obtained it can be concluded that the use of SDN for the transmission of multicast frames offers very significant improvements in scaling and signaling traffic, since thanks to the centralization offered by SDN it is possible to considerably reduce the Signaling traffic and, in turn, for increasingly larger networks the generated traffic also decreases, comparing a current network design with SDN.

Yo, **Pablo Góngora Luque**, alumno de la titulación de la **Grado en Ingeniería de Tecnologías de Telecomunicación**, con DNI XXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Pablo Góngora Luque

Granada a 12 de Septiembre de 2017.

D. **Juan Manuel López Soler**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

D. **Jorge Navarro Ortíz**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Evaluación del protocolo multicast PIM-SSM en entornos SDN***, ha sido realizado bajo su supervisión por **Pablo Góngora Luque**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 12 de Septiembre de 2017.

Los directores:

Juan Manuel López Soler **Jorge Navarro Ortiz**

Agradecimientos

En primer lugar me gustaría agradecer a mi familia y a Laura por haberme ayudado a conseguir llegar hasta el fin. Ha sido un largo camino lleno de dificultades donde he recibido un montón de apoyo por parte de todos. A mis padres por haberme dado ánimos siempre y haberme llevado todos los veranos a cualquier sitio donde pudiese estudiar sin ruido, a mi hermana por haberme dado ánimos antes de cada examen y a Laura, mi compañera de viaje estos cuatro años, por haberme ayudado a estudiar ante cualquier situación y haberme enseñado lo que realmente importa.

También agradecer a todos mis compañeros de clase por haberme ayudado de una manera o de otra a disfrutar de estos fantásticos años de carrera. Desde prestarme apuntes en verano hasta salir a tomar algo y pasar un buen rato después de tantas horas de estudio.

A todos los profesores de la facultad por haberme enseñado tantas cosas y haberme ayudado tanto siempre que lo he necesitado. En especial a mis tutores Juan Manuel y Jorge por haberme ayudado a realizar el TFG; y a Pablo Padilla por haberme recibido tan amablemente en cualquier tutoría que he necesitado para en las diferentes asignaturas.

Índice general

Índice de figuras

Índice de tablas

Lista de acrónimos

SDN Software-Defined Network

UDP User Datagram Protocol

MIMO Multiple-input Multiple-output

OFDM Orthogonal Frequency-Division Multiple Access

QoS Quality of Service

IoT Internet of Things

NOMA Non Orthogonal Multiple Access

FBMC Filter Bank Multi Carrier

SCMA Sparse Code Multiple Access

UF-OFDM Universal Filtered OFDM

IPTV Internet Protocol Television

NFV Network Function Virtualization

API Application Programming Interface

SSM Source-specific multicast

PIM Protocol Independent multicast

RP Rendevous point

NAT Network Address Translation

MPLS Multiprotocol Label Switching

OSPF Open Shortest Path First

RIP Routing Information Protocol

BGP Border Gateway Protocol

ADSL Asymmetric Digital Subscriber Line

IS-IS Intermediate system to intermediate system

PIM-SM PIM Sparse Mode

SSL Secure Sockets Layer

TLS Transport Layer Security

PCEP Path Computation Element Communication Protocol

NETCONF Network Configuration Protocol

RESTCONF Rest Configuration Protocol

XMPP Extensible Messaging and Presence Protocol

I2RS Interface to the Routing System

BGP Border Gateway Protocol

API Application Programming Interface

Capítulo 1

Introducción

En este primer capítulo se introducen todos los aspectos que han motivado la realización de este proyecto. Primero se hablará sobre la motivación de implementar nuevos modelos de redes que satisfagan los requisitos de los nuevos servicios; con necesidades de prestaciones cada vez mayores. Después se hablará sobre los diferentes tipos de soluciones que se han propuesto para implementar nuevas redes denominadas como 5G.

Finalmente, para facilitar la lectura en el último apartado se proporciona un resumen con la estructura de la documentación generada en la realización del proyecto.

1.1. Contexto y motivación

En este primer apartado se explica la importancia del estudio de nuevas tecnologías para el diseño de nuevas redes. Para ello, se identifican las limitaciones de las redes actuales, y posteriormente se enumeran los requisitos que se han definido para las futuras redes 5G, marco en el que se desarolla este proyecto. Finalmente se comentan las principales características del tráfico generado en los últimos años y sus tendencias.

1.1.1. Limitaciones de las redes actuales

En la actualidad, los dos tipos de redes dominantes para facilitar conectividad y acceso a usuarios finales y empresas son, por un lado, la red móvil 4G, que puede alcanzar velocidades superiores a los 300 Mbit/s con un ancho de banda de 8 MHz usando esquemas tales como Multiple-input Multiple-output (MIMO) y Orthogonal Frequency-Division Multiple Access (OFDM). Y por otro lado, tecnologías de acceso cableado donde Asymmetric Digital Subscriber Line (ADSL) (o sus variantes), sobre fibra y/o cobre, con Giga-

bit Ethernet y Multiprotocol Label Switching (MPLS) son las tecnologías dominantes.

Estas redes están especificadas mediante una inmensa cantidad de protocolos estandarizados. Para facilitar su interconexión, ambas usan hardware genérico como pueden ser switches, routers, etc junto con hardware de propósto específico (típicamente en la red móvil) para dar servicios de movilidad, tarificación, *roaming*, etc.

A pesar del caso de éxito que ha sido y está siendo Internet, hay una serie de inconvenientes, producidos por el aumento del tráfico de datos y por la provisión de servicios con requisitos cada vez más exigente [18], que dificultan su evolución a medio y largo plazo. Las limitaciones más importantes son:

- **Complejidad:** los protocolos cada vez se crean más complejos para poder proporcionar servicios más seguros y eficientes. Además, los protocolos se generan de forma aislada de manera que cada protocolo resuelve un problema específico sin aprovechar las ventajas de un diseño de protocolos más abstracto que dé lugar a una reutilización y en consecuencia a una menor complejidad.
- **Políticas inconsistentes:** Para implementar políticas que abarquen toda la red, los administradores deben configurar miles de mecanismos y aparatos. Dando lugar a la inviabilidad de añadir nuevos dispositivos. La complejidad de las redes actuales hace muy complicado implementar políticas de acceso, seguridad y QoS consistentes.
- **Imposibilidad de escalabilidad:** al aumento de centros de datos está causando que cada vez la red sea más compleja dando lugar a la necesidad de configurar y gestionar miles de dispositivos, lo que dificulta el escalado de la red. Esta dificultad se hará todavía más patente con la irrupción de un número masivo (varios órdenes de magnitud superior al actual) de dispositivos finales, como se prevé en el despliegue de la futura Internet of Things (IoT).
- **Dependencia del vendedor:** las nuevas capacidades y servicios perseguidas por los proveedores y empresas se ven frenados por los ciclos de producción de los equipamientos por parte de los vendedores.

1.1.2. Necesidad del 5G

Las redes actuales, debido al incremento de servicios multimedia que requieren grandes anchos de banda, están sufriendo problemas de sobrecarga y cada vez es más difícil asegurar una Quality of Service (QoS). Por ello, muchas empresas, instituciones, y estados están invirtiendo muchos recursos en diseñar la siguiente generación de red móvil conocida como 5G que

permita dar servicio a una gran cantidad de aplicaciones con necesidades de red muy diferentes e incluso dar paso al IoT (En español: Internet de las cosas).

El objetivo de esta nueva red es cumplir las siguientes características [14]:

- **Tasas binarias de pico de 1 Gbit/s.**
- **Reducción de la latencia a valores en torno a 1 milisegundo.**
- **Incrementar la capacidad agregada del sistema medida en bits por segundo y unidad de área 1000 veces respecto de 4G.**
- **Reducir el consumo de energía en Julios/bit a un valor cien veces menor.**
- **Mejorar la fiabilidad y cobertura de la red.**

Para desarrollar este nuevo modelo de red se están desarrollando una gran cantidad de nuevas tecnologías entre las que pueden destacar [14]:

- La utilización de técnicas como MIMO masivo.
- La utilización de altas frecuencias (mayores que 10 GHz)
- Nuevas formas de onda como modulación no ortogonal.
- Diferentes modelados de red como SDN, Network Function Virtualization (NFV) y *Network Slicing*.

También se está analizando la posibilidad de usar diferentes alternativas a OFDM, ya que se puede mejorar la eficiencia espectral si se reducen los requisitos de sincronización temporal. Se han propuesto soluciones como Non Orthogonal Multiple Access (NOMA), Filter Bank Multi Carrier (FBMC),Sparse Code Multiple Access (SCMA) y Universal Filtered OFDM (UF-OFDM) para sustituir OFDM como sistema de acceso al medio [14].

1.1.3. Tráfico de datos en las redes actuales

En la última década se ha producido un aumento exponencial del tráfico de datos cursado por las diferentes redes. En las imágenes 1.1 y 1.2 se puede observar cómo se prevé un tráfico de casi 400.000 petabytes nada más que en contenidos multimedia para 2020 a través de la red de internet. También

se muestra la evolución del tráfico de datos en redes móviles, que en 2016 se han llegado a consumir 7.240.550 terabytes [38].

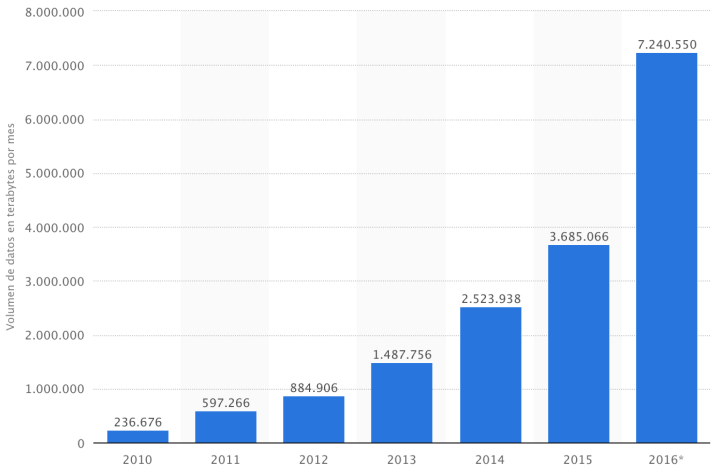

Figura 1.1: tráfico de datos en internet. [38]

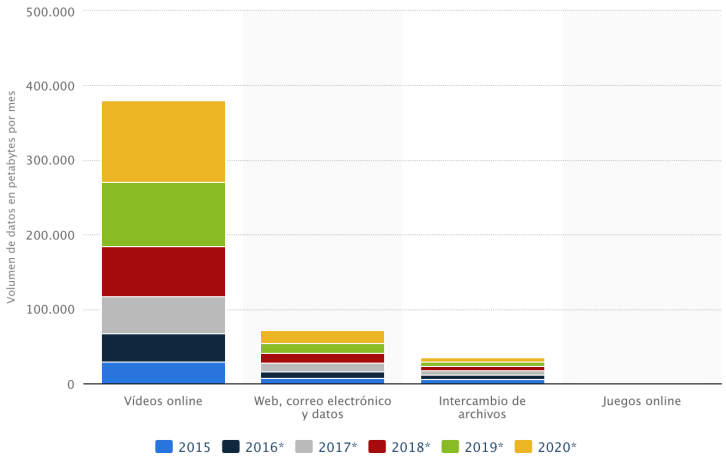

Figura 1.2: Tráfico de datos en redes móviles. [38]

A partir de ambas gráficas se puede deducir que el tráfico va a ir creciendo con el paso de los años, por lo que se hace necesario proporcionar las capacidades que se prevé que dispongan las redes 5G. Otro dato interesante que se puede obtener a partir de las gráficas es la influencia del consumo de servicios multimedia en el tráfico de red (*streaming*, Internet Protocol Television (IPTV), televisión, videoconferencia...).

Por un lado cada vez más empresas ofrecen productos con contenidos

multimedia como *Netflix* o *Yomvi* y también tenemos que este tipo de contenidos son los que más tráfico generan por lo que uno de los objetivos de las redes 5G será asegurar que este tipo de servicios se ofrezcan con una calidad garantizada sin que se produzcan latencias o retardos excesivos. Con ello se consigue que los usuarios finales puedan disfrutar de cualquier contenido multimedia adecuadamente.

1.1.4. Motivación y necesidad de usar *multicast*

En los escenarios donde se encuentran múltiples estaciones para realizar una función concreta, a menudo se necesita transmitir a todas las estaciones paquetes con los mismos datos, generalmente de gran tamaño. En entornos donde la transmisión se realiza de 1 a N, el modelo de transmisión *unicast* resulta ineficiente e inadecuado. Es por ello que surge el modelo *multicast* como opción para este tipo de funciones.

Este modelo esta diseñado para ser escalable en cuanto a cantidad de estaciones destino, ya que la transmisión se realiza una sola vez independientemente del número de receptores. Este mecanismo *multicast* es no orientado a conexión, por lo que la mayoría de protocolos multicast se desarrollan sobre la capa de transporte User Datagram Protocol (UDP). Debido a esto, es necesario construir un esquema de fiabilidad que ayuda a disminuir el número de pérdidas en transmisiones de aplicaciones de *streaming*.

Los modelos de transmisión *multicast* se utilizan para prácticamente casi todas las aplicaciones que necesiten transmitir contenido multimedia a varios clientes, como pueden ser de *streaming* o servicios que almacenan contenido multimedia en servidores para que los clientes de dichos servicios puedan acceder a cualquier contenido cuando quieran, también se utiliza para servicios de videollamada y, en resumen, para cualquier tipo de transmisión multimedia a varios clientes. Como ya se ha dicho anteriormente este tipo de servicios son los que cada vez más tráfico están generando. De lo que se puede deducir que cada vez son más importantes para los usuarios, esto se puede comprobar fácilmente observando el auge de servicios multimedia como *Netflix*, *Yomvi* o *Youtube*.

Para el usuario medio los dos tipos de contenidos más consumidos en Internet son los de páginas web y contenidos multimedia. Debido a la gran cantidad de tráfico generado por los contenidos multimedia cada vez es más importante desarrollar protocolos *multicast* que congestionen menos la red. Lo que incrementa en gran medida la importancia del entorno *multicast*.

En este proyecto se pretende demostrar que SDN gracias a su sistema centralizado favorece el uso de protocolos *multicast* y les da ciertas ventajas a la hora de reducir el tráfico generado y de aumentar la escalabilidad aun más que en las redes actuales. Por todo esto en este proyecto se va a estudiar

el funcionamiento de *multicast* en las redes SDN aportando las diferentes ventajas que puede ofrecer este nuevo modelo de redes para los protocolos *multicast*.

1.2. Principales contribuciones del trabajo Fin de Grado

En la realización del presente trabajo para la obtención del título de graduado en Ingeniería de Tecnologías de Telecomunicación se han conseguido las siguientes contribuciones:

1. Se ha realizado un estudio detallado del estado del arte en relación a multicast y SDN
2. Se ha desarrollado una metodología para evaluar el impacto que SDN tiene en comunicaciones multicast.
3. Se ha implementado un entorno experimental de evaluación que usando Mininet [[15]] y Opendaylight ha permitido simular diferentes entornos.
4. Una vez recogidos los datos se ha detallado en profundidad los diferentes tipos de tráfico generados en los diferentes modelos de red.
5. Se ha modificado la configuración del paquete *learning-switch* para poder controlar todo tipo de topologías.
6. Se han añadido nuevos protocolos (como Open Shortest Path First (OSPF)) a la *suite* de la herramienta *Quagga* [[13]] para poder evaluar una red más completa.
7. Se han obtenido resultados que permiten extraer una conclusión firme sobre las ventajas de utilizar SDN en entornos *multicast*.

1.3. Estructura de la memoria

Para terminar el capítulo de introducción se expondrá la estructura de la memoria describiendo brevemente en qué consistirá cada capítulo. En concreto, la presente memoria se ha organizado en siete capítulos:

- **Capítulo 1. Introducción.** En este capítulo se explica la importancia del desarrollo de nuevas tecnologías de red en el mundo actual debido al tráfico generado en Internet.

- **Capítulo 2. Motivación del proyecto y planificación.** En este capítulo se describen los motivos que han dado lugar a la realización de este proyecto y cuál ha sido la planificación del proyecto incluyendo recursos utilizados y una estimación de los costes.
- **Capítulo 3. Estado del arte.** Descripción del concepto de SDN y estudio sobre la transmisión de tramas en *multicast*; incluyendo protocolos actuales y diseños nuevos de protocolos *multicast* para las redes SDN.
- **Capítulo 4. Fundamentos teóricos y herramientas utilizadas.** En este capítulo se explicarán las diferentes herramientas que se han necesitado para poder realizar la evaluación de ambas redes explicando su funcionamiento.
- **Capítulo 5. Implementaciones para la evaluación.** Este capítulo es el más importante de la memoria. Se recoge toda la información necesaria para poder realizar una comparación objetiva, para ello se miden tiempos de retardo, carga de enlaces, etc.
- **Capítulo 6. Evaluación.** Con los datos recogidos en el capítulo anterior se procede a realizar una comparación, detallando las ventajas de usar redes SDN en multicast en lugar de adoptar una aproximación convencional.
- **Capítulo 7. Conclusión y vías futuras.** En este último capítulo se incluyen las conclusiones obtenidas una vez se ha finalizado el proyecto; proponiendo posibles trabajos futuros y mejoras.
- **Bibliografía.** También se incluye en la memoria la bibliografía utilizada, incluyendo todo el contenido consultado para la realización de la misma.

Capítulo 2

Motivación del proyecto y planificación

Una vez se ha introducido el contexto tecnológico actual; hablando sobre los diferentes aspectos que han motivado el desarrollo de nuevas tecnologías. Se procede a describir cuáles son los objetivos del proyecto y cuál será la planificación para completar de forma adecuada y con el tiempo suficiente el proyecto.

2.1. Motivación del proyecto

El proyecto va a consistir en una evaluación de los mecanismos de SDN en entornos *multicast*. De forma teórica se tratará desde un punto de vista general la tecnología SDN detallando sus características y ventajas; también se hablará posteriormente sobre los diferentes tipos de software que se pueden utilizar para desarrollar y simular redes SDN.

Con el objetivo de poder realizar una evaluación más detallada y exhaustiva de las ventajas de SDN nos apoyaremos en un entorno ya configurado [47]. Dicho entorno consiste en una máquina virtual con dos redes configuradas; la topología de ambas está diseñada con *Mininet* y utilizan el protocolo *multicast* PIM-SSM [25]. El trabajo desarrollado incluye un entorno SDN gracias a la configuración del controlador con la herramienta OpenDayLight y por otro lado, un despliegue *multicast* clásico usando la herramienta Quagga. Aunque el diseño toma como punto de partida un trabajo anterior [47], en este trabajo se detallan y explican las diferentes herramientas implicadas, ya que para la evaluación realizada serán necesarias algunas modificaciones. El diseño de la topología es el mismo para los dos escenarios considerados y se puede observar en la Figura (5.1).

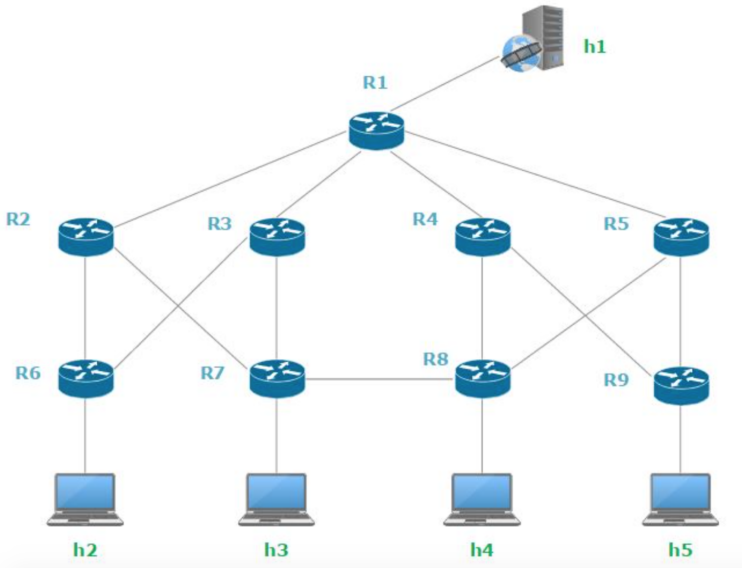

Figura 2.1: Topología implementada en Mininet. [[47]]

Se ha elegido evaluar un protocolo *multicast* debido a la importancia que tienen en los servicios ofrecidos actualmente (Netflix, Yomvi...) y al impacto de dichos servicios en la sociedad actual, tal y como ya se comentó en la introducción. *Multicast* es clave en el despliegue de estos servicios. Además pese a la existencia de otras tecnologías como NFV, este tipo de servicios, como se demostrará en este TFG se ven beneficiados por la arquitectura centralizada de SDN.

2.1.1. Objetivos del proyecto

El objetivo principal de este trabajo fin de grado es realizar una comparación entre un despliegue *multicast* con SDN frente a una aproximación clásica usando quagga, presentando sus diferentes ventajas gracias a la ayuda de herramientas de *benchmarking* como Wireshark o Iperf. Además, el proyecto también puede servir de guía para realizar evaluaciones con la herramienta mininet, ya que se explicará con detalle los diferentes métodos que se han utilizado para obtener los resultados. De forma resumida se podrían dividir los objetivos en los siguientes puntos:

- Revisión del estado del arte.
- Estudio de simulaciones de diferentes despliegues multicast (con y sin SDN).

- Evaluación y comparación de las características de ambas aproximaciones.
- Análisis crítico sobre las redes SDN.

2.1.2. Especificación de requisitos

En este apartado se identifican que requisitos serán necesarios para el desarrollo del proyecto; una vez identificados, se agrupan dichos requisitos en distintos bloques de trabajo para poder realizar una adecuada planificación. Los requisitos de este proyecto son:

- Familiarizarse con los entornos de mininet y quagga para poder realizar las configuraciones necesarias.
- Comprender el funcionamiento del protocolo *multicast*.
- Una vez se ha comprendido el funcionamiento de los diferentes entornos será necesario establecer diferentes técnicas para recoger los diferentes datos para poder realizar la evaluación.
- Finalmente será necesario interpretar los datos recogidos para poder establecer un análisis comparativo fiable.

2.1.3. Metodología

Para más claridad, se dedicará un apartado para ordenar de forma cronológica los pasos seguidos durante el proyecto. El primer paso es establecer los objetivos del proyecto, una vez hecho esto se procede a realizar un estudio teórico sobre el concepto de SDN.

Una vez se ha comprendido cómo funciona el entorno SDN de forma general se pasa a estudiar de forma teórica también todo lo relacionado con la tecnología *multicast*: protocolos multicast actuales, nuevos diseños de protocolos para SDN, nuevos algoritmos para las diferentes funciones de un protocolo *multicast*.

A continuación, se pone en funcionamiento la máquina virtual con las diferentes configuraciones y se familiariza con los entornos: Quagga, Mininet y OpenDaylight. Observando también el funcionamiento del protocolo PIM-SSM; como se distribuye el árbol en ambas aproximaciones, cuál es el tráfico generado en ambos casos y sus diferentes características.

Una vez se estudia el funcionamiento de ambos enfoques, se realizan varias configuraciones sencillas probando diferentes topologías, añadiendo

nuevas funcionalidades al controlador en el caso de SDN o añadiendo nuevos servicios en el caso de Quagga.

Una vez se han realizado todos los pasos anteriores se procede a realizar una evaluación de ambas tecnologías (con y sin SDN) con ayuda de varias herramientas como Wireshark o Iperf para recoger los diferentes datos que serán necesarios para realizar una comparación objetiva.

2.2. Planificación

En este apartado se expone una aproximación del tiempo que se va a dedicar a cada una de las tareas necesarias para la realización del proyecto. Para poder organizarlo de forma más clara se dividirán las tareas en bloques funcionales. Posteriormente se identificarán los diferentes recursos utilizados durante el proyecto y una estimación de los costes asociados a dichos recursos.

2.2.1. Estimación de la planificación

Para el comienzo de algunas tareas es necesario haber finalizado otras con anterioridad, por lo que existe la prioridad de establecer unas fechas de corte para cada una de las tareas con el fin de realizar el proyecto de forma estructurada. Para poder establecer dichas fechas de corte será de gran ayuda separar en los siguientes paquetes de trabajo:

- **PT1: Estudio teórico**

  En esta tarea se va a recopilar información sobre temas relacionados con el proyecto. Para ello se harán uso de herramientas de búsqueda como scopus o google académico para recoger toda la bibliografía necesaria para la elaboración del proyecto. De forma más específica, también se va a realizar una revisión del estado del arte en cuanto a técnicas *multicast* para entornos SDN. Todo esto ayudará a tener una mayor comprensión sobre el tema abordado y como consecuencia permitirá elaborar una evaluación útil.

