TRABAJO FIN DE GRADO

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Protocolos Multicast en redes SDN

Autor
Carlos Santamaría Espinosa

Directores
Jorge Navarro Ortiz
Juan Manuel López Soler

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Julio de 2016

PROYECTO FIN DE CARRERA

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Protocolos Multicast en redes SDN

Autor
Carlos Santamaría Espinosa

Directores
Jorge Navarro Ortiz
Juan Manuel López Soler

Departamento de Teoría de la Señal, Telemática y Comunicaciones

—

Granada, Julio de 2016

Protocolos Multicast en redes SDN

Carlos Santamaría Espinosa

Palabras clave: Software Defined Network (SDN), Multicast, OpenDaylight, PIM Source Specific Multicast (PIM-SSM), Mininet, Java, Controlador, Quagga, Multimedia, Internet Protocol Television (IPTV), Qpimd.

Resumen

Actualmente, el avance desmesurado de los servicios ofrecidos por Internet esta desencadenando un aumento vertiginoso del número de usuarios que deciden hacer uso de las distintas tecnologías multimedia ofrecidas por los proveedores de *Streaming* de vídeo o IPTV para satisfacer sus necesidades de ocio. Empresas como Netflix, Vodafone y Moviestar han apostado fuerte por estos servicios provocando que cada vez más, sea más atractivo hacer uso de estos servicios debido a las facilidades y versatilidad que ofrecen.

En consecuencia, el tráfico multicast cursado por la red esta aumentando exponencialmente, provocando en un futuro próximo que los servicios multimedia lleguen a un punto en el que no puedan ofrecer la calidad necesaria para satisfacer a sus clientes. En relación a estas limitaciones de la red actual surge un nuevo paradigma de red, las redes definidas por software (SDN).

Software Defined Networking se convierte en una solución eficiente para mejorar a las redes actuales, ofreciendo una alternativa a los problemas evidenciados por los servicios multimedia. SDN propone una separación del plano de control y datos, permitiendo gestionar la red a partir de un controlador con una visión global del sistema, presentando una versatilidad y flexibilidad en las redes, inexistentes en las soluciones actuales.

Del mismo modo surgen protocolos *Open Source* para la comunicación entre el controlador y los dispositivos de red, en ese sentido se impone OpenFlow como estándar *de facto* siendo el más conocido hasta la fecha. OpenFlow no tiene dependencias con los distintos proveedores de elementos de red, gracias a ser una especificación abierta y estandarizada. En base a lo anterior, grandes empresas como Cisco, HP, Juniper o Google se han involucrado en proyectos de desarrollo de las redes SDN.

En este contexto, nace la idea de el presente Trabajo Fin de Grado con el objetivo de diseñar e implementar una solución multicast para *software defined networks*. Uno de los protocolos multicast más común para las redes actuales es PIM-SSM, que se emplea por ejemplo en los servicios de IPTV. Este trabajo presentará una solución como PIM-SSM basada en el paradigma SDN, la cual será comparada con una implementación *Open Source*

del protocolo PIM-SSM usando Quagga una *suite* de *routing*. Esta solución se llevará a cabo mediante la plataforma OpenDaylight, como controlador SDN, y el emulador de red Mininet, como una red compuesta por *routers*, *switches* y *hosts*.

En resumen, el estudio y desarrollo de nuevos servicios y protocolos en esta nueva arquitectura de red por parte de la comunidad global, será clave en los años venideros, ya que sin ningún lugar a dudas las redes definidas por software serán la clave del *networking* moderno.

Multicast Protocols in SDN networks

Carlos Santamaría Espinosa

Keywords: Software Defined Networking (SDN), Multicast, OpenDaylight, PIM Source Specific Multicast (PIM-SSM), Mininet, Java, Controllerr, Quagga, Multimedia, Internet Protocol Television (IPTV), Qpimd.

Abstract

Nowadays, the excessive development of the services offered by Internet is causing a vertiginous raise on the number of users who decide to use the different multimedia technologies offered by the video Streaming or IPTV providers to satisfy their leisure necessities. Companies as Netflix, Vodafone or Movistar have strongly bet for these services causing even a bigger attraction to the use of these services due to the facilities and the versatility that these are providing.

In consequence, the multicast traffic carried by the network is exponentially rising, causing in a close future that the multimedia services arrive to a point where networks cannot guarantee enough quality to satisfy their customers. In relation to these actual network limitations a new network paradigm arise, software definded networks (SDN).

Software Defined Networking becomes an optimal solution to eliminate the actual networks limitation, offering an exit way to the evidence problems by the multimedia services. This architecture proposes a separation of the control and the data plane, allowing network operators to manage them from a controller with a global vision of the system, presenting versatility and flexibility on the networks never seen on current solutions.

On the same way, it also arise Open Source protocols for the communication between the controller and the network dispositive, being OpenFlow considered as *de facto* standard. OpenFlow eliminates the dependence to the different network elements manufacturers, thanks to the common characteristics use to develop it. Based on OpenFlow,, large companies, such as Cisco, HP, Juniper or Google have been involucrate on the development of SDN solutions.

On this context, this degree thesis was devised with the objective of designing and implementing a multicast solution for sotware defined newtorks. One of the most common multicast protocol for current networks is PIM-SSM, being employed for e.g IPTV services. This work will present a PIM-SSM-like solution based on the SDN paradigm, wich will be compared to an Open Source implementation of the PIM-SSM protocol using Quagga

routing suite. This solution will be implemented using the OpenDaylight platform, as the SDN controller, and the Mininet network emulator framework, as a realistic virtual network comprising routers, switches and hosts.

Summarizing, the study and development of the new services and protocols in this new network architecture from the global community, it will be the key for the years that are coming, since without a doubt the software defined networks will be the key of the modern networking.

Yo, **Carlos Santamaría Espinosa**, alumno de la titulación de Grado en Ingeniería de Tecnologías de la Telecomunicación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, 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: Carlos Santamaría Espinosa

Granada a 10 de Julio de 2016 .

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

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

Informan:

Que el presente trabajo, titulado ***Protocolos Multicast en redes SDN***, ha sido realizado bajo su supervisión por **Carlos Santamaría Espinosa** , 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 10 de Julio de 2016 .

Los directores:

Jorge Navarro Ortiz **Juan Manuel López Soler**

Agradecimientos

Ha sido un largo camino hasta llegar a este punto. Después de todo el esfuerzo y horas dedicadas, no me arrepiento en ninguna de las decisiones tomadas que me han hecho llegar aquí, con dedicación y constancia todo se puede conseguir. Quiero aprovechar para agradecer a todas aquellas personas que me han apoyado a lo largo de estos 4 años.

A mis Padres, a mi Hermano y a María por acompañarme durante todo el trayecto. Por su confianza y apoyo incondicional sobre cualquier decisión, por todos los valores y experiencias transmitidas que me han convertido en lo que soy hoy, por no dejarme rendirme, Gracias.

A mis compañeros de clase, amigos y todas aquellas personas que he tenido el placer de conocer durante mi camino por Granada , por los buenos y malos momentos, por las noches en vela estudiando y las amistades conseguidas. Seguro que podremos añadir nuevas experiencias a la lista.

A mis tutores Jorge y Juanma, sin ellos no habría sido posible la realización de este trabajo. Por sus consejos , confianza y su constante ayuda, por hacerme crecer como profesional y saber correr cuando más lo necesitaba.

Gracias a todos.

Índice general

Índice de figuras

Índice de tablas

Lista de acrónimos

ACL Access Control List

AD-SAL Aplication Driven Sal

API Application Programming Interface

ARP Address Resolution Protocol

ASM Any Source Multicast

AvRA Avalanche Routing Algorithm

BAERA Branch Aware Edge Reduction Algorihm

BGP Border Gateway Protocol

BST Branch-aware Steiner Tree

CBT Core Based Trees

CDN Content Delivery Network

CLI Call Level Interface

DLUX Opendaylight User Experience

DoS Denial of Service

DVMRP Distance Vector Multicast Routing Protocol

IDE Integrated Development Environment

IGMP Internet Group Management Protocol

IP Internet Protocol

IPTV Internet Protocol Television

LLDP Link Layer Discovery Packet

MAC Media Access Control

MD-SAL Module Driven SAL

MDT Multicast Distribution Tree

MFT Multicast Fowarding Table

MOS Mean Opinion Score

MOSPF Multicast Open Shortest Path First

NB NorthBound

OIL Outgoing Interface List

ONF Open Networking Foundation

OSPF Open Shortest Path First

PIM Protocol Independent Multicast

PIM-DM PIM Dense Mode

PIM-SM PIM Sparse Mode

PIM-SSM PIM Source Specific Multicast

QoE Quality of Experience

QoS Quality of Service

RAERA Recover Aware Edge Reduction Algorihm

RIP Routing Information Protocol

RFC Request for Comments

RP Rendevous Point

RPF Reverse Path Fowarding

RPT Rendevous Path Tree

RST Recover-aware Steiner Tree

SAL Service Abstraction Layer

SB SouthBound

SDN Software Defined Network

SPT Shortest Path Tree / Spanning Tree Protocol

SSM Source Specific Multicast

SSL Secure Sockets Layer

ST Steiner Tree

TFG Trabajo Fin de Grado

TTL Time to live

URL Uniform Resource Locator

Capítulo 1

Introducción y bibliografía

En este capítulo se introducen todos los aspectos relevantes en el proyecto. Primero se identifican las motivaciones principales que han originado la realización de este trabajo, la situación actual de las tecnologías directamente relacionadas y una valoración sobre la relevancia del mismo. En el Apartado 2 se incluye una revisión del estado del arte en la que se comentan los artículos, trabajos y casos de éxito relacionados con el presente proyecto.

Tras este apartado, se identifican los objetivos a alcanzar inicialmente planteados en el anteproyecto, junto con la planificación y estructura del proyecto. Esta información permitirá comprobar el grado de cumplimiento de los objetivos inicialmente planteados.

Finalmente, se identifican las distintas fuentes bibliográficas utilizadas para la comprensión y realización de forma satisfactoria del proyecto.

1.1. Contexto y motivación

En esta sección se justifica la elección de este proyecto, resaltando especialmente la importancia de las diferentes tecnologías empleadas y su repercusión en la red. Para ello, se considerarán informes y estadísticas reales recopilados por grandes empresas de este sector.

Además, se contextualiza el estado de los servicios multimedia en la actualidad, justificando de esta forma la importancia de los protocolos multicast hoy en día y cómo su desarrollo es realmente beneficioso para la mejora de las prestaciones y operativa de red.

1.1.1. El tráfico multimedia en la actualidad

En las últimas dos décadas se ha producido un aumento exponencial en la cantidad de tráfico IP cursado por la red. Hace más de veinte años, en 1992 la red transportaba unos 100GB por día, alcanzando actualmente los 16.144GB por segundo (véase la Tabla (1.1)) Esto se debe principalmente a la mejora de las conexiones y a la aparición de nuevos servicios de red, que hacen cada vez más atractivo y útil el uso de aplicaciones en Internet, provocando un aumento significativo del número de usuarios beneficiarios de las nuevas tecnologías. Además, según las previsiones anuales de Cisco, el número de dispositivos y conexiones por hogar aumentarán considerablemente con el paso de los años (Figura (1.1)).

Year	Global internet Traffic
1992	100GB per day
1997	100GB per hour
2002	100GBps
2007	2000GBps
2015	20.235GBps
2019	61.386GBps

Tabla 1.1: Histórico de Internet.

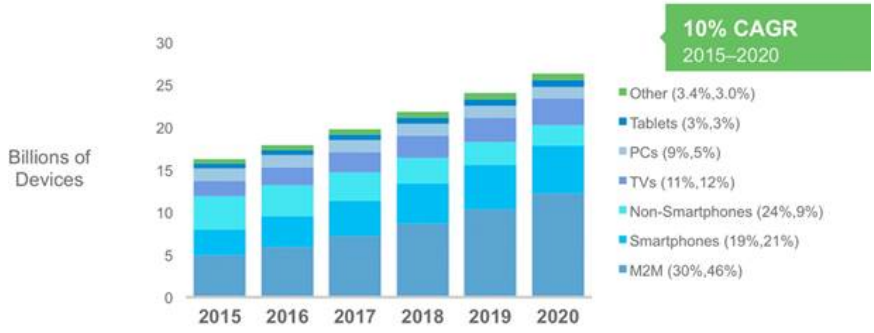


Figura 1.1: Tráfico IP global 2014-2019. [53]

El principal responsable del aumento de tráfico en la red, es el incremento de los servicios multimedia, sobre todo los basados en vídeo; *streaming* de vídeo, videoconferencias, IPTV, televisión, vídeo bajo demanda (ver Figura (1.2)).

Las compañías han visto en Internet una oportunidad para hacer llegar a todos los usuarios sus contenidos, de manera que sea fácil para los clientes

hacer uso de ellos y obtener grandes beneficios a bajo costo, ejemplos de estos servicios pueden ser Netflix [14], Youtube [35], wuaki.tv [34], Atresplayer [3], entre otros.

Para ver realmente la importancia de los servicios multimedia, según el informe anual de Cisco *Visual Networking Index* el contenido multimedia sera el 80 % de todo el tráfico de usuario en Internet en 2019 [52]. Por lo tanto, en unos años el volumen de tráfico será tan alto que se necesitarán nuevos métodos para hacer frente a esta carga, ya sean a partir de la mejora de los protocolos multicast o con el uso de nuevas tecnologías como por ejemplo las redes definidas por software (SDN).

Figura 1.2: Tráfico IP global por aplicaciones. [53]

1.1.2. Calidad de experiencia del usuario (Quality of Experience (QoE)

El crecimiento de la demanda de los servicios multimedia se estima que obedecerá a un incremento exponencial en la cantidad de datos que se cursarán en la red, lo que incrementará exponencialmente la cantidad de datos multicast que atravesarán la red. Como es de esperar, este incremento provocará una perdida en la eficiencia de la red, ya que habrá más congestión y la calidad que el usuario percibirá sera peor.

La provisión de todos estos servicios multimedia (y otros) tiene como principal finalidad la obtención de beneficios. Así, para dar una idea del impacto de esta fuerte demanda -si no cambia la tecnología actual- es conveniente definir el termino Calidad de Experiencia. La QoE se define por la ITU-T en [36] como la aceptabilidad general de una aplicación o servicio, tal y como la percibe subjetivamente el usuario final.

En [40] se propone un algoritmo para obtener el grado de opinión que tienen los usuarios en función del tiempo de respuesta de los contenidos. Para

medir la calidad percibida, se hace uso de la clasificación MOS, esquema empleado en los test de calidad del teléfono tradicional [9]. En la Tabla (1.2) se explica el significado que se suele asociar a cada valor MOS.

Score	User experienced Quality
5	Excellent, I really like the way it works
4	Good, but I can see a few possible improvements
3	Acceptable
2	Somewhat annoying, but I can live with it
1	Terrible, I will not use it unless it is absolutely necessary

Tabla 1.2: User experience quality ratings.

Como se puede observar en la Figura (1.3) el grado de opinión desciende rápidamente cuando el tiempo de respuesta deja de ser inmediato y, como se puede apreciar en la Tabla (1.2) anterior, a partir de un MOS de 3 los usuarios podrían empezar a utilizar otros servicios, ya que cabría pensar que puede haber una mejora notable en la experiencia si usaran otro proveedor.

De esta manera, se evidencia la importancia que tiene el tiempo de respuesta en la percepción de este tipo de servicios; concretamente, es necesario que este sea lo menor posible y por tanto, para ello la eficiencia de la red y de los protocolos utilizados tiene que ser máxima.

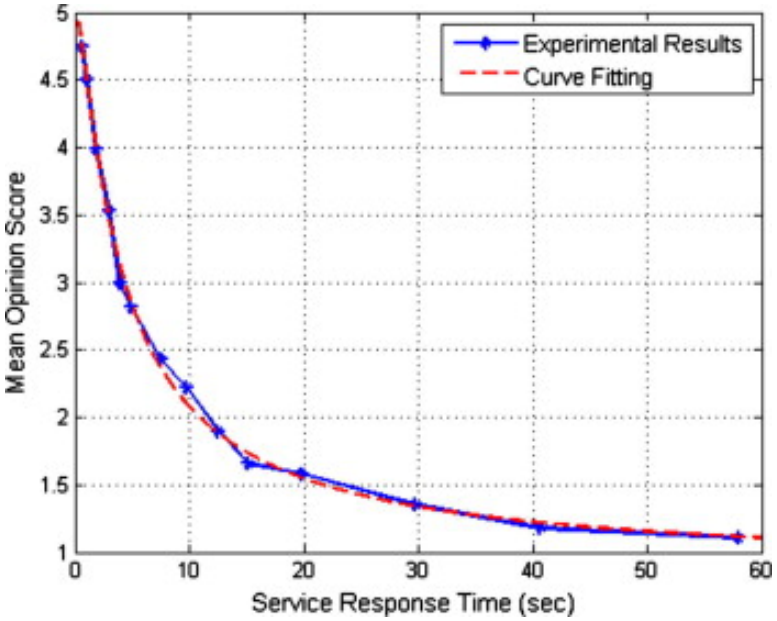


Figura 1.3: MOS en función del tiempo de respuesta. [40]

Como consecuencia, nótese que si no se realiza una inversión en la mejora de la red a partir del uso tecnologías más eficientes -como las redes definidas por software o incluso redes de entrega de contenidos (Content Delivery Network (CDN))- por las razones comentadas, se pueden producir grandes pérdidas para las empresas que basan su negocio en la entrega de servicios multimedia. Este aspecto hace interesante el principal objetivo de este trabajo fin de grado que no es otro sino adecuar los protocolos multicast a las novedosas redes definidas por software.

1.1.3. Limitaciones de las redes actuales

Para comprender mejor porqué las SDN son tan importantes es conveniente analizar las redes actuales. Tal y como se ha comentado anteriormente, con el incremento del tráfico de datos y la llegada de nuevas aplicaciones, se ha producido un cambio vertiginoso en la complejidad y velocidad necesarias en las redes. Además, el aumento de la popularidad de distintos servicios como *Big Data*, *Cloud Computing* y *Streaming*, requiere que los proveedores de servicios mejoren drásticamente la arquitectura de red, si quieren proveer una Quality of Service (QoS) y QoE apropiadas [69].

Las redes actuales tienen demasiadas limitaciones para el uso eficiente de este nuevo paradigma, de esta forma la arquitectura de red se esta volviendo inadecuada ante el inminente crecimiento de la complejidad, variabilidad y carga de tráfico.

Open Networking Foundation (ONF) identifica algunas de las limitaciones más importantes en las redes actuales [48]:

- **Complejidad:** Para acomodar las redes a las necesidades de sus usuarios en general, la industria ha mejorado los protocolos de red para ser más seguros y eficientes. Sin embargo, los protocolos tienden a definirse de forma aislada e individualmente, resolviendo cada uno un problema específico, renunciando al potencial beneficio de adoptar una solución conjunta.
- **Políticas inconsistentes:** Para implementar una política que abarque a la red completamente, los administradores de red deben configurar miles de mecanismos y dispositivos. Por ejemplo, cada vez que una nueva máquina virtual se introduce en la red, puede implicar horas e incluso días, hasta que el administrador encargado re-configura las listas de acceso (Access Control List (ACL)) en toda la red.
- **Imposibilidad de escalabilidad:** A la vez que las demandas de los centros de datos aumentan rápidamente, la red debe crecer de la misma forma. Sin embargo, la red se vuelve más compleja con la suma de

cientos de miles de dispositivos de red que deben ser configurados y gestionados.

- **Dependencia del fabricante:** Las nuevas capacidades y servicios perseguidos por proveedores y empresas en respuesta rápida a las necesidades dinámicas de negocios y demanda de clientes, se ven frenadas por los ciclos de producción de los equipamientos por parte de los fabricantes, que pueden implicar periodos de hasta más de tres años.

Como se observa, las redes modernas se deben caracterizar por tener una gran flexibilidad y rapidez para soportar los distintos servicios actuales. Por ello, *Open Data Center Alliance* [39] identifica para las redes actuales y futuras los siguientes requerimientos críticos:

- **Adaptabilidad.** Las redes futuras tiene que ser capaces de ajustarse y responder dinámicamente en función de las condiciones de la red y las necesidades de las aplicaciones.
- **Seguridad.** Las aplicaciones de red deben integrar seguridad como un servicio básico y no como un complemento.
- **Mantenimiento.** La introducción de nuevas características y capacidades (actualizaciones de software y optimizaciones) deben de ser transparentes con una interrupción mínima de las operaciones.
- **Gestión de red.** El software de configuración de la red debe permitir su gestión de forma conjunta, evitando la configuración de elementos individuales.
- **Escalabilidad.** Las implementaciones deben de tener la capacidad de escalar fácilmente, en función de las necesidades.
- **Automatización.** Los cambios de política de red deben propagarse automáticamente de modo que el trabajo manual y los errores se reduzcan.

Una vez estudiadas las limitaciones de las redes actuales y las necesidades de las redes futuras para poder soportar la inminente llegada de los nuevos servicios, en el apartado siguiente se explican los objetivos principales establecidos en el presente proyecto.

1.2. Objetivos principales del proyecto

En este apartado se establecen los objetivos principales del proyecto, para en el capítulo II analizar su grado de cumplimiento.

El principal objetivo de este TFG es analizar el comportamiento de los protocolos multicast en una red emulada actual y realizar una conceptualización de los protocolos multicast en una red basada en una arquitectura de redes definidas por software. Como subobjetivos en este trabajo fin de grado se consideran los siguientes:

- Análisis del funcionamiento del protocolo PIM-SSM.
- Revisión del estado del arte.
- Caracterización del protocolo PIM-SSM en un escenario de referencia.
- Puesta en marcha del emulador de redes Mininet utilizando el controlador OpenDaylight.
- Implementación de un protocolo multicast en una red SDN.
- Pruebas y evaluación de la implementación realizada.

1.3. Revisión del estado del arte

En esta sección se pretende obtener una visión global sobre las implementaciones más relevantes existentes de protocolos multicast diseñados sobre redes definidas por software, con el objetivo de disponer de un sistema de referencia para realizar una comparación con nuestro proyecto y no trabajar en algo ya creado.

1.3.1. Avalanche: Data Center Multicast using Software Defined Networking

En [56] se presenta Avalanche un sistema basado en SDN que permite multicast en *switches* usados en *data centers*. Avalanche adopta un nuevo algoritmo de *routing* llamado Avalanche Routing Algorithm (AvRA), el cual trata de minimizar el tamaño del árbol de enrutamiento para cualquier grupo de multidifusión dado. AvRA intenta solucionar el problema del Steiner Tree (ST) aprovechando las SDN para optimizar los caminos. Concretamente, dada la gran diversidad de posibles caminos comunes en las redes de centros de datos se puede conseguir un ancho de banda altamente eficiente. En la Figura (1.4) se muestra la arquitectura de Avalanche usando OpenDaylight.

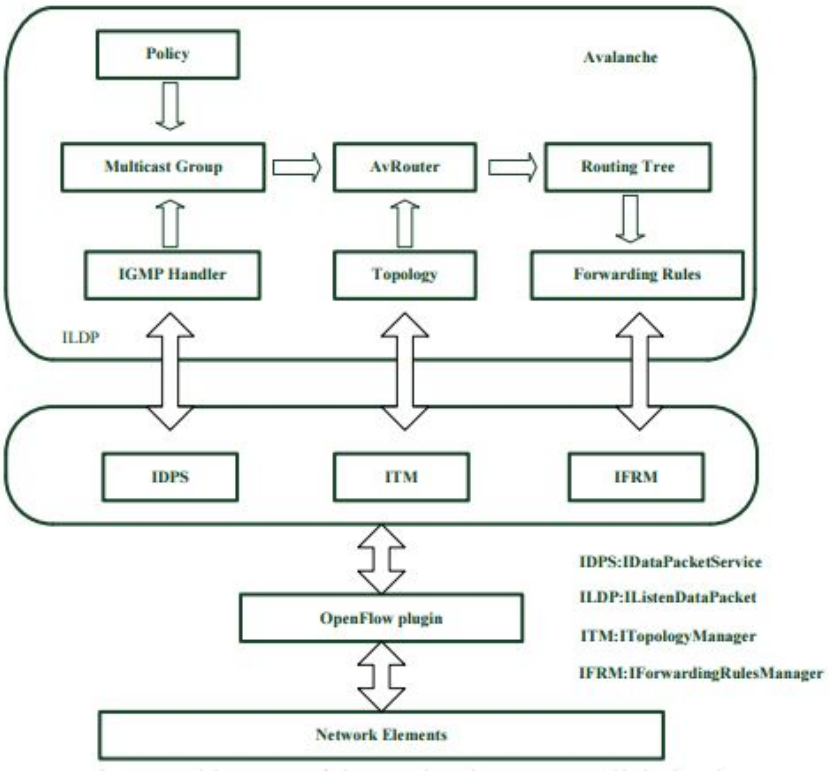


Figura 1.4: Arquitectura Avalanche. [56]

AvRA es un algoritmo que construye un árbol de enrutamiento intentando conectar cada nuevo miembro del grupo a un árbol existente en la intersección más cercana, en vez de intentar encontrar el camino más corto a un nodo especifico, como así lo hace PIM-SM. De esta forma AvRA es capaz de crear árboles de enrutamiento más eficientes que los algoritmos multicast actuales.

Avalanche se ha implementado como un módulo OpenFlow del controlador, así su emulación usando Mininet muestra una mejora en la velocidad de datos de un 12 % y reduce la pérdida de paquetes en un 51 % comparado con multidifusión IP. Además, se reducen el tamaño de los árboles de distribución y la cantidad de enlaces utilizados, confirmando que su algoritmo es más eficiente que el tradicional IP Multicast. La Figura (1.5) muestra que el porcentaje máximo de utilización de los distintos enlaces para PIM-SM está por encima del 150 % lo que significa un 50 % de sobre-suscripción para 5000 grupos de multidifusión, en cambio Avalanche se mantiene siempre por debajo del 100 %.

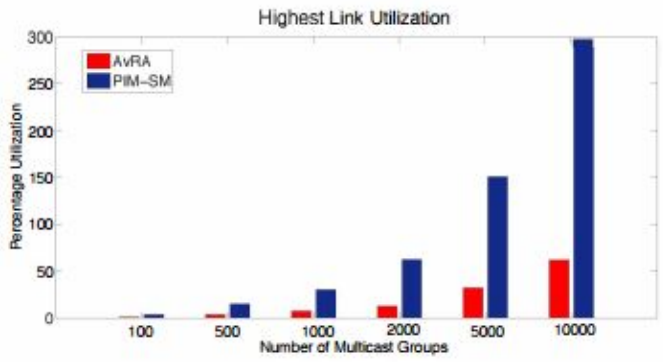


Figura 1.5: Porcentaje máximo de utilización links. [56]

1.3.2. Reliable Multicast Routing for Software-Defined Networks

La ingeniería de tráfico actual en SDN se centra principalmente en comunicaciones unicast. Por el contrario, en comparación con unicast, multicast puede reducir eficazmente el consumo de recursos de red sirviendo a varios clientes de forma conjunta. Dado que muchas aplicaciones importantes requieren transmisiones fiables, se prevé que la multidifusión juegue un papel relevante en las redes definidas por software. Sin embargo, el árbol Shortest Path Tree / Spanning Tree Protocol (SPT), adoptado en la red actual no es eficiente en términos de ancho de banda, mientras que el ST en la teoría de grafos no está diseñado para soportar transmisiones fiables, ya que la selección de los nodos de recuperación no se examina.

En [67] se propone un nuevo árbol de multidifusión fiable para SDN, llamado Recover-aware Steiner Tree (RST). El objetivo de RST es minimizar tanto los costes en la creación, como en la recuperación de árboles. Para ello se ha diseñado el algoritmo aproximado Recover Aware Edge Reduction Algorihm (RAERA) para resolver el problema. Tras realizar simulaciones en redes reales con tráfico real de Youtube, se demuestra que este esquema supera tanto a SPT como a ST, Tabla (1.3), además la implementación de un RST se realiza en unos pocos segundos, por lo que es práctico para redes SDN.

Algoritmos	Consumo de ancho de banda	Re-buffering
RAERA	13.18 MBytes	0.4 s
ST	16.39 MBytes	33.5 s
SPT	17.83 MBytes	7.8 s

Tabla 1.3: Comparación Algoritmos. [67]

1.3.3. Software Defined Network-Enabled Multicast for Multi-Party Video Conferencing Systems

En [75] se propone una arquitectura novedosa para videoconferencias multi-usuario (es decir, videoconferencias con múltiples participantes) mediante la utilización de SDN. El controlador ayuda al controlador multimedia en crear árboles multicast para los flujos de vídeo originados por la videoconferencia.

Para la realización de esta arquitectura se propone una innovadora construcción multicast y un método de empaquetado, en la que varios árboles de multidifusión basados en fuente se construyen y se integran para maximizar la utilidad de todo el sistema, al mismo tiempo que se garantiza un retardo de extremo a extremo. Tras extensas simulaciones se demuestra que esta solución proporciona una mejor distribución de vídeo en comparación con las soluciones convencionales en términos de velocidad de vídeo y latencia en redes de baja y alta densidad.

1.3.4. Scalable Steiner Tree for Multicast Communications in Software-Defined Networking

El problema de escalabilidad en SDN es más serio que en las redes tradicionales ya que el tráfico de red es más difícil de agregar, este problema ha sido estudiado para unicast pero no ha sido explorado en multicast. Para tratar de resolver este problema, en [51] se propone un nuevo árbol de multidifusión para SDN llamado Branch-aware Steiner Tree (BST).

La implementación del BST es difícil ya que necesita minimizar el número de enlaces y nodos del árbol.

Para resolver el problema anterior se ha diseñado un algoritmo de reducción llamado Branch Aware Edge Reduction Algorihm (BAERA). Los resultados de su simulación demuestran que los árboles obtenidos por BAERA son -en términos de ancho de banda- más eficientes y escalables que los árboles tradicionales *Steiner*. Lo más interesante es que BAERA es computacionalmente eficiente para ser desplegado en SDN, ya que puede generar un árbol en redes masivas en poco tiempo.

1.3.5. Scaling IP Multicast on Datacenter Topologies

IP multicast reduciría significativamente los *overhead* tanto de la red y de servidores para muchas aplicaciones de comunicación en *data centers*, desafortunadamente los protocolos tradicionales para la gestión de multidifusión no escalan adecuadamente con recursos hardware agregados en el número de grupos multicast soportados.

Así, en [59] se presenta un método general para ampliar el número de grupos de multidifusión compatibles. En lugar de tratar cada *switch* como una entidad independiente, aprovechan las ideas de almacenamiento horizontal para dividir el espacio de direcciones de multidifusión y distribuir las particiones de direcciones entre los *switches*.

Además, en [59] muestran como aprovechar la estructura topológica única de las redes de centros de datos con el objetivo de construir la primera arquitectura de multidifusión con escalabilidad horizontal.

1.4. Principales fuentes Bibliográficas

En este apartado, mostramos las referencias bibliográficas de mayor importancia para el desarrollo del proyecto, que han servido como apoyo y motivación para desarrollar los conceptos que en este trabajo se exponen.

- William Stalling. *FoundationS of Modern Networking.* Addison Wesley, 2016
- Beau Williamson. *Developing IP Multicast Networks*, volume 2. Cisco Press, 2000.
- *Mininet walkthrought* at `https://mininet.org/walkthrough`
- SDNHUB tutorial at `http://sdnhub.org/tutorials/opendaylight/`
- Ishiguro, KunihiroQuagga. *Software Routing Suite.*

1.5. Estructura de la memoria

En esta sección se enumeran los distintos capítulos del proyecto, realizando una breve descripción de ellos. En concreto la memoria técnica esta formada por ocho capítulos:

- **Capítulo 1. Introducción y bibliografía.** En este capítulo se introducen los aspectos relevantes en el proyecto, indicando las motivaciones que han hecho posible la implementación del mismo.
- **Capítulo 2. Análisis de objetivos y metodología.** En este capítulo se analiza el proyecto estableciendo los distintos objetivos para la correcta realización del proyecto.

- **Capítulo 3. Planificación y estimación de costes.** En este capítulo se proporciona una planificación a seguir en el desarrollo del proyecto, junto con una estimación de los costes previstos en el diseño del trabajo.
- **Capítulo 4. Desarrollo teórico.** En este capítulo, se resumen los contenidos teóricos necesarios para entender perfectamente el marco teórico relacionado con el proyecto.
- **Capítulo 5. Herramientas utilizadas.** En este apartado, se explican las distintas herramientas utilizadas en el desarrollo del proyecto, realizando ejemplos simples para comprender su funcionamiento.
- **Capítulo 6. Caracterización de un protocolo multicast. PIM-SSM.** Aquí se caracteriza el protocolo multicast PIM-SSM, con el objetivo de entender su funcionamiento para un posterior desarrollo de un protocolo multicast propio.
- **Capítulo 7. Desarrollo de un protocolo multicast sobre SDN.** Se describe el diseño de un protocolo multicast propio, basado en una arquitectura SDN, describiendo los pasos seguidos para la implementación del mismo.
- **Capítulo 8. Conclusiones y vías futuras.** En este último capítulo se aportan las conclusiones obtenidas tras la realización del proyecto, proponiendo posibles trabajos futuros.
- **Apéndices.** En los distintos apéndices, se muestran los diferentes códigos de topologías utilizadas en el desarrollo del proyecto, junto con varios manuales para la instalación y parcheado de las herramientas utilizadas.
- **Bibliografía**. En la parte final se muestra una bibliografía con toda las referencias que se han consultado para el desarrollo del trabajo.

Capítulo 2

Análisis de objetivos y metodología

En este capítulo se estudian distintas cuestiones previas necesarias para la realización del presente trabajo fin de grado.

Primero, se realiza un estudio de los distintos objetivos esenciales que se necesitan para el correcto desarrollo del proyecto, de tal manera que se implemente una planificación temporal que permita abordar el trabajo con éxito.

Después, se analizarán los distintos requisitos funcionales y no funcionales que se deben cumplir para el correcto funcionamiento de las distintas implementaciones; estos requisitos se obtendrán teniendo en cuenta los objetivos establecidos anteriormente.

A continuación, se expondrá la metodología establecida en el transcurso del proyecto, explicando como se han llevado a cabo cada una de las etapas.

Finalmente, se procederá a valorar los objetivos finales conseguidos, comparándolos con los establecidos en las primera fase del proyecto. Analizando de esta forma, el grado de cumplimiento en la realización del trabajo.

2.1. Objetivos

En esta sección se identifican los distintos objetivos planteados en el desarrollo del proyecto, junto con una planificación.

El principal objetivo de este proyecto es el desarrollo de un protocolo multicast propio en un entorno de redes definidas por software. Para ello, se programará un controlador de red para que se ajuste a las necesidades de los protocolos multicast.

Para el correcto desarrollo del protocolo, previamente se elaborará un escenario mínimo pero significativo donde se estudiará el protocolo multicast PIM-SSM; esto permitirá comprender su funcionamiento y las características de los protocolos de multidifusión, así como adquirir conocimiento para una futura implementación de un protocolo multicast basado en la arquitectura SDN.

Tras el estudio de los dos protocolos, se procederá a una etapa de evaluación y pruebas, donde se comprobará la existencia de diferencias significativas en términos de eficiencia y retardos en la red.

Como subobjetvos del proyecto, se incluyen el estudio teórico y práctico de la teoría a aplicar, junto con el estudio del estado del arte de implementaciones multicast usando arquitecturas SDN.

2.2. Especificación de requisitos

En este apartado se identifican los distintos requisitos necesarios para el correcto desarrollo del proyecto, con el objetivo de simplificar y aislar futuros problemas en la implementación. Los requisitos serán divididos en dos grupos, funcionales y no funcionales.

Gracias al estudio previo de los requisitos necesarios, se puede abordar con mayor facilidad la etapa de planificación del proyecto; además esto permitirá diferenciar y aislar en módulos diferentes los distintos problemas que se planteen durante el desarrollo del proyecto.

2.2.1. Requisitos funcionales

Pertenecen dentro de los requisitos funcionales todas aquellas características requeridas en la implementaciones a través de la adición de un sub-sistema o bloque de código. En concreto, existen cuatro requisitos funcionales esenciales:

- Primero, se necesita desarrollar un diseño de red a través de un *script* de python, se debe tener en cuenta la adición de la *suite* de *routing* Quagga en el *script* y las distintas características de la topología.
- Se necesita una correcta configuración de los ficheros que aporta Quagga, para que los protocolos multicast y la interconexión de la red se realice de forma correcta.
- En el caso de la implementación de un protocolo sobre arquitectura SDN, se debe implementar el mismo diseño de red, aunque esta vez no

conlleva la adición de Quagga, pero si la del controlador remoto de la red, lo que conlleva varios cambios en el script anterior.

- Finalmente, para la correcta implementación de el protocolo multicast es necesario establecer la programación del controlador remoto OpenDayLight, de forma que este pueda procesar el tráfico multicast, aprendiendo las diferentes rutas establecidas y direccionando los distintos paquetes hacia su destino.

2.2.2. Requisitos no funcionales

Los requisitos no funcionales son aquellas características requeridas del sistema, es decir, a diferencia de los funcionales, estos requisitos implican atributos o características no operativas del proyecto. Así, dentro de estos requisitos destacan las herramientas software utilizadas para el correcto desarrollo del proyecto.

- Para la implementación de los escenarios necesarios se utilizará el emulador de red *Mininet* [13], el cual permite la creación de las topologías personalizadas para realizar las pruebas multicast.
- Se usará la suite de routing *Quagga* [55] a la hora de aplicar funcionalidades de *routing* multicast en los diseños sin controlador.
- El controlador remoto utilizado será *OpenDayLight* [17], combinado con el emulador de red Mininet, que facilitará el desarrollo de nuestro protocolo multicast en un entorno SDN.
- En la creación de los escenarios y el diseño del protocolo multicast será necesario el uso de los lenguajes Java y Python, de forma que la programación sea sencilla y legible para los usuarios que quieran comprender su funcionamiento.
- Para el desarrollo del protocolo utilizara la maquina virtual SDN-HUB [32], la cual contiene instalado de forma nativa todo el software necesario para la correcta realización del proyecto.

En los capítulos siguientes se estudia detenidamente cada uno de los requisitos no funcionales utilizados durante el desarrollo, desde la instalación de los distintas herramientas software, hasta la utilización de los mismas.

