ugr Universidad de Granada

TRABAJO DE FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIONES

Análisis de redes SDN utilizando *Mininet* e implementación de un *Deep Packet Inspector*.

Autor

Manuel Sánchez López

Director

Jorge Navarro Ortiz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Julio de 2015

ugr Universidad de Granada

Análisis de redes SDN utilizando *Mininet* e implementación de un *Deep Packet Inspector*.

REALIZADO POR:

Manuel Sánchez López

DIRIGIDO POR:

Jorge Navarro Ortiz

DEPARTAMENTO:

Teoría de la Señal, Telemática y Comunicaciones

Granada, Julio de 2015

Análisis de redes SDN utilizando *Mininet* e implementación de un *Deep Packet Inspector*.

Manuel Sánchez López

PALABRAS CLAVE:

SDN (*Software defined Network*), *Mininet*, OpenFlow, ODL (*OpenDayLight*), DPI (*Deep Packet Inspector*), Calidad de Servicio (QoS - *Quality of Service*), Java, *Controller*, *ToS* (*Type of Services*), YouTube.

RESUMEN:

En la actualidad, se está produciendo un cambio de paradigma en las redes de comunicaciones con el objetivo de aumentar su flexibilidad y minimizar sus costes. Además, el desarrollo de nuevas arquitecturas de red se centra también en la implementación de calidad de servicio ofrecida a los usuarios finales.

En este contexto, *Software Defined Networking* (SDN) es un tipo de arquitectura de red dinámica, gestionable, económica y adaptable, siendo ideal para soportar las aplicaciones que se desarrollan hoy en día. Esta arquitectura separa los planos de control y de usuario, permitiendo a los programadores de red gestionar el plano de control a partir de un controlador centralizado. Dicho controlador dotará al sistema de abstracción con el objetivo de facilitar la creación de nuevas aplicaciones y servicios de red. Este nuevo paradigma, SDN, está siendo utilizado en las redes de gigantes como Google o Amazon, además de ser la tecnología que se prevé para las redes troncales en las futuras comunicaciones móviles 5G.

Para la comunicación entre controlador y los elementos de red (e.g. *switches*) surge el estándar OpenFlow, desarrollado por la *Open Networking Foundation* (ONF). Existen multitud de *switches* que implementan OpenFlow, tanto comerciales (e.g. Juniper) como de licencia gratuita (e.g. v*Switch*).

Además, con el objetivo de crear prototipos de forma rápida y barata, han surgido emuladores de este tipo de redes como *Mininet*. *Mininet* permite crear una red virtual con múltiples *switches* en un único PC, además de permitir la interacción con otras máquinas reales o virtuales. En cuanto al controlador para gestionar las redes SDN creadas, se ha optado por *OpenDayLight*, debido tanto a su popularidad como a sus características. Estas herramientas requieren de una familiarización previa debido a lo novedoso de este ámbito de la tecnología.

Tomando en consideración estos aspectos, el Trabajo de Fin de Grado a desarrollar pretende mostrar una idea del funcionamiento de estas redes en un entorno real y su potencial para mejorar la calidad de servicio actual. En concreto, en este trabajo se ha diseñado e implementado un *Deep Packet Inspector* en el controlador, de forma que los paquetes son marcados basándonos en el tipo de tráfico, para más tarde ser utilizadas por otras funcionalidades de la red (e.g. gestión de colas) con el objetivo de priorizar ciertos tipos de tráfico y, en última instancia, ofrecer calidad de servicio. Este DPI, además de permitir la detección de tipos de tráfico comunes como pueden ser el tráfico *web* o VoIP, se centra en la detección del tráfico de vídeo de YouTube. Por otro lado, su diseño modular facilita al desarrollador añadir nuevos tipos de tráfico al mismo. Para conseguir detectar este último tipo de tráfico, debido a que YouTube encripta los datos mediante HTTPS, se ha desarrollado un método basado en la detección de los paquetes DNS referentes a YouTube para determinar estos flujos.

Para su desarrollo se ha escogido el entorno de programación Java, al ser un lenguaje que ofrece muchas posibilidades en el desarrollo del controlador y su uso extendido en entornos de redes de comunicación. Además, se hace uso de Python para la implementación de topologías en *Mininet*, pues es el lenguaje que esta herramienta utiliza.

SDN network analysis using *Mininet* and implementation of a Deep Packet Inspector.

Manuel Sánchez López

KEYWORDS:

SDN (*Software defined Network*), *Mininet*, OpenFlow, ODL (*OpenDayLight*), DPI (*Deep Packet Inspector*), QoS (*Quality of Service*), Java, *Controller*, *ToS* (*Type of Services*), YouTube.

ABSTRACT:

In a society immerged in a world of new technologies, future ways to communicate people is one of the most important challenges. The development of new network architectures and their procedures has been focused in the past on the improvement of the quality of service experienced by the end-users, but it should also flexible and cost-effective.

In this regard, *Software*-Defined Networking (SDN) is an emerging architecture which is dynamic, manageable, economical, and adaptable, making it ideal for the dynamic nature of today's applications. This architecture decouples the control and user planes by allowing the operator/developer to program the control plane on a logically centralized controller. In this controller, the underlying network infrastructure is abstracted in order to create new applications and network services. This type of architecture is having an important impact in a large number of companies like Google or Amazon, which are investigating how to improve the performance of SDN networks in the near future, and also is expected to be part of the core of 5G mobile networks.

The communication between the controller and the network elements (e.g. *switches*) is performed by means of the OpenFlow protocol, which has been developed and standardized by the Open Networking Foundation (ONG). There are many switches that implement OpenFlow, both commercial (e.g. Juniper) and *open source* (e.g. Open vSwitch).

In addition, different emulators have emerged with the aim of creating prototypes quick and inexpensively. For that purpose, *Mininet* is used in this work, which permits creating a virtual network with multiple *switches* on a single PC, allowing them to interact with other real or virtual machines. *OpenDayLight* has been chosen as the SDN controller, due to its popularity and appropriate features. Due to the novelty of this topic, a significant part of this work has been devoted to the familiarization with the SDN paradigm and these tools.

Based on these tools, this thesis is intended to create a prototype of a *Deep Packet Inspector* (DPI) which may be deployed on a SDN network but also on a conventional network. This

DPI will mark packets based on their traffic type, which may be utilized by other network functionalities (e.g. packet scheduling or queue management) in order to prioritize traffic flows and, ultimately, improve their quality of service. This DPI has been designed on a modular fashion to easily allow the developer to incorporate new traffic types. In addition to some of the most common network traffic types –VoIP and web browsing–, the work of this thesis is focused on the traffic detection from the YouTube video service. For that purpose, since YouTube sends encrypted data using HTTPS, a method has been devised to recognize YouTube video flows based on the initial DNS signaling.

Java has been chosen as the programming language, due to its possibilities for application development for *OpenDayLight*, its portability and its extensive usage. Additionally, Python has been used for the design of network topologies for *Mininet* since its Python API is very well documented and very complete.

Yo, Manuel Sánchez López, alumno de la titulación de Grado en Ingeniería de Tecnologías de 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: Manuel Sánchez López

Granada a Julio de 2015

D. Jorge Navarro Ortiz, Profesor del Área de Telecomunicaciones del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada, como director del Trabajo Fin de Grado de D. Manuel Sánchez López

Informa de que el presente trabajo, titulado:

Análisis de redes SDN mediante Mininet e implementación de un Deep Packet Inspector

ha sido realizado bajo su supervisión por Manuel Sánchez López, y autoriza la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a Julio de 2015. El director:

Jorge Navarro Ortiz

Agradecimientos

A mi familia, por el apoyo constante en todos estos años de carrera. A los que están y a los que estuvieron. A mi madre, porque aunque a veces no encuentren mis respuestas, sus preguntas son el motor que me motiva a seguir creciendo. A mi padre, por el esfuerzo que hace cada día para poder darnos lo mejor. A mi hermano, el que siempre está cuando se le necesita.

A mis amigos, por los momentos tanto dentro de la ETSIIT como los que disfrutamos fuera. Porque sin esas cervezas todo sería más gris y aburrido. En especial a Cristian, un apoyo indispensable desde que nos conocimos y mi compañero de viaje en este proyecto.

Y, por supuesto, a Jorge, mi tutor, por su ayuda constante y su buen consejo. Sin su predisposición nada de esto hubiese sido posible.

Gracias a todos.

Índice general.

1. **Motivación e introducción** **1**

1.1. Contexto y motivaciones 1

1.1.1. Virtualización de redes. 2

1.1.2. QoS en las nuevas redes de comunicación. 3

1.1.3. Relevancia del *streaming* de vídeo: tráfico YouTube. 5

1.1.4. ¿Por qué apostar por las nuevas redes SDN? 8

1.2. Logros y aportaciones. 10

1.3. Organización de la memoria. 11

2. **Tecnologías implicadas.** **15**

2.1. SDN (*Software* Defined Network). 15

2.1.1. Arquitectura de SDN. 17

2.2. Protocolo OpenFlow. 18

2.2.1. OpenFlow *Switch*. 19

2.2.2. Componentes del protocolo OpenFlow. 21

2.2.3. Mensajes OpenFlow. 24

2.2.4. Comparativa de Versiones OpenFlow. 25

2.3. Open vSwitch. 28

3. **Estado del arte** **31**

3.1. DPIs *open source* 31

3.1.1. *nDPI*. 31

3.1.2. *Bro-IDS*. 32

3.1.3. *Snort*. 32

3.2. DPIs comerciales. 33

3.2.1. *Policy Enforcement Manager*. 33

3.2.2. 7750 Service *Router* - Mobile Gateway 36

3.2.3. DPI Application Platform (7000). 36

3.3. Comparativa con DPIs del mercado. 36

3.3.1. Comparativa con DPIs *open source*...36
3.3.2. Comparativa con DPIs comerciales...37

4. Análisis de objetivos, requisitos y metodología. 39

4.1. Objetivos... 39
4.2. Especificación de requisitos... 40
4.2.1. Requisitos funcionales...40
4.2.2. Requisitos no funcionales...41
4.3. Valoración del alcance de los objetivos... 42
4.4. Metodología... 42

5. Planificación y estimación de costes 45

5.1. Planificación... 45
5.2. Recursos utilizados... 49
5.2.1. Recursos humanos...49
5.2.2. Recursos *hardware*...50
5.2.3. Recursos *software*...50
5.3. Estimación de costes... 51
5.4. Presupuesto final... 53

6. Herramientas utilizadas 55

6.1. *Mininet*... 55
6.1.1. ¿Por qué utilizar *Mininet*?...55
6.1.2. Programación de topologías de red con *Mininet*...57
6.2. *OpenDayLight*... 65
6.2.1. ¿Por qué utilizar *OpenDayLight*? 66
6.2.2. *Orange* y el uso de *OpenDayLight*...67
6.2.3. Distribuciones disponibles...68
6.2.4. Comparativa de modelos *AD-SAL* y *MD-SAL*...71
6.2.5. Interfaz *web*...74
6.2.6. Flujo proactivo: REST API...76
6.2.7. Flujo reactivo...83

7. Diseño e implementación del DPI.............................. 87
7.1. Conceptos previos. 87
7.2. Implementación con MD-SAL. 88
7.2.1. Modificación de *ToS* a los flujos de tipo ARP. 88
7.3. Implementación con AD-SAL. 91
7.3.1. Detección de tráfico *web* y VoIP. 91
7.3.2. Detección de tráfico *YouTube*. 92
8. Implementación de controlador con DPI independiente y evaluación de resultados. 101
8.1. Implementación de controlador con DPI independiente. 101
8.2. Evaluación de resultados 105
9. Conclusiones y vías futuras 107
9.1. Conclusiones. 107
9.2. Vías futuras 108
9.2.1. Detección de otras aplicaciones relevantes 108
9.2.2. Desarrollo en *MD-SAL*. 108
9.2.3. Implementación en entornos reales. 109
9.3. Valoración personal. 109
A. Manual de instalación de *Mininet*. 111
B. Instalación de OpenDayLight y compilación de nuevos paquetes. 113
C. Manual de usuario de la aplicación. 117
D. Complicaciones y errores encontrados. 119
E. Blog en WordPress. 121
F. Comparativa de controladores de red. 125
G. Topologías para *Mininet*. 135

Bibliografía 139

Índice de figuras.

Figura 1.1. Incremento de la definición de vídeo estimado. 6
Figura 1.2. Consumo previsto de diferentes calidades de vídeo. 6
Figura 1.3. Tráfico IP Global según aplicación. 7
Figura 1.4. Comparativa en términos de búsqueda de YouTube, Hulu y Netflix. 7
Figura 1.5. Cambio de paradigma propuesto por SDN. 10
Figura 2.1. Esquema de conexión entre SDN y la red física. 15
Figura 2.2. Estructura de red SDN. 16
Figura 2.3. Arquitectura de SDN. 18
Figura 2.4. Diferencias entre un switch tradicional y un switch OpenFlow. 19
Figura 2.5. Esquema de conexión de switch OpenFlow. 20
Figura 2.6. Esquema general de un switch OpenFlow v1.0. 20
Figura 2.7. Flujo de paquetes sobre el procesado pipeline. 22
Figura 2.8. Diagrama de flujo del flujo de paquetes en un switch OpenFlow. 23
Figura 2.9. Diagrama de evolución del protocolo OpenFlow. 27
Figura 2.10. Diseño de Open vSwitch. 29
Figura 2.11. Descripción de Open vSwitch. 29
Figura 3.1. Definición de políticas de PEM. 34
Figura 3.2. Clasificación y acciones de política. 34
Figura 3.3. Detección de tráfico mediante identificación OTT. 35
Figura 3.4. Servicio de encadenamiento del control de políticas para SDN. 35
Figura 4.1. Diagrama de flujo del procesamiento de paquetes. 40
Figura 5.1. Diagrama de Gantt estimado del proyecto. 48
Figura 5.2. Gráfico de barras del coste de recursos humanos para cada paquete de trabajo. 52
Figura 5.3. Porcentaje de costes final. 53
Figura 6.1. Creación de topología básica en Mininet. 57
Figura 6.2. Ayuda básica de Mininet. 58
Figura 6.3. Muestra de los nodos de la topología con Mininet. 58
Figura 6.4. Muestra de los enlaces entre dispositivos de la red en Mininet. 59
Figura 6.5. Muestra de los dispositivos y su información con Mininet. 59
Figura 6.6. Test de conectividad de red en Mininet. 60
Figura 6.7. Interfaz gráfica de Miniedit. 62
Figura 6.8. Ejemplo de topología creada con miniedit. 63
Figura 6.9. Mensaje en la terminal al ejecutar topología de miniedit. 63
Figura 6.10. (a) Ping entre h1 y h4 en la topología creada en miniedit y (b) flujo añadido. 64
Figura 6.11. Lista de empresas que ofrecen apoyo al controlador OpenDayLight. 66
Figura 6.12. Interfaz gráfica de karaf. 70
Figura 6.13. Estructuras de AD-SAL y MD-SAL. 72

Figura 6.14. Ventana de login de la interfaz web de ODL. 75
Figura 6.15. Ejemplo de gráfico de topología creado por la interfaz web de ODL. 75
Figura 6.16. Pestaña para añadir flujos de forma proactiva en la interfaz web de ODL. 76
Figura 6.17. Menú de aplicaciones de google chrome donde se descarga Postman. 79
Figura 6.18. Interfaz de Postman. 79
Figura 6.19. Cabecera de opciones de Postman. 79
Figura 6.20. Test de ping en la red creada. 81
Figura 6.21. Imagen de la topología creada. 81
Figura 6.22. Flujos instalados antes de ejecutar el flujo proactivo. 81
Figura 6.23. Envío de flujo proactivo y respuesta del servidor. 83
Figura 6.24. Tabla de flujo tras instalación de flujo proactivo. 83
Figura 6.25. Comprobación de utilización del flujo proactivo. 83
Figura 6.26. Topología en árbol creada para probar la adición de flujo reactivo. 85
Figura 6.27. Ping realizado entre el host 1 y el host 4. 86
Figura 6.28. Logs en el controlador que avisan de la adición de flujos reactivos. 86
Figura 6.29. Flujos añadidos en el controlador debido al ping. 86
Figura 7.1. Ping entre dos host utilizando controlador con arquitectura MD-SAL. 90
Figura 7.2. Formato de paquetes DNS, 93
Figura 7.3. Reproducción de vídeo de YouTube en host 1. 97
Figura 7.4. Flujo añadido de tráfico de vídeo YouTube. 97
Figura 7.5. Captura de wireshark en la que se aprecia el cambio de ToS en los paquetes. 98
Figura 7.6. Distribución de interfaces de topología en árbol de tres niveles creada en Mininet. 98
Figura 7.7. Topología en árbol de tres niveles. 99
Figura 7.8. Reproducción de vídeo de YouTube por parte de h1. 99
Figura 7.9. Flujos añadidos en: (a) s1, (b) s2 y (c) s3. 100
Figura 8.1. Escenario de implementación de controlador con DPI independiente. 101
Figura 8.2. Configuración de la interfaz de red para red interna en la VM del controlador. 102
Figura 8.3. Reproducción de vídeo desde cuatro hosts diferentes. 105
Figura 8.4. Flujos de vídeos de YouTube correspondientes a cada host. 106
Figura B.1. Estructura de nuestro proyecto con Maven. 114
Figura E.1. Portada del blog “Aprendiendo OpenDayLight”. 121
Figura E.2. Visitas y visitantes al blog aprendiendoodl.wordpress.com. 122
Figura E.3. Países con más visitas al blog. 122
Figura F.1. Mensajes enviados por día comunidad a la NOX. 125
Figura F.2. Interfaz gráfica de NOX 126
Figura F.3. Interfaz gráfica de POX. 126
Figura F.4. Interfaz gráfica de Floodlight. 128
Figura F.5. Lista de empresas que ofrecen apoyo al controlador OpenDayLight. 129

Figura F.6. Interfaz web de OpenDayLight..129
Figura F.7. Evolución de búsqueda de "OpenDayLight" mediante google. 133

Índice de tablas.

Tabla 1.1. Número medio de dispositivos y conexiones per Cápita. 4
Tabla 2.1. Principales componentes de una entrada de flujo. 23
Tabla 3.1. Lista de los DPIs comerciales más importantes. 33
Tabla 5.1. Distribución temporal del proyecto. 49
Tabla 5.2. Coste estimado de recursos humanos. 51
Tabla 5.3. Costes estimados de recursos hardware. 52
Tabla 5.4. Presupuesto final estimado. 53
Tabla 6.1. Lista de algunos campos ejecutables en Mininet. 61
Tabla 6.2. Comparativa entre AD-SAL y MD-SAL 74
Tabla F.1. Comparativa de controladores de redes SDN 130
Tabla F.2. Búsquedas principales de OpenDayLight 133

Capítulo 1

Motivación e introducción

En este capítulo se plasmarán los conceptos básicos de este trabajo para situar al lector y facilitar la comprensión de los puntos que se expondrán en esta memoria.

En primer lugar se introducirá el concepto de virtualización de redes y se analizará la necesidad de introducir calidad de servicio en las nuevas redes de comunicación.

Más tarde, se justificarán las ventajas de las redes definidas por *software* (SDN – *"Software Defined Network"*) [1], pues son las redes en las que se fundamenta el presente proyecto.

A continuación se expondrá un apartado de motivaciones con el que se pretende mostrar la relevancia del trabajo realizado y de todas las tecnologías y herramientas involucradas en él.

En el siguiente apartado se continuará definiendo los logros y las aportaciones que se han conseguido con la realización del proyecto.

Seguidamente se expondrá, a modo de esquema didáctico, la estructura que seguirá la memoria, incluyendo un breve resumen de los conceptos que se tratarán en cada uno de los puntos.

Definida la estructura de este capítulo, se incluye el objetivo final del presente proyecto antes de comenzar con su desarrollo. El fin último del mismo consistirá en el diseño e implementación de un *Deep Packet Inspector* [2] en el controlador de una red SDN con la intención de detectar ciertos tipos de tráfico entre los que se incluyen el *streaming* de vídeo de YouTube, la voz sobre IP o el tráfico *web*. Una vez identificados, se les asignará una etiqueta que permita a la red ofrecerles un tratamiento diferenciado para conseguir una calidad de servicio (QoS – *"Quality of Service"*) determinada en nuestra red.

1.1. Contexto y motivaciones.

En esta sección vamos a tratar de contextualizar el momento en que se encuentran las redes de comunicación. Para ello incidiremos en la importancia del crecimiento latente de dispositivos electrónicos mediante los cuales un único usuario se conecta a la red y la creciente demanda del tráfico de vídeo.

Por otra parte, trataremos una serie de apartados acerca de diversos aspectos que han influido en la motivación de este proyecto. Se tratarán temas relacionados con la tecnología utilizada y su influencia determinante para entender el mismo, así como conceptos que son claves para comprender el porqué de la necesidad de estas redes.

1.1.1. Virtualización de redes.

La repercusión de Internet en las últimas décadas se hace patente en el porcentaje de personas en el mundo que tienen acceso a la red cada día. En la actualidad es complicado encontrar a alguien que no haga uso de Internet prácticamente a diario, ya sea para leer su correo electrónico o para visualizar un vídeo mediante *streaming*. Sin embargo, esta popularidad se ha convertido al mismo tiempo en un impedimento para su crecimiento. Debido a su naturaleza, adoptar una nueva arquitectura o introducir una modificación requiere de un gran consenso entre los distintos actores involucrados en la red, lo que dificulta estas tareas. Esto ha desembocado en una continua actualización simple e incremental de una arquitectura muy definida y cada vez más limitada. El punto de inflexión lo encontramos en la propuesta de virtualizar las redes.

¿Qué es la virtualización de redes?

La virtualización de redes [3] es la combinación de los recursos de red *hardware* con los recursos de red *software* en una única unidad administrativa. El objetivo de la virtualización de redes consiste en facilitar un uso compartido de recursos de redes eficaz, controlado y seguro para los usuarios y los sistemas.

Con la virtualización lo que se permite es la coexistencia de múltiples redes virtuales sobre el mismo soporte físico. Con ella se propone el desglose de las funcionalidades del entorno de red, separando el rol tradicional de los proveedores de servicios de Internet (ISPs) en dos: proveedores de infraestructura (InPs), que controlan la infraestructura física, y los proveedores de servicios (SPs), encargados de crear redes virtuales agregando recursos desde múltiples InPs y de ofrecer servicios sobre ellas. Lo que esto permite es la virtualización heterogénea de redes capaces en un mismo entorno físico funcionando de manera independiente y aislada. A partir de esta premisa se podrán implementar y gestionar servicios *end-to-end* a medida sobre las redes virtuales para los usuarios finales. Así conseguimos esquivar las limitaciones que comentábamos al principio de este capítulo.

No obstante, este tipo de redes es aún incipiente y presenta problemas de carácter técnico en cuanto a la creación de instancias, su funcionamiento o su gestión, postulándose como una línea de trabajo presente y futura muy interesante para explotar.

Redes activas y programables.

Dentro de las distintas posibilidades que nos permite la virtualización de redes (VLAN, VPN, etcétera) nosotros vamos a centrarnos en las redes activas y programables.

Las redes activas permiten de forma flexible la programación en línea de los nodos intermedios de la red, suponiendo un gran salto conceptual en la evolución de las tecnologías de red. De este modo se consigue un procesamiento de paquetes más preciso y diferenciado. La ventaja que supone la programación en línea por parte de los administradores de red

facilita una gestión proactiva de la misma. Esto nos permite acelerar la evolución de las redes, reducir el tiempo de desarrollo y despliegue de nuevos servicios o adaptar los servicios existentes de acuerdo a las necesidades concretas de sus aplicaciones.

Al mismo tiempo, al ser programable, nos permite programar a partir de un conjunto mínimo de APIs con las que se puede conseguir integrar infinidad de servicios de alto nivel. La programación de los servicios de red se consigue mediante la integración de capacidad computacional.

Metas en la Virtualización de Redes.

En primera instancia, el objetivo inicial llevado a cabo a cerca de la virtualización de redes fue el de proporcionar seguridad, flexibilidad y capacidad a las redes privadas virtuales o redes activas y programables. En la actualidad, se han añadido nuevas metas como pueden ser la heterogeneidad, capacidad de gestión, aislamiento, programabilidad, facilitar el desarrollo y la investigación o el soporte a sistemas tradicionales.

En este ámbito, *OpenFlow* es una tecnología emergente de virtualización de red que aporta a los usuarios flexibilidad y control para la configuración de sus entornos de red a partir de unas especificaciones concretas. Su despliegue hace posible la reducción de forma significativa de la complejidad de los dispositivos de red y la automatización de tareas utilizando una gestión simplificada. Además, reduce el tiempo dedicado a implementar cambios en la red y, como comentaremos más adelante, la administración dinámica en tiempo real de la misma.

1.1.2. QoS en las nuevas redes de comunicación.

En las redes actuales cada vez se hace más visible el interés de instaurar una metodología que proporcione a los usuarios una QoS determinada dependiendo de diversos factores. Es por ello que se comienzan a investigar nuevas vías para conseguir este objetivo, como la implementación de nuevas arquitecturas de red, nuevos protocolos de comunicación, etcétera.

Como podemos ver en la siguiente tabla [4] (donde CAGR es la tasa de crecimiento anual compuesto, *Compound Annual Growth Rate*), los estudios realizados hace años ya avanzaban el incremento de dispositivos y conexiones a la red, lo cual se traduce en una mayor demanda de servicios y, por lo tanto, una exigencia cada vez mayor por parte de los usuarios.

	2013	2018	CAGR
Asia Pacífico	1.41	2.24	9.7%
Centro y Este de Europa	2.10	3.39	10.1%
Latino América	1.75	2.58	8.1%
Oriente Medio y África	0.92	1.28	6.7%
América del Norte	5.34	9.26	11.7%
Oeste de Europa	3.89	6.52	10.9%
Global	1.73	2.73	9.5%

Tabla 1.1. Número medio de dispositivos y conexiones per Cápita.

Las tecnologías de calidad de servicio pretenden resolver problemas como la latencia, pérdida de paquetes de datos o *jitter*, garantizando una calidad de servicio óptima. Su principal función es la de priorizar el tráfico de red, ofreciendo un mayor ancho de banda al tráfico más prioritario y ralentizar el que lo es menos.

Cisco propone la implementación de las siguientes características en sus dispositivos para ofrecer QoS [5]:

- Clasificación y marcado del tráfico de manera que la red pueda diferenciar el tráfico.
- Tráfico condicionado a los flujos de tráfico a medida para el comportamiento de un tráfico específico y rendimiento.
- Marcado de tasas de tráfico por encima de los umbrales específicos como prioridad menor.
- Descartar paquetes cuando las tasas alcanzan umbrales específicos.
- Tanto la planificación de paquetes como los paquetes de mayor prioridad transmiten desde las colas de salida antes que los paquetes de menor prioridad.
- Gestión de colas de salida de tal manera que los paquetes de menor prioridad a la espera de transmisión no monopolicen el espacio del búfer.

Estas funcionalidades permiten, según Cisco [5] lo siguiente:

- Predecir tiempos de respuesta para flujos de paquetes *end-to-end*, operaciones entrada/salida, operaciones de datos, transacciones, etc.
- Gestionar y determinar las capacidades de las aplicaciones sensibles al *jitter* tales como aplicaciones de audio y vídeo correctamente.
- Racionalizar las aplicaciones sensibles al retardo, tales como VoIP.
- Control de la pérdida de paquetes durante el tiempo inevitable de congestión.
- Configurar las propiedades del tráfico en toda la red.
- Monitorizar e impedir la congestión de la red

Las dos arquitecturas QoS utilizadas en las redes IP son los modelos de *IntServ* y *DiffServ*. Estos servicios se diferencian por dos características: la manera en que posibilitan a las aplicaciones enviar datos, y la forma en que las redes deciden la clasificación de los datos respectivos con un nivel de servicio específico.

Existe un tercer modelo de servicio conocido como *best-effort*, el cual se define por defecto en las redes que carecen de QoS. En resumen, la siguiente lista define estos tres modelos de servicio [5]:

- *Best-effort*: La conexión estándar sin ningún tipo de garantías. Este tipo de servicio se basa en las colas "primero en entrar primero en salir" (FIFO), en las que simplemente se retransmiten los paquetes conforme van llegando sin ningún tipo de tratamiento preferente.
- Servicios Integrados: *IntServ* se basa en la reserva de recursos. En otras palabras, el modelo *IntServ* implica que los recursos necesarios para el flujo de tráfico se reservan explícitamente por todos los sistemas y medios de intermediarios.
- Servicios Diferenciados: *DiffServ* está basado en clases de servicio, donde algunas clases de tráfico reciben tratamiento preferencial sobre otras. Se utilizan preferencias estáticas, por lo que no ofrecen una gran garantía como la de los servicios integrados. Por lo tanto, *DiffServ* categoriza el tráfico y lo clasifica en colas de diferentes prioridades.

Elegir uno u otro modelo dependerá de las aplicaciones soportadas, el avance de la tecnología en cuanto a la velocidad y ancho de banda, y el coste.

1.1.3. Relevancia del *streaming* de vídeo: tráfico YouTube.

Además de este aumento de dispositivos en la red, las previsiones apuntaban a una importante mejora en la calidad de vídeo. Si echamos la vista atrás podemos corroborar este hecho ya que, en la actualidad, se apuesta cada vez más por los contenidos en alta definición

dejando atrás resoluciones de una calidad estándar. En las siguientes figuras [4] se muestran las perspectivas de evolución en este sentido; en 2018, más del 20% del conjunto de televisores de pantalla plana conectados serán 4K.

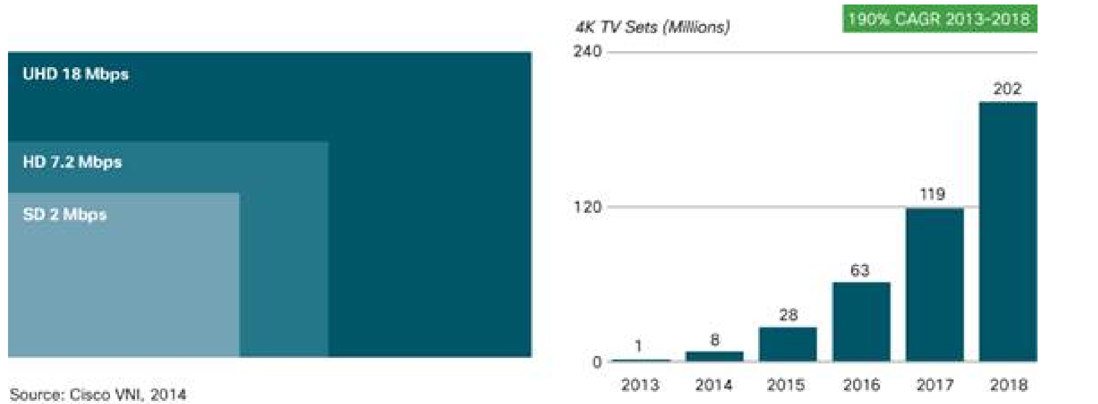


Figura 1.1. Incremento de la definición de vídeo estimado.

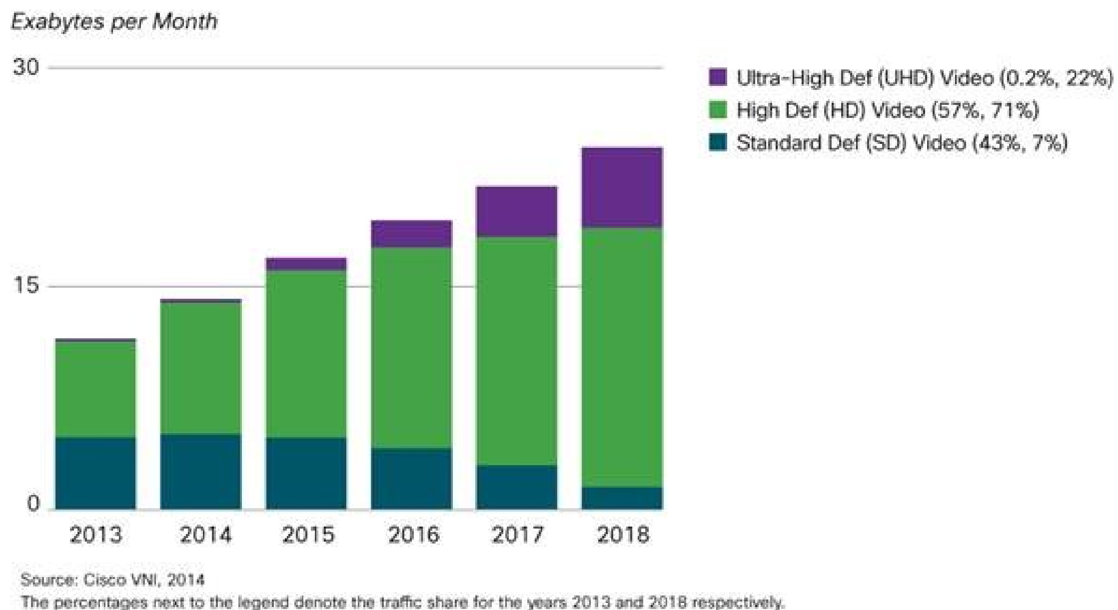


Figura 1.2. Consumo previsto de diferentes calidades de vídeo.

Según estos gráficos, los videos de calidad más baja decrecerán con el paso del tiempo, lo que influirá en el incremento tanto del vídeo en HD (*High Definition*) y UHD (*Ultra High Definition*).

Por lo tanto, es evidente que la calidad del servicio en la red se hace indispensable para soportar el tráfico que se generará en los próximos años.

Por último, se muestra una figura [4] en la que se referencian los servicios más demandados en la red para los próximos años. Partiendo de la base del aumento de prácticamente todos ellos, el porcentaje en que se prevé que evolucionen dichos servicios es bastante esclarecedor. Según esta gráfica, el vídeo en Internet es el único que aumentará significativamente su demanda en cuanto a porcentajes, aunque el tráfico Web seguirá teniendo una gran relevancia en la red.