- **PT2: Familiarización con mininet**

  Para el desarrollo del proyecto será necesario familiarizarse con el entorno de mininet y el lenguaje de programación Python para el diseño de diferentes topologías de red e implementación de nuevas características.

- **PT3: Familiarización con OpenDaylight y Quagga**

Este módulo es similar y depende en gran medida del PT2. Para familiarizarse con las dos redes creadas hace falta primero aprender el funcionamiento de mininet para posteriormente en este bloque familiarizarse con el controlador OpenDayLight y los demonios de Quagga.

- **PT4: Estudio práctico del protocolo PIM-SSM implementado en las redes SDN y de Quagga.**

  En este módulo se estudia el funcionamiento del protocolo PIM-SSM buscando información en diferentes artículos y páginas para después con ayuda de Wireshark comprobar de forma práctica que funciona exactamente como se dice en la teoría. Es importante conocer las diferencias entre las tramas de transmisión y las de generación de árboles para poder realizar una evaluación correcta.

- **PT5: Configuración de entornos para la realización de medidas**

  En este proyecto se va a evaluar la robustez, escalabilidad, ingeniería de tráfico, velocidades de paquetes y demás características en ambas aproximaciones. Para ello será necesario algunas configuraciones adicionales. Por ejemplo, en el caso de Quagga está configurado en encaminamiento estático por lo que de esta manera no se podría evaluar la robustez del protocolo; para evitar esto se cambiará la configuración por un encaminamiento dinámico. En el caso del controlador SDN también serán necesarias algunas modificaciones que se explicarán más adelante.

- **PT6: Interpretación de las medidas obtenidas durante la evaluación**

  Una vez recogidos los datos será necesario una interpretación objetiva para obtener diferentes conclusiones y poder así hacer una comparativa objetiva y crítica de las ventajas de utilizar SDN para protocolos multicast.

- **PT7: Elaboración de una memoria técnica del proyecto**

  Por último, será necesario realizar la memoria o documentación técnica del proyecto que contenga todos los aspectos teóricos y prácticos obtenidos durante el desarrollo del Trabajo Fin de Grado. La memoria se va a redactar durante la mayor parte del tiempo de realización del proyecto. Una vez se hayan establecido los objetivos del proyecto y se haya estudiado la documentación se empezará con este bloque.

Una vez definidos los paquetes de trabajo se procede a estimar el tiempo necesario de cada uno de ellos para posteriormente realizar un diagrama de

Gantt con las fechas de corte y una estimación exacta de la planificación del desarrollo de todo el proyecto.

Paquetes de trabajo	Estimación de tiempo
PT1	80 horas
PT2	30 horas
PT3	50 horas
PT4	30 horas
PT5	60 horas
PT6	70 horas
PT7	100 horas
Total:	420 horas

Tabla 2.1: Planificación temporal de las tareas.

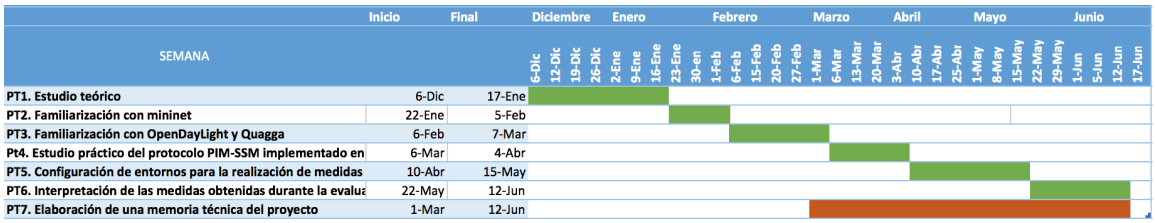

Figura 2.2: Diagrama de Gantt.

2.2.2. Recursos utilizados

En esta sección se enumeran los diferentes recursos utilizados para el desarrollo del proyecto. Por un lado, tenemos los recursos humanos utilizados y por otro lado los materiales necesitados tanto de hardware como software.

Recursos humanos

- D. Juan Manuel López Soler en calidad de tutor del proyecto y D. Jorge Navarro Ortiz en calidad de cotutor del proyecto, profesores del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.
- Pablo Góngora Luque, alumno del Grado en Ingeniería de Tecnologías de Telecomunicación y autor del presente proyecto.

Materiales necesitados

- Ordenador portátil HP 15-ay135ns con sistema operativo Windows 8.1 (64 bits) instalado como nativo.
- Mininet, OpenDayLight, Eclipse Luna, Quagga, Wireshark, Iperf, Python y Java instalado en una máquina virtual con Ubuntu 16.0.
- Overleaf, procesador de texto con el que se realiza la memoria.

2.2.3. Estimación de costes

En esta sección, se hace una estimación aproximada de los costes que supone la realización de este TFG. El coste no es elevado debido a que todos los recursos software necesarios son de código libre.

Para realizar el cálculo se tiene en cuenta:

- El sueldo de un ingeniero es de 20 euros/hora aproximadamente.
- El sueldo medio de un Doctor de la Universidad de Granada es de 50 euros/hora aproximadamente.

Con esos datos y estimando que el tiempo invertido por ambos profesores en tutorías y revisión de la memoria será de unas 35 horas se puede establecer una Tabla (2.2) con el coste total apoyándose en la Tabla (2.1).

Paquetes de trabajo	Horas	Coste
PT1	80	1600
PT2	30	600
PT3	50	1000
PT4	30	600
PT5	60	1200
PT6	70	1400
PT7	100	2000
Tutorías y revisiones	35	
Tutor	27	1350
Cotutor	8	400
Total	455	10150

Tabla 2.2: Costes de las diferentes tareas.

Si añadimos a los 10150 Euros de la tabla el precio del ordenador que se ha utilizado 799 tenemos unos 10949. Teniendo en cuenta que todo el software que se ha utilizado es de código abierto por lo que no tiene coste alguno tendríamos que el coste final del proyecto es de **10949 Euros**.

2.2.4. Valoración final

Se termina el capítulo valorando la importancia de una buena planificación. Ya que, pese a que no se han podido cumplir los plazos fijados al comienzo del capítulo, debido a diversos inconvenientes que han ido retrasando el desarrollo del proyecto. Gracias a dicha planificación se ha podido finalizar el proyecto de forma satisfactoria.

El paquete de trabajo relacionado con la configuración de entornos para la realización de medidas, es decir, el PT5 ha necesitado mucho más tiempo del que en un principio se pensó que necesitaría, debido sobre todo a la configuración del controlador OpenDayLight que en un principio se pensó que estaba programado para funcionar ante cualquier topología. Ante esto se ha necesitado comprender el código ya implementado y una vez comprendido; ha sido necesario modificarlo para poder realizar la evaluación de diferentes topologías. Pese al retraso que ha supuesto dicho paquete y algunos otros, el proyecto se ha podido finalizar adecuadamente en el tiempo planificado.

Capítulo 3

Estado del arte

En este capítulo se resumen las aportaciones y avances recientes más relevantes sobre SDN y *multicast* para ofrecer una visión global sobre la importancia del diseño de nuevas redes y las ventajas que ofrecen Se describirá el funcionamiento de SDN y posteriormente se explicarán algunos de los mecanismos que se están utilizando para adaptar los protocolos *multicast* a las redes SDN. Haciendo énfasis en los resultados obtenidos en simulaciones sobre mininet.

3.1. SDN: Software Designed Network

3.1.1. Arquitectura SDN

Las arquitecturas de red SDN se basan en el concepto de *Slicing*, dichas redes se caracterizan por separar las diferentes funcionalidades de la red en diferentes capas [45]. En el caso de SDN se distinguen tres capas diferentes: La capa de control, la capa de infraestructura y la capa de aplicación (ver Figura (3.1)).

- Capa de control: Esta capa está formada por los controladores, que son los encargados de administrar de forma inteligente el funcionamiento de toda la red. En dichos controladores se realiza todo el procesamiento y esta capa se comunica con las otras mediante protocolos definidos como OpenFlow. La separación de las funcionalidades de software respecto del hardware permite no depender de los fabricantes de hardware y con ello tener más independencia para poder configurar/operar/actualizar la red.
- Capa de infraestructura: Esta capa está formada por los dispositivos hardware que componen la red. En este tipo de redes los dispositivos

hardware son simples conmutadores con una funcionalidad y software básicos, ya que la inteligencia de la red reside en la capa de control.

- Capa de aplicación: la capa de control puede gestionarse a través de una aplicación que interactúe con los planos de control y de infraestructura. La capa de aplicación se encarga de abstraer la información de los servicios y de la configuración de los dispositivos; así como de la topología de la red y en definitiva, de la gestión del tráfico de red. Todo esto gracias a la información proporcionada por la capa de control. Además, esta capa también puede tomar decisiones teniendo en cuenta los datos que le proporcionan los diferentes dispositivos hardware. La capa de aplicación introduce los conceptos de *Northbound* y *Southbound*. *Southbound* es una API que se encarga de la comunicación desde la capa de control hacia la capa de infraestructura (típicamente normalizada con el protocolo OpenFlow [20]) mientras que *Northbound* se encarga de la comunicación desde la capa de infraestructura hacia la de control.

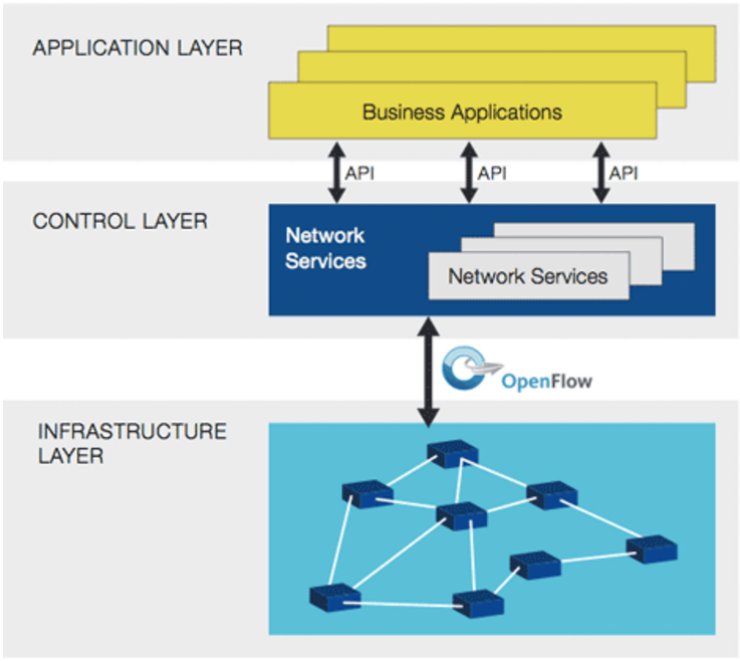

Figura 3.1: Arquitectura SDN. [[45]]

La estructura definida en la Figura (3.1) es conforme al enfoque *Network Slicing*. No obstante, una forma más intuitiva de explicar el diseño de redes SDN sería definiendo las diferentes funcionalidades a alto nivel 3.2, como se hace a continuación:

- Plano de control: esta capa se encarga del encaminamiento de los flujos de datos, decide cuándo habilitar o deshabilitar un flujo de datos, el comportamiento de las colas y cualquier actividad relacionada con el transporte de datos.
- Plano de reenvío: esta capa se encarga de la retransmisión y el procesamiento de datos de los diferentes servicios, basándose en instrucciones proporcionadas por el plano de control. El plano de control facilita el tratamiento de los datos proporcionados por los servicios, mientras que en la capa de infraestructura se realizan diferentes acciones basadas en las decisiones del plano de control.
- Plano de administración: Mientras que los planos de control y de reenvío se encargan del tráfico de datos, el plano de administración tiene las tareas de configurar, monitorizar y administrar los recursos para los servicios de red.
- Plano operacional: El estado operacional de los servicios se monitoriza gracias al plano operacional, que tiene una relación directa con las entidades de los servicios. Este plano trabaja directamente con el plano de administración para almacenar información y actualizar el estado operacional de los diferentes servicios.

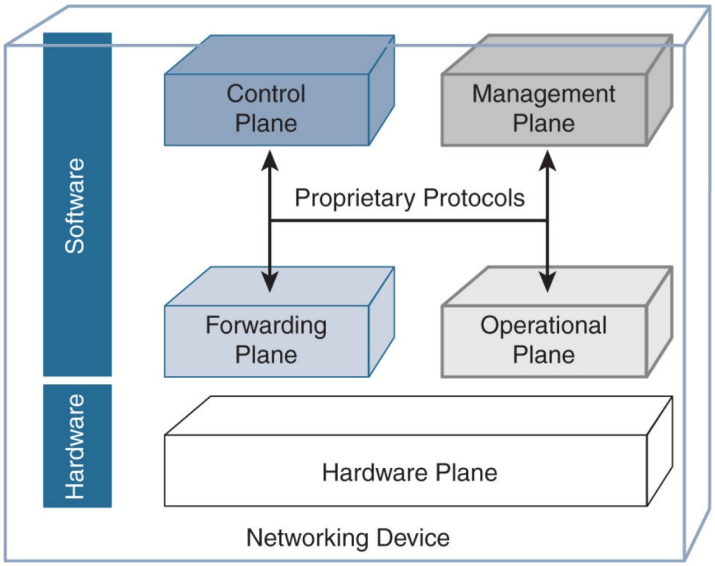

Figura 3.2: Estructura SDN desde nivel de aplicación.[[45]]

3.1.2. Ventajas de SDN

Aunque en un principio las arquitecturas SDN se consideraron solo para el ámbito académico, pronto las diferentes empresas empezaron a desarrollar el concepto de SDN gracias a las múltiples ventajas que presenta frente a las redes actuales. A continuación, se van a enumerar las más importantes.

1. Se pueden diseñar redes programables y automatizables: La habilidad para controlar la configuración de la red es una de las ventajas más importantes de SDN. Los servicios de hoy en día necesitan que las redes tengan una gran capacidad de restauración ante fallos, escalabilidad masiva, despliegue rápido y una gran capacidad de optimización. Gracias a los controladores las redes SDN pueden reprogramarse fácilmente cambiando la configuración del controlador y por definición; gracias al controlador la red funciona de forma automatizada y desacoplada del hardware.

2. Control centralizado: La centralización de las redes SDN permite que toda la información necesaria para la gestión de la red este accesible de forma sencilla. Esta aproximación permite disponer de una visión global o de conjunto, no disponible en una aproximación clásica. Además, permite crear redes más sólidas con puntos críticos identificados; uno de los inconvenientes de estas arquitecturas de redes es los posibles fallos que se puedan producir en el controlador. Un fallo en el controlador podría provocar la caída de toda la red por lo que pese a tener muchas ventajas tener un control centralizado también tiene el inconveniente de tener puntos muy vulnerables. Este problema se podría solucionar añadiendo redundancia con diferentes controladores o con nuevos diseños de red como es el caso de NFV.

3. Arquitecturas de red abiertas: SDN permite la independencia con los vendedores de hardware gracias a los protocolos estandarizados. En las redes actuales los dispositivos hardware de la red vienen con paquetes de software ya definidos lo que obliga a la red a adaptarse cada vez que se introduce un nuevo dispositivo. Esta problemática se acentúa debido a la variedad de proveedores de hardware y la inexistencia de estándares a la hora de diseñar los diferentes dispositivos hardware. Incluso utilizando protocolos estandarizados se pueden dar situaciones en las que haya problemas de interoperabilidad. Esta ventaja que ofrecen las redes SDN son uno de los principales objetivos ya que permitirían un ahorro significativo.

4. Implementación de los servicios: En las redes actuales los servicios tienen que tener programación adicional para encaminamiento, detección de errores, seguridad, etc. Con las arquitecturas SDN los servicios se

pueden implementar de forma más sencilla gracias a la capa de aplicación, la cual se encarga de todas estas funcionalidades secundarias.

3.2. Protocolos SDN

SDN necesita protocolos estandarizados para la comunicación entre las diferentes capas. Estos protocolos se agrupan en dos categorías protocolos *northbound* y protocolos *southbound*.

Los protocolos Southbound están situados lógicamente entre el plano de control y el plano de reenvío mientras que los protocolos Northbound se utilizan para que las aplicaciones se comuniquen con el controlador.

3.2.1. Protocolos Southbound

Los protocolos Southbound se dividen en dos categorías. Los del plano de control y los del plano de administración. Para el caso del plano de control entre otros destacan Openflow, PCEP (Path Computation Element Communication Protocol) [[31]], BGP Flow Spec [[43]].

Openflow se encarga de las comunicaciones entre la capa de control y la capa de infraestructura. OpenFlow tiene una tabla de flujos, que contiene información sobre como tienen que enviarse los datos. El controlador puede usar OpenFlow para modificar la configuración de un *switch* cambiando los valores de su tabla de flujo. OpenFlow tiene dos modos de operación reactivo y proactivo.

El modo reactivo es el modo por defecto y consiste en que OpenFlow considera que no hay implementada ningún tipo de inteligencia en los diferentes servicios de red. En este modo, el primer paquete que llegue al plano de reenvío es enviado al controlador y el controlador utiliza la información recibida para programar el encaminamiento del flujo a través de la red; esta acción modifica los valores de las tablas de flujo de los dispositivos por los que pasa el nuevo flujo.

En el modo proactivo, el controlador es preconfigurado con algunos valores de encaminamiento de flujo por defecto para que si recibe algún paquete los envíe directamente según esos valores predefinidos. También se utilizan los protocolos Secure Sockets Layer (SSL) [[32]] y Transport Layer Security (TLS) [[30]] para proporcionar confidencialidad en las comunicaciones.

Path Computation Element Communication Protocol (PCEP) es un protocolo que se sirve de dos protocolos PCC (Path Computation Client) y PCE (Path Computation Element) para calcular las diferentes rutas para los diferentes tipos de flujos. Este protocolo se ha diseñado para ingeniería de

tráfico y es compatible con RSVP-TE y MPLS.

Border Gateway Protocol (BGP) Flow Spec es una mejora del protocolo BGP para los sistemas SDN y sirve para intercambiar información de encaminamiento.

Por otro, lado tenemos los protocolos del plano de administración; estos protocolos sirven para configurar los diferentes dispositivos e interactúan con el plano de reenvío. Entre los más importantes tenemos Network Configuration Protocol (NETCONF) [34], Rest Configuration Protocol (RESTCONF) [37], OPENCONFIG, Extensible Messaging and Presence Protocol (XMPP) [33] e Interface to the Routing System (I2RS) [36].

NETCONF es un protocolo que utiliza el modelo cliente-servidor; los servicios actúan como cliente y los dispositivos hardware como servidor, de manera que los diferentes servicios solicitan a los dispositivos las configuraciones que necesitan para ser transportados a los dispositivos gracias a este protocolo.

RESTCONF es una alternativa a NETCONF que incluye el lenguaje de interpretación YANG para interpretar los cambios producidos entre los servicios y los dispositivos.

3.2.2. Protocolos Northbound

Los protocolos Northbound se encargan de las comunicaciones entre el controlador y la capa de aplicación, pero en sentido inverso a los protocolos Southbound. Dicha comunicación no es muy diferente de la que se produce entre dos entidades software por lo que no existen apenas estándares de protocolos definidos ya que se pueden implementar protocolos con lenguaje de programación como Python, Java o C++.

3.2.3. Tipos de controladores

Para terminar el apartado se mencionan los diferentes controladores que hay para las redes SDN. Entre los más importantes destacan OpenDaylight(ODL), RYU [9], ONOS [7], OpenContrail [8], VMware NSX [10], Cisco SDN Controllers [5], Juniper Contrail [6]. En este proyecto se hará uso de OpenDaylight por diversos motivos.

Uno de los principales motivos por los que se ha elegido el controlador OpenDaylight es que, a diferencia de la mayoría de los controladores mencionados anteriormente, OpenDaylight es de código abierto lo que posibilita su uso sin coste alguno. Está desarrollado mediante Java; que es un lenguaje de programación que se ha estudiado en el Grado y tiene características muy importantes a nivel de rendimiento. Es robusto, tiene buen soporte y

admite varias versiones del protocolo OpenFlow [44]. La única desventaja que afecta a este proyecto es su complejidad a la hora de desarrollar aplicaciones. Aunque en este caso al partir de un paquete ya definido se suaviza parcialmente esta problemática.

3.3. Multicast

3.3.1. Protocolos multicast

Una transmisión multicast [12] consiste en enviar un paquete IP con una dirección destino que englobe a un grupo de *hosts* interesados en recibir el paquete. Su uso está bastante extendido y se utiliza para aplicaciones que ofrecen servicios multimedia. Algunos de los protocolos multicast más importante son IGMP (Internet Group Management Protocol), MOSPF (Multicast Open Shortest Path First), DVMRP (Distance Vector Multicast Router Protocol), PIMDM (Protocol Independent Multicast Dense Mode), PIM-SM, CBT.

3.3.2. Protocolos multicast definidos para redes SDN

Ante las ventajas que presentan los protocolos *multicast* frente a unicast, como la descongestión de la red, se está estudiando la forma de implementar diferentes protocolos *multicast* en redes de *slicing*. En concreto para SDN tenemos algunos ejemplos como AvRA (Avalanche Routing Algorithm), RAERA (Recover Aware Edge Reduction) [27], BAREA (Branch Aware Edge Reduction Algorithm) [35] etc.

3.3.3. Evaluación de balanceo de carga con protocolos multicast en redes SDN

Hoy en día existen numerosas aplicaciones que gracias a Internet transmiten datos multimedia en tiempo real para diferentes receptores. Ejemplo de estas aplicaciones pueden ser IPTV, conferencias de vídeo y audio, juegos multijugador o VLANs.

En este proyecto se va a realizar una evaluación sobre la eficiencia de los protocolos *multicast* en redes SDN, para ello antes se hablará de las diferentes evaluaciones que se han realizado.

En [46] tienen como objetivo demostrar que el balanceo de carga a través de cambios en la configuración de enlaces en tiempo real se puede implementar de forma factible en un controlador SDN y que esta técnica de balanceo de carga proporciona beneficios significativos en cuanto a congestión de red.

Para realizar esto se utiliza el algoritmo de Dijkstra [40]. Se evalúan dos tipos de funciones para el cambio de costes de enlaces; una proporcionalmente lineal y otra inversa y se comparan los resultados con los obtenidos por algoritmos de camino más corto.

Las funciones las escogen por su forma de dirigir el tráfico a través de los enlaces menos congestionados, sin aplicar métodos complejos. En la expresión (3.1) se muestra la función proporcionalmente lineal que utilizan y en la expresión (3.2)la que es inversamente proporcional.

$$C_i = \begin{cases} \delta : U_i = 0 \\ \dfrac{U_i}{M_i} : U_i > 0 \end{cases} \tag{3.1}$$

Donde C_i es la función de coste del enlace i, U_i es la utilización del coste del enlace en Mbps, M_i es la capacidad máxima del enlace o ancho de banda, y σ es el valor mínimo de punto flotante del controlador

$$C_i = \begin{cases} \delta : U_i = 0 \\ \dfrac{1}{1 - \frac{U_i}{M_i}} - 1 : U_i < M_i 0 \\ \sigma : U_i \geq M_i 0 \end{cases} \tag{3.2}$$

Donde σ que es el valor máximo de punto flotante del controlador. Para dicha evaluación se configura el controlador SDN con los siguientes módulos:

1. **Gestión de la pertenencia a grupos de multidifusión** El controlador SDN tiene implementado IGMPv3 de forma semi-centralizada. Esto se realiza considerando a todos los enrutadores como uno solo funcionando con IGMPv3, cuyos puertos habilitados para IGMP están formados por la unión de todos los puertos de la red. Esta optimización se basa en la idea de que inundar toda la red con mensajes IGMP es redundante cuando ya se procesan los mensajes IGMP en el controlador. Este módulo solo depende del módulo de enrutamiento de *multicast* y se mantiene un mapa de estado de recepción; que consiste en listas de direcciones IP de grupos *multicast* solicitados y puertos en los que se solicita la entrega.

2. **Medición del tráfico en tiempo real** este módulo se encarga del seguimiento del uso de ancho de banda en tiempo real y registra las capacidades y utilizaciones de cada enlace. La estimación de tráfico se realiza con sondeos periódicos de toda la red. Los tiempos de consulta de cada conmutador se escalonan aleatoriamente basándose en el momento en que cada conmutador se conecta al controlador.

3. **Modelo de enrutamiento de *multicast***

   este modelo implementa la detección de los remitentes de *multicast*, el cálculo del árbol de multidifusión y la instalación de reglas para dirigir el tráfico *multicast*.