2.3. Metodología

A continuación, se muestra el desarrollo ordenado cronológicamente que se ha seguido en el presente proyecto.

Tras, el conocimiento del propósito del trabajo fin de grado, se procede a un estudio teórico de toda la tecnología implicada, con el objetivo de comprender a la perfección cada una de las implementaciones a realizar. Además, se realiza una búsqueda de las soluciones multicast existentes, de forma que no solapemos ningún conocimiento o diseño ya implementado.

Una vez estudiada la tecnología, se comienza con el aprendizaje en el uso de las herramientas software necesarias, como Mininet y Quagga, probando la creación de topologías básicas y comprendiendo el funcionamiento de la API sobre python para el diseño de topologías personalizadas. Durante este proceso, surgieron algunas dificultades, ya que no fue fácil la combinación de Quagga con Mininet, debido a problemáticas con las versiones del software.

A continuación, adquiridos los conocimientos sobre Mininet y Quagga se procede a la creación de un escenario simple pero significativo, donde se estudia el funcionamiento del protocolo PIM-SSM. Observando cómo se distribuye el tráfico de datos en la red y estudiando posibles mejoras a implementar en nuestro protocolo propio en una arquitectura SDN.

Terminado el primer escenario, se realiza un estudio de OpenDaylight, comprendiendo el funcionamiento de los distintos programas ejemplo aportados por el software. Además, se realiza un estudio de los distintos paquetes y funciones aportadas por la herramienta software Java necesarios a la hora de programar el controlador.

Adquiridos los conocimientos básicos sobre OpenDayLight se inicia la creación del protocolo multicast basado en SDN. Cabe destacar que esta etapa fue la más difícil, debido a la escasa información existente sobre OpenDayLight; en esta etapa hubo que solventar muchas dificultades a la hora de encontrar errores en la programación del controlador. Finalmente, conseguimos realizar el objetivo permitiendo cursar tráfico multicast a través de la red diseñada.

Por último, estudiamos y evaluamos el protocolo multicast implementado, comparándolo con la eficiencia de las versiones actuales de los protocolos multicast como PIM-SSM.

2.4. Valoración de objetivos

Tras la finalización del Trabajo Fin de Grado, se puede decir que se han conseguido realizar todos los objetivos planteados en los apartados anteriores de forma satisfactoria. De esta forma, podemos comentar que ha habido un gran aprendizaje que ha resultado en la adquisición de nuevos conceptos y capacidades, desde el inicio del proyecto hasta su fin, aprendiendo y estudiando una tecnología novedosa y compleja como son las redes SDN.

Se ha conseguido implementar un controlador con capacidad multicast sobre una arquitectura de redes definidas por software, de forma que este sea un protocolo más flexible y escalable comparado con su diseño en escenarios sin controlador.

Finalmente, una vez terminado el diseño, se identifican las distintas posibilidades que nos ofrecen las redes SDN, no solo en el ámbito multicast, sino en todos los aspectos del *networking* futuro, así existe un mundo nuevo por descubrir en los próximos años venideros.

Capítulo 3

Planificación y estimación de costes

En este capítulo se presentan todos los aspectos relacionados con la planificación de la implementación del proyecto, junto con una estimación de los costes previstos.

Primero, se aborda la división del proyecto en diferentes tareas o módulos, con el objetivo de poder establecer una planificación estructurada en el desarrollo. Así, se realiza una valoración del tiempo dedicado a cada tarea en función de su dificultad, plasmando los resultados en un diagrama de Grant para obtener una visión más gráfica.

Una vez decidida la planificación a realizar junto con la temporización de las tareas, se identifican los recursos necesarios para el correcto desarrollo del proyecto.

Finalmente, se estiman los costes previstos en función de los recursos humanos y materiales usados en el presente proyecto.

3.1. Planificación

En este apartado se planifican las distintas tareas que llevaremos a cabo en el desarrollo del proyecto. De esta forma, se pueden establecer fechas de corte (o *deadlines*) en la planificación temporal para la resolución exitosa del proyecto. Para ello, se ha realizado un análisis de las distintas etapas, para intentar ajustarse lo máximo posible a la realidad.

A continuación, se procede a la división de las tareas o paquetes de trabajo establecidos.

PT1: Estudio teórico

En tarea se centra en el estudio y análisis de la información relacionada con el presente proyecto, tanto en el ámbito de las redes definidas por software, junto con los distintos protocolos multicast más importantes en la actualidad. Los objetivos principales son obtener una buena bibliografía para el correcto desarrollo del proyecto, junto con la revisión del arte en lo referente a protocolos multicast en las redes definidas por software.

PT2: Familiarización con Mininet

En esta etapa, se analizará el software de emulación de red Mininet, vital para el correcto desarrollo del proyecto. Se estudiarán las distintas posibilidades que ofrece, basándonos en su rendimiento. Esta tarea incluye igualmente una toma de contacto con python, con el objetivo de la creación de topologías personalizadas.

PT3: Toma de contacto con Quagga

En este módulo se realiza una primera toma de contacto con la suite de *routing* Quagga, realizando las instalaciones y parcheados necesarios para adecuar la herramienta a las diferentes necesidades. Se estudiará la arquitectura de Quagga junto con los distintos protocolos de *routing* que se pueden utilizar a través del uso de los procesos ejecutables (también denominado *demonios*) específicos. En concreto, se da prioridad en el aprendizaje a la configuración de ficheros y comandos en el uso de los demonios de Zebra y Qpimd.

PT4: Caracterización del protocolo PIM-SSM

En esta tarea se creará un escenario simple pero significativo para la caracterización del protocolo multicast PIM-SSM. Para ello, se utilizará el conocimiento adquirido en las tareas anteriores sobre Mininet y Quagga, así se implementará una topología compuesta por *routers* con capacidad multicast, gracias al uso de Quagga en los diferentes *routers* emulados por Mininet y se comprobará el funcionamiento experimental de PIM-SSM para su posterior comparación con el protocolo multicast desarrollado en redes definidas por software.

PT5: Aprendizaje de OpenDayLight

En este paquete de trabajo se estudiará el uso del controlador remoto OpenDaylight, con el objetivo de realizar un futuro diseño, para satisfacer las necesidades de los protocolos multicast. Se recordará el uso de Java, a la hora de realizar la programación del controlador remoto. Se comprenderán cómo se modifican las distintas tablas de flujo de los *switches* junto con las diferentes reglas principales que se deben aplicar, para realizar el emparejamiento (*matching*) en el controlador.

PT6: Desarrollo Multicast en SDN

Este sera el proceso más largo y difícil. Se implementará un escenario con la ayuda de Mininet, que contenga un controlador, el cual tendrá una visión global de la red. Se programará el controlador OpenDayLight para que se adecue a un escenario con protocolos multicast, intentando reducir de esta forma los mensajes que circulan en la red y por lo tanto incrementando la eficiencia y rapidez de la misma ante distintos escenarios.

PT7: Evaluación y pruebas

Tras realizar todo el diseño especificado en el proyecto, el desarrollo del Protocolo PIM-SSM en un escenario sin controlador y la creación de un protocolo multicast en una arquitectura SDN, se llevarán a cabo un conjunto de pruebas y evaluaciones, con el objetivo de descubrir posibles errores no encontrados en las fases anteriores. Además, se comparará el protocolo en el ámbito desarrollado, comprobando las mejoras introducidas a nivel de eficiencia y carga en la red.

PT8: Elaboración de una memoria técnica del proyecto

Finalmente, esta última tarea se basa en la elaboración de una memoria técnica que contenga todos los conocimientos teóricos y prácticos desarrollados en el presente proyecto. Se mostrarán todos los aspectos de diseño en las diferentes etapas de implementación establecidas, junto con resultados que avalen la utilidad y validez del proyecto. La elaboración de la memoria se lleva a cabo desde el inicio del proyecto, por lo que la convierte en la tarea más larga dentro de la planificación establecida.

Una vez establecidas las distintas tareas que contiene el proyecto, se establece una planificación a la hora intentar cumplir diferentes plazos temporales, con el objetivo de realizar de manera satisfactoria el proyecto. Así, en la Figura (3.1) se puede observar un diagrama de Grant que muestra

gráficamente toda la planificación desarrollada.

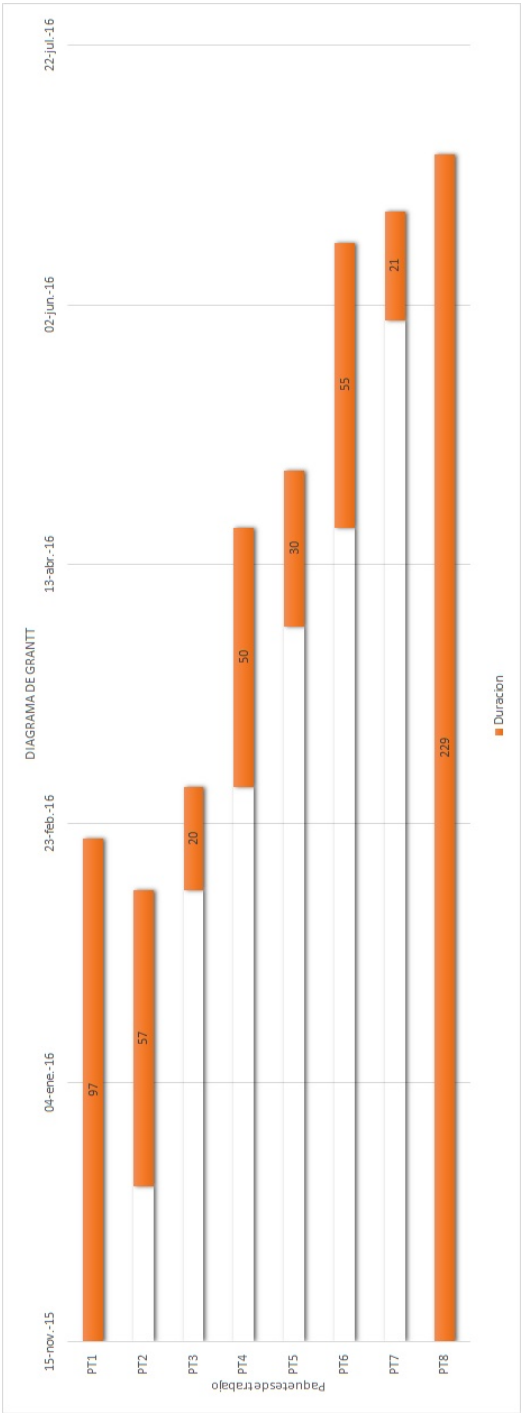


Figura 3.1: Diagrama de grantt

En el Diagrama de Grant se puede observar que se han establecido unos periodos temporales para la realización de las tareas, además algunas de ellas se realizan en paralelo con otras, de forma que la planificación diaria tiene que ser más exacta. En la Tabla (3.1) se pueden observar todas las estimaciones temporales realizadas en las distintas etapas establecidas en la planificación inicial.

Planificación temporal de las tareas		
Paquetes de trabajo	Descripción	Tiempo estimado
PT1	Estudio teórico	70 horas
PT2	Familiarización con Mininet	40 horas
PT3	Toma de contacto con Quagga	30 horas
PT4	Caracterización del protocolo PIM-SSM	60 horas
PT5	Aprendizaje de OpenDayLight	40 horas
PT6	Desarrollo multicast en SDN	70 horas
PT7	Evaluación y pruebas	30 horas
PT8	Elaboración de una memoria técnica del proyecto	120 horas

Tabla 3.1: Análisis temporal de las tareas.

Tras esto, tenemos planificado el desarrollo del proyecto en todos sus aspectos, cabe destacar que esta planificación es orientativa ya que probablemente los tiempos se modifiquen debido a posibles problemas en las distintas implementaciones.

3.2. Recursos utilizados

En esta sección se realiza un desglose de todos los recursos utilizados en el desarrollo del proyecto. Se clasifican los distintos recursos en tres categorías; humanos, hardware y software. Esta lista es de vital importancia a la hora de realizar una estimación de los costes necesarios.

3.2.1. Recursos humanos

- D. Jorge Navarro Ortiz y D. Juan Manuel López Soler, profesores del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada, en calidad de tutores del proyecto.

- Carlos Santamaría Espinosa, alumno del Grado en Ingeniería de Tecnologías de Telecomunicación y autor del presente proyecto.

3.2.2. Recursos hardware

- *Ordenador portátil Lenovo T510i Thinkpad*, con procesador intel core i5-M430 a 2.27GHz, memoria RAM de 4GB y disco duro de 287GB de capacidad. Usaremos el PC para la creación de los escenarios y la ejecución del controlador SDN.

3.2.3. Recursos software

- Sistema operativo *Windows 10 (64bits)*, instalado en el ordenador portátil sobre el que se implementan las pruebas.
- *Mininet*, emulador de red para el diseño de los distintos escenarios planteados.
- Maquina virtual *SDN-hub* que contiene instalada Mininet junto con el controlador OpenDayLight.
- *Eclipse Luna* (Integrated Development Environment (IDE) para desarrolladores de Java), entorno de programación para el diseño del controlador.
- *Quagga* suite de *routing* para el uso de protocolos Multicast en los elementos de nuestra red.
- *Wireshark*, *sniffer* de red para obtener la información sobre los distintos datos que atraviesan el escenario planteado.
- *Python*, Lenguaje de programación para el desarrollo de topologías personalizadas sobre Mininet.
- *TeXstudio*, procesador de texto para la elaboración de la memoria técnica del proyecto.

3.3. Estimación de costes

En esta sección, se aborda el desarrollo de una estimación de los costes que conlleva la realización del proyecto de una forma orientativa, mostrando visualmente el coste asociado a cada paquete de trabajo.

Cabe destacar, que no serán elevados ya que todos los recursos software son de código libre, por lo que no serán un conste añadido y por lo tanto no se plasmarán en el informe.

3.3.1. Recursos humanos

A continuación, se calculan los costes relativos a los recursos humanos, en función del tiempo invertido en las tareas predefinidas anteriormente. Así, recogeremos los datos temporales de la Tabla (3.1).

Para el calculo del coste se utilizan las siguientes reglas:

- El sueldo de un ingeniero junior se valora en 20 euros/hora.
- El sueldo medio de un Doctor de la Universidad de Granada se estima en 50 euros/hora.

Se considera un tiempo invertido de 20 horas, en el uso de tutorías y revisión de la memoria.

A partir de estas reglas se puede realizar el cálculo del coste asociado a las horas invertidas, Tabla (3.2).

Coste estimado de las tareas		
Paquetes de trabajo	Descripción	Coste
PT1	Estudio teórico	1400 euros
PT2	Familiarización con Mininet	800 euros
PT3	Toma de contacto con Quagga	600 euros
PT4	Caracterización del protocolo PIM-SSM	1200 euros
PT5	Aprendizaje de OpenDayLight	800 euros
PT6	Desarrollo multicast en SDN	1400 euros
PT7	Evaluación y pruebas	600 euros
PT8	Elaboración de una memoria técnica del proyecto	2400 euros
Coste Profesor Doctor		1000 euros
TOTAL :		10.200 euros

Tabla 3.2: Análisis monetario de las tareas.

A continuación, se muestra un gráfico con el coste asociado a cada paquete de trabajo Figura (3.2).

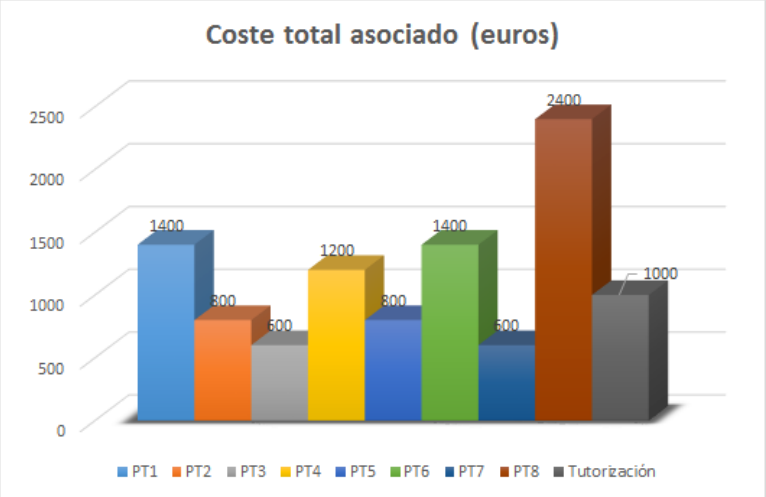


Figura 3.2: Coste asociado a cada tarea.

3.3.2. Recursos Hardware

Dado que el único coste asociado a los recursos hardware es el ordenador personal del alumno, se considera el siguiente coste asociado:

- Ordenador portátil, coste estimado 1.100 euros, con plazo de amortización de 36 meses.

3.3.3. Presupuesto Final

Tras realizar un análisis del coste asociado a cada uno de los recursos principales utilizados en el presente proyecto, se calcula el coste total en su elaboración. Así, en el Tabla (3.3) se puede observar los resultados calculados.

Calculo del coste total	
Recursos Humanos	10.200 euros
Recursos Hardware	1.100 euros
COSTE TOTAL	**11.300 euros**

Tabla 3.3: Coste Total.

3.4. Valoración Final

Como valoración final, tras el desarrollo del proyecto, se puede afirmar que las tareas han seguido el orden previsto en la planificación establecida. Si es cierto que la estimación temporal no ha sido totalmente exacta, ya que debido a la complejidad de algunos de los paquetes de trabajo, como el desarrollo del controlador, se ha necesitado más tiempo, tanto en el apartado de trabajo individual como en el uso de tutorías presenciales.

Aún así, gracias a esta planificación se ha podido desarrollar el proyecto de manera satisfactoria, incluyendo fechas de corte realistas, con el objetivo de conocer el avance en todo momento del desarrollo del proyecto.

Capítulo 4

Desarrollo teórico

En este capítulo se estudian las dos tecnologías claves en el presente proyecto, el tráfico multicast en las redes IP y las redes definidas por software, ambos esenciales para entender el diseño y la resolución del trabajo.

Se profundizara en todos los aspectos basados en el tráfico multicast, desde cómo se procede para establecer una petición hacia una fuente transmisora concreta, hasta cómo se finaliza la transmisión, junto con el estudio de los protocolos multicast que sean de interés para entender futuros capítulos.

Además, se abordará el porqué de la creación de las redes SDN y qué beneficios otorgan, junto con el estudio de las distintas capas que componen las redes definidas por software, explicando detenidamente cómo están formadas y cuáles son sus principales características.

4.1. Multicast en redes IP

Antes de adentrarnos en el desarrollo del trabajo, es conveniente tener clara la diferencia entre los distintos tipos de transmisión de información, esto es: unicast, broadcast y multicast.

IP Unicast se basa en el envío de información a un único receptor, es decir, se envía un paquete de datos por cada cliente. En cambio, **Broadcast** está basado en el envío de datos de un nodo emisor hacia múltiples destinos, estén o no interesados en recibir la información. Desgraciadamente estos dos protocolos potencialmente pueden generar episodios de congestión en la red para todos aquellos servicios que supongan varios receptores. Por este motivo surge **IP Multicast** con el objetivo de mitigar estos episodios, ya que proporciona una forma de enviar información a múltiples receptores disminuyendo el tráfico redundante en los enlaces, Figura (4.1).

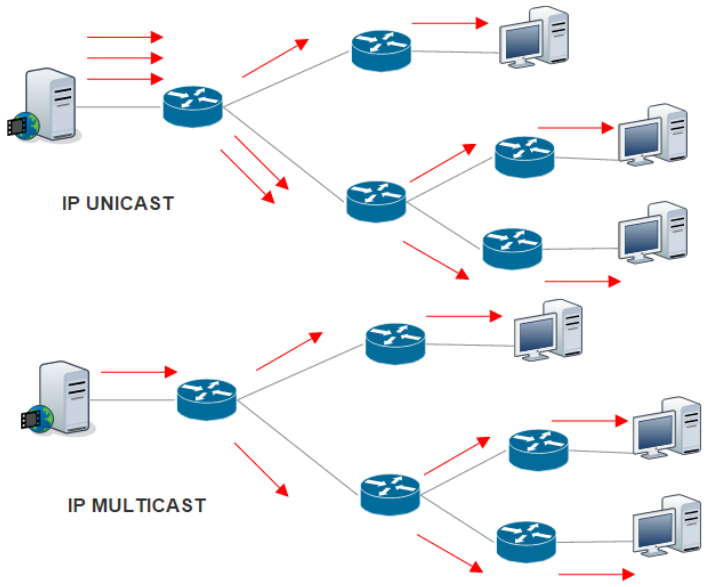


Figura 4.1: Diferencia entre IP Unicast y Multicast.

Se necesitan dos protocolos básicos para realizar IP Multicast. Primero, un protocolo para la asociación y disociación de los usuarios a un grupo multicast del cual los usuarios quieren recibir información; el *router* más cercano habilitado con multicast se encarga de este proceso; además se necesita un protocolo de encaminamiento entre *routers* para crear una árbol de distribución entre la fuente y los receptores, de manera que los encaminadores puedan hacer llegar los paquetes multicast a su destino correcta y eficientemente.

De esta forma, se puede observar, Figura (4.2), que se tendría la red dividida en dos áreas diferenciadas. Una zona donde se realiza la señalización (asociación y disociación a los grupos multicast) -típicamente en la red de acceso- y otra zona en la se produce el encaminamiento -involucrando a *routers* troncales-.

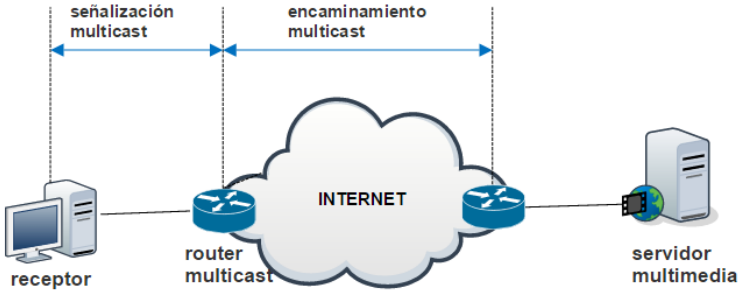


Figura 4.2: Diferenciación de zonas en multicast.

Otro aspecto a tener en cuenta es el uso de direcciones específicas clase D para el envío de datos multicast, éstas comprenden el rango comprendido entre 224.0.0.0 y 239.255.255.255 [38]. El rango de direcciones entre 224.0.0.0 y 224.0.0.255, ambos inclusive, está reservado para el uso de los protocolos de enrutamiento y otros de bajo nivel, tales como protocolos de descubrimiento de *gateways* o la notificación de pertenencia al grupo multicast.

Independientemente del Time to live (TTL), los *routers* multicast no reenviarán este tipo de paquetes. Existen otros rangos restringidos para direcciones privadas en este ámbito como por ejemplo el bloque 232.0.0.0/8 reservado para PIM-SSM.

A continuación, se expone en detalle los diferentes protocolos que tienen relevancia en la transmisión de datos multicast.

4.1.1. Internet Group Management Protocol

El protocolo utilizado para la para la adscripción dinámica de los *host* a grupos multicast se denomina IGMP. Gracias a la información recopilada mediante IGMP los *routers* son capaces de mantener una lista con los grupos multicast a los que están interesados los *hosts* conectados a sus interfaces. Todos los mensajes IGMP tienen TTL=1, por lo tanto solo pueden ser intercambiados por dispositivos conectados entre sí. Actualmente hay tres versiones de IGMP (1,2 y 3); cada versión es una actualización de la anterior. A continuación se describen cronológicamente las versiones.

IGMPv1 (aprobado en 1989) está especificado en el Request for Comments (RFC) 1112 [45]. Esta versión tiene solo dos tipos de mensajes *Membership Query* y *Membership Reply*. Los *routers* con capacidad multicast envían mensajes tipo *Membership Query* para preguntar a los *hosts* en qué grupos están interesados. Los *hosts* responden a este mensaje con un *Membership Reply* indicando a qué grupo quieren unirse; es decir, a qué dirección multicast están interesados. En esta versión de IGMP no existen mensajes

para abandonar el grupo multicast; así, un miembro dejará el grupo cuando no responda a tres mensajes *Membership Query* de forma consecutiva. Esta versión tiene una serie de problemas:

- Como se comento anteriormente, al no existir ningún tipo de mensaje para abandonar el grupo, y la frecuencia de los mensajes tipo *Membership Query* es de aproximadamente 1 minuto, pueden aparecer intervalos de 3 minutos durante los cuales se está mandando información redundante por esa interfaz.
- El valor por defecto de un minuto entre mensajes *Membership Query* no puede ser modificado de forma dinámica.
- Por último, IGMPv1 delega la elección del *Query router* al protocolo de *routing* en cuestión, lo que en algunos casos puede llegar a ser un problema.

IGMPv2 (aprobado en 1997) está especificado en el RFC 2236 [47]. Debido a la problemática de la no existencia de ningún mecanismo explícito para abandonar un grupo, en esta nueva versión se añade un nuevo tipo de mensaje *leave group*; de esta forma se logra que el tráfico innecesario se detenga antes y así no se creen mensajes redundantes.

IGMPv3 (aprobado en 2002) está especificado en el RFC 3376 [41]. IGMPv3 es la versión más reciente y la que utiliza PIM-SSM lo que es de nuestro interés. En esta versión existen dos tipos de mensajes nuevos *Group-source Report* que permite a los receptores especificar de qué emisores desean recibir información y *Group-source Leave* donde se puede indicar qué fuentes se quieren abandonar. Así, no solo se selecciona el grupo, sino también el conjunto de emisores de los cuales queremos recibir datos, bloqueando la información procedente de las otras fuentes.

4.1.2. Árboles de distribución Multicast

Para entender el funcionamiento de la transmisión multicast, es necesario conocer los árboles de distribución [73]. Tal y como se expuso anteriormente, el tráfico unicast se encamina a través de la red a lo largo de una sola trayectoria origen-destino. En cambio, en el caso de multicast, el emisor envía la información a un grupo de usuarios suscritos localizados a lo largo de la red, de esta forma se hace uso de árboles de distribución (o Multicast Distribution Tree (MDT)) para describir el camino que deben tomar los datos multicast a través de la red, permitiendo un uso eficiente de la red. Cabe destacar que estos árboles son dinámicos ya que las fuentes y los participantes varían en el tiempo.

Existen dos tipos de MDT, *Source Trees* donde existe un árbol para cada fuente y *Shared Trees* un mismo árbol compartido para todas las fuentes. A continuación, se explican los dos tipos de arboles detenidamente.

Source Trees

La forma más simple que toma un árbol de distribución en IP multicast es *Source Tree*, formando una topología tipo *Spanning Tree* entre las fuentes y los receptores. Este diseño utiliza el concepto del camino más corto; por eso se llama también SPT. Así, existe para cada origen multicast un árbol que conecta con los receptores. *Source Tree* se suele utilizar para protocolos de *routing* multicast en modo denso, explicados más adelante.

Cada uno de los enrutadores que pertenecen al árbol tienen una tabla de conmutación multicast (Multicast Fowarding Table (MFT)) para cada grupo. Esta tabla consiste en un conjunto de entradas que contienen las interfaces asociadas a la pareja de direcciones IP del grupo y la fuente de ese grupo. De este modo, se utiliza una notación (S,G), siendo S la dirección IP de la fuente y G la dirección IP multicast del grupo.

Este tipo de distribución implica la creación de una ruta óptima entre el origen y los participantes, garantizando un tiempo mínimo de latencia. Para cada fuente activa existe un *Source Tree*; por lo tanto, cada *router* debe mantener una tabla para cada uno de los grupos multicast, incluso distintas entradas en la tabla MFT si hay varias fuentes para un mismo grupo. Esto puede generar un problema de escasez de recursos y de escalabilidad en la red.

En la Figura (4.3) se muestra un ejemplo de *Source Tree*, donde existe una fuente S1 que distribuye información para un Grupo G1 y dos receptores interesados. Así, la entrada de la tabla MFT sería la dupla (S1,G1), si una nueva fuente empezara a transmitir para este grupo, habría que crear otra nueva entrada en la misma tabla.

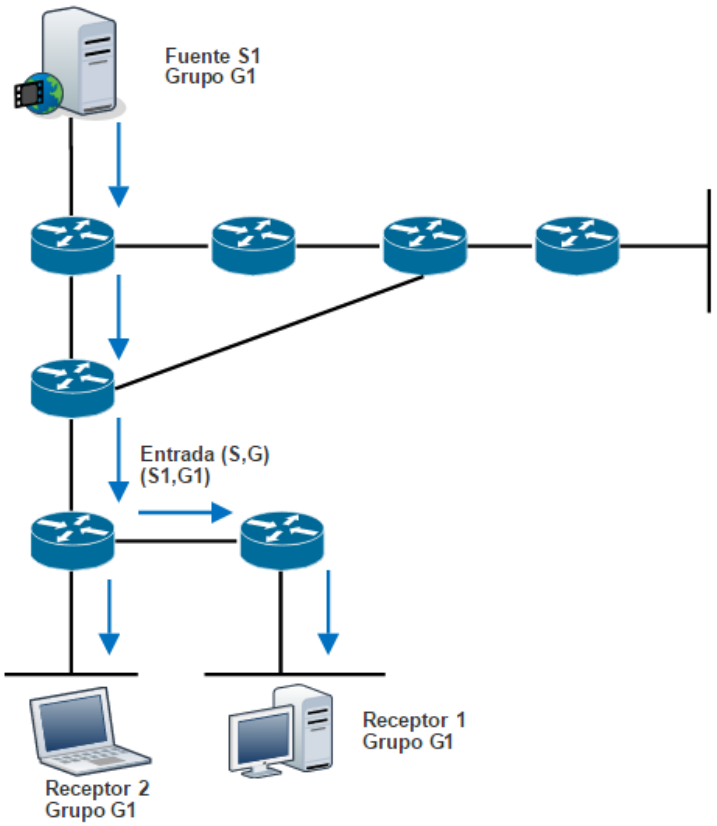


Figura 4.3: Ejemplo de Source Tree.

Shared Trees

La principal diferencia entre este tipo de topología con la estudiada anteriormente es que en este caso, existe una única raíz situada en algún punto de la red en función del protocolo utilizado, denominado Rendevous Point (RP) [73]. Así, las fuentes envían el tráfico al RP establecido y este lo encamina al grupo multicast destino; es decir, se podría decir que es el centro de la comunicación multicast. Además, cabe la posibilidad que existan varios RP para mejorar el rendimiento de la red.

Cuando un usuario quiere unirse un grupo, este envía una solicitud de asociación hacia el *router* al que este conectado, el enrutador será el encargado de hacer llegar la solicitud al RP, el cual retransmite el tráfico que llega desde la fuente e informa al *router* sobre las fuentes activas para ese grupo.

Debido a que todas las fuentes utilizan el mismo *root*, la notación adoptada para este caso es (*,G); siendo G la dirección IP multicast del grupo.

Como todas las fuentes comparten el mismo árbol, cuando se añade una nueva fuente no es necesario crear una nueva entrada en la tabla.

En la Figura (4.4) se muestra un ejemplo de *Shared Tree*.

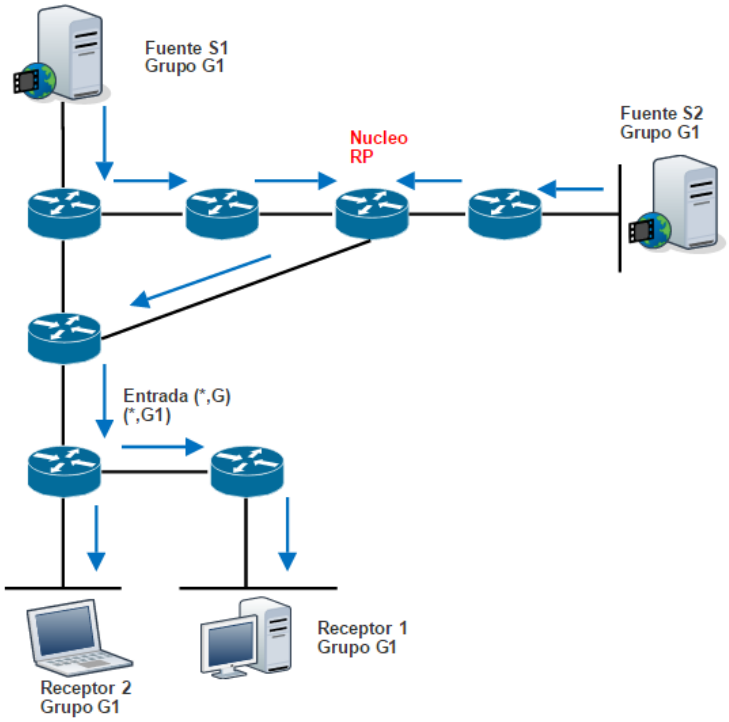


Figura 4.4: Ejemplo de Shared Tree.

Existen dos tipos de *Shared Trees*: **unidireccionales** y **bidireccionales**. En el caso unidireccional, el *Shared Tree* no se puede utilizar para enviar los datos hacia el RP, de manera que los flujos de datos hacia los RP tienen que ser transmitidos a partir de un *Source Tree*. En cambio, si es bidireccional la información puede transmitirse en ambos sentidos, sin necesidad de pasar por el RP; por lo tanto, con esta aproximación se obtiene una mejora en el rendimiento de la red, además de mantener las tablas lo más reducidas posibles.

Si se realiza una comparación entre los dos árboles de distribución estudiados, lo primero que se puede observar a primera vista, es que con el uso de este tipo de árbol se obtiene una solución del problema de escalabilidad en la red, ya que se reducen los recursos necesarios para mantener las tablas que contienen la información de los grupos multicast. Aún así, esto se logra con un aumento de la latencia, ya que hay que enviar toda la información al RP en cuestión, además, el árbol creado no es siempre óptimo para todas

las fuentes/destinos determinados.

4.1.3. Conmutación de tráfico Multicast

Queda claro que en el modelo unicast los *routers* cursan el tráfico a través de la red en un único camino desde la fuente hasta el emisor, por lo tanto cada enrutador toma una decisión de conmutación dependiendo de la dirección destino del paquete y tras consultar la tabla de encaminamiento para obtener la interfaz por la que encaminará el tráfico.

En cambio, como se estudio anteriormente en el caso de multicast, la fuente envía tráfico a un grupo arbitrario de *hosts* representados por una dirección de grupo multicast. Por lo tanto, en este caso no se puede realizar el encaminamiento considerando la dirección destino del paquete, ya que normalmente hay que conmutar el tráfico hacia múltiples interfaces para alcanzar a todos los receptores. Esto hace que el proceso de encaminamiento multicast sea más complejo que el usado para unicast.

A continuación, se estudia el concepto de RPF, el cual constituye el procedimiento básico para la creación de árboles en la mayoría de protocolos multicast.

Reverse Path Fowarding

Cuando un paquete llega a un *router*, este realiza un test RPF sobre él, si hay éxito, el paquete se encamina por la interfaz establecida; si no, este se descarta [73]. El procedimiento que se sigue para aplicar el test RPF ante la llegada de un paquete es el siguiente:

- El *router* examina la dirección origen del paquete para determinar si ha llegado a través de una interfaz que está en el camino de retorno hacia la fuente; es decir, si pertenece al árbol de distribución.
- Si es así, el paquete es reenviado por todas las interfaces pertenecientes a la Outgoing Interface List (OIL); es decir, por aquellas a las que se puede llegar hacia los receptores interesados, sin contar claro está, la interfaz de llegada del paquete.
- Si el proceso falla, se descarta el paquete.

En la Figura (4.5) se muestra un ejemplo de los dos casos mencionados. A la izquierda se puede observar un éxito en el chequeo RPF, donde se comprueba que S1 es una interfaz que pertenece al camino de vuelta de la fuente. En cambio, a la derecha se muestra un fallo, donde se comprueba

que la interfaz para esa dirección es S1, no S0, por lo tanto el paquete es descartado.

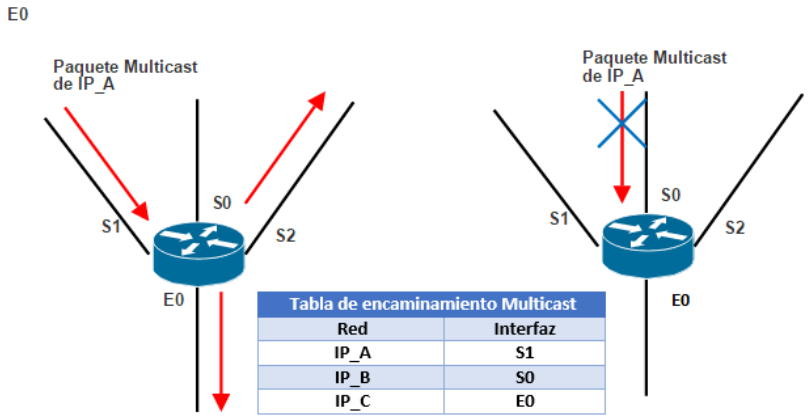


Figura 4.5: Ejemplo de test RPF.

En función del protocolo de *routing* utilizado se determinará de diferentes formas las interfaces que se encuentran en el camino de vuelta hacia la fuente. En algunos casos se usa una tabla de encaminamiento multicast para realizar el test RPF, como en Distance Vector Multicast Routing Protocol (DVMRP). En otros casos, el protocolo multicast utiliza la tabla unicast existente para determinar que interfaz está en el camino de vuelta de la fuente. Un buen ejemplo de su uso, se puede dar en los protocolo del tipo Protocol Independent Multicast (PIM).

4.1.4. Protocolos Multicast

Dentro de los protocolos multicast podemos hacer una división en tres categorías básicas:

- Protocolos modo denso (**DVMRP y PIM-DM**).
- Protocolos modo esparcido (**PIM-SM y Core Based Trees (CBT)**).
- Protocolos de estado de enlace (**Multicast Open Shortest Path First (MOSPF)**).

En este caso el estudio se centra en los protocolos basados en PIM, ya que son los utilizados en el desarrollo del proyecto. La justificación reside en que PIM es el protocolo multicast más popular y más ampliamente desplegado.