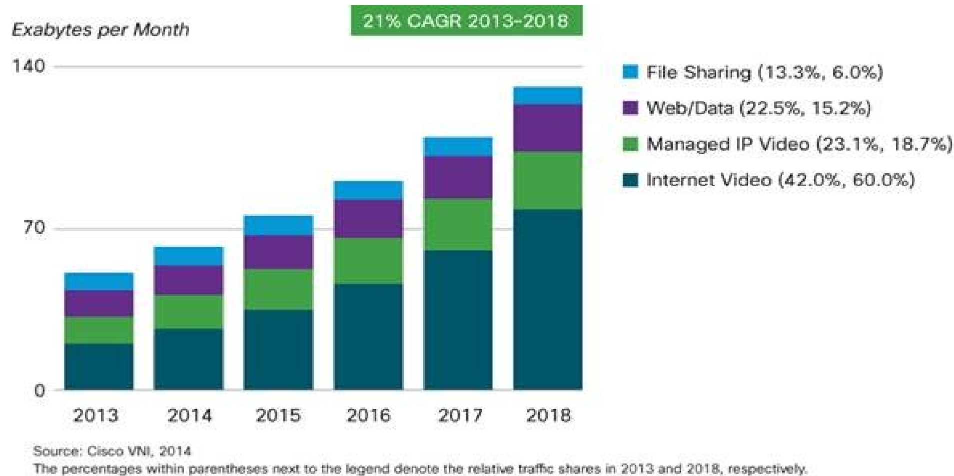


Figura 1.3. Tráfico IP Global según aplicación.

Dentro del tráfico de vídeo que se solicita en toda la red cabe destacar el que se encuentra en las páginas *web* más conocidas como YouTube, Hulu o Netflix, que ya en el año 2010 representaban el 40% del tráfico en Internet. Precisando más, YouTube se alza con diferencia con el primer puesto en cuanto a demanda de sus servicios como se puede apreciar en la siguiente figura [6].

Figura 1.4. Comparativa en términos de búsqueda de YouTube, Hulu y Netflix.

Pero su grado de influencia no es pionero únicamente a nivel de servicio de audio y vídeo, sino que también se sitúa entre las tres primeras posiciones de las páginas webs más visitadas a nivel global, siendo la primera de un servidor de *streaming* de vídeo.

Para más información acerca de la relevancia de YouTube [7] podemos echar un vistazo a sus estadísticas oficiales [8]. Como datos más relevantes podemos destacar:

- el aumento en la demanda de vídeos año tras año, incrementando en un 50% la visualización de contenidos.
- la mitad de las reproducciones de vídeo de YouTube tienen lugar en dispositivos móviles, aumentando los ingresos más de un 100% cada año.
- Su capacidad para atraer a los anunciantes de pequeñas empresas.

Una característica importante de YouTube es su sistema para la transmisión de vídeo, la cual se realiza con una descarga continua utilizando el protocolo de transporte TCP. Al solicitar un vídeo se envía una ráfaga inicial considerable de paquetes y poco a poco continúa llenándose el *buffer del reproductor*, permitiendo la visualización del contenido sin pérdidas de paquetes gracias a TCP e intentando minimizar la posibilidad de cortes en la reproducción. Además, este método permite que el vídeo pueda reproducirse sin necesidad de esperar a que se realice la descarga completa.

1.1.4. ¿Por qué apostar por las nuevas redes SDN?

Partiendo de lo mostrado en el apartado anterior, la conclusión nos lleva a pensar en la necesidad de nuevas alternativas y mejoras de la red actual que la adapten a las demandas futuras. Hacer frente a los requerimientos de telecomunicaciones por parte del público mundial se antoja harto complicado con las redes tradicionales. Por ello, los departamentos de IT de infinidad de empresas y proveedores de servicios de redes han comenzado a invertir en el aprovechamiento de las redes actuales. Sin embargo, ésta es una solución temporal, ya que ninguna arquitectura de red actual está diseñada para satisfacer las demandas actuales de usuarios, empresas y proveedores.

Las limitaciones de las redes actuales incluyen [9]:

- **Complejidad**: Los protocolos tienden a definirse para realizar funciones individuales, de forma que resuelven problemas específicos y no se benefician de una acción conjunta (mediante abstracciones).
- **Políticas inconscientes:** Para implementar una política que abarque a la red completamente, los administradores de red deben configurar miles de mecanismos y dispositivos.
- **Imposibilidad de escalabilidad**: A la vez que las demandas 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 vendedor:** Las nuevas capacidades perseguidas por proveedores y empresas en respuesta a las necesidades dinámicas de los negocios y la demanda de los clientes se ven frenadas por los ciclos de producción de los equipamientos por parte de los vendedores, que pueden abarcar hasta más de tres años.

Además, los servicios de red surgidos en los últimos tiempos y comentados en el apartado anterior están llevando a las redes tradicionales a sus límites. Algunos de estos factores son [10]:

- **Heterogeneidad en los patrones de tráfico**: En contraposición con las aplicaciones cliente-servidor, en las que la gran parte de la comunicación ocurre entre un cliente y un servidor, las aplicaciones modernas crean tráfico máquina a máquina mediante accesos a bases de datos y servidores, previo al retorno de los datos al usuario final.
- **Aumento de carga de trabajo considerable de los administradores de red.**
- **Aumento de los servicios de *cloud***: Este aumento, principalmente debido a la gran acogida por parte de las empresas en el plano público y privado, junto a la necesidad actual de aumentar la agilidad de acceso a aplicaciones, infraestructuras y otros recursos de telecomunicaciones y sumado a la complejidad de mejorar la seguridad de estos servicios, requieren una escalabilidad dinámica de la capacidad de cómputo, almacenamiento y recursos de red.
- ***Big Data* y el aumento del ancho de banda requerido**: Manejar un *Big Data* actualmente requiere procesamiento masivo por parte de miles de servidores. El aumento constante de éste lleva a una demanda equivalente de ancho de banda a los proveedores de red.

Para dar respuesta a estas necesidades nacen las redes definidas por *software* o SDN. Estas redes tienen como objetivo la implementación e implantación de servicios de red de una manera determinista, dinámica y escalable, evitando al administrador de red gestionar dichos servicios a bajo nivel. Esto permite a la red adaptarse al entorno dependiendo de cómo de saturada esté la red o los tipos de servicios que se estén prestando en un determinado momento. Por lo tanto, se podría ofrecer una calidad de servicio acorde al estado de la red.

Además de ofrecer redes centralizadas programables que pueden atender dinámicamente las necesidades de las empresas, SDN proporciona una serie de beneficios adicionales:

- Apostando por este tipo de redes se reduce el **Capex** (*Capital Expenditures*) mediante la posibilidad de reutilizar *hardware* existente y también se reduce el **Opex** (*Operating Expense*) ya que SDN permite un control algorítmico de los elementos de red como

switches o *routers* que cada vez son más programables, haciendo más sencilla la configuración y gestión de las redes. A esto se añade la reducción del tiempo de gestión por parte de los administradores, lo que reduce la probabilidad de error humano.

- Estas redes proporcionan **agilidad** y **flexibilidad**, permitiendo a las organizaciones desplegar aplicaciones, servicios e infraestructuras rápidamente para alcanzar los objetivos propuestos por empresas en el menor tiempo posible.
- Permite **innovar** creando nuevos tipos de aplicaciones y modelos de negocio por parte de las empresas, lo cual repercute en su beneficio y aumenta el valor de las redes.

Figura 1.5. Cambio de paradigma propuesto por SDN.

1.2. Logros y aportaciones.

El principal objetivo al comienzo de este proyecto consistía en la familiarización con las redes SDN, el controlador *OpenDayLight* [11] y su implementación en *Mininet* [12] además de conseguir implementar un DPI en el propio controlador que fuese capaz de etiquetar el tráfico según el servicio que ofreciese. De este modo, la aportación fundamental sería el desarrollo de dicho *Deep Packet Inspector* que permitiese detectar tráfico de YouTube y etiquetarlo con un *ToS* (*Type of Service*) [13] que lo diferenciara de otros tipos de tráfico a la hora de proporcionar cierta QoS en la red.

Por otra parte, se exponen dos soluciones según la arquitectura de OpenDayLight: una para AD-SAL (*API-Driven Service Abstraction Layer*) y otra en MD-SAL (*Model-Driven Service Abstraction Layer*). Mientras que la primera es la más explotada hasta ahora, la segunda apunta a ser la más utilizada en los próximos años para el desarrollo de las redes SDN.

Como punto adicional se ha conseguido desacoplar el controlador y *switch* correspondientes a la red SDN de tal forma que funcione como un producto independiente y compatible con otro tipo de redes, de tal forma que el tráfico pase por nuestro controlador y éste pueda modificar el tráfico y devolverlo a la red sin ningún tipo de incompatibilidad.

1.3. Organización de la memoria.

Este apartado ilustra cómo se va a estructurar la memoria de tal forma que se facilite la ubicación de los diferentes aspectos a tratar al lector. La presente memoria consta de nueve capítulos y siete anexos que se describen a continuación:

Capítulo 1: Motivación e introducción

En este capítulo se expondrán los diferentes aspectos que han motivado la realización de este proyecto así como una descripción introductoria de conceptos básicos de la tecnología implicada en la elaboración del mismo. Finalmente se presenta una recopilación de todos los logros y aportaciones que tratan de plasmar el alcance del proyecto.

Capítulo 2: Tecnologías implicadas

En este capítulo se profundizará en las tecnologías implicadas en este proyecto, centrando la atención en primer lugar en las redes SDN y a continuación exponiendo el protocolo OpenFlow. Dentro del protocolo OpenFlow se describirán sus componentes, mensajes y la estructura de los *switches* que lo soportan, y se llevará a cabo un recorrido por las diferentes versiones OpenFlow que han ido apareciendo a lo largo de su desarrollo. Por último se analizará *Open vSwitch*, un *software* que permite la virtualización de *switches* y que nos permitirá implementarlos en estas redes.

Capítulo 3: Estado del arte

El capítulo correspondiente al estado del arte trata de acercar al lector al entorno de los *Deep Packet Inspectors* que actualmente predominan en el mercado. Se expondrán DPIs tanto *open source* como comerciales y se realizará posteriormente una comparativa con el DPI que se propone en este proyecto.

Capítulo 4: Análisis de objetivos, requisitos y metodología

Este capítulo engloba cuatro bloques: el análisis de objetivos, la especificación de requisitos, una valoración acerca del alcance de los objetivos y la metodología seguida.

El primer bloque tratará de exponer los objetivos marcados para el desarrollo del proyecto. En el segundo se desglosarán los requisitos, tanto funcionales como no funcionales, que son necesarios para trabajar. Posteriormente se valorará si se han cumplido las expectativas marcadas al inicio del proyecto. Por último, se explicará cual ha sido la metodología que se ha seguido.

Capítulo 5: Planificación y estimación de costes

Este capítulo plasma detalladamente todos lo relacionado con la planificación temporal del proyecto, definiendo cada una de las fases en las que éste se divide y aportando una estimación del tiempo invertido en cada una de ellas.

Por otro lado, se analizarán todos los recursos necesarios y se estimarán los costes totales asociados a éstos con el objeto de presentar un presupuesto final en el que se muestre el coste total del proyecto.

Capítulo 6: Herramientas utilizadas

En este capítulo se presentarán las herramientas de más peso en este proyecto: *Mininet* y *OpenDayLight*.

En lo referente a *Mininet* se justificará su uso frente a otras herramientas parecidas y se expondrán diferentes posibilidades para la creación de topologías. En cuanto a *OpenDayLight*, también se justificará su uso frente a otros controladores y, además, se hará un repaso de las diferentes distribuciones de las que se dispone, se compararán los dos modelos de los que hace uso, se mostrará su interfaz *web* y se realizarán ejemplos en cuanto a la instalación tanto de flujos proactivos como reactivos.

Capítulo 7: Diseño e implementación de un DPI

Con este capítulo se pretende plasmar la implementación de nuestro DPI haciendo uso de los dos modelos disponibles: AD-SAL y MD-SAL. Previamente se hará referencia a una serie de conceptos relevantes para la correcta comprensión del DPI. Además, se explicará detalladamente la detección de tráfico YouTube, cuya implementación es la más importante y compleja dentro del DPI.

Capítulo 8: Implementación de DPI independiente y evaluación de resultados

En este capítulo se pasará a explicar cómo introducir nuestro controlador junto al DPI implementado en una máquina independiente. A este controlador se conectarán equipos externos (diferentes máquinas virtuales) que realizarán peticiones de vídeo de YouTube para comprobar que el flujo se instala correctamente gracias a la gestión del controlador de red. Por lo tanto en este capítulo se trata de demostrar la flexibilidad de nuestro DPI para introducir una red SDN dentro de las redes actuales.

Capítulo 9: Conclusiones y vías futuras

Con este capítulo concluye la memoria y será el que exponga las conclusiones a las que se han llegado una vez finalizado el proyecto. Así mismo, se propondrán una serie de posibles trabajos futuros a partir de este proyecto que mejorarían sus prestaciones. Por último, se ha incluido una valoración personal sobre todo lo que este trabajo ha acontecido.

Anexo A: Manual de instalación de *Mininet*

Se trata de un manual para la instalación de la herramienta de emulación *Mininet*. En este anexo se describen de manera detallada las diferentes posibilidades para instalar la herramienta dependiendo de los requisitos previos del usuario.

Anexo B: Instalación de *OpenDayLight* y compilación de nuevos paquetes

Este anexo detalla como instalar las distintas distribuciones que ofrece el controlador *OpenDayLight*. Además, se explica cómo se deben compilar los nuevos paquetes para agregarlos al controlador utilizando *Maven*.

Anexo C: Manual de usuario de la aplicación

En este anexo se detalla cómo poner en marcha el controlador *OpenDayLight* junto al DPI a la vez que se inicia una topología creada con *Mininet*. Por lo tanto será un manual básico de como comenzar a trabajar con estas herramientas al mismo tiempo.

Anexo D: Complicaciones y errores

Este anexo expone una serie de complicaciones y errores que se han encontrado a lo largo del proceso de desarrollo y que pueden ser útiles para cualquier usuario que comience a trabajar en este ámbito. Los errores incluidos tratan tanto sobre *OpenDayLight* como sobre *Mininet*.

Anexo E: Blog en *WordPress*

En este anexo presentamos el *blog* que se ha realizado junto con la ayuda de Cristian Alfonso Prieto Sánchez en el que se trata de acercar los conocimientos adquiridos en el desarrollo de sendos proyectos acerca de SDN y, sobre todo, el desarrollo del controlador *OpenDayLight*, a cualquier persona que lo necesite. En él podemos encontrar una serie de tutoriales para la puesta en marcha, ejemplos de implementaciones complejas en el controlador, etcétera.

Anexo F: Comparativa de controladores de red

Este anexo trata de ser un apoyo en la justificación del controlador *OpenDayLight* sobre el resto de controladores de red. Para ello se realiza una comparativa exhaustiva sobre los diferentes controladores presentes en el mercado y se plasman las ventajas que *OpenDayLight* frente al resto.

Anexo E: Topologías para *Mininet*

Este anexo incluye una serie de topologías de interés para ejecutar utilizando *Mininet*. De esta forma también se ejemplifica como crear una topología básica por medio de *scripts*. Dentro de las topologías se incluye la implementación tanto de NAT como de un *Bridge* que facilitan el acceso a la red externa desde los *hosts* virtuales creados por *Mininet*.

Bibliografía

En la bibliografía se exponen las referencias de las que se han hecho uso para adquirir ciertos conocimientos e información sobre las tecnologías utilizadas, herramientas, etcétera.

Capítulo 2

Tecnologías implicadas.

En este capítulo profundizaremos en dos conceptos de gran relevancia dentro de este proyecto como son las redes definidas por *software* y el estándar OpenFlow.

2.1. SDN (*Software* Defined Network).

SDN [10] es una arquitectura emergente de red dinámica, gestionable, rentable y adaptable, ideal para el gran ancho de banda de naturaleza dinámica de hoy en día.

Lo que se pretende con este tipo de redes es separar el plano de control (*software*) del plano de datos (*hardware* que conmuta los paquetes en la red) tal y como refleja la siguiente figura y, con esto, obtener redes más programables, automatizables y flexibles.

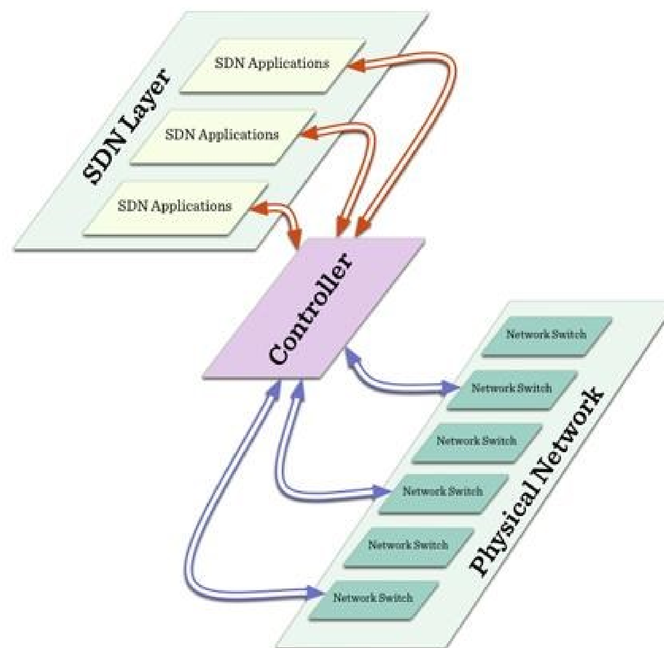


Figura 2.1. Esquema de conexión entre SDN y la red física.

En las redes actuales, la manera en que se procesan los paquetes depende de una programación previa del dispositivo, mientras que en una SDN está condicionada por una interfaz de programación con un *software* que gobierna su comportamiento. Los mensajes se envían de manera dinámica por parte de dicho *software*. Aquí radica la gran diferencia: la red pasa de ser determinista a dinámica. En la siguiente figura se aprecian las diferentes capas de las que consta una red SDN.

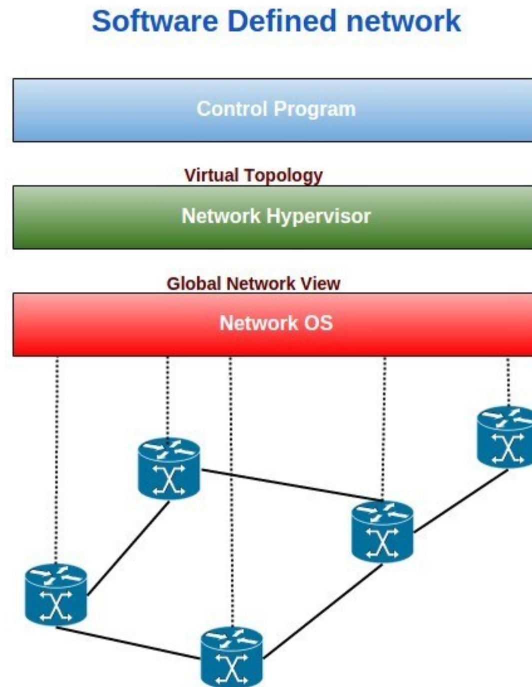


Figura 2.2. Estructura de red SDN.

Estos son los seis principios de las redes SDN y los correspondientes beneficios que ofrecen a los clientes:

- Separan de forma eficiente el *software* de red en cuatro niveles (planos): administración, servicios, control y reenvío. Esto ofrece el apoyo estructural necesario para optimizar cada plano dentro de la red.
- Centralizan los aspectos necesarios de los planos de administración, servicios y control para simplificar el diseño de red y para reducir los costes operativos.
- Utilizan la nube para una implementación flexible y con una escala adaptable, que permite una estructura de precios basada en el uso con el fin de reducir el tiempo de servicio y establecer una correlación entre costes/valor.
- Crean una plataforma para las aplicaciones de red, los servicios y la integración en los sistemas de administración, lo que permite el desarrollo de nuevas soluciones de negocio.
- Permiten una estandarización de los protocolos, lo que posibilita disponer de asistencia inter-operativa y heterogénea entre proveedores, dando lugar a la posibilidad de elección y de reducción del coste.
- Aplican con una amplia perspectiva los principios de las SDN a todos los servicios de red, incluidos los de seguridad. Esto abarca desde el centro de datos y el campus empresarial hasta las redes móviles e inalámbricas utilizadas por los proveedores de servicio.

2.1.1. Arquitectura de SDN.

Aunque la ONF (*Open Networking Foundation*) está en continuo cambio en cuanto a la terminología, los términos más comunes para los componentes de la arquitectura de la red son:

- **Aplicación SDN (*SDN App*)**: Las aplicaciones SDN son programas que comunican de forma directa las necesidades y los comportamientos deseados de su red al controlador SDN a través de los NBIs (*NorthBound Interfaces*). Están formados por una lógica de aplicación y uno o más NBIs.
- **Controlador SDN (*SDN Controller*)**: Entidad lógica de control encargada de traducir las peticiones de la aplicación SDN a las rutas de datos, dando a la capa de aplicación una visión abstracta de la red mediante estadísticas y posibles eventos. Un controlador SDN consiste en uno o más NBIs, la lógica de control SDN y el CDPI (*Control Data Plane Interface*) *driver*.
- **Ruta de datos SDN (*SDN Datapath*)**: Componente lógico que expone visibilidad y control sobre sus capacidades de reenvío y procesamiento. Está formado por un agente CDPI, un conjunto de motores de reenvío y de funciones de procesamiento, que incluyen simples reenvíos entre interfaces externas de ésta y procesamiento interno del tráfico. Las rutas de datos pueden contenerse en un único elemento de red (físico).
- **Interfaz SDN del plano de control al plano de datos (*SDN Control to Data Plane Interface, CDPI*)**: Interfaz entre el controlador y la ruta de datos, que proporciona programabilidad a la hora del reenvío, anuncio de capacidades, reporte estadístico y notificaciones de eventos.
- **Interfaces "hacia el norte" SDN (NBI)**: Son interfaces entre las aplicaciones SDN y los controladores que proporcionan vistas abstractas del comportamiento de la red y sus requerimientos.
- **"Controladores" y "agentes" de interfaz (*Interface Drivers & Agents*)**: Cada interfaz es implementada por un par de este tipo, que representa el fondo (relacionado con la infraestructura) y la cima (relacionado con la aplicación).
- **Gestión y administración (*Management & Admin*)**: El plano de gestión cubre tareas estáticas, manejadas mejor fuera de los planos de aplicación, control y datos, como la asignación de recursos a los clientes, la configuración de equipos físicos y la concordancia entre alcanzabilidad y credenciales entre entidades físicas y lógicas.

En la siguiente figura se muestra el esquema gráfico de la arquitectura de las redes SDN y todos sus componentes.

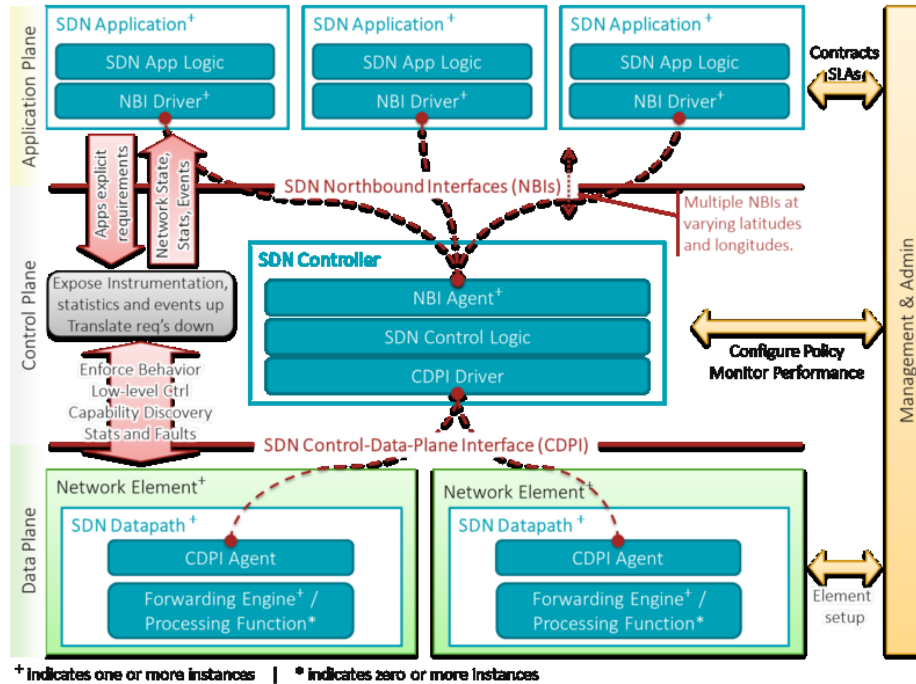


Figura 2.3. Arquitectura de SDN.

2.2. Protocolo OpenFlow.

OpenFlow [14] [15] es un protocolo de comunicaciones diseñado para dirigir el manejo y enrutamiento del tráfico en una red conmutada.

Este protocolo ha sido diseñado para que se apoye en tres componentes:

1. Las tablas de flujos instaladas en cada uno de los *switches* que indicarán a cada dispositivo qué hacer con el tráfico.

2. El controlador, que será la "inteligencia" que dialogue con todos los *switches* y les transmita a estos la información que necesiten.

3. Dispositivos (*switches*) con soporte para OpenFlow.

En un *router* o *switch* clásico, el paquete de mayor velocidad de reenvío y las decisiones de mayor alto nivel ocurren en la misma estrategia. Un *switch* OpenFlow separa estas dos funciones. La parte de reenvío de paquetes sigue residiendo en el *switch*, mientras que las decisiones de enrutamiento al más alto nivel se trasladan a un controlador separado de éste; típicamente un servidor estándar. El controlador y el *switch* se comunican a través del protocolo OpenFlow, el cual define los mensajes, tales como paquetes recibidos, paquetes enviados, modificación de tablas de encaminamiento y la identificación de estados.

El *data path* de un *switch* OpenFlow viene definido por una tabla de flujos que contiene una serie de campos de las cabeceras de los paquetes y una acción (tal como reenvío de puerto, modificación de campo o descartar). Cuando un *switch* OpenFlow recibe un paquete que no se ha recibido anteriormente, para el cual no tiene entradas de flujo de paquetes coincidentes, reenvía ese paquete al controlador. El controlador toma la decisión para gestionar dicho paquete, pudiendo descartarlo o bien agregar una entrada en el *switch* para indicarle cómo reenviar los paquetes similares en el futuro.

Por lo tanto, OpenFlow nos permite desplegar fácilmente una estrategia de reenvío innovadora e implementar protocolos de conmutación de red de forma centralizada y con una visión global. Se utiliza para aplicaciones tales como la movilidad de máquinas virtuales, redes de alta seguridad y redes de nueva generación.

2.2.1. OpenFlow *Switch*.

El protocolo OpenFlow permite la comunicación entre el *switch* y el controlador, separando así el plano de datos del plano de control. Para ello, los *switches* deben ser capaces de establecer comunicación con los controladores. Si el *switch* conoce la dirección del controlador, éste inicia una conexión TCP estándar con el controlador. En la siguiente figura se muestra la diferencia entre un *switch* tradicional y un *switch* OpenFlow.

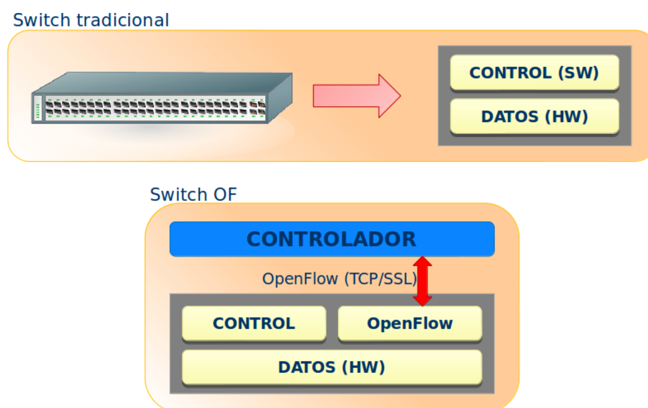


Figura 2.4. Diferencias entre un switch tradicional y un switch OpenFlow.

Los *switches* OpenFlow [15] se pueden dividir en tres partes: las tablas de flujo, con una acción asociada a cada entrada; la seguridad del canal que conecta al *switch* con el controlador; y, por último, el protocolo OpenFlow, el cual proporciona un camino abierto y estandarizado que permite la comunicación entre el *switch* y el controlador. La siguiente imagen describe estas tres partes de forma clara:

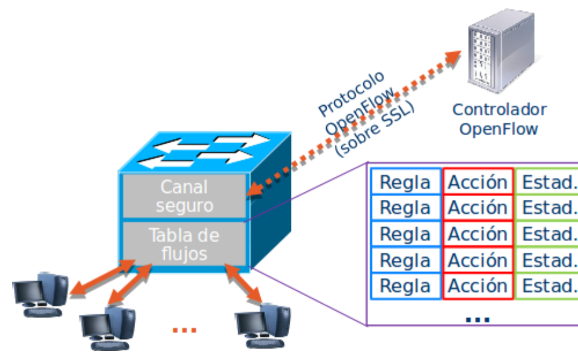


Figura 2.5. Esquema de conexión de switch OpenFlow.

Los *switches* OpenFlow dedicados son elementos simples de *datapath* que envían paquetes entre puertos según se indique por parte del controlador. En la figura siguiente [16] podemos apreciar un *switch* OpenFlow ideal; los flujos son ampliamente definidos y limitados solo por las capacidades de una implementación en particular de la tabla de flujos.

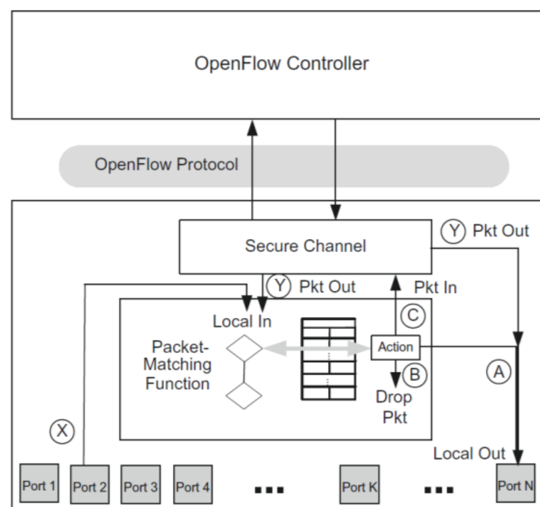


Figura 2.6. Esquema general de un switch OpenFlow v1.0.

A cada flujo se le asocia una acción, siendo las tres acciones básicas:

1. Enviar los paquetes del flujo en cuestión a un puerto o puertos establecidos, permitiendo el enrutado de los mismos en la red.
2. Encapsular y enviar flujos al controlador. Dicho paquete se envía por un canal seguro.
3. Eliminación de flujos, contribuyendo a potenciar la seguridad en algunos casos como, por ejemplo, frenar los ataques de denegación de servicio o reducir el descubrimiento de tráfico falso de *broadcast* por parte de los usuarios finales.

2.2.2. Componentes del protocolo OpenFlow.

Nos disponemos a presentar diferentes componentes del protocolo [17] necesarios para su funcionamiento como son los puertos (lógicos y físicos), las tablas de flujo, el *matching* o las acciones asociadas a las entradas de flujo.

Puertos OpenFlow.

Los puertos OpenFlow son las interfaces de red por las que pasan los paquetes entre el procesado OpenFlow y el resto de la red. La conexión entre los *switches* OpenFlow y los demás *switches* se realiza a través una conexión lógica mediante los puertos OpenFlow.

Un *switch* OpenFlow genera un número de puertos disponibles para el procesado OpenFlow. Este conjunto de puertos no tiene que ser idéntico al conjunto de interfaces que proporciona el *hardware* del *switch*, por lo que algunas interfaces son deshabilitadas por OpenFlow, del mismo modo el *switch* OpenFlow define puertos adicionales. En general, OpenFlow define tanto puertos físicos como lógicos:

- Los puertos físicos vienen definidos por el *switch* y corresponden a una interfaz *hardware* del mismo.
- Los puertos lógicos no tienen que corresponder directamente con una interfaz *hardware* del *switch*. Son de una abstracción de alto nivel que debe ser definida en los *switches* que no usan los métodos OpenFlow. Estos puertos incluyen encapsulación de paquetes y mapean a varios puertos físicos. El proceso realizado por los puertos lógicos debe ser transparente al procesado OpenFlow y esos puertos deben interaccionar con OpenFlow como si fueran puertos físicos. La diferencia entre puertos lógicos y físicos es que 1) un paquete asociado a un puerto lógico debe tener un campo de metadatos extra llamado *Tunnel-ID* asociado y 2) cuando un paquete recibido por un puerto lógico se manda al controlador, los puertos lógicos y sus puertos físicos subyacentes informan al controlador.

Por otra parte cabe mencionar que OpenFlow define una serie de puertos reservados como son *ALL* (representa todos los puertos por los que el *switch* puede reenviar), *CONTROLLER* (envía el paquete al controlador), *TABLE* (representa el comienzo del *pipeline* de OpenFlow), *IN_PORT* (puerto por donde se recibe el paquete), *ANY* (cualquier puerto cuando ningún puerto se especifica), *LOCAL* (representa la red local del *switch*), *NORMAL* (representa el proceso *pipeline* que no es OpenFlow) y *FLOOD* (reenvía el paquete por todos los puertos menos por el de entrada).

Tablas OpenFlow.

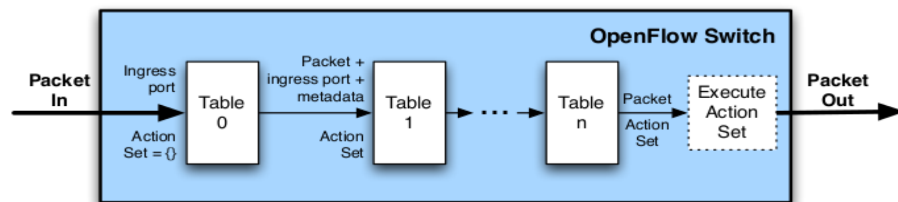
En esta sección se describirán los componentes de las tablas de flujos y el grupo de tablas, además de los mecanismos de *matching* y el gestionador de acciones.