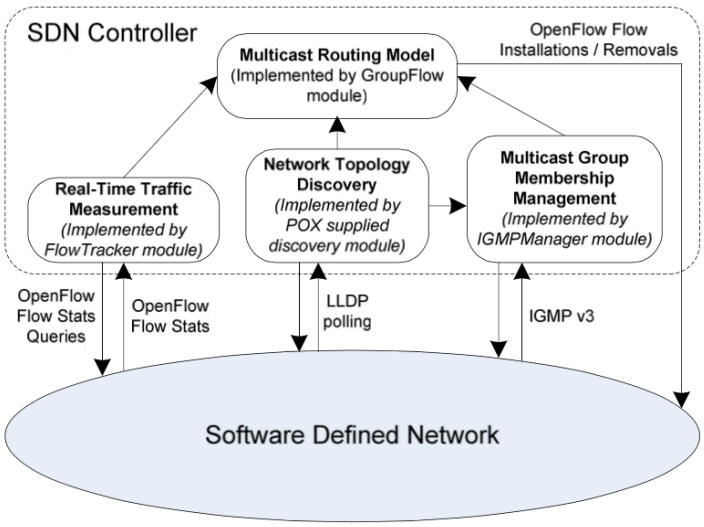

Figura 3.3: Configuración SDN.[[46]]

En las Figuras (3.4) y (3.5) se muestran los resultados obtenidos en [[46]].

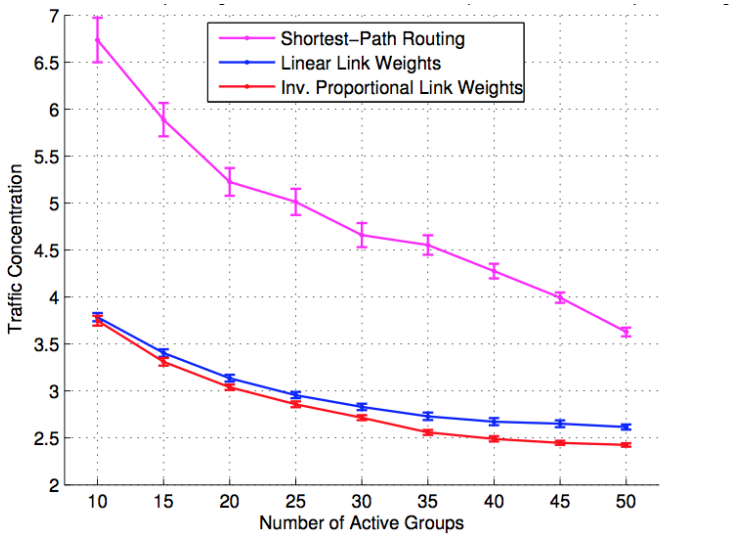

Figura 3.4: Concentración del tráfico en función del número de grupos activos de multicast. [[46]]

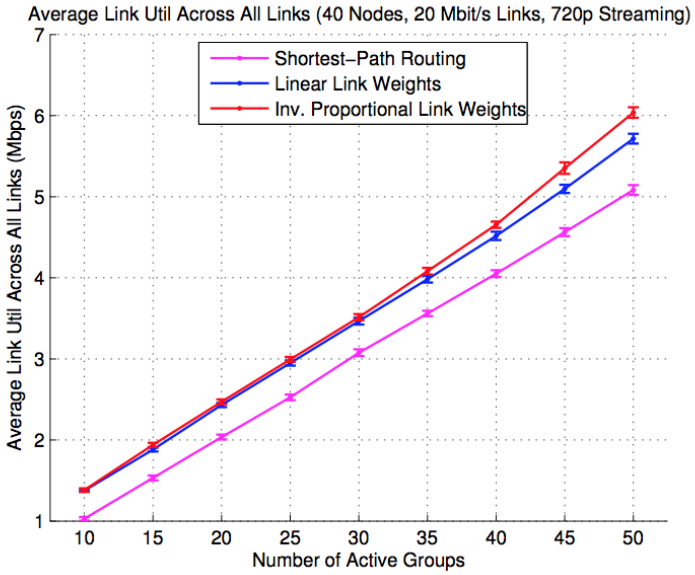

Figura 3.5: Throughput de todos los enlaces de la red representada en función del número de grupos multicast.[[46]]

Como se puede observar en ambas figuras al utilizar las dos funciones se consigue una reducción de la concentración de tráfico considerable. Además, una concentración baja del tráfico indica que se distribuye de forma más uniforme permitiendo una mayor fiabilidad en la entrega de paquetes *multicast*. En el caso de la función lineal se consigue una concentración de tráfico de un 48,6 % menor que en el caso en el que se utiliza el algoritmo del camino más corto, para la función inversamente lineal se consigue una reducción del 52,8 %. En la Figura (3.5) se observa que al disminuir la concentración del tráfico la utilización de la red aumenta generando así un mayor aprovechamiento de los recursos.

3.3.4. Evaluación de escalabilidad de técnicas multicast en redes SDN

El objetivo del articulo [48] es demostrar la posibilidad de ofrecer servicios multimedia con mejor QoS gracias a la estructura SDN y a las técnicas de *multicast*. Para ello, se parte de que en las redes SDN los *switches* no tiene que soportar la carga de los protocolos *multicast* y de que el controlador conoce todas las características de la red. Gracias a esta capacidad del controlador se crean árboles óptimos de difusión *multicast* para conseguir transmisiones más eficientes e incluso un modelo de difusión *multicasting* más escalable que los proporcionados por las redes actuales.

Los protocolos de *multicasting* se tienen que diseñar de formas diferentes para las redes SDN; una primera aproximación es hacer el cálculo de las rutas entre la fuente y el host cuando el controlador recibe el mensaje IGMP, aunque el problema de esta metodología es que se produce una latencia inicial bastante grande si se unen a un grupo multicast muchos hosts.

Otra aproximación es realizar el cálculo de todas las rutas posibles con antelación para reducir la latencia inicial. Sin embargo, ninguno de los enfoques anteriores podría abarcar un gran número de grupos de multidifusión ya que se basan en árboles de difusión. El objetivo del articulo consiste en reducir el número de árboles de multidifusión con el fin de conseguir mayor escalabilidad.

Gracias a un enfoque de *multicast local-aware* (LAMA) consiguen cumplir los requisitos de servicios de *streaming* de vídeo con mayor escalabilidad. En LAMA varios grupos de multidifusión se agrupan de forma adaptativa en un *cluster multicast*. Una vez desarrollado el método LAMA se evalúa el rendimiento de este nuevo método. Se utiliza para ello un controlador Ryu y la metodología LAMA, que consiste en cuatro bloques funcionales: descubridor de topología, administrador de grupos, constructor de árboles *multicast* y generador de flujos.

Para poder realizar una evaluación se utilizan dos servidores donde se emula el entorno; en uno de ellos se ha emulado toda la estructura SDN y en el otro, un generador de topologías características de internet BRITE [4] para simular diferentes entornos *multicast* propios de las redes actuales.

Una vez realizada la evaluación obtienen los resultados de la figura 3.6.

Se realizan diferentes medidas; se obtiene el número de flujos para los diferentes métodos de creación de árboles *multicast* en función del número de *switches*, fuentes *multicast* y en función del número de *hosts*. Se puede observar que el número de flujos para el caso de PST (Per Source Tree) aumenta significativamente a medida que aumenta el número de *switches*, *hosts* y fuentes *multicast* por lo que es un método muy poco escalable. Por otro lado, tenemos el mismo número de flujos tanto para el nuevo método LAMA como para SST (Single Shared Tree). Pero SST es irrealizable en un entorno real ya que no ofrece garantía en cuanto a garantías de QoS, por lo que el artículo demuestra el nuevo método de *multicast* escalable LAMA que da la posibilidad de implementar servicios sin que ocupen gran ancho de banda.

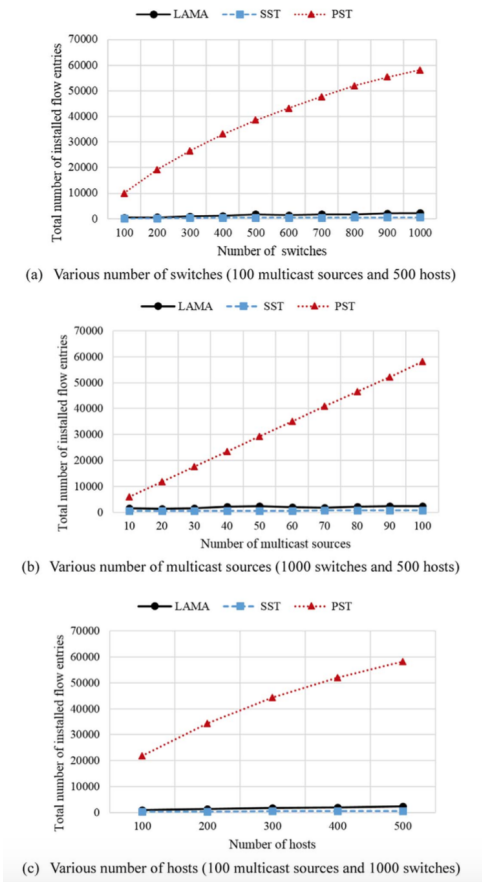

Figura 3.6: Evaluaciones.[[48]]

3.3.5. Evaluación de una gestión consistente en redes SDN con tráfico IP multicast

En el artículo [49] se propone un nuevo modelo de *multicast* llamado DYNSDM que complementa a SDM (Software Defined Multicast) para conseguir mayor flexibilidad, eficiencia y escalabilidad en entornos de red de gran escala (ISPs). DYNSDM presenta un diseño detallado del proceso interno de tráfico y gestión de servicios de cada ISP e introduce un conjunto de nuevos mecanismos para la capa de red basados en SDN para balanceo de carga. En la Figura (3.7) se puede observar cómo funciona el protocolo DYNSDM en cuanto a generación de árbol y administración de procesos.

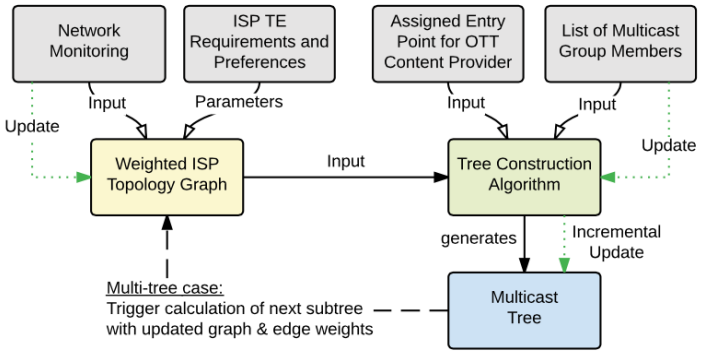

Figura 3.7: Modelo DYNSDM.[[49]]

No obstante, no se entrará en detalle sobre la estructura del modelo ya que, pese a ser interesante el nuevo diseño de multicast, está enfocado para ISP y el enfoque de este proyecto no va dirigido a esos entornos, por lo que solo se ha comentado DYNSDM para ofrecer una visión completa del avance de los protocolos multicast impulsados por el desarrollo de nuevas modelos de red como SDN. En el siguiente capítulo, se explica el motivo por el que se ha hablado sobre los resultados de los tres artículos terminando con una conclusión sobre el capítulo.

3.3.6. Conclusión sobre las diferentes evaluaciones

Como se ha podido apreciar se están investigando nuevas formas de transmisión de tramas multicast para las nuevas redes SDN, ya que dichas redes ofrecen un buen número de nuevas posibilidades. En este proyecto se va a evaluar el funcionamiento de una técnica multicast ya implementada en las redes actuales para ver qué ventajas ofrecen las redes SDN para los diferentes protocolos multicast. El objetivo a la hora de revisar el estado del arte era comprobar que los protocolos multicast se ven beneficiados de las técnicas SDN. Además, basándose en los diferentes aspectos tenidos en cuenta por los diferentes artículos considerados, en este proyecto se hará énfasis en las características de balanceo de carga, escalabilidad y eficiencia.

Capítulo 4

Fundamentos teóricos y herramientas utilizadas

Tras comentar la importancia de las redes SDN para los protocolos *multicast*, y habiendo mencionado algunas de las propuestas más relevantes en este campo, en el presente capítulo se explican las herramientas que harán posible la caracterización de los diferentes entornos, además de repasar los fundamentos teóricos en los que se basa el desarrollo del proyecto.

4.1. Estudio de ambos escenarios

4.1.1. Herramientas utilizadas

En este apartado primero se estudia la herramienta Mininet que se utiliza en este proyecto para diseñar nuevas topologías de red y poder realizar evaluaciones sobre los diferentes aspectos a estudiar. Posteriormente se define el protocolo PIM-SSM tanto para Quagga como en el controlador OpenDayLight ya que tienen algunas diferencias en cuanto a su funcionamiento. El objetivo de poder identificar los diferentes tipos de tráfico que generan y así como poder realizar una evaluación más precisa.

4.1.2. Mininet

Mininet es un emulador de redes capaz de crear redes virtuales formadas por dispositivos como *hosts*, *switches*, *controladores* y enlaces [15]. Los *hosts* de Mininet se simulan con un sistema operativo Linux estándar y los *switches* pueden soportar OpenFlow, permitiendo desarrollar escenarios con tecnología SDN.

Mininet permite que en las simulaciones los dispositivos puedan intercambiar paquetes de diferentes protocolos y ejecutar programas instalados en el sistema. También se puede llevar a cabo *debugging*, de manera que varios desarrolladores pueden trabajar simultánemamente en una misma topología.

Una de las mayores ventajas es su rapidez a la hora de configurar diferentes topologías, el diseño de una red simple se puede llevar muy fácilmente.

Comandos útiles

Mininet tiene la posibilidad de crear redes mediante comandos o mediante programación en Python. Esta última opción es posible gracias a la provisión de una Application Programming Interface (API) en Python, que en definitiva será la que se utilizará para el desarrollo del proyecto. Sin embargo, es conveniente crear un par de redes mediante comandos para familiarizarse con el entorno tanto de Mininet como OpenDayLight ya que para hacer pruebas de conectividad del controlador se han utilizado diseños simples, manualmente mediante comandos.

Con el comando `mn` se pueden crear topologías muy sencillas desde un terminal. Si se escribe tan solo `sudo mn`, `sudo` es obligatorio ya que es necesario ejecutar el comando `mn` como administrador, se crea una topología formada por dos *hosts* y un *switch* OpenFlow conectado al controlador.

También se pueden configurar diferentes topologías ya establecidas por defecto. Con `--topo` y varios parámetros se pueden obtener diferentes diseños. A continuación se muestra una tabla con los diferentes parámetros que se pueden añadir.

Comando	**Descripción**
`--topo=minimal`	Configuración de topología formada por dos hosts, un switch y un controlador.
`--topo=single,n`	Configuración de topología formada por un switch, un controlador y n hosts conectados al switch.
`--topo=linear,n`	Topología formada por un controlador, n switches y un host conectado a cada switch.
`--topo=tree, depht = N, fanaut = M`	Topología en forma de árbol con N niveles y M hosts por switch.

Tabla 4.1: Parámetros

Además de `mn` y `--topo` también se pueden añadir otros parámetros de configuración como:

- `-switch`: Para elegir que tipo de *switch* se va a utilizar. Por ejemplo ovs sería *switches* Open VSwitch con los protocolos OpenFlow, Netflow, VLAN ...
- `-controller`: con esta opción se elige el controlador que se va a utilizar.
- `-link`: asigna enlaces entre los dispositivos.
- `-topo`: con esta opción se pueden elegir topologías predeterminadas como tipo árbol, simple, lineal ...
- `-custom`: sirve para leer clases o parámetros de un archivo .py.
- `-mac`: asigna direcciones mac automáticamente.
- `-test`: Esta opción permite realizar comprobaciones como `ping`, `iperf`, `iperfudp`, etc.

Una vez creada la topología y puesta en marcha se pueden utilizar diferentes comandos desde el cliente CLI para realizar diferentes pruebas. Entre los más importantes destacan `help` que permite obtener información sobre el resto y `xterm` que permite abrir un terminal con el dispositivo que se identifique como argumento. En la Figura (4.1) se pueden observar los diferentes comandos disponibles mediante el uso de `help`.

```
EOF    gterm  iperfudp  nodes        pingpair      py      switch
dpctl  help   link      noecho       pingpairfull  quit    time
dump   intfs  links     pingall      ports         sh      x
exit   iperf  net       pingallfull  px            source  xterm
```

Figura 4.1: CLI de mininet

Para la realización del proyecto se necesitarán los siguientes comandos: [17]

- `xterm`: este comando sirve para abrir el terminal de cualquier dispositivos, tanto hosts, como switches y routers.
- `ping`: este comando sirve para utilizar la utilidad ping y poder comprobar la conectividad entre los diferentes dispositivos.
- `nodes`: este comando muestra todos los elementos que forman parte de la red.
- `net`: este comando sirve para ver las conexiones entre las diferentes interfaces de los diferentes dispositivos.

- `dump`: este comando es bastante similar a net aunque incluye las direcciones IP de las interfaces de los dispositivos.
- `iperf`: este comando sirve para comprobar el ancho de banda disponible para cada enlace, también se puede utilizar para comprobar las funcionalidades multicast de la siguiente manera: En el servidor se escribe *iperf -s -u -B 224.1.1.1 -i 1* y en el cliente *iperf -c 224.1.1.1 -u -T 32 -t 3 -i 1*. En los comandos anteriores -i sirve para indicar el intervalo de tiempo que en ese caso es de un segundo entre cada paquete, se podría especificar la longitud de cada paquete UDP mediante -l, también esta -T para indicar el intervalo de vida, -t el tiempo que tarda en transmitir cada petición y -u que sirve para indicar que las tramas son UDP.
- `iperfudp`: Con este comando podemos comprobar el ancho de banda disponible para los paquetes UDP, para la evaluación que se va a realizar en este proyecto se ha comprobado que en todas las topologías el ancho de banda UDP es el mismo (10Mbit/s).

Ventajas y desventajas de Mininet

La utilización de la herramienta mininet tiene una serie de ventajas y desventajas con respecto a otras herramientas similares como podrían ser Estinet o STS.

En primer lugar se identifican las siguientes ventajas: [16]

1. Es muy fácil de poner en marcha, crear una topología y ponerla en funcionamiento puede llevar tan solo un par de segundos.
2. Permite crear redes de forma muy escalable, se pueden añadir hosts y switches de forma muy sencilla.
3. Es capaz de simular anchos de banda de hasta 2 Gbps con un hardware que disponga de la suficiente potencia.
4. Es muy fácil de instalar, tan solo hace falta una máquina virtual con VMware o VirtualBox.
5. Es muy sencillo de reconfigurar gracias a las ayudas que facilita como su interacción con Python y el CLI del que dispone.
6. Se puede conectar a redes reales.
7. Ofrece la posibilidad de disponer de un rendimiento interactivo, de manera que mientras está funcionando la red se pueden modificar los parámetros sin necesidad de reiniciar la emulación.

8. Es compatible con los diferentes tipos de controladores como OpenDayLight o Ryu y también con el protocolo OpenFlow.

9. En los *hosts* simulados de la red se pueden ejecutar programas reales como Wireshark o servidores.

Aunque también presenta algunas limitaciones: [16]

1. Las redes diseñadas con Mininet no pueden exceder la memoria de la CPU o el ancho de banda disponible de un solo servidor.

2. Mininet no puede ejecutar *switches* con el protocolo OpenFlow integrado que no sean compatibles con Linux.

3. Mininet utiliza un solo *kernel* para todos los hosts de la red, por lo que no se pueden ejecutar programas de sistemas operativos diferentes a Linux.

Creación de topologías mediante Python

Este es el apartado más importante en lo que se refiere a mininet ya que las diferentes topologías que se desarrollen para este proyecto se harán mediante la programación en Python ya que es la que mejor permite personalizar una red.

Este API es parecido al de Java ya que cuenta con un conjunto de librerías orientados a los objetos: *host*, *switch* o *router*. Algunos de los métodos que se van a utilizar en el siguiente proyecto son:

- **addHost()**: esta función permite añadir *hosts*. También se puede indicar su dirección IP, MAC y ruta por defecto.
- **addSwitch()**: esta función es similar pero permite añadir *Switches*.
- **addLink**: esta función permite añadir enlaces entre las interfaces que se indiquen.
- **addController()**: esta función permite añadir un controlador.
- **get().start()**: esta función sirve para poner en funcionamiento los diferentes dispositivos.
- **host.cmd()**: esta función sirve para escribir comandos del terminal del dispositivo que se quiera. Por ejemplo, si se quiere añadir una interfaz nueva se podría utilizar el argumento `ifconfig eth2 192.168.1.1/24`.

Herramienta Miniedit

Mininet también cuenta con una interfaz gráfica para diseñar redes, para poder utilizarla tan solo hay que utilizar el comando `sudo python miniedit.py`.

La interfaz permite añadir los diferentes elementos de una red dando la posibilidad también de conectar mediante enlaces. Para poner en funcionamiento la topología que se cree tan solo hay que pulsar el botón de `run` y también se puede habilitar en esta interfaz el CLI configurándolo en la pestaña de preferencias.

4.1.3. Protocolo PIM-SSM en Quagga

Introducción

El protocolo PIM-SSM es el protocolo *multicast* objeto de estudio de este TFG. Como se ha visto en el capítulo 3.3.2 se están proponiendo nuevos protocolos *multicast* para las redes SDN, aunque en el siguiente proyecto se va a evaluar el funcionamiento del mismo protocolo sobre tecnologías diferentes. El protocolo PIM-SSM implica la definición de dos estándares, Source-specific multicast (SSM) y Protocol Independent multicast (PIM).

Originalmente el RFC 1112 [42] *multicast* soportaba una transmisión de un conjunto de orígenes a otro conjunto o de un solo dispositivo a un conjunto. Este protocolo se empezó a conocer como ASM y permitía tener grupos de tráfico con una o varias fuentes. Sin embargo, las redes ASM tenían que localizar todos los destinatarios para un grupo *multicast*. Para ello se emplean dos técnicas:

1. **modo denso**: que consiste en inundar la red para localizar los receptores
2. **modo esparcido**: en este caso el emisor *multicast* permanece a la escucha de los diferentes dispositivos que quieren unirse al grupo *multicast*

En el modo denso se utilizan mensajes PIM-PRUNE para distinguir los *hosts* que se quieren unir al grupo *multicast* de los que no, dichos mensajes indican que se eliminen del árbol *multicast*.

En el modo esparcido los *hosts* se comunican con el servidor con un mensaje `(S,G) Join` indicando que se quieren unir al grupo *multicast*.

PIM-SSM sólo admite un modelo SSM es decir transmisión de uno a muchos y solo crea los árboles de trayectoria más corta (SPT). PIM-SSM pasa

por alto la etapa Rendevous point (RP) gracias a los árboles de distribución compartida. PIM-SSM puede utilizarse técnicamente en toda la gama de direcciones 224.0.0.0/4, aunque la operación PIM-SSM está garantizada solo para 232.0.0.0/8. [26]

Mensajes utilizados en PIM-SSM

Para la realización de evaluación se estudiarán los paquetes intercambiados ejn los dos casos considerados, aunque en ambos escenarios se implementa el mismo protocolo, para quagga los paquetes utilizados son los propios de PIM-SSM mientras que en SDN, gracias al controlador se simula el mismo funcionamiento, aunque ya no se utilizan el mismo tipo de paquetes. En quagga se generan los siguientes mensajes:

- **PIMv2- hello**: este mensaje se utiliza para comprobar la conectividad entre los diferentes dispositivos que soportan multicast.
- **PIMv2- Join/Prune**: mensaje encargado de crear los árboles multicast.
- **IGMP**: La petición de unirse al grupo se realiza mediante un mensaje IGMP.

A continuación, se puede observar un ejemplo de paquetes PIMv2 capturados con la herramienta Wireshark:

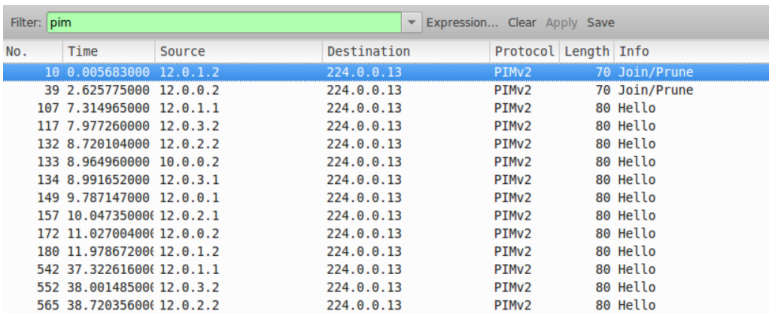

Figura 4.2: Captura de Wireshark.