Como dijimos anteriormente, a diferencia de otros protocolos de multidifusión como DVMRP o MOSPF, PIM no mantiene una tabla de enrutamiento por separado, sino que basa sus decisiones considerando la tabla unicast para comprobar que el paquete multicast ha llegado por la interfaz correcta. Si la comprobación es exitosa, la interfaz se identifica como interfaz RPF y es almacenada con las entradas (S,G) o (*,G) en la tabla MFT, para con ello evitar que el test RPF de cada paquete, necesite consultar la tabla de enrutamiento. Si la tabla de unicast cambia, entonces la interfaz RPF se actualiza en la tabla MFT para reflejar ese cambio de enrutamiento.

PIM establece adyacencias entre los enrutadores directamente conectados que tengan habilitado el protocolo. Para ello envía mensajes periódicos denominados *Hello* utilizando paquetes IP multicast con dirección destino 224.0.13.0 [73]. Una vez establecidas las adyacencias, PIM utiliza dos métodos para el envío de paquetes multicast:

- **Any Source Multicast (ASM)**: Emplea un modelo muchos-a-muchos donde varias fuentes están enviando tráfico a un grupo multicast.
- **Source Specific Multicast (SSM)**: Emplea un modelo uno-a-muchos donde el receptor se asocia directamente a la fuente del tráfico multicast.

Asimismo, de forma independiente al método utilizado para transmitir el tráfico multicast, existen dos variantes de PIM que son el modo extendido PIM-DM y el PIM-SM. Tanto PIM-DM como PIM-SM pueden funcionar como ASM y SSM.

A continuación, se explican brevemente estas variantes de PIM.

PIM Dense Mode

PIM-DM hace uso de *source tree* debido a que está pensado para redes con multitud de receptores, ya que asume que por cada subred existe al menos un receptor para cada flujo (S,G). Por lo tanto los paquetes multicast son transmitidos a todos los puntos de la red, lo que provoca la llegada de información a nodos que no la han solicitado. Además, todos los *routers* tienen que tener tablas almacenadas con el estado del resto de los enrutadores [37].

- *PIM-DM* utiliza la técnica *flood-and prune* para la construcción de los árboles, ideal para grupos densos. La fuente empieza a transmitir incluyendo todos los enrutadores adyacentes dentro del árbol SPT, así se distribuyen las tramas multicast por todas las interfaces, excepto por la que ha llegado en cada nodo, Figura (4.6).

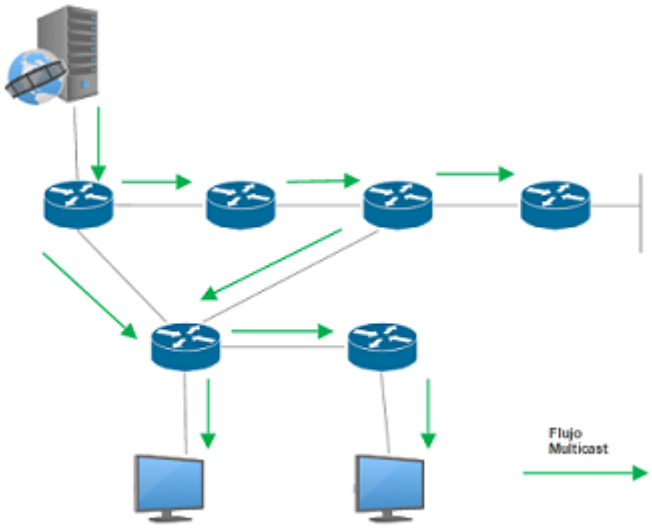


Figura 4.6: Ejemplo PIM-DM

- Aquellos *routers* que no están interesados en el flujo, envían un mensaje tipo *PIM prune* (poda) para indicar que sea eliminado del árbol *SPT*. De esta manera, se eliminarán las ramas del árbol innecesarias, Figura (4.7). Sin embargo, las entradas (S,G) seguirán permaneciendo en las tablas.

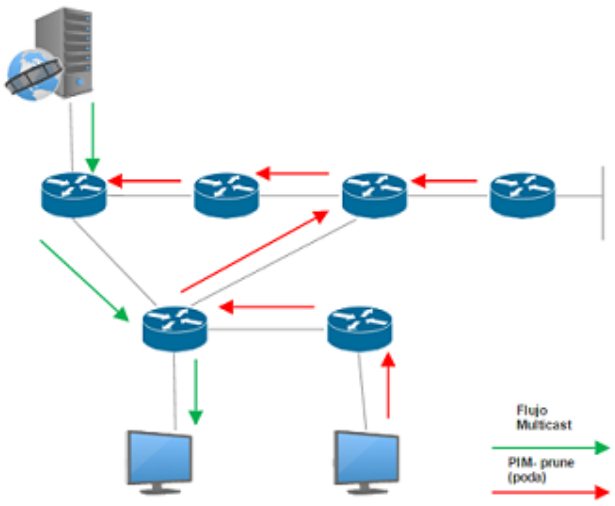


Figura 4.7: Procedimiento PIM-prune

- Así, se crea una topología donde todos los receptores están interesados en el contenido multicast Figura (4.8). Si surge algún nuevo receptor interesado en unirse al grupo, se envía un mensaje *PIM graft* (injerto) al router para ïnjertarün enlace previamente podado. Además, periódicamente los mensajes *prune* expiran y el tráfico multicast vuelve a fluir por la red.

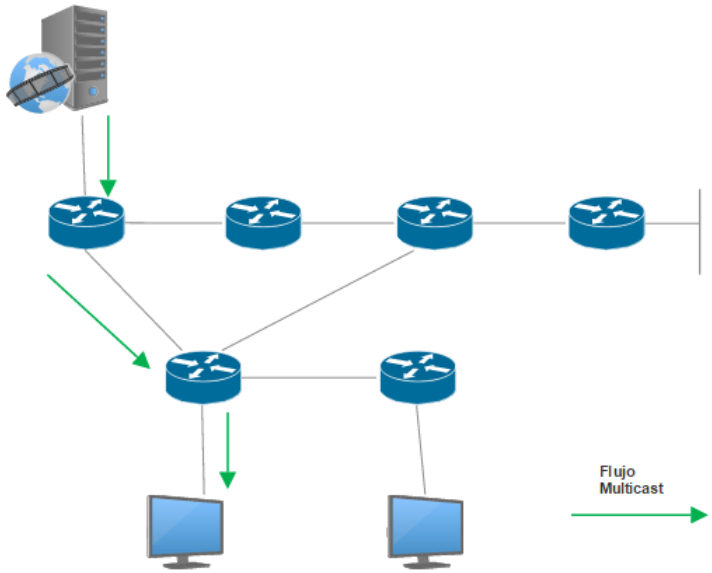


Figura 4.8: Estado final PIM-DM

PIM Sparse Mode

PIM-SM a diferencia de PIM-DM distribuye el tráfico hacia los receptores que están interesados en dicha información. Para ello, hace uso de los mensajes *join* que solicitan la asociación a un grupo multicast de un enrutador. Esta variante de *PIM* permite tanto árboles compartidos como árboles basados en fuente.

PIM-SM establece tres fases para el envío de tráfico multicast [46]:

- **RP-Tree.** Cuando un receptor está interesado en unirse a grupo, envía un mensaje IGMP a su *router* con capacidad multicast, indicando a qué grupo quiere unirse. Entonces, el enrutador envía un mensaje tipo *join* (*,G) a su vecino en la tabla *RPF*, así sucesivamente se irá transmitiendo hasta llegar al RP en cuestión, construyéndose de esta forma el árbol compartido, Figura (4.9). Lógicamente, cada *router*

añade la interfaz por la que recibe el *join* a la lista de interfaces por los que se puede alcanzar a los receptores interesados de un determinado grupo (OIL).

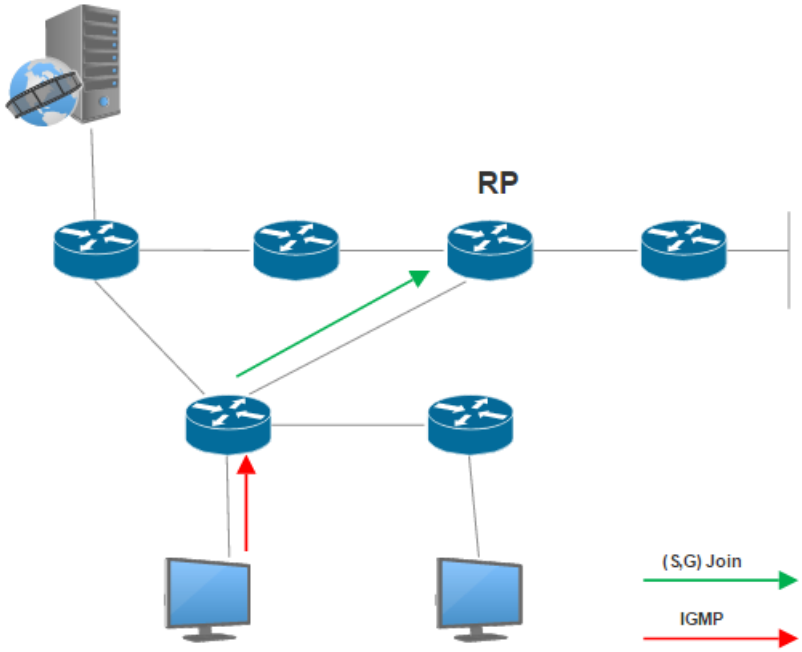


Figura 4.9: Ejemplo de unión a un shared tree.

- **Register-Stop.** Cuando una fuente se activa, esta envía información al *router* con soporte multicast que haya sido seleccionado (el así llamado como enrutador de ingreso). Este *router* le indica al RP la existencia de una fuente a través de mensajes tipo *registrer* vía unicast. Entonces el RP desencapsula los paquetes y los envía a los receptores registrados al grupo. Además, se envía un mensaje *PIM-join* hacia el enrutador de ingreso para crear un *source tree*. Una vez se haya establecido la ruta del router al RP, se dejan de encapsular los paquetes y se empiezan a enviar en multicast nativo. Figura (4.10). Con este procedimiento el registro ya estará hecho.

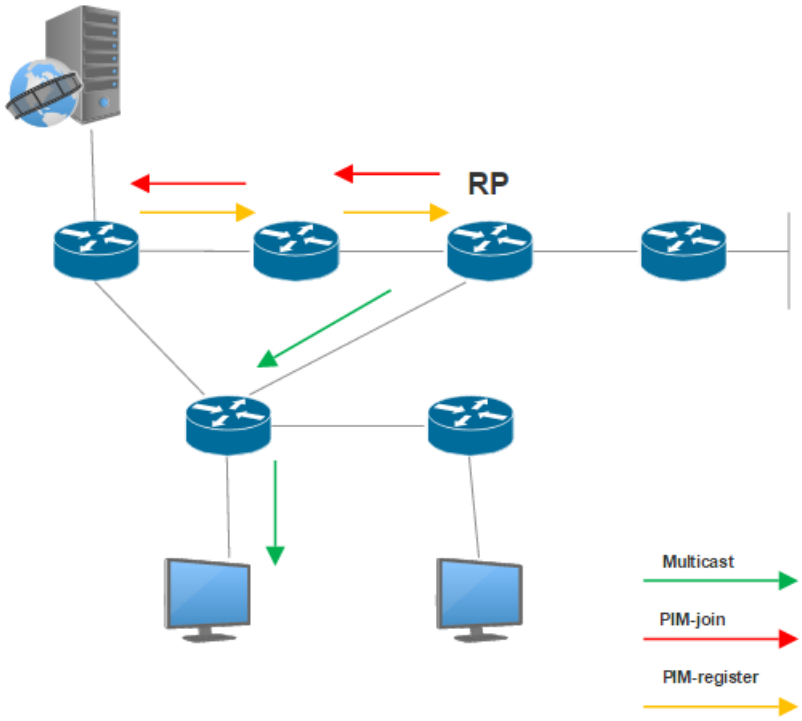


Figura 4.10: Ejemplo de unión de una fuente.

- **Shortest Path Tree.** A pesar de eliminar el encapsulado, realmente no se optimiza completamente el trayecto. Para muchos receptores, la ruta a través del RP puede implicar un desvío significativo cuando se compara con el camino más corto desde la fuente al receptor. Por lo tanto, para reducir la latencia, desde el *router* multicast del receptor(enrutador de egreso) se inicia la construcción de un SPT hasta el enrutador de ingreso. Así, a través de mensajes *PIM-join* se crea el árbol desde el enrutador de egreso. Una vez construido el SPT, los mensajes multicast se enviarán directamente desde la fuente al receptor sin pasar por el RP, Figura (4.11). Para no recibir dos copias de cada paquete (uno a través del nuevo SPT y otro a través del Rendevous Path Tree (RPT)) se realiza una "poda"para eliminar el envío a través del RP, quedando entonces únicamente el camino más corto.

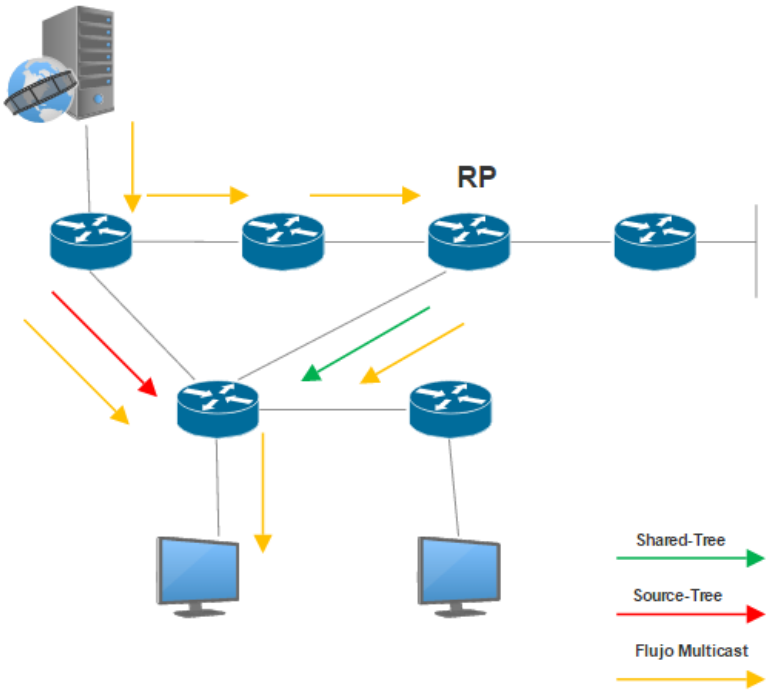


Figura 4.11: Ejemplo PIM-SM.

Como se ha visto este método es más escalable y consume menos recursos de ancho de banda que PIM-DM, por lo que es claramente más eficiente para la transmisión de datos multicast.

PIM-SSM

Source-specific multicast [50] elimina la necesidad de la existencia del RP, pero es necesario que los *routers* de egreso conozcan la dirección IP de la fuente del grupo para la que los clientes están interesados en unirse. Para ello, es necesario el uso de IGMPv3, donde el receptor indica el grupo y la fuente de la que quiere recibir información (S,G). Una vez que el *enrutador* conoce la IP de la fuente, este puede enviar un mensaje (S,G) *Join* directamente al emisor. De esta manera, el tráfico multicast sigue una trayectoria eficiente hacia los usuarios, sin tener que pasar por el RP.

PIM-SSM puede ser utilizando en todo el rango de direcciones multicast, pero la operación solo esta garantizada para las direcciones IP 232.0.0.0/24, lo que le permite al enrutador de egreso identificarlas como SSM. Aún así, PIM-SSM permite la coexistencia con otras variantes de PIM.

Realmente, en un entorno donde existen muchas fuentes que nacen y mueren dinamicamente, por ejemplo en un servicio de videoconferencia, ASM es apropiado. Sin embargo, si se estudia la cantidad de tráfico distribuido a lo largo de la red -como ya se hizo en el Capítulo I-, se puede llegar a la conclusión que el modelo de uno a varias fuentes SSM tiene mayor utilidad. Por lo tanto todas las aplicaciones de multidifusión, como la distribución de canales de televisión a través de Internet pueden ser más rápidas y eficientes usando esta variante que si se opta por la funcionalidad ASM a lo largo de la red.

4.2. Software Defined Network

Tal y como se estudio en el Capítulo I, la aparición de nuevos servicios y aplicaciones, junto con el incremento del número de dispositivos de usuario y servicios en la nube, han provocado que la arquitectura de red actual sea insuficiente para satisfacer las necesidades de este nuevo paradigma de red. Así, surgen las redes definidas por software como un método para solucionar esta problemática sin realizar un cambio radical y significativamente costoso en la red [71].

Las redes definidas por software implican una arquitectura de red emergente, flexible y programable que rompe con la verticalidad presente en las redes actuales. Su estructura permite que el comportamiento de la red sea más flexible y adaptable a las necesidades de cada grupo de usuarios u organizaciones [57] [71]. Además, su diseño centralizado facilita que la información de vital importancia que transcurre por la red sea recolectada y utilizada para mejorar el comportamiento y la eficiencia de esta de una forma dinámica y rápida, aprovechando las ventajas que puede aportar la disposición de una visión de conjunto.

La repercusión más importante es la concesión al gestor de red de un pleno control a través de un único interfaz para todos los dispositivos; es decir, mientras que una aproximación clásica exigía ir dispositivo a dispositivo realizando configuraciones. Incluso para actualizar los sistemas, gracias a las redes definidas por software se elimina esta necesidad ahorrando de esta forma mucho tiempo y dinero, Figura (4.12).

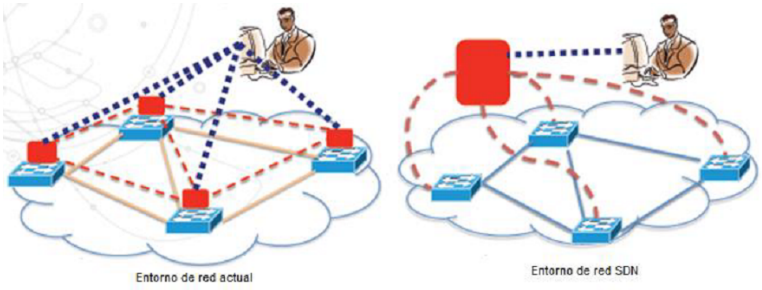


Figura 4.12: red tradicional vs red SDN. [25]

En [10] Juniper propone los seis principios básicos que tienen que tener las redes definidas por software para que sean beneficiosas para el usuario:

- Separar el software de red en cuatro capas: servicios, control, gestión y *fowarding*, optimizando así cada plano en la red.
- Centralizar todos los servicios de las capas anteriores para disminuir y simplificar el diseño de red y los costes de operación.
- Hacer uso de la nube para un desarrollo flexible y escalable, reduciendo el tiempo de servicios y estableciendo una correlación entre el costo y el valor obtenido.
- Creación de una plataforma para las aplicaciones y servicios de red, integrándolas en los sistemas de gestión y por tanto, creando nuevas soluciones de negocio.
- Estandarización de los protocolos para eliminar la dependencia de los distintos proveedores existentes y de esta forma reducir costes aumentando la re-utilización de los sistemas.
- Aplicación de los principios de las redes definidas por software a toda la red y servicios, incluyendo la seguridad, desde *data centers* y campus empresariales hasta las redes móviles usadas por los proveedores de servicios.

4.2.1. Arquitectura.

La principal característica de las redes definidas por software es la separación del plano de control (software) del plano de datos (hardware). Es decir, se puede ver como la existencia de una capa superior a la capa física compuesta por un controlador -que con una visión global de toda la red- es quien se encargará de la gestión de la red.

La arquitectura se basa en tres capas, Figura (4.13).

- Una primera capa compuesta por la infraestructura de red que sería el plano de datos (hardware), el cual estaría formado por el conjunto de *hosts*, *routers* y *switches* de la red encargados de transportar el tráfico hacia su destino. Todos estos elementos serán gestionados por la capa superior.
- La segunda capa, el plano de control estaría formado por el controlador encargado de gestionar la red y controlar el flujo de la capa de datos. Es decir, tendría el control sobre las tablas de encaminamiento de distintos elementos de red. Estas dos capas se comunicaran entre sí con las llamadas *SouthBoud APIs*, las cuales permiten al controlador comunicarse con los elementos de conmutación de red. Existen varias implementaciones de esta API pero debido a la popularidad de una de ellas, en concreto el protocolo OpenFlow, se ha convertido en un estándar *de facto*.
- Finalmente, se dispone de una tercera capa de aplicación con una abstracción superior a las demás capas. Está formada por todos aquellos servicios y aplicaciones de usuario cuya misión es comunicar al controlador sus necesidades, a través de las *NorthBoud APIs*.

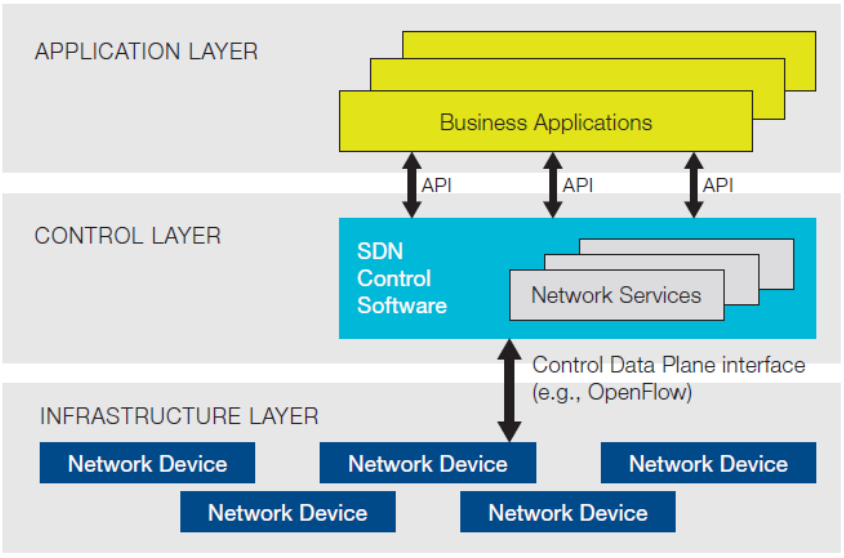


Figura 4.13: Arquitectura de las redes definidas por software. [48]

4.2.2. Protocolo OpenFlow.

OpenFlow [24] es el primer protocolo libre y estandarizado para establecer la comunicación entre la capa de datos y la capa de control. OpenFlow

facilita el intercambio de mensajes entre el controlador y los *switches*. Con OpenFlow los gestores de la red pueden acceder a las tablas de flujos permitiendo controlar el tráfico que circula a través de la modificación de estas tablas.

La calidad y cantidad de políticas que se pueden gestionar en este tipo de sistemas y el aumento de la rapidez para actuar frente cambios en la red, hace que esta aproximación mejore considerablemente la dinámica en la gestión de la red, permitiendo una reducción en los costes considerables [62] [66] [74].

OpenFlow se basa en tres aspectos [48]:

- El controlador, que será el cerebro de la red, el cual dialogará con todos los *switches* para transmitirles la información
- Las tablas de flujos instaladas en cada uno de los *switches* que indicarán qué hacer con el tráfico entrante.
- Los dispositivos de red (*switches*) con capacidad OpenFlow.

4.2.3. Switch OpenFlow.

Se ha estudiado que uno de los factores que han provocado el éxito de las redes SDN es la eliminación de la dependencia con los distintos fabricantes de elementos de red. Para logar esto, el protocolo OpenFlow explota el hecho de que todos los *switches* y *routers* de última generación contienen tablas de flujo. Aunque es cierto que cada sistema dependiendo del proveedor tiene características distintas, se han identificado un conjunto de ellas que son comunes en todos los elementos.

Un *switch* OpenFlow tiene que constar de al menos tres partes [61], Figura (4.14):

- Una **tabla de flujo** con una acción asociada a cada entrada, de tal manera que el *switch* tenga la información necesaria para procesar todo tipo de datos.
- Un **canal seguro** que conecta el *switch* con el controlador remoto, permitiendo la conexión entre ambos.
- El uso del **Protocolo OpenFlow** que provee al controlador de una forma libre y estandarizada para comunicarse con el *switch*.

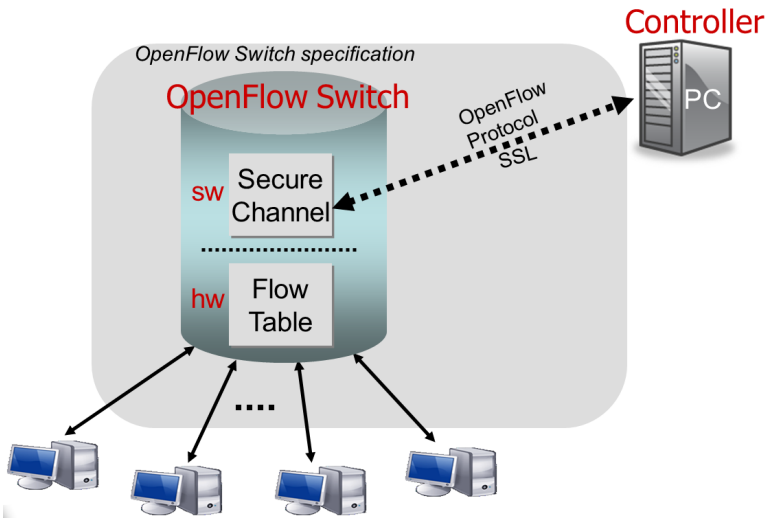


Figura 4.14: Switch OpenFlow. [61]

4.2.4. Tablas de flujo

Las tablas de flujo están formadas por un conjunto de entradas, identificadas por distintos campos que permiten realizar un *match* con los paquetes entrantes, con el objetivo de aplicar una serie de instrucciones a los paquetes coincidentes [64]. Los principales campos de la tabla de flujo son los siguientes:

- **Match Field.** Se utiliza para comprobar coincidencias de un paquete con la entrada de la tabla, contiene el puerto y la cabecera, opcionalmente también puede contener un metadato.
- **Counters.** Campo que se actualiza cuando hay una coincidencia.
- **Instructions.** Indica una modificación en las acciones.
- **Timeout.** Tiempo máximo antes de que el *switch* descarte el flujo.
- **Priority.** Prioridad de flujo para dar preferencia a una entrada cuando existen varias coincidencias de entrada.
- **Cookie.** Campo para filtrar estadísticas del tráfico.

Flow Table					
Match Field	Priority	Counters	Timeout	Cookie	Instructions

Tabla 4.1: Campos Tabla de flujos OpenFlow.

Los switches OpenFlow pueden contener varias tablas de flujo, las cuales contienen múltiples entradas de flujo. En este contexto tenemos que hablar del proceso *Pipeline* OpenFlow, mostrado en la Figura (4.15), el cual define como interactúan los paquetes con esas tablas de flujo. Las distintas tablas de flujo están numeradas, de este modo el proceso comienza siempre por la primera tabla [64].

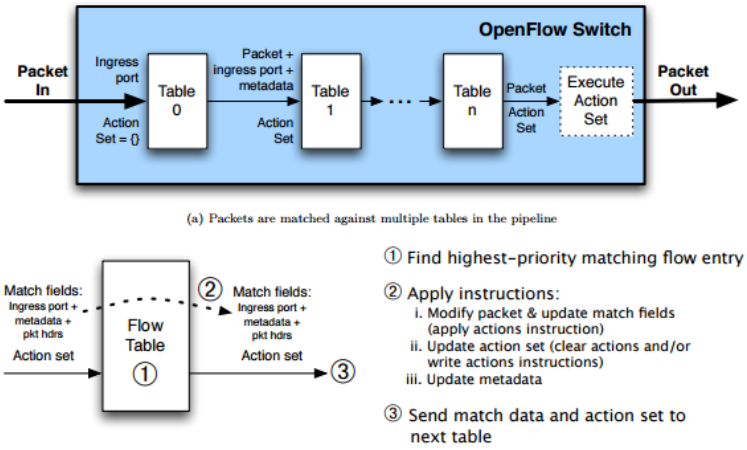


Figura 4.15: Proceso Pipeline. [64]

Si el paquete de entrada no encuentra una coincidencia en la primera tabla pasaría a la siguiente, en cambio si el paquete de entrada obtiene una coincidencia, se ejecuta la acción correspondiente. Esta instrucción puede redireccionar el paquete hacia otra tabla de flujo o directamente indicar el puerto hacia donde conmutar el paquete. En el caso de que no se encuentre coincidencia en ninguna de las tablas, se produce lo que se llama un *table miss*. En este caso se enviará el paquete al controlador o se descartará.

Como se ha comentado, cada entrada de flujo contiene un conjunto de instrucciones que son ejecutadas cuando un paquete obtiene una coincidencia en la tabla de flujos, Figura (4.16). Estas instrucciones pueden provocar cambios en el paquete, o en el proceso de *pipeline*. Las tres acciones más comunes que soportan todos los *switches* OpenFlow son las siguientes:

- Enviar el flujo de datos hacia un puerto o puertos específicos; es decir, permitir el enrutamiento de los datos a lo largo de la red.
- Encapsular y enviar los paquetes hacia el controlador, de este modo el controlador puede decidir qué se debe hacer con el flujo de datos. El canal establecido hacia el *controller* suele ser a través de un canal seguro Secure Sockets Layer (SSL).

- Descartar los paquetes de un flujo, para eliminar la posibilidad de recibir algún tipo de ataque tipo Denial of Service (DoS) o incluso como un procedimiento para reducir la congestión en la red.

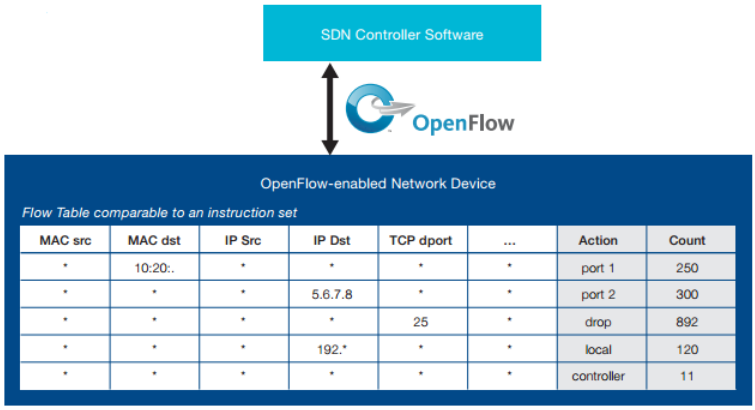


Figura 4.16: Tabla de flujo Openflow. [48]

Entonces, se puede decir que existe una regla que define el flujo a través del campo *Match Field* compuesto por las direcciones MAC, direcciones IPs, puerto origen o destino de los flujos entrantes. A partir de esta clasificación, se procederá a realizar la acción pertinente definida en la tabla. Finalmente, se recogerán ciertas estadísticas en el campo *counters*, ya sean el número de paquetes, bytes enviados, tiempo transcurrido, etc.

4.2.5. Matching OpenFlow.

En este apartado, se va a explicar qué curre dentro del *switch* o qué procedimientos sigue una vez recibe un paquete. Una vez se recibe un paquete en un *switch* OpenFlow, se activa el proceso mostrado en la Figura (4.17) [64]. El *switch* comienza realizando una búsqueda en la primera tabla de flujo en busca de una coincidencia y en base a esto, comenzará el proceso de *pipeline* explicado anteriormente, llevando a cabo la consulta en otras tablas de flujo en función de las reglas establecidas.

Primero, se extrae la cabecera de los paquetes que contiene información relevante, como por ejemplo la dirección IP origen y destino, las direcciones Media Access Control (MAC), para establecer una coincidencia en las tablas de flujo a partir del campo *Match Field*.

Una vez se encuentra una coincidencia, se comprueba que la entrada tiene la mayor prioridad dentro de las todas las posibles; si es así se actualizan los *counters* y se ejecutan las acciones que se indiquen en la tabla. En el caso de

que la acción establezca el re-direccionamiento hacia otra tabla, se realizará el mismo proceso explicado anteriormente.

Existe la posibilidad de que no exista ninguna coincidencia para el flujo de datos entrante, en estos casos se procederá a descartar los paquetes o incluso se enviarán al controlador para que este decida qué acciones deben de tomarse.

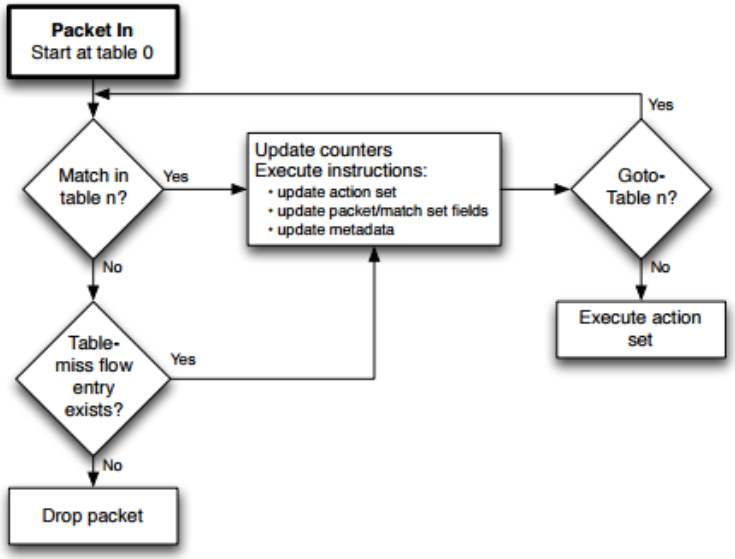


Figura 4.17: Diagrama de estados openflow. [64]

4.2.6. Mensajes OpenFlow.

OpenFlow define una serie de mensajes que son intercambiados entre el controlador y un *switch* OpenFlow para gestionar la red y hacer posible la comunicación entre ambos. La conexión suele ser establecida por el *switch* sobre SSL en el puerto 6634 para proveer de seguridad los distintos mensajes intercambiados.

Los diferentes tipos de mensajes que utiliza el protocolo OpenFlow para controlar la comunicación entre el controlador y los diferentes *switches* pertenecientes a la red se pueden dividir en tres grupos ([64] [69]):

- **Mensajes desde el controlador hacia el switch:** Dentro de este grupo están todos aquellos mensajes que son iniciados desde el controlador y en algunos casos requieren respuesta del *switch*, pertenecen a este grupo aquellos que modifican las tablas de flujo de los switches.
- **Mensajes asíncronos:** Estos mensajes están caracterizados por no

necesitar la demanda del controlador.

- **Mensajes simétricos:** No necesitan la solicitud de ninguna de las partes, suelen ser del tipo control de la conexión o comprobación de conectividad.

A continuación, se van a presentar tres tablas con los mensajes más importantes de los grupos presentados anteriormente.

Mensajes controlador-switch	
Mensaje	Descripción
Flow-Mod	Modifica, añade y elimina las entradas de las tablas de flujo del *switch*.
Read-State	Colecta estadísticas de tráfico o configuraciones del *switch*.
Packet-out	Redirige un paquete hacia un puerto especifico del *switch*.
Features	Solicita las capacidades del *switch*.
Configuration	Establece algunos parámetros de configuración.

Tabla 4.2: Mensajes Openflow desde el controlador hacia el *switch*.

Mensajes asíncronos	
Mensaje	Descripción
Packet-in	El *switch* envía un paquete hacia el controlador.
Flow-Removed	Informa sobre la eliminación de una entrada de la tabla de flujos.
Port-Status	Informa al controlador del cambio de un puerto.
Flow-monitor	Indica al controlador un cambio en la tabla de flujos.

Tabla 4.3: Mensajes Openflow asíncronos.

Mensajes simétricos	
Mensaje	Descripción
Error	Usado para indicar un problema en la conexión entre ambos.
Hello	Mensajes intercambiados durante el establecimiento de la conexión.
Echo request/reply	Mensajes para comprobar la conectividad entre ambos.

Tabla 4.4: Mensajes Openflow simétricos.

4.2.7. Plano de control

El plano de control es el encargado de enviar al plano de datos, a través del controlador, las instrucciones pertinentes para satisfacer las necesidades de la red y de las aplicaciones situadas en la capa superior. El controlador es el elemento principal de las redes definidas por software, se considera como el sistema operativo de la misma, a través del protocolo OpenFlow es capaz de comunicarse con los distintos dispositivos de la red de la misma manera que los elementos de la red se comunican con él pidiendo instrucciones cuando no conoce qué hacer con un flujo determinado.

Tal como sugieren en [57] (Figura (4.18)) las funcionalidades principales que necesita un controlador son las siguientes:

- **Shortest path fowarding.** Uso de la información recolectada de los *switches* para crear rutas eficientes.
- **Notification manager.** Procesar y recibir alertas y eventos.
- **Security mechanisms.** Proveer seguridad entre aplicaciones y servicios.
- **Topology manager.** Crear y mantener las topologías en función de la dinámica de la red.
- **Statistics manager.** Recopilación de datos sobre el tráfico que transcurre por la red.
- **Device manager.** Configuración de parámetros y manejo de las tablas de flujo.

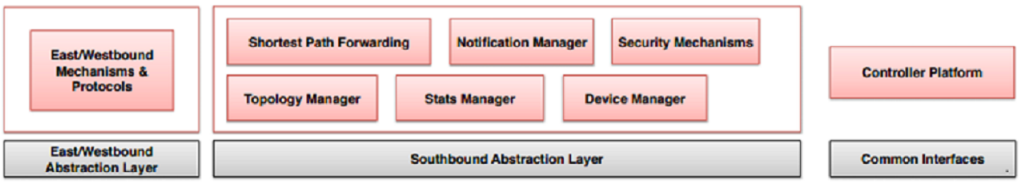


Figura 4.18: Funciones controlador. [57]

De esta forma, se puede ver al controlador como un sistema operativo de red acNOS [69] y como tal, provee servicios esenciales, como por ejemplo la aplicación de interfaces de programación(APIs) y la abstracción de los elementos de la capa inferior. Estas funciones permiten a los desarrolladores definir políticas de red y gestionar las redes sin tener un gran conocimiento sobre las características de cada uno de los elementos de la red.

Existen varias soluciones comerciales y *open source* como implementaciones del controlador SDN. En la Figura (4.19) se hace una comparación de las distintas iniciativas.