· Proceso *pipeline*.

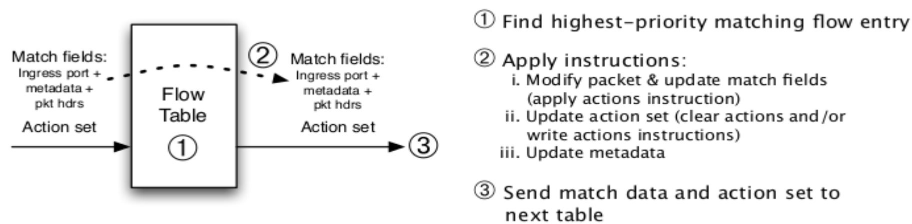
Los *switches* OpenFlow pueden dividirse en dos tipos: *OpenFlow-only*, y *OpenFlow-hybrid*. Los primeros soportan únicamente las operaciones relacionadas con el protocolo OpenFlow; en estos *switches* todos los paquetes son procesados por el *pipeline* de OpenFlow, y no pueden ser manejados de otro modo.

Los *switches* híbridos de OpenFlow (*OpenFlow-hybrid*) ofrecen soporte tanto de las funcionalidades de OpenFlow como las normales que aporta la conmutación Ethernet como, por ejemplo, el aislamiento de VLANs, enrutado de la capa 3 (enrutado IPv4, enrutado IPv6...), ACL (*Access Control List*) o procesado de QoS. Estos *switches* deberían proporcionar un mecanismo de clasificación independiente a OpenFlow que enrute el tráfico tanto si se realiza desde el *pipeline* de OpenFlow como si se hace desde el *pipeline* normal. Este mecanismo de clasificación no se contempla en la especificación de OpenFlow. Otra característica de este tipo de *switches* es que deben permitir moverse entre el *pipeline* de OpenFlow y el normal por los puertos reservados *NORMAL* y *FLOOD*.

El *pipeline* de OpenFlow de todos los *switches* contiene tablas de flujos, y cada tabla de flujos contiene múltiples entradas. El proceso *pipeline* de OpenFlow define como interaccionan los paquetes con las tablas de flujo. Un *switch* OpenFlow con una única tabla de flujo tendría un proceso *pipeline* muy simplificado, tal como muestra la siguiente figura.



(a) Packets are matched against multiple tables in the pipeline



(b) Per-table packet processing

Figura 2.7. Flujo de paquetes sobre el procesado pipeline.

Las tablas de flujo de un *switch* OpenFlow son numeradas secuencialmente, empezando por 0. El proceso de *pipeline* siempre comienza por la primera tabla de flujo: en primer lugar se comprueba el *match* entre el paquete y las entradas de flujo de la tabla 0. A partir de ahí, se utilizarán otras tablas en función del resultado del *match* con la primera tabla y del tipo de versión del protocolo OpenFlow.

Tablas de flujo.

Una tabla de flujos está compuesta por una serie de entradas de flujos. Cada tabla de flujos contiene los siguientes campos:

Campos de Match	Prioridad	Contadores	Instrucciones	Timeouts	Cookie

Tabla 2.1. Principales componentes de una entrada de flujo.

- Campos de *match*: para comprobar que el paquete coincide con esta entrada de flujo. Consiste en el puerto de entrada del paquete y las cabeceras del paquete, y opcionalmente especificación de metadatos por una tabla previa.
- Prioridad: precedencia de *matching* de una entrada de flujo.
- Contadores: para actualizar los paquetes con *match*.
- Instrucciones: para modificar el conjunto de acciones o el proceso de *pipeline*.
- Timeouts: tiempo máximo antes de que un flujo expire en un *switch*.
- Cookie: es un dato seleccionado por el controlador. Su cometido es filtrar las estadísticas de flujos, la modificación de los mismos y su eliminación.

Una entrada en una tabla de flujo se puede identificar por los campos de *match* y su prioridad.

Matching.

La siguiente figura esquematiza el proceso que sigue un paquete al en el *switch* y procederse a comprobar las tablas de flujo.

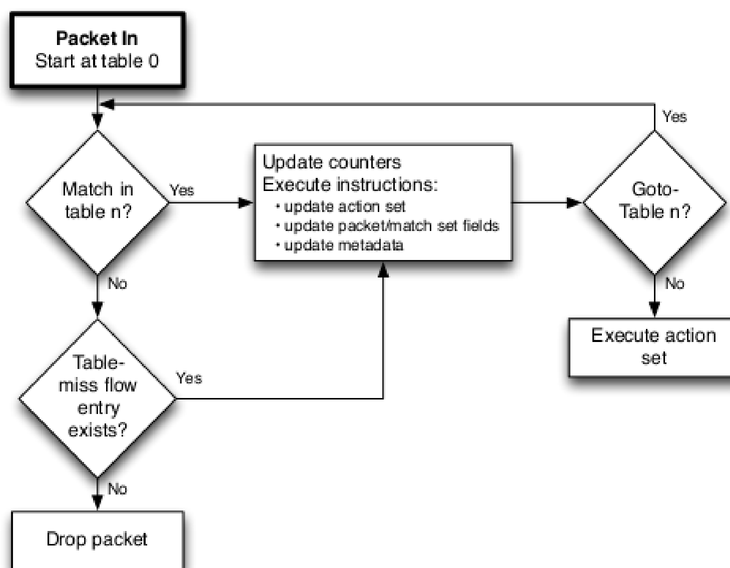


Figura 2.8. Diagrama de flujo del flujo de paquetes en un switch OpenFlow.

El *switch* comienza realizando una búsqueda en la primera tabla de flujo, y basado en el procesamiento *pipeline*, realizará búsquedas en otras tablas de flujo en caso de tenerlas. Este proceso dependerá de la versión de OpenFlow que se esté utilizando.

El campo de *match* se extrae de los paquetes y se utiliza para esta búsqueda dependiendo del tipo de paquete. Como comentamos anteriormente, éste suele depender de las cabeceras y otros campos como los metadatos.

A un paquete se le aplicará una entrada de una tabla de flujos si el valor del campo de *match* de dicho paquete coincide con el de dicha entrada. En caso de encontrar varias entradas con *match* se optará siempre por utilizar la entrada con mayor prioridad.

Acciones

Los *switches* no tienen que soportar todos los tipos de acciones definidos en el estándar. El controlador indica los tipos de acciones que soporta durante la negociación inicial. Algunas acciones que interesa destacar son las siguientes:

- *Output*: Con esta acción reenviamos el paquete a un puerto OpenFlow especificado.
- *Drop*: Esta acción descarta el paquete.
- *Set-Field*: Esta acción es importante en nuestro proyecto ya que se encarga de modificar un valor en la cabecera del paquete. En nuestro caso la utilizaremos para añadir un nuevo *ToS*.
- *Change-TTL*: Se encarga de modificar el campo TTL del paquete.

2.2.3. Mensajes OpenFlow.

OpenFlow define una serie de mensajes para realizar la comunicación entre las diferentes partes implicadas en el proceso de encaminamiento de paquetes por las redes SDN. En este apartado se hará mención a cada uno de ellos y se detallará el cometido de los mismos.

Los mensajes OpenFlow pueden ser de tres tipos: controlador a *switch*, asíncrono y síncronos.

- **Mensajes del controlador al *switch***. Se generan en el controlador, y el conmutador necesariamente responde al mensaje. En esta categoría se encuentran los siguientes mensajes:
  - *Features*: Se envía cuando el controlador requiere obtener las características del *switch*. Se utiliza normalmente en el establecimiento de una conexión OpenFlow.
  - *Configuration*: Mensajes de consulta de los parámetros de configuración.
  - *Modify-State*: Estos mensajes los envía el controlador para gestionar los estados en el *switch*. Un claro ejemplo podría ser el de añadir o quitar flujos o cambiar el estado de un determinado puerto.

  - *Read-State*: Su cometido es adquirir las características del *switch* por parte del controlador.
  - *Packet-Out*: Con estos mensajes el controlador puede enviar paquetes por un determinado puerto del *switch* o redireccionar paquetes *Packet-In*. Antes de ser redireccionado encapsula el paquete y las acciones que se aplicarán al mismo.
  - *Barriel*: El controlador los usa para asegurarse de que las dependencias de un mensaje se han cumplido o para recibir notificaciones sobre operaciones finalizadas.
- **Mensajes asíncronos**. Se envían entre el *switch* y el controlador al llegar un paquete. Estos mensajes son:
  - *Packet-In*: Lo envía el *switch* al controlador cuando no tiene una entrada en su tabla de flujos para el paquete entrante al mismo. El controlador procesa el paquete y contesta con un mensaje *Packet-Out*.
  - *Flow-Removed*: Se envía cuando el tiempo de inactividad de un flujo expira. Este mensaje puede ser enviado tanto por el controlador como por el *switch*.
  - *Port-Status*: Se envía desde el *switch* al controlador cuando cambia la configuración de un puerto.
  - *Error*: El *switch* lo utiliza para notificar la existencia de algún problema con estos mensajes.
- **Mensajes síncronos**. Se envían desde cualquier dispositivo sin solicitud previa. Dentro de este tipo de mensajes se encuentran los siguientes:
  - *Hello*: Son mensajes que se intercambian al momento del establecimiento de la conexión entre los conmutadores y el controlador.
  - *Echo*: Su función es medir la latencia o el ancho de banda para verificar que un dispositivo esté activo. Por cada petición que llegue al destino se creará una respuesta que será enviada al origen. Estos mensajes pueden enviarlos tanto controlador como *switch*.
  - *Experimenter*: Permite dar funcionalidades adicionales a un conmutador OpenFlow. Su desarrollo está enfocado a versiones futuras de este protocolo.

2.2.4. Comparativa de Versiones OpenFlow.

El protocolo OpenFlow, al igual que toda la tecnología que engloba a las redes SDN, ha evolucionado en sus prestaciones con el paso del tiempo. Desde OpenFlow 1.0 hasta OpenFlow 1.5 se han ido implementando nuevos avances al protocolo.

La versión 1.3 es la última versión de OpenFlow que ofrece soporte por parte de los fabricantes de *switches*. Esta versión es significativamente diferente a la 1.0 (la cual fue la versión previa con soporte de varios fabricantes). Entre otras, las principales características de las diferentes versiones son:

OpenFlow 1.0.

- Canal seguro para las comunicaciones (conexión TLS con encriptación asimétrica).
- Entradas de tabla de flujo que incluyen componentes como patrón de coincidencia, prioridad, contadores, instrucciones, temporizadores y *cookies*.
- Distingue entre reglas exactas (se especifican todos los campos) en las que sólo puede haber una regla por flujo activo y reglas comodín (al menos un campo es variable) en las que varias reglas pueden ser válidas para un paquete y dependerá de las prioridades.
- Permite acciones como reenvío por puertos físicos o virtuales, encolamiento, descarte e incluso modificación de algunos campos (añadir/eliminar etiquetas VLAN, *ToS*, TTL, dirección IP, puerto, etcétera).

OpenFlow 1.1.

- Soporte para MPLS, Q-inQ, VLANs, *multipath*, múltiples tablas de flujos y puertos lógicos.
- Posibilita las tablas encadenadas (GOTO), véase la siguiente figura.
- Permite añadir/modificar/borrar conjunto de acciones que se ejecutarán al finalizar el procesado.
- Añade acciones inmediatas, ejecutadas en el transcurso del procesamiento del paquete.
- Se añade el concepto de grupos: conjunto de acciones aplicadas a grupos de flujos.
- Incluye mecanismos mejorados frente a posibles fallos de conexión al controlador. El modo de fallo seguro permite continuar la operatividad salvo los paquetes con destino al controlador, que son descartados. El modo de fallo solitario permite al *switch* OF operar como un *switch* normal.

OpenFlow 1.2.

- Soporte para extensión de cabeceras (*match*, *packet-in, set-field*), IPv6.
- Posibilidad de conexión a varios controladores.

OpenFlow 1.3.

- Soporte de *tunneling* y provisión de *Backbone Bridging*.
- Uso de MULTIPART REQUEST/REPLY para solicitar capacidades y estadísticas, con formato extensible.
- Entrada de flujo por defecto para posibles fallos.
- Métrica de tráfico por flujo, lo cual permite soportar capacidades de QoS.
- Conexiones auxiliares que permiten varias conexiones paralelas al mismo controlador (primaria con TCP y secundarias con UDP).
- *Cookies* para entradas de flujo utilizadas para guardar una caché reducida de flujos antiguos y enviar sólo la *cookie* al controlador si dicho flujo vuelve a aparecer.

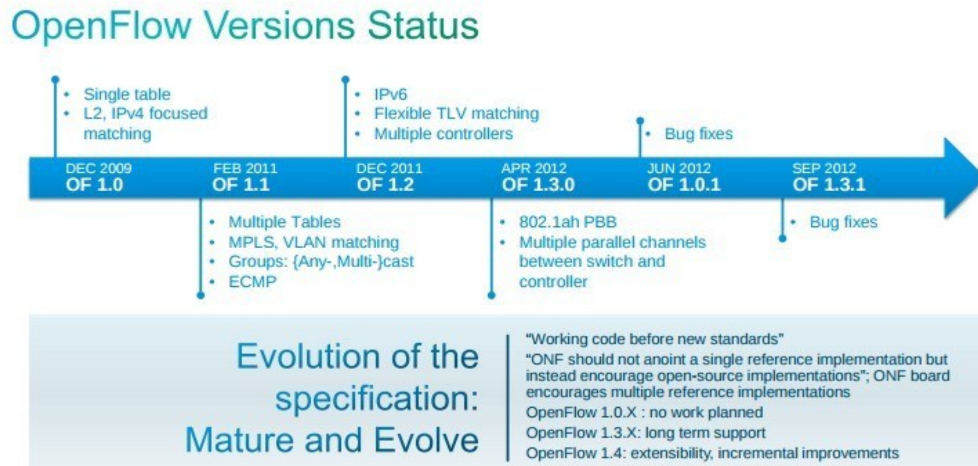


Figura 2.9. Diagrama de evolución del protocolo OpenFlow.

La versión estable más reciente y con mejores prestaciones es la 1.3, y por ello es la versión del protocolo más utilizada en este proyecto. No obstante, en algunos casos se hace preciso la utilización de la versión 1.0 por motivos de compatibilidad con la implementación del entorno. Estas dos versiones son las únicas estables de OpenFlow y es por ello que reciben un mayor apoyo de la industria.

Además de estas versiones existen dos más recientes pese a no tener soporte comercial por parte de los distribuidores de *switches*: OpenFlow 1.4 y OpenFlow 1.5. En los dos siguientes puntos vamos a mencionar algunas de las novedades que presentan.

OpenFlow 1.4.

- Mayor descripción de los paquetes entrantes en el proceso *pipeline*. Permite al controlador distinguir qué parte del *pipeline* le redirige el paquete.
- Extensión de conexión del protocolo. Algunas partes del protocolo se han adaptado a la estructura TLV (*Time, Length, Value*).
- Añade las razones relacionadas con la métrica de por qué se borran flujos.
- Monitorización de flujos. Permite al controlador definir un número de monitores, los cuales se encargarán de un rango de tablas de flujo asignado por éste. Cuando un flujo es modificado por estos monitores se indicará al controlador enviando un mensaje.
- Posibilidad de enviar un mensaje desde el *switch* al controlador indicándole un cambio de rol en caso de la existencia de varios controladores (cambio de controlador "*master*").
- Implementación de una serie de mensajes de error (prioridad errónea, error de configuración asíncrona etcétera).

OpenFlow 1.5.

- Introduce tablas de salida, posibilitando el procesado en el contexto del puerto de salida. Cuando llega un flujo, puede comenzar a ser procesado en las tablas de entrada y ser redirigido a estas tablas de salida.
- Define tipos de paquetes diferentes a *Ethernet*, mientras que en otras versiones los paquetes debían ser de este tipo.
- Posibilidad de *match* a partir de las *flags* de los paquetes TCP como *SYN, ACK* y *FIN*. Pueden ser utilizadas para detectar el inicio y el fin de las comunicaciones TCP.

2.3. Open vSwitch.

Open vSwitch es un *software* multicapa bajo la licencia de código abierto Apache 2.0. Este *software* fue diseñado para ser utilizado como un *switch* virtual en entornos de servidores virtualizados y se encarga de reenviar el tráfico entre diferentes máquinas virtuales (VMs) en el mismo *host* físico y entre las máquinas virtuales y la red física.

Open vSwitch soporta numerosas tecnologías de virtualización basadas en Linux como Xen [18], KVM [19], VirtualBox [20] o Proxmox VE [21]. También se ha integrado en muchos sistemas de gestión virtuales como OpenStack [22], OpenQRM [23], OpenNebula [24] y oVirt [25].

Permite más capacidades que los módulos regulares de kernel de Linux, aun cuando el *datapath* está dentro del propio kernel GNU/Linux, lo que le hace ideal para la construcción de esquemas de redes virtuales para nubes o para investigación de nuevos protocolos de red.

Open vSwitch dispone de un diseño de mayor complejidad que los *bridges*, estando éste compuesto por varios componentes. Mientras que los *bridges* sólo se ejecutan en el espacio del kernel del *host*, Open vSwitch, al necesitar un código más complejo para poder proporcionar todas las funcionalidades avanzadas, además de hacer uso del kernel también se ejecuta en el *user space*. El *kernel space* y el *user space* son dos separaciones lógicas de la memoria que tienen los Sistemas Operativos, para así poder proteger al sistema de fallos o ataques. En el espacio del kernel se ejecutarían los módulos y los controladores del sistema, mientras que en el espacio de usuario se encontraría la mayor parte del *software*. La siguiente figura muestra las dos separaciones lógicas Así como el camino seguido por los paquetes recibidos. Como vemos, el primer paquete lo gestiona el *user space* mientras que los siguientes los gestiona el *kernel space*.

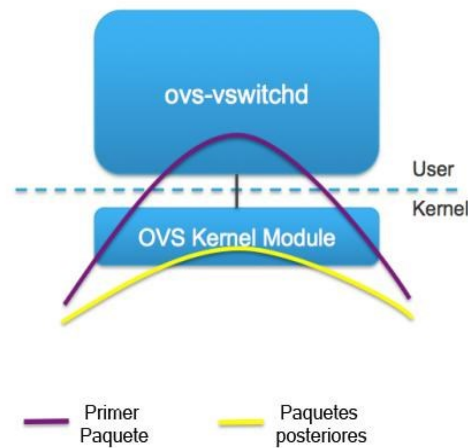


Figura 2.10. Diseño de Open vSwitch.

Los principales componentes de Open vSwitch son los siguientes:

- *Ovs-vswitchd*: Un demonio que implementa las funcionalidades del *switch*, con un módulo del kernel de Linux para conmutación basada en flujos.
- *Ovsdb-server*: Un servidor de base de datos ligero que *ovs-vswitchd* consulta para obtener su configuración. En la base de datos que gestiona se encuentran todos los parámetros de configuración del Open vSwitch, los cuales se almacenan de forma que dicha información se mantenga tras un reinicio del *host*.

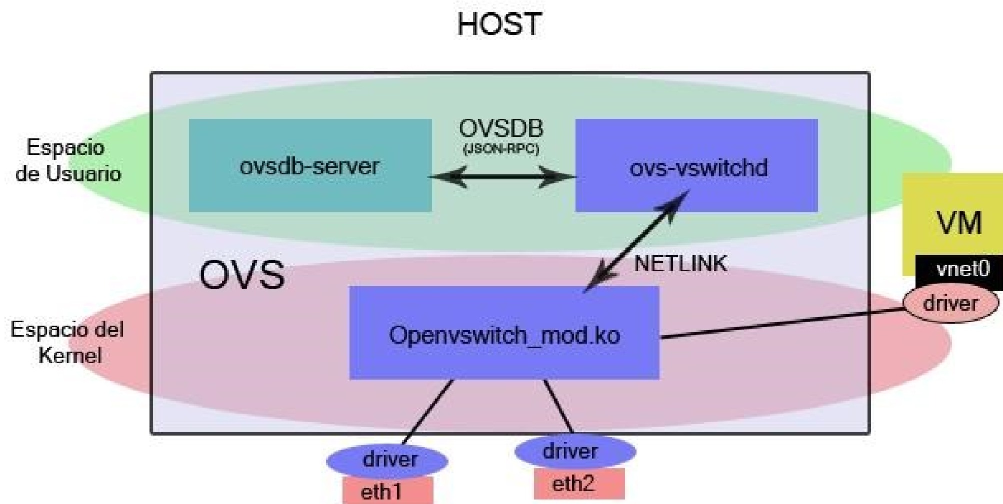


Figura 2.11. Descripción de Open vSwitch.

Algunos de los motivos por los que se ha optado por su utilización en este proyecto son su licencia de código abierto, su compatibilidad con OpenFlow 1.0 y sus extensiones posteriores

y su capacidad para definir niveles de calidad de servicio en la red como disponibilidad, ancho de banda, ratio de error, latencia, priorizar tráfico, etcétera.

Por otro lado cabe destacar la incorporación de funcionalidades de red similares a los *switches hardware*, su compatibilidad con *software* basado en *bridges* y su capacidad para poder realizar procesamiento en *hardware*, de tal forma que se mejore el rendimiento.

Capítulo 3

Estado del arte

En este capítulo vamos a tratar de acercar al lector a la actualidad referente a la inspección profunda de paquetes. La idea principal será la de dar una visión sobre estas herramientas presentes en el mercado y tratar de comparar nuestro proyecto con lo que ya existe. Para ello expondremos DPIs tanto del tipo *open source* como comerciales, alcanzando un espectro de posibilidades lo más amplio posible.

3.1. DPIs *open source*.

A continuación se presentan algunos de los DPIs de código abierto más destacados actualmente. Estos DPIs son los que más se acercan a la realización de nuestro proyecto en tanto en cuanto no son privativos.

3.1.1. *nDPI*.

Este inspector de paquetes [26] forma parte de la herramienta para monitorizar paquetes *ntop*. Hace uso de las librerías de *OpenDPI*. Publicado bajo licencia LGPL, su objetivo es ampliar la biblioteca original mediante la adición de nuevos protocolos que están únicamente disponibles en la versión de pago de *OpenDPI*. Este DPI está disponible tanto para las plataformas UNIX como para Windows, con el fin de proporcionar una experiencia multiplataforma. Además, se ha adaptado la monitorización de tráfico mediante la desactivación de características específicas que ralentizan el motor del DPI a la vez que son innecesarias para la vigilancia del tráfico de red.

nDPI se utiliza por parte de *ntop* para añadir la detección de protocolos en la capa de aplicación, independientemente del protocolo utilizado. Esto significa que es posible detectar tanto protocolos conocidos pero que no utilicen puertos estándar (por ejemplo, detectar el protocolo http que utilice un puerto distinto al 80) como protocolos no conocidos pero que utilicen puertos estándar (como por ejemplo, el tráfico *de Skype* por el puerto 80). Esto se debe a que el concepto de que el puerto depende de la aplicación se está extinguiendo.

La lista de protocolos soportados por este DPI es muy extensa y puede verse en su totalidad aquí. No obstante, caben destacar algunos como FTP, POP, SMTP, IMAP, DNS, IPP, HTTP, AVI, MPEG, RTSP, RTP, SSH, SSL o YouTube. La detección de éste último es importante ya que es uno de los tráficos que nosotros detectamos en el DPI que conforma nuestro proyecto. No obstante, este no se implementa como un módulo si no que está dentro de la detección de otros servicios, al igual que el tráfico de Facebook.

Además, este DPI incluye decodificador para certificados SSL, lo que le permite soportar conexiones encriptadas. Esto es importante ya que la tendencia del tráfico en Internet es la encriptación del contenido utilizando SSL. Así, puede detectar protocolos como Citrix Online y Apple iCloud que, de otro modo, serían indetectables. Además, este DPI incluye decodificador para certificados SSL, lo cual le permite soportar conexiones encriptadas. Esto es importante ya que la tendencia del tráfico en Internet es la encriptación del contenido utilizando SSL. Esto le permite detectar protocolos como *Citrix Online* y *Apple iCloud* que, de otro modo, serían indetectables.

3.1.2. *Bro-IDS*.

Bro Network Security Monitor [27] es una herramienta de análisis de red bastante potente. Es adaptable, lo cual permite:

- proporcionar políticas de monitorización específicas dependiendo del entorno.
- realizar un análisis profundo que facilite el reconocimiento de un gran número de protocolos.
- habilitar el análisis de la semántica de alto nivel en la capa de aplicación.

Además, es flexible, no restringiendo a detecciones particulares, sino que proporciona interfaces abiertas. Esto posibilita el intercambio de información con aplicaciones en tiempo real. Por último, es de código abierto, permitiendo su uso gratuito sin restricciones.

3.1.3. *Snort*.

Aunque no es un DPI como tal, *Snort* [28] sí que cumple algunas características que pueden ajustarse a una inspección profunda de paquetes. Esta herramienta es un sistema de detección de intrusos vinculada a aspectos de seguridad principalmente. Su objetivo es detectar o monitorizar eventos ocurridos en un determinado sistema.

Este trabajo de detección de eventos es lo que más se relaciona con un DPI. *Snort* realiza un análisis del paquete completo. Por lo tanto, previamente a tomar decisiones en cuanto a lo seguro que un paquete es para el sistema, esta herramienta debe inspeccionar el paquete profundamente en busca de patrones o características que ayuden a tomar dicha decisión.

En cuanto a su rendimiento como DPI, se ha realizado un análisis de rendimiento con la técnica benchmark. La prueba fue dirigida por NSS Labs, siendo capaz de superar 80 Gbps, que implica 60 millones de flujos, medio millón de los cuales eran de tipo TCP y HTTP-CPS, con una mezcla de tamaños de paquetes.

3.2. DPIs comerciales.

La siguiente tabla muestra una lista con los inspectores profundos de paquetes más importantes del mercado y las compañías que venden estos productos según la página *web* de *Policy Control* [29].

Ranking	Empresa	DPI
1	F5	*Policy Enforcement Manager*
2	Alcatel-Lucent	*7750 Service Router - Mobile Gateway*
3	Allot Communications	*Service Gateway Sigma E*
4	Bivio Networks	*DPI Application Platform (7000)*
5	Blue Coat Systems	*PacketShaper*
6	Citrix (ByteMobile)	*Adaptative Traffic Manager (T3100)*
7	Cisco Systems	*Service Control Engine Packet Data Network Gateway*
8	Comverse	*Policy Enforcer*
9	Ericsson	*Smart Service Router (SSR8000)*
10	Huewei Technologies	*Huewei Service Inspection Gateway (9800)*

Tabla 3.1. Lista de los DPIs comerciales más importantes.

Vamos a ver las características de algunos de ellos para hacernos una idea de lo que estos DPIs ofrecen.

3.2.1. *Policy Enforcement Manager*.

Este DPI [30] proporciona la información necesaria para entender el comportamiento del tráfico de los clientes y permite gestionarlo eficazmente, ofreciendo una amplia gama de aplicación de políticas. Con este DPI se pueden crear planes de servicio a medida, regular el uso de la red y, en última instancia, aumentar la rentabilidad. En la siguiente figura [31] podemos ver algunas de sus características.

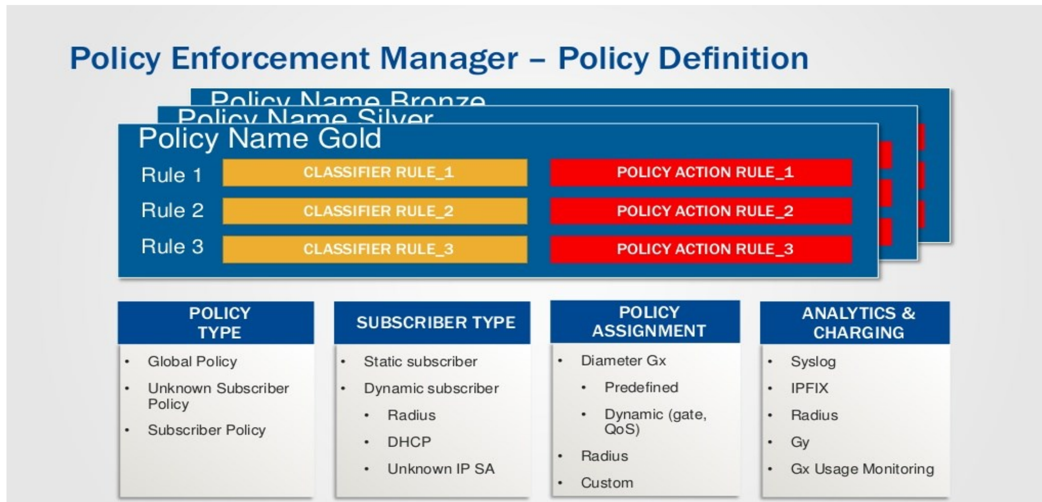


Figura 3.1. Definición de políticas de PEM.

Como otro de sus puntos fuertes expone la posibilidad de nuevos modelos de negocio y servicios. Asegura la consecución de mayores ingresos por usuario ofreciendo una mejor calidad de experiencia para los suscriptores. También ofrece la posibilidad de crear servicios adaptados basados en el uso de aplicaciones de abonado, la clasificación del tráfico y búsqueda de patrones, y lanzamiento de nuevos servicios de manera más rápida.

Por otro lado permite la monitorización de tráfico en tiempo real, posibilitando la modificación del ancho de banda dinámicamente e implementando ciertas políticas para reducir la congestión de la red. Permite también controlar la dirección del tráfico de forma inteligente en las capas 4-7 e incluye virtualización de red.

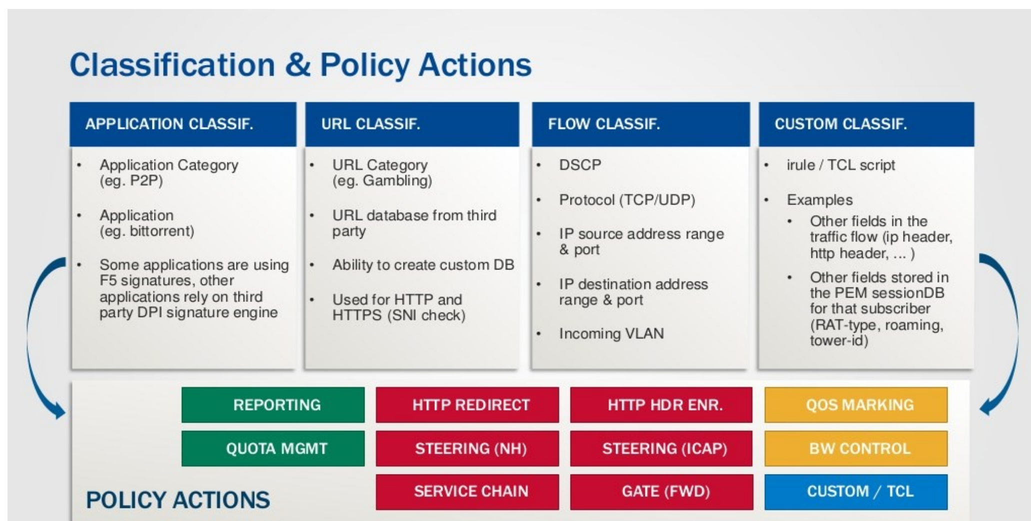


Figura 3.2. Clasificación y acciones de política.

Otra de sus características es su escalabilidad en cuanto a las sesiones de usuarios, *throughput* y transacciones por segundo. Soporta 320 gigabits por segundo de *throughput* de la capa 7 y soporta hasta 24 millones de clientes.

Una característica que se asemeja a lo que se intenta realizar en este proyecto es la capacidad para detectar tráfico de diferentes aplicaciones como pueden ser Skype o Spotify mediante la identificación OTT (*Over The Top content*).

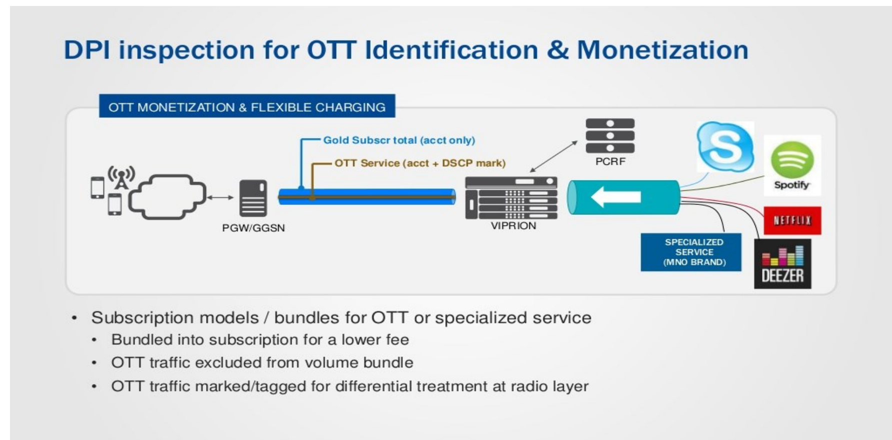


Figura 3.3. Detección de tráfico mediante identificación OTT.

Además, como veremos a continuación, este DPI también tiene cabida en las redes SDN.

Policy Enforcement Manager en redes SDN.

F5 también apuesta por las redes SDN en su DPI, ofreciendo un servicio de arquitectura flexible y dinámica que responde eficientemente a las condiciones del tráfico y a los requerimientos de las aplicaciones. En la siguiente imagen podemos ver las políticas de control de servicios en este tipo de redes.