4.1.4. Protocolo PIM-SSM en OpenDayLight

En el caso de la tecnología SDN tenemos la topología diseñada con mininet y administrada por el controlador OpenDayLight. En este caso para poder implementar el protocolo PIM-SSM es necesario utilizar el protocolo

de comunicaciones entre el controlador y la red OpenFlow. Se implementa código en C con la ayuda de la herramienta Eclipse. Para el caso de la primera aproximación, la simulación de la tecnología actual usada en redes, ya viene implementado todo el código necesario y para poder evaluar las nuevas redes habrá que generalizar dicho código aunque se realización será bastante sencilla.

El protocolo OpenFlow es un protocolo de acceso libre y estandarizado. Se encarga de las comunicaciones entre la capa de datos y la capa de control según la ONF [23]. Gracias al protocolo OpenFlow una red puede gestionarse como un todo y no como un número de dispositivos, los cuales necesitan gestionarse individualmente.

La solución que OpenFlow propone [20] consiste en separar el tráfico en los diferentes dispositivos de la capa de datos (*switches* o *routers*) según las características de los diferentes tipos de tráfico, para ello se utilizan tablas de flujo parecidas a las que utilizan los *switches* Ethernet.

Funcionamiento

El protocolo OpenFlow se encarga de cumplir las funciones de protocolos como STP, SPB o TRILL. Cualquier cambio que se produce en la red es informado al controlador mediante mensajes del protocolo OpenFlow como puede ser la caída de un enlace o el cambio del camino de un flujo determinado. OpenFlow también se encarga de modificar los valores de las tablas de flujo de los switches o routers de la red que el controlador ordena.

Switch OpenFlow

Los switches Ethernet utilizan unas tablas de flujo para implementar firewalls, Network Address Translation (NAT), QoS y recolectar estadísticas. En las redes actuales no SDN, cada fabricante utiliza un modelo de tablas de flujo, la idea de los switches OpenFlow es proporcionar un estándar de tablas de flujo para que las empresas puedan desarrollar nuevos protocolos de manera sencilla y en resumen conseguir una mayor interoperabilidad y compatibilidad. Con las nuevas tablas de flujo los administradores de la red pueden separar el tráfico entre producción y dedicado a investigación. De manera, que se puedan implementar nuevos modelos de seguridad, esquemas de direccionamiento, alternativas para IP y desarrollar un nuevo modelo de VLANs.

La función principal de las tablas de flujo es establecer los caminos de datos asociando en dichas tablas una acción para cada entrada de datos. Para poder utilizar el protocolo OpenFlow en un *switch* tiene que cumplir unos requerimientos básicos: disponer de una tabla de flujos y tener un

canal seguro conectado al controlador. En la Figura (4.3) se puede observar el diseño de un *switch* OpenFlow [22].

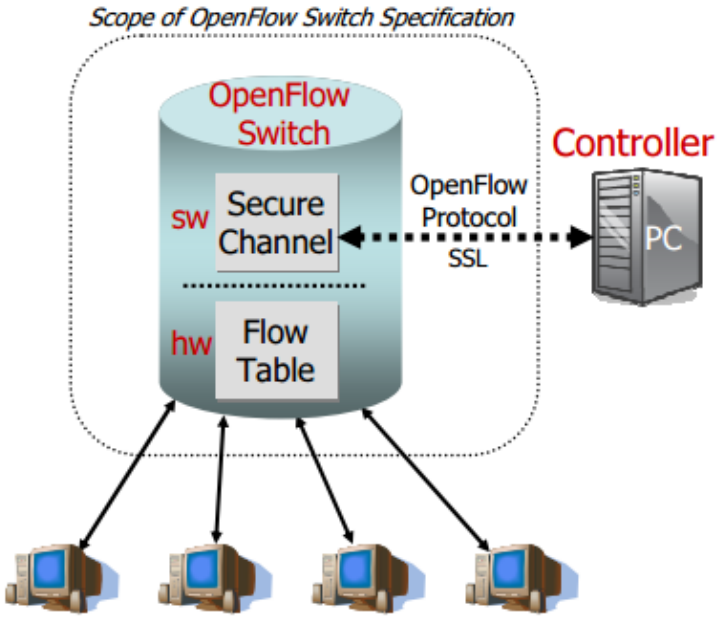

Figura 4.3: Switch OpenFlow.[[39]]

Existen dos modelos de *switches* OpenFlow, *Switches* OpenFlow-only y *Switches* OpenFlow-híbridos.

- **Switches OpenFlow-only**: este tipo contienen múltiples tablas de flujo y los flujos están ampliamente definidos.
- **Switches OpenFlow-híbridos**: estos switches soportan tanto las tablas de flujo de OpenFlow como el Switchin Ethernet convecional.

Un switch OpenFlow está formado por dos partes:

1. Tabla de flujos: cada entrada de la tabla de flujos tiene una acción asociada e indica al *switch* como se debe procesar un flujo.
2. Canal seguro que conecte el *switch* a un controlador: permite la conexión entre el *switch* y el controlador mediante el protocolo OpenFlow.

Tabla de flujos

Todas las entradas de flujo tienen el mismo formato y se identifican por sus coincidencias o prioridad. Los campos definidos son:

1. Una cabecera del paquete que define el tipo de flujo.
2. La acción de cómo se deben procesar los paquetes.
3. Un campo que recoge las estadísticas del número de paquetes, bytes de cada flujo y el tiempo desde que se envió el último paquete.

Además, una tabla de flujos puede tener las siguientes entradas:

- **Match fields:** Este tipo de entrada comprueba que un paquete recibido coincide con la entrada de flujo, comprueba el puerto y la cabecera del paquete.
- **Priority:** Comprueba la prioridad de los paquetes.
- **Counters:** contador que lleva un seguimiento del número de entradas, puertos ...
- **Instructions:** esta acción modifica el conjunto de acciones que se tienen que realizar con un paquete.
- **Timeouts:** controlan el tiempo máximo que puede tardar un switch procesando un paquete antes de descartarlo.
- **Cookie:** Sirve para filtrar las estadísticas de flujos.

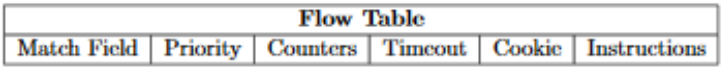

Figura 4.4: Tabla de flujos.

Tabla de grupos

Los grupos permiten una manera eficiente de definir un conjunto de acciones para un grupo de flujos. Esta utilidad permite implementar de manera sencilla multicast y unicast. Cada entrada está formada por los siguientes campos:

- Identificador de grupo: sirve para definir cada grupo.
- Tipo de grupo: determina la semántica de cada grupo.
- Action buckets: lista de acciones.

Se diferencian cuatro tipos de grupos:

- Todos: Este tipo de grupos pertenece a los flujos que se transmiten mediante multicast o broadcast. Se clonan los paquetes y se envía uno a cada destinatario.
- Seleccionado: Los paquetes son enviados a un cubo único de grupo, se basa en un algoritmo de selección de interrupción.
- Indirecto: permite múltiples flujos o grupos soportando convergencia más rápida. Este tipo de grupo es similar al grupo Todos solo que con un solo cubo.
- Conmutación por error: Cada acción se asocia con un puerto específico. Esto permite al switch cambiar la transmisión sin necesitar una consulta al controlador.

Open vSwitch

Open Vswitch es un software de código abierto bajo la licencia de Open Source Apache 2.0 [24]. Está diseñado para permitir la automatización masiva de la red a través de una extensión programable, a la vez que soporta interfaces y protocolos estándares de administración como (NetFlow, sFlow, IPFIX, LACP, etc).

Open vSwitch está adaptado para que funcione como un *switch* virtual y de esta manera facilitar el desarrollo de la tecnología SDN y del protocolo OpenFlow permitiendo la simulación de switches y routers OpenFlow.

Mensajes de OpenFlow

Este apartado es el más importante en lo que respecta a OpenFlow ya que nos permitirá definir qué mensajes se utilizan de este protocolo en el escenario SDN montado y realizar una correcta evaluación. Los mensajes que nos encontraremos de este protocolo son los siguientes: [21]

1. **Ofpt_multipart_request/reply:** Los mensajes `multipart` son para establecer los grupos de transmisión, realmente en el entorno SDN no se utiliza exactamente el protocolo PIM-SSM aunque la idea es la misma. En vez de establecer un árbol multicast con los mensajes join/prune se establece una configuración de las tablas de flujo en los diferentes *switches* que conceptualmente es lo mismo que hacen los mensajes join/prune de la aproximación clásica sin SDN. Para establecer una analogía con respecto a quagga los grupos *multicast* se establece gracias a los siguientes mensajes multipart.

- **Ofpmp_group:** para solicitar la creación de un grupo, en este caso el grupo multicast que se forma.
- **Ofpmp_flow:** para conseguir estadísticas de un flujo de entrada.
- **ofpmp_meter_config:** Este mensaje es el que sirve para establecer la configuración necesaria en los switches para las transmisiones multicast.
- **ofpmp_table:** Cuando el controlador necesita conocer estadísticas del estado de flujos envía este mensaje para recibir una respuesta con el número de paquetes que se están procesando, cuantos se están perdiendo y demás información. Este mensaje sirve para evitar saturación.
- **ofpmp_port_stats:** Este mensaje sirve para comprobar el estado de los diferentes puertos de los *switches*. También puede modificar la configuración de algún puerto como puede ser cambiar el puerto por el que transmite la trama *multicast* debido a la caída de un enlace.
- **ofpmp_queue:** Este mensaje multipart proporciona estadísticas de colas para los diferentes puertos.
- **ofpmp_meter_config:** Este mensaje sirve para modificar la configuración de varios parámetros como, por ejemplo, el campo meter_id.
- **ofpmp_table:** Con este mensaje el controlador obtiene la información de las tablas de los *switches*.
- **ofpmp_port_stats:** Este mensaje sirve para obtener información sobre el estado de los diferentes puertos de los switches.

2. **Ofpt_packet in:** Este mensaje lo envían los *switches* al controlador y sirve para enviar información del estado de los switches al controlador.

3. **Ofpt_packet out:** Este mensaje se envía desde el controlador hacía los *switches* y sirve para redirigir un paquete hacia un puerto específico del *switch*.

4. **Ofpt_echo_request/reply:** Estos mensajes se utilizan para comprobar la conectividad entre los diferentes dispositivos. `Request` es la petición de información y `reply` la respuesta.

En la siguiente captura de paquetes con la ayuda de Wireshark se muestran los diferentes paquetes del protocolo OpenFlow:

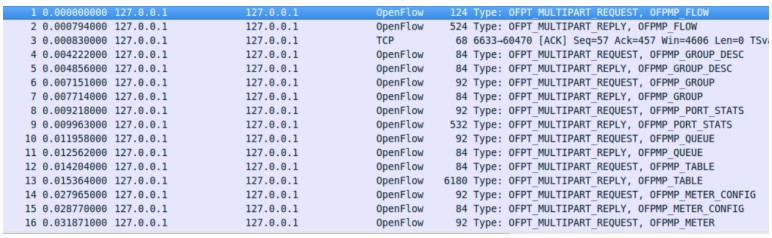

Figura 4.5: Captura de Wireshark.

Uno de los aspectos más importantes que hay que tener en cuenta a la hora de realizar una evaluación de ambas aproximaciones es notar que mientras que Quagga solo envía los paquetes necesarios para funcionar de una forma determinada (si solo se utiliza el protocolo *multicast* no se generarán paquetes para comprobar el estado de los enlaces o cualquier otro tipo de paquetes), SDN funciona como una red real (Con una serie de paquetes y protocolos típicos de redes comerciales) por lo que a la hora de evaluar la carga que se produce en ambos diseños habrá que tener este aspecto en cuenta y medir por separado los paquetes *multicast* que se generan del resto, en el caso de SDN. Otros paquetes que también se tienen que obviar son los producidos por Quagga al iniciarse los diferentes demonios ya que en un entorno real no serían necesarios.

4.1.5. Eclipse

Eclipse [11] es un programa informático formado por varias herramientas de programación de código abierto multiplataforma que sirve para poder generar los paquetes que el controlador utiliza para operar. El lenguaje de programación que se utiliza para el siguiente proyecto es C y en este caso tan solo será necesario modificar el código desarrollado durante [47] para generalizar el comportamiento del controlador para diferentes topologías de red.

Esta plataforma integra el entorno de desarrollo integrado IDE de Java denominado JDT (Java Development Toolkit) e incluye un compilador ECJ.

Eclipse ha sido desarrollado por IBM y es el sucesor de VisualAge, en la actualidad la Fundación Eclipse se encarga de desarrollar la herramienta Eclipse.

4.2. Herramientas necesarias para la evaluación

Para realizar una evaluación de ambas aproximaciones y poder obtener conclusiones sobre las ventajas de utilizar la tecnología SDN para la trans-

misión multicast es necesario utilizar herramientas que permitan recoger información sobre el funcionamiento de ambas redes.

Los tres aspectos principales en los que se va a basar este proyecto son la robustez, la escalabilidad y la carga generada. Para poder realizar una evaluación objetiva es necesario recoger información relacionada con el número de paquetes que circulan por la red; obteniendo tamaños en bytes, velocidades de transmisión de paquetes y demás información relacionada directamente con el tráfico de la red. Para ello la herramienta más útil es Wireshark, la cual da la posibilidad de recoger todo el tráfico generado, proporcionando información sobre dicho tráfico y una gran variedad de opciones a la hora de visualizar el contenido. También se utilizará de forma breve la aplicación Iperf para el estudio del ancho de banda de los enlaces.

4.2.1. Wireshark

Wireshark es un analizador de protocolos utilizado para realizar análisis de tráficos y solucionar problemas en redes de comunicaciones.

La funcionalidad que proporciona es parecida a la de la herramienta tcpdump, pero con una interfaz gráfica y varias opciones de visualización de la información recogida. Permite examinar datos de redes activas y también permite filtrar la información recogida.

En este proyecto se va a capturar todo el tráfico mediante la herramienta de Wireshark y para analizarlo se utilizarán las opciones de la ventana *Estadísticas* entre las que destacan: [41]

Item del menu	**Descripción**
summary	Muestra información sobre los paquetes capturados.
Protocol Hierarchy	Muestra la información de los protocolos en forma jerárquica de árbol.
Conversations	Muestra los paquetes separando en conversaciones según los nodos origen y destino que tenga cada paquete.
EndPoints	Muestra una lista de nodos finales.
Packet Lengths	Muestra los paquetes ordenados según su tamaño.
IO Graphs	Permite la posibilidad de mostrar diferentes tipos de gráficas como el número de paquetes por tiempo.
Service Response Time	Permite calcular el tiempo que tarda en enviarse y recibirse la respuesta de un paquete.

Tabla 4.2: Opciones de Wireshark.

Capítulo 5

Implementaciones para la evaluación

En este capítulo se van a detallar las configuraciones realizadas durante el proyecto para poder realizar la evaluación.

Este proyecto considera como punto de partida el TFG realizado por el alumno Carlos Santamaría Espinosa [47], en el que se propone la configuración de dos topologías mediante Mininet, los archivos Quagga para que en el escenario sin SDN funcione el protocolo multicast PIM-SSM y un paquete *sdnhub-tutorial-learning-switch* para que el controlador ejecute las acciones necesarias para implementar el protocolo *multicast* PIM-SSM en la red SDN.

Toda esa configuración se obtiene a través de una máquina virtual. En el presente proyecto se añaden las siguientes características: cuatro topologías más con la herramienta mininet, se añade a la herramienta Quagga un protocolo de enrutamiento dinámico OSPF y para el controlador se modifica el código del paquete *sdnhub-tutorial-learning-switch* de manera que pueda funcionar de forma genérica para cualquier tipo de topología.

5.1. Diseño de diferentes topologías mediante Mininet

Como el funcionamiento de la herramienta Mininet en 5.3 ya se ha explicado, en este apartado solo se detallarán las topologías implementadas. En primer lugar partimos del TFG mencionado donde se diseñó la siguiente topología.

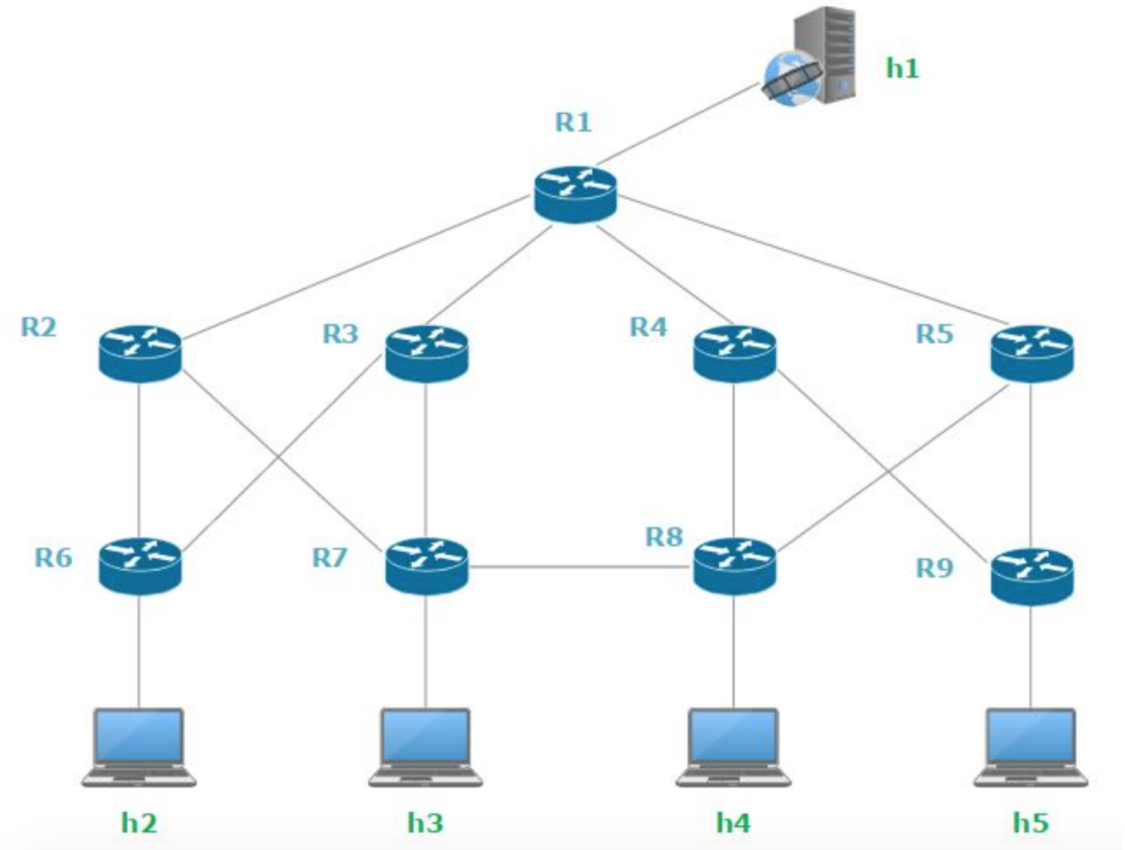

Figura 5.1: Topología árbol.

5.1.1. Topología en anillo

Una vez estudiado el código en Python, se diseña una topología formada por 9 *routers* o *switches* -en el caso de SDN- en forma de anillo como se puede ver en la Figura (5.2).

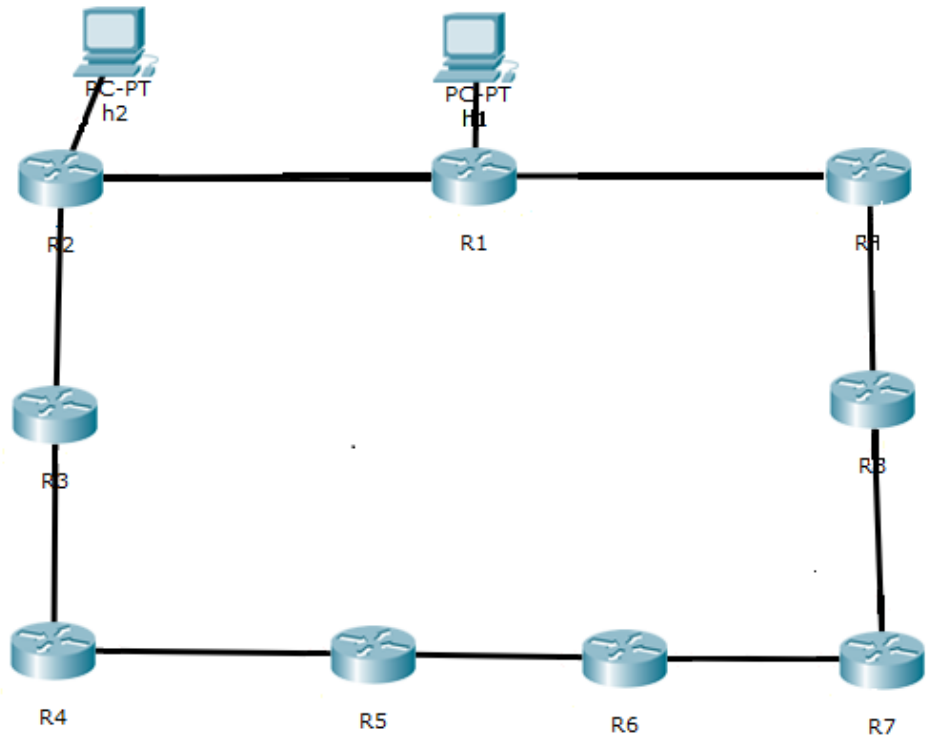

Figura 5.2: Topología en forma de anillo.

Esta topología se diseña con el objetivo de estudiar la robustez de ambas aproximaciones. El *host* h1 hace de servidor, mientras que h2 solicita unirse al grupo *multicast*, en un comienzo la transmisión se hará mediante el enlace que une R2 con R1. Cuando se corte dicho enlace, en los dos escenarios evaluados, se reestablecerá la ruta de encaminamiento gracias a los diferentes protocolos de enrutamiento dinámico y la comunicación se hará a través de todos los *routers*. Desde el router R2 hasta R1 pasando por R3 y el resto de *routers*.

Si se estudia la robustez para la configuración anterior 5.1, el tiempo en establecerse otra ruta es muy pequeño y apenas se pueden apreciar diferencias entre ambas redes. Mientras que en este tipo de topologías el tiempo en volver a establecerse una nueva ruta es mayor debido a todas las rutas que tienen que recorrer todos los paquetes. Por eso se elige esta nueva topología para estudiar la robustez del protocolo PIM-SSM en ambas redes.

Para el caso de Quagga en mininet hay que añadir el siguiente código 5.3, para activar los diferentes procesos (o *demonios*). En nuestro proyecto se utilizaran los demonios `qpimd, zebra y ospfd`. El primero para simular el protocolo PIM-SSM, `ospfd` para añadir enrutamiento dinámico OSPF y `zebra` para controlar el resto de demonios y configurar todas las interfaces

de los *routers*. También se puede observar en el código que se accede a un archivo .pid, esto se hace para asignarle una identificación a cada proceso y que no los asigne el ordenador de forma aleatoria; gracias a esto se consigue que no se pierdan los procesos con el tiempo y se pueda mantener en funcionamiento la red durante el tiempo que se necesite.

```
    for router in net.switches:
        router.cmd(" /usr/local/quagga/sbin/zebra -f /usr/local/quagga/etc/red2_zebra--%s.conf -d -
i /usr/local/quagga/etc/zebra--%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router.name,
router.name))
        router.waitOutput()
        router.cmd("/usr/local/quagga/sbin/pimd -f /usr/local/quagga/etc/red2_pimd--%s.conf  -d -i /
usr/local/quagga/etc/pimd--%s.pid > log/%s--pimd-stdout 2>&1" %(router.name,
router.name,router.name), shell=True)
        router.waitOutput()
        router.cmd(" /usr/local/quagga/sbin/ospfd -f /usr/local/quagga/etc/red2_ospf--%s.conf -d -i /
usr/local/quagga/etc/ospf--%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router.name,
router.name))
        router.waitOutput()
        log("Starting zebra ospf and pimd on %s" % router.name)
```

Figura 5.3: Código Python.

Para configurar las interfaces de los *hosts* y añadir sus *default gateway* se implementa el siguiente código.5.4

```
net.hosts[0].cmd("ifconfig h1-eth0 10.0.0.1")
net.hosts[1].cmd("ifconfig h2-eth0 6.0.0.1")
net.hosts[0].cmd("route add default gw  10.0.0.2")
net.hosts[1].cmd("route add default gw  6.0.0.2")
```

Figura 5.4: Configuración de hosts mediante función cmd.

El código anterior utiliza la función cmd() para acceder a los terminales de los diferentes dispositivos. Esta función se utiliza también para capturar tráfico Wireshark, aunque no se muestra aquí, ya que para cada captura diferente se modifica el código.

Por ejemplo, con la siguiente instrucción: `wireshark -i2 -k -f \host 192.168.1.5" -s512 -w captura_ejemplo.pcap -afilesize:10` se realiza una captura de todos los paquetes con la dirección IP de origen o destino 192.168.1.5 y se guarda en un archivo llamado captura_ejemplo.pcap.