	Beacon	Floodlight	NOX	POX	Trema	Ryu	ODL
Soporte OpenFlow	OF v1.0	OF v1.0	OF v1.0	OF v1.0	OF v1.3	OF v1.0, v1.2, v1.3 y extensiones Nicira	OF v1.0
Virtualización	Mininet y Open vSwitch	Mininet y Open vSwitch	Mininet y Open vSwitch	Mininet y Open vSwitch	Construcción de una herramienta virtual de simulación	Mininet y Open vSwitch	Mininet y Open vSwitch
Lenguaje de desarrollo	Java	Java	C++	Python	Rudy/C	Python	Java
Provee REST API	No	Si	No	No	Si (Básica)	Si (Básica)	Si
Interfaz Gráfica	Web	Web	Python+, QT4	Python+, QT4, Web	No	Web	Web
Soporte de plataformas	Linux, Mac OS, Windows y Android para moviles	Linux, Mac OS, Windows	Linux	Linux, Mac OS, Windows	Linux	Linux	Linux, Mac OS, Windows
Soporte de OpenStack	No	Si	No	No	Si	Si	Si
Multiprocesos	Si	Si	Si	No	Si	No	Si
Código Abierto	Si	Si	Si	Si	Si	Si	Si
Tiempo en el mercado	4 años	2 años	6 años	1 años	2 años	1 años	5 meses
Documentación	Buena	Buena	Media	Pobre	Media	Media	Media

Figura 4.19: Controladores OpenFlow. [42]

En concreto, el proyecto se basará en Opendaylight para la implementación del controlador. Esta elección principalmente se debe a que su código

base esta escrito en Java, un lenguaje de programación estudiado en el desarrollo de la del grado, sin olvidar la gran documentación existente comparada con las otras alternativas.

Además, a diferencia de los otros controladores de red, OpenDaylight permite el uso de protocolos de control distintos a OpenFlow. Por todas estas características OpenDaylight está siendo apoyado a nivel económico y de desarrollo por grandes empresas como DELL, Cisco, HP, IBM, entre otras.

4.2.8. Importancia de las redes definidas por software.

Tal y como se comentó en los primeros capítulos, una de las soluciones propuestas para abordar los problemas actuales son las redes definidas por software. SDN está reemplazando al modelo tradicional de red, rompiendo la integración vertical de las redes vigentes, proveyendo a las nuevas implementaciones de un grado superior de flexibilidad y personalización, para soportar las necesidades presentes.

A medida que las redes crecen se incrementa la dificultad en su mantenimiento y administración. La solución vigente es incrementar los recursos IT, es decir, al final se necesita más mano de obra para configurar y mantener los nuevos equipos. En cambio, SDN permite la actualización y configuración de la red gracias al control y visión centralizado de la red, como así se muestra en la Figura (4.20) [69].

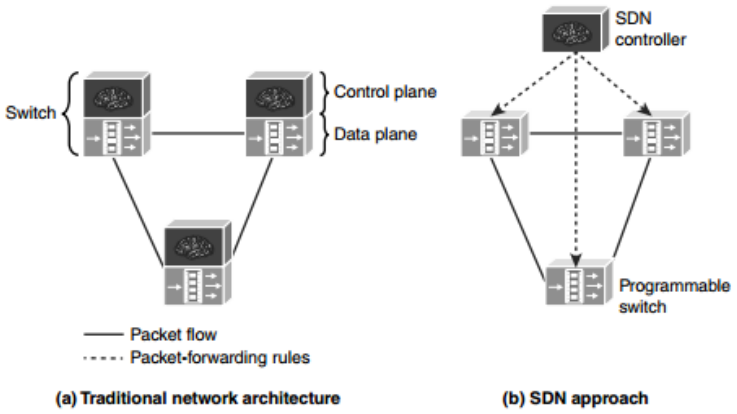


Figura 4.20: Comparación entre redes tradicionales y SDN. [69]

Las redes Definidas por Software además ofrecen un conjunto de ventajas que las convierten en una opción muy atractiva [48]:

- La gestión de la red esta basada en un controlador central que tiene una visión global de la red, facilitando así el mantenimiento y mejorando eficientemente la congestión de la red.
- Están basadas en protocolos abiertos y libre de proveedores, de esta manera las redes SDN permiten el uso de cualquier dispositivo habilitado para su uso, aunque sea de distintos fabricantes, solucionando así la problemática existente de estar siempre a merced de los distintos fabricantes y protocolos propios.
- Permiten a los gestores de la red programar en tiempo real para satisfacer necesidades específicas y adaptar el comportamiento de la red en cada momento
- SDN acelera la innovación empresarial permitiendo la posibilidad de crear nuevas aplicaciones mediante la virtualización de la arquitectura de red, introduciendo nuevos servicios y capacidades en cuestión de horas.
- Añaden la posibilidad de reducir el CapEx limitando la necesidad de renovar los sistemas mediante la reutilización de los mismos.

Para comprobar a nivel experimental los resultados, unos de los numerosos estudios proporcionados por CISCO mantiene que se producen hasta un 40 % de beneficios y una reducción asombrosa en el mantenimiento y gestión de red por el uso de su propuesta de SDN ACI (Figura (4.21)).

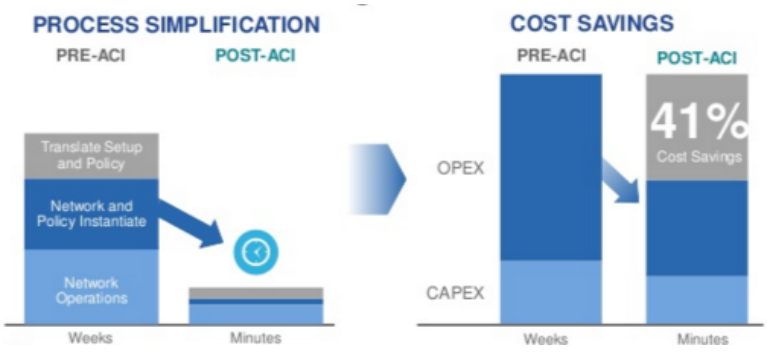


Figura 4.21: Beneficios redes SDN. [5]

Tras comprender todos los conceptos relacionados con SDN y como se estudió en el Capítulo I introductorio, las redes definidas por software se van y se están convirtiendo en un aspecto fundamental para el *networking* actual, y serán una pieza fundamental para el *networking* futuro.

Capítulo 5

Herramientas utilizadas

En este capítulo se van a estudiar las distintas herramientas utilizadas para el correcto desarrollo del proyecto. Además, se expondrá algunos ejemplos simples para la comprensión futura de casos más complicados en capítulos posteriores del proyecto.

En concreto, se estudiará Mininet: un emulador de red que permite crear topologías de red para probar los protocolos multicast. Además, con el fin de caracterizar un escenario sin la tecnología SDN, se tiene que hacer uso de Quagga, una *suite* de software de enrutamiento que provee a los *switches* de capacidad multicast. Finalmente, se estudiará el controlador OpenDayLight, elegido para gestionar la red, se comprobará su funcionamiento y sus peculiaridades más relevantes a la hora de programar sobre él.

5.1. Mininet

Mininet es un emulador de red capaz de crear redes basadas en estaciones finales, *switches*, *routers* en un solo *core* de linux. Se utiliza una virtualización muy ligera para hacer que un único sistema emule una red completa [58].

Los *host*, *switches* y *routers* de Mininet se comportan como dispositivos reales, se pueden enviar datos a través de ellos, ejecutar programas instalados en el sistema, comprobar sus características, la única diferencia es que están creados a base de software y no de hardware.

De esta forma, Mininet permite crear topologías muy complejas de una forma sencilla, en las que se puedan realizar distintas pruebas para caracterizar distintos diseños de red a través de su Call Level Interface (CLI), sin tener que hacer implementaciones en hardware.

Además, debido a sus características se convierte en un entorno adecuado

para las redes definidas por software, ya que permite la creación y uso de controladores en las topologías diseñadas y por tanto, la interacción a través del protocolo OpenFlow entre los elementos de la red. En el Apéndice A.1 se indican las distintas vías para realizar la instalación del *software*.

5.1.1. Ventajas y limitaciones de Mininet

Tal y como se ha visto, Mininet contiene un gran número de ventajas para el usuario, a continuación se enumeran algunas de ellas [58]:

- Es muy rápido, crear una red solo lleva unos segundos, por lo tanto se puede re-editar las topologías sin perder mucho tiempo.
- Se pueden crear topologías personalizadas de todo tipo, además Mininet ofrece un conjunto de diseños pre-diseñados para facilitar más aún la experiencia al usuario.
- Se pueden ejecutar programas reales en los *hosts* finales, ya sea un simple wireshark hasta servidores multimedia.
- Mininet es proyecto de código abierto, con la posibilidad de examinar su código fuente en [1], a la hora de aportar ayuda para solucionar distintos *bugs* o problemas.
- Proporciona una Application Programming Interface (API) de Python para facilitar la creación de topologías personalizadas.
- Los *switches* de Mininet son programables utilizando el protocolo OpenFlow, así se permite la personalización del reenvío de paquetes y un fácil diseño de redes SDN.

Como todas las herramientas Mininet también contiene un conjunto de limitaciones bien identificadas:

- Ejecutar el software en un único sistema tiene sus ventajas, pero también limitaciones en los recursos permitidos, en función de las capacidades y limitaciones del equipo disponible, así hay que realizar pruebas para comprobar cuál es el balance adecuado de *hosts-switches* que permita al sistema funcionar adecuadamente.
- mininet no tiene un controlador por defecto, si se quieren realizar cambios en los flujos, hay que desarrollar el propio controlador.
- Mininet usa un único *kernel* para todos los *hosts* virtuales, de este modo, no se pueden ejecutar programas que dependan de BSD, Windows u otros sistemas operativos.

- Todos los *hosts* comparten el mismo sistema de archivos y PID, esto significa que hay que tener cuidado a la hora de ejecutar procesos (demonios) que creen archivos en /etc y de no eliminar algunos procesos que pertenezcan a varios sistemas.

Se ha comprobado que Mininet tiene ciertas ventajas y desventajas, pero a la hora de su elección como software principal hay que ver que otras alternativas tenemos. Así, si se piensa en el uso de las máquinas virtuales, Mininet soluciona el problema de la escalabilidad, permitiendo la creación de muchos sistemas de manera sencilla. Además, como última opción está el uso de hardware, aunque los resultados sean más fiables comparados con la emulación, debido a su coste elevado y la lentitud en el diseño, hacen de Mininet en una alternativa a considerar.

Finalmente, es conveniente comparar Mininet con algunos de sus competidores. En concreto existen tres software similares a Mininet.

- En primer lugar, Estinet [72] que al igual que mininet es un emulador de redes SDN con soporte para OpenFlow, su principal ventaja es que posee un motor gráfico superior al de mininet, en el se puede ver el comportamiento de la red de forma gráfica. Lamentablemente no es de código abierto lo que es un problema para nuestro desarrollo.
- Otra alternativa es STS [31]: otro simulador que facilita la creación de diseños de red, con la posibilidad de solucionar y localizar problemas concretos.
- Una última alternativa es Ns-3 [15] un simulador de red que permite simular elementos OpenFlow, pero con menos versatilidad y configuración que mininet.

Tras este análisis, debido a las facilidades de Mininet y el rápido entrenamiento producido gracias a la amplia documentación del software, es más beneficioso su uso comparándolo con las otras alternativas, ya que existe una infinidad de proyectos y *papers* sobre este software, lo que facilita el aprendizaje y la puesta a punto a la hora de estudiar el sistema.

5.1.2. Topologías básicas

Mininet facilita a los nuevos usuarios el estudio y aprendizaje de su software, por ello tiene varios comandos que nos ayudan en la creación de topologías simples sin tener mucho conocimiento sobre la herramienta. De esta manera, mininet permite la creación de una topología mínima solo usando el comando básico mostrado en la Figura (5.1).

```
root@carlos-VirtualBox:/home/carlos/mininet# sudo mn
```

Figura 5.1: Comando básico en mininet para una topología mínima.

Desde el interfaz de comandos (Figura (5.2)) se puede apreciar cómo se van añadiendo los distintos elementos de red establecidos, junto con el inicio del controlador. En concreto, este diseño ejemplo está compuesto por un *switch* OpenFlow conectado a dos *hosts* y un controlador OpenFlow.

```
root@carlos-VirtualBox:/home/carlos/mininet# sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

Figura 5.2: Creación topología mínima.

Con el uso del comando *mn* se pueden crear varias topologías preestablecidas por defecto, solo tenemos que añadir el comando $--topo$ junto con diferentes parámetros para obtener un diseño previamente establecido por mininet que queramos diseñar. Es cierto que esto supone grandes facilidades para alguien nuevo en esta tecnología, aunque también esto limita a la hora de querer hacer un diseño personalizado.

Así, a continuación se puede observar una tabla con algunas de las posibles combinaciones de topologías permitidas en Mininet a través de la opción $--topo$ y *mn* (Tabla (5.1)).

Opciones en la creación de topologías	
Comando	Descripción
- -topo=minimal	Crea una topología mínima formada por 2 *hosts*, 1 *switch* y 1 controlador.
- -topo=single,n	Crea una topología formada por 1 switch, 1 controlador y "n" *hosts* conectados al *switch*.
- -topo=reverse,n	Similar a la opción single, pero el orden de conexión está revertido.
- -topo=linear,n	Crea una topología formada por un controlador, "n" *switches* y un *host* conectado a cada *switch*.
- -topo= tree, depht = N, fanaut = M	Crea una topología tipo árbol, siendo N el número de niveles del árbol y M el número de *hosts* por *switch*.

Tabla 5.1: Comandos para crear topologías por defecto.

Para ver experimentalmente cómo funciona realmente la opción *topo*, a continuación en las Figuras (5.3) y (5.4) se puede ver la creación de una topología tipo árbol de dos niveles ($M = 2$), con dos *hosts* por *switch* ($N = 2$).

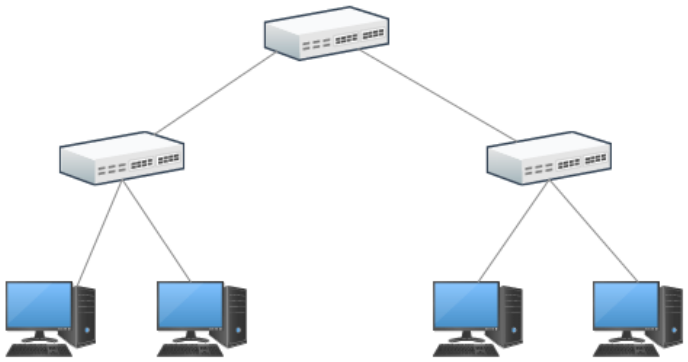


Figura 5.3: Diseño de árbol con 2 niveles.

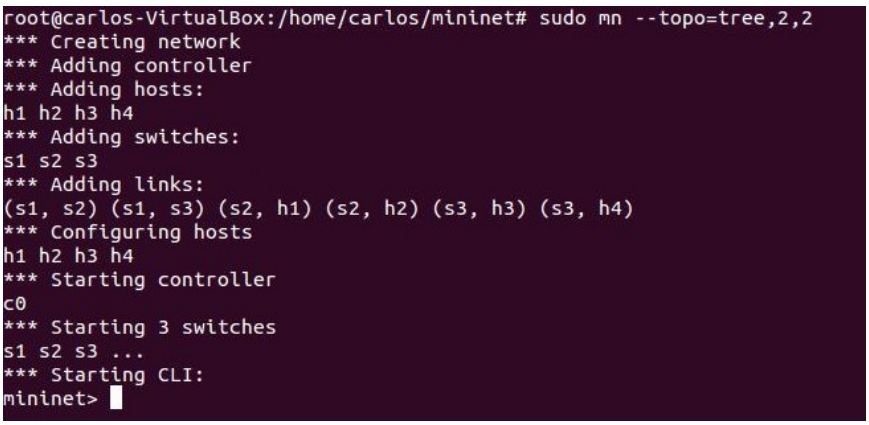


Figura 5.4: Creación de topología en árbol con 2 niveles.

Además de estas opciones para crear topologías por defecto, existen otros comandos que pueden personalizar un poco más los diseños implementados. Como por ejemplo, elegir el tipo de controlador que se va a utilizar, realizar test tras crear las distintas topologías, asignar la MAC automáticamente o incluso cargar topologías personalizadas sobre python. En la Tabla (5.2) se muestran algunas de las alternativas más significativas.

Opciones en la creación de topologías	
Comando	Descripción
- -controller = [default, remote, ryu, nox...]	Especifica el controlador que se va a utilizar en el diseño
- -switch = [default, ovs, user, ovsk...]	Crea un *switch* con las características especificadas
- -mac	Asigna una dirección MAC a todos los *hosts*.
- -test = [pingall, cli, iperf, build...]	Realiza un test tras cargar la topología
- -custom	Lee y carga un *script* de una topología mininet en python.
-h help	muestra la ayuda del comando *mn*.

Tabla 5.2: Opciones en la creación de topologías por defecto.

5.1.3. Comandos básicos

Tras estudiar cómo se crean topologías básicas a través del uso de diseños preestablecidos por el software, se comentan los comandos básicos para ver las características de la topología implementada o las opciones disponibles

dentro del CLI de mininet, como por ejemplo comprobar la conectividad o visualizar los dispositivos de nuestra red.

Para comprobar el funcionamiento de los distintos comandos, consideraremos el diseño del apartado anterior; es decir, un topología de árbol de 2 niveles.

Primero existe la opción *nodes*, a través del uso de este comando se muestran los distintos elementos que forman parte de la red (Figura (5.5)). Como se puede observar en la imagen aparecen los cuatro *hosts* junto con los tres *switches* y el controlador.

```
mininet> nodes
available nodes are:
c0 h1 h2 h3 h4 s1 s2 s3
```

Figura 5.5: Comando *nodes* Mininet.

Quizá, el comando más básico pero el más importante es el *help*, a través de este comando se puede observar toda la ayuda proporcionada por Mininet, junto con una lista de comandos disponibles (Figura (5.6)).

```
mininet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes        pingpair      py      swit
ch
dpctl  help   link      noecho       pingpairfull  quit    time

dump   intfs  links     pingall      ports         sh      x

exit   iperf  net       pingallfull  px            source  xter
m

You may also send a command to a node using:
  <node> command {args}
For example:
  mininet> h1 ifconfig

The interpreter automatically substitutes IP addresses
for node names when a node is the first arg, so commands
like
  mininet> h2 ping h3
should work.

Some character-oriented interactive commands require
noecho:
  mininet> noecho h2 vi foo.py
However, starting up an xterm/gterm is generally better:
  mininet> xterm h2
```

Figura 5.6: Comando *help* Mininet.

El comando *net* muestra información sobre los *links* existentes en la topología implementada (Figura (5.7)). Tal y como aparece en la imagen se pueden observar los enlaces que hay entre todos los elementos del sistema.

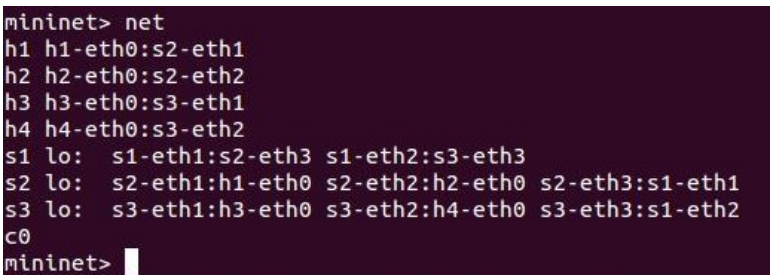


Figura 5.7: Comando *net* mininet.

A la hora de realizar comprobaciones en el diseño, como por ejemplo, si hay conexión entre cada uno de ellos, se puede realizar un test que transmite un mensaje ICMP a cada uno de los *hosts* usando el comando *pingall* (Figura (5.8)), en este caso se puede comprobar que se realiza con éxito.

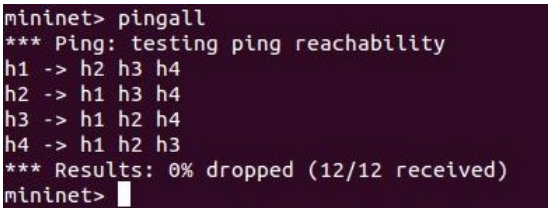


Figura 5.8: Comando *pingall* mininet.

A continuación, en la tabla (5.3) se muestran algunos de los comandos restantes que se pueden ejecutar en mininet.

Comandos básicos Mininet	
Comando	**Descripción**
Xterm [nodo]	Abre un terminal del nodo especificado
[nodo] *ping* [nodo]	Realiza un *ping* desde y hasta los nodos especificados
Exit	Realiza un *logout* del sistema.
Link[nodo1][nodo2][up/down]	Crea o elimina una interfaz entre dos nodos
sudo wireshark	Ejecuta el analizador de paquetes Wireshark.
Dump	muestra información de los nodos de la red.
[nodo] ifconfig	Indica información sobre las interfaces del nodo en cuestión.

Tabla 5.3: Comandos básicos de Mininet.

5.1.4. Topologías personalizadas

Tal y como se ha explicado, existe la posibilidad de crear diseños más personalizados a través de una API en python, que permite la creación de *scripts* para el diseño de topologías más complejas.

Esta API es similar al planteamiento que se realiza en JAVA, ya que es orientada a objetos. Se tiene un conjunto de objetos como Mininet, Topo, Link, Node de los cuales pueden heredar otros tipos como Controller, switch o host. A continuación, se proporciona un ejemplo sobre cómo se debería de programar un *script* para crear una topología.

En este caso, se realiza un diseño en el cual se crean dos *switches* interconectados entre si con dos *hosts* cada uno, junto con un controlador. Para empezar, se tiene que importar los paquetes que se van a utilizar en el *script* implementado (Figura (5.9)), en este caso se importaran varios paquetes, el paquete Mininet es el más importante ya que es el encargado de crear iniciar y parar la red.

personalizado.py ×

```
#!/usr/bin/python
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.link import Link, TCLink
```

Figura 5.9: Importación de paquetes Mininet.

Se han añadido otros tipos de paquetes que permiten obtener información sobre el *log* del sistema, añadir características a los enlaces o el uso de controladores específicos. Una vez importados los paquetes, se procede a empezar con el diseño de la red propuesta a través del uso de métodos y funciones para definir la red (Figura (5.10)).

```
def myNetwork():

        net = Mininet(topo=None, controller=RemoteController, link=TCLink)

        print "*** Creando hosts\n"

        h1 = net.addHost( 'h1', mac='00:00:00:00:00:01', ip='10.0.0.1/24' )
        h2 = net.addHost( 'h2', mac='00:00:00:00:00:02', ip='10.0.0.2/24' )
        h3 = net.addHost( 'h3', mac='00:00:00:00:00:03', ip='10.0.1.1/24' )
        h4 = net.addHost( 'h4', mac='00:00:00:00:00:04', ip='10.0.1.2/24' )

        print "*** Creando Switches\n"

        s1 = net.addSwitch( 's1', cls=OVSSwitch, mac='00:00:00:00:00:10', protocols='OpenFlow13' )
        s2 = net.addSwitch( 's2', cls=OVSSwitch, mac='00:00:00:00:00:11', protocols='OpenFlow13'  )
        print"***Creando controlador\n"
        c0 = net.addController( 'c0', controller=RemoteController, ip='127.0.0.1', protocols='OpenFlow13' )
        print "*** Creando enlaces\n"

        net.addLink(h1, s1)
        net.addLink(h2, s1)
        net.addLink(h3, s2)
        net.addLink(h4, s2)
        net.addLink(s1, s2)

        print"***Starting network"

        net.build()
        net.start()
        print"***Starting switches"
        s1.start([c0])
        s2.start([c0])
        print"***Starting controller"
        c0.start()

        print"***running CLI"
        CLI(net)
        print"***Stopping network"
        net.stop()

if __name__=='__main__':
        setLogLevel( 'info' )
        myNetwork()
```

Figura 5.10: Script en python.

Como se puede observar, existe un método myNetwork(), con el que se crea el diseño de red. Así, primero a través del objeto Mininet se configura la topología de red y con el uso de las siguientes funciones se pueden ir añadiendo elementos en la topología.

- **addHost(nombre, opciones):** esta función permite añadir un nuevo *host*. Se puede configurar su dirección Internet Protocol (IP), MAC, ruta por defecto, entre otras características.
- **addSwitch(nombre, opciones):** esta función añade un switch, igual que con la función anterior se pueden definir ciertas reglas como el tipo de switch a usar.
- **addLink(nodo1, nodo2, opciones):** añade un enlace entre los dos nodos indicado, permite la definición del ancho de banda y la velocidad.
- **addController(nombre, opciones):** añade un controlador, se puede indicar de qué tipo es, el puerto en el que escucha o incluso los protocolos de comunicación que utiliza.

Definidos los sistemas que van a constar en la red, solo queda iniciar la red y los distintos elementos que pertenecen a ella, una vez se llame a la

función. Así, usando el siguiente comando en la carpeta donde se encuentre el archivo creado, se ejecuta el programa:

```
$ sudo python personalizado.py
```

Como se ha visto es muy sencillo crear redes personalizadas, incorporando las distintas características que se necesiten en la red planteada, lo que facilita la creación de topologías más complejas a la hora de el desarrollo del proyecto.

Miniedit

Quizá, una de las peculiaridades más importantes de Mininet es la posibilidad de realizar los diseños de red de forma gráfica. Para ello, Mininet cuenta con la opción de un interfaz gráfico diseñado a partir de python, para usarlo solo hay que llamarlo tal y como se ha hecho con el *script* anterior en la carpeta de ejemplos dentro de Mininet:

```
$ sudo python miniedit.py
```

La interfaz es muy visual, a la izquierda se tienen los elementos que se pueden utilizar, como *switches* con capacidad OpenFlow, *hosts*, enlaces, controlador, entre otros. En el centro se encuentra la zona donde se pueden arrastrar estos elementos para crear el diseño de la red. Para comprobar su funcionamiento, se decide realizar una prueba muy simple, así se crea un escenario simple formado por seis *hosts*, tres *switches* y un controlador (Figura (5.11)). Para poner en marcha la red solo hay que seleccionar la pestaña *run* y en el interfaz de Mininet se ejecutará el diseño en cuestión.

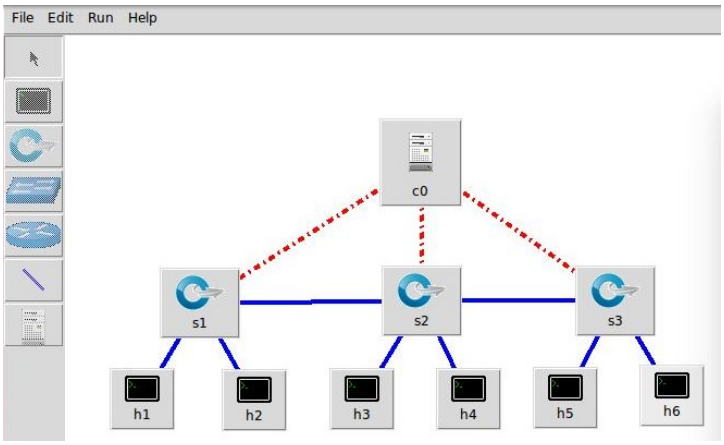


Figura 5.11: Miniedit.

Cabe destacar que para cargar el CLI antes hay que marcar la casilla *Start CLI* que se encuentra en Edit, preferencias, (Figura (5.12)). Además, desde este menú también se puede asignar el protocolo de comunicación usado por el controlador, el tipo de *switch* utilizado o incluso la dirección IP base del diseño. También, existe la posibilidad de poner el cursor encima de algún elemento de la red y pulsando el botón derecho se tiene la opción de entrar en un menú para configurar su dirección IP, protocolos, entre otras opciones.

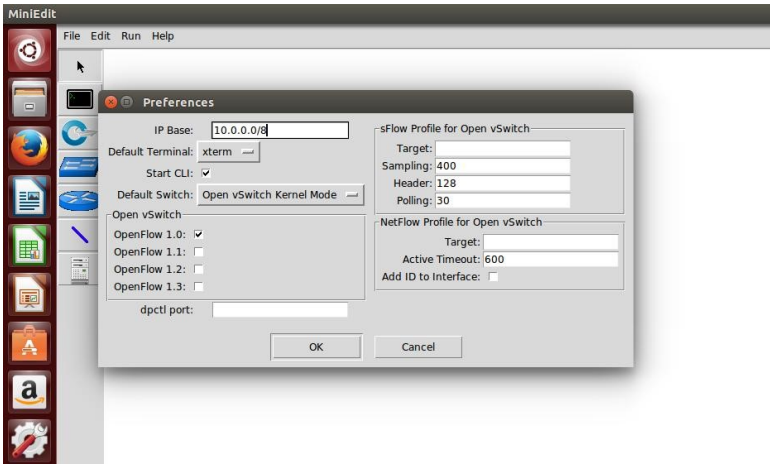


Figura 5.12: Habilitar CLI miniedit.

5.2. Quagga

Quagga [55] es una *suite* de *routing* que proporciona servicios de enrutamiento basados en TCP/IP, incluye protocolos como Routing Information Protocol (RIP) [60], Open Shortest Path First (OSPF) [44], Border Gateway Protocol (BGP) [65], PIM-SM [50], entre otros. Quagga utiliza una avanzada arquitectura de software para proveer un motor de alta calidad de enrutamiento. Además, tiene una interfaz de usuario interactiva para cada protocolo de enrutamiento soportando comandos comunes de los clientes. Actualmente, Quagga se distribuye bajo la licencia pública general de GNU y soporta GNU/LINUX, BSD y Solaris.

En resumen, Quagga aporta funcionalidades de *routing* a un sistema en concreto, así la máquina será capaz de intercambiar información de enrutamiento con otros routers, utilizando los protocolos que hayan sido establecidos, permitiendo de esta forma actualizar las tablas de routing y ver la información de estas tablas desde la interfaz de terminal de Quagga.

5.2.1. Arquitectura

Los software de *routing* tradicionales se basan en un único proceso que provee todas las funcionalidades de enrutamiento. Quagga en cambio adopta una aproximación diferente [54]. Está diseñado por una colección de procesos en *background* (demonios) que trabajan juntos para construir la tabla de enrutamiento. Existen demonios de protocolos de *routing* específicos que se ejecutan junto con "Zebra", el demonio principal que administra los demás.

En función del protocolo a utilizar se tienen que ejecutar distintos demonios, por ejemplo en la Tabla (5.4) se muestran algunas de las posibilidades.

Demonios Quagga	
Demonio	**Protocolo de *routing***
ospfd	Maneja el protocolo OSPFv3.
bgpd	Controla el protocolo BGP-4.
ripd	Soporta el protocolo RIP.
pimd	Basado en el protocolo PIM-SM.
qpimd	Demonio que ejecuta el protocolo PIM-SSM.
isisd	Ejecuta el protocolo IS-IS de estado de enlace

Tabla 5.4: Demonios básicos de Quagga.

Zebra es el encargado de interactuar con el kernel, para ello proporciona una API (Zserv) para el uso de los otros demonios y así estos no tienen que comunicarse con el kernel directamente (Figura (5.13)). Para modificar la tabla de *routing* del *kernel* se usa Zebra.

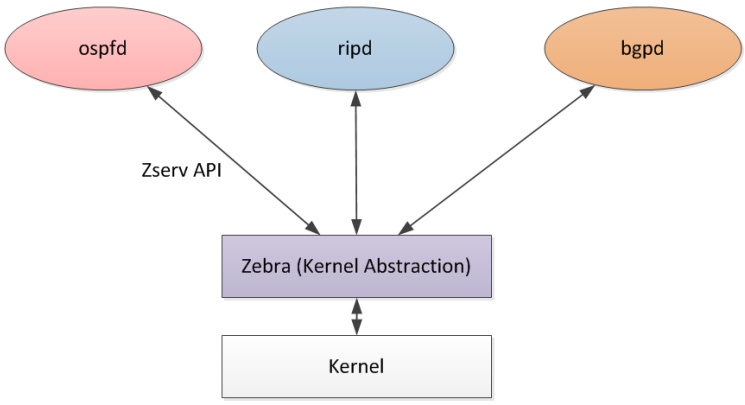


Figura 5.13: Arquitectura sistema Quagga [16].

Esta arquitectura permite ejecutar distintos demonios en la misma máqui-

na, creando un escenario con muchas posibilidades de *routing*. Así, este diseño multiproceso, permite un sistema modular con múltiples ficheros de configuración para cada demonio. De esta forma, es sencillo añadir nuevos demonios sin afectar a otros.

Como cada demonio tiene su fichero de configuración, el funcionamiento de cada protocolo se puede parametrizar. Asimismo, a la hora de configurar las rutas por defecto o las direcciones IP de cada elemento de la red, se pueden modificar mediante los ficheros de configuración de Zebra, lo que facilita realmente la operación, a la hora de su uso conjunto con Mininet. Además, también existe la posibilidad de acceder a una interfaz, a través del terminal de cada demonio, solo realizando Telnet al puerto en el que el demonio esté escuchando.

La interfaz de linea de comandos de Quagga es muy similar al de Cisco IOS software, por lo tanto si se está familiarizado con Cisco, la configuración de los demonios no es compleja.

5.2.2. Configuración de Quagga

Cada demonio de Quagga tiene sus propias peculiaridades a la hora de su configuración. Este Trabajo Fin de Grado (TFG) se centra en Zebra y qpimd, ya que son los dos demonios necesario para el futuro desarrollo de un escenario específico.

Tras instalar Quagga (véase Apéndice A.2) y creado los ficheros de configuración solo se necesita ejecutar el siguiente comando (en función de dónde se haya instalado Quagga) para lanzar los demonios necesarios:

- $ /usr/local/quagga/sbin/zebra -f /usr/local/quagga/etc/zebra-R1.conf -d -i /usr/local/quagga/etc/zebra-R1.pid

- $ /usr/local/quagga/sbin/pimd -f /usr/local/quagga/etc/pimd-R1.conf -d -i /usr/local/quagga/etc/pimd-R1.pid

Zebra

Anteriormente, se ha comentado que Zebra es el gestor de enrutamiento IP, realiza las actualizaciones de la tabla de encaminamiento del *kernel*, junto con la gestión de las rutas de los distintos protocolos de *routing* utilizados. Cada sistema o *switch* en este caso, tiene un fichero Zebra en función de los parámetros que necesite. En la Figura (5.14) se muestra un ejemplo de un fichero de configuración Zebra.

```
! -*- zebra -*-
!
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 9.0.0.1/24

interface R1-eth3
  ip address 11.0.0.1/24

!
ip route 6.0.0.0/8 9.0.0.2
ip route 7.0.0.0/8 11.0.0.2
ip route 10.0.0.0/8 10.0.0.1

log file /tmp/R1.log
```

Figura 5.14: Fichero Zebra.

Como se puede ver, es un fichero muy simple, primero se configura el nombre y el *password* del elemento en cuestión, de esta forma se añade seguridad al interfaz virtual del demonio. Además, tal y como se estudió en apartados anteriores se han añadido las rutas por defecto que interesan configurar en el *switch*, junto con las direcciones de las distintas interfaces. Así, cuando se ejecute el demonio de Zebra en el diseño planteado sobre en Mininet, se configurará la topología según las indicaciones configuradas. Finalmente, se ha añadido la creación de un *log* para diagnosticar posibles problemas ocurridos.

Una vez esté el demonio ejecutándose, se puede acceder a la interfaz virtual de Zebra a través de un Telnet al puerto 2601 usando la contraseña configurada anteriormente. Dentro de la interfaz virtual se tiene la posibilidad de ejecutar un conjunto de comandos similares a los *routers* de Cisco IOS, como por ejemplo para configurar interfaces, añadir rutas estáticas o modificar su descripción Figura (5.15).

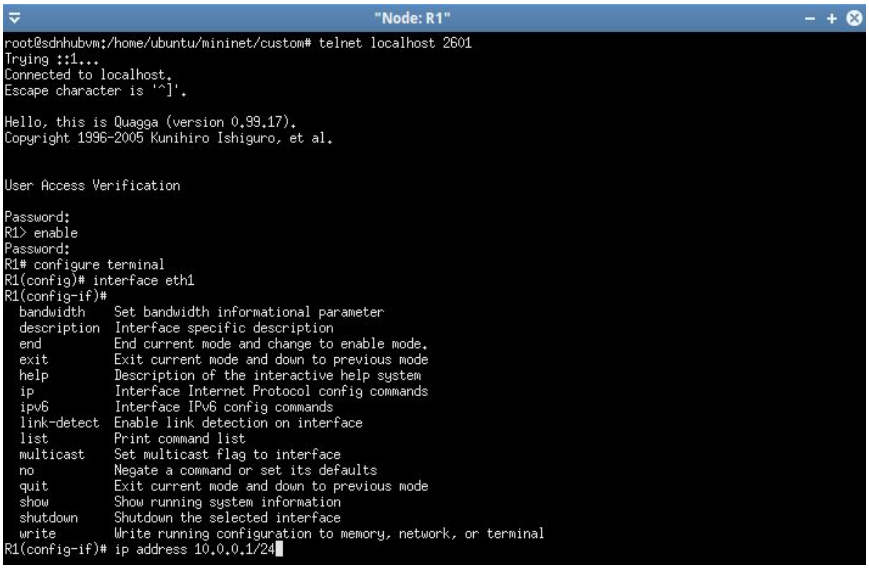


Figura 5.15: Interfaz Zebra.

Una vez realizado el Telnet y proporcionada la contraseña, se puede observar que es muy similar a un *router* Cisco, así se ejecuta el comando *enable* y *configure terminal*, entrando de esta forma al menú de configuración. Desde ahí se puede acceder a una interfaz y configurar el enlace según se desee. También, se tiene la posibilidad de usar el símbolo "?" para ver la variedad de comandos que se permiten utilizar.

Qpimd

Qpimd es un demonio de código abierto que implementa el protocolo PIM-SSM para el enrutamiento de paquetes multicast. Al igual que Zebra cada *switch* de la red tiene un fichero Qpimd para ser configurado.

Este documento es más simple incluso que el de Zebra, ya que solo existen dos posibilidades, añadir a la interfaz la capacidad de IGMPv3 o el funcionamiento de PIM-SSM (Figura (5.16)).