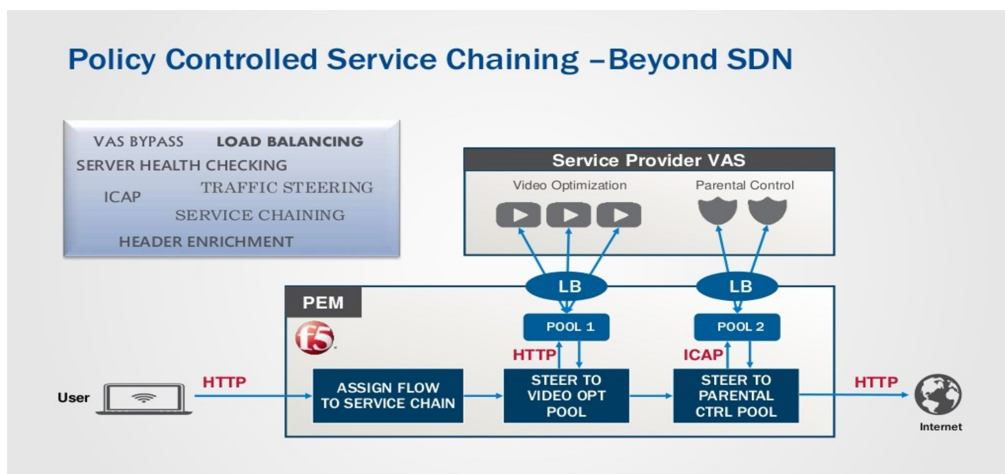


Figura 3.4. Servicio de encadenamiento del control de políticas para SDN.

Este DPI identifica y extrae información en las redes SDN en tiempo real, proporcionando una perspectiva global de tráfico real mediante la identificación de protocolos, tipos de aplicaciones y extracción adicional de metadatos.

3.2.2. 7750 Service *Router* - Mobile Gateway.

Este *router* posee capacidades de DPI [32] además de ser una pasarela para la red móvil. Por lo tanto, este DPI actúa en las capas 4-7 de las redes móviles procesando el tráfico IP.

Implementando este DPI, Alcatel pretende monitorizar, informar y optimizar los usos de los recursos de la red, proteger estos recursos, habilitar nuevos servicios y crear una política de control de la red.

3.2.3. DPI Application Platform (7000).

Este DPI [33] creado por *Bivio* ofrece un alto rendimiento y es totalmente programable. Combina un único *hardware* de procesado de paquetes con una plataforma *software* que incluye estándares basados en Linux. Está diseñado específicamente para proporcionar una velocidad de procesado de paquetes sobre los 10Gbps, fusionando componentes de procesamiento de red con las CPU de procesamiento de aplicaciones para ofrecer un rendimiento y flexibilidad muy altos. La serie *Bivio 7000* incluye dos configuraciones principales que proporcionan características de rendimiento optimizado para ofrecer un procesamiento de paquetes de línea de 4Gbps a más de 10 Gbps de rendimiento utilizando tecnología de escalado.

3.3. Comparativa con DPIs del mercado.

Vista la oferta de DPIs en el mercado actual, vamos a realizar una comparativa con lo que se ha implementado en este proyecto.

3.3.1. Comparativa con DPIs *open source*.

Quizás el DPI más competitivo de código abierto sea *nDPI*. Este DPI ofrece la detección de una cantidad de protocolos y aplicaciones, entre las que se incluye el tráfico *YouTube* tal y como permite nuestro DPI. No obstante, su integración en las redes SDN no existe, por lo que en este aspecto el DPI implementado en este proyecto se encuentra a la vanguardia en este tipo de redes.

Si comparamos con el resto de DPIs, la gran diferencia, a parte de nuestra implementación en redes SDN, es la inspección profunda del tráfico *YouTube* que se realiza y la capacidad dar soporte para calidad de servicio a partir de la detección.

3.3.2. Comparativa con DPIs comerciales.

La principal ventaja si comparamos nuestro DPI con los DPIs comerciales radica en que nuestro DPI, al ser de licencia GPL, puede ser modificado y mejorado por el usuario que desee desarrollarlo. Además, es gratuito y permite la posibilidad de mejorar bastantes aspectos en cuanto a las redes SDN a partir de las herramientas utilizadas. En cuanto al precio por el que podemos conseguir estos DPIs, como el ofrecido por *Bivio* o *Alcatel*, vienen con *hardware* incorporado, por lo que sus precios se incrementan notablemente y, para obtenerlos, se debe contactar con los representantes previamente. De igual manera ocurre con el que oferta *F5*, aunque en este se puede adquirir de forma virtual.

Por otro lado, no existen muchos DPIs orientados a este tipo de redes, por lo que es una buena vía para comenzar a desarrollar este tipo de herramientas en las que presumiblemente serán el futuro de las comunicaciones en red.

Capítulo 4

Análisis de objetivos, requisitos y metodología.

El tema a tratar en este capítulo versa sobre las cuestiones previas que se han abordado como antesala al diseño e implementación del presente proyecto.

En primer lugar, se expondrán los diferentes objetivos que serán fundamentales para dicho diseño e implementación. A continuación, se identificarán todos los requisitos tanto funcionales como no funcionales que se deben cumplir de un modo genérico en el proyecto. Dichos requisitos emergen a raíz de los objetivos previamente marcados.

Se incluye una valoración en cuanto a la consecución de esos objetivos previamente marcados así como la inclusión de nuevos objetivos surgidos a lo largo del diseño e implementación del proyecto.

Por último, se describirá cuál ha sido la metodología empleada para conseguir los resultados finales de manera satisfactoria. Se tratará de describir a grandes rasgos cada una de las fases y sus aspectos más destacados.

4.1. Objetivos.

El objetivo principal de este proyecto es la implementación de un *Deep Packet Inspector* en un entorno de red SDN. Su desarrollo se lleva a cabo en el controlador de red por el que pasarán previamente los paquetes antes de añadir los flujos.

Dentro de nuestro DPI detectaremos diferentes tipos de flujo como son el tráfico *web*, VoIP y tráfico de vídeo procedentes de YouTube. Este último tipo de tráfico es la piedra angular de nuestro detector de paquetes y al que mayor tiempo se le ha dedicado para su desarrollo.

Una vez detectados estos flujos se añadirá cierta calidad de servicio mediante la modificación del campo *DiffServ* del protocolo IP. Para ello modificaremos el *ToS* (*Type of Service*) de los paquetes a un valor que lo diferencie del resto de tráfico de la red.

Como objetivos adicionales cabe destacar dos fundamentalmente. El primero trata de reproducir el DPI tanto en la arquitectura *AD-SAL* (*API-Driven Service Abstraction Layer*), que es la que más desarrollada está dentro de *OpenDayLight*, como *MD-SAL* (*Model-Driven SAL*), la cual se postula como la dominante en los próximos años. El segundo objetivo adicional corresponde a la consecución de un *producto* aislado e independiente de *Deep Packet Inspector* capaz de ser introducido en una red actual sin ningún tipo de incompatibilidad. De esta forma lo que se promueve es la inclusión de SDN y QoS en las redes que imperan a día de hoy sin necesidad de grandes cambios en la misma.

4.2. Especificación de requisitos.

En este apartado se exponen los requisitos tanto funcionales como no funcionales que se han tenido en cuenta para el diseño de este proyecto. Este proceso es importante para conceptualizar los requisitos que son necesarios en la implementación posterior.

Por lo tanto, con esta sección se facilitará la comprensión del diseño, reduciendo el tiempo empleado en la puesta en marcha de todos los elementos implicados, y se obviarán los problemas encontrados durante su desarrollo, los cuales serán expuestos en el anexo D.

4.2.1. Requisitos funcionales.

Pasamos ahora a exponer los requisitos mínimos funcionales necesarios para alcanzar los objetivos expuestos en la sección 4.1 de este documento. Fundamentalmente se trata de requisitos que debe cumplir nuestro *Deep Packet Inspector* para cumplir su trabajo satisfactoriamente. En el siguiente diagrama de flujo vemos cómo se desarrolla el procesamiento de los paquetes de manera global.

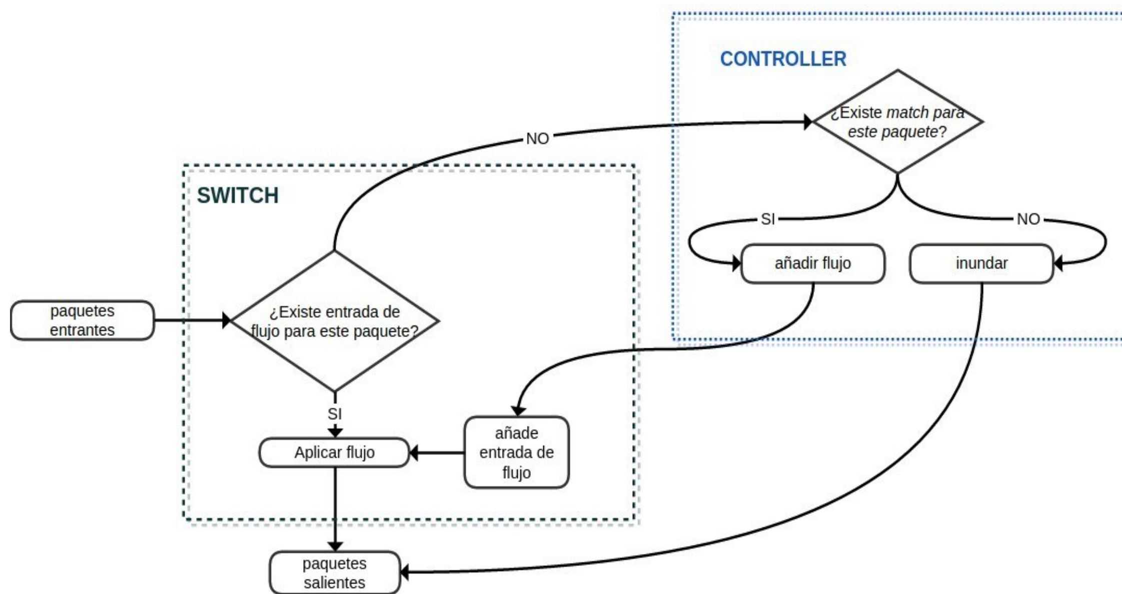


Figura 4.1. Diagrama de flujo del procesamiento de paquetes.

Dentro del controlador es donde se ubica nuestro DPI, el cual será el encargado de analizar el tipo de paquete antes de proceder a asignarle una entrada de flujo o inundar.

Si ahondamos más en nuestro sistema y nos centramos en el controlador podemos hacer un examen más exhaustivo del funcionamiento del DPI. Concretamente lo que nos interesa apreciar es el proceso para detectar y clasificar los paquetes.

Detección de tráfico *web*.

Nuestro DPI detectará el tráfico *web* a partir del puerto que se emplea para este servicio (puerto 80 para *http* y 443 para *https*). No obstante, antes de comprobar el puerto buscaremos los paquetes de tipo IP y, dentro de estos, los que pertenezcan al protocolo TCP para evitar el procesado del puerto de todos los paquetes que nos llegan.

Detección de VoIP.

Para detectar este tipo de tráfico hemos seguido un procedimiento similar al anterior. Nos hemos focalizado en *Skype* y hemos realizado un estudio de los puertos utilizados por esta aplicación de forma predefinida, por lo que su detección se basa en los mismos. Como ya mencionamos en la detección de tráfico *web*, previo análisis del puerto se realizará una clasificación de paquetes para comprobar que son del tipo UDP, protocolo que utiliza esta aplicación, para el análisis del puerto.

Detección de tráfico YouTube.

Este proceso de detección es el más complejo de todos. Para conseguir detectar tráfico YouTube hemos de seguir una serie de pasos con diferentes tipos de paquetes. Su detección se detalla con mayor precisión en el capítulo 7. En términos generales, en primer lugar se detectará el paquete DNS correspondiente al servidor de vídeo de YouTube y, a partir de éste, se conseguirá la dirección IP de éste, por lo que esperaremos hasta detectar esa IP para añadir el flujo de tráfico YouTube.

4.2.2. Requisitos no funcionales.

Dentro de los requisitos no funcionales vamos a repasar las herramientas de las que se ha hecho uso, tanto el entorno de trabajo como el *software* que ha servido de apoyo.

Entornos de trabajo.

Para llevar a cabo este proyecto se ha trabajado con el sistema operativo Ubuntu 14.04. Se escogió este sistema porque era el que mejor se adapta a las herramientas que son necesarias utilizar a lo largo del proceso de desarrollo. Como principal ventaja sobre Windows destaca la posibilidad de crear el entorno ejecutando la topología desde el terminal, sin necesidad de programas adicionales para conectar con el entorno de *Mininet* mediante *Putty*. Además, para la realización de algunas pruebas adicionales se utilizó la máquina virtual *VirtualBox*.

Software y compatibilidad.

Como herramienta de emulación de redes SDN hemos utilizado *Mininet*. Esta herramienta nos permite crear redes virtuales realistas con gran facilidad y su página *web* [12] nos ofrece

una gran cantidad de información para su puesta en marcha. La programación de topologías y su ejecución se realiza utilizando el lenguaje de programación *Python*.

A las topologías que creamos con *Mininet* les hemos de añadir un controlador remoto que, en nuestro caso será el de *OpenDayLight*, por lo que se hace necesario también dicho *software*. Los motivos por los que se utiliza este controlador se explican en el anexo F en el que se lleva a cabo una comparativa entre los controladores más requeridos para estos términos.

Ambas herramientas de *software* serán ampliamente descritas en el capítulo 6 de la presente memoria.

4.3. Valoración del alcance de los objetivos.

En este punto vamos a proceder a valorar el alcance de los objetivos propuestos una vez se ha completado la realización del presente proyecto.

Finalizado éste, podemos afirmar que se han alcanzado todos los objetivos que se plantearon en el apartado 4.1 por completo. Hemos conseguido implementar un DPI en la estructura de una red SDN de tal forma que detecte y clasifique los flujos de nuestro interés añadiendo un *ToS* diferente dependiendo del tipo de flujo. Esta modificación permitirá realizar un tratamiento diferenciado a dichos paquetes por parte de la red para ofrecer calidad de servicio a los flujos detectados. Dicho tratamiento diferenciado queda fuera del ámbito de este proyecto.

Como nota adicional a los objetivos previamente marcados se ha conseguido encuadrar nuestro DPI en dos arquitecturas diferentes como son AD-SAL y MD-SAL, siendo el presente y futuro de las arquitecturas a día de hoy.

Por otra parte, hemos logrado crear un producto independiente, flexible y adaptable que incluye el controlador de la red SDN con su DPI, de manera similar a una caja negra que se puede utilizar en cualquier parte de una red convencional. De esta forma podemos implementar nuestro sistema en una red actual de forma que interactúe con ella sin ningún tipo de incompatibilidad.

4.4. Metodología.

A modo de resumen de este trabajo del que ha surgido el presente proyecto vamos a desgranar cómo ha sido el desarrollo del mismo. Para ello, vamos a describir cronológicamente los pasos que se han dado para su elaboración, comenzando por las fases previas en las que ha sido necesario recabar información acerca de los diferentes aspectos que aquí se exponen.

En primer lugar, se realizó un estudio bibliográfico para obtener los conocimientos básicos y esenciales acerca de las redes SDN. En ella recopilamos herramientas y artículos claves posteriormente para la fase de diseño, profundizando en el conocimiento del protocolo utilizado para este tipo de redes: OpenFlow.

Una vez claros los conceptos básicos nos lanzamos a trabajar con las herramientas básicas e indispensables del proyecto como son *Mininet* y el controlador *OpenDayLight*. Comenzamos creando y analizando topologías básicas y se continuó con escenarios más ambiciosos, no sin encontrar problemas por el camino que dificultaron el avance y que comentaremos en el anexo D.

El siguiente paso fue comenzar a programar en el controlador nuestro DPI. Este proceso se lleva a cabo con el lenguaje de programación Java, por lo que previamente se instaló la herramienta *Eclipse* para facilitar la tarea. En el transcurso de esta fase se fueron entendiendo con más claridad algunos conceptos abstractos sobre el papel, pero que se acabaron comprendiendo en el desarrollo práctico. Éste fue el proceso más largo y complicado de todos, ya que el desarrollo de este campo no es aún muy grande y hubo que investigar y hacer un gran número de pruebas, además de tener que enfrentarse a los problemas de implementación (*bugs*) de las herramientas utilizadas.

Con el DPI implementado, se pasó a comprobar el correcto funcionamiento del sistema tanto en una topología virtual como interconectando con equipos reales para verificar su capacidad de exportar el controlador a una red actual de manera flexible.

Como conclusión final, se ha evaluado el DPI obteniendo resultados satisfactorios con respecto a los objetivos marcados *a priori*. Además, en el capítulo 9 se presentan vías futuras de desarrollo a partir de este proyecto que podrían ser de interés.

Capítulo 5

Planificación y estimación de costes

Este capítulo describe los aspectos relacionados con la planificación del diseño y la estimación de costes prevista.

En primer lugar nos centraremos en los diferentes paquetes de trabajo que han ido dando forma a este proyecto desde que se comenzó a elaborar. También se añade una estimación del tiempo empleado en cada una de estas etapas, que culminará con la elaboración de un diagrama de Gantt que sirva como representación gráfica del desarrollo.

Una vez clara la planificación, pasaremos a exponer los recursos involucrados en este trabajo. Dentro de éstos haremos una distinción entre recursos humanos, *software* y *hardware*.

Por último, se mostrará una estimación de costes necesarios para abordar el presente proyecto en el ámbito económico.

5.1. Planificación.

En este apartado se realizará la descripción de cada uno de los paquetes de trabajo en los que se divide este proyecto. La idea es acercar lo máximo posible a la realidad el proceso de desarrollo de las diferentes partes de manera detallada y precisa.

Los paquetes de trabajo en los que se divide este proyecto son los siguientes:

PT1: Búsqueda de información

Este primer paquete consiste en recabar el máximo de información posible que facilite tanto la comprensión como la posterior puesta en marcha del presente proyecto. Por lo tanto será importante aclarar todos los aspectos ligados a las tecnologías implicadas en nuestro trabajo para conocer lo que ya existe y los avances que se pueden realizar en este campo.

PT2: Familiarización con *Mininet*.

El siguiente paso consiste en la toma de contacto con la herramienta de emulación de red *Mininet*. Esta herramienta es fundamental para el funcionamiento de todas las partes, por lo que es importante descubrir todas y cada una de las posibilidades que nos ofrece. Un aspecto a destacar es que *Mininet* hace uso de Python, por lo que al mismo tiempo será necesario acercarnos a este lenguaje de programación.

PT3: Toma de contacto con *OpenDayLight*.

Con este paquete pasamos a implementar el controlador remoto en nuestra herramienta *Mininet* y a realizar pruebas de mayor complejidad. Para la comprensión de este nuevo elemento añadido a nuestra red es necesario adquirir conocimientos de *Java*, lenguaje con el que ya estamos familiarizados, *Osgi* (*Open Services Gateway Initiatives*), que facilitará la puesta en marcha de los distintos paquetes en el controlador, y *Maven*, que es una herramienta de *software* para la gestión y construcción de proyectos Java necesaria para compilar nuestros paquetes.

PT4: Estudio de aplicaciones para instalación de flujos.

Antes de comenzar a desarrollar nuestro DPI es recomendable investigar acerca de la instalación de flujos en el controlador de *OpenDayLight*. En esta etapa se estudia la forma de conseguir instalar flujos de manera reactiva en el controlador mediante la herramienta *Postman*, que envía la información de los flujos a instalar vía *web* mediante código XML, por lo que también será necesaria una pequeña introducción a este lenguaje. Una vez conseguido instalar flujos proactivos comenzamos con la instalación de flujos reactivos, los cuales son la base de lo que más tarde se convertiría en nuestro DPI. Para ello creamos nuestro paquete *Java* y comenzamos a programar. De esta forma se consigue poner en funcionamiento nuestro controlador y familiarizarse con la creación de nuevos flujos en el controlador.

PT5: Desarrollo de un DPI con detección de tráfico sencillo.

En este paquete se consigue la implementación más básica de un DPI, el cual consigue detectar diferentes tipos de tráfico de manera sencilla. Para estas primeras pruebas se clasificaron los paquetes contenidos en el protocolo IP: TCP, UDP, DNS, etcétera.

Una vez comprobado su funcionamiento se pasa a complicar más el escenario, filtrando por puertos los paquetes y añadiendo acciones sencillas como descarte de paquetes o reenvío por un puerto determinado. De esta forma se van añadiendo etiquetas de *match* y acciones a los flujos.

Este paquete de trabajo será fundamental para adquirir los conocimientos básicos que luego serán de gran utilidad a la hora de implementar flujos más complejos.

PT6: Estudio del tráfico YouTube e implementación en el DPI.

Esta etapa es la más costosa y trabajada de todas las realizadas a lo largo de todo el proceso. Previamente a desarrollar la detección de tráfico de YouTube en el DPI habrá que estudiar con detenimiento los paquetes de este servicio mediante la herramienta *Wireshark*. Es ahí donde se trata de encontrar la clave que permitiese descubrir los paquetes en cuestión sin ningún tipo de ambigüedad y permitiendo flexibilizar el proceso de tal forma que pudiera ser útil para otros tipos de tráfico. Una vez claros los conceptos clave, se opta por programar en el DPI el proceso de detección de este tipo de flujo y sus consiguientes acciones a realizar

una vez detectado este tráfico. En esta segunda parte se comienza a investigar al mismo tiempo la manera de etiquetar el tráfico de forma que se posibilite la implantación de QoS mediante la modificación del *ToS*.

PT7: Fase de pruebas

Una vez implementadas con éxito todas las partes del proyecto se establece un periodo de pruebas en el que se examina el sistema en diferentes escenarios. En primer lugar se prueba su correcto funcionamiento mediante topologías creadas por *Mininet*. Más tarde se testea el funcionamiento conectando diferentes máquinas virtuales como si de equipos externos a la red se tratasen. Por último, se conectan equipos externos al equipo principal con el controlador y se verifica su correcto comportamiento en la red. De esta forma se consiguen detectar posibles fallos no contemplados en el periodo de implementación además de recolectar una serie de datos acerca del rendimiento del sistema y su comportamiento.

PT8: Elaboración de la memoria técnica del proyecto

Con la elaboración de la memoria se pretende documentar todo lo relativo al trabajo realizado, recogiendo tanto los aspectos teóricos como las implementaciones prácticas realizadas de manera detallada. También se pretende recoger todos los resultados obtenidos en el análisis del sistema una vez puesto en marcha. Su elaboración se realiza paralelamente al desarrollo del proyecto, pese a que finalmente se recoja todo en un documento final.

Identificados y descritos todos los paquetes de trabajo que componen nuestro proyecto es necesario realizar una planificación. La planificación que se expone a continuación tiene un carácter meramente orientativo, ya que son varios los factores que influyen a la hora de desarrollar el proyecto, pero la intención es acercarlo lo máximo posible a la realidad de los tiempos marcados en cada plazo. En la siguiente tabla se muestra todo el proceso desglosado en un diagrama de Gantt.

Paquetes de trabajo	Fecha de inicio	Fecha de fin
Búsqueda de información	15/09/2014	30/09/2014
Familiarización con Mininet	01/10/2014	15/11/2014
Toma de contacto con OpenDayLight	01/11/2014	15/12/2014
Estudio de aplicaciones para instalación de flujos	01/12/2014	01/02/2015
Desarrollo de un DPI con detección de flujos sencillos	01/02/2015	01/03/2015
Estudio de tráfico YouTube e implementación en el DPI	01/03/2015	01/06/2015
Fase de pruebas	20/05/2015	20/06/2015
Elaboración de la memoria técnica del proyecto	01/10/2014	30/06/2015

Figura 5.1. Diagrama de Gantt estimado del proyecto.

En cuanto a horas de trabajo se refiere, en la siguiente tabla hemos desglosado cada uno de los paquetes de trabajo con una aproximación del tiempo a emplear en cada uno de ellos.

Paquete de trabajo	Descripción	Tiempo estimado
PT1	Búsqueda de información	50 horas
PT2	Familiarización con *Mininet*	60 horas
PT3	Toma de contacto con OpenDayLight	40 horas
PT4	Estudio de aplicaciones para instalación de flujos	60 horas
PT5	Desarrollo de un DPI con detección de flujos sencillos	20 horas
PT6	Estudio de tráfico YouTube e implementación en el DPI	100 horas
PT7	Fase de pruebas	20 horas
PT8	Elaboración de la memoria técnica del proyecto	100 horas
TOTAL		**450 horas**

Tabla 5.1. Distribución temporal del proyecto.

De esta manera queda conformada la planificación de nuestro proyecto, desglosada tanto en las acciones a realizar con en el tiempo estimado que deberá emplearse en cada una de las diferentes etapas.

5.2. Recursos utilizados.

A continuación se van a tratar de identificar todos los recursos involucrados en el proyecto. Se realizará una clasificación diferenciando entre los recursos de tipo humano, *hardware* y *software*.

5.2.1. Recursos humanos.

· D. Jorge Navarro Ortiz, Profesor Contratado Doctor de la Universidad de Granada en el Departamento de Teoría de la Señal, Telemática y Comunicaciones, en calidad de tutor del proyecto.

· Manuel Sánchez López, alumno del Grado de Ingeniería de Tecnologías de Telecomunicación y autor del presente proyecto.

5.2.2. Recursos *hardware*.

· Ordenador portátil HP Pavilion dv7-3160es, con procesador Intel Core i5-430M a 2.26 GHz, memoria RAM de 4 GB y disco duro de 500 GB de capacidad. Este ordenador se utiliza tanto para programar el controlador como para la ejecución de topologías virtuales y su correspondiente análisis práctico.

· Línea de acceso a Internet. Necesaria para la realización de pruebas con tráfico real. Imprescindible para verificar la detección de tráfico YouTube.

5.2.3. Recursos *software*.

· Sistema operativo *Linux Ubuntu 14.04* (64 bits), utilizado en el ordenador portátil, sobre el que se programará el controlador y se trabajará con las herramientas *Mininet* y *OpenDayLight* para el análisis de topologías de red SDN.

· *Eclipse Luna* (IDE para desarrolladores de Java), entorno de programación para el desarrollo del controlador.

· *Maven*, herramienta *software* para la compilación y manejo de paquetes del proyecto Java.

· *Mininet*, *software* esencial para la emulación de redes SDN.

· *OpenDayLight*, controlador seleccionado para ser la piedra angular de las topologías y encargado de supervisar y redireccionar los paquetes de la red. Dentro del controlador se incluirá el *Deep Packet Inspector*.

· *Google Drive*, para la elaboración de la memoria del presente proyecto. Facilita el seguimiento del tutor del proyecto permitiendo añadir comentarios y sugerencias a los documentos.

· *Wireshark*, para capturar los paquetes procesados por nuestro controlador en las topologías de red creadas.

· Lenguaje de programación *Python*, para la programación de las distintas topologías de red de *Mininet*.

· Navegador web *Firefox* para generar tráfico de YouTube fuera de la red interna.

5.3. Estimación de costes.

En este punto trataremos de plasmar la estimación de costes prevista al abordar este proyecto. Es notable que los costes no son elevados ya que, como puede comprobarse en el punto anterior, el *software* es prácticamente en su totalidad gratuito.

Recursos humanos

Los costes relacionados con los recursos humanos se van a calcular tomando como base el tiempo invertido en cada uno de los paquetes de trabajo. Es por ello que tomaremos como referencia la *Tabla 5.2* en la que se detalla de manera aproximada el tiempo empleado en cada una de las diferentes etapas. Adicionalmente estableceremos las siguientes premisas:

· El sueldo medio de un graduado en Ingeniería en Tecnologías de Telecomunicación es de 20 euros / hora.

· El sueldo de un Profesor Contratado Doctor de la Universidad de Granada se estima en torno a 50 euros / hora. Se considera que en total ha podido invertir entre tutorías para orientar al alumno y la posterior revisión del trabajo unas 15 horas.

De este modo se estipula el presupuesto que presentamos en la siguiente tabla:

Paquete de trabajo	Descripción	Coste estimado
PT1	Búsqueda de información	1000 euros
PT2	Familiarización con *Mininet*	1200 euros
PT3	Toma de contacto con OpenDayLight	800 euros
PT4	Estudio de aplicaciones para instalación de flujos	1200 euros
PT5	Desarrollo de un DPI con detección de flujos sencillos	400 euros
PT6	Estudio de tráfico YouTube e implementación en DPI	2000 euros
PT7	Fase de pruebas	400 euros
PT8	Elaboración de la memoria técnica del proyecto	2000 euros
Tutorización por Profesor Contratado Doctor		750 euros
TOTAL		**9750 euros**

Tabla 5.2. Coste estimado de recursos humanos.

Para una comparativa más visual vamos a representar el coste asociado a cada paquete en el siguiente gráfico de barras:

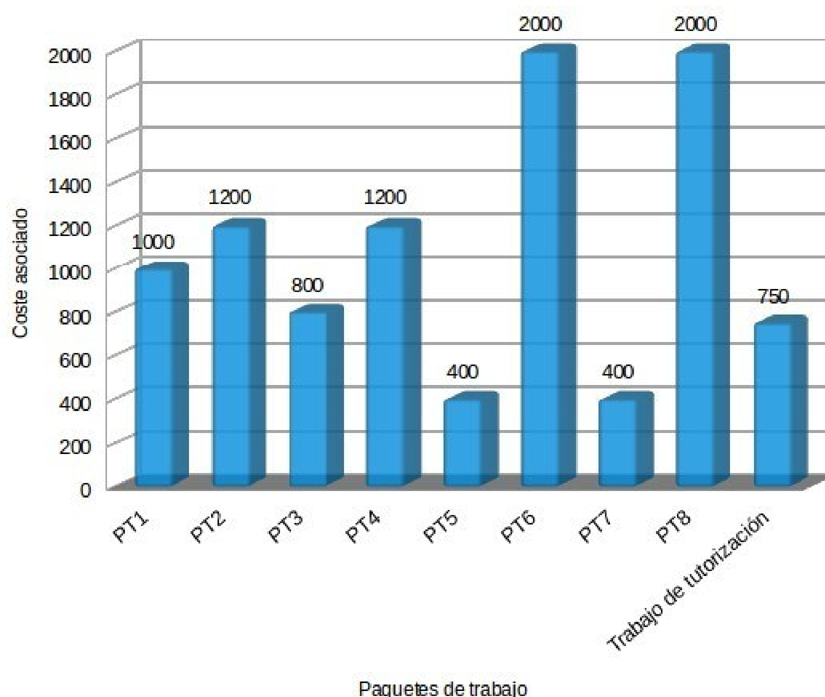


Figura 5.2. Gráfico de barras del coste de recursos humanos para cada paquete de trabajo.

Recursos *hardware*

Los costes asociados a los recursos *hardware* están asociados al material previamente mencionado en el apartado 5.2.2. A continuación se exponen con detalle:

Recurso	Coste estimado	Vida media
Ordenador portátil	1000 €	36 meses
Línea acceso a Internet	30 euros / mes	-

Tabla 5.3. Costes estimados de recursos hardware.

Recursos *software*

Como ya se comentó con anterioridad, los recursos *software* empleados para la realización de este proyecto son gratuitos, por lo que no supondrán ningún tipo de coste adicional a nuestro proyecto. Por lo tanto, esto supondrá un importante abaratamiento en los costes finales.

5.4. Presupuesto final.

Como resumen de los costes surgidos de los diferentes recursos utilizados vamos a exponer este apartado, con el objetivo de presentar una estimación final del presupuesto necesario para abordar este proyecto en su totalidad. En la *Tabla 5.4* se recogen los costes asociados a cada recurso considerando una amortización de 10 meses para los recursos *hardware*; periodo empleado para elaborar el proyecto.

Concepto	Coste
Recursos humanos	**9750 euros**
Ordenador portátil	**277,78 euros** *(1000 euros x 10 meses / 36 meses)*
Línea de acceso a Internet	**300 euros** *(30 euros / mes x 10 meses)*
TOTAL	**10.327,78 euros**

Tabla 5.4. Presupuesto final estimado.

Si echamos un vistazo al presupuesto final estimado, podemos comprobar que la mayor parte está ligada a los recursos humanos con un 94,4% de los costes totales. Por lo tanto, podemos concluir que este proyecto no requiere de un gran esfuerzo económico en cuanto a recursos *hardware* y *software*, sino que centra su importancia en la capacidad de trabajo de los ingenieros contratados.

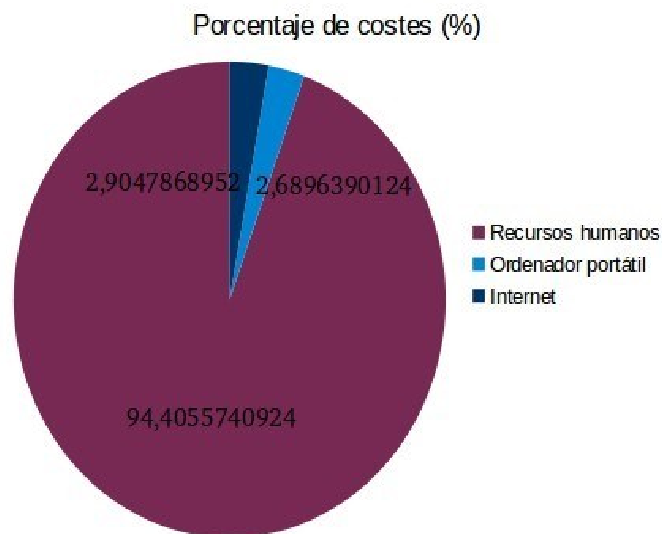


Figura 5.3. Porcentaje de costes final.

5.5. Consideraciones finales sobre la planificación.

Una vez finalizado el proyecto podemos concluir que se ha cumplido prácticamente acorde con los plazos marcados desde un principio para la consecución de los objetivos de este trabajo. No obstante, es cierto que el desarrollo del paquete de trabajo referente a la implementación en el DPI del tráfico YouTube no se desarrolló de forma continuada durante el periodo marcado debido a complicaciones surgidas, ya que, una vez realizadas una serie de pruebas, se descubrieron algunos errores. Pese a ello, no hubo problema alguno para solventar estos fallos y el proyecto se finalizó según lo previsto.