Para establecer los enlaces, se utiliza la función `addLink()`, en el caso de la red anillo se crean los siguientes enlaces. 5.6

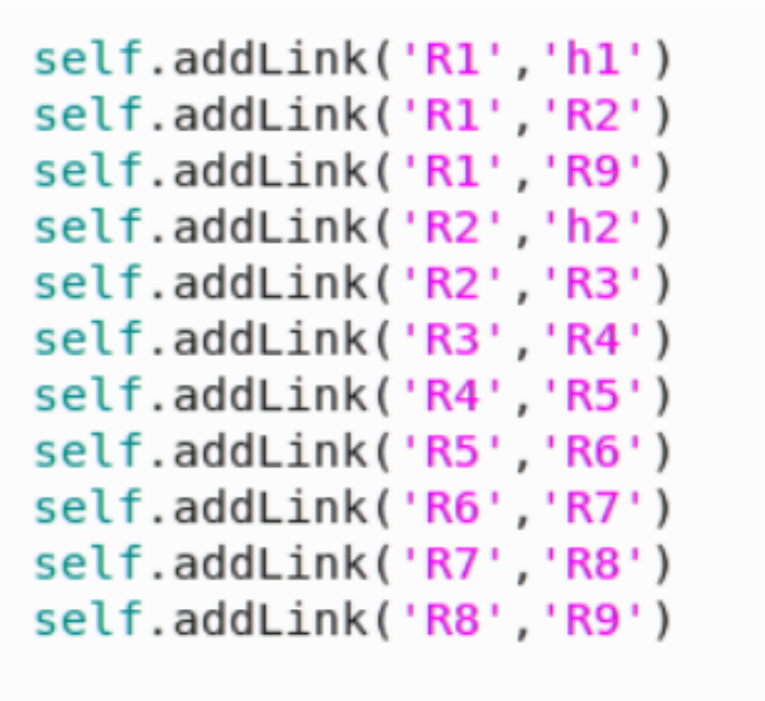

Figura 5.5: Creación de enlaces en la topología anillo.

Para el caso de SDN no es necesario activar los demonios pero hay que enlazar la red con el controlador con el siguiente código.

```
info('*** Adding controller\n')
c0=net.addController(name='c0', controller=RemoteController,protocols='OpenFlow13', ip='127.0.0.1')
```

Figura 5.6: Conexión con el controlador.

Para SDN, es necesario también añadir las direcciones MAC para que el controlador se pueda comunicar con los *switches* o *hosts*. 5.7

```
s1=net.addSwitch('s1', protocols='OpenFlow13',mac='00:00:00:00:00:06')
s2=net.addSwitch('s2', protocols='OpenFlow13',mac='00:00:00:00:00:07')
s3=net.addSwitch('s3', protocols='OpenFlow13',mac='00:00:00:00:00:08')
s4=net.addSwitch('s4', protocols='OpenFlow13',mac='00:00:00:00:00:09')
s5=net.addSwitch('s5', protocols='OpenFlow13',mac='00:00:00:00:00:10')
s6=net.addSwitch('s6', protocols='OpenFlow13',mac='00:00:00:00:00:11')
s7=net.addSwitch('s7', protocols='OpenFlow13',mac='00:00:00:00:00:12')
s8=net.addSwitch('s8', protocols='OpenFlow13',mac='00:00:00:00:00:13')
s9=net.addSwitch('s9', protocols='OpenFlow13',mac='00:00:00:00:00:14')

info('***adding host\n')
h1=net.addHost('h1', cls=Host,mac='00:00:00:00:00:01')
h2=net.addHost('h2', cls=Host,mac='00:00:00:00:00:02')
```

Figura 5.7: Configuración dispositivos.

Por último, en SDN es necesario activar los correspondientes *switches*, a diferencia de en Quagga, en la cual ya se encarga el demonio `zebra` de activar los *routers*, con el siguiente comando. 5.8

```
info('***Starting switches\n')
net.get('s1').start([c0])
net.get('s2').start([c0])
net.get('s3').start([c0])
net.get('s4').start([c0])
net.get('s5').start([c0])
net.get('s6').start([c0])
net.get('s7').start([c0])
net.get('s8').start([c0])
net.get('s9').start([c0])
```

Figura 5.8: Activación de los switches.

Mientras que en SDN tenemos direcciones MAC para los *switches*, en Quagga con ayuda del demonio `zebra` se establecerá en cada *router* las diferentes interfaces con direcciones IP asignadas como se verá en el apartado 5.2.

El código completo para los dos escenarios se puede ver en los diferentes apéndices A, tanto el de las redes anillo como el del resto, diseñadas en este proyecto.

5.1.2. Topología en árbol para estudiar escalabilidad

Con la red diseñada en el TFG precedente, se puede estudiar el tráfico generado cuando varios dispositivos piden unirse a un grupo *multicast*. Para poder ver cuál de las dos aproximaciones es más escalable, igualmente se diseña una topología en forma de árbol pero añadiendo más dispositivos. De manera, que al estudiar el tráfico tanto en la red en árbol con cinco *hosts* como en la red en árbol con diecisiete *hosts* se puede observar quién es más escalable. A continuación, se muestra la topología en árbol con diecisiete *hosts* y trece *routers* 5.9. En el caso de SDN sería sustituir los *routers* por *switches*.

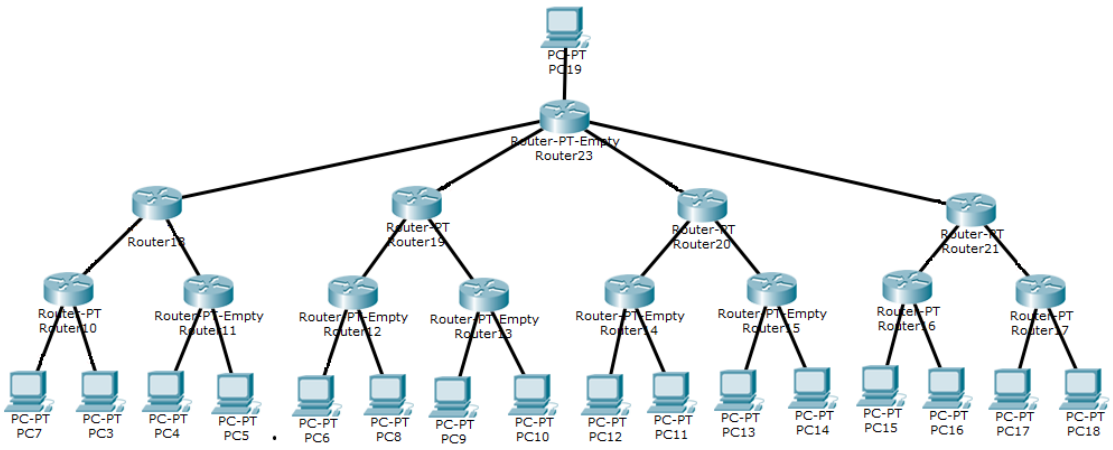

Figura 5.9: Topología en forma de árbol.

El código de ambas topologías también se puede ver en los apéndices y cómo añadido se puede observar el apartado donde se configuran los *hosts* con sus direcciones MAC para el caso de SDN.5.10

```
info('***adding host\n')
h1=net.addHost('h1', cls=Host,mac='00:00:00:00:00:01')
h2=net.addHost('h2', cls=Host,mac='00:00:00:00:00:02')
h3=net.addHost('h3', cls=Host,mac='00:00:00:00:00:03')
h4=net.addHost('h4', cls=Host,mac='00:00:00:00:00:04')
h5=net.addHost('h5', cls=Host,mac='00:00:00:00:00:05')
h6=net.addHost('h6', cls=Host,mac='00:00:00:00:00:06')
h7=net.addHost('h7', cls=Host,mac='00:00:00:00:00:07')
h8=net.addHost('h8', cls=Host,mac='00:00:00:00:00:08')
h9=net.addHost('h9', cls=Host,mac='00:00:00:00:00:09')
h10=net.addHost('h10', cls=Host,mac='00:00:00:00:00:10')
h11=net.addHost('h11', cls=Host,mac='00:00:00:00:00:11')
h12=net.addHost('h12', cls=Host,mac='00:00:00:00:00:12')
h13=net.addHost('h13', cls=Host,mac='00:00:00:00:00:13')
h14=net.addHost('h14', cls=Host,mac='00:00:00:00:00:14')
h15=net.addHost('h15', cls=Host,mac='00:00:00:00:00:15')
h16=net.addHost('h16', cls=Host,mac='00:00:00:00:00:16')
h17=net.addHost('h17', cls=Host,mac='00:00:00:00:00:17')
```

Figura 5.10: Diseño de hosts.

Una vez diseñadas las topologías mediante Mininet, hay que configurar todos los archivos de los diferentes *demonios* de Quagga y para SDN

modificar el código del controlador.

5.2. Quagga

5.2.1. Introducción

Quagga [13] es una herramienta para añadir servicios de enrutamiento TCP/IP. Utiliza una arquitectura software muy avanzada que permite proveer servicios de enrutamiento de alta calidad. Los protocolos de enrutamiento más importantes que incluye son Routing Information Protocol (RIP) [29], BGP [43], Intermediate system to intermediate system (IS-IS) [28] ,PIM Sparse Mode (PIM-SM) [1] y OSPF [2].

Quagga ofrece una interfaz de usuario para configurar dispositivos aunque también se pueden crear archivos con extensión *.conf* para realizar la configuración de los dispositivos, estos archivos hacen más cómoda la configuración de redes con multitud de dispositivos y va a ser la forma en la que se configure Quagga en el siguiente proyecto.

Para implementar los protocolos Quagga utiliza *daemons*, en nuestro caso se van a utilizar los *daemons*: `qpimd`, `ospfd` y `zebra`.

- `zebra`: este demonio es el encargado de la administración de todos los protocolos, interactúa directamente con el kernel y sirve para configurar las interfaces de los dispositivos añadiendo sus direcciones IP. También se pueden añadir rutas estáticas con ayuda del demonio `zebra` aunque en este proyecto se utilizará `ospfd`.
- `qpimd`: este demonio se encarga de simular el funcionamiento del protocolo *multicast* PIM-SSM.
- `ospfd`: este demonio se encarga del enrutamiento dinámico.

La arquitectura general de Quagga se puede ver en la Figura 5.11.

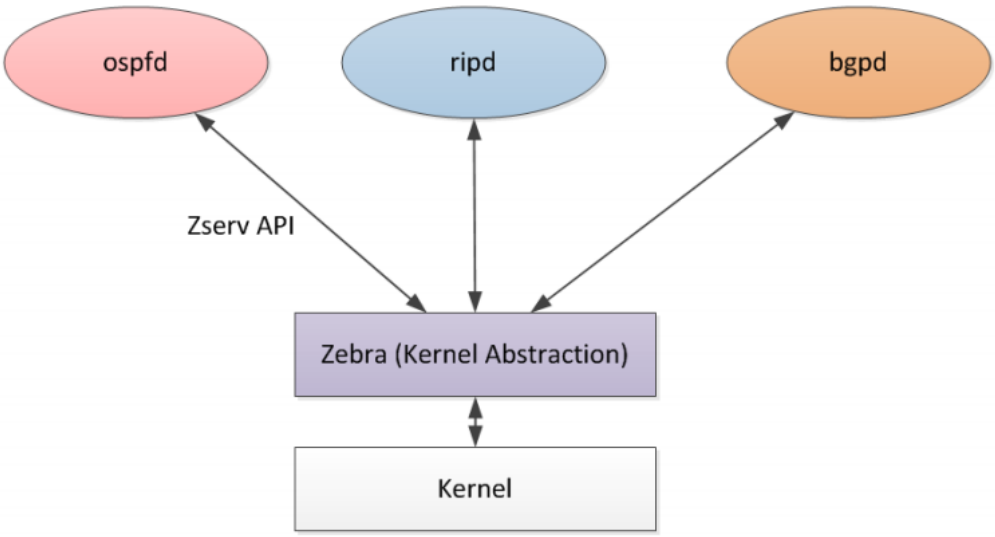

Figura 5.11: Arquitectura de la herramienta Quagga.

5.2.2. Configuración de Quagga

Para configurar los diferentes *demonios* en cada router de la red es necesario añadir los siguientes comandos (en el archivo Python de mininet) dentro de un bucle que va recorriendo todos los archivos *.conf* de los routers:

```
router.cmd(/usr/local/quagga/sbin/zebra -f /usr/local/quagga/etc/zebra--%s.conf
-d -i /usr/local/quagga/etc/zebra--%s.pid"(router.name,router.name))
```

router.cmd(/usr/local/quagga/sbin/pimd -f /usr/local/quagga/etc/pimd–%s.conf -d -i /usr/local/quagga/etc/pimd–%s.pid ”(router.name, router.name), shell=True)

```
router.cmd(/usr/local/quagga/sbin/ospfd -f /usr/local/quagga/etc/ospf--%s.conf
-d -i /usr/local/quagga/etc/ospf--%s.pid"(router.name,router.name))
```

En este proyecto se utilizarán los archivos *.conf* aunque otra forma de configurar los servicios es utilizando el siguiente comando `telnet localhost "puerto del demonio"` siendo 2601 para `zebra`, 2604 para `ospfd` y 2611 para `pimd`.

Zebra

Para configurar `zebra` hay que crear un archivo *.conf* por cada router de la red. En la imagen 6.3 se puede ver un ejemplo de configuración. Hay que indicar las interfaces con sus direcciones IP y en caso de no disponer de un protocolo de enrutamiento dinámico, se pueden añadir rutas estáticas en estos archivos.

```
hostname R1
password en
enable password en

!

interface lo
  ip address 127.0.0.1/32

interface R1-eth1
  ip address 10.0.0.2/24

interface R1-eth2
  ip address 20.0.0.1/24

interface R1-eth3
  ip address 20.0.8.2/24

!


log file /tmp/R1.log
```

Figura 5.12: Fichero Zebra.

Qpimd

Al igual que `zebra`, para configurar el protocolo PIM-SSM en la red hay que crear un archivo *.conf* por cada router como en el ejemplo 5.13. En este caso hay que indicar que interfaces se habilitan para transmitir multicast con el comando `ip pim ssm` y si alguna interfaz está conectada directamente con un host hay que añadir el comando `ip igmp` para activar la funcionalidad de IGMPv3.

```
hostname R1
password zebra
!enable password zebra
!
interface R1-eth1
ip pim ssm
ip igmp
!
interface R1-eth2
ip pim ssm
!
interface R1-eth3
ip pim ssm
!
interface R1-eth4
ip pim ssm
!
interface R1-eth5
ip pim ssm
!
ip multicast-routing
!
line vty
!
end
```

Figura 5.13: Fichero Qpimd.

Ospfd

Para el caso del protocolo OSPF hay que indicar también las interfaces de cada router, habilitar el protocolo OSPF mediante la frase `router ospf` e indicar las redes conectadas directamente al router. Por último hay que indicar a que área pertenece cada red, en este proyecto se ha utilizado el área *backbone* ya que las redes no son muy complejas y no interesa dividir en secciones como sería el caso de VLANs. A continuación se puede ver un ejemplo de un archivo de configuración.6.8

```
hostname ospfd
password zebra
log stdout

interface R1-eth1
interface R1-eth2
interface R1-eth3
interface R1-eth4
interface R1-eth5

router ospf
network 10.0.0.0/24 area 0
network 12.0.0.0/24 area 0
network 12.0.1.0/24 area 0
network 12.0.2.0/24 area 0
network 12.0.3.0/24 area 0

line vty
```

Figura 5.14: Fichero Ospf.

En el anexo B se puede ver las tablas IPv4 de los dos escenarios evaluados.

5.3. Configuración del controlador OpenDayLight

En este apartado primero se explica la lógica implementada en el controlador OpenDayLight mediante la herramienta Eclipse para simular un entorno con capacidad de transmitir tramas *multicast*.

Una vez explicado el paquete de trabajo en Eclipse, realizado en [47].

Con sus diferentes funciones y clases programadas en Java, se detallarán los cambios necesarios para adaptar el código a cualquier topología de red.

Como el código no se ha realizado por completo en este proyecto tan solo se mostrarán las funciones modificadas durante este proyecto y se mencionarán las variables nuevas necesarias para la adaptabilidad del código.

5.3.1. Lógica del código

El código implementado parte del paquete *learning-switch* como punto de partida. Este paquete ya trae la lógica necesaria para realizar *routing* y transmisión unicast.

Las nuevas funcionalidades que hay que añadir al paquete *learning-switch* son: que el controlador reconozca las peticiones IGMPv3 enviadas por los *hosts*, implementar el algoritmo de Dijkstra [3] para el *routing multicast* e implementar un método para la creación del árbol de distribución.

Para la creación de grupos *multicast* es necesario que el controlador sea capaz de distinguir los mensajes IGMP, de esta manera los *hosts* podrán unirse o darse de baja en los grupos *multicast*. Lo primero que se hace es crear una regla en todos los *switches* para que los mensajes IGMP de los *hosts* los envíen directamente al controlador. Esto se hace mediante el comando:

ovs-ofctl add-flow -00penFlow13 (nombre del switch)
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller

Para no tener que escribir todas las reglas cada vez que se ejecuta la red se crea un archivo *.sh* para agilizar el proceso de configuración. Una vez solucionado este problema se configura el controlador para que pueda procesar los paquetes IGMP. Para ello se utiliza la clase *PacketUtils.java* que dispone de métodos para descomponer los paquetes y obtener información como la dirección MAC de un paquete de origen o destino. Se modifica ligeramente la clase *PacketUtils.java* para que sea capaz de analizar el campo *Record type* que es el que interesa para definir los grupos multicast y se añaden dos métodos nuevos al código:

- *multicastAddFlow*
- *multicastDeleteFlow*

Estos métodos nuevos añaden o eliminan las entradas de flujo en los switches para el routing multicast.

Una vez se reconocen los paquetes IGMP es necesario implementar un algoritmo de *routing multicast*, en este caso se ha utilizado el algoritmo de Dijkstra.

Primero se crean 4 clases nuevas para dibujar un grafo de la topología y definir las rutas más cortas en función de los enlaces. Las clases creadas son:

- **Clase Graph**: Esta clase guarda el grafo de la topología.
- **Clase Vertex**: Crea los vértices de la topología.
- **Clase Edge**: Compara los enlaces en función del peso de cada uno para obtener los caminos más cortos.
- **Clase Dijkstra**: Esta clase ejecuta el algoritmo completo de Dijkstra.

Una vez definidas las clases en la clase principal *TutorialL2Forwarding* se crean nuevos métodos `GetPath2`, `whoIs`, `getHost` y `getSwitches2` que se encargan del *routing multicast*. Los métodos nuevos se explican en los siguientes apartados ya que son los que hay que modificar.

Por último, se implementa también en `TutorialL2Forwarding` el método `modifyL3FlowSeveralOutputConnectors()` que sirve para añadir o eliminar los flujos en los *switches*.

5.3.2. Modificaciones del código

En el proyecto [47] se configuró el controlador para funcionar solo con la topología sencilla que se utilizo en el mismo, por lo que para poder realizar este proyecto es necesario modificar algunos métodos del código. De forma resumida; los métodos que más hay que modificar son: `whoIs`, `getPath2`, `getSwitches` y `getHost2`.

A parte de los métodos mencionados anteriormente, hay que modificar las llamadas a dichos métodos en otras partes del código y añadir variables nuevas como *nodes* que es una cadena tipo *String* que recoge en cada componente el identificador de cada *host* y *nodes_switch* que también es una cadena tipo *String* con el identificador de cada *switch*.

Método whoIs(srcIP,nodes)

Este método sirve para devolver el identificador de un elemento concreto, para adaptarlo a cualquier tipo de topología se añade un bucle que recorre cada dispositivo con el método `getHost`, el cuál devuelve el identificador de cada dispositivo y posteriormente se compara con el dato que se pasa al método para ver a cual corresponde.

```
1     public static int whoIs(String srcIP,String[] nodes) {
```

```
        String srcIP_2[]=new String[nodes.length];
        int client = 0;
        for (int i= 0; i < nodes.length; i++){

           srcIP_2[i]="InetAddress.getByName(getHost(i))";
        }

         for (int i= 0; i < nodes.length; i++){
            if(srcIP.compareTo(srcIP_2[i]) == 0){
                client=i;
            }
        }

        return client;


    }
```

Método getPath2(srcNode,dstNode,nodes,nodes_switch,enlaces)

El método `GetPath2` sirve para calcular la ruta más corta desde la fuente *multicast* hacia el cliente que realiza la petición, para ello se ayuda del algoritmo de Dijkstra. Para poder utilizar este método para cualquier tipo de topología es necesario darle las variables `nodes`, `nodes_switch` y `enlaces`. La variable `nodes` es de tipo *String* y contiene la identificación de cada *host* mientras que `nodes_switch` contiene la indentificación de cada *switch*; `enlaces` es un vector que contiene el número de enlaces de cada dispositivo empezando por los *hosts*.

A parte de añadir estas nuevas variables hay que cambiar la lógica del método añadiendo un doble bucle que vaya creando cada vértice con la clase *Edge* para cada enlace de la red y un nuevo bucle que calcule el número de *switches* que contiene la red y almacene este dato en una variable *total*.

```
    public static String[] getPath2(int srcNode, int dstNode, String[]
         nodes,String[] nodes_Switch,int[] enlaces ) {



      Graph graph = new Graph();
      Vertex[] vertices = new Vertex[nodes_Switch.length];
      int y = 1;

      for(int i = 0; i < vertices.length; i++) {

         vertices[i] = new Vertex(y + "");
         graph.addVertex(vertices[i], true);
         y = y + 1;

      }

 switches.
      int total=0;
      for(int i=0;i<enlaces.length-nodes.length;i++){
```

```
20            total=total+enlaces[i];
21
22         }
23
24        Edge[] edges= new Edge[total];
25
26        int z=0;
27
28
29        for(int i = 0; i < (nodes.length + nodes_Switch.length); i++){
30            for(int j = 0; j < enlaces[i]; j++){
31                    edges[z]= new Edge(vertices[i], vertices[j],1);
32                    z++;
33            }
34        }
35
36
37
38        for(Edge e: edges){
39           graph.addEdge(e.getOne(), e.getTwo(), e.getWeight());
40        }
41
42        Dijkstra dijkstra = new Dijkstra(graph, vertices[srcNode-1].
              getLabel());
43
44        String dst = getHost2(dstNode,nodes);
45        List<Vertex> lista = dijkstra.getPathTo(dst);
46
47
48        Vertex aux;
49
50        int [] numbers = new int[lista.size()];
51
52        for(int i = 0; i < lista.size(); i++) {
53
54            aux = lista.get(i);
55
56            path[i] = aux.toString();
57            numbers[i] = Integer.parseInt(path[i].replaceAll("[\\D]", ""
                  ));
58
59        }
60
61        int x = 0;
62        String[] ports= new String[numbers.length];
63
64        for(int i = 0; i < ( numbers.length-1); i++) {
65
66
67                  ports[i] = getPort(numbers[i],numbers[i+1]);
68
69                  x = x +1;
70
71          }
72        ports[x] = getHost(dstNode-1);
73
74          return ports;
75      }
```

Método getSwitches2(nodes_switches)

Este método funciona de manera similar a `whoIs`, solo que en este caso se comprueban los dispositivos que son *switches*. También se adapta incluyendo un bucle que recorre cada *switch*.

```
1         private int[] getSwitches2(String[] nodes_switches) {
2
3                 int [] switches = new int[nodes_switches.length];
4
5                 for (int i = 0; i< nodes_switches.length; i++) {
6
7                 String path = nodes_switches[i].substring(0,1);
8                 switches[i]= Integer.parseInt(path);
9
10                }
11
12                return switches;
13        }
```

Método gethost2(Node,nodes)

Este método se utiliza en `whoIs` para obtener el identificador de un *host*. También en este método hay que incluir un bucle.

```
1  public static String getHost2(int Node, String[] nodes) {
2
3         String client = "";
4
5         for (int i= 0; i < nodes.length; i++){
6
7             if(Node==i){
8              client= ""+i;
9             }
10        }
11
12
13        return client;
14
15    }
```

Una vez identificados y explicados todos los detalles de la implementación, en el siguiente capítulo se procede a realizar diferentes medidas en ambas tecnologías con el fin de comprobar las ventajas del comportamiento de SDN.

Capítulo 6

Evaluación

Una vez explicado el diseño propuesto para las diferentes tecnologías a evaluar, y explicado el funcionamiento del protocolo PIM-SSM, en este capítulo se realizan diferentes evaluaciones.