```
!
! pimd sample configuration file
!
!
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
!
ip multicast-routing
!
line vty
!
end
```

Figura 5.16: Fichero Qpimd.

Se puede observar que con el uso del comando *ip pim ssm* se añade la funcionalidad de PIM-SSM a esa interfaz y con el comando *ip igmp* la funcionalidad de IGMPv3. Lógicamente, la capacidad IGMP hay que añadirla a los enlaces que se conectan hacia un *host*.

Al igual que en Zebra existe la posibilidad de acceder a la interfaz virtual del demonio a través de Telnet al puerto 2611 y usando la contraseña establecida. En este caso, una vez dentro existen distintos comandos para la configuración de las interfaces multicast y la verificación de algunos parámetros (Figura (5.17)).

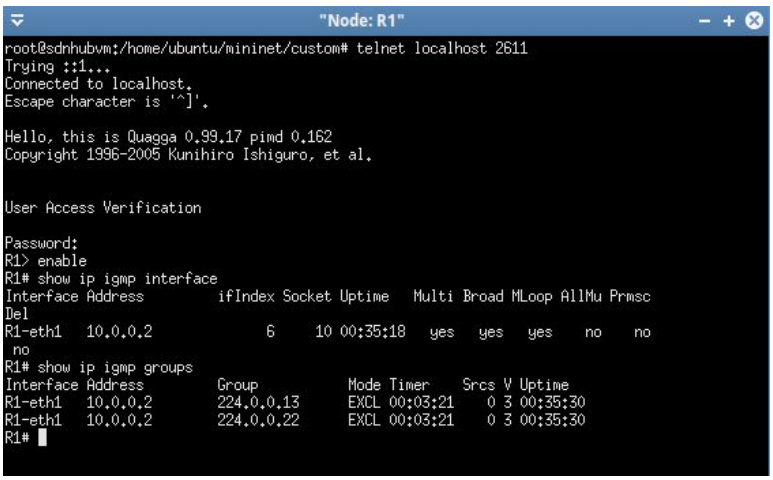


Figura 5.17: Interfaz Qpimd.

Con el comando *show ip igmp interface* se muestran las interfaces con esta funcionalidad, así se observa que la interfaz eth-1, en este caso, es la configurada para establecer una comunicación con el *host* usando IGMP. Además, a través de *show ip igmp groups* se pueden ver los grupos multicast que conoce el *router*. Para más información sobre los distintos comandos que se pueden utilizar en este demonio, consultar la referencia [6]. Cabe destacar que para el uso de qpimd hay que realizar un parcheado del demonio pimd, de ahí que aparezca como pimd en los ficheros de configuración, este parcheado se muestra en el Apéndice A.2.

5.3. OpenDaylight

OpenDaylight [17] es un controlador SDN de código abierto, creado con el fin de avanzar en la adopción de las redes definidas por software, recibe el apoyo de "Linux Foundation" junto con otras empresas importantes como Cisco, HP, IBM, Intel, etc... El objetivo del proyecto OpenDaylight es ofrecer una plataforma SDN funcional, sin necesidad de otros componentes. Es un controlador implementado en software y como una maquina virtual que es, se puede ejecutar desde cualquier sistema operativo que soporte Java.

En Febrero De 2014 OpenDaylight anunció su primera distribución *Hydrogen*, una edición base compuesta por el controlador y simples funcionalidades. La distribución actual es *Beryllium* (Figura (5.19)), anunciada en Marzo de 2016, la cual contiene mejoras considerables en modelos abstractos de red, junto con una mejora en el análisis y control de *clusters* de controladores.

Tras la versión base en 2013, se han lanzado cuatro distribuciones, a la

espera de nuevas incorporaciones tras la incorporación de *Beryllium* en el transcurso del proyecto (Figura (5.18)).

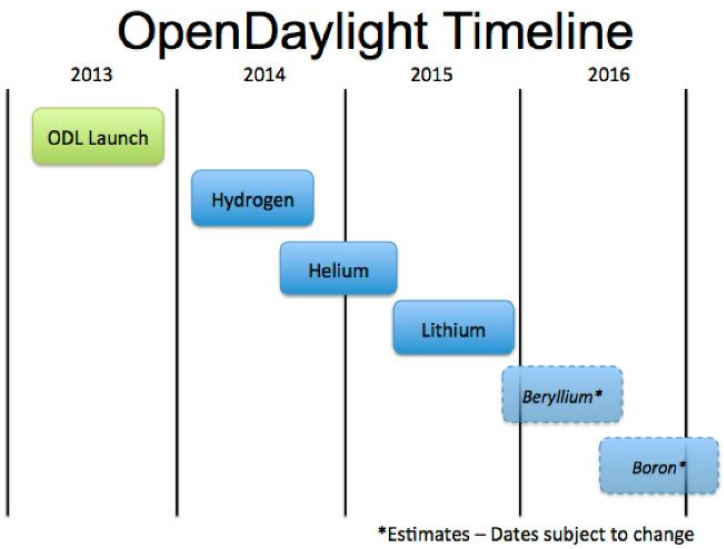


Figura 5.18: Timeline OpenDaylight [27]

En este TFG se hará uso de la distribución *Hellium*, debido a la gran cantidad de documentación encontrada y las facilidades que ofrece en la interfaz, debido al uso del contenedor *Apache Karaf*.

El controlador hace uso de las siguientes herramientas [21] para la programación de las distintas posibilidades existentes:

- **Maven**: OpenDaylight usa maven para facilitar la construcción de los proyectos. Maven utiliza el fichero pom.xml(Project Objetc Model) para las escritura de las dependencias entre paquetes y para describir que paquete tiene que iniciarse.
- **OSGi**. Este *framework* es el *back-end* de OpenDaylight, ya que permite cargar dinámicamente los *bundles* y los archivos .jar para el intercambio de información.
- **JAVA interfaces**: Las interfaces java se utilizan para la escucha de eventos, especificaciones y formar patrones. Es la forma principal a partir de la cual los *bundles* implementan funciones de *call-back* para eventos y estados específicos. Esta será la herramienta que usaremos para desarrollar el controlador, debido al conocimiento del lenguaje.
- **REST APIs**: Compuestas por las *NorthBound APIs*, para facilitar la comunicación con las aplicaciones, usa un lenguaje definido en YANG.

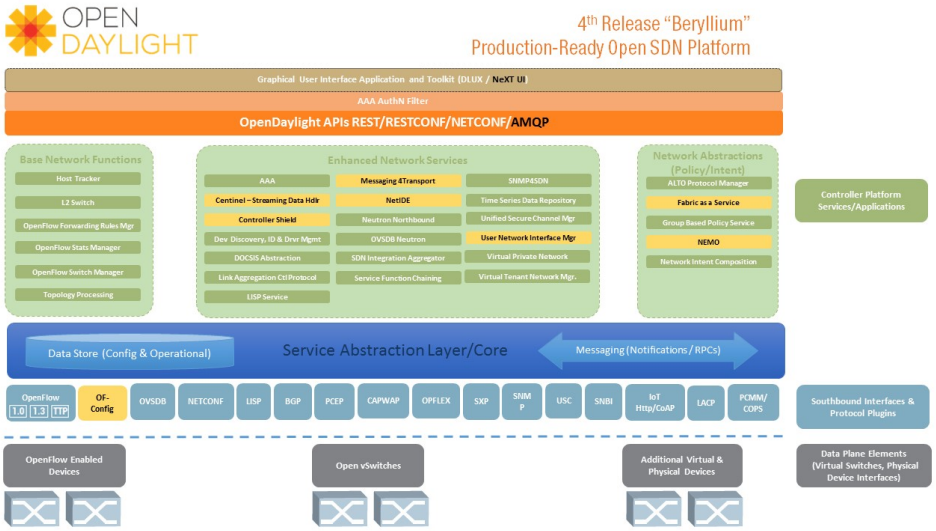


Figura 5.19: Arquitectura OpenDaylight [28]

El controlador OpenDaylight [27] expone *NorthBound APIs* que son utilizadas por las aplicaciones. OpenDaylight es compatible con la infraestructura OSGi y REST API bidireccional hacia las *NorthBound APIs*. OSGi se utiliza para aplicaciones que se ejecutan en el mismo espacio de direcciones que el controlador, mientras las REST API se utilizan para aplicaciones en distintas direcciones. La administración de la red reside en las aplicaciones que utilizan el controlador de red para reunir inteligencia, ejecutar algoritmos y obtener análisis, para después usar el controlador para producir nuevas normativas.

La interfaz *SouthBound* es capaz de dar soporte a múltiples protocolos (como *plugins* por separado) como OpenFlow, BGP-LS, NETCONF, etc. Estos módulos están vinculados dinámicamente en una capa de abstracción de servicio (Service Abstraction Layer (SAL)), donde se exponen los servicios del dispositivo. Así, SAL permite a OpenDaylight ejecutar la tarea solicitada, independientemente del servicio o protocolo subyacente que se utilice entre el controlador y los dispositivos de red.

5.3.1. Service Abstraction Layer

SAL es un intento de simplificar el desarrollo de la interfaz programable en OpenDaylight. La idea detrás de SAL es especificar un modelo independiente del lenguaje de programación para los servicios, en concreto esta especificación se llama YANG. Este enfoque se denomina *desarrollo dirigido por modelos*. Una especificación abstracta de SAL debería permitir la unificación de las APIs norte y sur del controlador, reduciendo de esta forma la

cantidad de código y permitiendo que los servicios se puedan desarrollar en una variedad de lenguajes.

Básicamente SAL actúa como un gran registro de los servicios anunciados por los distintos módulos. SAL debe saber cómo cumplir con un servicio solicitado, independientemente del protocolo subyacente que se utilice entre el controlador y los dispositivos de red. Así, cuando una aplicación determinada solicita un servicio a través de una API genérica, SAL es el responsable de la unión de la comunicación de los dos extremos.

SAL tiene hoy en día dos tipos de arquitecturas Aplication Driven Sal (AD-SAL) y Module Driven SAL (MD-SAL). Actualmente el enfoque AD-SAL se ha demostrado que es menos eficiente, por lo que todos los proyectos están migrando a MD-SAL. En la Figura (5.20) se muestra un esquema de las dos arquitecturas.

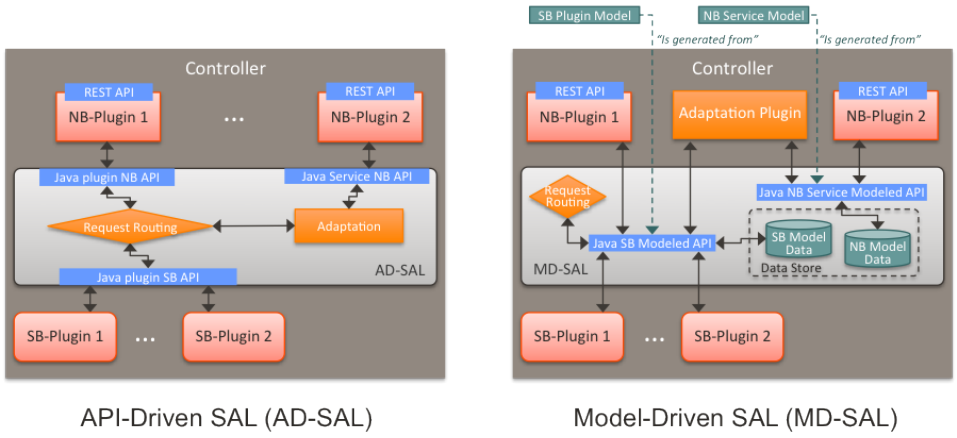


Figura 5.20: AD-SAL y MD-SAL [23]

AD-SAL proporciona solicitud de *routing* (seleccionado un SouthBound (SB) *plugin* en función del tipo de servicio) y opcionalmente realiza una adaptación del servicio. Como se puede observar en la Figura (5.20) se realiza una solicitud desde NorthBound (NB) *plugin* 1 hacia SB *plugin* 1 y 2, siendo estos del mismo tipo. Además, también se puede ver como en el caso de NB *plugin* 2 usa una API abstracta para poder acceder a los servicios que proveen los SB *plugin* 1 y 2, esta abstracción se realiza por un módulo de adaptación en AD-SAL.

MD-SAL [23] proporciona solicitud de *routing* y la infraestructura para realizar la adaptación del servicio. En cambio, el servicio de adaptación es proporcionado por *plugins*, este *plugin* provee de información a SAL y obtiene información a partir de las APIs generadas por los modelos. Por lo tanto, se realiza una adaptación mediante el *plugin* generado por unos modelos an-

teriormente definidos a través de YANG [19], proporcionando una abstracción superior a AD-SAL. Las MD-SAL APIs generadas a partir de modelos son equivalentes a las APIs de las funciones de AD-SAL. Además, MD-SAL puede almacenar información sobre los modelos definidos por los *plugins*, así el proveedor(proporciona funcionalidad a las aplicaciones a través de cualquiera de las API específicas) y consumidor(aplicación que utiliza el modelo y / o API proporcionadas por otro Proveedores) pueden intercambiar datos a través de este almacenamiento.

La estructura AD-SAL tiene típicamente tanto API NB y SB incluso para funciones o servicios que se asignan 1: 1 entre SB y NB plugins. En cambio, MD-SAL permite una única API para ambos, así un plugin se convierte en un proveedor de API; el otro se convierte en una API de los consumidores. Esto elimina la necesidad de definir dos APIs diferentes [20].

A continuación, se muestra una tabla que identifica las diferencias de las dos arquitecturas citadas [26]:

AD-SAL	**MD-SAL**
Las API de SAL requieren solicitud de *enrutamiento* entre los clientes y proveedores y las adaptaciones de datos son estáticamente definidas en tiempo de compilación o construcción.	Las API de SAL requieren solicitud de *enrutamiento* entre clientes y proveedores definidos desde modelos, las adaptaciones de datos son proporcionados por los *plugins* de adaptación internos.
AD-SAL tiene NB y SB APIs incluso para funciones y servicios que se asignan 1:1 entre SB y NB *plugins*.	MD-SAL permite el uso de NB Y SB *plugins* que utilizan la misma API generada a partir de un modelo. Así, un *plugin* se convierte en un proveedor de API y el otro en un cliente de API.
En AD-SAL existe una REST API dedicada para cada *northbound/-southbound plugin*.	MD-SAL contiene una REST API común para el acceso de datos y las funciones definidas en los modelos.
AD-SAL provee peticiones de *routing* y opcionalmente provee adaptaciones de servicio si una *North-Bound* API es diferente al protocolo usado en una *SouthBound* API.	MD-SAL provee peticiones de *routing* y la infraestructura para soportar adaptaciones del servicio, pero este es basado en *plugins*.

AD-SAL	**MD-SAL**
Las peticiones de *routing* están basadas en el tipo de *plugin*: SAL sabe que instancia de nodo es servido por que *plugin*, y cuando un *plugin* acNB solicita una operación en un nodo, la solicitud se envía al *plugin* adecuado que luego transfiera la solicitud al nodo apropiado.	Las peticiones de *enrutamiento* en MD-SAL se realizan en casos tanto de tipo de protocolo y de nodo, ya que las instancias de los nodos se exportan desde el *plugin* dentro de SAL.
AD-SAL no tiene estados.	MD-SAL puede almacenar datos para los modelos definidos por los *plugins* proveedores y consumidores pueden intercambiar datos a través del almacenamiento de MD-SAL.
Limitado a modelos de dispositivos y dispositivos *flow-capable*.	Modelo agnóstico. Puede soportar cualquier modelo de dispositivo o servicio y no se limita a fluir con capacidad solo modelos de dispositivos y servicios.
Los servicios de AD-SAL proporcionan versiones síncronas y asíncronas del mismo método API.	En MD-SAL, las API de modelo de servicio sólo proporcionan APIs asíncronas, pero devuelven un objeto *java.concurrent.Future* que permite bloquearse hasta que se procese la llamada y un resultado esté disponible. La misma API se puede utilizar para enfoques asíncronos y síncronos. De este modo MD-SAL promueve el enfoque asíncrono en el diseño de aplicaciones pero no se opone al uso síncrono.

Tabla 5.5: AD-SAL vs MD-SAL.

En este proyecto se ha decidido el uso de MD-SAL, ya que aunque la cantidad de documentación existente sea muy reducida comparada con AD-SAL, se postula como la referencia en OpenDaylight ofreciendo una flexibilidad y rendimiento superior, tal y como se ha explicado.

5.3.2. Uso de flujos proactivos

Para el desarrollo del controlador, existen dos formas de agregar flujos en las tablas, de forma reactiva o de forma proactiva. Los flujos reactivos crean entradas cuando se produce un cierto *match* en las tablas, mientras los flujos proactivos son los que se introducen explícitamente por parte del administrador.

A continuación, se expondrán dos ejemplos prácticos donde se crean entradas en las tablas de las dos formas posibles.

Primero, se va a mostrar un ejemplo en el cual se creara una topología sencilla tipo árbol de dos niveles, a partir de este diseño de red se usan distintos comandos para agregar entradas a las tablas de flujo, y de esta forma se logra la conectividad entre los distintos *hosts*, añadiendo las entradas a la tabla de flujos.

Para la creación de la topología tipo árbol, tal y como se estudió en el capitulo IV, hay que usar el siguiente comando, donde se indica el uso de un controlador remoto y la configuración de las direcciones MAC directamente:

```
$ sudo mn –topo=tree,2,2 –controller remote –mac
```

Una vez creada la topología se añaden las entradas a la tabla de flujo de forma manual, para ello se indica la MAC de destino y origen, de forma que exista conectividad entre el *host* 1 y 3. Además, se añade una regla para que los *switches* puedan procesar las tramas Address Resolution Protocol (ARP) a la hora de descubrir los *hosts* para realizar el *ping* (Figura (5.21)).

```
sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:3 -O OpenFlow13
sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:1 -O OpenFlow13
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:2 -O OpenFlow13
sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:1 -O OpenFlow13
sh ovs-ofctl add-flow s3 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:1 -O OpenFlow13
sh ovs-ofctl add-flow s3 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:3 -O OpenFlow13

sh ovs-ofctl add-flow s1 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
sh ovs-ofctl add-flow s2 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
sh ovs-ofctl add-flow s3 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
```

Figura 5.21: Flujos de forma manual.

Tras añadir los flujos, se puede realizar un *ping* entre los *hosts* indicados (Figura (5.22)). A la hora de añadir flujos existen otras opciones, como por ejemplo, se puede indicar que todos los paquetes entrantes de un puerto se encaminen hacia otro, o incluso ver qué entradas se han creado en la tabla. En [63] se pueden consultar todos los comandos disponibles.

```
mininet> sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:3
 -O OpenFlow13
mininet> sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:1
 -O OpenFlow13
mininet> sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:2
 -O OpenFlow13
mininet> sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:1
 -O OpenFlow13
mininet> sh ovs-ofctl add-flow s3 dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03,action=output:1
 -O OpenFlow13
mininet> sh ovs-ofctl add-flow s3 dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01,action=output:3
 -O OpenFlow13
mininet>
mininet> sh ovs-ofctl add-flow s1 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
mininet> sh ovs-ofctl add-flow s2 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
mininet> sh ovs-ofctl add-flow s3 dl_type=0x806,nw_proto=1,actions=flood -O OpenFlow13
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=3.84 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.346 ms
```

Figura 5.22: Ping host 1 y 3.

5.3.3. Uso de flujos reactivos

En este segundo ejemplo, se explica cómo se puede programar el controlador para el uso de flujos reactivos. Es decir, que tras la llegada de un paquete se realice una operación de búsqueda para encontrar algún tipo de *match*. Para ello, el ejemplo se basa en un *LearningSwitch* creado por [32], este código está desarrollado para su funcionamiento en un único *switch*, capítulos posteriores se desarrollara un *LearningSwitch* para cualquier número de *switches* con el objetivo de facilitar el intercambio de datos entre host, eliminando la necesidad de añadir rutas estáticas a mano.

En este código se crea un *Learning switch*. Es decir, se programa el controlador para que siempre y cuando le llegue un paquete que un *switch* no haya podido obtener hacia donde debe enviar el paquete correctamente, este pueda procesar hacia dónde hay que encaminar el paquete.

Para ello el controlador hace uso de la función *public void onPacketReceived(PacketReceived notification)* Figura (5.23), donde la variable *notification* sería el paquete entrante. Así, se extraen de este paquete una serie de parámetros como el tipo de paquete y las MAC destino y origen. Además, se hacen uso de diversas clases que identifican los diferentes nodos existentes en la red (*switches*) como por ejemplo *NodeConnectorRef*. A partir de esta información se busca un *match* en la tabla del controlador, con el objetivo de encaminar el paquete; en el caso de no existir ninguna entrada, se realiza un *flood* hacia todos los *switches* conectados menos por el puerto desde donde haya llegado el paquete.

```
public void onPacketReceived(PacketReceived notification) {
    LOG.trace("Received packet notification {}", notification.getMatch());

    NodeConnectorRef ingressNodeConnectorRef = notification.getIngress();
    NodeRef ingressNodeRef = InventoryUtils.getNodeRef(ingressNodeConnectorRef);
    NodeConnectorId ingressNodeConnectorId = InventoryUtils.getNodeConnectorId(ingressNodeConnectorRef);
    NodeId ingressNodeId = InventoryUtils.getNodeId(ingressNodeConnectorRef);
```

(a) Función OnPacketReceived

```
byte[] payload = notification.getPayload();
byte[] dstMacRaw = PacketParsingUtils.extractDstMac(payload);
byte[] srcMacRaw = PacketParsingUtils.extractSrcMac(payload);
```

(b) Extracción de payload de un paquete

Figura 5.23: Pseudo-código *Learning Switch*.

A continuación, se puede ver el ejemplo comentado anteriormente, en el que se crea una topología simple y se puede observar el funcionamiento del *Learning switch*.

Entonces, se crea una topología simple a través de mininet y se pone en marcha el controlador:

$ sudo mn –topo single,3 –mac –switch ovsk,protocols=OpenFlow13

–controller remote

Si se prueba a realizar algún tipo de *ping* a algún *host*, no se tiene conectividad entre ellos, ya que no existe ninguna ruta establecida. Ahora, se instala el tutorial *learning switch* integrado en la máquina virtual de SDNHUB [32], para poder realizar la función de *learning switch* que se ha explicado anteriormente (Figura (5.24)).

$opendaylight - user@root > feature : installsdnhub - tutorial-$

$learning - switch$

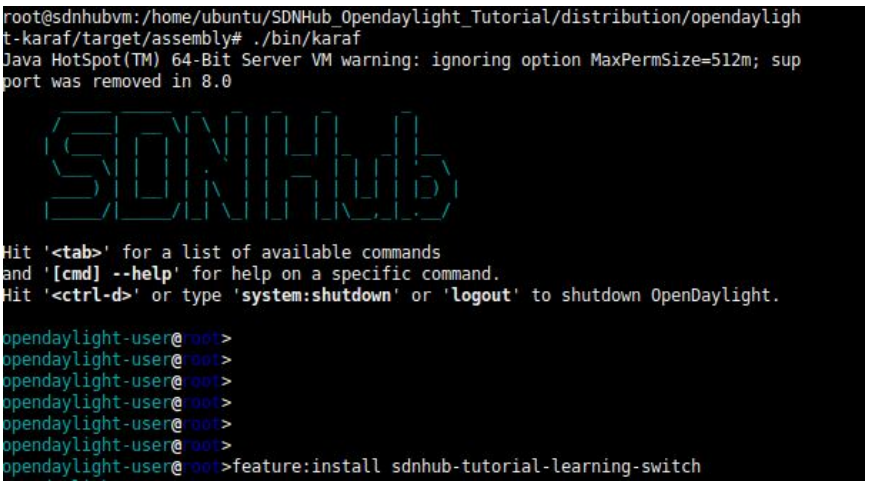


Figura 5.24: Puesta en marcha del controlador.

Tras realizar esto, solo hay que añadir un flujo que encamine los paquetes sin *match* al controlador y para que este los procese. Además, se pueden ver los flujos que se han creado en la tabla del *switch* y otros datos como la cantidad de paquetes enviados o el número de bytes (Figura (5.25)).

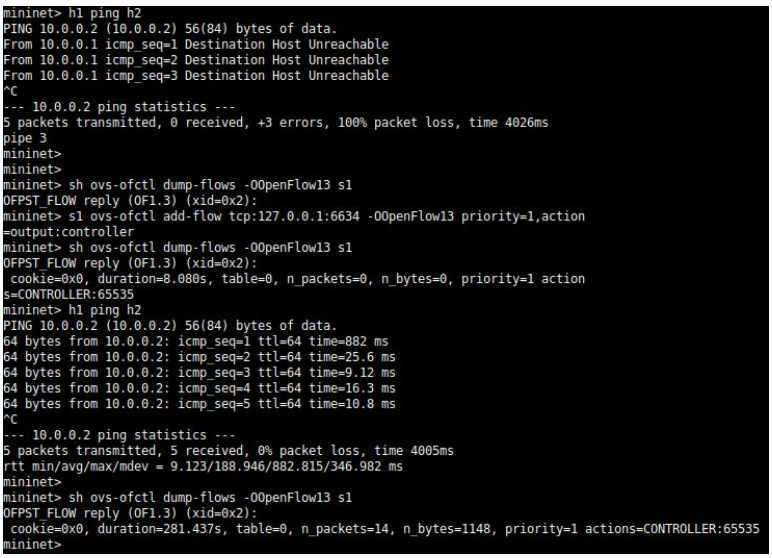


Figura 5.25: Ejemplo Opendaylight.

5.3.4. Interfaz gráfico

En todas las distribuciones de OpenDaylight se puede utilizar la interfaz gráfica DLUX [22]. DLUX es una aplicación de gestión de red para el con-

trolador OpenDaylight. Esta aplicación permite ver de una manera gráfica información detallada sobre el panorama general del controlador y la arquitectura de red. DLUX utiliza los servicios de SAL para obtener información relacionada con la red y proporcionar funciones de administración de red. En el Apéndice A.3 se pueden ver los requisitos para instalar DLUX.

Una vez esta instalado DLUX, para entrar en él, solo se tiene que poner en marcha el controlador, junto con una topología para poder ver las curiosidades de la interfaz gráfica. Así, para acceder a él hay que usar la siguiente Uniform Resource Locator (URL) en el navegador:

$$http://IP\_Controlador:8181/index.html$$

En este caso, la IP del controlador era 127.0.0.1, así indicando la dirección anterior aparece un menú de *login*, en el que el usuario y la contraseña son *admin* y *admin*, respectivamente (Figura (5.26)). Cabe destacar que en la guía ofrecida por OpenDaylight aparece la siguiente dirección $http://IP\_Controlador:8181/dlux/index.html$, pero en este proyecto, junto a otros usuarios en la red da problemas y se encontró la solución utilizando la anterior dirección propuesta.

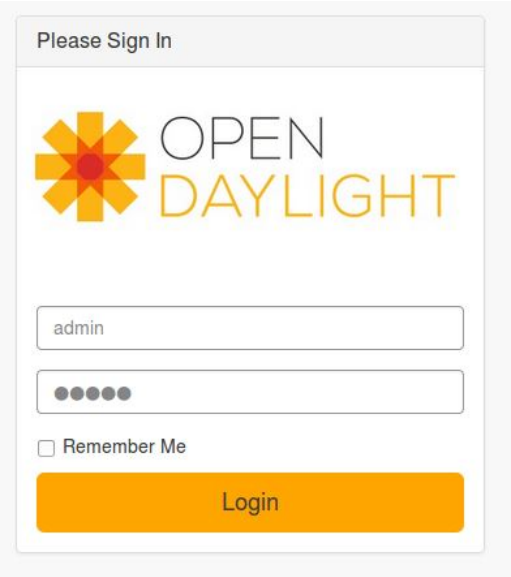


Figura 5.26: Login DLUX.

Tras iniciar sesión en DLUX se pueden ver todos los módulos disponibles en el panel izquierdo, siempre y cuando estén habilitadas en la distribución *karaf*. En concreto en MD-SAL solo se pueden obtener los siguientes módulos:

- **Nodes**. Donde aparece información sobre todos los elementos de la red, como su nombre, *Node_id*, conexiones y distintas estadísticas sobre flujos.
- **Yang UI**. El módulo de interfaz de usuario YANG permite interactuar con OpenDaylight.
- **Topology**. Esta pestaña muestra una representación gráfica de la topología de red creada.

En este caso se ha habilitado solo la opción *Topology* para realizar una prueba, así se implementa una topología árbol 2,2 y tras crearla, si se abre el navegador y se inserta en la dirección antes comentada, aparecerá la interfaz DLUX, en concreto en el apartado *Topology* solo se pueden ver los tres *switches* existentes conectados entre si. Esto se debe a que el controlador, con el objetivo de descubrir el diseño de la red, envía paquetes Link Layer Discovery Packet (LLDP) hacia todos los *switches* que han establecido conexión con este, así cada vez que un *switch* recibe un LLDP *packet* lo reenvía por todos sus puertos, de tal forma que cuando un *switch* lo recibe realiza un *flow lookup* para descubrir el camino que ha seguido el paquete. Entonces el *switch* envía un packet_in al controlador con la información; así el controlador puede analizar todos los paquetes y puede tener una visión global de la red. Cuando los *hosts* realizan cualquier acción de comunicación, como un *ping*, el controlador descubre su localización (Figura (5.27)).

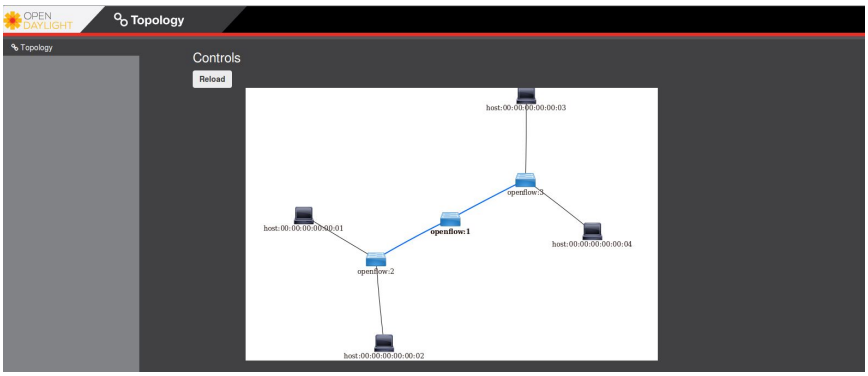


Figura 5.27: Topología DLUX.

Dentro de la interfaz gráfica existe la opción de agregar entradas a las tablas de flujos o mirar su contenido, aún así tal y como pasaba con *Miniedit*, no se recomienda su uso, ya que de vez en cuando ocurren errores debido a que la versión no está 100 % depurada.

Capítulo 6

Caracterización de un protocolo Multicast, PIM-SSM

En el presente capítulo se estudiarán todos los aspectos referentes al diseño e implementación de un escenario multicast, con el objetivo de caracterizar el protocolo PIM-SSM.

Primero, se expondrá la topología elegida para el desarrollo del escenario, describiendo cómo se ha combinado el uso de Quagga con Mininet y especificando los distintos módulos necesitados en los *switches* para que realicen su función como *routers*. A continuación, se comentará los distintos ficheros Quagga necesarios para el correcto encaminamiento de los datos multicast, junto con el *script* usado para la creación de la topología. Finalmente, se realiza una evaluación del escenario con el uso de un demonio multicast, con el objetivo verificar su funcionamiento y estudiar cómo se crean los distintos árboles en la topología creada.

6.1. Creación de un escenario Multicast

A la hora de decidir qué topología sería la adecuada para la caracterización del protocolo, se pensó implementar un escenario sencillo pero significativo, debido a los futuros problemas que tendríamos al añadir el controlador a la red, junto con las limitaciones de computación del ordenador portátil utilizado.

De esta forma, la topología creada (Figura (6.2)) es similar a una topología tipo árbol, hay cinco *hosts*; el *host* superior (o raíz) actuará siempre de proveedor de servicios multicast, mientras los demás usuarios serán clientes

del servicio ofrecido. Se han añadido varios enlaces redundantes con el objetivo ver claramente cómo pueden cambiar los árboles tras la caída de un enlace, observando los distintos mensajes *join/prune* que se envían a través de la red y realizar futuras evaluaciones del protocolo.

Para la creación del escenario se hizo uso de Mininet, tal y como se estudió en el Capítulo V. Así se implementa un *script* en python para la creación de una topología personalizada. El *script* es más complejo comparado con los ejemplos vistos, además existe la necesidad de convertir los *switches* usados en Mininet por *routers*, con el objetivo de añadir las funcionalidad de *routing* a través de Quagga. En el Apéndice C.1 se puede observar el código implementado.

Para transformar los *switches* en *routers* se ha adoptado una aproximación basada en [4] -donde se realiza un *BGP path hijacking attack*- y en [12], donde se utiliza un nodo como *router*. En ambos ejemplos se activa la funcionalidad de *routing*, de esta forma solo se tiene que crear una nueva clase *Router* en el código en cuestión para crear *switches* específicos, con el objetivo de permitir *IP forwarding*, finalmente se asigna esta clase en la estructura de Mininet.

Una vez se ha programado la creación de los distintos elementos se tiene que poner en marcha Quagga, para ello en cada *router* creado se ejecutan los demonios de zebra y qpimd (Figura (6.1)). Para llevar a cabo este paso, previamente se deben de haber creado los ficheros zebra y qpimd, pertenecientes a cada *router*.

```
    for router in net.switches:
        router.cmd("/usr/local/quagga/sbin/zebra -f /usr/local/quagga/etc/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/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()
        log("Starting zebra and pimd on %s" % router.name)
```

Figura 6.1: Puesta en marcha de demonios quagga.

Tal y como se estudió en el capitulo V, a partir de los ficheros zebra y qpimd se pueden configurar distintas características de la red. En concreto, la configuración de qpimd es bastante sencilla, solo hay que establecer los enlaces en los que se tendrá funcionalidad IGMPv3 y en los que se establecerá capacidad PIM-SSM. Por lo tanto, aquí no se comentará de nuevo en la configuración de los ficheros qpimd, ya que su implementación es trivial y fue ya estudiada en el capitulo V.

Por otra parte, los ficheros zebra permiten una mayor versatilidad y flexibilidad, permitiendo configurar los distintos parámetros de red, como las direcciones IPs de los elementos de red, lo que facilita la configuración

de la topología para permitir la interconexión entre todos los dispositivos de la red.

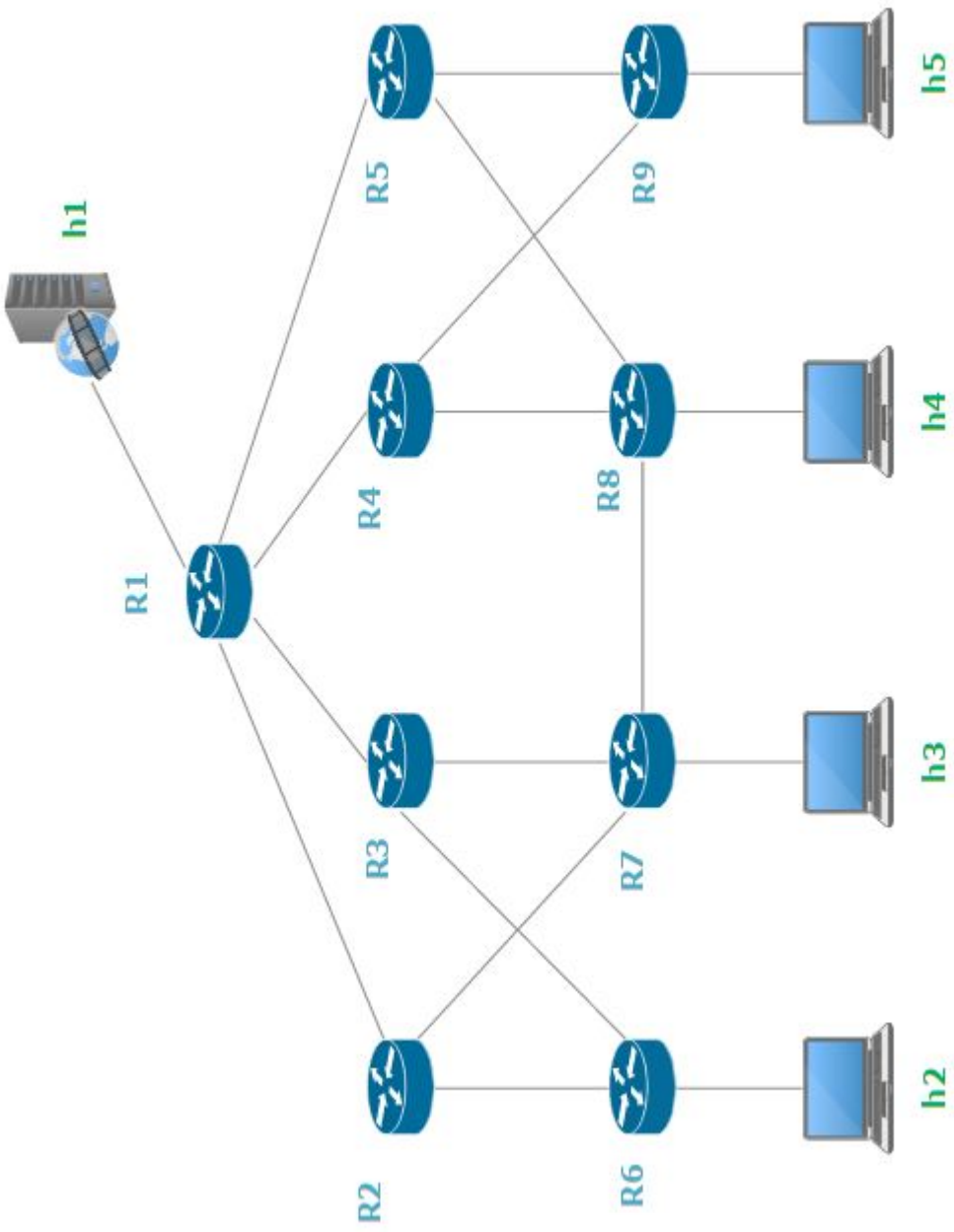


Figura 6.2: Escenario PIM-SSM

Para la establecer la conexión entre de los distintos *hosts*, se ha añadido en los ficheros zebra las rutas por defecto para llegar a los diferentes dispositivos de la red, junto con las direcciones IPs de las interfaces de cada *router*, tal y como se puede en la Figura (6.3). Los demás ficheros serán similares al anterior, por lo que no es de interés detallar la configuración de cada uno de ellos.