Capítulo 6

Herramientas utilizadas

Este capítulo versa sobre las herramientas utilizadas en la elaboración del *Deep Packet Inspector*. En él, definiremos con detalle dichas herramientas y se expondrán las funcionalidades que han ayudado a la consecución del proyecto.

En primer lugar introduciremos *Mininet*, una herramienta de emulación de redes que nos facilitará la emulación de los escenarios de redes definidas por *software*. Posteriormente describiremos el controlador encargado de gestionar la red emulada por *Mininet*: *OpenDayLight*.

6.1. *Mininet*.

Mininet es un emulador de red que ejecuta una colección de dispositivos finales, *switches*, *routers* y enlaces en un solo *core* de Linux. En los siguientes apartados vamos a ir desgranando sus características principales e incidiremos en las posibilidades que ofrece para la creación de un sin fin de topologías.

6.1.1. ¿Por qué utilizar *Mininet*?

Las principales ventajas de *Mininet* son su rapidez para poner en marcha toda una red, su capacidad para crear topologías personalizadas y la posibilidad de interactuar con otros programas como, en nuestro caso, utilizando un controlador remoto que se adapte a la perfección a nuestra topología.

Mininet es, además, una buena forma de desarrollar y experimentar con el protocolo OpenFlow y los sistemas SDN, por lo que se amolda perfectamente a los requerimientos de nuestro trabajo.

Otra de sus principales ventajas es que está activamente en desarrollo y mantienen un gran soporte sobre esta herramienta, además de que posee una licencia de código abierto.

Nosotros mismos, como desarrolladores, tenemos la posibilidad de contribuir a la comunidad aportando nuevo código, reportando *bugs*, documentación o cualquier mejora del sistema.

Su página web [12] ofrece una gran cantidad de información, desde como instalar mediante cuatro opciones diferentes hasta un amplio tutorial para familiarizarnos con el entorno *Mininet* de manera sencilla.

Entre sus principales características destacan:

- Flexibilidad: se pueden añadir topologías y características nuevas por *software* usando lenguajes de programación y sistemas operativos comunes.
- Aplicabilidad: se pueden introducir las implementaciones en redes basadas en *hardware* sin cambiar su código.
- Interactividad: la administración y la simulación tienen lugar en tiempo real.
- Escalabilidad: es escalable a redes grandes con un sólo computador.
- Realista: el comportamiento del prototipo se asemeja al de la realidad con un alto grado de confianza. Esto permite que aplicaciones y pilas de protocolos se puedan utilizar sin cambiar código.
- Compatible: todo el material puede ser compartido con otros colaboradores y desarrolladores para que puedan ejecutar y modificar los prototipos.

Comparativa con *EstiNet*

EstiNet [34] es otra herramienta que permite tanto simulación como emulación de redes SDN.

La simulación *software* no tiene costes, es flexible, controlable y escalable en contraposición a las operaciones realizadas por los sistemas reales y aplicaciones. No obstante, si el modelo utilizado en el simulador no es lo suficientemente correcto, los resultados pueden variar en gran medida de los obtenidos al implementarlo en un entorno real. Para abordar estos problemas está la emulación, haciendo uso de dispositivos reales que recrean las aplicaciones que interactúan con los dispositivos simulados. Una diferencia entre ambos métodos es el reloj; mientras que en la simulación podemos hacer que acelere o disminuya su velocidad, en la emulación el reloj es invariable. Además, este reloj dependerá del *kernel* de la CPU, por lo que, si éste se encuentra ejecutando otras acciones, repercutirá en el resultado obtenido.

Una propiedad de *EstiNet* es que utiliza la metodología "*kernel re-entering*" la cual posibilita ejecutar programas reales sin modificaciones en *hosts* simulados. Esto permite que los resultados de la simulación se acerquen con bastante precisión a los emulados, permitiendo además una ejecución más rápida. Controladores OpenFlow como *NOX/POX*, *Ryu* y *FloodLight* se pueden simular sin ninguna modificación. Por otra parte, el emulador de *EstiNet* permite de igual forma poner en marcha los programas controladores y emularlos en *switches* OpenFlow en la misma máquina.

Por su parte, *Mininet* sólo permite emulación y hace uso de *hosts* virtuales, *switches* y enlaces para crear una red en un único *kernel* del sistema operativo, y utiliza la red real para el procesado de paquetes y conexiones. Además, las aplicaciones basadas en Unix/Linux pueden ejecutarse en los *hosts* virtuales creados. En una red OpenFlow emulada con *Mininet*,

pueden ejecutarse aplicaciones referentes a los controladores OpenFlow en una máquina externa o en la misma máquina virtual donde los *hosts* virtuales se emulan.

Según los estudios realizados [35], *EstiNet*, pese a ser preciso y escalable, necesita mucho tiempo para simular muchos *switches* OpenFlow. *Mininet* se comporta bastante bien en general, aunque en algunos tests se generan resultados algo inconsistentes.

Una ventaja de *Mininet* es que permite aislar los *hosts* virtuales creados de manera que separa sus interfaces de red, las tablas de encaminamiento y las tablas ARP. Por otra parte, *Mininet* proporciona una página *web* con gran cantidad de información para su instalación y puesta en marcha que resulta más clara que la disponible en la página de *EstiNet*.

Sin embargo, la razón principal por la que nos hemos decantado por *Mininet* es su código abierto y que se apoya en una comunidad abierta también para su desarrollo, mientras que en *EtsiNet* dependemos del soporte privado.

6.1.2. Programación de topologías de red con *Mininet*.

Interacción con *Hosts* y *Switches*

Pasamos ahora a profundizar en la creación de topologías utilizando *Mininet*. La topología básica de *Mininet* está formada por dos *hosts*, un *switch* y un controlador básico y se crea ejecutando en el terminal:

```
$sudo mn
```

Al ejecutar este comando la topología se creará y podremos comenzar a realizar pruebas.

```
manu@manu-HP-Pavilion-dv7-Notebook-PC:~$ 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 6.1. Creación de topología básica en Mininet.

Una vez con la topología arrancada, podemos ver la ayuda que facilita *Mininet* ejecutando:

```
mininet> help
```

Lo que se nos mostrar en la terminal será lo siguiente:

```
mininet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes        pingpair      py      switch
dpctl  help   link      noecho       pingpairfull  quit    time
dump   intfs  links     pingall      ports         sh      x
exit   iperf  net       pingallfull  px            source  xterm

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 6.2. Ayuda básica de Mininet.

Para ver los nodos que forman parte de nuestra red basta con ejecutar:

```
mininet> nodes
```

En este caso mostrará, como ya describimos anteriormente, un par de *hosts*, un *switch* y el controlador:

```
mininet> nodes
available nodes are:
c0 h1 h2 s1
```

Figura 6.3. Muestra de los nodos de la topología con Mininet.

Otra posibilidad es la de ver cómo se interconectan los nodos anteriores:

```
mininet> net
```

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
```

Figura 6.4. Muestra de los enlaces entre dispositivos de la red en Mininet.

También tenemos la oportunidad de recabar más datos de nuestra red ejecutando:

```
mininet> dump
```

Lo que se muestra en esta ocasión serán las direcciones IP de sus interfaces:

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=3726>
<Host h2: h2-eth0:10.0.0.2 pid=3728>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=3733>
<Controller c0: 127.0.0.1:6633 pid=3718>
```

Figura 6.5. Muestra de los dispositivos y su información con Mininet.

Además, si queremos ejecutar algún comando concreto de Linux en cualquiera de los equipos de los que está compuesta la red *Mininet* únicamente deberíamos ejecutar "<nombre_equipo> <comando>". Por ejemplo:

```
mininet> h1 ifconfig
```

Este comando nos mostrará las interfaces de red activas en el *host* 1.

Si lo que queremos es abrir un terminal independiente para un equipo concreto podremos conseguirlo mediante la siguiente línea:

```
mininet> xterm <equipo>
```

Test de conectividad entre *hosts*

Por otra parte, podemos comprobar la conectividad entre equipos mediante la ejecución del comando *ping*. Para comprobar la conectividad de la totalidad de la red se ejecuta directamente:

```
mininet> pingall
```

Con este comando se enviará un paquete ICMP a todos los equipos interconectados.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Figura 6.6. Test de conectividad de red en Mininet.

Creación básica

Se disponen de tres vías para crear topologías en *Mininet*: la básica, utilizando el comando "*mn*" acompañado de ciertos campos, utilizando la interfaz gráfica *miniedit* o mediante la elaboración de un *script* en python. En primer lugar vamos a centrarnos en la creación básica de topologías.

Crear una topología desde la línea de comandos se antoja una tarea bastante sencilla. Únicamente habrá que tener claro que opciones nos permite seleccionar *Mininet* y con qué valores pueden seleccionarse.

Para tener una idea más clara de todas las posibilidades que ofrece *Mininet* presentamos la *Tabla 6.1*. En ella se indican la mayoría de los campos que pueden acompañar a "*mn*" al ejecutar una topología.

Opción	Parámetros	Descripción
-h, --help	-	muestra la ayuda
--*switch* = *SWITCH*	default|ivs|lxbr|ovs|ovsbr|ovsk|ovsl|user[,param=value ...]	crea un *switch* con los parámetros indicados si se desea
--*host=HOST*	cfs|proc|rt[,param=value...]	crea *host*
--controller = CONTROLLER	default|none|nox|ovsc|ref|remote|ryu[,param=value...] --link=LINK default|tc[,param=value...]	crea el controlador con los parámetros indicados en caso de añadirlos
--link = LINK	linear|minimal|reversed|single|torus|tree[,param=value...]	crea los link con los parámetros indicados

--topo = TOPO	linear|minimal|reversed|single|torus|tree[,param=value...]	crea una topología de las indicadas con los parámetros deseados
-c, --clean	-	elimina todo lo referente a *mininet* y sale del mismo
--custom = CUSTOM	-	lee clases customizadas o parámetros de un .py
--test = TEST	cli|build|pingall|pingpair|iperf|all|iperfudp|none	ejecuta uno de estos tests
-i IPBASE	-	especifica la ip base para los *host*
--mac	-	añade una mac automáticamente a los *hosts*
--arp	-	genera todas las entradas ARP
--listenport = LISTENPORT	-	puerto para la escucha de *switches*
--nat	-	añade NAT a la topología, lo cual conecta *Mininet* con la red física
--cluster = server1, server2...	-	arranca múltiples servidores

Tabla 6.1. Lista de algunos campos ejecutables en Mininet.

Como ejemplo de una topología ejecutada mediante línea de comandos exponemos el siguiente:

```
$ sudo mn --controller=remote --topo single,3 --mac --switch ovsk,protocols=OpenFlow10 --nat
```

Descriptivamente, podemos ver que se crea una topología con un controlador remoto (*OpenDayLight en nuestro caso*), con una topología formada por un *switch* unido a tres *hosts*

(*single, 3*), a los cuales se les asigna una dirección MAC. La dirección del controlador remoto será por defecto la dirección local *127.0.0.1*. En caso de que sea otra se le puede asignar añadiendo *“ip=<dir_ip_controlador>”*. Además, se define el tipo de *switch* a utilizar (ovsk) y el protocolo que va a soportar, en este caso OpenFlow en su versión 1.0. Por último, se especifica que se desea realizar NAT, lo que posibilitará la conexión con la red externa.

Miniedit

Miniedit nos permite crear topologías mediante un interfaz gráfico que facilita esta tarea para los usuarios poco familiarizados con la línea de comandos de Linux. Para ejecutar este *software* tendremos que dirigirnos al directorio de *Mininet* donde se encuentra, el cual es “*mininet/examples*”. Una vez en el directorio basta con ejecutar:

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

Al ejecutar el comando anterior nos aparecerá la siguiente interfaz:

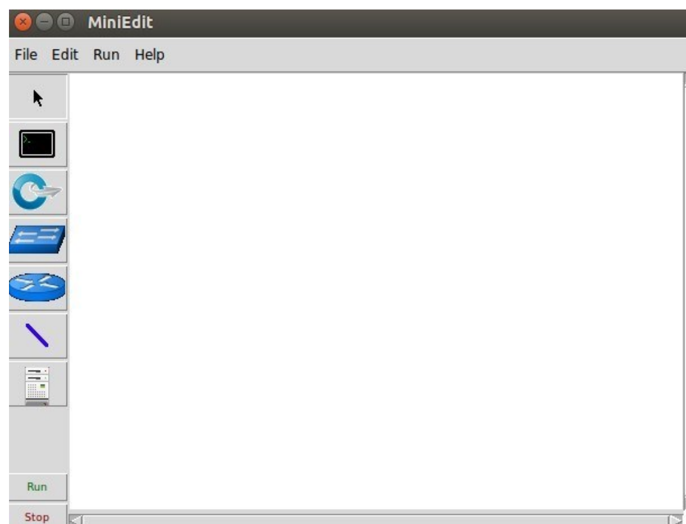


Figura 6.7. Interfaz gráfica de Miniedit.

Como vemos, los elementos que podemos incluir están representados en la barra lateral de la izquierda. En ella se incluyen *hosts*, *switches* (tanto con soporte de *OpenFlow* como sin él), *routers*, enlaces y el controlador.

Una vez colocamos nuestros dispositivos en el panel, podemos editar sus propiedades haciendo clic derecho sobre ellos y seleccionando la pestaña de *“Properties”*. En la imagen siguiente se muestra un ejemplo de topología realizado con *Miniedit* formado por tres *switches*, el primero con dos *hosts* y los otros dos con uno, unidos al controlador.

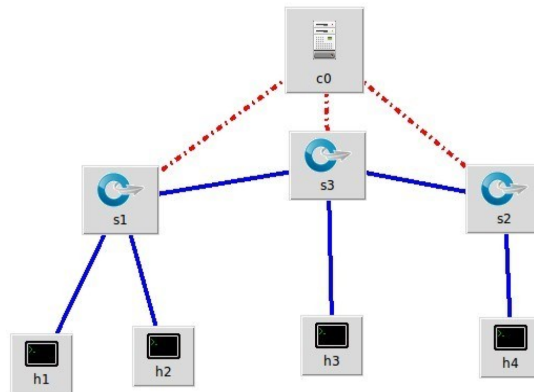


Figura 6.8. Ejemplo de topología creada con miniedit.

Para guardar la topología seleccionamos en el menú la pestaña *File* y seguidamente exportamos la topología como un *script* de nivel 2 para guardarla como un *script* de *python.* Si deseamos poner en marcha nuestra topología presionamos la pestaña *Run*. Para poder tener acceso a la misma desde el terminal deberemos habilitar la opción de "*Start CLI*" seleccionando la pestaña *Edit* → *Preferences*. Si realizamos este procedimiento veremos en nuestro terminal lo siguiente:

```
Build network based on our topology.
Getting Hosts and Switches.
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
Getting controller selection:ref
<class 'mininet.node.Host'>
<class 'mininet.node.Host'>
Getting Links.
*** Configuring hosts
h3 h4 h2 h1
**** Starting 1 controllers
c0
**** Starting 3 switches
s1 s3 s2
No NetFlow targets specified.
No sFlow targets specified.


 NOTE: PLEASE REMEMBER TO EXIT THE CLI BEFORE YOU PRESS THE STOP BUTTON. Not exiting wil
l prevent MiniEdit from quitting and will prevent you from starting the network again du
ring this sessoin.

*** Starting CLI:
mininet>
```

Figura 6.9. Mensaje en la terminal al ejecutar topología de miniedit.

Ahora podemos probar a realizar un ping para comprobar que todo está en orden e incluso ver si el controlador ha instalado flujos en los *switches*. En las siguientes imágenes se reflejan estas dos acciones.

```
mininet> h1 ping h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=7.84 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.638 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.082 ms
```

(a)

```
mininet> sh ovs-ofctl -O OpenFlow10 dump-flows s1
 cookie=0x0, duration=21.76s, table=0, n_packets=3, n_bytes=294, idle_timeout=60, idle_a
ge=19, priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=36:79:92:ce:9a:f8,dl_dst=86:
c8:e5:82:83:7a,nw_src=10.0.0.4,nw_dst=10.0.0.1,nw_tos=0,icmp_type=0,icmp_code=0 actions=
output:1
 cookie=0x0, duration=21.763s, table=0, n_packets=3, n_bytes=294, idle_timeout=60, idle_
age=19, priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=86:c8:e5:82:83:7a,dl_dst=36
:79:92:ce:9a:f8,nw_src=10.0.0.1,nw_dst=10.0.0.4,nw_tos=0,icmp_type=8,icmp_code=0 actions
=output:3
```

(b)

Figura 6.10. (a) Ping entre h1 y h4 en la topología creada en miniedit y (b) flujo añadido.

Por lo tanto, queda demostrado que se pueden crear topologías de manera sencilla y visual haciendo uso de esta herramienta.

Implementación de *scripts* en *Mininet*

Como último método para crear topologías vamos a comentar la implementación de *scripts* en *python*. Este método es quizás el más complejo, por el hecho de que necesitaremos conocer este lenguaje de programación, pero a la vez el que más posibilidades a la hora de desarrollar nos permite.

En primer lugar, hemos de colocar los paquetes que deseamos importar a nuestro *script*. Esto se realiza indicando de qué paquete vamos a importar el elemento indicado.

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
```

Una vez hecho esto podemos pasar a definir funciones, en nuestro caso nuestra topología. Es recomendable realizar un boceto explicativo de la red antes de su implementación para la correcta comprensión por parte del lector.

En esta función podremos definir todos los elementos que formarán parte de nuestra red además de configurar sus propiedades. A modo de ejemplo mostramos la siguiente función para crear una topología formada por el controlador conectado a un *switch* que tiene un *hosts* y puede hacer NAT con el exterior.

```
def myNetwork():
    net = Mininet( topo=None, listenPort=6633,
                   build=False,ipBase='10.0.0.0/8',
                   link=TCLink,)
    info( '*** Adding controller\n' )
    c0=net.addController(name='c0', controller=RemoteController,
                         protocols='OpenFlow13', ip='127.0.0.1')
```

```
info( '*** Add switches\n')
s1 = net.addSwitch('s1', cls=OVSSwitch, mac='00:00:00:00:00:10',
                   protocols='OpenFlow13')
info( '*** Add hosts\n')
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', mac='00:00:00:00:00:01',
                 defaultRoute='via 10.0.0.2') # defaultRoute es la ip del nat
info('*** Add NAT\n')
net.addNAT().configDefault()
net.addLink(s1, h1, bw=10, delay='0.2ms')
info( '*** Starting network\n')
net.build()
net.start()
info( '*** Starting controllers\n')
for controller in net.controllers:
    controller.start()
info( '*** Starting switches\n')
net.get('s1').start([c0])
info( '*** Post configure switches and hosts\n')
CLI(net)
net.stop()
```

Como podemos ver, en primer lugar se toma como referencia el objeto *Mininet* para crear la topología, objeto que se ha importado previamente. Luego vamos dando forma a nuestra red añadiendo el controlador e indicando algunas propiedades del mismo como su nombre, el tipo de controlador (remoto), el protocolo OpenFlow que se va a utilizar en la comunicación con el *switch* o la IP del mismo. Así vamos creando tanto el *switch* como el *host* con sus propiedades particulares. Una vez definidos los elementos, creamos la interfaz que posibilitará a la red hacer NAT y añadimos el *link* entre el *switch* y el *host*. Ya solo queda arrancar el controlador y el *switch* para que, una vez llamada la función, comience a ejecutarse la topología.

6.2. *OpenDayLight*.

Una vez claros los conceptos de *Mininet* pasamos ahora al controlador encargado de manejar las topologías creadas: *OpenDayLight*. Como vimos en la sección anterior, basta con indicar que el controlador que vamos a utilizar en nuestra topología será remoto para poder compatibilizar *OpenDayLight* con *Mininet*. Únicamente tendremos que arrancar el controlador por un lado y la topología por otro y el *switch* se pondrá en contacto con dicho controlador.

No obstante, para desarrollar nuestro propio módulo deberemos hacer uso de las APIs que proporciona *OpenDayLight*:

- ***DOM API***. *DOM (Document Object Model)* [36] es un API para documentos HTML y XML. *DOM* es una representación del documento como un grupo de nodos y objetos estructurados que tienen propiedades y métodos. Proporciona una representación

estructural del documento, permitiendo la modificación de su contenido o su presentación visual. Esencialmente, comunica las páginas *web* con los scripts o los lenguajes de programación.

- ***REST API***. Esta *API* nos permite comunicarnos haciendo uso del protocolo HTTP en cualquier formato (XML, JSON, etcétera). Los datos están definidos en *YANG* (lenguaje de modelado definido para *NETCONF* [37]), y se hace uso de almacenamiento definido por *NETCONF* (*Network Configuration Protocol*). Esta *API* será la que utilicemos para añadir flujos de forma proactiva como veremos más tarde.
- ***Java APIs*** para consumidores y proveedores. Se basa en utilizar *Java* para el desarrollo de los módulos, por lo que será la opción que utilicemos para crear nuestro módulo debido a la familiarización previa con este lenguaje. El *Java API* utilizado será la proporcionada por Cisco [38], ya que implementa una serie de funcionalidades para manejar flujos que nos facilitarán el trabajo.

Antes de meternos de lleno con el desarrollo del controlador vamos a introducirlo y a comentar algunos aspectos de especial relevancia.

6.2.1. ¿Por qué utilizar *OpenDayLight*?

OpenDayLight es un proyecto de Linux apoyado por la industria. Cuenta con un gran apoyo de la comunidad y listas de correo para la discusión acerca de temas relacionados con el mismo. En la lista de correo se puede observar fácilmente el dominio de desarrolladores de *Cisco* en la misma. Por otro lado, *OpenDayLight* sigue un modelo de controlador que además de OpenFlow, puede introducir otros protocolos. Esta faceta diferencia significativamente a *OpenDayLight* de otros controladores y le permite utilizar *switches* que emplean los protocolos de control de propiedad no OpenFlow. En la siguiente imagen se muestra el apoyo recibido por muchas de las empresas importantes en el desarrollo de OpenFlow a este controlador:

Figura 6.11. Lista de empresas que ofrecen apoyo al controlador OpenDayLight.

Además del apoyo de la industria es importante destacar su código abierto y su compatibilidad con la herramienta de emulación que vamos a utilizar en este proyecto: *Mininet*.

Otra característica influyente en la decisión de utilizar este controlador es que el lenguaje de programación en el que se basa es *Java*, con el cual estamos familiarizados.

En el Anexo B se realiza una comparativa entre los principales controladores para este tipo de redes en el cual se justifica su elección.

A continuación veremos algunos aspectos que decantan la elección de *OpenDayLight*.

6.2.2. *Orange* y el uso de *OpenDayLight*.

Como se mencionó anteriormente, grandes empresas han dado su apoyo a este proyecto. Como dato relevante, cabe destacar el vínculo existente entre una gran empresa del sector como es *Orange* y el controlador de *OpenDayLight* [39].

OpenDayLight de la mano de *Orange* ha mantenido un largo historial de participación y contribución a las comunidades *open source* y han liderado el soporte de estándares abiertos. Esta empresa, como una de las más importantes en el sector de los operadores de telecomunicaciones, proporciona un servicio móvil que supera los 247 millones de clientes en 29 países diferentes. Con la incorporación de *OpenDayLight* en el año 2013, el apoyo a la comunidad aumentó considerablemente. Además, *Orange* participa en otros proyectos como *OpenStack* y es uno de los fundadores de la OPNFV (*Open Platform for Network Functions Virtualization*).

Con las redes SDN en *Orange*, se prevén grandes beneficios, incluyendo el cumplimiento de pedidos, garantías, nube interconectada de documentos, funciones de red de virtualización (NFV), conectividad y mucho más. Hay muchas funcionalidades que pueden hacer que SDN llegue a los operadores de red, empezando por la natural dinámica de esta tecnología. Por otro lado, los operadores pueden ejercer un control dinámico sobre los recursos de red, lo que puede repercutir en claros beneficios. No menos importante es la capacidad de potenciar nuevos modelos de compra de los consumidores para el autoservicio y aprovisionamiento bajo demanda. Aunque la automatización es importante, serviría de poco en ambientes excesivamente complejos. Sin embargo, SDN puede gestionar las abstracciones de tal forma que consiga ocultar la complejidad y simplifique la solución global.

Las principales oportunidades que abren las redes SDN a los operadores de telecomunicación son:

- Posibilidad de controlar dinámicamente la demanda de los recursos conectados a la red y automatizar las configuraciones.
- Ocultar la complejidad sobre los recursos de abstracción de red.

- Mejorar el mantenimiento de la conectividad para redes heterogéneas.

Orange ha contribuido al desarrollo de:

- Extender la superposición de la capa 2 usada en *datacenters* a los operadores de acceso a la red.
- Red *Self-Healing* (auto-sanación) basada en el diagnóstico auto-modelado. Se utiliza el controlador de topologías de ODL para generar automáticamente un modelo de la topología de red. El inconveniente es que se necesita mucho tiempo para recuperar la topología.
- Aprovisionamiento de VPNs dinámicas de capas 2 y 3 utilizando elementos de red de dichas capas y la interfaz *Netconf*.
- PCE (*Path Computation Element*) basado en el controlador SDN WAN (*Wide Area Network*). Incluye test BGP (*Border Gateway Protocol*) para el descubrimiento y sincronización de topologías y PCEP (*Path Computation Element Protocol*) para descubrimiento y aplicación de túneles.

Por lo tanto, queda reflejado que *Orange* es un importante apoyo para este proyecto y su involucración en el mismo es patente.

6.2.3. Distribuciones disponibles.

Distribución base

Esta distribución [40] es la más básica para comenzar a trabajar con *OpenDayLight*. En ella se incluye el controlador y los paquetes básicos instalados. Para implementar nuevos paquetes puede consultarse el manual de instalación de *OpenDayLigth* en el Anexo B.

Para poner en marcha el controlador basta con entrar en el directorio *OpenDayLight* y ejecutar:

```
$ ./run.sh
```

De este modo el controlador comenzará su funcionamiento y, una vez detecte una topología, comenzará a gestionarla.

Esta será la distribución que utilicemos para el desarrollo de nuestro DPI debido a que es la que menos complementos instala, los cuales nosotros no utilizaremos en su gran mayoría, y porque su ejecución es la más rápida y directa.

Distribución *Hydrogen*

Esta distribución fue la primera en ser distribuida por parte de *OpenDayLight*. En esta versión se incluyen paquetes que facilitan el funcionamiento del controlador en la red sin necesidad de grandes configuraciones como, por ejemplo, el control de la capa física.

No obstante, ha quedado relegada a un segundo plano tras la incorporación de la distribución *Helium* que veremos a continuación. Es por ello que no se ha hecho uso de esta versión en nuestro proyecto, dado que la versión de la que hablaremos a continuación es más reciente y completa.

Distribución *Helium*

Esta distribución es bastante más avanzada que la primera. En ella se proporciona de una interfaz y una instalación simple y personalizable gracias a la utilización del contenedor *Apache Karaf*.

Los complementos que se adhieren a *Helium* permiten interactuar con el controlador de forma más transparente. Así, *Karaf* nos proporciona una plataforma genérica que proporciona funciones y servicios de alto nivel diseñados específicamente para la creación de servidores basados en *OSGi*. Esta distribución viene con una serie de proyectos incluidos que son fáciles de instalar gracias a *Karaf*.

La plataforma *OpenDayLight* con *Helium* evoluciona en áreas clave como alta disponibilidad, *clustering* y seguridad, así como fortaleciendo y añadiendo nuevos protocolos con relación a *OpenFlow*, multimedia, un marco de políticas de aplicaciones y herramientas para el servicio de función de encaminamiento.

· Karaf

Como hemos comentado, este complemento facilita la comunicación con el controlador. *Apache Karaf* [41] es uno de los contenedores de *OSGi* más populares que permite desplegar aplicaciones de forma rápida y fácil y nos proporciona varias funcionalidades como las siguientes:

- **Rápido despliegue**. *Karaf* monitoriza los archivos "*.jar*" de tal forma que facilita el despliegue de los *bundles*. Al tiempo en que estos archivos se copian en un directorio se instalarán. Podremos actualizarlos o borrarlos y estos cambios se verán reflejados automáticamente.
- **Configuración dinámica.** Los servicios se configuran habitualmente a través de *OSGi*, el cual es respaldado por *Karaf*.
- **Sistema de *Logging*.** Dispone de un *backend* centralizado de *logging* gestionado por *Log4j*.
- **Suministro**. Al suministrar librerías y aplicaciones éstas se descargarán localmente, instalarán e iniciarán.

- **Integración en Sistema Operativo nativo**. *Karaf* se puede integrar en el propio Sistema Operativo como un servicio, por lo que el ciclo de vida irá ligado al Sistema Operativo.
- **Consola de *Shell* extensible.** Dispone de una consola de texto para gestionar servicios e instalar nuevas funcionalidades así como gestionar su estado.
- **Acceso remoto**. Se puede conectar con *Karaf* mediante cualquier cliente SSH.
- ***Framework* de seguridad** basado en JAAS.

Para obtener la versión *Helium* basta con ejecutar:

```
$ wget
http://nexus.opendaylight.org/content/groups/public/org/opendaylight/integration/distribution-karaf/0.2.0-Helium/distribution-karaf-0.2.1-Helium-SR1.zip
```

Una vez hecho esto y descomprimido el archivo descargado, nos adentramos en la carpeta y arrancamos *Karaf*:

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

En este caso lo que veremos será la interfaz gráfica que facilita esta herramienta, con un aspecto similar al de la siguiente figura.

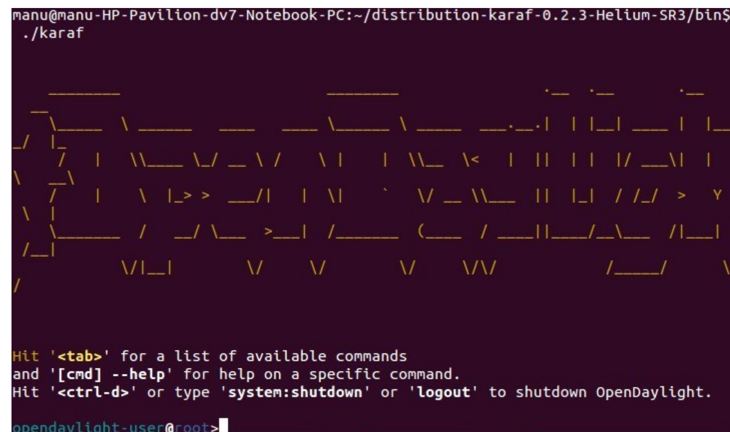


Figura 6.12. Interfaz gráfica de karaf.

Dentro de la consola podemos instalar algunos paquetes básicos [42] para habilitar algunas funcionalidades esenciales como el reenvío de paquetes mediante un *switch* de capa 2,

habilitar la comunicación mediante HTTP, poner a disposición las APIs para el modelo MD-SAL o habilitar la interfaz *web* de *DLUX*. Los paquetes a instalar se indican a continuación:

```
osgi> feature:install odl-restconf odl-l2switch-switch odl-mdsal-apidocs odl-dlux-core
```

A partir de aquí ya podríamos crear una topología en *mininet* y utilizar el controlador remoto previamente arrancado.

Distribución *Lithium*

Es la versión más reciente de *OpenDayLight*, presentada el 29 de junio de 2015. Con esta distribución se pueden componer arquitecturas propias de servicio o adaptar en *OpenDayLight* los servicios de red dinámicos en un entorno de nube, aplicar políticas de calidad de forma dinámica y aplicar virtualización con la tecnología *Service Function Chaining* (SFC), la cual permite utilizar de forma flexible y rápida varias funciones de red.

6.2.4. Comparativa de modelos *AD-SAL* y *MD-SAL*.

OpenDayLight ofrece dos arquitecturas con las que trabajar: MD-SAL (*Model Driven-Service Abstraction Layer*) y AD-SAL (*API-Driven Service Abstraction Layer*). En este punto nos disponemos a realizar una comparativa [43] entre ambas arquitecturas y describir las propiedades que ofrecen ambas.

La *Service Abstraction Layer* (SAL) es la capa encargada de "traducir" las órdenes o servicios requeridos abstrayéndose de los dispositivos que se encuentran debajo de ésta. Es la que permite programar en *OpenDayLight* mediante tres formas diferentes como son *Java API, DOM API* y *REST API*, como expusimos anteriormente.

OpenDayLight hace uso de dos caminos para operar con el controlador. Por una parte se encuentran los *bundles* que se instalan dentro del *framework OSGi* y que se ejecutan en local y, por otra parte, las instrucciones *REST*, que se ejecutan de forma remota utilizando la comunicación mediante *http*.

El hecho de trabajar con *bundles* nos permite la instalación tanto de flujos reactivos (flujos que se instalan al llegar un paquete que cumple unas características prefijadas) como proactivos (flujos instalados con antelación a la llegada de cualquier paquete anticipadamente), no siendo así posible con *REST API*, la cual sólo nos permitirá la instalación de los segundos.

Para trabajar con estos *bundles* se presentan el método antiguo para su programación, *API Driven SAL*, y el método que se está comenzando a utilizar y depurar en estos momentos,

Model Driven SAL. La diferencia entre ambos no radica tanto en lo funcional sino en los conceptos de estructura y modo de aplicación.

En MD-SAL, los proveedores son los encargados de facilitar la información acerca de los *plugins* actuando como módulos del núcleo, mientras que los consumidores son las aplicaciones, las cuales llaman a estos *plugins*. La adaptación del servicio se encarga de llevar a cabo esta comunicación entre ambos. En MD-SAL, las APIs SAL y las peticiones de enrutado entre consumidores y proveedores se definen a partir de modelos, a diferencia de lo que sucede en AD-SAL, y la adaptación de datos la proporcionan los *plugins de adaptación* de forma interna. Por lo tanto, no se realiza una adaptación para cada API como sucedía en AD-SAL, sino que todas ellas se adaptan mediante el *plugin de adaptación*. En este sentido la abstracción en MD-SAL es mayor, ya que con *YANG* se definen todos los modelos. La siguiente imagen muestra la diferencia entre estructuras AD-SAL y MD-SAL:

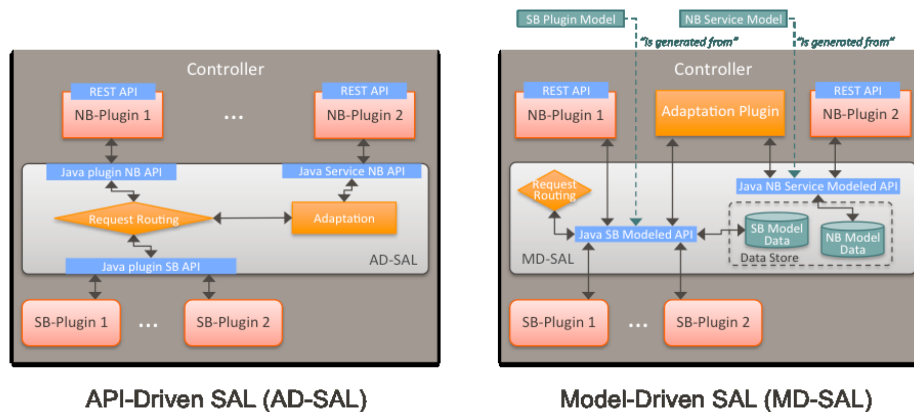


Figura 6.13. Estructuras de AD-SAL y MD-SAL.

La funcionalidad proporcionada por MD-SAL es, básicamente, facilitar la instalación de conexiones entre proveedores y consumidores. Así, un proveedor o un consumidor podrán registrarse con MD-SAL, permitiendo diferentes interacciones entre ellos. Por ejemplo, un consumidor puede encontrar un proveedor; un proveedor puede generar notificaciones; un consumidor puede recibir notificaciones y expedir *RPCs* (*Remote Procedure Calls*) para obtener datos de los proveedores; un proveedor puede insertar datos en el almacén de SAL; un consumidor puede leer datos de ese almacén; etcétera.

En AD-SAL, se proporciona petición de enrutado (selecciona un *plugin* SB basado en un tipo de servicio) y, opcionalmente, se proporciona la adaptación de servicio en caso de que una API NB (abstracción) sea diferente de su correspondiente API SB (protocolo). Por su parte, MD-SAL proporciona también enrutado de peticiones y una infraestructura para soportar la adaptación de servicio a partir de *plugins*, a diferencia de AD-SAL. Además, en cuanto al intercambio de datos, el proveedor y consumidor de *plugins* pueden intercambiarlos con el almacén de MD-SAL, en contraste con AD-SAL, el cual carece de estados.

A continuación presentamos una tabla [44] a modo de resumen en la que se clarifican las diferencias entre ambos modelos:

AD-SAL	MD-SAL
En las APIs de SAL, la solicitud de enrutamiento entre consumidores y proveedores y la adaptación de datos son definidos estáticamente en tiempo de compilación	En las APIs de SAL, las peticiones de enrutado entre consumidores y proveedores se definen mediante modelos, y la adaptación de datos se proporciona mediante los *plugins de adaptación interna*
AD-SAL tiene tanto NB como SB APIs incluso para funciones/servicios mapeados 1:1 entre *plugins* SB y NB	MD-SAL permite ambos *plugins*, tanto SB como NB, utilizados en la misma API generada a partir de un modelo. Un *plugin* pasa a ser un proveedor API; el otro llega a ser un consumidor API
En AD-SAL hay una *REST API* dedicada para cada *plugin southbound/northbound*	MD-SAL provee una *REST API* para acceder a los datos y funciones definidas en modelos
AD-SAL provee solicitud de enrutado (selecciona un *plugin* SB basado en el tipo de servicio) y opcionalmente provee adaptación de servicio, si una NB API (Servicio, abstracto) es diferente de su correspondiente SB (protocolo) API	MD-SAL provee solicitud de enrutado y de infraestructura para dar soporte a la adaptación de servicios, pero no provee adaptación de servicios por sí mismo, sino que se apoya en los *plugins*
La solicitud de enrutado se basa en el tipo de *plugin*: SAL sabe qué instancia de nodo es servida por cada *plugin*, y cuando un *plugin* NB solicita una operación de un nodo, la solicitud es enviada al *plugin* apropiado el cual envía la petición al nodo pertinente	La solicitud de enrutado en MD-SAL se realiza tanto en el tipo de protocolo como en las instancias del nodo, ya que la instancia de datos se exporta desde el *plugin* dentro de SAL
AD-SAL no tiene estados	MD-SAL puede almacenar datos de modelos definidos por *plugins*: proveedores y consumidores pueden

	intercambiar datos sobre el almacén de MD-SAL
Limitado a flujos de capacidad en equipos y modelos de servicio	Modelo agnóstico. Puede soportar algunos equipos y/o modelos de servicio y no está limitado a flujos de capacidad en equipos y modelos de servicio únicamente
Los servicios AD-SAL suelen proveer tanto versiones asíncronas como síncronas del mismo método API	En MD-SAL, el modelo de Servicio de APIs únicamente provee APIs asíncronas, pero devuelven un objeto 'java.concurrent.Future', el cual permite una llamada para bloquear antes de que la llamada sea procesada y un objeto resultado esté disponible. La misma API puede ser utilizada tanto para un enfoque síncrono como asíncrono. Así MD-SAL alienta el enfoque asíncrono de diseño de la aplicación, pero no impide que las aplicaciones sean síncronas.

Tabla 6.2. Comparativa entre AD-SAL y MD-SAL.

Como se puede apreciar, la mayoría de las diferencias radican en el método de aplicación, mientras que AD-SAL se compila a través del código incluido en el mismo, MD-SAL es algo más abstracto y se compila mediante la aplicación de modelos.

El desarrollo de este proyecto se ha centrado más en el modelo AD-SAL, ya que se encuentra totalmente desarrollado y es el modelo que ha presentado menos problemas. No obstante, también se ha trabajado con MD-SAL como veremos más tarde.

6.2.5. Interfaz *web*.

En todas las distribuciones se dispone de una interfaz *web* con la que se pueden monitorizar algunos aspectos de la red creada. Al arrancar el controlador podremos acceder a la interfaz a través de nuestro navegador al entrar en la dirección "*localhost:8080*".

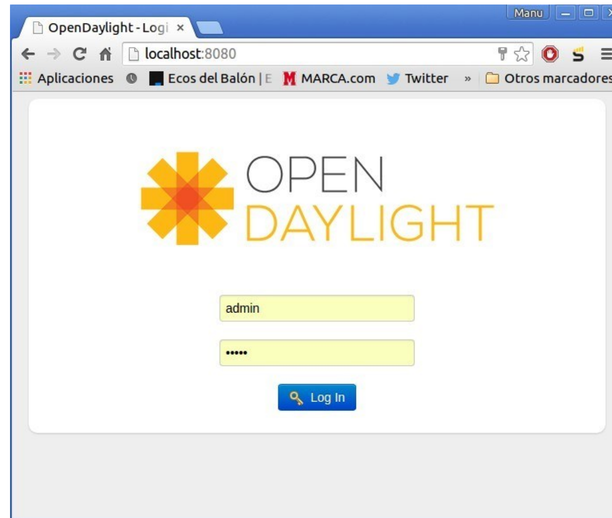


Figura 6.14. Ventana de login de la interfaz web de ODL.

El *login* y el *password* para acceder serán en ambos casos "*admin*". Una vez dentro tendremos acceso a visualizar nuestra topología, ver los diferentes interfaces de red, añadir y borrar rutas estáticas, obtener estadísticas de cada equipo etc. La topología se irá descubriendo según la vaya detectando el controlador. Al hacer un "*pingall*" y recibirse todos los paquetes se generará la topología completa en nuestra gráfica. En la siguiente imagen se muestra una topología de prueba formada por un *switch* y cuatro *host*:

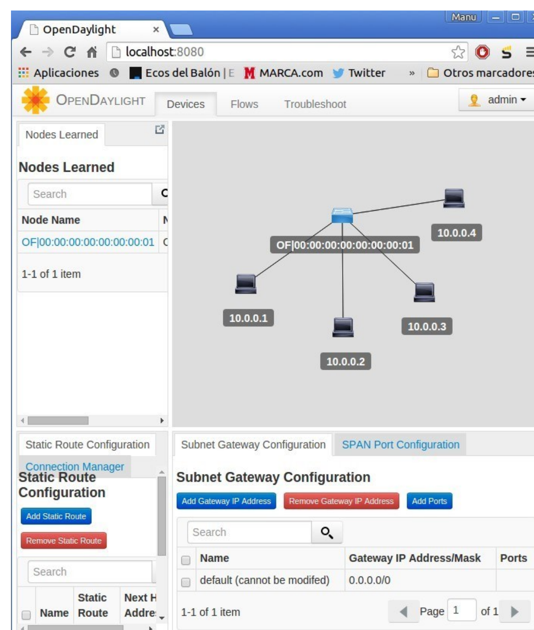


Figura 6.15. Ejemplo de gráfico de topología creado por la interfaz web de ODL.

La interfaz nos permite añadir flujos de forma proactiva si seleccionamos la pestaña "*Flows*". Podemos configurar parámetros de las capas 2, 3 y 4, siendo accesibles campos como la *vlan* o las direcciones MAC en la capa 2, las direcciones IP o el *ToS* en la capa 3 o los puertos origen y destino y el tipo de protocolo en la capa 4. Además, podremos añadir acciones a estos flujos configurados de manera proactiva.

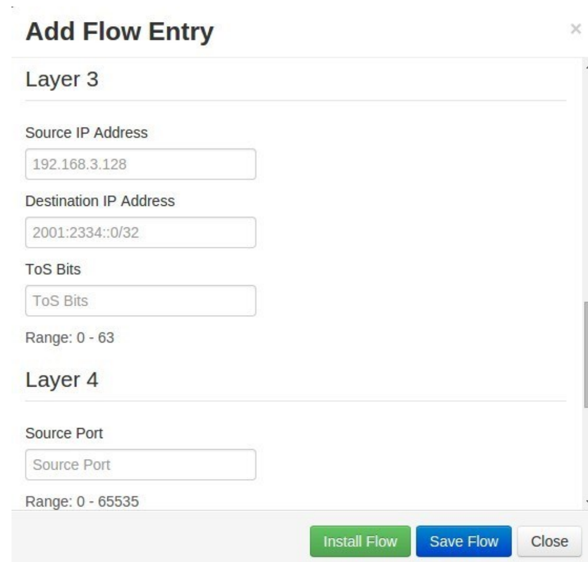


Figura 6.16. Pestaña para añadir flujos de forma proactiva en la interfaz web de ODL.

No obstante, esta forma de trabajar no se recomienda ya que, tras experimentar con ella, hemos detectado algunos problemas en cuanto a la instalación de entradas en la tabla de flujos. Pese a todo, es una buena fuente de información para recabar estadísticas.

6.2.6. Flujo proactivo: REST API.

Pasamos ahora a hablar sobre la instalación de flujos de forma proactiva. Antes de adentrarnos en el concepto de *REST API* debemos introducir *RESTCONF*, ya que la primera es una *northbound API* de la segunda.

RESTCONF

Un *REST* (*REpresentational State Transfer*) [45] es un protocolo que se ejecuta sobre *http* para acceso a datos definido en *YANG* utilizando almacenes de datos definidos en *NETCONF* (*Network Configuration Protocol*). En primer lugar vamos a definir estos conceptos.

YANG [37] es un lenguaje modelado escrito para soportar equipos basados *NETCONF*, el cual es un protocolo que se encarga de instalar, manipular y borrar la configuración de los

equipos de red. En *OpenDayLight* se utiliza *YANG* para describir la estructura de datos proporcionados por los componentes del controlador.

Una vez definidos los conceptos de *YANG* y *NETCONF* vamos a definir *RESTCONF*. *RESTCONF* es un proyecto *IETF* (*Internet Engineering Task Force*) que describe cómo asignar una especificación *YANG* a una interfaz *REST*. Todo el contenido *RESTCONF* identificado ya sea como un recurso de datos, recursos de operación, o recursos de secuencia de eventos se define con el lenguaje *YANG*. Su vinculación con la *REST API* se basa en la proporción de una interfaz adicional simplificada siguiendo los principios de *REST* y que es compatible con un dispositivo de abstracción orientado a recursos. La clasificación de los datos como la configuración o la no configuración se deriva de la afirmación "*config*" *YANG*.

RESTCONF permite el acceso al almacén de datos localizado en el controlador. Hay dos almacenes de datos:

- *Config*: contiene datos insertados vía controlador.
- *Operational*: contiene datos insertados vía red.

Para acceder se utilizan peticiones *http* sobre el puerto 8080, soportando las operaciones OPTIONS, GET, PUT, POST, DELETE. Estas peticiones y respuestas de datos pueden ser representadas tanto en XML como en JSON. XML tiene estructura de acuerdo a yang con *XML-YANG* y JSON con *JSON-YANG*.

REST API* utilizando *xml

Para añadir una entrada a la tabla de flujos en nuestros *switches* podemos utilizar el lenguaje XML. Aquí tenemos un ejemplo en el que se descartan los paquetes según la dirección MAC del emisor:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
    <strict>false</strict>
    <instructions>
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <drop-action/>
                </action>
            </apply-actions>
        </instruction>
    </instructions>
    <table_id>0</table_id>
    <id>1</id>
    <cookie_mask>255</cookie_mask>
    <installHw>false</installHw>
    <match>
         <ethernet-match>
            <ethernet-source>
```

```
                <address>4a:97:2a:0f:63:d5</address>
            <ethernet-source>
        <ethernet-match>
    <match>
      <cookie>3</cookie>
    <flow-name>FooXf3</flow-name>
    <priority>15</priority>
    <barrier>false</barrier>
</flow>
```

Como podemos ver, en el código se muestran diferentes campos referentes al flujo como son direcciones MAC, tablas de flujo, prioridades o la propia acción de descarte de paquetes (*drop*).

Además, podemos añadir el flujo de dos formas diferentes: mediante línea de comandos o mediante una aplicación facilitada por *google chrome* llamada *Postman*.

· Añadir y eliminar flujo mediante línea de comandos.

Para añadir un flujo mediante comandos primero creamos un archivo .xml con la instrucción que deseamos añadir y luego ejecutamos:

```
# sudo curl –user admin:admin -i -X PUT -H “Content-Type:
application/xml; charset=utf-8” -d
@”/home/…/directorioficheroxml/prueba.xml”
http://127.0.0.1:8181/restconf/config/opendaylight-
inventory:nodes/node/OpenFlow:1/table/0/flow/1
```

De esta forma estaríamos añadiendo el flujo al *switch* 1 (*OpenFlow:1*), en la tabla 0 (*table/0*) y con identificador de flujo “1” (*flow/1*).

Para borrar reglas de flujo ejecutamos:

```
# sudo curl –user admin:admin -i -X DELETE -H “Content-Type:
application/xml; charset=utf-8”
http://127.0.0.1:8181/restconf/config/opendaylight-
inventory:nodes/node/OpenFlow:1/table/0/flow/1
```

· Añadir y eliminar flujo mediante la aplicación de *Postman*.

Si deseamos comunicarnos con nuestro controlador utilizando una Rest Api podemos descargar la aplicación de Chrome llamada *Postman* [46] que nos permitirá de forma sencilla llevar a cabo esta interacción con el controlador vía *http*.

Esta aplicación nos permitirá añadir flujos a nuestras tablas entre otras acciones posibles.

Abrimos la aplicación que se muestra en la sección aplicaciones del navegador *google chrome*:

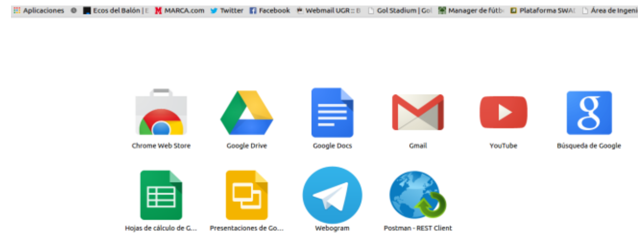


Figura 6.17. Menú de aplicaciones de google chrome donde se descarga Postman.

Una vez abierta encontraremos la siguiente interfaz:

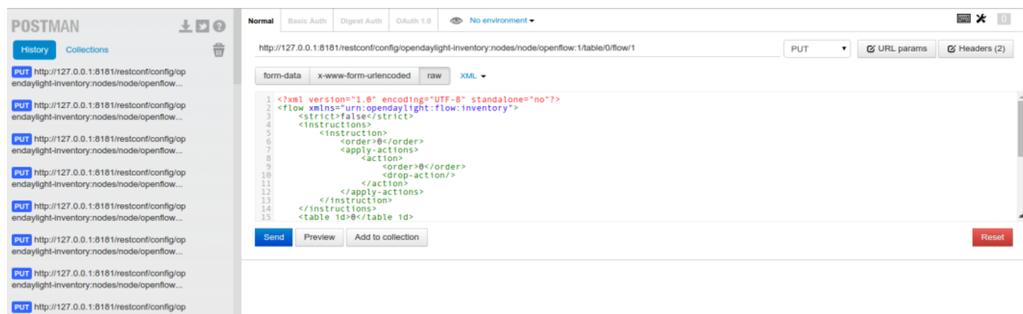


Figura 6.18. Interfaz de Postman.

A la izquierda tenemos todas y cada una de las acciones que hemos ejecutado con anterioridad. En el centro de la pantalla vemos una línea para rellenar con los elementos a modificar.

A continuación vamos a describir los pasos a seguir para la adición de flujo utilizando *Postman*.

1) En primer lugar, indicamos mediante una *url* donde vamos a instalar el flujo: http://127.0.0.1:8181/restconf/config/opendaylight-inventory:nodes/node/OpenFlow:1/table/0/flow/1

2) Ahora presionamos en el botón Headers y añadimos lo siguiente:

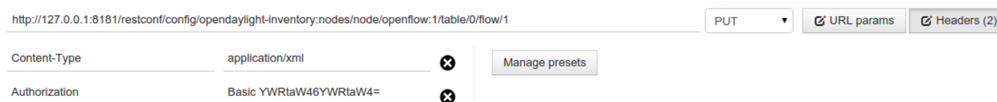


Figura 6.19. Cabecera de opciones de Postman.

En el campo *authorization* indicamos la contraseña, que, para *OpenDayLight*, ya comentamos que era *admin*.

3) Añadimos nuestro fichero XML en el que se describe el flujo que vamos a añadir.

Centrando nuestra atención en la dirección HTTP del primer paso, podemos señalar lo siguiente:

· OpenFlow:#, indica el *switch* que vas a tocar, en este caso el 1, por lo que ahí colocarías el número del *switch*.

· table/#/ es la tabla a modificar (por ahora solo he visto que funciona con la 0).

· flow/# es el id del flujo.

Estos datos deberán coincidir con lo que rellenemos en el XML que veremos a continuación. En el lateral se puede ver un desplegable. Si se desea introducir un flujo debe estar en *PUT* como se ve en la imagen.

Algunos tipos de flujo estándar utilizando *xml* pueden ser consultados en la *Wiki* de *OpenDayLight* [47]. Sólo faltaría rellenar el campo de texto *xml* con el flujo que deseamos añadir, como podría ser el que comentamos anteriormente para descartar paquetes, y enviar la petición.

· Ejemplo práctico

A continuación vamos a describir los pasos necesarios para realizar un ejemplo haciendo uso del método por línea de comandos.

En este ejemplo hemos tomado como referencia una topología formada por tres *switches* conectados entre sí con un *host* en cada uno de ellos.

Arrancamos el controlador, en este caso la versión *Helium* con *karaf*:

```
$ cd distribution-karaf-0.2.1-Helium-SR1
$ ./bin/karaf
```

Ponemos en marcha la topología, la cual puede consultarse en el Anexo G:

```
$ cd Escritorio/TFG
$ python 3_host_3_switch_manu.python
```

Una vez que se ha cargado el controlador y se conecta a los *switches* vamos a proceder a ejecutar un pingall en *mininet* para ver que todo funciona correctamente:

```
$ pingall
```

```
mininet> pingall
*** Ping: testing ping reachability
h3 -> h2 h1
h2 -> h3 h1
h1 -> h3 h2
*** Results: 0% dropped (6/6 received)
```

Figura 6.20. Test de ping en la red creada.

Podemos ver la topología que se ha creado si en nuestro navegador entramos en la dirección http://127.0.0.1:8181/dlux/index.html#/login.

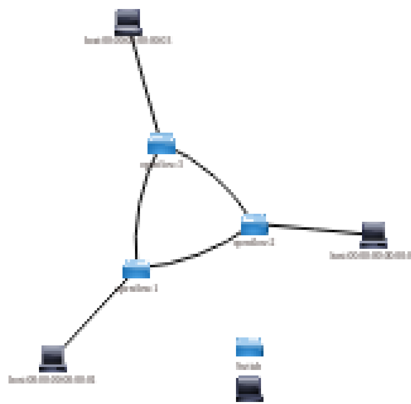


Figura 6.21. Imagen de la topología creada.

Una vez creada la topología y visto su correcta conectividad entre equipos vamos a echar un vistazo a los flujos instalados hasta el momento:

```
$ sh ovs-ofctl -O OpenFlow13 dump-flows s1
```

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b00000000000006, duration=72.999s, table=0, n_packets=15, n_bytes=1162, prior
ity=2,in_port=3 actions=output:2,output:1,CONTROLLER:65535
 cookie=0x2b00000000000005, duration=72.999s, table=0, n_packets=31, n_bytes=4574, prior
ity=2,in_port=1 actions=output:2,output:3
 cookie=0x2b00000000000004, duration=73.001s, table=0, n_packets=26, n_bytes=4084, prior
ity=2,in_port=2 actions=output:1,output:3
 cookie=0x2b00000000000004, duration=76.797s, table=0, n_packets=32, n_bytes=2016, prior
ity=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000000, duration=76.797s, table=0, n_packets=18, n_bytes=1316, prior
ity=0 actions=drop
```

Figura 6.22. Flujos instalados antes de ejecutar el flujo proactivo.

Como vemos, se han instalado una serie de flujos al hacer el ping. Ahora vamos a añadir un flujo mediante el cual indicaremos al *switch* 1 que, todo paquete cuya MAC fuente sea la del *host* 1 y cuya MAC destino sea la del *host* 2 la encamine por el puerto 1:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow xmlns="urn:opendaylight:flow:inventory">
<instructions>
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <output-action>
                        <output-node-connector>1</output-node-connector>
                        <max-length>60</max-length>
                    </output-action>
                </action>
            </apply-actions>
          </instruction>
    </instructions>
    <table_id>0</table_id>
    <id>1</id>
    <cookie_mask>255</cookie_mask>
    <installHw>false</installHw>
    <match>
        <ethernet-match>
            <ethernet-source>
                <address>00:00:00:00:00:01</address>
            </ethernet-source>
            <ethernet-destination>
                <address>00:00:00:00:00:02</address>
            </ethernet-destination>
        </ethernet-match>
    </match>
    <cookie>3</cookie>
    <flow-name>FooXf3</flow-name>
    <priority>15</priority>
    <barrier>false</barrier>
</flow>
```

Abrimos un nuevo terminal en el que enviaremos el flujo proactivo a nuestro controlador:

```
$ sudo curl –user admin:admin -i -X PUT -H “Content-Type:
application/xml; charset=utf-8” -d
@”/home/manu/put_flow/prueba.xml”
http://127.0.0.1:8181/restconf/config/opendaylight-
inventory:nodes/node/OpenFlow:1/table/0/flow/1
```

En la siguiente imagen vemos tanto la petición de envío como la respuesta recibida tras enviarla:

```
root@manu-HP-Pavilion-dv7-Notebook-PC:/home/manu# sudo curl --user admin:admin -
i -X PUT -H "Content-Type: application/xml; charset=utf-8" -d @"/home/manu/put_f
low/prueba.xml" http://127.0.0.1:8181/restconf/config/opendaylight-inventory:nod
es/node/openflow:1/table/0/flow/1
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Content-Length: 0
Server: Jetty(8.1.14.v20131031)
```

Figura 6.23. Envío de flujo proactivo y respuesta del servidor.

Ahora vamos a comprobar si se ha añadido el flujo indicado. Volvemos a mostrar la tabla de flujo:

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x3, duration=1.277s, table=0, n_packets=0, n_bytes=0, idle_timeout=300
 hard_timeout=600, send_flow_rem priority=15,dl_src=00:00:00:00:00:01,dl_dst=00
00:00:00:00:02 actions=output:1
 cookie=0x2b0000000000000b, duration=1.283s, table=0, n_packets=0, n_bytes=0, pr
iority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b0000000000000b, duration=1.283s, table=0, n_packets=10, n_bytes=808,
 priority=0 actions=drop
```

Figura 6.24. Tabla de flujo tras instalación de flujo proactivo.

Como vemos, el flujo se ha añadido correctamente, pero falta comprobar si nuestro *switch* encaminará los paquetes si se cumplen los requisitos marcados. Se realiza un ping entre el *host* 1 y el *host* 2 y se vuelve a comprobar el flujo:

```
 cookie=0x3, duration=105.43s, table=0, n_packets=624, n_bytes=61152, idle_timeo
ut=300, hard_timeout=600, send_flow_rem priority=15,dl_src=00:00:00:00:00:01,dl_
dst=00:00:00:00:00:02 actions=output:1
```

Figura 6.25. Comprobación de utilización del flujo proactivo.

Con la imagen anterior queda demostrado que el método para añadir flujos de forma proactiva mediante *REST API* funciona correctamente.

6.2.7. Flujo reactivo.

Continuamos ahora hablando sobre la instalación de flujos reactivos en el controlador. A diferencia del flujo proactivo, que se incluye de manera explícita por parte del administrador de red, la instalación del flujo reactivo está prevista en caso de cumplirse ciertos requisitos. Este tipo de flujos son los que mayor importancia tienen en el presente proyecto, ya que un DPI se basa en detectar flujos que cumplan un cierto *match*.

En este apartado vamos a ver cómo decodificar los paquetes entrantes, cómo añadir flujos a las tablas de encaminamiento incluyendo *match* y acciones y, por último, cómo reenviar los paquetes.

Desarrollo del controlador.

Siempre que llegue un paquete sin un *matching* en las entradas de las tablas de flujo en el *switch*, será enviado al controlador y se llamará a *receiveDataPacket()*. Aquí será donde gestionemos el paquete para ver si se amolda a algún tipo de *match* preestablecido y se asignarán las acciones pertinentes. Una implementación sencilla de esta clase puede ser:

```
public PacketResult receiveDataPacket(RawPacket inPkt) {
        if (inPkt == null) {
            return PacketResult.IGNORED;
        }

        logger.trace("Received a frame of size: {}", inPkt.getPacketData().length);

        Packet formattedPak = this.dataPacketService.decodeDataPacket(inPkt);
        NodeConnector incoming_connector = inPkt.getIncomingNodeConnector();
        Node incoming_node = incoming_connector.getNode();

        if (formattedPak instanceof Ethernet) {
            byte[] srcMAC = ((Ethernet)formattedPak).getSourceMACAddress();
            byte[] dstMAC = ((Ethernet)formattedPak).getDestinationMACAddress();

            long srcMAC_val = BitBufferHelper.toNumber(srcMAC);
            long dstMAC_val = BitBufferHelper.toNumber(dstMAC);

            Match = new Match();
            match.setField( new MatchField(MatchType.IN_PORT, incoming_connector) );
            match.setField( new MatchField(MatchType.DL_DST, dstMAC.clone()) );

            // Set up the mapping: switch -> src MAC address -> incoming port
            if (this.mac_to_port_per_switch.get(incoming_node) == null) {
                this.mac_to_port_per_switch.put(incoming_node, new HashMap<Long,
                                                NodeConnector>());
            }
            this.mac_to_port_per_switch.get(incoming_node).put(srcMAC_val,
                                                             incoming_connector);

            NodeConnector dst_connector = this.mac_to_port_per_switch
                                          .get(incoming_node).get(dstMAC_val);

            // Do I know the destination MAC?
            if (dst_connector != null) {

                List<Action> actions = new ArrayList<Action>();
                actions.add(new Output(dst_connector));

                Flow f = new Flow(match, actions);

                // Modify the flow on the network node
                Status = programmer.addFlow(incoming_node, f);
                if (!status.isSuccess()) {
                    logger.warn("SDN Plugin failed to program the flow: {}. The failure
```

```
                                is: {}",f, status.getDescription());
                        return PacketResult.IGNORED;
                    }
                    logger.info("Installed flow {} in node {}",
                                f, incoming_node);
                }
                else
                    floodPacket(inPkt);
            }
            return PacketResult.IGNORED;
        }
```

Como vemos, en este ejemplo simplemente se comprueba si el paquete es de tipo *Ethernet* para añadir el flujo. Una vez detectado un paquete de este tipo añadimos los *matches* de direcciones *MAC* y puertos para, más tarde, añadir una acción indicando hacia donde se debe reenviar el paquete. De esta forma estaremos añadiendo un flujo reactivo a nuestra tabla, de tal forma que cuando llegue un paquete que cumpla el *matching* previamente marcado se procederá a aplicar dicho flujo previamente instalado.

En caso de que el paquete no sea de tipo *Ethernet* se inundará por todos los puertos del *switch*.

· Ejemplo práctico.

Tras una breve explicación del desarrollo del controlador para la instalación de un flujo reactivo bastante sencillo, vamos a pasar a realizar un pequeño ejemplo que demuestre el correcto funcionamiento de lo implementado.

La topología creada es la mostrada en la figura siguiente. Como vemos, es una topología en árbol formada por dos *switches* que penden de un tercero. Cada uno de estos dos *switches* tienen en su red a dos *hosts* cada uno.

```
manu@manu-HP-Pavilion-dv7-Notebook-PC:~$ sudo mn --controller=remote --topo tree
,2 --mac --arp --switch ovsk,protocols=OpenFlow10
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3
*** Starting CLI:
```

Figura 6.26. Topología en árbol creada para probar la adición de flujo reactivo.

Realizamos un *ping* entre el *host* 1, que pende del *switch* 2, y el *host* 4, que pende del *switch* 3.