La estructura del capítulo está formada por cuatro apartados, en el primero se estudia todo el tráfico generado en los diferentes escenarios para tener una concepción general de su funcionamiento; en los siguientes apartados se estudian la carga, la robustez y la escalabilidad para determinar las ventajas de SDN frente a las tecnologías actuales en el ámbito *multicast*.

6.1. Tráfico total generado

Antes de empezar es necesario comprobar que la velocidad de transmisión de los enlaces es la misma en todas las topologías, para ello se utiliza el comando de Mininet `iperfudp` que da como resultado el ancho de banda disponible para paquetes UDP en los enlaces. Una vez comprobado que en todas las topologías el ancho de banda UDP es de 10 Mbit/s se procede a realizar las diferentes evaluaciones.

Primero, se va a utilizar una topología 5.1 para calcular el tráfico total generado durante diez minutos. Para ello se ejecuta el archivo Python correspondiente, en el caso de Quagga *topo-router.py* y en el caso de SDN *topo-sdn.py*, gracias al comando `sudo python "nombre del archivo"`.

Para reproducir el entorno el *host 1* se va a utilizar como servidor, gracias al comando `ssmpingd` y se van a utilizar como clientes el *host 2* y el *host 3*, gracias al comando `ssmping`. Como el *host 1* tiene la dirección IP 10.0.0.1, en los clientes el comando para enviar la petición de unirse al grupo *multicast* se haría mediante el comando `ssmping 10.0.0.1`.

Antes de ejecutar el escenario es necesario abrir un terminal del *router 1*

y ejecutar Wireshark. En este caso, todos los paquetes de la red van a pasar por dicho *router* por lo que se puede capturar todo el tráfico de la red desde dicho router.

Una vez recogidos los paquetes mediante Wireshark, se separan los tipos de tráficos según se ha visto en 4.1.3 y 4.1.4; y se hace una gráfica con los datos recogidos. El resultado obtenido se muestra en la Figura (6.1).

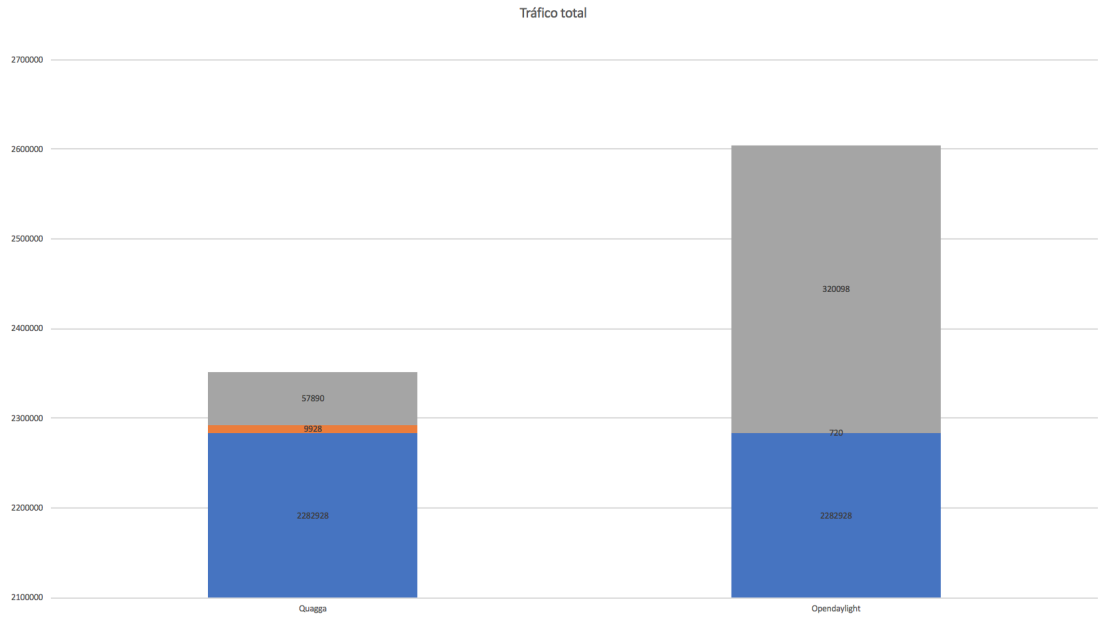

Figura 6.1: Gráfica.

Se representa con el color azul el tráfico UDP, con naranja el tráfico correspondiente al protocolo PIM-SSM y con gris el tráfico característico de cada tipo de red.

Los datos recogidos pertenecen a una simulación de diez minutos y se puede apreciar que para diez minutos el tráfico de señalización del protocolo PIM-SSM para el caso de OpenDayLight es mucho menor, en el siguiente apartado del capítulo se comentan con más detalle los resultados obtenidos. El objetivo de la representación de estos datos es familiarizarse con las características de ambas tecnologías.

Los 57890 bytes generados en Quagga son los que producen los *demonios* de los protocolos OSPF, zebra y pimd. Mientras que el protocolo OpenFlow ha generado 320098 bytes, de los cuales se han descartado los paquetes FLOW_MOD ya que dichos paquetes se han considerado de señalización del protocolo PIM-SSM.

Como se puede apreciar el tráfico generado por OpenFlow es mucho ma-

yor que en el caso de Quagga. La explicación se debe a que OpenFlow genera una gran cantidad de paquetes, como se comentó en el apartado 4, que se encargan del mantenimiento de la red completa. Por lo que la comparación no es adecuada, ya que Quagga es una red con solo tres protocolos implementados mientras que SDN incluye una gran cantidad de funcionalidades gracias a OpenFlow, que en el caso de las redes actuales -o aproximación clásica- hay que implementar mediante nuevos protocolos.

De forma aproximada se puede afirmar que SDN al incluir una cantidad de mensajes genéricos que controlan el estado de la red, permite el desarrollo de protocolos con menor carga en bytes al no tener dichos protocolos la obligación de controlar el estado de la red. Por lo que una de las ventajas de SDN se debe gracias al protocolo OpenFlow que ya tiene una serie de mensajes que facilitan el desarrollo de nuevos protocolos más sencillos y con menos carga.

Si se incrementase el número de protocolos en ambos casos, al final Quagga acabaría superando en bytes a SDN debido a que con tan solo tres protocolos ya llega a 50.000 bytes.

6.2. Tráfico de señalización del protocolo PIM-SSM

En el apartado anterior para una muestra de diez minutos se ha podido comprobar que el tráfico de señalización del protocolo PIM-SSM es mayor en el caso de Quagga. En este apartado se van a realizar un par de capturas para poder verificar que el tráfico de señalización es mayor siempre en Quagga.

Para este caso se van a realizar capturas hasta los diez minutos teniendo dos clientes y un servidor de la misma manera que se describió en el apartado anterior. En la imagen (6.2) se puede comprobar el funcionamiento del protocolo *multicast* para el caso de Quagga; para SDN el resultado sería exactamente igual por lo que no se va a mostrar ninguna imagen.

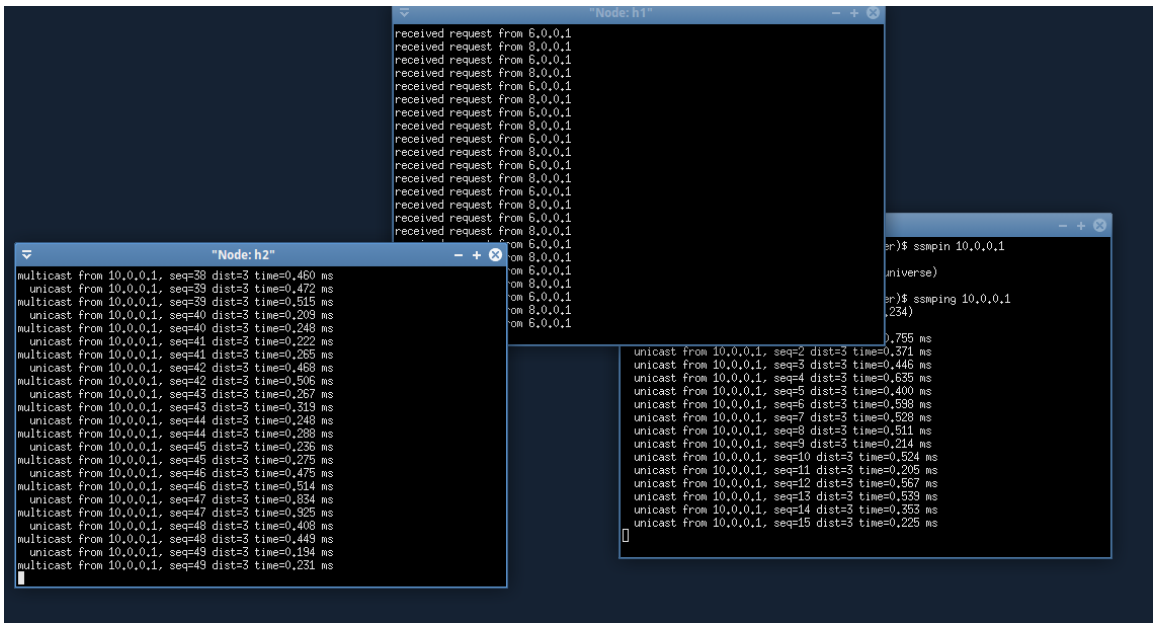

Figura 6.2: Gráfica.

Al estudiar las capturas Wireshark, se puede observar el funcionamiento de los protocolos. Para el caso de Quagga cuando se activa el servidor con ayuda del comando `ssmpingd` se puede observar cómo los *demonios* se activan mediante la transmisión de paquetes TCP por toda la red 6.3.

11	2.713147000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
12	2.713228000	127.0.0.1	127.0.0.1	TCP	68 2600→39523 [ACK] Seq=1 Ack=11 Win=86 Len=0 TSval=3529424 TSecr=3529424
13	2.737902000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply
14	2.738020000	127.0.0.1	127.0.0.1	TCP	68 39523→2600 [ACK] Seq=11 Ack=22 Win=86 Len=0 TSval=3529430 TSecr=3529430
15	2.738117000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
16	2.738224000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply
17	2.738503000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
18	2.738631000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply
19	2.740173000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
20	2.740303000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply
21	2.740489000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
22	2.740605000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply
23	2.740730000	127.0.0.1	127.0.0.1	ZEBRA	78 ZEBRA Request
24	2.740844000	127.0.0.1	127.0.0.1	ZEBRA	89 ZEBRA Reply

Figura 6.3: Paquetes Zebra.

En el caso de SDN, cuando se utiliza `ssmpingd` no se transmite ningún mensaje ya que es el controlador el que procesa esta información. Los paquetes de OpenFlow se transmiten de forma periódica para comprobar el estado de la red. En la Figura (6.4) se puede ver un ejemplo.

```
   4 0.004328000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REQUEST, OFPMP_GROUP_DESC
   5 0.004962000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REPLY, OFPMP_GROUP_DESC
   6 0.006905000 127.0.0.1       127.0.0.1       OpenFlow     92 Type: OFPT_MULTIPART_REQUEST, OFPMP_GROUP
   7 0.007292000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REPLY, OFPMP_GROUP
   8 0.008883000 127.0.0.1       127.0.0.1       OpenFlow     92 Type: OFPT_MULTIPART_REQUEST, OFPMP_PORT_STATS
   9 0.009700000 127.0.0.1       127.0.0.1       OpenFlow    532 Type: OFPT_MULTIPART_REPLY, OFPMP_PORT_STATS
  10 0.016210000 127.0.0.1       127.0.0.1       OpenFlow     92 Type: OFPT_MULTIPART_REQUEST, OFPMP_QUEUE
  11 0.016714000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REPLY, OFPMP_QUEUE
  12 0.019837000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REQUEST, OFPMP_TABLE
  13 0.020376000 127.0.0.1       127.0.0.1       OpenFlow   6180 Type: OFPT_MULTIPART_REPLY, OFPMP_TABLE
  14 0.039720000 127.0.0.1       127.0.0.1       OpenFlow     92 Type: OFPT_MULTIPART_REQUEST, OFPMP_METER_CONFIG
  15 0.040253000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REPLY, OFPMP_METER_CONFIG
  16 0.046175000 127.0.0.1       127.0.0.1       OpenFlow     92 Type: OFPT_MULTIPART_REQUEST, OFPMP_METER
  17 0.046622000 127.0.0.1       127.0.0.1       OpenFlow     84 Type: OFPT_MULTIPART_REPLY, OFPMP_METER
  18 0.048332000 127.0.0.1       127.0.0.1       OpenFlow    124 Type: OFPT_MULTIPART_REQUEST, OFPMP_FLOW
```

Figura 6.4: Paquetes OpenFlow.

Gracias a la opción de Wireshark `conversation` se pueden cuantificar el número de paquetes generados de un tipo. En `conversation` aparecen los paquetes agrupados por protocolo de transmisión y dirección IP destino y origen. Para Quagga tan solo hay que fijarse en los paquetes tipo *hello* y *join prune* mientras que para SDN hay que separar del protocolo OpenFlow los paquetes tipo *FLOW_MOD*. Una vez se han separado los tipos de datos, se crea con ayuda de Excel una gráfica con el número de bytes generados en cada caso como se puede ver en la Figura (6.5).

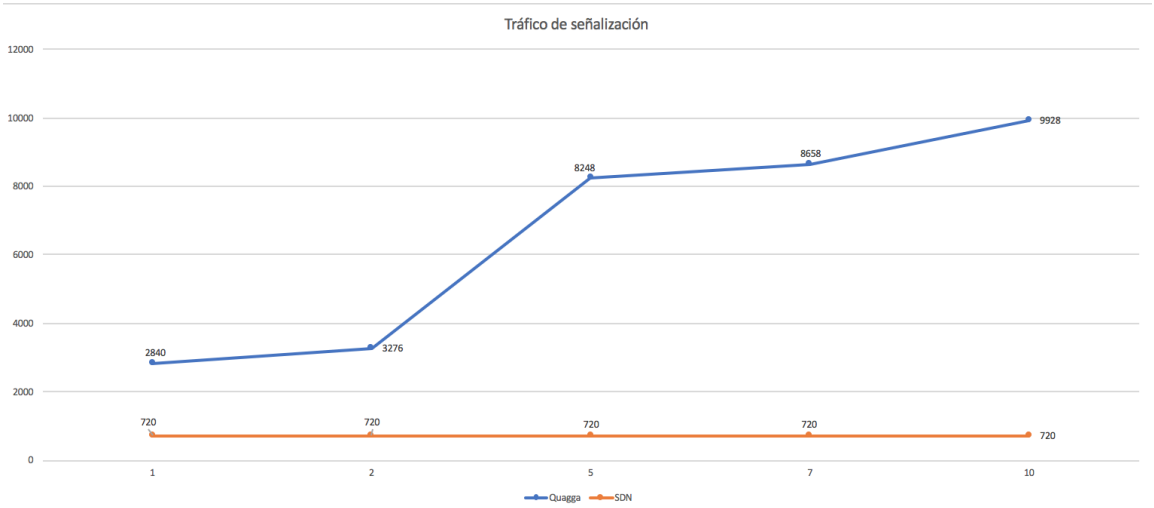

Figura 6.5: Tráfico de señalización.

En la gráfica se puede apreciar como el tráfico de señalización generado en Quagga es mucho mayor, esto se debe a que en el caso de SDN cuando un cliente quiere unirse a un grupo *multicast*, el controlador se encarga de recoger sus datos unirlo a dicho grupo y demás; y posteriormente en la red solo se envían paquetes *FLOW_MOD* para que los *switches* sepan como encaminar correctamente. Mientras que en el caso de las redes actuales, la mayoría de

los protocolos como, por ejemplo, PIM-SSM lo que hacen es transmitir mensajes *JOIN/PRUNE* desde el cliente al servidor para que el servidor sepa que un *host* quiere unirse a un grupo *multicast* y posteriormente el servidor también envía mensajes *JOIN/PRUNE* constantemente para confirmar que los clientes siguen suscritos al grupo *multicast*. Esto ya hace que el tráfico se incremente, además también se envían mensajes HELLO de forma periódica para comprobar el estado de los enlaces lo que incrementa aún más el tráfico de señalización.

En este caso como solo se han unido dos clientes y no se han llegado a quitar del grupo multicast en el caso de SDN en la gráfica aparecen los bytes correspondientes a los mensajes *FLOW_MOD*; los cuales, una vez enviados, ya no se vuelven a repetir hasta que no sean necesarios. Por eso, a lo largo del tiempo en el caso de SDN no aumenta el número de bytes en la red. Mientras que en el caso de Quagga constantemente se están enviado mensajes *HELLO* y *JOIN/PRUNE* por lo que la carga es mucho mayor.

Esta ventaja que presenta SDN es una de las más importantes en cuanto a innovación ya que permitiría a todos los servicios de *streaming* generar mucha menos cantidad de tráfico. Además de ser el ahorro de ancho de banda en las redes una de las características más importantes de SDN gracias a los controladores.

6.3. Robustez del protocolo PIM-SSM

Para evaluar la robustez se utiliza la topología en forma de anillo. Lo que se hace es utilizar el *host 1* como servidor y el *host 2* como cliente, una vez está funcionando el escenario y se está capturando con Wireshark el tráfico de paquetes se corta el enlace por el que se están transmitiendo los paquetes UDP. Gracias a Wireshark se puede observar el último paquete enviado correctamente antes del corte y en que momento empiezan a enviarse otra vez los paquetes.

En la Figura (6.6) se puede ver cómo se han transmitido 24 paquetes pero solo se han recibido 23, solo se ha perdido un paquete porque el protocolo OSPF en cuanto detecta que no se ha transmitido bien un paquete deja de enviar más paquetes UDP. No se ha capturado el momento preciso en el que se corta el enlace ya que sucede bastante rápido y no se puede apreciar en el terminal de h1.

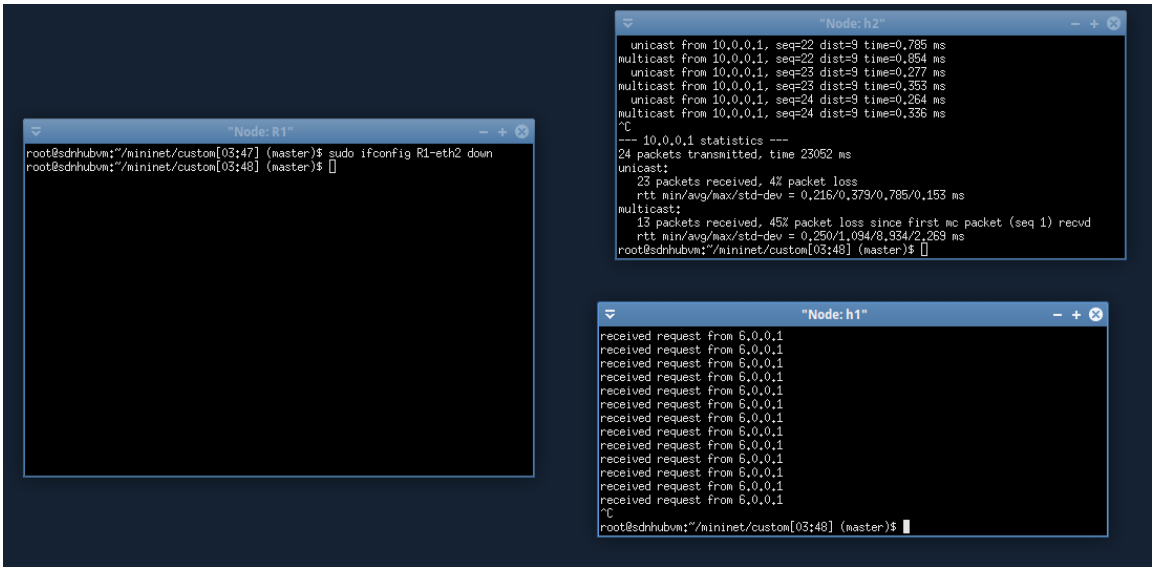

Figura 6.6: Ejemplo.

Desde Wireshark también se puede detectar el momento en el que falla el enlace y se tiene que volver a encaminar la red, en la Figura (6.8) se puede ver el paquete ICMP que informa de que no se ha enviado correctamente un paquete UDP, a continuación se pone en funcionamiento OSPF.

98	10.42917700(	127.0.0.1	127.0.0.1	ZEBRA	96 ZEBRA Reply
99	10.42921600(	127.0.0.1	127.0.0.1	TCP	68 39522→2600 [ACK] Seq=1 Ack=230 Win=86 Len=0 TSval=3531353 TSecr=3531353
100	10.42932500(	127.0.0.1	127.0.0.1	ZEBRA	96 ZEBRA Reply
101	10.42934700(	127.0.0.1	127.0.0.1	TCP	68 39522→2600 [ACK] Seq=1 Ack=258 Win=86 Len=0 TSval=3531353 TSecr=3531353
102	10.42940400(	127.0.0.1	127.0.0.1	ZEBRA	96 ZEBRA Reply
103	10.42942100(	127.0.0.1	127.0.0.1	TCP	68 39522→2600 [ACK] Seq=1 Ack=286 Win=86 Len=0 TSval=3531353 TSecr=3531353
104	10.42952800(	127.0.0.1	127.0.0.1	ZEBRA	96 ZEBRA Reply
105	10.42954700(	127.0.0.1	127.0.0.1	TCP	68 39522→2600 [ACK] Seq=1 Ack=314 Win=86 Len=0 TSval=3531353 TSecr=3531353
106	10.42965600(	127.0.0.1	127.0.0.1	ZEBRA	96 ZEBRA Reply
107	10.42968200(	127.0.0.1	127.0.0.1	TCP	68 39522→2600 [ACK] Seq=1 Ack=342 Win=86 Len=0 TSval=3531353 TSecr=3531353
108	10.62167500(	20.0.1.1	224.0.0.5	OSPF	112 LS Update
109	10.67177600(	6.0.0.1	10.0.0.1	UDP	82 Source port: 51680 Destination port: 4321
110	10.67181600(	6.0.0.2	6.0.0.1	ICMP	110 Destination unreachable (Network unreachable)
111	10.82279800(	127.0.0.1	127.0.0.1	ZEBRA	87 ZEBRA Request
112	10.82287500(	127.0.0.1	127.0.0.1	TCP	68 2600→39524 [ACK] Seq=101 Ack=20 Win=86 Len=0 TSval=3531451 TSecr=3531451
113	10.82637900(	127.0.0.1	127.0.0.1	ZEBRA	87 ZEBRA Request
114	10.82640200(	127.0.0.1	127.0.0.1	TCP	68 2600→39524 [ACK] Seq=101 Ack=39 Win=86 Len=0 TSval=3531452 TSecr=3531452
115	10.82644900(	127.0.0.1	127.0.0.1	ZEBRA	87 ZEBRA Request

Figura 6.7: Captura de Wireshark de OSPF.

En el caso de SDN cuando se corta el enlace al momento se envía otro paquete *FLOW_MOD*, para medir el tiempo se toma como referencia el último paquete UDP enviado correctamente antes de recibir el paquete ICMP de *Network unreachable*.

En el caso de Quagga cuando OSPF se ha actualizado envía un mensaje *LS Acknowledge*.

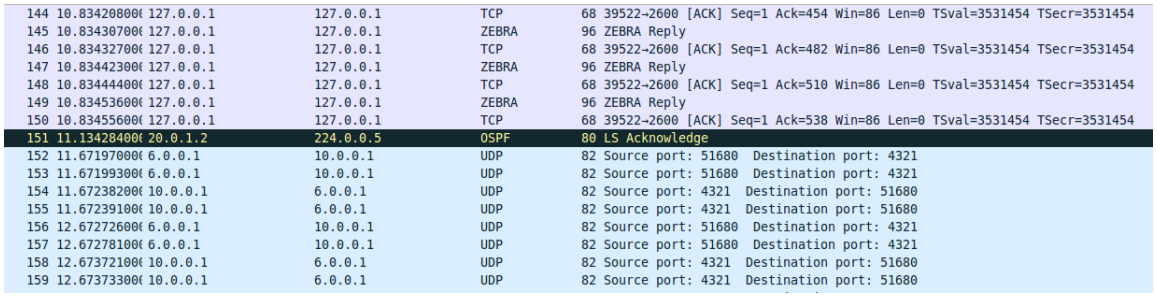

Figura 6.8: Captura de Wireshark mensaje de actualización OSPF.

Una vez recogidas las diferentes capturas de Wireshark se miden los diferentes tiempos. Se han realizado cinco medidas para cada escenario (con y sin SDN) y se ha observado que los tiempos obtenidos aunque nunca salen iguales su aproximación en una misma red es del orden de 0,00000001 por lo que los datos mostrados corresponden sólo a la última prueba realizada:

Quagga	SDN
1,000154	0,9976

Tabla 6.1: Tiempos de robustez

Se observa que SDN -para la topología considerada- tarda 10 milisegundos menos que en el caso de Quagga en recuperarse. Hay que tener en cuenta que la topología en forma de anillo está formada por tan solo 9 *routers* o *switches* en el caso de SDN y que este mismo ejemplo trasladado a una red real conllevaría una diferencia de tiempos de recuperación mucho más apreciable.