```
! -*- zebra -*-
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 12.0.0.1/24
interface R1-eth3
 ip address 12.0.1.1/24
interface R1-eth4
 ip address 12.0.2.1/24
interface R1-eth5
 ip address 12.0.3.1/24
!
ip route 6.0.0.0/8 12.0.0.2
ip route 7.0.0.0/8 12.0.1.2
ip route 8.0.0.0/8 12.0.2.2
ip route 9.0.0.0/8 12.0.3.2
ip route 10.0.0.0/8 10.0.0.1

log file /tmp/R1.log
```

Figura 6.3: Fichero zebra R1.

Para terminar la configuración del escenario, es necesario añadir las direcciones de los distintos *hosts* del diseño y la puerta de enlace por defecto, parámetros que no se pueden modificar con Quagga. Para ello, se han añadido un par de lineas en el *script*, que permite añadir comandos en los terminales de cada uno de los *hosts* existentes siendo i el número de host y X la interfaz a configurar:

net.hosts[i].cmd("ifconfig hi-ethX DIRECCION_IP")

net.hosts[i].cmd("route add default gw DIRECCION_IP")

A continuación, en la Figura (6.4) se puede observar la salida del comando **"net"** en mininet, donde se muestran las distintas interfaces de cada elemento de la red en la topología que se ha implementado.

Además, en la Tabla (6.1) se muestra una descripción con la configuración final del escenario, indicando las direcciones IPs de cada uno de los interfaces de los distintos elementos de la topología, con el objetivo de entender la estructura de la red. Cabe destacar que en la asignación de IPs no se ha sido restrictivo, debido a que no se tenían limitaciones en lo referente al número de IPs disponibles.

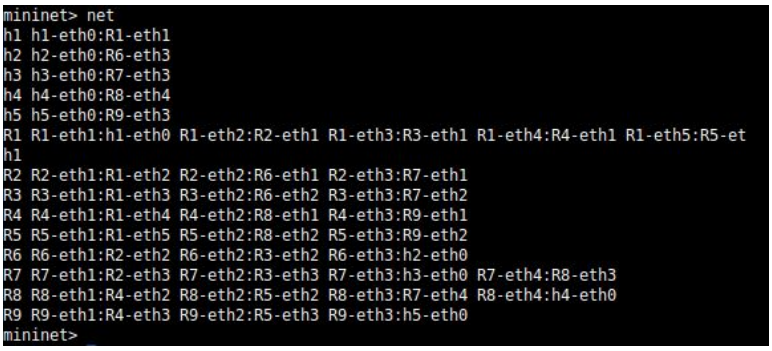


Figura 6.4: Salida del comando net.

Asignación de direcciones de red	
Elemento/Interfaz	Dirección IP
R1	
Eth1	10.0.0.2/24
Eth2	12.0.0.1/24
Eth3	12.0.1.1/24
Eth4	12.0.2.1/24
Eth5	12.0.3.1/24
R2	
Eth1	12.0.0.2/24
Eth2	12.0.4.1/24
Eth3	12.0.5.1/24
R3	
Eth1	12.0.1.2/24
Eth2	12.0.6.1/24

Elemento/Interfaz	Dirección IP
Eth3	12.0.7.1/24
R4	
Eth1	12.0.2.2/24
Eth2	12.0.8.1/24
Eth3	12.0.9.1/24
R5	
Eth1	12.0.3.2/24
Eth2	12.0.10.1/24
Eth3	12.0.11.1/24
R6	
Eth1	12.0.4.2/24
Eth2	12.0.621/24
Eth3	6.0.0.2/24
R7	
Eth1	12.0.5.2/24
Eth2	12.0.7.2/24
Eth3	7.0.0.2/24
Eth4	12.0.12.1/24
R8	
Eth1	12.0.8.2/24
Eth2	12.0.10.2/24
Eth3	12.0.12.2/24
Eth4	8.0.0.2/24
R9	
Eth1	12.0.9.2/24
Eth2	12.0.11.2/24
Eth3	9.0.0.2/24
Hosts	
H1	10.0.0.1/24
H2	6.0.0.1/24
H3	7.0.0.1/24
H4	8.0.0.1/24
H5	9.0.0.1/24

Tabla 6.1: Direccionamiento de red.

Una vez configurada la red se realizan pruebas con el objetivo de comprobar la interconexión entre todos los elementos de la red, con un simple *ping* se comprueba la correcta configuración de el diseño de red.

Tras estudiar cómo se ha llevado a cabo la creación de la topología de red y cómo se ha establecido la configuración de el diseño, con el objetivo de

entender futuras secciones, se procede a la evaluación del protocolo multicast en el escenario anterior.

6.2. Evaluación de PIM-SSM

En esta sección se pretende evaluar el protocolo PIM-SSM en el escenario implementado anteriormente. Se estudiará cómo se forman los distintos árboles en la red para hacer llegar el tráfico multicast a los clientes, se comprobará qué mensajes se intercambian entre los distintos nodos de red y cómo se modifican las distintas de tablas de los *routers* con el establecimiento de los árboles.

Para realizar la evaluación, se tiene que hacer uso de algún tipo de aplicación o servicio que nos permita enviar tráfico multicast. En este caso se ha utilizado **ssmping** [29] [43], es una herramienta de gestión de red multicast que ofrece la posibilidad de comprobar si se pueden recibir paquetes de multidifusión a través de SSM de un servidor.

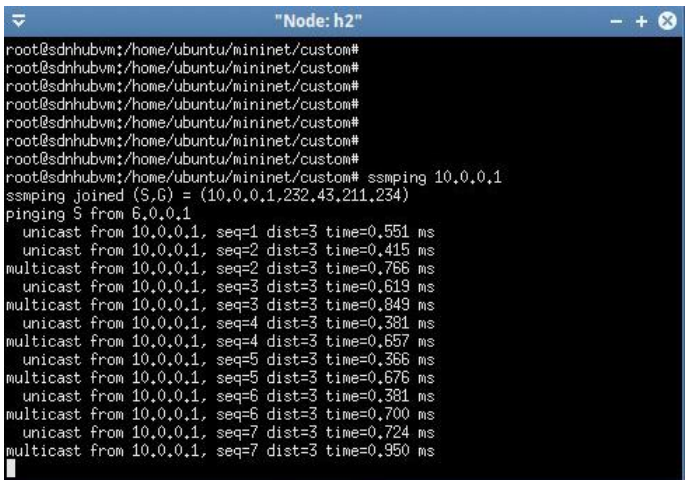
El funcionamiento es muy simple, el *host* que hace de servidor tiene que iniciar un demonio **ssmpingd**, el cual estará en escucha en el puerto 4231 a espera de algún tipo de petición, entonces los clientes interesados envían un mensaje tipo unicast *ssmping query*. Cuando el servidor recibe una petición, este responde con mensajes tipo unicast al cliente y multicast a un grupo multicast SSM, que en este caso al utilizar IPv4 la dirección del grupo es 232.43.211.234, y de esta forma se puede comprobar que el escenario en cuestión es capaz de transmitir datos multimedia.

6.2.1. Ejemplo práctico

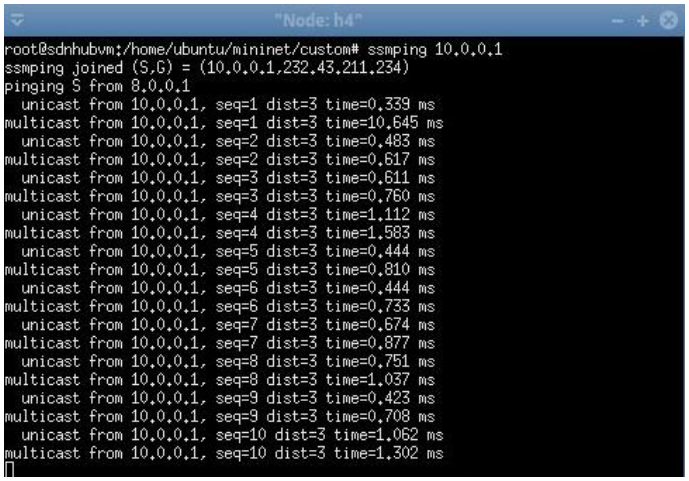
Tal y como se explicó en secciones anteriores el host-1 será el servidor de contenidos multimedia, mientras los demás *hosts* serán clientes del servicio. A continuación, se va a realizar un ejemplo en el cual los host 2 y 4 realizarán una petición al servidor.

Primero, se inicia el demonio *ssmpingd* en el host-1, de tal forma que este se quede en espera hasta que reciba peticiones de clientes. Una vez iniciado el demonio desde el host 2 y 4 se realiza una petición al supuesto servidor de contenidos.

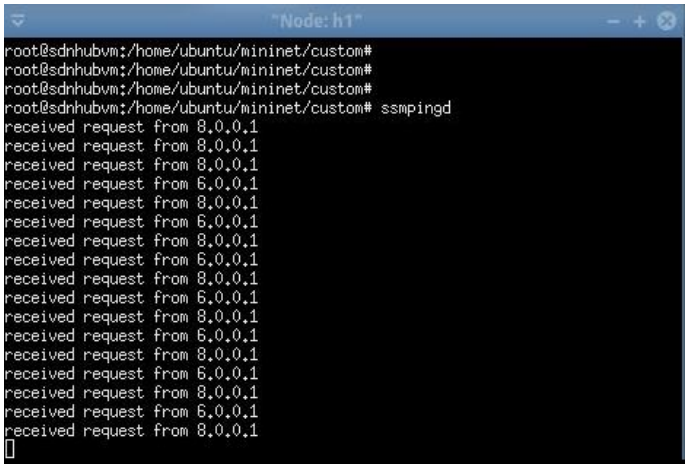
Si la configuración de los distintos ficheros *zebra* y *qpimd* de red es correcta se debería de ver cómo llegan los paquetes multicast a los terminales de los host 2 y 4 y las peticiones de los clientes a la fuente, comprobando así el correcto funcionamiento del escenario (Figura (6.5)).



(a) Terminal host 2



(b) Terminal host 4



(c) Terminal host 1

Figura 6.5: Prueba tráfico multicast.

Como se explico en el Capítulo IV, PIM-SSM es una mejora de PIM-SM que permite la creación de árboles tipo *Source Distribution Trees* sin necesidad de un RP. Esto se logra haciendo que el receptor conozca a la fuente de antemano y se una a la misma utilizando IGMPv3, así los clientes realizan una petición IGMPv3 tipo (S,G), en donde se especifica la dirección IP de la fuente y el grupo de que se quieren recibir los datos multimedia.

Tras comprobar que se ha establecido el envío de datos multicast de forma correcta, se va a estudiar cuáles han sido los árboles establecidos por el escenario, junto con los mensajes implicados que han hecho posible la transmisión multimedia.

Primero, se realiza una captura a través de la herramienta wireshark [33] del *router* R6, el cual está conectado directamente con el host-2, tal y como se muestra en la Figura (6.6) recibe una petición IGMPv3 para la adhesión al grupo multicast 232.43.211.234, tras recibir la petición se realiza un join con el objetivo de crear el árbol. También se pueden observar los distintos mensajes *PIMv2 Hello* que comprueban la conectividad con los diferentes dispositivos vecinos con capacidad multicast en la red.

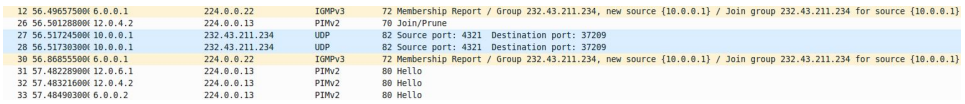


Figura 6.6: Captura wireshark R6.

A continuación, si se observan los paquetes recibidos por R2 (Figura (6.7)), se puede apreciar que se reciben las peticiones unicast de la fuente y el cliente -en el establecimiento de conexión a través de la herramienta *pimssm*- junto con los mensajes tipo *join* hacia los *routers* R1 y R6. Estos tienen como objetivo de crear el árbol de multidifusión y finalmente facilitar el tráfico multicast hacia el grupo especifico.

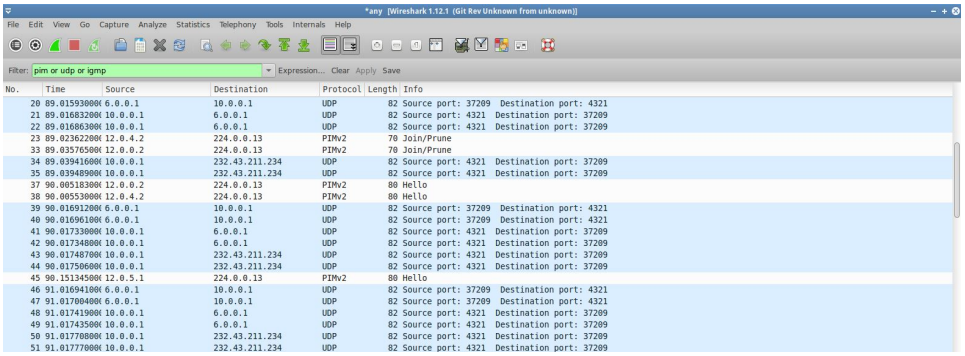


Figura 6.7: Captura wireshark R2.

Finalmente, se realiza una captura del *router* R1, tal y como se aprecia en la Figura (6.8) e igual que en R2 se recibe los paquetes unicast para establecer la conexión; se envían los datos multicast al grupo establecido 232.43.211.234 y se pueden observar los mensajes tipo *join* de nuevo, solo que esta vez están destinados a los dos clientes h2 y h4, ya que R1 cuelga directamente de la fuente multicast. No se van a mostrar los capturas de tráfico de los routers R8 Y R4 ya que son similares a las mostradas anteriormente.

51	149.022584000	10.0.0.1	6.0.0.1	UDP	82	Source port: 4321 Destination port: 37209
52	149.022853000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209
53	149.041687000	12.0.0.2	224.0.0.13	PIMv2	70	Join/Prune
59	149.045163000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209
61	150.008801000	12.0.1.2	224.0.0.13	PIMv2	80	Hello
62	150.010969000	12.0.0.2	224.0.0.13	PIMv2	80	Hello
63	150.020062000	12.0.2.1	224.0.0.13	PIMv2	80	Hello
64	150.022760000	6.0.0.1	10.0.0.1	UDP	82	Source port: 37209 Destination port: 4321
65	150.022798000	6.0.0.1	10.0.0.1	UDP	82	Source port: 37209 Destination port: 4321
66	150.023053000	10.0.0.1	6.0.0.1	UDP	82	Source port: 4321 Destination port: 37209
67	150.023087000	10.0.0.1	6.0.0.1	UDP	82	Source port: 4321 Destination port: 37209
68	150.023221000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209
69	150.023244000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209

(a) join host 2

89	152.723117000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 49470
90	152.723147000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 49470
91	152.733946000	12.0.2.2	224.0.0.13	PIMv2	70	Join/Prune
92	153.022203000	6.0.0.1	10.0.0.1	UDP	82	Source port: 37209 Destination port: 4321
93	153.022232000	6.0.0.1	10.0.0.1	UDP	82	Source port: 37209 Destination port: 4321
94	153.022490000	10.0.0.1	6.0.0.1	UDP	82	Source port: 4321 Destination port: 37209
95	153.022516000	10.0.0.1	6.0.0.1	UDP	82	Source port: 4321 Destination port: 37209
96	153.022736000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209
97	153.022766000	10.0.0.1	232.43.211.234	UDP	82	Source port: 4321 Destination port: 37209

(b) join host 4

Figura 6.8: Captura wireshark R1.

A continuación, se estudia de una manera más visual los distintos mensajes intercambiados por la red a la hora del establecimiento de conexión entre el servidor de contenidos y los distintos clientes. Cabe destacar que los mensajes transmitidos por el protocolo PIM-SSM son significativamente menores comparando con las otras propuestas multicast existentes en la red.

En la Figura (6.9) se puede ver gráficamente cómo se ha establecido el árbol de distribución. Los clientes a través de un *membership report IGMPv3* envían una solicitud de unirse al grupo multicast con fuente especifica (10.0.0.1, 232.43.211.234). Una vez realizada la petición se envían mensajes tipo *PIM-join* con destino hacia la fuente, creándose en el camino entradas (S,G) que formarán el árbol *Source Distribution Tree*, estableciendo el camino SPT.

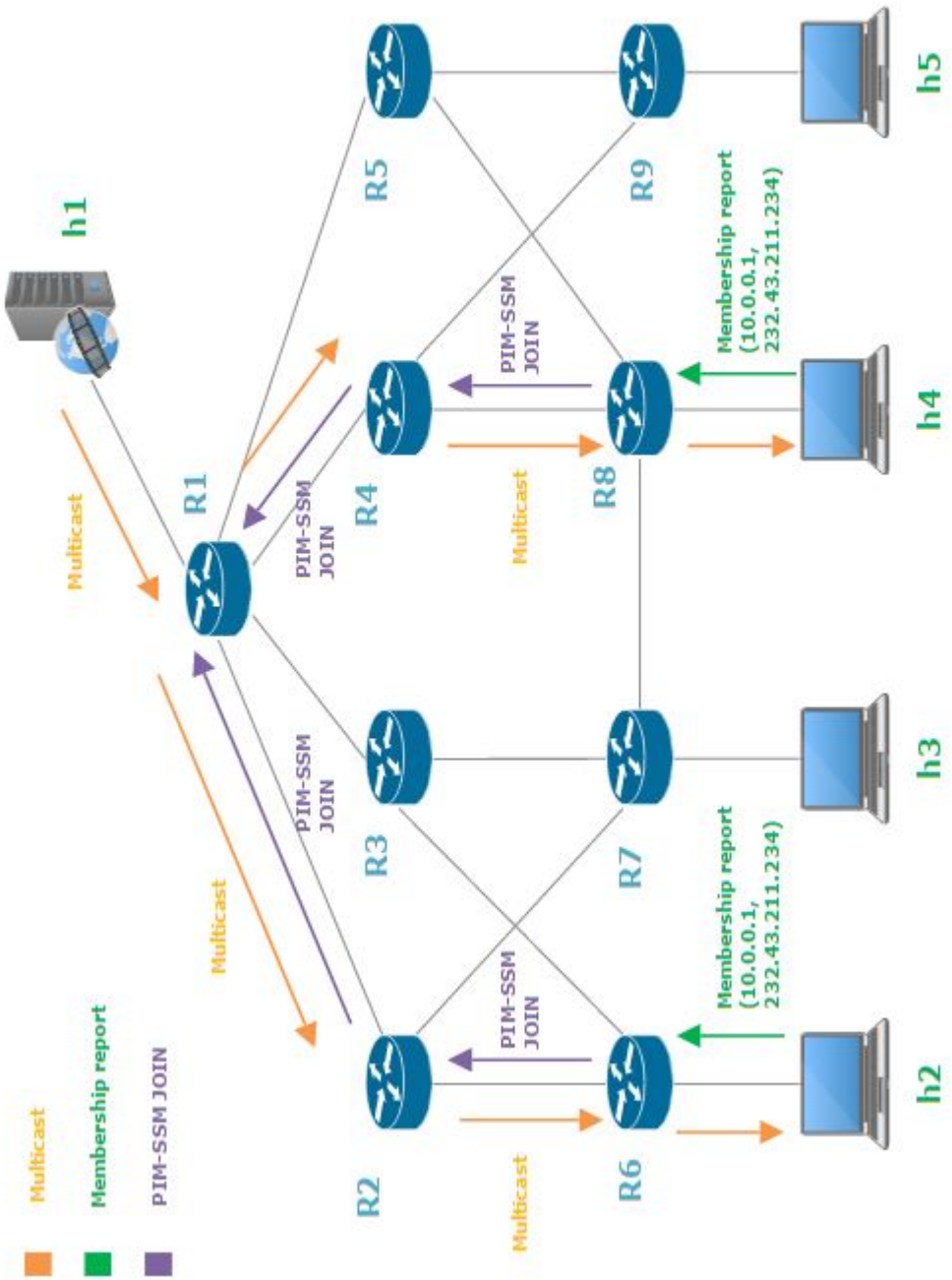


Figura 6.9: Escenario PIM-SSM

6.2.2. Demonio qpimd

Para aprovechar las distintas funcionalidades que ofrece el demonio de Quagga *qpimd* basado en PIM-SSM, se puede realizar un telnet hacia el puerto apropiado en un router determinado tal y como vimos en capítulos anteriores:

```
$ telnet localhost 2611
```

Una vez dentro del sistema, tras realizar un *enable* para ser *root* del sistema, existen varios comandos posibles para obtener distintos datos referentes al protocolo multicast. Por ejemplo se pueden ver las distintas tablas creadas en la transmisión multicast que se ha estudiado anteriormente, o incluso estadísticas sobre la cantidad de datos transmitidos. Así, en la Figura (6.10) se muestra la tabla de *routing* multicast del *router* R1.

```
R1# show ip mroute
Proto: I=IGMP P=PIM

Source          Group           Proto Input iVifI Output oVifI TTL Uptime
10.0.0.1        232.43.211.234  P     R1-eth1     6 R1-eth2     2   1 00:01:52
10.0.0.1        232.43.211.234  P     R1-eth1     6 R1-eth4     4   1 00:01:33
R1#
```

Figura 6.10: Salida del comando show ip mroute

Además, se observa cómo existen dos entradas en la tabla, en ambas existe una fuente común y un mismo grupo multidifusión. En cambio, las salidas corresponden con las interfaces que llevan hacia los dos clientes h2 y h4, además se indica el tiempo que llevan activos.

```
R1# show ip multicast
Mroute socket descriptor: 5
Mroute socket uptime: 00:05:12

Current highest VifIndex: 6
Maximum highest VifIndex: 31

Upstream Join Timer: 60 secs
Join/Prune Holdtime: 210 secs

RPF Cache Refresh Delay:    10000 msecs
RPF Cache Refresh Timer:    0 msecs
RPF Cache Refresh Requests: 10
RPF Cache Refresh Events:   1
RPF Cache Refresh Last:     00:05:01

Interface Address         ifi Vif  PktsIn PktsOut    BytesIn   BytesOut
R1-eth1   10.0.0.2          6   6     321       0      21186          0
R1-eth2   12.0.0.1          2   2       0     321          0      21186
R1-eth3   12.0.1.1          3   3       0       0          0          0
R1-eth4   12.0.2.1          4   4       0     300          0      19800
R1-eth5   12.0.3.1          5   5       0       0          0          0
R1#
```

Figura 6.11: Salida del comando show ip multicast

En la Figura (6.11) se puede observar información global multicast, entre las distintas posibilidades se puede obtener el número de paquetes o bytes transmitidos, varios parámetros de RPF, como por ejemplo el tiempo transcurrido desde la última actualización o los eventos recibidos.

Finalmente, se puede obtener información sobre los diferentes grupos igmp establecidos en un router. En la Figura (6.12) se puede observar la salida de dos comandos en el router R6, igual que antes se muestra la tabla de routing multicast a través del comando *show ip mroute*, donde aparece el grupo 232.43.211.234 con interfaz de salida destino el host-2, además se muestra la salida del comando *show ip igmp groups* que indica qué grupos de multidifusión están conectados directamente al router, y los cuales se aprenden a través del protocolo igmp, se puede utilizar este comando para verificar que una fuente o receptor se han unido al grupo concreto en la interfaz del router.

```
R6# show ip mroute
Proto: I=IGMP P=PIM

Source          Group           Proto Input iVifI Output oVifI TTL Uptime
10.0.0.1        232.43.211.234  I     R6-eth1     2 R6-eth3     4   1 00:03:45
R6# show ip igmp groups
Interface Address         Group           Mode Timer    Srcs V Uptime
R6-eth3   6.0.0.2         224.0.0.13      EXCL 00:03:21    0 3 00:06:17
R6-eth3   6.0.0.2         224.0.0.22      EXCL 00:03:21    0 3 00:06:17
R6-eth3   6.0.0.2         232.43.211.234  INCL --:--:--    1 3 00:03:56
R6#
```

Figura 6.12: Salida del comando show ip igmp groups.

Capítulo 7

Desarrollo de un protocolo Multicast sobre SDN

En este capítulo se pretende explicar los procedimientos llevados a cabo para la implementación de un controlador OpenDaylight que permita el intercambio de tráfico multicast entre servidores multimedia y clientes, en un escenario basado en una arquitectura SDN.

Primero, se indicará la lógica llevada a cabo en el desarrollo del código basado en Java, se estudiará qué procedimientos sigue el controlador OpenDaylight tras la llegada de un paquete tipo multicast, para permitir la transmisión de contenido multimedia. A continuación, se indicarán el funcionamiento de las diferentes funciones y clases utilizadas en el código para que realice su función correctamente.

Finalmente, se realizarán un conjunto de pruebas con el objetivo de verificar el correcto desarrollo del protocolo multicast en un entorno basado en SDN, comprobando la viabilidad de esta tecnología para la gestión del tráfico de uno a muchos.

7.1. Lógica de la implementación

En esta sección se va explicar la lógica llevada a cabo para la implementación del controlador OpenDaylight con funcionalidad multicast. Para ello, se abordarán las funciones de interés usadas para el procesado de los paquetes y su correcto encaminamiento a través de la red.

Para el desarrollo del controlador se optó por una topología emulada sobre Mininet igual a la mostrada en el Capítulo VI (véase Figura 7.1), aunque en este caso no es imprescindible la funcionalidad de *routing*, si es necesario eliminar los enlaces que crean bucles en la topología, debido a que se esta

trabajando en *layer 2*, además en este caso no se añaden las rutas unicast de forma estática, si no que son introducidas por el controlador, así al no existir la posibilidad de usar *TTL* para eliminar el tráfico redundante, es necesario eliminar los enlaces que añadan bucles al escenario. La implementación es muy similar a la anterior por lo que no se va a entrar en cómo se ha creado el *script* en python para diseñar el escenario, para más detalles consultar el código proporcionado en el Apéndice C.2.

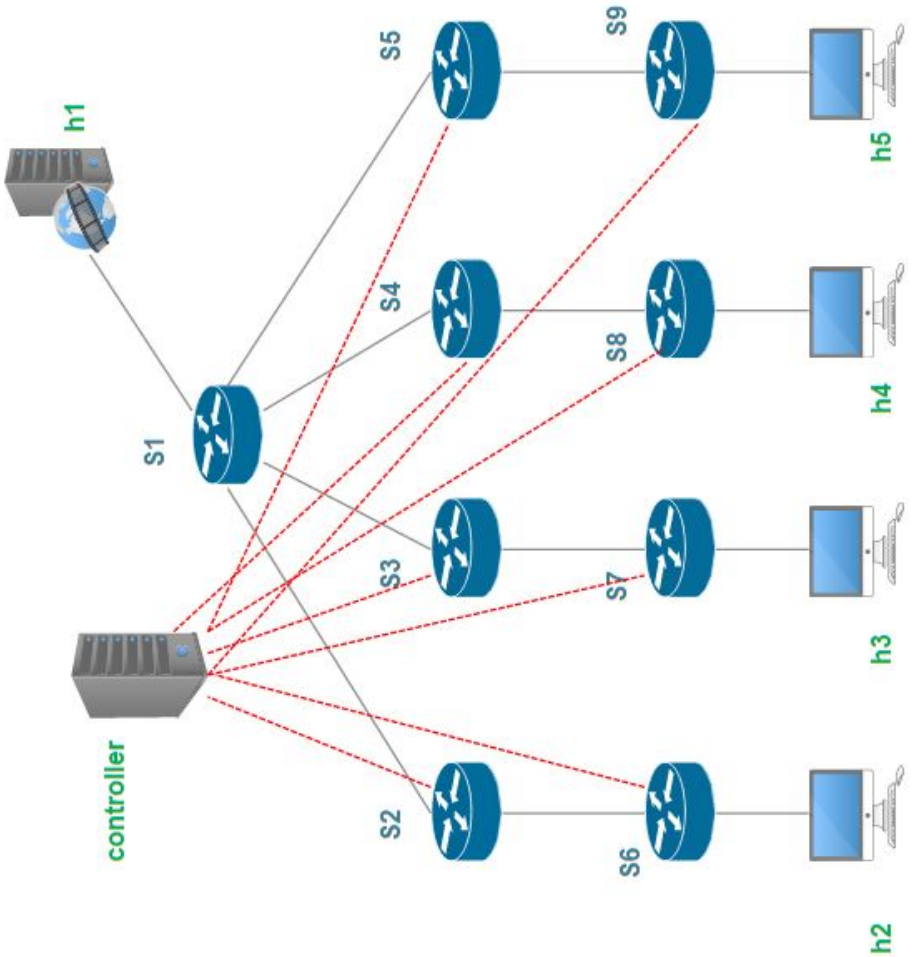


Figura 7.1: Escenario OpenDaylight.

7.1.1. Regla IGMP

Una vez está la red diseñada, se tiene que buscar una lógica en el desarrollo del controlador, para solucionar la problemática propuesta, es decir, el controlador debe de permitir el tráfico multicast en la red diseñada en función de las peticiones de los clientes.

Primero, se necesita que el controlador tenga conocimiento de las peticiones IGMPv3 *Membership Report* transmitidas por los host hacía los *switches*, para unirse o darse de baja de un grupo especifico. De esta forma, el controlador tiene conocimiento de las direcciones IP de las fuentes activas y de los clientes interesados en ellas. Así, se tiene que crear una regla en los *switches* que tengan conexiones a *hosts* que reenvíen todos los paquetes tipo IGMPv3 hacia el controlador.

Esta regla tiene que tener una prioridad elevada ya que se requiere que cualquier cambio en las necesidades de los *hosts* sea percibido por el controlador, en concreto esta regla se aplicará para los *switches* $s1 - s2 - s6 - s7 - s8 - s9$ (como así se muestra en la Figura (7.2)), ya que son los nodos que contienen *hosts* en la red diseñada.

```
ovs-ofctl add-flow -OOpenFlow13 s1
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller
ovs-ofctl add-flow -OOpenFlow13 s6
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller
ovs-ofctl add-flow -OOpenFlow13 s7
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller
ovs-ofctl add-flow -OOpenFlow13 s8
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller
ovs-ofctl add-flow -OOpenFlow13 s9
dl_type=0x0800,nw_proto=2,priority=65535,actions=output:controller
```

Figura 7.2: Regla IGMP.

Con el uso de esta regla, se tendría solucionado el primero de los problemas encontrados, el controlador recibe las peticiones de los clientes y por lo tanto tiene el control de la red.

En la Figura (7.3) se puede observar un diagrama secuencial, que muestra el intercambio de mensajes entre la red y el controlador. Cuando un host quiere unirse un grupo, este envía un mensaje tipo IGMPv3 *Membership Report* hacia el switch de egreso, a continuación el *switch* busca en su tabla de flujo algún tipo de *match* para este tipo de paquete, así el *switch* encuentra la regla que establece encaminar todos los paquetes tipo IGMP hacia el controlador. Una vez el paquete esta en el controlador, este realiza una serie de procesos para extraer la información de interés contenida en el paquete y en base a esta se crean las rutas necesarias a través de mensajes *Packet Out*

a los *switches* pertenecientes al camino. De esta forma el tráfico multicast se transmite correctamente desde la fuente hacia el cliente.

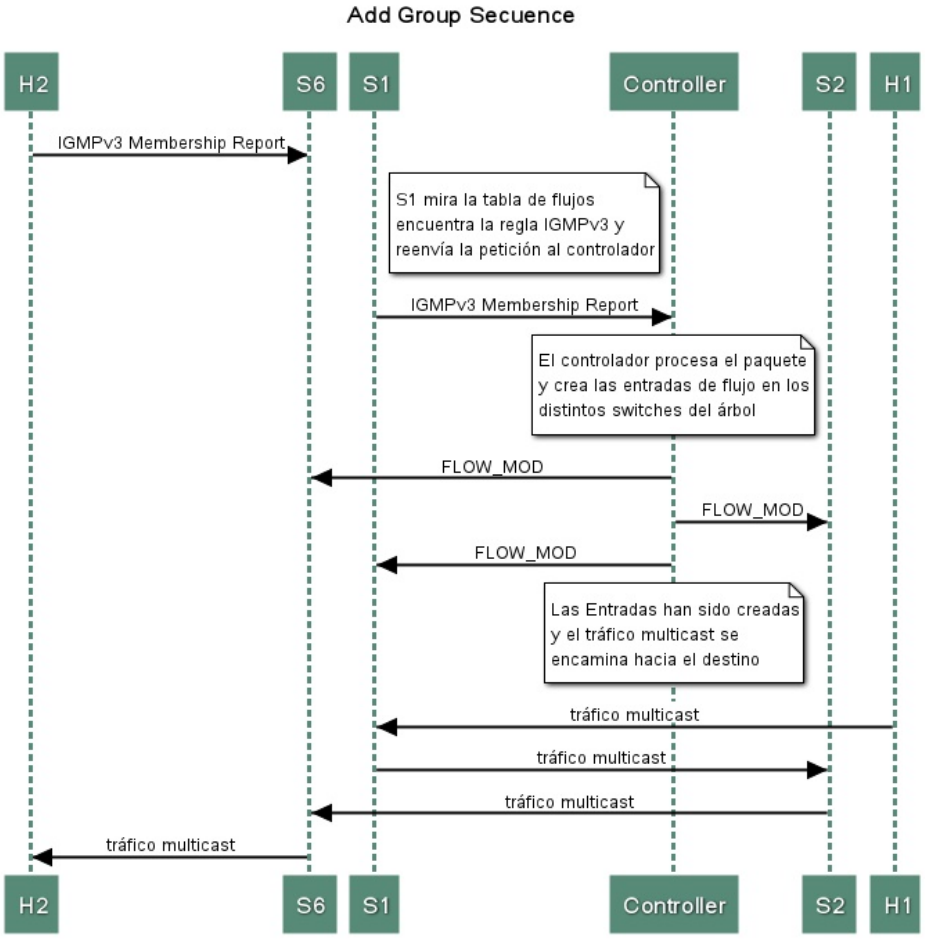


Figura 7.3: Diagrama secuencial.

En el siguiente sub-apartado se muestra como se ha realizado el análisis del paquete recibido por el controlador, de forma que se extraigan toda la información de interés contenida en el paquete.

7.1.2. Análisis de un paquete IGMP

Cuando un paquete llega al controlador, este tiene que ser capaz de procesarlo y extraer toda la información necesaria para gestionar la red (en este caso en el ámbito multimedia), de modo que este tenga todos los parámetros imprescindibles para añadir las rutas que encaminan el tráfico

multicast hacia los clientes desde la fuente.

Para ello, el controlador necesita obtener algunos datos del paquete entrante, así se realizó un *parsing* del paquete IGMPv3, para comprobar qué campos eran de intereses para su procesado en el controlador.

Si observamos la Figura (7.4) se puede ver la descomposición de un paquete IGMP, en este caso es de *Type 0x22*; es decir, *IGMPv3 Membership Report*, este campo indica el tipo de mensaje IGMP, en concreto tras realizar capturas wireshark se comprobó que este era el modelo de mensaje enviado por los *hosts* para indicar la adhesión a un grupo.

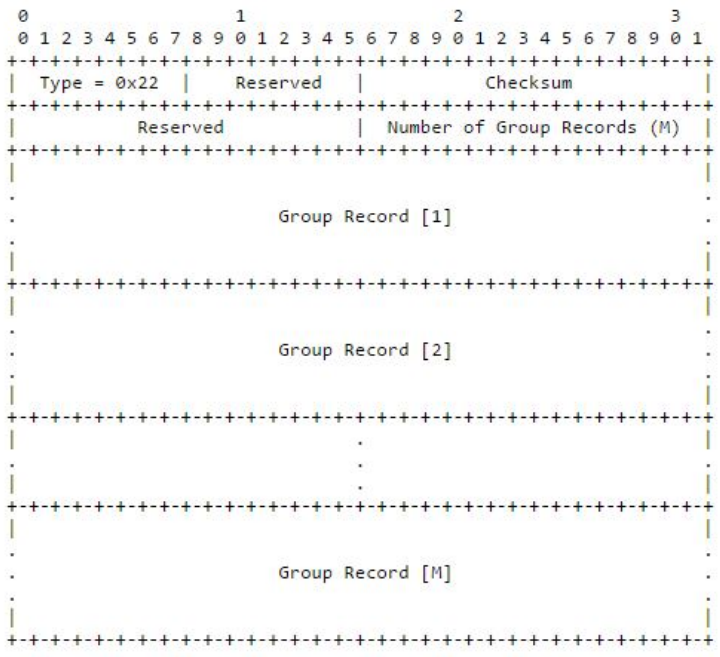


Figura 7.4: Formato de un paquete IGMP.

Aún así, el campo más importante en el ámbito de este TFG es el *Group Record* Figura (7.5). Este campo indica la dirección IP de la fuente, junto con la dirección multicast del grupo al que el *host* se quiere unir, además contiene un campo llamado *Record Type* que indica si se requiere unirse a un grupo o por el contrario, ser eliminado de uno (el valor 5 indica la unión a un grupo y el valor 6 indica la desunión a un grupo concreto).

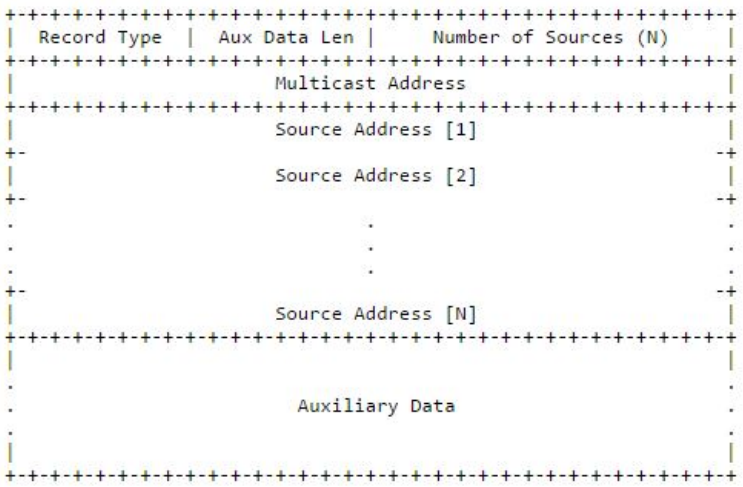


Figura 7.5: Contenido Group Record.

Como se comentó en capítulos anteriores, se utiliza la máquina virtual SDN-HUB que ofrece varias funcionalidades, dentro del uso de un controlador MD-SAL, se encuentra la clase *PacketUtils.java* que contiene algunas funciones para descomponer algunos campos de los paquetes entrantes, como obtener las MAC destino y origen. Sin embargo, no contiene los campos que se necesitan para descomponer los datos importantes de un paquete IGMPv3.

Tras obtener los datos del *parsing*; es decir, dónde comienza y termina cada campo. Se tuvo que realizar una mejora de esta clase para obtener algunos campos de interés, como las direcciones IPs de Grupo y de fuente, junto con el valor del *Record type*. Además, se realizaron algunas mejoras adicionales para futuros trabajos, como la obtención de las direcciones IPs origen y destino, junto con alguna función para obtener de forma legible estos datos.

Tras obtener estos datos, el controlador será capaz de saber cuando un *host* quiere unirse o no, a un grupo; es decir, tendrá la capacidad de añadir o eliminar las rutas necesarias en función de las peticiones de los clientes, analizando el campo *Record type*.

Así, en el código implementado se crean dos funciones principales en función del *Record type*, *multicastAddFlow* y *multicastDeleteFlow*, las cuales añaden o eliminan las entradas de flujo correspondientes en los distintos *switches* que sean de interés (ver Figura (7.6)) siempre y cuando el paquete sea de tipo IGMP.