```
mininet> h1 ping h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=9 ttl=64 time=0.227 ms
64 bytes from 10.0.0.4: icmp_seq=10 ttl=64 time=0.094 ms
64 bytes from 10.0.0.4: icmp_seq=11 ttl=64 time=0.099 ms
```

Figura 6.27. Ping realizado entre el host 1 y el host 4.

Podemos ver cómo, de inmediato, en el controlador se añaden los flujos:

```
2015-06-14 14:28:30.599 CEST [SwitchEvent Thread] INFO  o.o.t.t.i.TutorialL2Forw
arding - Installed flow Flow[match = Match [fields={IN_PORT=IN_PORT(OF|3@OF|00:0
0:00:00:00:00:00:03), DL_DST=DL_DST(00:00:00:00:00:04)}, matches=5], actions = [
OUTPUT[OF|2@OF|00:00:00:00:00:00:00:03]], priority = 0, id = 0, idleTimeout = 0,
 hardTimeout = 0] in node OF|00:00:00:00:00:00:00:03
2015-06-14 14:28:31.563 CEST [SwitchEvent Thread] INFO  o.o.t.t.i.TutorialL2Forw
arding - Installed flow Flow[match = Match [fields={IN_PORT=IN_PORT(OF|1@OF|00:0
0:00:00:00:00:00:02), DL_DST=DL_DST(00:00:00:00:00:04)}, matches=5], actions = [
OUTPUT[OF|3@OF|00:00:00:00:00:00:00:02]], priority = 0, id = 0, idleTimeout = 0,
 hardTimeout = 0] in node OF|00:00:00:00:00:00:00:02
```

Figura 6.28. Logs en el controlador que avisan de la adición de flujos reactivos.

Comprobamos que el flujo se ha añadido en el controlador:

```
mininet> sh ovs-ofctl -O OpenFlow10 dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=39.493s, table=0, n_packets=4, n_bytes=392, idle_age=35, p
riority=0,in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=41.509s, table=0, n_packets=6, n_bytes=588, idle_age=35, p
riority=0,in_port=1,dl_dst=00:00:00:00:00:04 actions=output:2
```

Figura 6.29. Flujos añadidos en el controlador debido al ping.

Podemos observar cómo se añadieron los flujos y cuántos paquetes han sido clasificados en los mismos (en este caso 4 y 6).

Capítulo 7

Diseño e implementación del DPI

Este capítulo puede ser considerado el más importante dentro de esta memoria. En él vamos a describir el proceso de implementación y diseño del DPI; objetivo principal del proyecto.

Tras haber desgranado una serie de conceptos relacionados con las redes SDN y lograr familiarizarnos con las herramientas de trabajo fundamentales como son *Mininet* y *OpenDayLight*, nos disponemos a hacer uso de estos conocimientos para conseguir la implementación de un *Deep Packet Inspector* que sea capaz de detectar varios tipos de tráfico, como el referente a *YouTube*, e instalar en los *switches* los flujos reactivos correspondientes para conferir ciertas etiquetas que faciliten un tratamiento diferenciado a dichos flujos.

7.1. Conceptos previos.

Antes de adentrarnos en la implementación del DPI vamos a dar una visión sobre algunos aspectos de relevancia que facilitarán la comprensión de los siguientes puntos.

Como ya se hizo referencia anteriormente, el desarrollo de la mayor parte del DPI ha utilizado el modelo AD-SAL. El motivo principal de esta elección se fundamenta en que se encontraron algunos problemas que se comentarán el en Anexo D con el modelo MD-SAL los cuales, pese a que no imposibilitan del todo nuestra labor, son un obstáculo para algunas implementaciones. No obstante, también se ha desarrollado un ejemplo con este modelo, ya que las perspectivas futuras están fundamentadas en MD-SAL.

Por otra parte, para nuestro DPI hemos focalizado nuestros esfuerzos en la detección de tráfico *YouTube*, por lo que será este tipo de tráfico el que mayor importancia tenga a la hora del desarrollo del DPI. Sin embargo, también se han implementado las detecciones de otros tipos de tráfico más sencillos.

También se mostrará la flexibilidad de nuestro DPI para poder ser implementado como un producto independiente en una red actual, acoplándose a otros entornos de red sin verse afectadas sus características.

Comenzamos pues a describir la implementación de un DPI en *OpenDayLight* ofreciendo una emulación de red con *Mininet*. Vamos a centrarnos en los métodos que afectan de forma directa o indirecta al desarrollo del *Deep Packet Inspector* en el controlador.

7.2. Implementación con MD-SAL.

Para la implementación de un DPI con *MD-SAL* hemos optado por un ejemplo bastante sencillo debido a una serie de fallos encontrados y a la complicación del mismo. Para la realización del ejemplo se ha utilizado el paquete que proporciona *SDNHUB* [48]. Este paquete ofrece tanto la arquitectura *AD-SAL* como *MD-SAL*. El controlador en *MD-SAL* tiene las siguientes clases:

· *Activator.java*

· *DataChangeListenerRegistrationHolder.java*

· *FlowComminWrapper.java*

· *FlowCommitWrapperImpl.java*

· *FlowUtils.java*

· *InstanceIdentifierUtils.java*

· *LearningSwitchHandler.java*

· *LearningSwitchHandlerSimpleImpl.java*

· *LearningSwitchManager.java*

· *LearningSwitchManagerSimpleImpl.java*

· *PacketUtils.java*

· *WakeupOnNode.java*

Todas ellas son indispensables para el funcionamiento del conjunto. En nuestro caso, para implementar un sencillo DPI, nos vamos a focalizar en *LearningSwitchHandlerSimpleImpl.java* y en *FlowUtils.java*.

7.2.1. Modificación de *ToS* a los flujos de tipo ARP.

El objetivo de este ejemplo será detectar el tráfico ARP y añadirle un nuevo *Type of Service*. Para ello, lo primero será detectar este tipo de paquetes.

Dentro de la clase *LearningSwitchManagerSimpleImpl.java*, la función *opPacketReceived(PacketReceived notification)* será la encargada de recibir y analizar los paquetes. Aquí realizaremos la inspección de los mismos y detectaremos los que son de tipo ARP. Para ello se utilizará el siguiente código:

```
    if(Arrays.equals(ETH_TYPE_ARP, etherType)){
         LOG.info("Es de tipo ARP");
         añadirNuevoToS=true;
  }
```

Con la variable *booleana* "añadirNuevoToS" estaremos indicando que este tipo de tráfico cambiará su *ToS* en la función que añade los flujos.

Esta función es *addBridgeFlow()*. Dentro de la función mencionada se llama al constructor de flujos, el cual se encuentra dentro de la clase *FlowUtils.java*, que será donde añadamos la variable *booleana* y, en caso de *true*, se añadirá la acción de cambio de *ToS* como vemos aquí:

```
private void addBridgeFlow(MacAddress srcMac, MacAddress dstMac, NodeConnectorRef
destNodeConnector) {
        synchronized (coveredMacPaths) {
            String macPath = srcMac.toString() + dstMac.toString();
            if (!coveredMacPaths.contains(macPath)) {
                LOG.debug("covering mac path: {} by [{}]", macPath,
                           destNodeConnector.getValue().firstKeyOf(NodeConnector.class,
                           NodeConnectorKey.class).getId());

                coveredMacPaths.add(macPath);
                FlowId = new FlowId(String.valueOf(flowIdInc.getAndIncrement()));
                FlowKey = new FlowKey(flowId);
                /**
                 * Path to the flow we want to program.
                 */
                InstanceIdentifier<Flow>flowPath =
                InstanceIdentifierUtils.createFlowPath(tablePath, flowKey);

                Short tableId = InstanceIdentifierUtils.getTableId(tablePath);
                LOG.info("Creando mactomacFlow");
                FlowBuilder srcToDstFlow = FlowUtils.createDirectMacToMacFlow(tableId,
                                           DIRECT_FLOW_PRIORITY, srcMac,dstMac,
                                           destNodeConnector, añadirNuevoToS);
                srcToDstFlow.setCookie(new FlowCookie(BigInteger.valueOf(
                                       flowCookieInc.getAndIncrement())));

                dataStoreAccessor.writeFlowToConfig(flowPath, srcToDstFlow.build());
            }
        }
```

Nos vamos ahora a la función *createDirectMacToMacFlow()* dentro de la clase *FlowUtils.java*. En ella añadimos la acción en caso de que la variable *booleana* sea *true* como ya indicamos. La manera de hacerlo es la siguiente:

```
 if(añadirToS){
               LOG.info("Añadiendo nuevo TOS");
               SetNwTosAction nwTos = new SetNwTosActionBuilder()
                              .setTos(252)
                              .build();
               Action nwTosAction = new ActionBuilder() //
                              .setOrder(1)
                              .setAction(new SetNwTosActionCaseBuilder()
                                             .setSetNwTosAction(nwTos)
                                             .build())
                              .build();
              actions.add(outputToControllerAction);
              actions.add(nwTosAction);
        }
```

Por último vamos a probar que realmente funcione la implementación. Antes de comenzar cabe señalar un problema que se explicará de forma más detallada en el Anexo D y es que, para que los flujos se añadan, tenemos que reiniciar la topología.

Para las pruebas hemos optado por una topología sencilla con un *switch* y tres *hosts*:

```
$ sudo mn --controller=remote --topo single,3 --mac --arp --
switch ovsk,protocols=OpenFlow10
```

Arrancamos el controlador de *MD-SAL* dentro del paquete *SDNHUB* y en su respectivo directorio:

```
root@manu-HP-Pavilion-dv7-Notebook-
PC:/home/manu/SDNHub_Opendaylight_Tutorial/distribution/opendayl
ight-osgi-mdsal/target/distribution-osgi-mdsal-1.1.0-SNAPSHOT-
osgipackage/opendaylight# ./run.sh
```

Realizamos un ping entre *h1* y *h2*:

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=50.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=6.28 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=5.19 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=4.27 ms
```

Figura 7.1. Ping entre dos host utilizando controlador con arquitectura MD-SAL.

Como vemos, los tiempos son bastante altos. Esto se debe a lo que comentamos acerca de añadir el flujo, y es que éste no se añadirá hasta haber reiniciado la topología.

Al reiniciar la topología y ver los flujos instalados podemos apreciar el cambio de *ToS* en el paquete ARP:

```
 cookie=0x2a00000000000001, duration=8.983s, table=0, n_packets=0, n_bytes=0, id
le_age=8, priority=512,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02 actions
=output:2,mod_nw_tos:252
```

Figura 7.2.2. Adición de ToS a paquete ARP en arquitectura MD-SAL.

7.3. Implementación con AD-SAL.

7.3.1. Detección de tráfico *web* y VoIP.

Para la detección de estos dos tipos de tráfico nos hemos centrado en detectar los puertos que utilizan ambos servicios a parte del protocolo sobre el que estos servicios se soportan. Por lo tanto, su detección es más bien sencilla a priori.

Para el tráfico *web* hemos implementado una función que detecta si el puerto utilizado es el 80 (*http*) o el 443 (*https*). En el caso de *web* lo que buscaremos son paquetes de tipo TCP mientras que en VoIP son los paquetes UDP los que nos interesan.

```
   /**
     * Función que comprueba si es tráfico VoIP
     * @param udpPacket paquete udp que vamos a analizar
     * @return devolvemos una booleana que informará si el paquete es VoIP o no
     */

private boolean isVoIPTraffic(UDP udpPacket){

        boolean VoIPTraffic = false;

        if( udpPacket.getDestinationPort()==4569 /*IAX --> UDP*/
         || udpPacket.getDestinationPort()==9082 /*Prueba Skype --> UDP*/ ){
                VoIPTraffic=true;
        }

        return VoIPTraffic;
    }

    /**
     * Función que comprueba si el tráfico es de tipo WEB
     * @param tcpPacket Paquete tcp recibido
     * @return devolvemos una booleana que informará si el paquete es http o no
     */
    private boolean isWebTraffic(TCP tcpPacket){

        boolean webTraffic = false;
        int dstPort = tcpPacket.getDestinationPort();
        int srcPort = tcpPacket.getSourcePort();
        if(dstPort==80 || srcPort==80 || dstPort==443 || srcPort==443){
                webTraffic=true;
        }

        return webTraffic;
    }

}
```

Con estas dos funciones seremos capaces de identificar estos dos tipos de tráfico. Esta sería la parte correspondiente al DPI. No obstante, el controlador deberá implementar dichas funciones y añadir el flujo correspondiente.

7.3.2. Detección de tráfico *YouTube*.

La detección de tráfico *YouTube* podemos dividirla en dos partes. En primer lugar, la detección de los paquetes DNS referentes a este tipo de tráfico es fundamental para conseguir la dirección del servidor que nos proporcionará el vídeo. Por lo tanto, en primera instancia hay que realizar un análisis (*parsing*) de los mensajes DNS. En segundo lugar habrá que esperar la llegada de los paquetes con la dirección previamente almacenada para añadir el flujo correspondiente.

Análisis de paquetes DNS.

La primera condición para comenzar el proceso de detección de tráfico YouTube será la llegada de un paquete DNS. Si dicho paquete DNS contiene el *string* "*XXX.googlevideo.com*" podremos identificarlo como un paquete enviado por los servidores DNS con información relativa al tráfico de *YouTube*. El método implementado para este cometido se muestra a continuación:

```
/**
   * Función que comprueba si el tráfico pertenece a un flujo de
   * streaming de vídeo enviado por YouTube. Con esta función
   * detectamos el servidor DNS que envía la dirección IP del
   * servidor de vídeo.
   * @param arrayPacketData: es el paquete de datos en el que
   * buscaremos el string "googlevideo"
   * @return Nos devuelve una booleana indicando si el paquete es
   * de tráfico YouTube o no.
   */
  private boolean isYoutubeTraffic(byte[] arrayPacketData){

      boolean youtubeTraffic = false;
      String datosUDP = new String(arrayPacketData, UTF_8);
       if(datosUDP.contains("googlevideo")){
              // ...añadiremos el ToS más adelante en el flujo en caso de "true"
              log.info("Es YouTube");
              youtubeTraffic=true;
       }
      return youtubeTraffic;
  }
```

Una vez detectemos dicho *string* pasaremos a analizar el paquete DNS. El análisis hemos de implementarlo ya que, a día de hoy, no existe un módulo de *parsing* para DNS implementado en *OpenDayLight*. Por lo tanto, a partir de los paquetes UDP que vayan dirigidos al puerto 53 se buscará la respuesta a la petición DNS realizada por el equipo que desea acceder al contenido del vídeo de YouTube.

En primer lugar habrá que comprobar que el paquete es de tipo *YouTube* y que el puerto origen es el 53, ya que lo que buscamos son las respuestas DNS, en las que se incluye la dirección IP del servidor de vídeo. Una vez comprobados estos dos requisitos añadimos la

dirección IP del servidor DNS de *YouTube* a una lista de direcciones. El motivo de añadir el servidor DNS se debe a que necesitamos no añadir flujos con esta dirección, ya que en ese caso no podríamos optar a detectar nuevos tráficos de vídeo, ya que el tráfico dejaría de pasar por el controlador. Tras registrar esta dirección, pasamos a realizar el análisis del paquete mediante la función "*parseoDNS*". El *parsing* se realizará a partir del *payload* del paquete UDP en los paquetes referentes a la respuesta DNS.

Primero se muestran 4 *bytes* que son la cabecera del paquete DNS, los cuales no contienen información interesante para nosotros. Seguidamente encontramos 16 *bits* indican el número de peticiones. Esto es importante para después poder determinar dónde empiezan las respuestas. Otros 16 *bits* que indican el número de respuestas. Después tenemos 16 *bits*, que representan el número de autoridades. A continuación, 16 *bits*, indican el número de registros de recursos adicionales. A partir de aquí empiezan las peticiones (preguntas que se han hecho en mensajes anteriores y que se responden con este paquete). Tienen primero la cadena de la pregunta, 16 *bits* que indican el tipo de registros de recursos (*Address, Canonical Name,* etcétera) y 16 *bits* que indican la clase.

Para determinar la longitud de las peticiones (cuya longitud es variable), hemos optado por la detección del carácter "\0" (un *byte* a 00) que indica el fin de la cadena. Tras la detección de todas las peticiones pasamos a las respuestas, donde estará la dirección *IP* del servidor de vídeo que buscamos.

Las respuestas tienen un campo de 16 *bits* que hace referencia al recurso sobre el que se responde, después 16 *bits* de tipo (buscaremos una respuesta de tipo *Address=1*, ya que en esta respuesta se incluirá la dirección IP del servidor), 16 *bits* de clase, 32 *bits* de *TTL*(*Time To Live*), 16 *bits* de longitud de la respuesta (4 *bytes* si la respuesta es de tipo *Address*, ya que hay que poner la dirección IP; pero cualquier valor si es una respuesta de tipo *Canonical Name*, ya que vendría representado por una cadena de caracteres indicando la *url*) y después la respuesta (con el número de *bytes* indicado por el campo anterior). En la siguiente figura se ve con claridad el formato de los paquetes DNS.

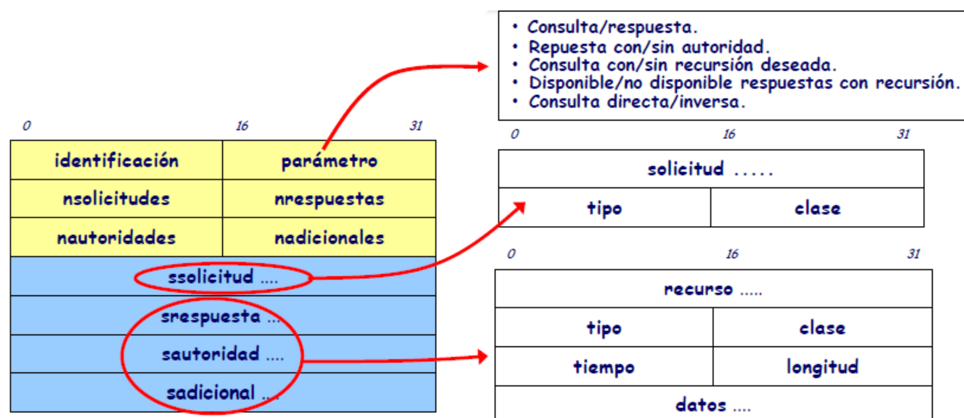


Figura 7.2. Formato de paquetes DNS,

Una vez claros estos conceptos los hemos plasmado en el código para implementar un análisis que facilitara la dirección IP del servidor:

```
/**
 * Función encargada de parsear los paquetes DNS con el objeto
 * de encontrar la dirección IP del servidor encuadrada dentro
 * de las respuestas de este tipo de paquetes.
 * @param udpRawPayload: paquete UDP en el que se encuentra
 * incluido el paquete DNS
 * @return srcIPservidorVideoYoutube: dirección IP del servidor.
 **/
private InetAddress parseoDNS(byte[] udpRawPayload){
    String udpRawPayDataISO = new String(udpRawPayload, ISO_8859_1);
    byte[] numPeticiones= new byte[2];
    byte[] numRespuestas = new byte[2];
    byte[] Type;
    byte[] Class;
    boolean urlencontrada=false;
    int longitudPeticiones=12;
    int longitudRespuestas=0;
    int longitudURL=0;
    numPeticiones[0]=udpRawPayload[4];
    numPeticiones[1]=udpRawPayload[5];
    numRespuestas[0]=udpRawPayload[6];
    numRespuestas[1]=udpRawPayload[7];
    int numQueries=0;
    int numAnswers=0;
    numQueries=numPeticiones[1];
    numAnswers=numRespuestas[1];
    if(numAnswers!=0){
            for(int j=0;j<numQueries;j++){
                    urlencontrada=false;
                    for(int i=longitudPeticiones;i<udpRawPayload.length-2;i++){
                        if(udpRawPayload[i+1]==0 && !urlencontrada){
                            log.info("Primer 0 encontrado");
                            if(udpRawPayload[i+2]==0){
                                    log.info("Segundo 0 encontrado");
                                    longitudPeticiones=i+8;
                                    log.info("número de petición "+j);
                                    j++;
                                    urlVideoYoutube=udpRawPayDataISO.substring(13,i+1);
                                    longitudURL=urlVideoYoutube.length();
                                    urlencontrada=true;
                            }
                        }
                    }
            }
            longitudRespuestas=longitudPeticiones;
            for(int j=0; j<numAnswers;j++){
                    Type = new byte[numAnswers];
                    Type[j]=udpRawPayload[longitudRespuestas+1];
                    log.info("Type: "+Type[j]);
                    Class = new byte[numAnswers];
                    Class[j]=udpRawPayload[longitudRespuestas+3];
                    log.info("Class: "+Class[j]);
                    if(Type[j]==5){
                        longitudRespuestas=longitudRespuestas+14+(longitudURL-17);
                        //El primary Name tiene 17 bytes menos que la url
                    }
                    if(Type[j]==1){
```

```
                        byte[] ipAddr=new byte[4];
                        ipAddr[0]=udpRawPayload[longitudRespuestas+10];
                        ipAddr[1]=udpRawPayload[longitudRespuestas+11];
                        ipAddr[2]=udpRawPayload[longitudRespuestas+12];
                        ipAddr[3]=udpRawPayload[longitudRespuestas+13];
                        try {
                             srcIPservidorVideoYoutube =
                             srcIPservidorVideoYoutube.getByAddress(ipAddr);
                        } catch (UnknownHostException e) {
                             // TODO Auto-generated catch block
                             e.printStackTrace();
                        }
                 }
          }
      }
      log.info("IP del servidor de Vídeo YouTube: "+srcIPservidorVideoYoutube);
      return srcIPservidorVideoYoutube;
  }
```

En este punto ya tendríamos la dirección IP del servidor de *YouTube*. El siguiente paso sería almacenarla en una lista de direcciones para que, una vez llegue un paquete de esta dirección, se detecte y se proceda a añadir un nuevo flujo cuyo *ToS* lo diferencie del resto para asignarle si es preciso una calidad de servicio mayor.

Dentro del manejador de paquetes recibidos *receiveDataPacket()*, en primer lugar, si el paquete es de tipo IP guardaremos la dirección IP en una variable y compararemos con nuestra lista de direcciones de servidores de vídeo.

```
      if (l3Pkt instanceof IPv4) {
             IPv4 ipv4Pkt = (IPv4) l3Pkt;
             Object l4Datagram = ipv4Pkt.getPayload();
             srcIPaddr = intToInetAddress(ipv4Pkt.getSourceAddress());
             dstIPaddr = intToInetAddress(ipv4Pkt.getDestinationAddress());
             ///
             if(srcIPservidorVideoYoutubeList.contains(srcIPaddr) && srcIPaddr!=null){
                    servidorVideoYoutube=true;
                    log.info("Lista de direcciones de vídeo YouTube:
                            "+srcIPservidorVideoYoutubeList);
             } else{
                    log.info("srcIP: "+srcIPaddr+" ipDNS:
                            "+srcIPservidorVideoYoutube);
             }
```

Podemos ver como se incluye una variable *booleana* para indicar si es tráfico *YouTube* o no. Esta variable *booleana* la utilizaremos a la hora de añadir el flujo. En el siguiente fragmento de código se llevaría a cabo la implementación de acciones al flujo.

```
if (dst_connector != null) {
                List<Action> actions = new ArrayList<Action>();
                if(servidorVideoYoutube){
                      servidorVideoYoutube=false;
                      añadeFlujo=true;
```

```
                log.info("Nuevo ToS para tráfico YouTube");
                actions.add(new SetNwTos(8));
            } else if(direccionesServidoresDNSYoutube.contains(srcIPaddr)){
                añadeFlujo=false;
                log.info("IP de servidor DNS YouTube; no añadir flujo");
...
```

Ejemplo práctico.

Pasamos ahora a realizar un ejemplo de detección de tráfico *YouTube* en una red conformada por un *switch* del que penden tres *hosts*.

En primer lugar arrancamos el controlador y la topología en dos terminales diferentes. La topología será:

```
$ sudo mn --controller=remote --topo single,3 --mac --arp --
switch ovsk,protocols=OpenFlow13 --nat
```

Una vez la red está en marcha vamos a desplegar una consola para el *host* 1:

```
mininet> xterm h1
```

En dicho terminal vamos a configurar el servidor DNS que recibirá nuestras peticiones. Esto variará según la red a la que estemos conectados. En nuestro proyecto se ha trabajado en dos redes: la red doméstica y la red de la UGR. Para la red doméstica indicamos el servidor DNS 8.8.8.8 correspondiente a *Google*. Si estamos en la red de la UGR podemos indicar el 150.214.35.10. Estos cambios se realizan en el fichero */etc/resolv.conf*, el cual quedaría de la siguiente manera para la red de la UGR:

```
# Dynamic resolv.conf(5) file for glibc resolver(3)
generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL
BE OVERWRITTEN
nameserver 150.214.35.10
search ugr.es
```

Ejecutamos el navegador de *Firefox* en la terminal del *host* 1 y buscamos un vídeo de *YouTube* para reproducirlo:

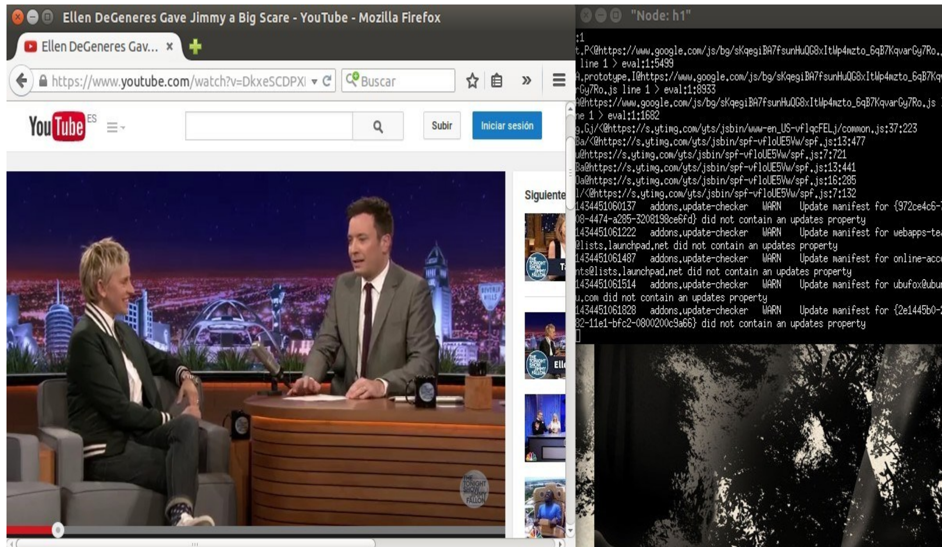


Figura 7.3. Reproducción de vídeo de YouTube en host 1.

Ahora vamos a pasar a ver si se han instalado los flujos del servidor de vídeo de *YouTube* tal y como deseamos. Primero ejecutamos el comando que muestra las tablas de flujo del *switch*:

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
```

En la siguiente imagen se muestra el flujo correspondiente con el *ToS* cambiado al valor que deseábamos:

```
 cookie=0x0, duration=44.784s, table=0, n_packets=9986, n_bytes=14136627, send_f
low_rem priority=0,tcp,in_port=4,dl_src=00:00:00:00:00:04,dl_dst=00:00:00:00:00:
01,nw_src=130.206.193.12,nw_dst=10.0.0.1,tp_dst=40434 actions=set_field:8->nw_to
s_shifted,output:1
```

Figura 7.4. Flujo añadido de tráfico de vídeo YouTube.

Esta dirección a la que se le asigna un nuevo *ToS* y que, por lo tanto, se presenta como la del servidor de vídeo *YouTube* se ha contrastado utilizando *wireshark*. Lo que se va a realizar mediante el cambio del *ToS* será modificar el campo *DSCP* (*Differentiated Services Code Point*) dentro del campo *DiffServ* de la cabecera IP al valor indicado. Además, como podemos comprobar el número de paquetes es muy elevado, claro reflejo de que es un *streaming* de vídeo.

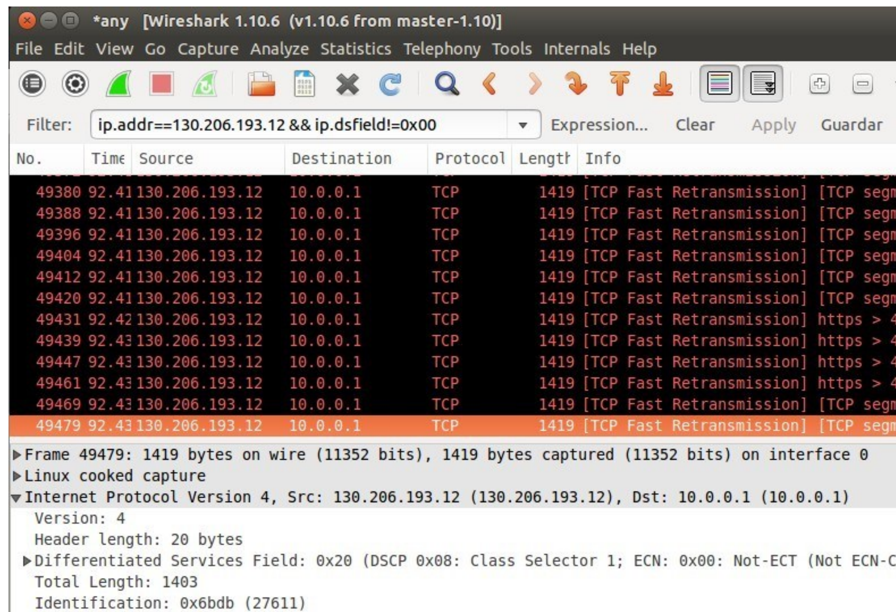


Figura 7.5. Captura de wireshark en la que se aprecia el cambio de ToS en los paquetes.

A continuación vamos a complicar un poco más la topología. En este caso optamos por una topología en árbol de tres niveles. El objetivo es cerciorarnos de que se añadirá una acción para modificar el *ToS* en las tablas de flujo de cada *switch*.

Con el controlador arrancado, procedemos a crear la nueva topología:

```
$ sudo mn --controller=remote --topo tree,3 --mac --arp --switch
ovsk,protocols=OpenFlow13 --nat
```

La composición de la misma se especifica en las siguientes figuras.

```
mininet> net
h1 h1-eth0:s3-eth1
h2 h2-eth0:s3-eth2
h3 h3-eth0:s4-eth1
h4 h4-eth0:s4-eth2
h5 h5-eth0:s6-eth1
h6 h6-eth0:s6-eth2
h7 h7-eth0:s7-eth1
h8 h8-eth0:s7-eth2
nat0 nat0-eth0:s1-eth3
s1 lo:  s1-eth1:s2-eth3 s1-eth2:s5-eth3 s1-eth3:nat0-eth0
s2 lo:  s2-eth1:s3-eth3 s2-eth2:s4-eth3 s2-eth3:s1-eth1
s3 lo:  s3-eth1:h1-eth0 s3-eth2:h2-eth0 s3-eth3:s2-eth1
s4 lo:  s4-eth1:h3-eth0 s4-eth2:h4-eth0 s4-eth3:s2-eth2
s5 lo:  s5-eth1:s6-eth3 s5-eth2:s7-eth3 s5-eth3:s1-eth2
s6 lo:  s6-eth1:h5-eth0 s6-eth2:h6-eth0 s6-eth3:s5-eth1
s7 lo:  s7-eth1:h7-eth0 s7-eth2:h8-eth0 s7-eth3:s5-eth2
c0
```

Figura 7.6. Distribución de interfaces de topología en árbol de tres niveles creada en Mininet.

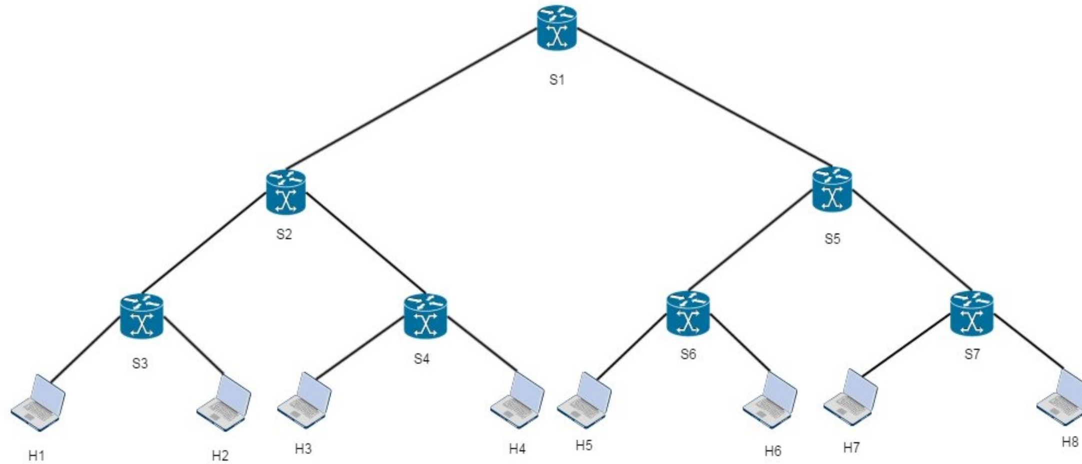


Figura 7.7. Topología en árbol de tres niveles.

Como vemos, si el vídeo lo solicita el *host* 1, el tráfico deberá pasar primero por el *switch* 1 (*s1*), de éste al *switch* 2 (*s2*) y, por último, al *switch* 3 (*s3*) que es donde está ubicado el *host*.

Procedemos de la misma forma que en el ejemplo anterior y reproducimos el vídeo que se muestra en la figura.

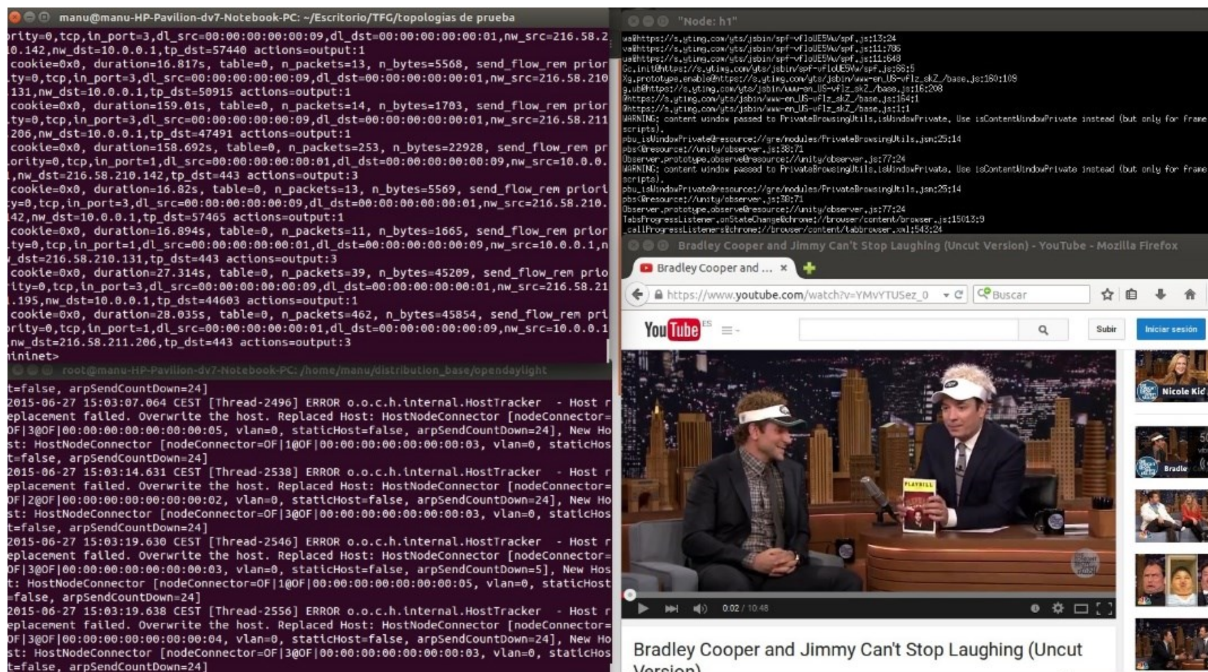


Figura 7.8. Reproducción de vídeo de YouTube por parte de h1.

Pasamos ahora a identificar los flujos creados en los *switches* 1, 2 y 3 en ese orden:

```
 cookie=0x0, duration=35.299s, table=0, n_packets=14917, n_bytes=21738085, send_flow_r
m priority=0,tcp,in_port=3,dl_src=00:00:00:00:00:09,dl_dst=00:00:00:00:00:01,nw_src=74
125.168.44,nw_dst=10.0.0.1,tp_dst=48480 actions=set_field:8->nw_tos_shifted,output:1
```

(a)

```
 cookie=0x0, duration=134.465s, table=0, n_packets=14918, n_bytes=21738159, send_flow_
em priority=0,tcp,in_port=3,dl_src=00:00:00:00:00:09,dl_dst=00:00:00:00:00:01,nw_src=7
.125.168.44,nw_dst=10.0.0.1,tp_dst=48480 actions=set_field:8->nw_tos_shifted,output:1
```

(b)