Pese a ser una topología pequeña se puede deducir que SDN presenta una mayor robustez en cuanto fallos en la red, debido a que es más rápido que el controlador procese el protocolo OSPF sin tener que enviar mensajes a que entre todos los routers se intercambien mensajes del protocolo OSPF. Además en redes actuales con mayores tiempos de transmisión, la diferencia sería mucho mayor y SDN sería bastante más rápido en cuanto a detección y recuperación de errores.

Para terminar este apartado, cabe decir que también se genera más tráfico cuando se produce un error en la red simulada con Quagga que en la red SDN por lo dicho anteriormente. Por ejemplo, en la siguiente Figura (6.9) se pueden ver paquetes del protocolo OSPF en el caso de Quagga mientras que en OpenDayLight no aparece ningún tipo de mensajes del protocolo OSPF.

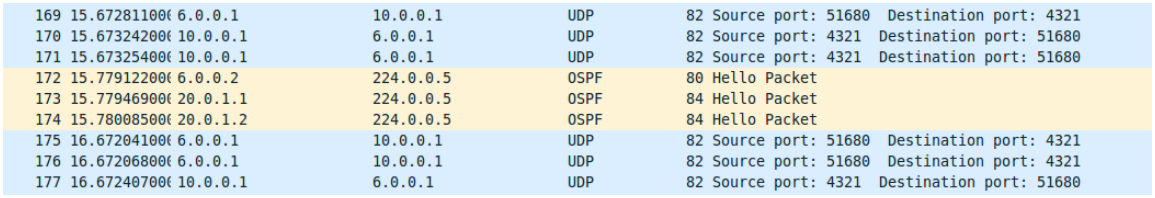

Figura 6.9: Paquetes OSPF.

6.4. Escalabilidad del protocolo PIM-SSM

Además de las ventajas que presenta SDN en cuanto a carga por paquetes de señalización y robustez también se va a comprobar que al aumentar el número de clientes *multicast* las redes con SDN son más escalables y permiten dar servicios a varios clientes sin aumentar en exceso la carga del tráfico en comparación con una aproximación sin SDN.

Para comprobar la escalabilidad de los servicios *multicast* en redes SDN lo que se ha hecho es realizar diferentes medidas en ambas redes con diferentes números de clientes. Para que se genere el suficiente tráfico, cada medida se ha realizado durante cinco minutos, de manera que los datos recogidos sean fiables.

Para realizar estas medidas se ha utilizado la topología en forma de árbol con 17 *hosts* y se ha empezado con dos clientes hasta llegar a los ocho clientes. Los resultados obtenidos se han recogido en la Fifura (6.10).

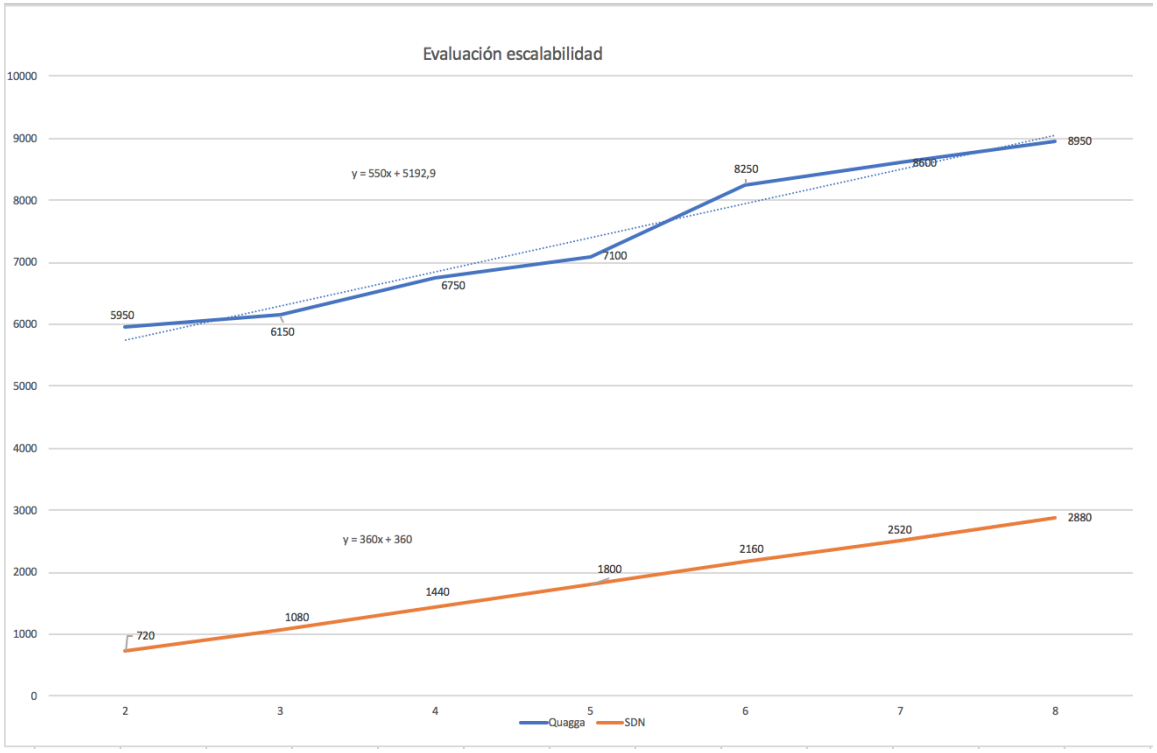

Figura 6.10: Gráfico para comparar la escalabilidad.

La gráfica anterior presenta un tráfico mayor para Quagga desde que se utilizan dos clientes. Este hecho no tiene mayor relevancia ya que al ser las medidas de cinco minutos ya se ha demostrado anteriormente que el tráfico de señalización con el tiempo es mayor en el caso de Quagga. Para comprobar la escalabilidad en lo que hay que fijarse es en la pendiente de las rectas que se forman, con la ayuda de excel se ha calculado la pendiente de ambas rectas y como se puede ver en la gráfica 6.10 la pendiente de la red de Quagga es de 550 mientras que la pendiente de SDN es de 360.

Por lo que, cuanto más aumenta el número de clientes *multicast* en una red, menor es el tráfico que se genera en el caso de SDN comparado con las redes actuales.

6.5. Conclusión de la evaluación

Una vez explicados los resultados obtenidos, es sencillo llegar a la conclusión de que la tecnología SDN supone un avance muy importante para los servicios multicast. Hoy en día las compañías que ofrecen este tipo de servicios tienen la problemática de tener que enfrentarse a demandas cada

vez más exigentes en cuanto a calidad de servicio. Lo que desemboca en un aumento del tráfico prácticamente insostenible, además de que un corte en la red puede suponer para los clientes que disfrutan de este tipo de servicios un motivo de queja para la empresa.

Por ello, el desarrollo de las nuevas tecnologías de redes como SDN son fundamentales para una gran variedad de aspectos. En este proyecto se ha podido comprobar que en topologías muy comunes en Internet el uso de SDN supone una carga de tráfico mucho menor que en las redes actuales. Las comprobaciones llevadas a cabo durante este proyecto no simular redes reales ya que la simulación de tantos dispositivos es prácticamente inviable. Pero, si para redes tan pequeñas las ventajas en cuanto a carga de tráfico son tan apreciables, cuanto mayor es la red menor será el tráfico generado en el caso de SDN.

Con la robustez y escalabilidad sucede lo mismo, cuanto mayores sean las redes en el caso de la tecnología actual mayores serán los tiempos de transmisión y más tardará una conexión multicast en restablecerse tras un fallo, mientras que en el caso de SDN el controlador ejecutará todos los cálculos necesarios y enviará un par de paquetes para realizar el encaminamiento necesario, en cuanto a escalabilidad ya se ha podido comprobar que cuantos más clientes se unen a un grupo multicast; en el caso de SDN menor es el tráfico generado en comparación con las redes actuales. Por lo que SDN supone un avance muy importante para los servicios que necesitan transmisiones *multicast*.

Capítulo 7

Conclusión y vías futuras

En este último capítulo de la memoria se indican una conclusión global desarrollada durante la realización de todo el proyecto, se muestran los diferentes obstáculos que han surgido durante el desarrollo del proyecto y se indican las posibles vías futuras de estudio partiendo del siguiente proyecto.

7.1. Conclusiones

En este proyecto, se ha partido desde la configuración realizada por Carlos Santamaría el año pasado en su TFG. Una vez estudiada la máquina virtual con todas sus configuraciones, se ha procedido a una evaluación más exhaustiva del protocolo PIM-SSM. Se han comparado las nuevas tecnologías SDN con ayuda del controlador OpenDaylight, la herramienta Mininet y el programa Eclipse para la configuración del controlador.

Antes de realizar todas las configuraciones se ha llevado a cabo un estudio sobre el funcionamiento de los protocolos *Multicast*, tecnologías de 5G como SDN y NFV, en particular se ha estudiado la teoría del protocolo PIM-SSM y el funcionamiento de todas las herramientas necesarias para el desarrollo del proyecto.

Una vez implementados todos los cambios necesarios para poder realizar una evaluación más completa se han realizado una serie de medidas mediante la herramienta Wireshark para recoger todos los datos que se han mostrado en el capítulo anterior. Finalmente se ha llegado a la conclusión de que SDN presenta una gran variedad de ventajas con respecto a las redes actuales.

Las principales contribuciones de este proyecto son:

- Se ha recopilado información sobre las redes SDN, identificando sus ventajas y desventajas en el ámbito del *networking* actual.

- Se ha detallado la configuración de topologías con ayuda de la herramienta *mininet*, lo que proporciona ayuda familiarizarse con esta herramienta así como con todas las que se han utilizado durante el proyecto.
- Se ha explicado el funcionamiento de Quagga y también se han dado unas nociones básicas sobre el funcionamiento de los correspondientes *demonios*. Lo cual ayuda a comprender mejor el funcionamiento de los procesos ocultos tan característicos en cualquier ordenador.

7.2. Problemas que han surgido durante el desarrollo del proyecto

Durante la realización del proyecto han surgido distintos problemas que han ido ralentizando el avance del mismo. A continuación, se exponen algunos de ellos:

- La limitación más importante ha sido el tiempo que se ha empleado en restaurar el estado original de la máquina virtual de Carlos. Ya que dicha máquina venía con configuraciones hechas posteriormente al trabajo de Carlos y al no estar familiarizado con las diferentes herramientas ha supuesto una cantidad enorme de tiempo volver a la configuración correcta. Aún habiéndose perdido tiempo debido a lo dicho anteriormente también ha servido para familiarizarse de forma temprana con todas las herramientas.
- Otra limitación ha sido el tiempo de compilación del controlador, que tarda aproximadamente unos 15 minutos cada vez que se quiere realizar cualquier cambio. Además de que a veces deja de funcionar por cualquier motivo y hay que volver a compilarlo todo. Pese al tiempo perdido por este motivo se ha tenido la ventaja de trabajar con máquina virtual, la cual ha permitido una vez el controlador funcionaba correctamente guardar el estado de la máquina y no tener que volver a compilar.
- Otro problema ha sido saber identificar desde qué dispositivo de cada topología capturar todo el tráfico ya que activar Wireshark en todos los dispositivos de cada topología era inviable. Al final, se ha solucionado proponiendo topologías que sirviesen para evaluar correctamente los diferentes aspectos y que tuvieran un *router* por donde fuese imprescindible que circulase todo el tráfico de la red.

7.3. Trabajos futuros

Pese a que este proyecto es una continuación de uno desarrollado anteriormente, aún quedan muchas líneas futuras de trabajo ya que el 5G lleva poco tiempo desarrollándose y aún queda mucho por descubrir. Algunas trabajos futuros partiendo de este proyecto podrían ser:

- En este proyecto se han simulado topologías pequeñas ya que el material utilizado no tiene la suficiente potencia para simular redes tan grandes como las que se utilizan en la realidad. Una posible vía futura podría ser intentar simular topologías más grandes que las utilizadas en este proyecto.
- En este proyecto se ha utilizado el algoritmo de Dijkstra para el encaminamiento de los paquetes. Otra posible vía futura podría ser desarrollar un algoritmo de encaminamiento diferente que se adecuara a las comodidades que ofrece el controlador OpenDaylight y que utilizará los mensajes del protocolo OpenFlow de forma complementaria para generar aún menos tráfico.
- Como se ha dicho en el primer punto en este proyecto se han realizado simulaciones de los diferentes entornos. Otra línea de trabajo podría ser utilizar dispositivos reales, aunque solo fuese el controlador se podrían aportar nuevos resultados que diesen más información sobre las ventajas del funcionamiento de estas nuevas tecnologías.

7.4. Valoración personal

Para terminar con el proyecto me gustaría acabar exponiendo una valoración personal de lo que ha supuesto para mi la realización de este proyecto y todo el camino que he tenido que seguir para poder llegar hasta aquí.

Durante estos cuatro años de carrera he ido aprendiendo una infinidad de valores positivos entre los que destacan el esfuerzo y la constancia. El paso final para terminar este grado ha sido la realización de un proyecto donde he tenido que poner en práctica todo lo aprendido durante el grado y para mi ha sido una gran satisfacción poder comprobar que pese a las dificultades que puede presentar afrontar un tema tan innovador y con tan poco información ilustrada actualmente. Se ha podido finalizar gracias a todo el tiempo empleado y a la ayuda que he recibido de mis tutores.

Otro punto importante del TFG ha sido familiarizarme con lo que me puede esperar en el mundo laboral. Ya que pese a haber recibido ayuda; casi todo el proyecto ha tenido que realizarse a base de buscar soluciones, resolver fallos, paciencia y un montón de desafíos que he tenido que afrontar.

Finalmente, queda destacar que la satisfacción personal alcanzada tras la realización del proyecto ha compensando todas las horas de trabajo y frustraciones.

Apéndice A

Topologías en Mininet

A.1. Topología Anillo sin controlador

```
#!/usr/bin/env python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.log import lg, info, setLogLevel
from mininet.util import dumpNodeConnections, quietRun, moveIntf
from mininet.cli import CLI
from mininet.node import Switch, OVSKernelSwitch

from subprocess import Popen, PIPE, check_output
from time import sleep, time
from multiprocessing import Process
from argparse import ArgumentParser

import sys
import os
import termcolor as T
import time

setLogLevel('info')

parser = ArgumentParser("Configure a network composed of routers in
    Mininet.")
parser.add_argument('--sleep', default=3, type=int)
args = parser.parse_args()

def log(s, col="green"):
    print T.colored(s, col)


class Router(Switch):
    """Defines a new router that is inside a network namespace so that
        the
    individual routing entries don't collide.   """
    ID = 0
    def __init__(self, name, **kwargs):
        kwargs['inNamespace'] = True
```

```
36          Switch.__init__(self, name, **kwargs)
37          Router.ID += 1
38          self.switch_id = Router.ID
39
40      @staticmethod
41      def setup():
42          return
43
44      def start(self, controllers):
45          pass
46
47      def stop(self):
48          self.deleteIntfs()
49
50      def log(self, s, col="magenta"):
51          print T.colored(s, col)
52
53
54  class SimpleTopo(Topo):
55      """topologia en anillo
56
57      """
58
59
60      def __init__(self):
61
62          # Add default members to class
63          super(SimpleTopo, self ).__init__()
64          num_host=2
65          num_routers = 9
66          routers = []
67          for i in xrange(num_routers):
68              router = self.addSwitch('R%d' % (i+1))
69              routers.append(router)
70              hosts = []
71
72          for i in xrange(num_host):
73              hostname = 'h%d' % (i+1)
74              host = self.addNode(hostname)
75              hosts.append(host)
76
77          self.addLink('R1','h1')
78          self.addLink('R1','R2')
79          self.addLink('R1','R9')
80          self.addLink('R2','h2')
81          self.addLink('R2','R3')
82          self.addLink('R3','R4')
83          self.addLink('R4','R5')
84          self.addLink('R5','R6')
85          self.addLink('R6','R7')
86          self.addLink('R7','R8')
87          self.addLink('R8','R9')
88
89
90
91          return
92
93  def main():
94
95      os.system("rm -f /tmp/R*.log /tmp/R*.pid logs/*")
96      os.system("mn -c >/dev/null 2>&1")
97      os.system("killall -9 zebra pimd ospfd > /dev/null 2>&1")
```

```
98
99      net = Mininet(topo=SimpleTopo(), switch=Router, controller=None)
100     net.start()
101
102     for router in net.switches:
103         router.cmd("sysctl -w net.ipv4.ip_forward=1")
104         router.waitOutput()
105
106     log("Waiting %d seconds for sysctl changes to take effect..."
107     % args.sleep)
108     sleep(args.sleep)
109
110     for router in net.switches:
111         router.cmd(" /usr/local/quagga/sbin/zebra -f /usr/local/quagga
                /etc/red2_zebra--%s.conf -d -i /usr/local/quagga/etc/zebra
                --%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router
                .name, router.name))
112         router.waitOutput()
113         router.cmd("/usr/local/quagga/sbin/pimd -f /usr/local/quagga/
                etc/red2_pimd--%s.conf  -d -i /usr/local/quagga/etc/pimd
                --%s.pid > log/%s--pimd-stdout 2>&1" %(router.name, router
                .name,router.name), shell=True)
114         router.waitOutput()
115         router.cmd(" /usr/local/quagga/sbin/ospfd -f /usr/local/quagga
                /etc/red2_ospf--%s.conf -d -i /usr/local/quagga/etc/ospf
                --%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router
                .name, router.name))
116         router.waitOutput()
117         log("Starting zebra ospf and pimd on %s" % router.name)
118
119     net.hosts[0].cmd("ifconfig h1-eth0 10.0.0.1")
120     net.hosts[1].cmd("ifconfig h2-eth0 6.0.0.1")
121     net.hosts[0].cmd("route add default gw  10.0.0.2")
122     net.hosts[1].cmd("route add default gw  6.0.0.2")
123
124
125     CLI(net)
126     net.stop()
127     os.system("killall -9 zebra pimd ospfd")
128
129 if __name__ == "__main__":
130     main()
```

A.2. Topología Anillo con controlador

```
#!/usr/bin/env python
"""

Copyright (C)       anillo OpenDayLight



"""

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.log import lg, info, setLogLevel
from mininet.util import dumpNodeConnections, quietRun, moveIntf
from mininet.cli import CLI
from mininet.node import Switch, OVSKernelSwitch, OVSSwitch
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import IVSSwitch
from mininet.link import TCLink, Intf



from subprocess import Popen, PIPE, check_output
from time import sleep, time
from multiprocessing import Process
from argparse import ArgumentParser

import sys
import os
import termcolor as T
import time

setLogLevel('info')

def myNetwork():


        net= Mininet(topo=None, listenPort=6633, build=False, ipBase='
            10.0.0.0/8',link=TCLink)
        "net= Mininet(topo=None, listenPort=6633, build=False, ipBase
            ='10.0.0.0/8',link=TCLink, autoStaticArp=True)"


        info('*** Adding controller\n')
        c0=net.addController(name='c0', controller=RemoteController,
            protocols='OpenFlow13', ip='127.0.0.1')

        s1=net.addSwitch('s1', protocols='OpenFlow13',mac='
            00:00:00:00:00:06')
        s2=net.addSwitch('s2', protocols='OpenFlow13',mac='
            00:00:00:00:00:07')
        s3=net.addSwitch('s3', protocols='OpenFlow13',mac='
            00:00:00:00:00:08')
        s4=net.addSwitch('s4', protocols='OpenFlow13',mac='
            00:00:00:00:00:09')
        s5=net.addSwitch('s5', protocols='OpenFlow13',mac='
            00:00:00:00:00:10')
```

```
50          s6=net.addSwitch('s6', protocols='OpenFlow13',mac='
                00:00:00:00:00:11')
51          s7=net.addSwitch('s7', protocols='OpenFlow13',mac='
                00:00:00:00:00:12')
52          s8=net.addSwitch('s8', protocols='OpenFlow13',mac='
                00:00:00:00:00:13')
53          s9=net.addSwitch('s9', protocols='OpenFlow13',mac='
                00:00:00:00:00:14')
54
55          info('***adding host\n')
56          h1=net.addHost('h1', cls=Host,mac='00:00:00:00:00:01')
57          h2=net.addHost('h2', cls=Host,mac='00:00:00:00:00:02')
58
59
60          info('***Creating links\n')
61
62          net.addLink('s1','h1')
63          net.addLink('s1','s2')
64          net.addLink('s1','s9')
65          net.addLink('s2','h2')
66          net.addLink('s2','s3')
67          net.addLink('s3','s4')
68          net.addLink('s4','s5')
69          net.addLink('s5','s6')
70          net.addLink('s6','s7')
71          net.addLink('s7','s8')
72          net.addLink('s8','s9')
73
74
75
76          info('***Starting network\n')
77          net.build()
78          net.start()
79
80
81          info('***Starting controller\n')
82          for controller in net.controllers:
83              controller.start()
84
85          info('***Starting switches\n')
86          net.get('s1').start([c0])
87          net.get('s2').start([c0])
88          net.get('s3').start([c0])
89          net.get('s4').start([c0])
90          net.get('s5').start([c0])
91          net.get('s6').start([c0])
92          net.get('s7').start([c0])
93          net.get('s8').start([c0])
94          net.get('s9').start([c0])
95
96
97          info('***Post configure switches and hosts\n')
98          CLI(net)
99          net.stop()
100
101 if __name__ == "__main__":
102         setLogLevel( 'info' )
103         myNetwork()
```