```
//IGMP PACKET
    if (protoTypeRaw == 2){

    LOG.debug("packet in igmp type: " +igmpTypeRaw);
    LOG.debug("packet in record type: " +recordTypeRaw);

         if(recordTypeRaw == 5) {

           multicastAddFlow(ingressNodeId, dstMac, srcMac, dstIp, srcIp,
srcMulticast, GroupMulticast, ingressNodeConnectorId, payload);

           }
         if(recordTypeRaw == 6) {

           multicastDeleteFlow(ingressNodeId, dstMac, srcMac, dstIp,
srcIp, srcMulticast, GroupMulticast, ingressNodeConnectorId, payload);
          }

     }
```

Figura 7.6: Código OpenDaylight.

Además, si se requiere como en este caso que se ejecute las funciones implementadas cada vez que el controlador recibe un paquete, el método principal tiene que implementar *PacketProcessingListener* para que ejecute el código desarrollado siempre y cuando le llegue un paquete.

7.1.3. Algoritmo Dijkstra

OpenDaylight identifica los distintos nodos que contiene la red con la dupla "$openflow : X : Y$" siendo X el número de *switch* e Y un puerto concreto, por lo tanto, se tiene que hallar una forma para obtener el camino hacia los clientes en este formato.

Así, una vez el controlador tiene conocimiento sobre la acción a realizar (añadir o eliminar entradas de flujo), necesita conocer el camino desde la fuente hasta los clientes que estén unidos al grupo multicast concreto. Para ello, como el paquete IGMP proporciona la dirección IP de la fuente y la dirección IP del *host* cliente, se hace uso de una función *GetPath* que devuelve el camino a seguir en forma de "$openflow : X : Y$" tal y como se necesita para identificar los diferentes nodos de red (véase Figura 7.9).

Además, con el objetivo de facilitar la programación se han creado una serie de funciones que devuelven el identificador de un elemento concreto (*whoIs()*), o un *array* con el contenido de los identificadores de un camino (*getSwitches()*) dado.

```
private void multicastAddFlow(NodeId ingressNodeId, String dstMac, String
srcMac, String dstIp, String srcIp, String srcMulticast , String
GroupMulticast, NodeConnectorId ingressNodeConnectorId, byte[] payload) {

      LOG.debug("IGMP PACKET--> CREAR GRUPO");
      Ipv4Prefix GroupMulticast_IP= new Ipv4Prefix(GroupMulticast+"/32");

      int sourcehost = whoIs(srcIp);
      int server = whoIs(srcMulticast);

      String [] nodes = getPath(server, sourcehost);
      int [] switches = getSwitches(server, sourcehost);
      String string = "openflow:";
      Ipv4Prefix GroupMulticast_IP= new Ipv4Prefix(GroupMulticast+"/32");
```

Figura 7.7: Código OpenDaylight.

La función *GetPath(server, sourcehost)*, calcula la ruta más corta desde la fuente hacia un cliente concreto, a través del algoritmo *Dijkstra*. El algoritmo *Dijkstra* calcula el camino más corto de un vértice origen hacia los demás vértices de la topología, en base a un grafo establecido con distintos pesos en los enlaces [68]. El algoritmo analiza todos los caminos hacia todos los nodos existentes, finalizando encontrados los caminos más cortos (Figura 7.8).

Algorithm DijkstraShortestPaths(G, v):
Input: A simple undirected weighted graph G with nonnegative edge weights, and a distinguished vertex v of G
Output: A label $D[u]$, for each vertex u of G, such that $D[u]$ is the distance from v to u in G
$D[v] \leftarrow 0$
for each vertex $u \neq v$ of $\vec{G}$ **do**
  $D[u] \leftarrow +\infty$
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  $u \leftarrow Q$.removeMin()
  for each vertex z adjacent to u such that z is in Q **do**
    {perform the ***relaxation*** procedure on edge (u, z)}
    if $D[u] + w((u, z)) < D[z]$ **then**
      $D[z] \leftarrow D[u] + w((u, z))$
      Change to $D[z]$ the key of vertex z in Q.
return the label $D[u]$ of each vertex u

Figura 7.8: Algoritmo Dijkstra [49].

Para la creación del algoritmo *Dijkstra* se necesita guardar el grafo de la topología para encontrar la ruta más corta en función de los pesos de los enlaces, así se han creado cuatro clases para permitir la creación del grafo y el uso de *Dijkstra*:

- **Clase Vertex**. Esta clase contiene el constructor para crear los vérti-

ces existentes en la topología, para mayor comodidad, se ha añadido un label para cada uno de los vértices. Además, contiene una serie de métodos para obtener información sobre los nodos, como por ejemplo, comprobar la cantidad de vecinos para un vértice concreto.

- **Clase Edge**. Esta clase contiene el constructor para crear los enlaces existentes en el escenario, a partir de dos vértices y el peso del enlace. Además, contiene funciones para comparar enlaces en función del peso de cada uno de ellos, necesario para obtener el camino más corto.

- **Clase Graph**. Esta clase contiene el grafo de la topología, a partir del uso de dos tipos de mapas, uno de ellos contiene cada uno de los vértices en función de un *label* único, mientras el otro contiene la lista de enlaces junto con los vértices que forman cada uno de ellos. Además, la clase *Graph* contiene las funciones para añadir o eliminar vértices y enlaces al grafo.

- **Clase Dijkstra**. Esta clase contiene un constructor que inicializa un objeto *Dijkstra* en función de un grafo y un vértice inicial (sería el nodo origen) ejecutando el algoritmo *Dijkstra* obteniendo los caminos más cortos hacia todos los nodos desde ese nodo origen. Además, contiene funciones para encontrar el *path* y la distancia mínima hacia un nodo destino.

7.1.4. Creación del árbol de distribución

Una vez se tiene conocimiento del camino que tiene que seguir el tráfico multicast, en función de la cantidad de elementos de la ruta se crean las distintas entradas en la tabla de flujo, en cada uno de los switch que pertenecen al *path* en cuestión. Para ello, se hace uso de una función llamada *modifyL3FlowSeveralOutputConnectors()*, la cual añade, elimina o modifica las entradas de las tablas de flujo, en función de unos parámetros de entrada. Estos parámetros son la dirección del grupo multicast con el que se hace el *matching* del tráfico, el identificador del nodo en el cual se realiza la modificación, junto con dos listas en formato "$openflow : X : Y$" que contienen las entradas a modificar o eliminar.

El primer procedimiento que realiza esta función es crear un *matchBuilder* con la dirección del grupo multicast. Para la creación del *matching* en cada *switch*, existen varias opciones. La opción por defecto es mapnear las rutas en función de la MAC destino, pero se considero de interés que en las tablas de flujo aparezcan las entradas en función de la dirección IP del grupo multicast, así que se modificó el código para mapear las rutas en función de la dirección del grupo. Para ello, se tiene que crear una variable que contenga la dirección multicast del grupo de tipo **Ipv4Prefix**. La clase

IPv4-Prefix representa una dirección IPv4 formada por 32 bits que contiene una máscara específica.

A partir del *matchBuilder* se define un flujo específico que será el que se añada en las tablas de los *switches*. Al flujo creado se le pueden añadir distintos identificadores, junto con prioridades y otras características, con el objetivo de diferenciarlo de otros (véase Figura 7.9).

```
//Create match object
 MatchBuilder matchBuilder = new MatchBuilder();
 MatchUtils.createDstL3IPv4Match(matchBuilder, new Ipv4Prefix(GroupIp));


    //Create flow
    FlowBuilder flowBuilder = new FlowBuilder();
    flowBuilder.setMatch(matchBuilder.build());
    String flowId = "L3_MULTICAST_FUENTE_H"+host+" "+GroupIp;
    flowBuilder.setId(new FlowId(flowId));
    FlowKey key = new FlowKey(new FlowId(flowId));
    flowBuilder.setBarrier(true);
    flowBuilder.setTableId((short)0);
    flowBuilder.setKey(key);
    flowBuilder.setPriority(32768);
    flowBuilder.setFlowName(flowId);
    flowBuilder.setHardTimeout(0);
    flowBuilder.setIdleTimeout(0);
```

Figura 7.9: Código OpenDaylight.

A continuación, se comprueban las entradas de la tabla para no añadir entradas que ya existan en ella. Además, se obtiene el contenido de las listas que contienen los distintos parámetros para eliminar o añadir entradas en las tablas, en función del contenido de las variables se aplicaran las acciones pertinentes para añadir las nuevas entradas Figura (7.10). Existe la posibilidad de que alguna de las dos listas esté vacía, por ejemplo en el caso de que se vaya a añadir una nueva ruta, en este caso no es necesario descartar ninguna entrada; por lo tanto, la lista de entradas a eliminar estará vacía.

```
//CHECK OUTPUT CONNECTORS TO REMOVE
List<Uri> egressNodeConnectorsToRemoveUri = new ArrayList<Uri>(); // Casting NodeConnectorId to Uri
for (NodeConnectorId tmpNodeConnectorId : egressNodeConnectorsToRemove) {
    egressNodeConnectorsToRemoveUri.add(new Uri(tmpNodeConnectorId)); }

List<NodeConnectorId> existingEgressNodeConnectors = new ArrayList<NodeConnectorId>();
for (Action existingAction : existingActionList) {
    if (existingAction.getAction() instanceof OutputActionCase) {
        OutputActionCase opAction = (OutputActionCase)existingAction.getAction();
        tmpOutputNodeConnectorUri = opAction.getOutputAction().getOutputNodeConnector();

        existingEgressNodeConnectors.add(new NodeConnectorId(tmpOutputNodeConnectorUri));

        if (egressNodeConnectorsToRemoveUri.contains(tmpOutputNodeConnectorUri)) {

        } else {
            tmpOutputNodeConnectorId = new NodeConnectorId(tmpOutputNodeConnectorUri);
            finalEgressNodeConnectors.add(tmpOutputNodeConnectorId);
        }
    }
}
//CHECK OUTPUT CONNECTORS TO ADD
for (NodeConnectorId tmpNodeConnector : egressNodeConnectorsToAdd) {
    if (finalEgressNodeConnectors.contains(tmpNodeConnector)) {

    } else {
        finalEgressNodeConnectors.add(tmpNodeConnector);

    }
}
```

Figura 7.10: Código OpenDaylight.

Finalmente, se aplican las acciones extraídas de las listas, permitiendo el intercambio de tráfico multicast entre la fuente y los clientes del servicio. De igual manera, cuando uno de los clientes se da de baja de un grupo, se procede a la eliminación de las rutas creadas.

En resumen, en la Figura (7.11) se puede apreciar un diagrama de estados en el que se muestra todo el procesado llevado a cabo por el controlador. Cuando el controlador recibe un paquete, comprueba si es tipo IGMP, si es así, en base al campo *Record type* añade o elimina entradas en las tablas en función del camino devuelto por el algoritmo *Dijkstra*, transmitiendo entonces mensajes tipo $FLOW\_MOD$ con las acciones a modificar en los distintos *switches* destino.

En cambio, en el caso de que el paquete recibido no sea de tipo IGMP, se lleva a cabo el *learning-switch* del Capítulo 5 modificado para su funcionamiento para cualquier tipo de topología, así el controlador procede a buscar un *matching* en sus tablas en busca del destino del paquete, si no existe ninguna entrada para el destino en cuestión, se realiza un *flooding* hacia todos los *switches* de la red, de esta forma se realiza el aprendizaje de las rutas hacia todos los host de la topología en cuestión.

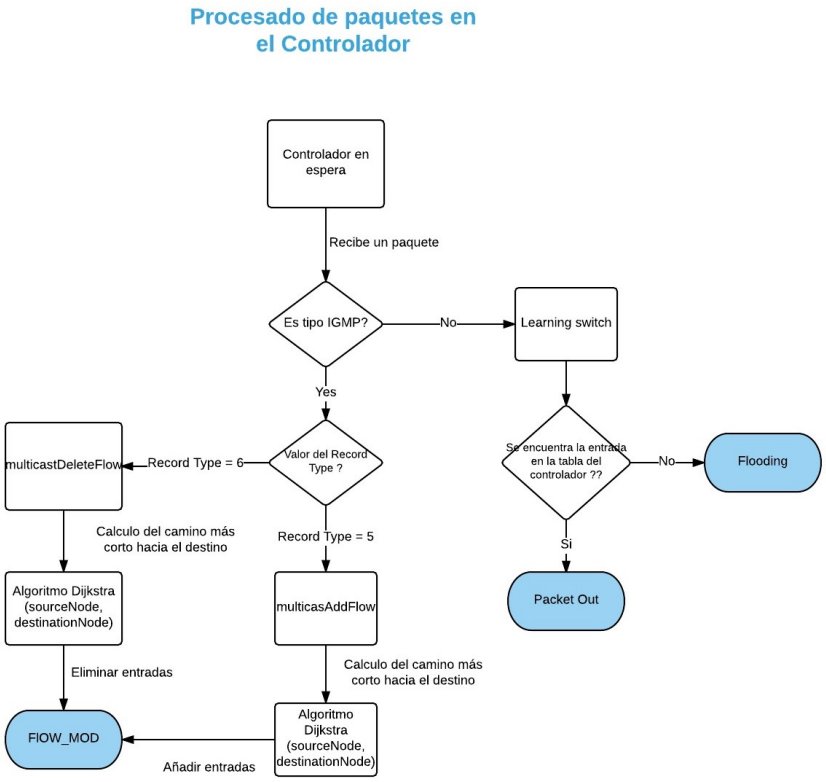


Figura 7.11: Diagrama de estados controlador.

En la siguiente sección, se evalúa experimentalmente el código implementado. Para ello, se harán las mismas simulaciones que en el Capítulo VI, de esta forma se pretende comprobar que el desarrollo del controlador se ha realizado correctamente.

7.2. Evaluación del protocolo Multicast

De la misma forma que en el Capítulo VI, en esta sección se procede a evaluar la implementación del controlador desarrollado. Para ello, se decide que el *host-1* será el servidor de contenidos multimedia, mientras el *host-2* y el *host-4* harán una petición para recibir tráfico multicast de la fuente.

Así, primero se inicia el controlador en modo *debug* para observar qué eventos ocurren en él. Esta aproximación permite comprobar cómo son procesados los paquetes cuando son recibidos en el controlador. Tras iniciar el controlador, emulamos la topología creada con Mininet:

```
# python topo-sdn.python
```

Una vez esté todo en funcionamiento, se añaden las entradas de flujo que indican qué todos los paquetes IGMP sean encaminados al controlador (véase Figura 7.12

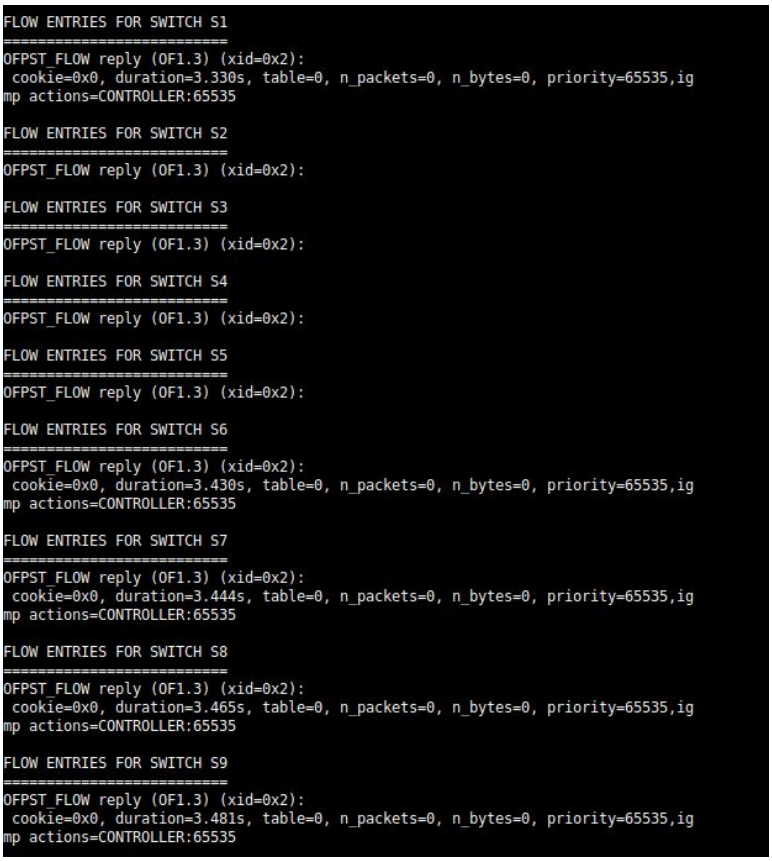


Figura 7.12: Regla igmp.

Para reducir la carga que genera el procesado de paquetes en el controlador, se han añadido las rutas por defecto unicast. De esta forma no se tienen problemas en la limitación debida al uso de un PC sin grandes prestaciones. Así, se han creado varios archivos .sh ejecutables que encaminan el tráfico *unicast* a través de la red en función de las direcciones MAC origen y destino.

Una vez añadidas las reglas, con el uso del comando **"xterm"** se abren los terminales de los *hosts* interesados para el intercambio de tráfico multicast. Primero, se inicia el proceso en segundo plano (o *demonio*) **ssmpingd** en el *host-1* provocando que el host se quede en espera hasta que reciba alguna petición. A continuación, se accede al *host-2* y se realiza una petición

hacia el *host-1*:

```
# ssmping 10.0.0.1
```

Tras realizar la petición, se tiene que esperar a que el controlador procese el paquete y realice las acciones pertinentes. Así se puede observar cómo llegan varias peticiones hacia la fuente hasta que se empieza a encaminar el tráfico multicast a través de la red hasta llegar al *host-2* enviado por el *host-1* (véase Figura 7.13).

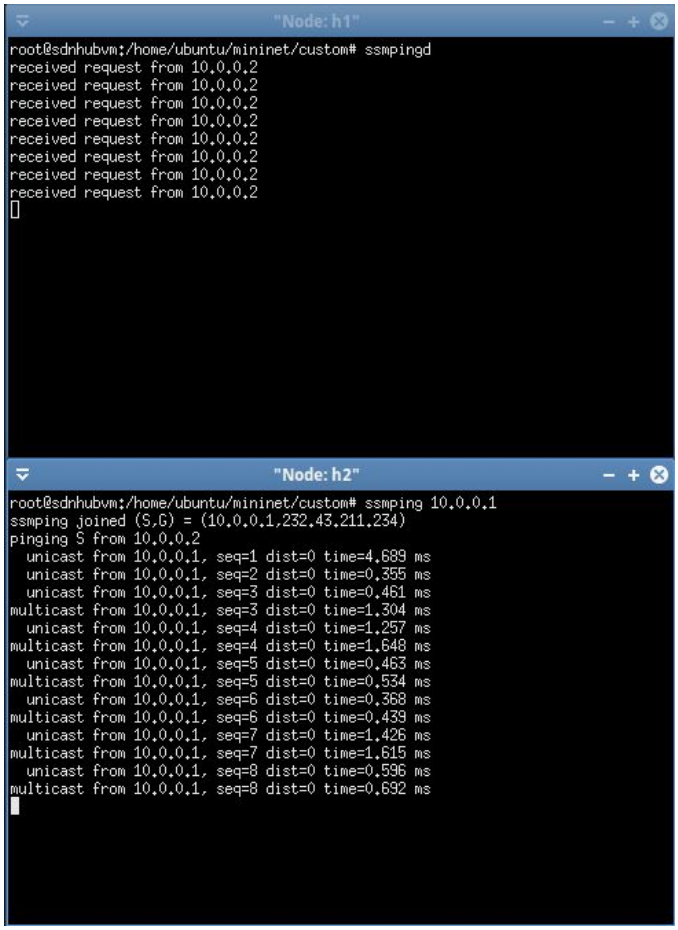


Figura 7.13: Intercambio de tráfico multicast.

Si se comprueban las tablas de flujo de la red (Figura 7.14), se puede observar cómo se han creado las entradas en los diferentes nodos pertenecientes al camino más corto hasta el cliente. Además, se puede ver la cantidad de bytes transmitidos por cada regla, comprobando de esta forma que el tráfico se está transmitiendo correctamente y no se devuelven los paquetes al

controlador.

```
FLOW ENTRIES FOR SWITCH S1
========================
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=275.474s, table=0, n_packets=0, n_bytes=0, priority=65535,igmp actions=CONTROLLER:65535
 cookie=0x0, duration=98.039s, table=0, n_packets=47, n_bytes=3760, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 acti
ons=output:2
 cookie=0x0, duration=98.022s, table=0, n_packets=47, n_bytes=3760, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 acti
ons=output:1
 cookie=0x0, duration=93.438s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=
output:3
 cookie=0x0, duration=93.419s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=
output:1
 cookie=0x0, duration=93.370s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:04 actions=
output:4
 cookie=0x0, duration=93.363s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:01 actions=
output:1
 cookie=0x0, duration=93.326s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:05 actions=
output:5
 cookie=0x0, duration=93.319s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:05,dl_dst=00:00:00:00:00:01 actions=
output:1
 cookie=0x0, duration=46.023s, table=0, n_packets=45, n_bytes=3600, ip,nw_dst=232.43.211.234 actions=output:2
```

(a) Tabla de flujos Switch-1

```
FLOW ENTRIES FOR SWITCH S2
========================
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=98.043s, table=0, n_packets=47, n_bytes=3760, dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 acti
ons=output:2
 cookie=0x0, duration=98.013s, table=0, n_packets=47, n_bytes=3760, dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:01 acti
ons=output:1
 cookie=0x0, duration=46.050s, table=0, n_packets=45, n_bytes=3600, ip,nw_dst=232.43.211.234 actions=output:2
```

(b) Tabla de flujos Switch-2

```
FLOW ENTRIES FOR SWITCH S6
==========================
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=42.805s, table=0, n_packets=2, n_bytes=140, priority=65535
,igmp actions=CONTROLLER:65535
 cookie=0x0, duration=40.234s, table=0, n_packets=10, n_bytes=800, dl_src=00:00:
00:00:00:01,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=40.224s, table=0, n_packets=10, n_bytes=800, dl_src=00:00:
00:00:00:02,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=37.513s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:02,dl_dst=00:00:00:00:00:03 actions=output:1
 cookie=0x0, duration=37.505s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:03,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=37.433s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:02,dl_dst=00:00:00:00:00:04 actions=output:1
 cookie=0x0, duration=37.428s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:04,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=37.380s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:02,dl_dst=00:00:00:00:00:05 actions=output:1
 cookie=0x0, duration=37.374s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:
00:00:05,dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=8.355s, table=0, n_packets=9, n_bytes=720, ip,nw_dst=232.4
3.211.234 actions=output:2
```

(c) Tabla de flujos Switch-6

Figura 7.14: Tabla de flujos.

Si se realiza una nueva petición desde otro cliente, por ejemplo desde el *host-4* y se observan de nuevo las tablas de flujo, se puede ver cómo se han creado nuevas entradas en las tablas y cómo en el *switch-1*, que pertenece a las dos rutas hacia los *hosts*, se ha modificado la entrada existente siendo ahora la salida a varios puertos en vez de uno (Figura (7.15)). Además, se puede observar como el tráfico sigue intercambiándose correctamente desde la fuente hacia los dos clientes (véase Figura 7.16).

```
FLOW ENTRIES FOR SWITCH S1
==========================
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=182.864s, table=0, n_packets=0, n_bytes=0, priority=65535,igmp actions=CONTROLLER:6
5535
 cookie=0x0, duration=180.317s, table=0, n_packets=149, n_bytes=11920, dl_src=00:00:00:00:00:01,dl_dst=00
:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=180.305s, table=0, n_packets=149, n_bytes=11920, dl_src=00:00:00:00:00:02,dl_dst=00
:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=177.709s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:01,dl_dst=00:00:00
:00:00:03 actions=output:3
 cookie=0x0, duration=177.698s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:03,dl_dst=00:00:00
:00:00:01 actions=output:1
 cookie=0x0, duration=177.650s, table=0, n_packets=12, n_bytes=960, dl_src=00:00:00:00:00:01,dl_dst=00:00
:00:00:00:04 actions=output:4
 cookie=0x0, duration=177.645s, table=0, n_packets=12, n_bytes=960, dl_src=00:00:00:00:00:04,dl_dst=00:00
:00:00:00:01 actions=output:1
 cookie=0x0, duration=177.610s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:01,dl_dst=00:00:00
:00:00:05 actions=output:5
 cookie=0x0, duration=177.605s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:05,dl_dst=00:00:00
:00:00:01 actions=output:1
 cookie=0x0, duration=177.225s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:02,dl_dst=00:00:00
:00:00:03 actions=output:3
 cookie=0x0, duration=177.216s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:03,dl_dst=00:00:00
:00:00:02 actions=output:2
 cookie=0x0, duration=177.208s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:02,dl_dst=00:00:00
:00:00:04 actions=output:4
 cookie=0x0, duration=177.197s, table=0, n_packets=0, n_bytes=0, dl_src=00:00:00:00:00:04,dl_dst=00:00:00
:00:00:02 actions=output:2
 cookie=0x0, duration=148.407s, table=0, n_packets=160, n_bytes=12800, ip,nw_dst=232.43.211.234 actions=o
utput:2, output:4
```

Figura 7.15: Tabla de flujo Switch-1.

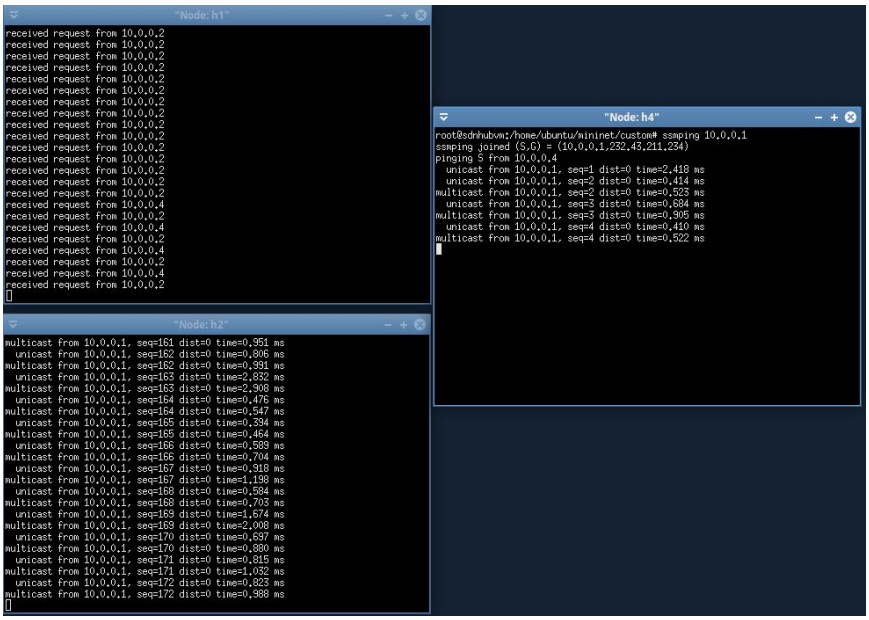


Figura 7.16: Intercambio de tráfico multicast.

Finalmente, si se observa el terminal donde ha sido iniciado el controlador en modo *debug*, se puede observar el *LOG* que muestra todos los procesos realizados por el controlador, indicando cómo se procesan y se van creando y eliminando las rutas en función de los paquetes recibidos en el controlador (véase Figura 7.17).

```
     | 189 - org.sdnhub.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | CS
e>> MODIFYING L3 flow in node openflow:1 with IP Ipv4Prefix [_value=232.43.211.2
34/32] and several output connectors: initial connectors = [], connectors to add
 = [Uri [_value=openflow:1:2]], connectors to remove = [] -> final output connec
tors = [Uri [_value=openflow:1:2]]
```

(a) Adición de una entrada

```
     | 189 - org.sdnhub.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | CS
e>> MODIFYING L3 flow in node openflow:1 with IP Ipv4Prefix [_value=232.43.211.2
34/32] and several output connectors: initial connectors = [Uri [_value=openflow
:1:2]], connectors to add = [Uri [_value=openflow:1:4]], connectors to remove =
[] -> final output connectors = [Uri [_value=openflow:1:2], Uri [_value=openflow
:1:4]]
```

(b) Adición de una entrada con dos puertos

```
     | 189 - org.sdnhub.odl.tutorial.learning-switch.impl - 1.0.0.SNAPSHOT | CS
e>> MODIFYING L3 flow in node openflow:1 with IP Ipv4Prefix [_value=232.43.211.2
34/32] and several output connectors: initial connectors = [Uri [_value=openflow
:1:2], Uri [_value=openflow:1:4]], connectors to add = [], connectors to remove
= [Uri [_value=openflow:1:4]] -> final output connectors = [Uri [_value=openflow
:1:2]]
```

(c) Eliminación de una entrada

Figura 7.17: LOG de OpenDaylight.

Estudiando las salidas del controlador, se comprueba que las entradas se eliminan correctamente. En cambio tras ver las tablas de flujo, se observó que no se eliminan correctamente. Tras realizar varios análisis de la programación desarrollada, junto con el estudio de las salidas producidas por el modo *debug* del controlador, se llegó a la conclusión que el problema era de la propio de la distribución de OpenDaylight. Esto se puede deber a los pocos proyectos en los cuales se añadan reglas multipuerto usando esta tecnología, por lo que es posible que sea un error no especificado actualmente. Además, OpenDaylight es una plataforma en desarrollo que no está exenta de *bugs* que se van solventando en cada actualización de la plataforma.

Localizado el problema, se comprobó que el controlador volvía a añadir las entradas que habían sido introducidas en último lugar. Para solucionarlo, se decidió añadir reglas con puertos inexistentes, de tal forma que se sobrescribían los puertos a eliminar, solucionando de esta forma la problemática encontrada.

7.2.1. Análisis de los escenarios desarrollados

Una vez desarrollado un escenario basado en SDN que permite el intercambio de tráfico multicast, es de interés comprobar la cantidad de tráfico transmitido en los escenarios implementados. Es decir, en un escenario con el uso del protocolo PIM-SSM y en un escenario usando un controlador SDN.

Para realizar este análisis se han producido capturas *wireshark* en cada

uno de los elementos de la red, comprobando el tráfico de señalización multicast producido durante determinados instantes de tiempo. Para analizar los datos se han tenido en cuenta varias aproximaciones:

- No se ha tenido en cuenta el tráfico provocado por los demonios Zebra y Qpimd, debido a que en una red real no sería necesario el uso de estos demonios para la implementación de PIM-SSM, por lo que sólo se han medido los datos referentes al protocolo anterior.
- Mientras que PIM-SSM es un protocolo especifico para el intercambio de tráfico multicast, el controlador **OpenDaylight** está diseñado para la gestión completa de la red. Es decir, no solo debe permitir el intercambio multicast, sino que engloba todos los protocolos o métodos que una red necesita para su correcto funcionamiento, ya sea controlando la congestión de la red u obteniendo las mejores rutas a través del algoritmo *Dijkstra*.
- Teniendo en cuenta el anterior punto, se ha despreciado todo el tráfico producido por el controlador para el control y gestión de la red, ya que no es un tráfico provocado por la implementación desarrollada. Es decir, para tener en consideración estos datos se tendría que tener en cuenta dos redes que contengan todos los protocolos que se utilizan actualmente en la red para evaluar la carga del controlador en una red real.

Tras aclarar las aproximaciones realizadas, en la Figura (7.18) se muestra la cantidad de tráfico producido por la señalización multicast durante diferentes instantes de tiempo, para una fuente activa y un único cliente. Se puede observar como la cantidad de tráfico en el escenario con PIM-SSM aumenta con el paso del tiempo, esto se debe al intercambio de mensajes tipo *PIM HELLO* cada 30 segundos, para comprobar periódicamente los vecinos con capacidad PIM a su alrededor, provocando que haya un constante envío de tráfico en la red, además tras un minuto se vuelven a transmitir los mensajes *join/prune* para confirmar la existencia del árbol.

En cambio, con el uso de OpenDaylight relacionado con la transmisión multicast a través de la red solo se transmiten los paquetes tipo *FLOW_MOD*, que contienen las acciones pertinentes a realizar en cada uno de los *switches* de la red.

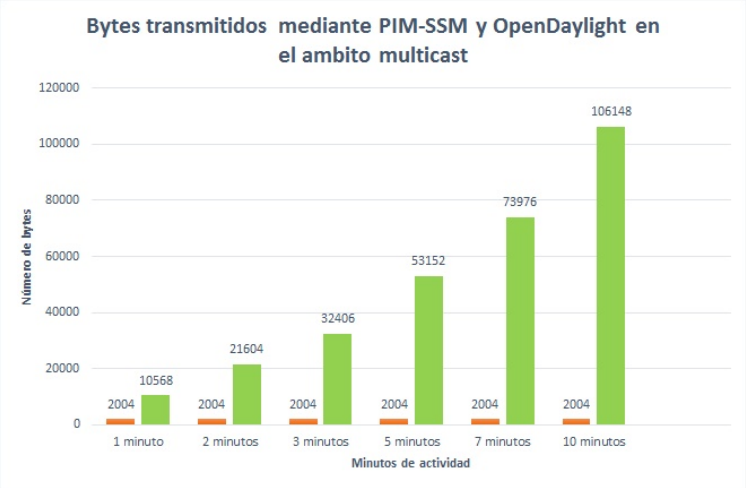


Figura 7.18: Análisis de escenarios.

Por lo tanto, aunque sea una evaluación aproximada, se demuestra que el uso de SDN provocaría las mejoras necesarias en la red actual, en referencia a la liberación de tráfico en la red, sin tener en cuenta las mejoras en las prestaciones de gestión y flexibilidad para los administradores, no solo en lo referente a multicast, sino en un ámbito completo de la red .

Capítulo 8

Conclusiones y vías futuras

En este último capítulo se recogen todas las reflexiones finales obtenidas tras la realización del proyecto. Primero se indicarán unas conclusiones globales, donde se mostrarán los objetivos alcanzados, junto con las limitaciones y problemas encontrados durante su desarrollo.

A continuación, se proponen posibles lineas futuras a desarrollar tras la realización del TFG, a partir del conocimiento adquirido durante su elaboración. En esta sección, se comentarán algunas de las posibles mejoras que se pueden realizar en un futuro.

Finalmente, se incluyen unas breves valoraciones personales del autor del presente proyecto, con el objetivo de dar a conocer todos los beneficios personales obtenidos a lo largo de su desarrollo.

8.1. Conclusiones

En este proyecto, se ha realizado una conceptualización del protocolo multicast PIM-SSM sobre una red emulada basada en Mininet junto con el uso de la *suite* de *routing* Quagga. De esta forma, se ha podido estudiar cómo se transmite el tráfico multicast (tanto los datos como la señalización asociada) en un escenario real, identificando qué tipo de mensajes se intercambian, la formación de árboles, junto con todos los protocolos involucrados en la transmisión multicast.

Además, el estudio del esquema PIM-SSM, se procedió a la implementación de una solución multicast para un escenario basado en una arquitectura SDN. Para ello, se hizo uso de OpenDaylight, un controlador SDN, junto con Mininet para emular un escenario ilustrativo. De este modo, se programó el controlador en cuestión, con el objetivo de añadirle capacidad para procesar tráfico multicast a través de la red, permitiendo diferenciar los distintos

tipos de peticiones de los clientes a través del protocolo IGMPv3 y por tanto, pemitiendo modificar las distintas entradas en las tablas de flujo de los conmutadores en función de las mejores rutas.

Finalmente, se ha realizado una evaluación de los dos escenarios implementados con el objetivo de obtener datos de interés que demuestren las ventajas de las redes SDN frente a las redes de la actualidad. Así, se ha comprobado como se produce una reducción en la cantidad de tráfico necesario para la transmisión de datos multicast a través de la red.

Las principales contribuciones de este proyecto son:

- Se ha documentado y recopilado información relativa a las redes SDN, identificando las ventajas que pueden ofrecer estas nuevas arquitecturas emergentes en el ámbito del *networking* actual. Además, se ha motivado a los desarrolladores de esta tecnología para continuar formándose en esta área con el objetivo de obtener unas redes futuras más flexibles y versátiles.

- Para la emulación de redes reales se ha hecho uso de Mininet, así se ha ofrecido en este TFG las nociones mínimas necesarias para el inicio en el uso de esta herramienta, no solo en el ámbito que este proyecto desarrolla, sino identificando todas las posibilidades que este software ofrece para cualquier tipo de proyecto relacionado con redes.

- Estudio de la *suite* de *routing* Quagga para realizar un estudio del protocolo multicast PIM-SSM. Se ha experimentado con esta herramienta observando cómo funciona un protocolo multicast en un entorno emulado. Además, se han identificado todas las ventajas que ofrece este software, no solo en la utilización de PIM-SSM, sino también en el uso de otros protocolos importantes como OSPF o incluso RIP.

- Se ha hecho uso de un controlador OpenDaylight para desarrollar en él capacidad para procesar el tráfico multicast. Así, el controlador implementado permite el intercambio de tráfico multidifusión gracias a un procesado de las peticiones IGMP de los clientes interesados en algún tipo de grupo multicast. De esta forma, el controlador se encarga de crear las rutas necesarias en función del camino más corto calculado a través del algoritmo *Dijkstra*, con el objetivo de hacer llegar los paquetes a su destino. Además se ha incluido la gestión de la poda del árbol, eliminando las entradas pertinentes cuando los clientes decaen en su interés en una fuente dada.

- Se ha adoptado una metodología en bucle cerrado que ha permitido introducir mejoras en la implementación. Particularmente se ha mejorado el análisis ("*parsing*") de los paquetes recibidos por el controlador. Así, se han desarrollado nuevos métodos que permiten obtener

información vital para el procesado de los paquetes proveniente de los mensajes de tipo IGMP.

En resumen, se ha realizado un estudio de los protocolos multicast sobre redes SDN, identificando las ventajas que ofrecen, ya que, cómo se comentó en el Capítulo I serán vitales para solucionar los problemas actuales y futuros en el ámbito de la transmisión de datos multimedia.

8.2. Problemas y limitaciones encontrados

Durante el transcurso del proyecto han surgido distintos problemas o limitaciones que han causado la ralentización del avance del mismo. A continuación, se enumeran alguno de ellos:

- Una de las principales limitaciones ha sido el tiempo que tardaba en compilar el controlador. Cada vez que se realizaba cualquier mínimo cambio en el código, implica un periodo de aproximadamente 15 minutos para realizar la compilación. Este hecho ha implicado un retraso considerable a la hora de la programación del controlador.

- Otra problemática se encuentra cuando se realiza una segunda prueba con el controlador con una topología distinta (otro tipo de escenario). En estos casos, el controlador mantenía los flujos instalados anteriormente, lo que obligaba a compilar de nuevo todo el proyecto. Esto es debido a que el controlador esta diseñado para redes reales, suponiendo que en el caso de que deje de funcionar, durante unos instantes este mantenga la programación existente en la red.

- Sin duda una de las limitaciones más importantes, han sido las impuestas por las elevadas demandas de las herramientas de emulación utilizadas en el propio equipo utilizado. Aún siendo un portátil con procesador intel core i-5 y 4GB de memoria RAM, a la hora de ampliar las topologías o incluso cuando se intercambian muchos mensajes a través de la red emulada, lo habitual es que aparezcan problemas de ejecución dentro del propio ordenador.

- Durante el transcurso del proyecto se comprobó que OpenDaylight no gestiona correctamente la eliminación de las entradas, en las tablas de flujo en el caso de reglas multi-puerto. Para solucionar esta problemática, se decide la creación de reglas con puertos virtuales, provocando la correcta eliminación de las entradas en los distintos *switches* de la red.

- Otro de los problemas encontrados -que se ha solventado adecuadamente- ocurrió durante la instalación de Quagga junto con Mininet. Inicialmente Quagga no soportaba todas las versiones de Mininet, así ocurrían errores sin solución. La resolución de estos problemas ha implicado un tiempo no despreciable hasta descubrir el por qué de estos errores. Por lo tanto para que se cumplieran las especificaciones, se tuvo que reducir la versión instalada en la maquina virtual.

- Finalmente, al iniciar el controlador OpenDaylight e iniciar la topología en algunos casos esta no se conectaba correctamente. Tras usar el modo *debug* se observó que aunque el controlador esté iniciado necesita un tiempo no despreciable hasta inicializarse completamente.

8.3. Vías futuras

Tras la realización del proyecto fin de grado se han conseguido cumplir con todos los objetivos propuestos al inicio del mismo. Aún así, durante el transcurso del proyecto han surgido unas posibles lineas futuras para la mejora o continuación en el ámbito que este TFG desarrolla.

Estas vías futuras se resumen en los siguientes puntos:

- Se ha conseguido que el controlador pueda procesar tráfico tipo multicast sobre cualquier tipo de topología. Una posible línea de trabajo sería la evaluación del controlador desarrollado en diversos diseños de red valorando como aumenta la carga de red, en función del número de elementos, en comparación con una red sin controlador.

- Se ha utilizado el algoritmo *Dijkstra* para la obtención del camino más corto hacia la fuente, a la hora de crear un árbol de distribución multicast. Una posible mejora sería crear un algoritmo propio, que no se basara en la distancia más corta de un nodo origen hacia un nodo destino, sino que las modificaciones de los árboles aprovechen las rutas ya creadas conectando los nuevos miembros a las intersecciones más cercanas.

- Todas pruebas y simulaciones se han realizado en entornos emulados. Sería de interés introducir el controlador implementado en un escenario real, de forma que se puedan realizar una evaluación cuantitativa de lo aquí propuesto en un entorno en explotación. Esto permitiría hacer una evaluación del impacto real de lo propuesto en comparación con el estado del arte actual.

- Sin duda una de las vías futuras más interesantes sería realizar una evaluación sobre redes de gran tamaño, actualmente limitada por las

demandas de las herramientas utilizadas. En este caso, se podría apreciar realmente a gran escala la diferencia a nivel de utilización de red, entre ambos aproximaciones (con SDN y sin SDN) durante el transcurso de la etapa de creación de los árboles de distribución. Además, esto permitiría evaluar la celeridad con la que son modificadas las distintas intersecciones de los árboles tras una baja de un cliente en comparación con la aproximación de referencia.

8.4. Valoración personal

Para finalizar con la elaboración de la memoria en esta sección se exponen una serie de reflexiones y valoraciones personales tras la finalización del presente proyecto.

Sin ningún lugar a duda el TFG es el punto final de un camino que ha durado 4 años, que finaliza con la obtención del Grado en Tecnologías de Telecomunicación. Este proyecto tiene -entre otros- como objetivo poner en práctica y usar todos los conocimientos y aptitudes adquiridas a lo largo del transcurso de la titulación, intentando mostrar el nivel de madurez adquirido tras 4 años de estudio.

A nivel personal, se han logrado todos y cada uno de los objetivos propuestos en el anteproyecto, resolviendo cualquier problemática surgida y creciendo profesionalmente día tras día.

La realización del trabajo fin de grado, cumple todos los requisitos que se necesitarán en los siguientes caminos tomados; constancia, esfuerzo y paciencia. Se han tenido que hacer frente a situaciones comunes en el mundo laboral: se han resuelto problemas sin tener "*a priori*" ninguna solución, fechas de corte o incluso limitaciones en lo referente a material, pero sin ningún lugar a duda, una de las lecciones aprendidas tras la finalización del presente proyecto, es la importancia de la planificación. Se ha demostrado que es vital e implica el primer paso hacia la resolución de cualquier tipo de proyecto.

Para la elaboración del proyecto han sido necesarios múltiples conocimientos. Se ha realizado un estudio de los protocolos multicast, tecnología de la que se partía con un conocimiento muy básico, lejos del necesario para afrontar con éxito el problema abordado. También, ha sido necesario aprender el lenguaje python y ampliar el conocimiento en Java. Se ha aprendido a utilizar Mininet, una herramienta muy útil para cualquier tipo de proyecto en el que sea necesario el uso de emular topologías concretas para acercarse lo máximo posible a redes reales. Además, se ha usado un controlador OpenDaylight, estudiando cómo se programa en bajo nivel y los beneficios que puede ofrecer a todo tipo de redes SDN. para todo ello ha sido necesaria

una adecuada planificación y mucha constancia, para hacer frente a todos los retos planteados, adquiriendo en el camino nuevos conocimientos necesarios para el correcto desarrollo de este TFG.

Finalmente, destacar la satisfacción alcanzada tanto a nivel personal como profesional. Se ha sido capaz de abordar con éxito un proyecto desafiante sobre una tecnología actual, pese a la poca documentación existente. Todo esto se debe al esfuerzo constante realizado para la comprensión e investigación precedente al desarrollo del controlador. Sin duda, ha sido un proyecto enriquecedor en todos los aspectos posibles.

Apéndice A

Manual de instalación.

En este apéndice se indica todo el proceso necesario para la correcta instalación de los distintas herramientas software usadas en el desarrollo del proyecto. El objetivo principal de este apéndice es ofrecer la posibilidad de realizar pruebas para uso personal en el ámbito que este TFG desarrolla.

A continuación, se muestran tres apartados principales donde se aborda el procedimiento a seguir en la instalación de las tres herramientas básicas utilizadas, Mininet, Quagga y OpenDaylight, además se resuelven los problemas que pueden surgir durante su instalación.

A.1. Mininet.

Existen cuatro opciones a la hora de instalar Mininet [70]. A continuación se exponen los cuatro métodos.

Opción 1: Mininet VM Instalación

Es la opción recomendada por Mininet, únicamente se tiene que descargar la imagen de maquina virtual ofrecida por Mininet en [11]; tras la descarga, solo se tiene que hacer uso de un sistema de virtualización como *Virtual Box* o *VMware*.

Opción 2: Instalación nativa

Esta opción asume que se empieza desde un punto de partida nuevo, es decir, una maquina virtual Ubuntu [30] limpia. Se recomienda el uso de la versión de Ubuntu más reciente.

Para instalar de forma nativa desde la fuente, primero se necesita obtener el código fuente, para ello hay que descargar el repositorio github:

```
$ git clone git://github.com/mininet/mininet
```

Este comando descarga la versión más reciente de mininet. Para la instalación se debe entrar en el directorio raíz de mininet y una vez en él, se puede proceder a la instalación de mininet:

```
$ mininet/util/install.sh [options]
```

Existen varias opciones para realizar la instalación; estas se pueden consultar en [70], todas ellas hacen referencia a la inclusión de distintos paquetes en la instalación.

Opción 3: Instalación a partir de paquetes

Si se está ejecutando una versión reciente de Ubuntu, se pueden instalar los paquetes de mininet, aunque esta opción puede ofrecer una versión anterior de mininet. Primero, hay que borrar cualquier rastro de instalaciones anteriores, para ello:

```
$ sudo rm -rf /usr/local/bin/mn /usr/local/bin/mnexec
/usr/local/lib/python*/*/*mininet* /usr/local/bin/ovs-*
/usr/local/sbin/ovs-*
```

Tras realizar esto, hay que instalar el paquete mininet en función de la distribución de Ubuntu que se utilice, en este caso Ubuntu 14.10:

```
$ sudo apt-get install mininet
```

Opción 4: Actualizar una versión existente

Para actualizar la versión de mininet solo hay que hacer uso de los siguientes comandos:

```
cd mininet
git fetch
git checkout master
git pull
sudo make install
```

Cabe destacar que este método solo actualiza la versión de mininet, otros componentes como OpenvSwitch, deben de ser actualizados por separado.

A.2. Quagga

La instalación de Quagga y su parcheado para el uso del demonio Qpimd tiene bastante complejidad; esto es debido a que las instrucciones ofrecidas en el *Readme* son erróneas. Tras una búsqueda exhaustiva se encontró la solución en las listas gnu [8].

Primero hay que descargar Quagga y Qpimd:

```
wget http://www.quagga.net/download/quagga-0.99.17.tar.gz
```

```
wget http://download.savannah.gnu.org/releases/qpimd/qpimd-0.162.tar.gz
```

Cuando se hayan descargado los archivos, se procede a descomprimirlos. Una vez estén descomprimidos hay que aplicar el parche:

```
patch -p1 -d quagga-0.99.17 < qpimd-0.162/pimd-0.162-quagga-0.99.17.patch
```

En este paso hay que tener cuidado, debido a que no todas las versiones de quagga soportan las distintas versiones de qpimd y viceversa. Tras aplicar el parche, se accede a la carpeta de quagga y se permite el uso de los demonios zebra y qpimd (en este caso pimd parcheado):

```
$ cd quagga-0.99.17
```

```
$ ./configure –prefix=/usr/local/quagga –enable-pimd –enable-tcp-zebra
```

A continuación, se procede a compilar y eliminar las antiguas instalaciones en el caso de que existiensen:

```
$ make
```

```
sudo mv /usr/local/quagga /usr/local/quagga.old
```

Se instalan los binarios (debajo de /usr/local/quagga) y se copian los ejemplos de la configuración:

```
$ sudo make install
```

```
cp /usr/local/quagga/etc/zebra.conf.sample /usr/local/quagga/etc/zebra.conf
```

```
cp /usr/local/quagga/etc/pimd.conf.sample /usr/local/quagga/etc/pimd.conf
```

Antes de probar su funcionamiento, se crean el usuario quagga y el grupo quagga, ya que ejecutaremos los demonios con ellos, además se establecen permisos para configurar archivos:

```
sudo groupadd quagga
```

```
sudo useradd quagga -g quagga
```

```
sudo chown -R quagga:quagga /usr/local/quagga/etc
```

Finalmente, se comprueba que todo se ha instalado correctamente poniendo en funcionamiento el demonio zebra y pimd:

```
/usr/local/quagga/sbin/zebra -u quagga -g quagga -i /usr/local/quagga/etc/zebra.pid
```

```
sudo /usr/local/quagga/sbin/pimd -i /usr/local/quagga/etc/pimd.pid
```

A.3. OpenDaylight

Para instalar OpenDaylight, como se expuso a lo largo del proyecto, se hizo uso de una maquina virtual SDN-HUB [32], en la cual ya estaban instaladas las distribuciones de OpenDaylight junto con otros servicios necesarios. Aún así, para actualizar las dependencias es recomendable realizar un *pull* de github de los archivos:

```
$ cd SDNHub_Opendaylight_Tutorial
```

```
$ git pull −−rebase
```

Para poner en marcha el controlador tenemos que entrar en el árbol de carpetas que se crea del tutorial SDN-HUB e iniciarlo:

```
$ cd SDNHub_Opendaylight_Tutorial/distribution/opendaylight/target/assembly/
```

```
$ ./bin/karaf
```

Para instalar nuevas funcionalidades de Opendaylight solo tenemos que utilizar el comando:

$$opendaylight > feature : install < feature >$$

Dentro de estas funcionalidades, destacan entre otras *odl-restconf*, habilita *MD-SA data broker* y el servicio *openflow plugin*, *odl-l2switch-switch* se asegura que la los hosts sean visibles en la topología, *odl-dlux-core* se necesita para activar el uso de la interfaz DLUX como se estudio en el Capítulo 5.

Compilación

A la hora de crear proyectos OpenDaylight se hace uso de Maven [2], Eclipse [7] y OSGi. Tal y como se comentó en capítulos anteriores, a través de maven se pueden compilar los proyectos creados en Eclipse basados en Java. Maven utiliza el archivo *pom.xml* para describir el proyecto a construir, sus dependencias con otros módulos, componentes y el orden de construcción. Cada vez que se cree un nuevo proyecto o módulo, o cualquier tipo de modificación en el proyecto, es necesario actualizar los archivos *pom.xml* y *features.xml* a través de maven en directorio donde se encuentra el proyecto:

```
$mvn clean install
```

Esta compilación necesaria, es una de las limitaciones más importantes que se han encontrado al programar el controlador, debido a que cualquier cambio mínimo obliga a compilar el proyecto entero. Este proceso, en función de las características del ordenador, puede tardar aproximadamente diez minutos, lo que puede dilatar la configuración de todas las necesidades requeridas en el controlador.

Además, OSGi permite que los distintos módulos se carguen dinámicamente de un sistema, encapsulando dichos módulos. Estos módulos se llaman *bundles* y proveen servicios a través de un entorno de configuración. Los *bundles* son básicamente archivos JAR, compuesto por java, recursos, manifiestos (pom.xml), que indican qué se ha exportado a otros paquetes, y lo que hay que importar de otros, ver Figura (A.1). Por lo tanto, realmente se implementa un *bundle* que se añade a las funcionalidades del controlador.

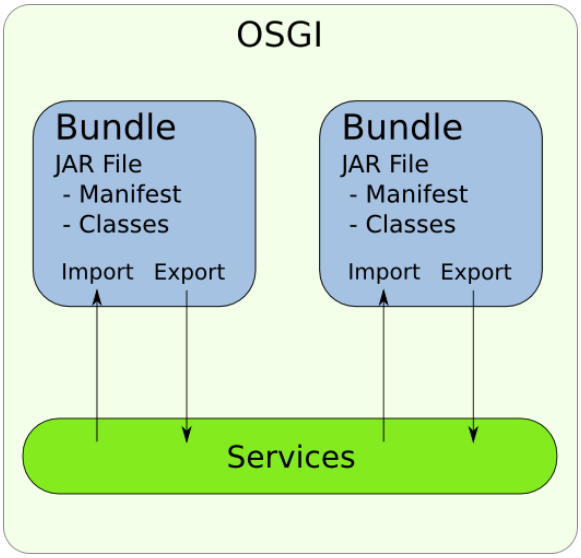


Figura A.1: Esquema OSGi [18].

Para cargar el *bundle* al controlador, una vez iniciado este, instalamos el *bundle* de igual forma que se hace con las funcionalidades:

$$opendaylight > installfile : /[rutadelarchivo]$$

Este comando devuelve un número, así solo hay que realizar un *start* hacia ese *bundle number* y se tendrán las funcionalidades implementadas iniciadas.

$$opendaylight > start(bundle\_number)$$

Apéndice B

Manual de usuario

En este apéndice se indican los pasos a seguir para arrancar los escenarios basados en la *suite* de *routing* Quagga y el escenario basado en OpenDaylight junto con las configuraciones pertinentes a establecer con el objetivo de poder realizar pruebas a partir de la implementación desarrollada.

Quagga

Para hacer uso del escenario implementado con el uso de la *suite* de *routing* Quagga, se tiene que acceder al directorio *home/mininet/custom* donde se encuentra el fichero desarrollado en python de la topología desarrollada. Una vez en la carpeta especifica, solo hay que ejecutar el fichero:

$ python topo-version.py

Para acceder a los demonios Qpimd y Zebra, se requiere realizar un *telnet* desde el terminar de uno de los *routers* de la topología. Para acceder a un terminal hay que utilizar el siguiente comando:

mininet> xterm Rx

Siendo "x" el número del router al que se accede. Los puertos donde escuchan los demonios Zebra y Qpimd son 2601 y 2611 respectivamente. Una vez se haga *telnet* a los demonios, se requiere una contraseña para acceder correctamente. La contraseñas para acceder a los distintos demonios son las siguientes:

Contraseña Zebra$---> en$

Contraseña Qpimd$---> zebra$

Finalmente, se puede realizar cualquier tipo de prueba multicast con el uso de la herramienta *ssmping*. Solo es necesario activar la fuente a través del comando "*ssmpingd*" permitiendo a los clientes clientes realizar una petición hacia la fuente para recibir los datos multicast:

```
$ ssmping DIRECCION_IP_FUENTE
```

OpenDaylight

Primero, se requiere compilar el proyecto desarrollado haciendo uso de maven. Para ello hay que acceder al directorio *SDNHub_Opendaylight_Tutorial* que se encuentra en la carpeta *home* a través de un terminal:

```
$ mvn clean install -nsu -DskipTests
```

Una vez se ha compilado el proyecto hay que acceder al directorio donde se encuentra el ejecutable para iniciar el controlador. Así, se accede a */SDNHub_Opendaylight_Tutorial/distribution/opendaylightkaraf/target/assembly* y se ejecuta el controlador:

```
$ ./bin/karaf
```

Tras arrancar el controlador, se tiene que esperar unos instantes hasta que se inicie completamente. Para cerciorarse de una correcta ejecución del controlador, el mejor método es utilizar el modo *debug* del controlador:

```
$ log:set DEBUG org.sdnhub.odl.tutorial
```

```
$ log:tail
```

A continuación, se tiene que acceder al directorio de Mininet, donde se encuentra la carpeta *custom* con el escenario creado. Así, se abre un terminal nuevo y se crea la topología:

```
$ python topo-sdn.py
```

Teniendo el controlador y la topología en funcionamiento, solo queda añadir las reglas que encaminen los paquetes hacia el controlador, para ello se hace uso de varios ficheros *sh* localizados en el mismo directorio:

```
$ add_multicast_entries_to_contoller
```

```
$ add_flow_cse
```

Para que las rutas unicast sean instaladas correctamente, es necesario realizar *Ping* entre todos los host de la red, con el objetivo de que el controlador programe las rutas en todos los elementos de la red:

```
mininet> pingall
```

Finalmente, el escenario esta configurado correctamente para realizar cualquier tipo de prueba que se desee en el ámbito que el proyecto desarrolla.

Apéndice C

Topologías en Mininet

En este apéndice se incluyen algunos script en python desarrollados a lo largo del proyecto, en concreto se muestran las dos implementaciones de topologías creadas para mininet.

C.1. Topología sin controlador

```
1  #!/usr/bin/env python
2
3
4  """ Copyright (C) 2016 Carlos Santamaria Espinosa
5
6
7  """
8
9  from mininet.topo import Topo
10 from mininet.net import Mininet
11 from mininet.log import lg, info, setLogLevel
12 from mininet.util import dumpNodeConnections, quietRun, moveIntf
13 from mininet.cli import CLI
14 from mininet.node import Switch, OVSKernelSwitch
15
16 from subprocess import Popen, PIPE, check_output
17 from time import sleep, time
18 from multiprocessing import Process
19 from argparse import ArgumentParser
20
21 import sys
22 import os
23 import termcolor as T
24 import time
25
26 setLogLevel('info')
27
```

```
28 parser = ArgumentParser("Configure a network composed of routers
       in Mininet.")
29 parser.add_argument('--sleep', default=3, type=int)
30 args = parser.parse_args()
31
32 def log(s, col="green"):
33         print T.colored(s, col)
34
35
36 class Router(Switch):
37
38         """Se define la clase router, con el objetivo de poder
               utilizar la suite de routing quagga en ellos
39
40         """
41         ID = 0
42         def __init__(self, name, **kwargs):
43         kwargs['inNamespace'] = True
44         Switch.__init__(self, name, **kwargs)
45         Router.ID += 1
46         self.switch_id = Router.ID
47
48         @staticmethod
49         def setup():
50         return
51
52         def start(self, controllers):
53         pass
54
55         def stop(self):
56         self.deleteIntfs()
57
58         def log(self, s, col="magenta"):
59         print T.colored(s, col)
60
61
62 class SimpleTopo(Topo):
63
64
65 """Se crean los routers y host a la red """
66
67         def __init__(self):
68
69         # Add default members to class.
70                 super(SimpleTopo, self ).__init__()
71                 num_routers = 9
72                 num_host=5
73                 routers = []
74                 for i in xrange(num_routers):
75                         router = self.addSwitch('R%d' %(i+1))
76                         routers.append(router)
77                         hosts=[]
78                 for i in xrange(num_host):
79                         hostname='h%d' %(i+1)
```

```
80                          host=self.addNode(hostname)
81                          hosts.append(host)
82
83          """ Se crean los enlaces entre los distintos elementos
                de la red
84          """
85
86          self.addLink('R1','h1')
87          self.addLink('R1','R2')
88          self.addLink('R1','R3')
89          self.addLink('R1','R4')
90          self.addLink('R1','R5')
91          self.addLink('R2','R6')
92          self.addLink('R2','R7')
93          self.addLink('R3','R6')
94          self.addLink('R3','R7')
95          self.addLink('R4','R8')
96          self.addLink('R4','R9')
97          self.addLink('R5','R8')
98          self.addLink('R5','R9')
99          self.addLink('R6','h2')
100         self.addLink('R7','h3')
101         self.addLink('R7','R8')
102         self.addLink('R8','h4')
103         self.addLink('R9','h5')
104         return
105
106
107 def main():
108
109 os.system("rm -f /tmp/R*.log /tmp/R*.pid logs/*")
110 os.system("mn -c >/dev/null 2>&1")
111 os.system("killall -9 zebra pimd > /dev/null 2>&1")
112
113
114 net = Mininet(topo=SimpleTopo(), switch=Router, controller=None)
115 net.start()
116
117 for router in net.switches:
118         router.cmd("sysctl -w net.ipv4.ip_forward=1")
119         router.waitOutput()
120
121         log("Waiting %d seconds for sysctl changes to take
                effect..."
122         % args.sleep)
123         sleep(args.sleep)
124
125 for router in net.switches:
126         router.cmd("/usr/local/quagga/sbin/zebra -f /usr/local/
                quagga/etc/zebra--%s.conf -d -i /usr/local/quagga/etc
                /zebra--%s.pid > log/%s--zebra-stdout 2>&1" %(router
                .name,router.name, router.name))
127         router.waitOutput()
```

```
128         router.cmd("/usr/local/quagga/sbin/pimd -f /usr/local/
                quagga/etc/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)
129         router.waitOutput()
130         log("Starting zebra and pimd on %s" %router.name)
131
132
133         """ Configuracion de los host 1-5 de la red """
134
135         net.hosts[0].cmd("ifconfig h1-eth0 10.0.0.1")
136         net.hosts[1].cmd("ifconfig h2-eth0 6.0.0.1")
137         net.hosts[2].cmd("ifconfig h3-eth0 7.0.0.1")
138         net.hosts[3].cmd("ifconfig h4-eth0 8.0.0.1")
139         net.hosts[4].cmd("ifconfig h5-eth0 9.0.0.1")
140         net.hosts[0].cmd("route add default gw 10.0.0.2")
141         net.hosts[1].cmd("route add default gw 6.0.0.2")
142         net.hosts[2].cmd("route add default gw 7.0.0.2")
143         net.hosts[3].cmd("route add default gw 8.0.0.2")
144         net.hosts[4].cmd("route add default gw 9.0.0.2")
145
146
147
148
149         CLI(net)
150         net.stop()
151         os.system("killall -9 zebra pimd")
152
153
154
155 if __name__ == "__main__":
156 main()
```

C.2. Topología con controlador

```
1
2 #!/usr/bin/env python
3
4
5 """ Copyright (C) 2016 Carlos Santamaria Espinosa
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
40
41         info('*** Adding controller\n')
42         c0=net.addController(name='c0', controller=
              RemoteController,protocols='OpenFlow13', ip='
              127.0.0.1')
43
44         s1=net.addSwitch('s1', protocols='OpenFlow13',mac='
              00:00:00:00:00:06')
```

```
45          s2=net.addSwitch('s2', protocols='OpenFlow13',mac='
                00:00:00:00:00:07')
46          s3=net.addSwitch('s3', protocols='OpenFlow13',mac='
                00:00:00:00:00:08')
47          s4=net.addSwitch('s4', protocols='OpenFlow13',mac='
                00:00:00:00:00:10')
48          s5=net.addSwitch('s5', protocols='OpenFlow13',mac='
                00:00:00:00:00:11')
49          s6=net.addSwitch('s6', protocols='OpenFlow13',mac='
                00:00:00:00:00:12')
50          s7=net.addSwitch('s7', protocols='OpenFlow13',mac='
                00:00:00:00:00:13')
51          s8=net.addSwitch('s8', protocols='OpenFlow13',mac='
                00:00:00:00:00:14')
52          s9=net.addSwitch('s9', protocols='OpenFlow13',mac='
                00:00:00:00:00:15')
53
54          info('***adding host\n')
55          h1=net.addHost('h1', cls=Host,mac='00:00:00:00:00:01')
56          h2=net.addHost('h2', cls=Host,mac='00:00:00:00:00:02')
57          h3=net.addHost('h3', cls=Host,mac='00:00:00:00:00:03')
58          h4=net.addHost('h4', cls=Host,mac='00:00:00:00:00:04')
59          h5=net.addHost('h5', cls=Host,mac='00:00:00:00:00:05')
60
61          info('***Creating links\n')
62
63          net.addLink('s1','h1')
64          net.addLink('s1','s2')
65          net.addLink('s1','s3')
66          net.addLink('s1','s4')
67          net.addLink('s1','s5')
68          net.addLink('s2','s6')
69          net.addLink('s2','s7')
70          net.addLink('s3','s6')
71          net.addLink('s3','s7')
72          net.addLink('s4','s8')
73          net.addLink('s4','s9')
74          net.addLink('s5','s8')
75          net.addLink('s5','s9')
76          net.addLink('s6','h2')
77          net.addLink('s7','h3')
78          net.addLink('s7','s8')
79          net.addLink('s8','h4')
80          net.addLink('s9','h5')
81
82
83          info('***Starting network\n')
84          net.build()
85          net.start()
86
87
88          info('***Starting controller\n')
89          for controller in net.controllers:
90          controller.start()
```

```
91
92          info('***Starting switches\n')
93          net.get('s1').start([c0])
94          net.get('s2').start([c0])
95          net.get('s3').start([c0])
96          net.get('s4').start([c0])
97          net.get('s5').start([c0])
98          net.get('s6').start([c0])
99          net.get('s7').start([c0])
100         net.get('s8').start([c0])
101         net.get('s9').start([c0])
102
103
104         info('***Post configure switches and hosts\n')
105         CLI(net)
106         net.stop()
107
108 if __name__ == "__main__":
109         setLogLevel( 'info' )
110         myNetwork()
```

Bibliografía

[1] Source code Mininet. http://github.com/mininet. [Online; Última visita: 04/04/2016].

[2] Apache Maven. http://maven.apache.org/index.html. [Online; Última visita: 05/06/2016].

[3] Atresplayer. http://www.atresplayer.com/. [Online; Última visita: 20/12/2015].

[4] BGP Path Hijacking Attack Demo. https://github.com/mininet/mininet/wiki/BGP-Path-Hijacking-Attack-Demo. [Online; Última visita: 10/04/2016].

[5] Cisco Application Centric Infrastructure. http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html. [Online; Última visita: 10/02/2016].

[6] Comandos qpimd. https://github.com/udhos/qpimd/blob/pim/pimd/COMMANDS. [Online; Última visita: 10/04/2016].

[7] Eclipse (IDE). https://eclipse.org/downloads/. [Online; Última visita: 05/06/2016].

[8] Instalación Quagga. https://lists.gnu.org/archive/html/qpimd-users/2011-08/msg00009.html. [Online; Última visita: 20/02/2016].

[9] Itu-t recommendation p.800, p.800: Telephone transmission quality. methods for objective and subjective assessment of quality, august 1996. Technical report.

[10] "JUNIPER. Whitepaper-Decoding SDN". http://www.juniper.net/us/en/local/pdf/whitepapers/2000510-en.pdf. [Online; Última visita: 20/05/2016].

[11] Mininet download. https://github.com/mininet/mininet/wiki/Mininet-VM-Images. [Online; Última visita: 12/11/2015].

[12] Mininet Examples. https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py. [Online; Última visita: 10/03/2016].

[13] Mininet Walkthrough. http://mininet.org/walkthrough. [Online; Última visita: 20/05/2016].

[14] Netflix. https://www.netflix.com/es/. [Online; Última visita: 15/12/2015].

[15] ns-3. project, «ns-3: OpenFlow switch support,» National Science Foundation. https://www.nsnam.org/. [Online; Última visita: 03/04/2016].

[16] "Open Source Routing: A Comparison". https://keepingitclassless.net/2015/05/open-source-routing-comparison/. [Online; Última visita: 01/04/2016].

[17] OpenDayLight. https://www.opendaylight.org/. [Online; Última visita: 30/05/2016].

[18] OpenDaylight and OSGI basics. https://fredhsu.wordpress.com/2013/05/03/opendaylight-and-osgi-basics/. [Online; Última visita: 06/06/2016].

[19] OpenDaylight Application Developer's tutorial. http://sdnhub.org/tutorials/opendaylight/. [Online; Última visita: 10/06/2016].

[20] OpenDaylight Controller:MD-SAL:MD-SAL App Tutorial. https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:MD-SAL_App_Tutorial. [Online; Última visita: 15/05/2016].

[21] OpenDaylight. OpenDaylight User Guide . https://www.opendaylight.org/sites/opendaylight/files/bk-user-guide.pdf. [Online; Última visita: 01/05/2016].

[22] OpenDaylight User Guide. https://nexus.opendaylight.org/content/sites/site/org.opendaylight.docs/master/userguide/manuals/userguide/bk-user-guide/content/_logging_in.html. [Online; Última visita: 26/05/2016].

[23] OpenDayLight, «Wiki OpenDayLight: OpenDaylight Controller:MD-SAL:FAQ,». https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ. [Online; Última visita: 10/05/2016].

[24] OpenFlow. http://archive.openflow.org/wp/learnmore/. [Online; Última visita: 22/2/2016].

[25] "OpenLab. Software Defined Networking technology details and openlab research overview. http://openlab.web.cern.ch/sites/openlab.web.cern.ch/files/presentations/0%5B1%5D.pdf. [Online; Última visita: 20/03/2016].

[26] SDN Tutorials, «Difference Between AD-SAL/MD-SAL,». http://sdntutorials.com/difference-between-ad-sal-and-md-sal/. [Online; Última visita: 20/05/2016].

[27] SDX - OpenDaylight Project. https://www.sdxcentral.com/projects/opendaylight-project/. [Online; Última visita: 10/01/2016].

[28] SDX - What is the OpenDaylight Project (ODL). https://www.sdxcentral.com/sdn/resources/opendaylight-project/. [Online; Última visita: 28/04/2016].

[29] SSMPING. http://manpages.ubuntu.com/manpages/trusty/man1/ssmping.1.html. [Online; Última visita: 20/4/2016].

[30] Ubuntu download. http://www.ubuntu.com/download/desktop. [Online; Última visita: 20/10/2015].

[31] ucb-sts, «STS - SDN troubleshooting simulator». http://ucb-sts.github.io/sts/. [Online; Última visita: 03/03/2016].

[32] VM SDN-HUB. http://sdnhub.org/tutorials/sdn-tutorial-vm/. [Online; Última visita: 30/05/2016].

[33] Wireshark. https://www.wireshark.org/. [Online; Última visita: 05/11/2015].

[34] Wuaki.tv. https://es.wuaki.tv/. [Online; Última visita: 10/12/2015].

[35] Youtube. https://www.youtube.com/. [Online; Última visita: 10/12/2015].

[36] Itu-t recommendation p.10/g.100 (2006) amendment 1 (01/07), p.10: New appendix i – definition of quality of experience (qoe). Technical report, January 2001.

[37] A Adams, J Nicholas, and W Siadak. Rfc 3973-protocol independent multicast-dense mode (pim-dm): Protocol specification (revised). *IETF January*, 2005.

[38] Zaid Albanna, Kevin Almeroth, David Meyer, and M Schipper. Iana guidelines for ipv4 multicast address assignments. *IETF RFC3171*, 2001.

[39] Open Data Center Alliance. Open data center alliance: Software-defined networking rev. 2.0. *Open Data Center Alliance, Tech. Rep*, 2014.

[40] Pablo Ameigeiras, Juan J Ramos-Munoz, Jorge Navarro-Ortiz, Preben Mogensen, and Juan M Lopez-Soler. Qoe oriented cross-layer design of a resource allocation algorithm in beyond 3g systems. *Computer Communications*, 33(5):571–582, 2010.

[41] B Cain, S Deering, I Kouvelas, B Fenner, and A Thyagarajan. Ietf rfc 3376:ïnternet group management protocol version 3. *IGMPv3), oct*, 2002.

[42] Alejandro García Centeno, Carlos Manuel Rodríguez Vergel, Caridad Anías Calderón, and Frank Camilo Casmartiño Bondarenko. Controladores sdn, elementos para su selección y evaluación. *Revista Telem@ tica*, 13(3):10–20, 2014.

[43] Tim Chown and Stig Venaas. Multicast troubleshooting with ssmping and asmping. 2006.

[44] R Coltun, D Ferguson, and J Moy. A. lindem,.ºspf for ipv6. Technical report, RFC 5340, July, 2008.

[45] Steve Deering. Rfc-1112: Host extension for ip multicasting. *Request For Comments*, 1989.

[46] B Fenner, M Handley, H Holbrook, and I Kouvelas. Rfc 4601-protocol independent multicast-sparse mode. Technical report, tech. rep., IETF, 2006.

[47] W Fenner. Rfc 2236: Internet group management protocol, version 2, november 1997. 1989.

[48] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[49] Michael T Goodrich and Roberto Tamassia. Algorithm design. *Wiely India*, 2002.

[50] Hugh Holbrook and Brad Cain. Source-specific multicast for ip. Technical report, RFC 4607, August, 2006.

[51] Liang-Hao Huang, Hui-Ju Hung, Chih-Chung Lin, and De-Nian Yang. Scalable steiner tree for multicast communications in software-defined networking. *arXiv preprint arXiv:1404.3454*, 2014.

[52] Cisco Visual Networking Index. Forecast and methodology, 2014-2019 white paper. Technical report, Technical Report, Cisco, 2015.

[53] Cisco Visual Networking Index. The zettabyte era–trends and analysis. *Cisco white paper*, 2016.

[54] K Ishiguro et al. A routing software package for tcp/ip networks (quagga 0.99. 4), 2006.

[55] Kunihiro Ishiguro. Quagga software routing suite, 1991.

[56] Amrit Iyer, Pranaw Kumar, and Vijay Mann. Avalanche: Data center multicast using software defined networking. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*, pages 1–8. IEEE, 2014.

[57] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[58] B Lantz, Nikhil Handigol, Brandon Heller, and Vimal Jeyakumar. Introduction to mininet, 2012.

[59] Xiaozhou Li and Michael J Freedman. Scaling ip multicast on datacenter topologies. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 61–72. ACM, 2013.

[60] G Malkin. Rfc 2453: Rip version 2. *Request for Comments*, 2453, 1998.

[61] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[62] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Rules placement problem in openflow networks: a survey. 2016.

[63] B Pfaff. Open vswitch manual. *Manual, Open vSwitch, nd [Accessed 5 October, 2014].*

[64] B Pfaff et al. Openflow switch specification version 1.1. 0 implemented (wire protocol 0x02), 2011.

[65] Y Rekhter and T Li. Rfc 1771. *A Border Gateway Protocol*, 4:1–54, 1995.

[66] Charalampos Rotsos, Richard Mortier, Anil Madhavapeddy, Balraj Singh, and Andrew W Moore. Cost, performance & flexibility in openflow: Pick three. In *Communications (ICC), 2012 IEEE International Conference on*, pages 6601–6605. IEEE, 2012.

[67] Shan-Hsiang Shen, Liang-Hao Huang, De-Nian Yang, and Wen-Tsuen Chen. Reliable multicast routing for software-defined networks. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 181–189. IEEE, 2015.

[68] S Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.

[69] William Stalling. *Foundations of Modern Networking*. Addison Wesley, 2016.

[70] Mininet Team. Download/get started with mininet. *URL: http://mininet. org/download/(visited on 03/17/2015)*, 2015.

[71] Ángel Leonardo Valdivieso Caraguay, Alberto Benito Peral, Lorena Isabel Barona López, and Luis Javier García Villalba. Sdn: evolution and opportunities in the development iot applications. *International Journal of Distributed Sensor Networks*, 2014, 2014.

[72] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[73] Beau Williamson. *Developing IP Multicast Netwoks*, volume 2. Cisco Press, 2000.

[74] Nan Zhang and Heikki Hammainen. Cost efficiency of sdn in lte-based mobile networks: Case finland. In *Networked Systems (NetSys), 2015 International Conference and Workshops on*, pages 1–5. IEEE, 2015.

[75] Miao Zhao, Bin Jia, Mingquan Wu, Haoyong Yu, and Yang Xu. Software defined network-enabled multicast for multi-party video conferencing systems. In *Communications (ICC), 2014 IEEE International Conference on*, pages 1729–1735. IEEE, 2014.