A.3. Topología árbol sin controlador

```
#!/usr/bin/env python

from mininet.topo import Topo
from mininet.net import Mininet
from mininet.log import lg, info, setLogLevel
from mininet.util import dumpNodeConnections, quietRun, moveIntf
from mininet.cli import CLI
from mininet.node import Switch, OVSKernelSwitch

from subprocess import Popen, PIPE, check_output
from time import sleep, time
from multiprocessing import Process
from argparse import ArgumentParser

import sys
import os
import termcolor as T
import time

setLogLevel('info')

parser = ArgumentParser("Configure a network composed of routers in
    Mininet.")
parser.add_argument('--sleep', default=3, type=int)
args = parser.parse_args()

def log(s, col="green"):
    print T.colored(s, col)


class Router(Switch):
    """Defines a new router that is inside a network namespace so that
         the
    individual routing entries don't collide.   """
    ID = 0
    def __init__(self, name, **kwargs):
        kwargs['inNamespace'] = True
        Switch.__init__(self, name, **kwargs)
        Router.ID += 1
        self.switch_id = Router.ID

    @staticmethod
    def setup():
        return

    def start(self, controllers):
        pass

    def stop(self):
        self.deleteIntfs()

    def log(self, s, col="magenta"):
        print T.colored(s, col)


class SimpleTopo(Topo):
    """host: 17
       routers: 13
```

```
57
58      """
59
60
61      def __init__(self):
62
63          # Add default members to class
64          super(SimpleTopo, self ).__init__()
65          num_host=17
66          num_routers = 13
67          routers = []
68          for i in xrange(num_routers):
69              router = self.addSwitch('R%d' % (i+1))
70              routers.append(router)
71              hosts = []
72
73          for i in xrange(num_host):
74              hostname = 'h%d' % (i+1)
75              host = self.addNode(hostname)
76              hosts.append(host)
77
78          self.addLink('R1','h1')
79          self.addLink('R1','R2')
80          self.addLink('R1','R3')
81          self.addLink('R1','R4')
82          self.addLink('R1','R5')
83          self.addLink('R2','R10')
84          self.addLink('R2','R6')
85          self.addLink('R3','R11')
86          self.addLink('R3','R7')
87          self.addLink('R4','R12')
88          self.addLink('R4','R8')
89          self.addLink('R5','R9')
90          self.addLink('R5','R13')
91          self.addLink('R10','h2')
92          self.addLink('R10','h3')
93          self.addLink('R6','h4')
94          self.addLink('R6','h5')
95          self.addLink('R11','h6')
96          self.addLink('R11','h7')
97          self.addLink('R7','h8')
98          self.addLink('R7','h9')
99          self.addLink('R12','h10')
100         self.addLink('R12','h11')
101         self.addLink('R8','h12')
102         self.addLink('R8','h13')
103         self.addLink('R9','h14')
104         self.addLink('R9','h15')
105         self.addLink('R13','h16')
106         self.addLink('R13','h17')
107
108         return
109
110 def main():
111
112     os.system("rm -f /tmp/R*.log /tmp/R*.pid logs/*")
113     os.system("mn -c >/dev/null 2>&1")
114     os.system("killall -9 zebra pimd ospfd > /dev/null 2>&1")
115
116     net = Mininet(topo=SimpleTopo(), switch=Router, controller=None)
117     net.start()
118
```

```
119     for router in net.switches:
120         router.cmd("sysctl -w net.ipv4.ip_forward=1")
121         router.waitOutput()
122 
123     log("Waiting %d seconds for sysctl changes to take effect..."
124     % args.sleep)
125     sleep(args.sleep)
126 
127     for router in net.switches:
128         router.cmd(" /usr/local/quagga/sbin/zebra -f /usr/local/quagga
                /etc/red3_zebra--%s.conf -d -i /usr/local/quagga/etc/zebra
                --%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router
                .name, router.name))
129         router.waitOutput()
130         router.cmd("/usr/local/quagga/sbin/pimd -f /usr/local/quagga/
                etc/red3_pimd--%s.conf  -d -i /usr/local/quagga/etc/pimd
                --%s.pid > log/%s--pimd-stdout 2>&1" %(router.name, router
                .name,router.name), shell=True)
131         router.waitOutput()
132         router.cmd(" /usr/local/quagga/sbin/ospfd -f /usr/local/quagga
                /etc/red3_ospf--%s.conf -d -i /usr/local/quagga/etc/ospf
                --%s.pid > log/%s--zebra-stdout 2>&1" %(router.name,router
                .name, router.name))
133         router.waitOutput()
134         log("Starting zebra ospfd and pimd on %s" % router.name)
135 
136     net.hosts[0].cmd("ifconfig h1-eth0 10.0.0.1")
137     net.hosts[1].cmd("ifconfig h2-eth0 6.0.0.1")
138     net.hosts[2].cmd("ifconfig h3-eth0 6.0.0.2")
139     net.hosts[3].cmd("ifconfig h4-eth0 7.0.0.1")
140     net.hosts[4].cmd("ifconfig h5-eth0 7.0.0.2")
141     net.hosts[5].cmd("ifconfig h6-eth0 8.0.0.1")
142     net.hosts[6].cmd("ifconfig h7-eth0 8.0.0.2")
143     net.hosts[7].cmd("ifconfig h8-eth0 9.0.0.1")
144     net.hosts[8].cmd("ifconfig h9-eth0 9.0.0.2")
145     net.hosts[9].cmd("ifconfig h10-eth0 11.0.0.1")
146     net.hosts[10].cmd("ifconfig h11-eth0 11.0.0.2")
147     net.hosts[11].cmd("ifconfig h12-eth0 12.0.0.1")
148     net.hosts[12].cmd("ifconfig h13-eth0 12.0.0.2")
149     net.hosts[13].cmd("ifconfig h14-eth0 13.0.0.1")
150     net.hosts[14].cmd("ifconfig h15-eth0 13.0.0.2")
151     net.hosts[15].cmd("ifconfig h16-eth0 14.0.0.1")
152     net.hosts[16].cmd("ifconfig h17-eth0 14.0.0.2")
153 
154 
155     net.hosts[0].cmd("route add default gw  10.0.0.2")
156     net.hosts[1].cmd("route add default gw  6.0.0.3")
157     net.hosts[2].cmd("route add default gw  6.0.0.4")
158     net.hosts[3].cmd("route add default gw  7.0.0.3")
159     net.hosts[4].cmd("route add default gw  7.0.0.4")
160     net.hosts[5].cmd("route add default gw  8.0.0.3")
161     net.hosts[6].cmd("route add default gw  8.0.0.4")
162     net.hosts[7].cmd("route add default gw  9.0.0.3")
163     net.hosts[8].cmd("route add default gw  9.0.0.4")
164     net.hosts[9].cmd("route add default gw  11.0.0.3")
165     net.hosts[10].cmd("route add default gw  11.0.0.4")
166     net.hosts[11].cmd("route add default gw  12.0.0.3")
167     net.hosts[12].cmd("route add default gw  12.0.0.4")
168     net.hosts[13].cmd("route add default gw  13.0.0.3")
169     net.hosts[14].cmd("route add default gw  13.0.0.4")
170     net.hosts[15].cmd("route add default gw  14.0.0.3")
171     net.hosts[16].cmd("route add default gw  14.0.0.4")
```

```
172
173
174
175     CLI(net)
176     net.stop()
177     os.system("killall -9 zebra pimd ospfd")
178
179 if __name__ == "__main__":
180     main()
```

A.4. Topología árbol con controlador

```
1  #!/usr/bin/env python
2  """
3
4  Copyright (C)
5
6
7
8  """
9
10 from mininet.topo import Topo
11 from mininet.net import Mininet
12 from mininet.log import lg, info, setLogLevel
13 from mininet.util import dumpNodeConnections, quietRun, moveIntf
14 from mininet.cli import CLI
15 from mininet.node import Switch, OVSKernelSwitch, OVSSwitch
16 from mininet.node import Controller, RemoteController, OVSController
17 from mininet.node import CPULimitedHost, Host, Node
18 from mininet.node import IVSSwitch
19 from mininet.link import TCLink, Intf
20
21
22
23 from subprocess import Popen, PIPE, check_output
24 from time import sleep, time
25 from multiprocessing import Process
26 from argparse import ArgumentParser
27
28 import sys
29 import os
30 import termcolor as T
31 import time
32
33 setLogLevel('info')
34
35 def myNetwork():
36
37
38         net= Mininet(topo=None, listenPort=6633, build=False, ipBase='
              10.0.0.0/8',link=TCLink)
39         "net= Mininet(topo=None, listenPort=6633, build=False, ipBase
              ='10.0.0.0/8',link=TCLink, autoStaticArp=True)"
40
41
42         info('*** Adding controller\n')
```

```
43          c0=net.addController(name='c0', controller=RemoteController,
                protocols='OpenFlow13', ip='127.0.0.1')
44
45          s1=net.addSwitch('s1', protocols='OpenFlow13',mac='
                00:00:00:00:00:20')
46          s2=net.addSwitch('s2', protocols='OpenFlow13',mac='
                00:00:00:00:00:21')
47          s3=net.addSwitch('s3', protocols='OpenFlow13',mac='
                00:00:00:00:00:22')
48          s4=net.addSwitch('s4', protocols='OpenFlow13',mac='
                00:00:00:00:00:23')
49          s5=net.addSwitch('s5', protocols='OpenFlow13',mac='
                00:00:00:00:00:24')
50          s6=net.addSwitch('s6', protocols='OpenFlow13',mac='
                00:00:00:00:00:25')
51          s7=net.addSwitch('s7', protocols='OpenFlow13',mac='
                00:00:00:00:00:26')
52          s8=net.addSwitch('s8', protocols='OpenFlow13',mac='
                00:00:00:00:00:27')
53          s9=net.addSwitch('s9', protocols='OpenFlow13',mac='
                00:00:00:00:00:28')
54          s10=net.addSwitch('s10', protocols='OpenFlow13',mac='
                00:00:00:00:00:29')
55          s11=net.addSwitch('s11', protocols='OpenFlow13',mac='
                00:00:00:00:00:30')
56          s12=net.addSwitch('s12', protocols='OpenFlow13',mac='
                00:00:00:00:00:31')
57          s13=net.addSwitch('s13', protocols='OpenFlow13',mac='
                00:00:00:00:00:32')
58
59
60          info('***adding host\n')
61          h1=net.addHost('h1', cls=Host,mac='00:00:00:00:00:01')
62          h2=net.addHost('h2', cls=Host,mac='00:00:00:00:00:02')
63          h3=net.addHost('h3', cls=Host,mac='00:00:00:00:00:03')
64          h4=net.addHost('h4', cls=Host,mac='00:00:00:00:00:04')
65          h5=net.addHost('h5', cls=Host,mac='00:00:00:00:00:05')
66          h6=net.addHost('h6', cls=Host,mac='00:00:00:00:00:06')
67          h7=net.addHost('h7', cls=Host,mac='00:00:00:00:00:07')
68          h8=net.addHost('h8', cls=Host,mac='00:00:00:00:00:08')
69          h9=net.addHost('h9', cls=Host,mac='00:00:00:00:00:09')
70          h10=net.addHost('h10', cls=Host,mac='00:00:00:00:00:10')
71          h11=net.addHost('h11', cls=Host,mac='00:00:00:00:00:11')
72          h12=net.addHost('h12', cls=Host,mac='00:00:00:00:00:12')
73          h13=net.addHost('h13', cls=Host,mac='00:00:00:00:00:13')
74          h14=net.addHost('h14', cls=Host,mac='00:00:00:00:00:14')
75          h15=net.addHost('h15', cls=Host,mac='00:00:00:00:00:15')
76          h16=net.addHost('h16', cls=Host,mac='00:00:00:00:00:16')
77          h17=net.addHost('h17', cls=Host,mac='00:00:00:00:00:17')
78
79          info('***Creating links\n')
80
81          net.addLink('s1','h1')
82          net.addLink('s1','s2')
83          net.addLink('s1','s3')
84          net.addLink('s1','s4')
85          net.addLink('s1','s5')
86          net.addLink('s2','s10')
87          net.addLink('s2','s6')
88          net.addLink('s3','s11')
89          net.addLink('s3','s7')
90          net.addLink('s4','s12')
```

```
91          net.addLink('s4','s8')
92          net.addLink('s5','s9')
93          net.addLink('s5','s13')
94          net.addLink('s10','h2')
95          net.addLink('s10','h3')
96          net.addLink('s6','h4')
97          net.addLink('s6','h5')
98          net.addLink('s11','h6')
99          net.addLink('s11','h7')
100         net.addLink('s7','h8')
101         net.addLink('s7','h9')
102         net.addLink('s12','h10')
103         net.addLink('s12','h11')
104         net.addLink('s8','h12')
105         net.addLink('s8','h13')
106         net.addLink('s9','h14')
107         net.addLink('s9','h15')
108         net.addLink('s13','h16')
109         net.addLink('s13','h17')
110
111
112
113         info('***Starting network\n')
114         net.build()
115         net.start()
116
117
118         info('***Starting controller\n')
119         for controller in net.controllers:
120             controller.start()
121
122         info('***Starting switches\n')
123         net.get('s1').start([c0])
124         net.get('s2').start([c0])
125         net.get('s3').start([c0])
126         net.get('s4').start([c0])
127         net.get('s5').start([c0])
128         net.get('s6').start([c0])
129         net.get('s7').start([c0])
130         net.get('s8').start([c0])
131         net.get('s9').start([c0])
132         net.get('s10').start([c0])
133         net.get('s11').start([c0])
134         net.get('s12').start([c0])
135         net.get('s13').start([c0])
136
137         net.hosts[0].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h1-eth0")
138         net.hosts[1].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h2-eth0")
139         net.hosts[2].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h3-eth0")
140         net.hosts[3].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h4-eth0")
141         net.hosts[4].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h5-eth0")
142         net.hosts[5].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h6-eth0")
143         net.hosts[6].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h7-eth0")
144         net.hosts[7].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h8-eth0")
```

```
145         net.hosts[8].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h9-eth0")
146         net.hosts[9].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h10-eth0")
147         net.hosts[10].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h11-eth0")
148         net.hosts[11].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h12-eth0")
149         net.hosts[12].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h13-eth0")
150         net.hosts[13].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h14-eth0")
151         net.hosts[14].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h15-eth0")
152         net.hosts[15].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h16-eth0")
153         net.hosts[16].cmd("route add -net 224.0.0.0 netmask 224.0.0.0
                dev h17-eth0")
154
155         info('***Post configure switches and hosts\n')
156         CLI(net)
157         net.stop()
158
159 if __name__ == "__main__":
160         setLogLevel( 'info' )
161         myNetwork()
```

Apéndice B

Tablas IPv4

B.1. Red anillo

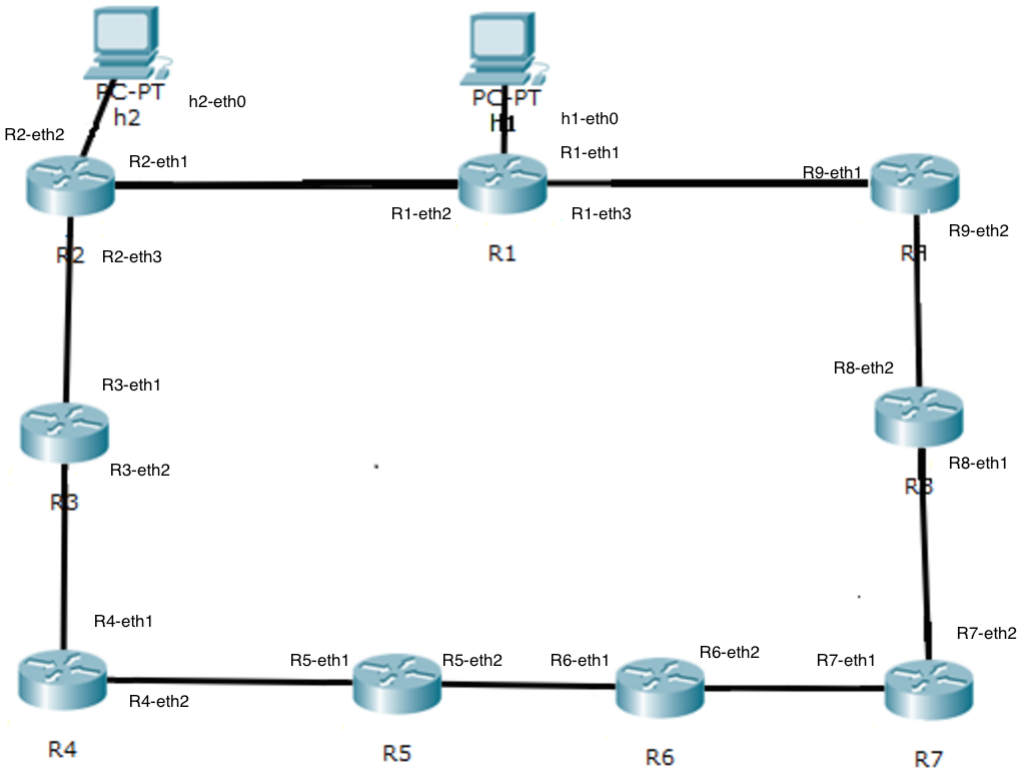

Figura B.1: Red anillo.

Interfaz	Dirección IP
R1-eth1	10.0.0.2/24
R1-eth2	20.0.0.1/24
R1-eth3	20.0.8.2/24
R2-eth1	20.0.0.2/24
R2-eth2	6.0.0.2/24
R2-eth3	20.0.1.1/24
R3-eth1	20.0.1.2/24
R3-eth2	20.0.2.1/24
R4-eth1	20.0.2.2/24
R4-eth2	20.0.3.1/24
R5-eth1	20.0.3.2/24
R5-eth2	20.0.4.1/24
R6-eth1	20.0.4.2/24
R6-eth2	20.0.5.1/24
R7-eth1	20.0.5.2/24
R7-eth2	20.0.6.1/24
R8-eth1	20.0.6.2/24
R8-eth2	20.0.7.1/24
R9-eth1	20.0.8.1/24
R9-eth2	20.0.7.2/24
h1-eth0	10.0.0.1/24
h2-eth0	6.0.0.1/24

B.2. Red en forma de árbol

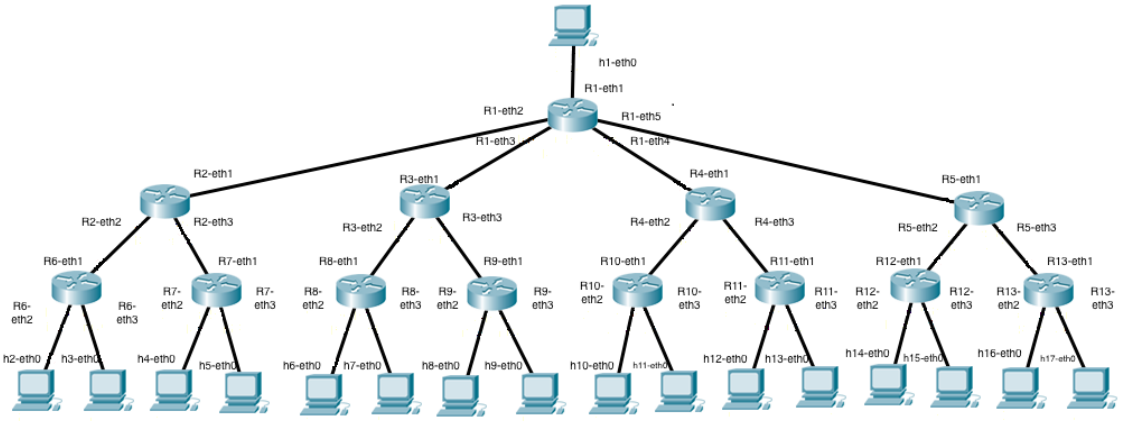

Figura B.2: Red árbol.

Interfaz	Dirección IP	Interfaz	Dirección IP
h1-eth0	10.0.0.1	R4-eth2	20.0.7.1
h2-eth0	6.0.0.1	R4-eth3	20.0.8.1
h3-eth0	6.0.0.2	R5-eth1	20.0.9.2
h4-eth0	7.0.0.1	R5-eth2	20.0.10.1
h5-eth0	7.0.0.2	R5-eth3	20.0.11.1
h6-eth0	8.0.0.1	R6-eth1	20.0.2.2
h7-eth0	8.0.0.2	R6-eth2	7.0.0.3
h8-eth0	9.0.0.1	R6-eth3	7.0.0.4
h9-eth0	9.0.0.2	R7-eth1	20.0.5.2
h10-eth0	11.0.0.1	R7-eth2	9.0.0.3
h11-eth0	11.0.0.2	R7-eth3	9.0.0.4
h12-eth0	12.0.0.1	R8-eth1	20.0.8.2
h13-eth0	12.0.0.2	R8-eth2	12.0.0.3
h14-eth0	13.0.0.1	R8-eth3	12.0.0.4
h15-eth0	13.0.0.2	R9-eth1	20.0.10.2
h16-eth0	14.0.0.1	R9-eth2	13.0.0.3
h17-eth0	14.0.0.2	R9-eth3	13.0.0.4
R1-eth1	10.0.0.2	R10-eth1	20.0.1.2
R1-eth2	20.0.0.1	R10-eth2	6.0.0.3
R1-eth3	20.0.3.1	R10-eth3	6.0.0.4
R1-eth4	20.0.6.1	R11-eth1	20.0.4.2
R1-eth5	20.0.9.1	R11-eth2	8.0.0.3
R2-eth1	20.0.0.2	R11-eth3	8.0.0.4
R2-eth2	20.0.1.1	R12-eth1	20.0.7.2
R2-eth3	20.0.2.1	R12-eth2	11.0.0.3
R3-eth1	20.0.3.2	R12-eth3	11.0.0.4
R3-eth2	20.0.4.1	R13-eth1	20.0.11.2
R3-eth3	20.0.5.1	R13-eth2	14.0.0.3
R4-eth1	20.0.6.2	R13-eth3	14.0.0.4

Bibliografía

[1] Hugh Holbrook and Brad Cain. Source-specific multicast for ip. Technical report, RFC 4607, August, 2006. https://tools.ietf.org/html/rfc4601.

[2] R Coltun, D Ferguson, and J Moy. A. lindem,.ospf for ipv6. Technical report, RFC 5340, July, 2008. https://tools.ietf.org/html/rfc1247.

[3] Algoritmo de Dijkstra, Wikipedia. https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra.

[4] BRITE. https://www.cs.bu.edu/brite/.

[5] Controlador Cisco SDN Controllers. https://www.cisco.com/c/en/us/products/cloud-systems-management/open-sdn-controller/index.html.

[6] Controlador Juniper Contrail. http://www.juniper.net/support/downloads/.

[7] Controlador ONOS. https://wiki.onosproject.org/display/ONOS/Device+Driver+Subsystem.

[8] Controlador OpenContrail. http://www.opencontrail.org/.

[9] Controlador Ryu, SDN hub. http://sdnhub.org/tutorials/ryu/.

[10] Controlador VMware. https://my.vmware.com/web/vmware/downloads.

[11] Eclipse. https://marketplace.eclipse.org/category/free-tagging/sdn.

[12] IP Multicast Technology Overview, Cisco Visual Networking Index. http://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/ip_multicast/White_papers/mcst_ovr.html.

[13] Kunihiro Ishiguro. Quagga software routing suite, 1991. http://www.nongnu.org/quagga/.

[14] La nueva interfaz radio 5G. http://www.raing.es/sites/default/files/La%20nueva%20interfaz%20radio%205G%20-%20v01.pdf.

[15] Mininet. http://mininet.org/. [Online; Última visita: 18/05/2017].

[16] Mininet, overview. http://mininet.org/overview/. [Online; Última visita: 17/06/2017].

[17] Mininet, Walkthrough. http://mininet.org/walkthrough/. [Online; Última visita: 17/06/2017].

[18] OPEN NETWORK FUNDATION. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[19] OpenDayLight. https://www.opendaylight.org/.

[20] OpenFlow. http://archive.openflow.org/wp/learnmore/.

[21] *OpenFlow Cookbook*. Packt Publishing Ltd., 35 Livery Street Birmingham B3 2PB, UK, 1st edition edition.

[22] OpenFlow Wikipedia. https://es.wikipedia.org/wiki/OpenFlow.

[23] OpenNetworking. https://www.opennetworking.org/sdn-resources/openflow.

[24] OpenvSwitch. http://openvswitch.org/.

[25] PIM-SSM. https://www.juniper.net/documentation/en_US/junos/topics/concept/multicast-pim-ssm.html.

[26] PIM-SSM. https://www.juniper.net/documentation/en_US/junos/topics/concept/multicast-pim-ssm-introduction.html.

[27] Raera.

[28] RFC 1142: IS-IS version 1. 1990. https://tools.ietf.org/html/rfc1142.

[29] RFC 2453: RIP version 2.Request for Comments, 2453, 1998. https://tools.ietf.org/html/rfc2453.

[30] RFC 5246: The Transport Layer Security Protocol, Version 1.2. https://tools.ietf.org/html/rfc5246.

[31] RFC 5440: Path Computation Element (PCE) Communication Protocol (PCEP). https://tools.ietf.org/html/rfc5440.

[32] RFC 6101: The Secure Sockets Layer Protocol, Version 3.0. https://tools.ietf.org/html/rfc6101.

[33] RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core. https://tools.ietf.org/html/rfc6120.

[34] RFC 6241: Netconf. https://tools.ietf.org/html/rfc6241.

[35] Rfc 7426: Barea.

[36] RFC 7922: Interface to the Routing System (I2RS) Traceability: Framework and Information Model. https://tools.ietf.org/html/rfc7922.

[37] RFC 8040: Restconf. https://tools.ietf.org/html/rfc8040.

[38] statica.com. https://es.statista.com/estadisticas/600134/trafico-ip-total-del-segmento-hogar-en-el-mundo-2015-2020/.

[39] Switch OpenFlow. http://yuba.stanford.edu/cs244wiki/index.php/Overview.

[40] Wikipedia, Algoritmo de Dijkstra. https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra.

[41] Wireshark, user guide. https://www.wireshark.org/download/docs/user-guide-a4.pdf. [Online; Última visita: 19/06/2017].

[42] www.ietf.org. https://www.ietf.org/rfc/rfc1112.txt.

[43] Y Rekhter and T Li. Rfc 1771. A Border Gateway Protocol, 4:1–54, 1995. https://tools.ietf.org/html/rfc1771.

[44] Alexandru Vulpe George Suciu Octavian Fratu Eduard C. Popovici Alexandru L. Stancu, Simona Halunga. A comparison between several software defined networking controllers. 90:223–226, January 2015.

[45] Rajendra Chayapathi. *Network function virtualization (nfv) with a touch of sdn*. Addison-Wesley Professional, Indianapolis, IN, 1st edition edition, 2016.

[46] Alexander Craig, Biswajit Nandy, Ioannis Lambadaris, and Peter Ashwood-Smith. Load balancing for multicast traffic in SDN using real-time link cost modification. In *Communications (ICC), 2015 IEEE International Conference on*, pages 5789–5795. IEEE, 2015.

[47] Carlos Santamaría Espinosa. Protocolos multicast en redes sdn. 152:180, 2016.

[48] Ying-Dar Lin, Yuan-Cheng Lai, Hung-Yi Teng, Chun-Chieh Liao, and Yi-Chih Kao. Scalable multicasting with multiple shared trees in software defined networking. *Journal of Network and Computer Applications*, 78:125–133, January 2017.

[49] Julius Ruckert, Jeremias Blendin, Rhaban Hark, and David Hausheer. Flexible, Efficient, and Scalable Software-Defined Over-the-Top Multicast for ISP Environments With DynSdm. *IEEE Transactions on Network and Service Management*, 13(4):754–767, December 2016.