```
 cookie=0x0, duration=85.801s, table=0, n_packets=14918, n_bytes=21738159, send_flow_r
m priority=0,tcp,in_port=3,dl_src=00:00:00:00:00:09,dl_dst=00:00:00:00:00:01,nw_src=74
125.168.44,nw_dst=10.0.0.1,tp_dst=48480 actions=set_field:8->nw_tos_shifted,output:1
```

(c)

Figura 7.9. Flujos añadidos en: (a) s1, (b) s2 y (c) s3.

Como podemos comprobar, los flujos se han añadido en los *switches* por los que se envía el tráfico *YouTube* de forma satisfactoria.

Capítulo 8

Implementación de controlador con DPI independiente y evaluación de resultados.

En este capítulo vamos a realizar algunas pruebas de rendimiento de nuestro sistema. En primer lugar vamos a explicar la implementación de nuestro controlador de redes SDN como producto independiente en una máquina para su posible introducción en las redes actuales.

Una vez hecho esto pondremos a prueba nuestro sistema ejecutando en varias máquinas virtuales un navegador que solicite la reproducción de videos de *YouTube*.

8.1. Implementación de controlador con DPI independiente.

Una vez tenemos nuestro controlador junto con su respectivo DPI en funcionamiento vamos a pasar a aislar nuestro producto de tal forma que sea capaz de operar correctamente al incluirlo dentro de una red actual. El objetivo por tanto será poder incluir nuestro controlador en la red y que éste inspeccione todo el tráfico que pasa por ella y sea capaz de modificar el *ToS* de los paquetes *YouTube*.

Para realizar este proceso hemos optado por la utilización de máquinas virtuales, las cuales facilitan el proceso de conexión entre equipos sin la necesidad de disponer de ellos físicamente. La herramienta utilizada para trabajar con las máquinas virtuales ha sido *VirtuaBox* [20]. A continuación se muestra cómo quedaría la topología que pasaremos a explicar detalladamente en los siguientes párrafos.

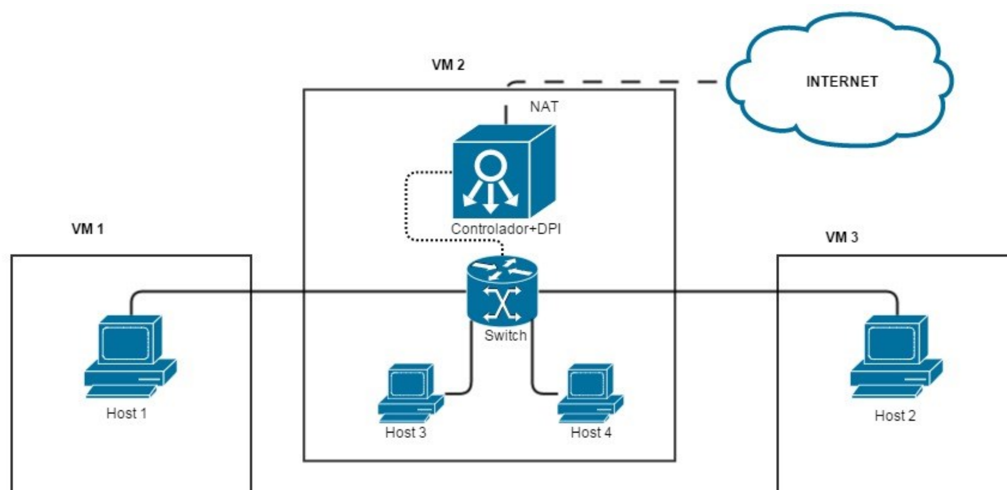


Figura 8.1. Escenario de implementación de controlador con DPI independiente.

· Controlador.

La máquina virtual *VM 2* es la que contiene al controlador. Esta máquina se ha creado a partir del sistema operativo *Ubuntu 12.04* de 32 bits. Se le ha dado una memoria base de 1536 MBytes.

En cuanto a los ajustes de red, dispondrá de 3 adaptadores. El primero será una conexión *NAT* que permitirá conectarse a Internet a través de la red de la que dispone el PC en el que se encuentra la máquina virtual. Las otras dos corresponderán a las redes internas de los dos *hosts* que conectaremos a nuestro controlador, que actuará como un equipo independiente. Es importante permitir el "modo promiscuo" en estos adaptadores, ya que, sino, el controlador no podrá recibir paquetes con una dirección MAC de destino diferente a la suya.

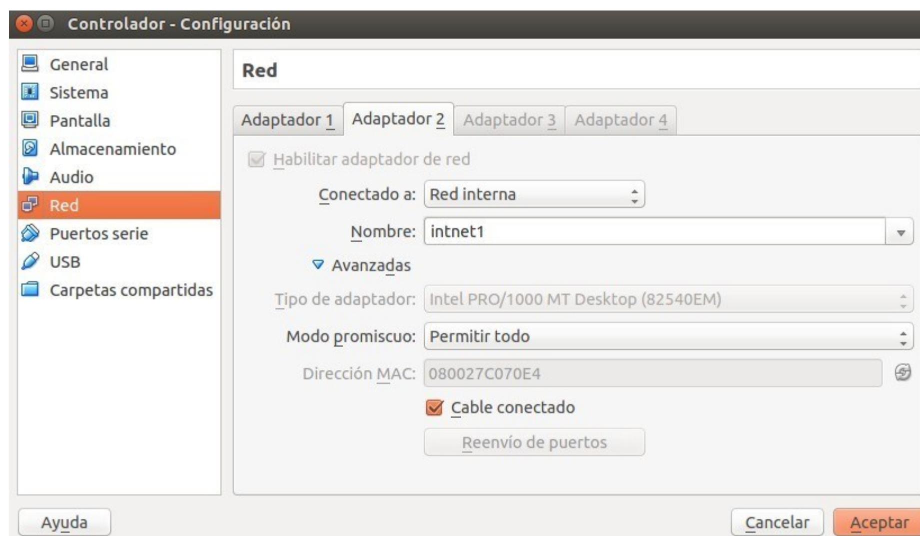


Figura 8.2. Configuración de la interfaz de red para red interna en la VM del controlador.

· *Hosts*.

Se han requerido dos máquinas virtuales para sendos *hosts* creados: *VM 1* y *VM 3*. En ellos se ha instalado *Ubuntu 14.04* y únicamente habilitan un adaptador de red correspondiente a sus redes internas, i.e. para el *host 1* la red "*intnet1*" y para el *host 2* la red "*intnet2*".

· Puesta en marcha de la red.

Una vez configuradas las máquinas virtuales las arrancamos y pasamos a configurar algunos aspectos internos para conectar los *hosts* al *switch* manejado por el controlador.

En la máquina virtual del controlador, deberemos ejecutar la topología deseada. En este caso será un *switch* con dos *hosts* virtuales (creados por *Mininet*) y dos *hosts* reales que se corresponderán a las otras dos máquinas virtuales (creadas por VirtualBox). A su vez, el *switch* indicado tendrá acceso a la red externa mediante el adaptador configurado como *NAT*.

La topología es la siguiente:

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.cli import CLI
from mininet.node import Controller, OVSKernelSwitch, RemoteController, OVSSwitch
from mininet.log import setLogLevel, info, error
from mininet.link import Intf, TCLink
from mininet.util import quietRun

from mininet.nodelib import NAT
from mininet.topolib import TreeNet

import re
import sys

def checkIntf( intf ):
    "Make sure intf exists and is not configured."
    if ( ' %s:' % intf ) not in quietRun( 'ip link show' ):
        error( 'Error:', intf, 'does not exist!\n' )
        exit( 1 )
    ips = re.findall( r'\d+\.\d+\.\d+\.\d+', quietRun( 'ifconfig ' + intf ) )
    if ips:
        error( 'Error:', intf, 'has an IP address,' 'and is probably in use!\n' )
        exit( 1 )
def myNetwork():
       #net = Mininet( controller=RemoteController )
       net = Mininet(topo=None, listenPort=6633, build=False,
                     ipBase='192.168.1.0/24', link=TCLink,)
       info( '*** Adding controller\n' )
       c0=net.addController(name='c0', controller=RemoteController,
                           protocols='OpenFlow13', ='127.0.0.1')

       info( '*** Add switch\n' )
        s1=net.addSwitch('s1', cls=OVSSwitch, mac='00:00:00:00:00:10',
                        protocols='OpenFlow13')
       info( '*** Add eth1 to s1\n' )
       intfName='eth1'
       info( '*** Checking', intfName, '\n' )
       checkIntf( intfName )
       Intf( intfName, node=s1 )
       info( '*** Add eth2 to s1\n' )
       intfName='eth2'
       info( '*** Checking', intfName, '\n' )
       checkIntf( intfName )
       exitIntf( intfName, node=s1 )
       info( '*** Add hosts\n')
       h1 = net.addHost( 'h1', mac='00:00:00:00:00:01', ip='192.168.1.1/24',
                        defaultRoute='via 192.168.1.3' )
       h2 = net.addHost( 'h2', mac='00:00:00:00:00:02', ip='192.168.1.2/24',
                        defaultRoute='via 192.168.1.3' )
        info('*** Add NAT\n')
        net.addNAT().configDefault()
       info( '*** Add links\n')
       #net.addLink( h1, s1 )
       #net.addLink( h2, s1 )
       net.addLink( s1, h1 )
       net.addLink( s1, h2 )
       info( '*** Starting network\n')
```

```
        net.build()
        net.start()
        info( '*** Starting controllers\n')
        for controller in net.controllers:
                controller.start()
        net.get('s1').start([c0])
         # Open Mininet Command Line Interface
        CLI(net)
        # Teardown and cleanup
        net.stop()
if __name__ == '__main__':
        setLogLevel( 'info' )
        myNetwork()
```

Una vez instalados *OpenDayLight* y *Mininet* en la máquina virtual del controlador abrimos dos terminales. En la primera ponemos en marcha el controlador:

```
$ ./run.sh
```

En la segunda terminal arrancamos la topología con *Mininet*:

```
$ sudo python topo--sw-2ports.py
```

Lo primero que vamos a configurar en la topología será el acceso a Internet. Para ello tenemos que configurar la dirección del servidor DNS. El fichero *resolv.conf* quedaría de la siguiente manera configurado para una red doméstica:

```
# Dynamic resolv.conf(5) file for glibc resolver(3)
generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE
OVERWRITTEN
nameserver 127.0.0.1
nameserver 8.8.8.8
search home
```

Una vez hecho esto, pasamos a la configuración de los *hosts*. Arrancamos las otras dos máquinas virtuales que harán de *hosts*. Como vemos en la topología, las direcciones de los dos *hosts* creados directamente en la misma son la *192.168.1.1* y *192.168.1.2*. Además, al configurar *NAT* en la topología, se crea una interfaz con dirección *192.168.1.3* por la que serán reenviados todos los paquetes que quieran acceder a Internet. Por lo tanto, al primer *host* le asignaremos la dirección *192.168.1.4* y al segundo la *192.168.1.5*.

La siguiente línea muestra el ejemplo de configuración de IP para el primer *host*:

```
$ ifconfig eth0 192.168.1.4
```

El siguiente paso será indicar por dónde debe reenviar el tráfico, es decir, que lo envíe por defecto a la interfaz de entrada/salida de la red SDN que en este caso es la *192.168.1.3*:

```
$ sudo route add default 192.168.1.3
```

Por último, indicaremos en este caso también el servidor DNS de la misma forma que lo hicimos con la máquina virtual del controlador.

Ya tenemos configurado el escenario para poder realizar cualquier prueba. Ahora sólo queda iniciar los navegadores y comprobar el funcionamiento.

8.2. Evaluación de resultados.

Una vez configurado el sistema ejecutamos en cada uno de los cuatro *hosts* (dos dentro de la propia topología de *Mininet* más los dos incluidos en sendas máquinas virtuales) abrimos un navegador y reproducimos un vídeo de *YouTube* como se muestra en la *Figura 8.2*.

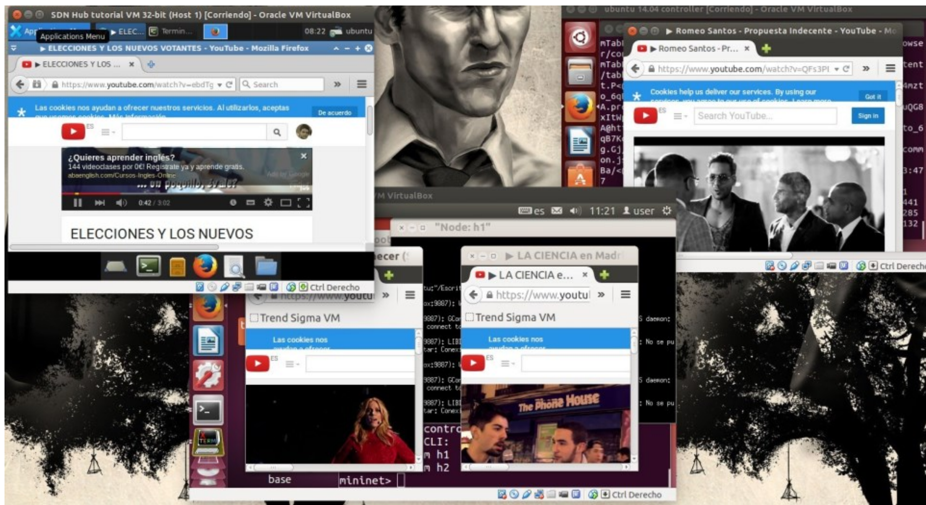


Figura 8.3. Reproducción de vídeo desde cuatro hosts diferentes.

Como vemos, los videos se están reproduciendo al mismo tiempo por lo que debería haber añadido cuatro flujos al menos referentes a dichos flujos. En efecto, al comprobar los flujos instalados, se corrobora que se han instalado estos cuatro flujos con el *ToS* cambiado para las direcciones IP de los servidores de vídeo. Además se incluyen otros flujos que no son propiamente del vídeo en sí pero que tienen que ver con el servidor de vídeo. En las

siguientes figuras mostramos cada uno de los cuatro flujos instalados que presumiblemente transportan el vídeo por la red:

```
 cookie=0x0, duration=1934.096s, table=0, n_packets=14788, n_
bytes=22368104, priority=0,tcp,in_port=3,dl_src=d6:ce:c5:e8:4
8:74,dl_dst=00:00:00:00:00:01,nw_src=173.194.15.179,nw_dst=19
2.168.1.1,tp_dst=47961 actions=mod_nw_tos:32,output:5
```

```
 cookie=0x0, duration=40.29s, table=0, n_packets=12300, n_bytes
=18479798, priority=0,tcp,in_port=3,dl_src=d6:ce:c5:e8:48:74,dl
_dst=00:00:00:00:00:02,nw_src=173.194.9.233,nw_dst=192.168.1.2,
tp_dst=59726 actions=mod_nw_tos:32,output:4
```

```
 cookie=0x0, duration=276.62s, table=0, n_packets=13877, n_by
tes=20848624, priority=0,tcp,in_port=3,dl_src=d6:ce:c5:e8:48:
74,dl_dst=08:00:27:5c:93:8c,nw_src=173.194.9.145,nw_dst=192.1
68.1.4,tp_dst=46868 actions=mod_nw_tos:32,output:1
```

```
 cookie=0x0, duration=279.344s, table=0, n_packets=2118, n_by
tes=3169482, priority=0,tcp,in_port=3,dl_src=d6:ce:c5:e8:48:7
4,dl_dst=08:00:27:8c:d9:3d,nw_src=173.194.9.134,nw_dst=192.16
8.1.5,tp_dst=49070 actions=mod_nw_tos:32,output:2
```

Figura 8.4. Flujos de vídeos de YouTube correspondientes a cada host.

El sistema, pese a estar ejecutado en un entorno bastante limitado ya que se dispone únicamente de un portátil con tres máquinas virtuales corriendo en él, ha respondido ante cuatro equipos diferentes solicitando vídeos de *YouTube*. Además de los flujos referentes a *YouTube* se han añadido multitud de flujos de otro tipo al mismo tiempo. Con esta prueba se trata de acercar el concepto de integrar nuestro DPI en una red real de forma flexible. Para realizar pruebas de carga habría que tener acceso a una red donde haya carga real como, por ejemplo, la red de la UGR. No obstante estas pruebas están más allá del ámbito de este proyecto.

Capítulo 9

Conclusiones y vías futuras

En este capítulo se recogerán las conclusiones una vez finalizado el presente Trabajo de Fin de Grado en todo lo referente al mismo así como una serie de vías futuras posibles.

Por lo tanto se expondrán algunas cuestiones relevantes surgidas a raíz de la elaboración así como las contribuciones aportadas al campo en que hemos trabajado. También se comentarán las posibilidades que se presentan para el desarrollo de nuevas líneas futuras a partir del trabajo realizado en este proyecto.

Por último, se ofrecerá una valoración personal del autor de este proyecto para dar por finalizado el mismo.

9.1. Conclusiones.

En el presente Trabajo de Fin de Grado se ha diseñado un *Deep Packet Inspector* haciendo uso de *Mininet* y gracias a otra serie de herramientas que se han ido acoplando en diferentes partes del proyecto. De esta forma se ha conseguido detectar, entre otros, el tráfico generado por YouTube dentro de una red emulada SDN. Una vez realizada esta detección hemos podido modificar dicho tráfico de tal forma que se añadiese una acción a la tabla de flujo capaz de modificar el *ToS* de los paquetes y propiciar así la posibilidad de ofrecer una Calidad de Servicio (QoS) en la red. Además, se ha experimentado con la implementación para comprobar el correcto funcionamiento del sistema desarrollado.

Las principales contribuciones que nuestro proyecto ofrece son:

- Ofrecer una gran recopilación de información en lo referente a este tipo de redes emergentes como son las SDN. Con el trabajo realizado se ofrece una idea de las posibilidades que estas redes ofrecen tanto a las empresas como a los usuarios en cuanto a diferentes aspectos como son la gestión de red, monitorización o calidad de servicio.
- Se ha experimentado con *Mininet* y *OpenDayLight* de manera que se aporta una gran fuente de información para comenzar a trabajar con estas herramientas. Esto es una ventaja para los desarrolladores que deseen introducirse en estas nuevas redes ya que la investigación en este campo aún es incipiente.
- El DPI desarrollado permite la detección precisa de los paquetes del servidor de vídeo YouTube. Para ello se ha realizado previamente un análisis exhaustivo de paquetes DNS necesario para obtener la dirección IP del servidor, a parte de un procedimiento posterior para detectar dicha IP tras la llegada de paquetes. Lo que nos permite este DPI por tanto será dar los medios (a través del campo *ToS*) para que la red proporcione

calidad de servicio a ciertas aplicaciones o servicios, como en este caso el servicio de vídeo de *YouTube*.

- En cuanto al análisis de paquetes DNS, una aportación importante es el desarrollo de un analizador de este tipo de paquetes, el cual no estaba implementado hasta el momento en *OpenDayLight*.
- Otra aportación importante es la posibilidad de implementar a partir del DPI la detección de una gran multitud de tráfico referente a otras aplicaciones o servicios, ya que es un DPI modular y flexible en su desarrollo.

9.2. Vías futuras.

9.2.1. Detección de otras aplicaciones relevantes.

Pese a que los objetivos propuestos para este proyecto se han conseguido, a partir del mismo se pueden realizar algunas mejoras que podrían ser de gran utilidad para la consecución de un producto más completo y competitivo en el mercado.

Una de estas mejoras adicionales podría ser implementar la detección de otros tipos de servicios o aplicaciones dentro de nuestro DPI. Además del tráfico *YouTube*, del que ya se comentó su relevancia, existen otros tipos de tráfico muy solicitados por los usuarios y que necesitan de cierta *QoS*. Un ejemplo puede ser el *streaming* ofrecido por algunas compañías en el que se ofrece televisión por Internet, como son *Yomvi*, *TotalChannel* etcétera.

Por otro lado también podría apostarse por la detección de tráfico referente a redes sociales como *Facebook*, *Instagram*, *Twitter* etcétera. Dada la importancia de las redes sociales en los últimos años podría ser interesante poder detectar el tráfico ligado a ellas.

9.2.2. Desarrollo en *MD-SAL*.

Como se comentó en el transcurso de esta memoria, la implementación más relevante del proyecto se llevó a cabo mediante *AD-SAL*. No obstante, el modelo *MD-SAL* se postula como el referente para las próximas implementaciones en *OpenDayLight* y se está desarrollando actualmente. Es por ello que sería de gran interés mejorar la adaptación de nuestro DPI al nuevo modelo. Bien es cierto que ya se ha conseguido implementar la detección de tráfico sencillo utilizando *MD-SAL*, pero los problemas encontrados (*bugs*) en las versiones actuales de *OpenDayLight* hacen que aún se necesite la mejora en cuanto a la adaptación de nuestro producto con el nuevo modelo para conseguir un funcionamiento óptimo.

9.2.3. Implementación en entornos reales.

Debido a las limitaciones encontradas en cuanto a dispositivos utilizados y las posibilidades para introducir nuestro sistema en alguna red en explotación, no se han podido realizar pruebas en entornos reales. La posibilidad de implementar el sistema en un entorno real posibilita la obtención de una serie de información muy interesante para conocer las prestaciones que podrían ofrecerse, por lo que sería un posible reto para el futuro.

9.3. Valoración personal.

Para finalizar, vamos a realizar una serie de apreciaciones y reflexiones personales acerca del desarrollo de este Trabajo de Fin de Grado.

Este trabajo se postula como el punto final para la consecución de las aptitudes necesarias que acreditan la obtención del título como Graduado en Ingeniería de Tecnologías de Telecomunicación. Por ello, comprende una serie de capacidades que hay que poner en práctica para demostrar todo lo aprendido durante el transcurso del Grado. El objetivo final es poner de manifiesto la comprensión de conocimientos referentes a un área de estudio, aplicar dichos conocimientos al propio trabajo, obtener la capacidad de analizar, reunir e interpretar diferentes datos relevantes, transmitir información, ideas, problemas y soluciones a un público (tanto si es especializado en ese tema como si no) y, como punto final, la adquisición de las competencias de aprendizaje que permitan emprender posteriores estudios con un alto grado de autonomía.

Todos estos puntos se intentan plasmar en este trabajo con la ayuda de esta memoria en la que se aportan una gran cantidad de conocimientos adquiridos acerca del tema tratado, como son las redes SDN en general y la implementación de un DPI en particular. Se ha intentado ofrecer una visión global de lo que SDN puede aportar a las redes futuras y, al mismo tiempo, se ha propuesto una solución para implementar un DPI a partir del análisis de información y la comparativa de las diferentes herramientas ofrecidas en el mercado, de tal manera que hemos escogido lo que mejor se adaptaba a los requisitos esenciales de nuestro proyecto.

Además, estos análisis y comparativas se han intentado justificar de modo que el lector pudiese comprender el porqué de las diferentes decisiones tomadas a lo largo del desarrollo.

Por otro lado, con este proyecto nos acercamos bastante a lo que podría ser un escenario de trabajo real debido a que se nos solicita en primera instancia la obtención de un producto, en este caso un DPI para redes SDN, y se marcan unos plazos para conseguirlo. Dentro del objetivo final se marcan ciertas etapas en las que dividir el trabajo para conseguir cumplir con los plazos y presentar el producto solicitado. Finalizado éste, se evalúa si se ha cumplido con lo previamente establecido y en qué medida se han cumplido los requisitos marcados.

Centrándonos ahora en la valoración sobre este tipo de redes, podemos decir que sus perspectivas son bastante alentadoras, ya que la idea en la que se fundamentan abre muchas puertas de cara a ofrecer calidad de servicio al consumidor. Además, según la información recogida, son muchas las empresas que han depositado sus esfuerzos en investigar estas redes para un futuro cercano.

Como punto final, cabe destacar la satisfacción que supone tanto a nivel personal como profesional la consecución de todo un proyecto por parte del autor. Este trabajo se presentaba como un gran reto debido a su complejidad y dimensiones, ya que se trata de un proyecto de investigación en el que, a día de hoy, existen pocas fuentes de información. Por ello se ha realizado un enorme trabajo para la obtención de los conocimientos previos a abordar la implementación del DPI. Finalmente hemos conseguido alcanzar los objetivos propuestos e incluso se han logrado algunos otros de forma adicional.

A. Manual de instalación de *Mininet*.

Para la instalación de *Mininet* existen cuatro opciones diferentes. En este anexo se explicarán cada una de ellas, las cuales pueden ser consultadas en la propia página de *Mininet* [12].

· Opción 1: Instalación mediante máquinas virtuales.

Para ello deberemos instalar la máquina virtual de *Mininet* [49] y una herramienta de virtualización de sistemas. En este proyecto se ha trabajado con *VirtualBox* ya que es libre y su comportamiento en este entorno es óptimo.

· Opción 2: Instalación nativa desde código.

En este caso únicamente tendríamos que descargar el repositorio de *GitHub* correspondiente a la herramienta *Mininet*:

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

Una vez hecho esto instalamos *Mininet*. Primero entramos en el directorio *mininet/util* y seguidamente deberemos ejecutar el instalador. La instalación dependerá de lo que deseemos instalar, ya que puede indicarse si se desea la instalación de *Open vSwitch*. En nuestro caso instalamos todo, así que ejecutamos:

```
$ install.sh -a
```

· Opción 3: Instalación mediante paquetes.

Es recomendable en caso de utilizar una versión de Ubuntu reciente. El problema es que la versión de *Mininet* que nos proporcionará será algo antigua. Ejecutamos:

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

El siguiente paso será instalar el paquete *Mininet* correspondiente a la versión de *Ubuntu* utilizada. Las opciones son:

- Ubuntu 14.10 y Ubuntu 14.04:

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

- Ubuntu 12.04:

```
$ sudo apt-get install mininet/precise-backports
```

· Opción 4: Actualizando una versión existente de *Mininet*.

Podemos ejecutar lo siguiente:

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

B. Instalación de OpenDayLight y compilación de nuevos paquetes.

Para instalar *OpenDayLight* tenemos dos opciones. La primera es descargarnos la versión *Helium-SR3* [50] disponible en el apartado de descargas de su página *web* y la segunda es instalar la distribución base [40].

Instalación de *OpenDayLight*

· Instalación de *OpenDayLight Helium*.

Tras descargar el paquete en la página de *OpenDayLight* tendremos que descomprimirlo. Esta distribución no viene con las funciones habilitadas por defecto, pero pueden habilitarse fácilmente. Para arrancar *Karaf* ejecutamos lo siguiente:

```
$ cd distribution-karaf-0.2.0-Helium
$ ./bin/karaf
```

Si deseamos instalar algunos componentes basta con ejecutar dentro de *Karaf*:

```
karaf> install <feature1-name> <feature2-name> … <featureN-name>
```

· Instalación de distribución base.

Para la distribución base basta con elegir una entre las que ofrecen en la página *web* [40].

Una vez descargado basta con descomprimir el archivo, el cual creará la carpeta *opendaylight*. Dentro de esta se encuentra el ejecutable para arrancar el controlador:

```
$ ./run.sh
```

Compilación de nuevos paquetes.

Este punto se va a explicar esencialmente para la distribución base, ya que es la que permite añadir y compilar nuevos paquetes de manera más sencilla. No obstante, el procedimiento se asemeja bastante en el fundamento para las demás distribuciones.

Los conceptos de *Maven* [51] y *OSGi* [52] son clave para la compilación de paquetes en *OpenDayLight*. El controlador, en última instancia, es un conjunto de *bundles OSGi*, clases java, recursos y/o manifiestos (*pom.xml*) que se apoyan sobre el *framework OSGi*. Por lo tanto, al desarrollar nuestro proyecto lo que estamos haciendo es desarrollar un *bundle* que acoplaremos al controlador.

· Estructura de un proyecto *maven*.

La estructura de *maven*, en la que se basa *OpenDayLight,* será la que adoptemos para introducir nuestro proyecto en el controlador. En concreto, nuestra estructura es la siguiente:

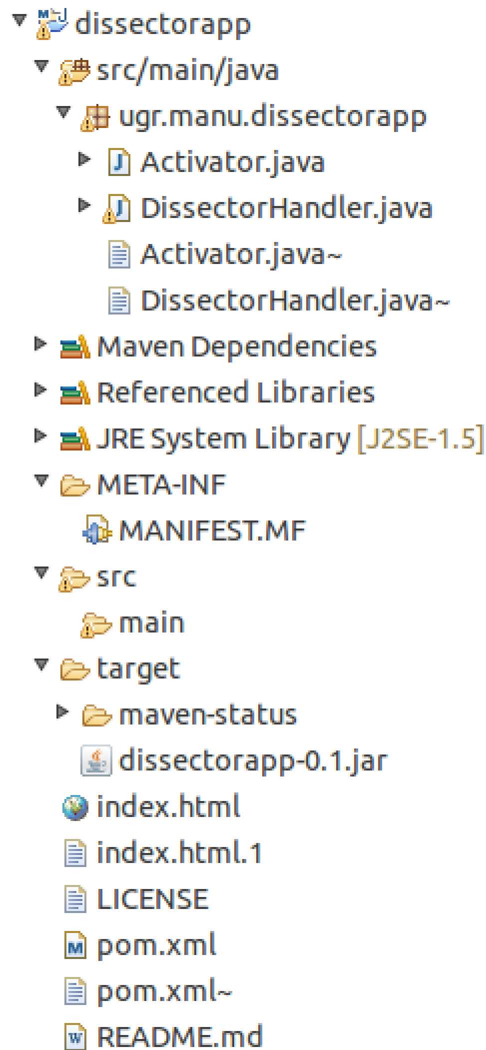


Figura B.1. Estructura de nuestro proyecto con Maven.

La carpeta *dissectorapp* es la que contiene todo lo referente a nuestro proyecto. Dentro de la misma deben coexistir tres archivos fundamentales:

- ***pom.xml***. Este archivo escrito en *xml* es el que contiene toda la información referente a las dependencias, nombres de paquetes para situarlos en nuestro proyecto e indispensable para su compilación.

- ***Activator.java***. Es el archivo que se invoca cada vez que el *bundle* que hemos creado se necesite. En él están contenidos los métodos que pueden ser invocados así como las interfaces que implementaremos en el archivo *DissectorHandler.java*.
- ***DissectorHandler.java***. Este es el archivo que contiene nuestra implementación así como las distintas interfaces de las que hará uso de *Activator.java*.
  El resto de archivos serán generados por *Maven* directamente al compilar o cualquier otro fichero que implementemos nosotros mismos.

Para compilar un proyecto basta con situarnos en el directorio donde se encuentra nuestro archivo *pom.xml* y ejecutar:

```
$ sudo mvn clean install
```

De esta forma se actualizará cualquier cambio realizado y se generará un *.jar* de nuestro *bundle* en el directorio *target*. Si copiamos este *bundle* en la carpeta *plugins* dentro del directorio *opendaylight* nuestro paquete se cargará en el controlador.

También podemos optar por ligar el *bundle* una vez arrancado el controlador. Para ello, primero habrá que instalar dicho *bundle* indicando la ruta donde se encuentra y, una vez descubramos el número de *bundle* asociado al mismo, arrancarlo:

```
osgi> install file:/<ruta-bundle>
osgi> start <nº_bundle>
```

Para activar los *logs* del paquete instalado basta con ejecutar dentro del controlador:

```
osgi> setLogLevel <ruta_bundle> trace
```

C. Manual de usuario de la aplicación.

En este anexo vamos a explicar cómo arrancar nuestro sistema de manera sencilla. En primer lugar tendremos que descargar el material del proyecto y descomprimirlo. Además, mostraremos el código de nuestro proyecto.

Manual de usuario de la aplicación

El material consta de un directorio llamado *distribution_base* dentro del cual se encuentran tanto la distribución base de *OpenDayLight* como nuestro proyecto, denominado *dissectorapp* y algunas topologías creadas con *scripts* de *python*.

En primer lugar abrimos una terminal y nos situamos en el directorio *dissectorapp*. Dentro del mismo está el fichero *pom.xml* de nuestro proyecto, así que lo compilamos:

```
$ sudo mvn clean install
```

Para arrancar el controlador nos vamos al directorio *opendaylight* y ejecutamos:

```
$ ./run.sh
```

Ya solo falta poner en marcha una topología. En nuestro caso la más utilizada ha sido una topología basada en un *switch* y tres *hosts*. Abrimos un terminal nuevo y ejecutamos:

```
$ sudo mn --controller=remote --topo single,3 --mac --arp --switch
ovsk,protocols=OpenFlow10 --nat
```

A partir de aquí ya podemos trabajar con diferentes pruebas en nuestra red SDN con un DPI implementado en el controlador remoto.

D. Complicaciones y errores encontrados.

En este anexo indicaremos una serie de errores que se nos han presentado durante la elaboración del proyecto y que resaltamos para informar tanto del error (*bug*) en sí como de su posible resolución si es que se ha encontrado.

· Error al arrancar el controlador *OpenDayLight*.

En algunas ocasiones, se detectaron problemas al arrancar el controlador. Un error muy común se debe a que, a la hora de ejecutar el comando que activa el controlador, no lo hacemos como *superusuario*. Esto provoca que el controlador no se inicie correctamente.

· Error con el paquete de *MD-SAL* de *SDN HUB TURORIAL*.

El problema con este paquete es que, al arrancar el controlador y añadir un flujo, este no se refleja en las tablas hasta haber reiniciado la topología creada en *Mininet*. Además, sólo añade los flujos si el protocolo utilizado es la versión 1.0 de OpenFlow. Para este problema no hemos encontrado solución alguna. No obstante, parece ser causado por la comunicación asíncrona que utiliza este modelo, que parece quedar bloqueado en algún punto.

· Problemas al iniciar NAT en una topología en *Mininet*.

Cuando iniciamos NAT sin el cable de red debemos indicar antes el servidor DNS al que preguntaremos las direcciones. Para ello debemos editar el fichero "*/etc/resolv.conf*" y cambiar el nameserver por 8.8.8.8 en caso de estar conectados a una red doméstica y utilizar el DNS de *Google*, o la dirección del servidor DNS de la red pública (150.214.35.10 en caso de la UGR).

En caso de no hacerlo tendremos acceso a Internet pero no podremos llevar a cabo la resolución de nombres y, por lo tanto, no podremos hacer búsquedas desde un navegador.

· Problemas al instalar nuevos paquetes en la distribución *Helium*.

En el transcurso de la realización de este proyecto se intentó en primera instancia crear nuestros propios *bundles* a partir de la distribución *Helium*. Sin embargo, debido a la complejidad de ésta, nos fue imposible ponerlos en marcha. Esto es debido a la cantidad de dependencias y de rutas a las que accede desde diversos *pom.xml*. Es por ello que optamos por la distribución base.

· Problemas al programar en Python.

Un aspecto que debemos tener presente al programar en *python* es que los espacios y tabulaciones son muy importantes, ya que un mal uso de los mismos puede provocar que no compile nuestro proyecto. Por lo tanto éste es un aspecto que debemos cuidar.

· Problemas para detectar flujos en el controlador.

Un aspecto que hay que tener en cuenta para la detección de flujos en el controlador es que en primera instancia debemos indicar que el paquete es de tipo Ethernet. Eso se realiza en nuestro caso añadiendo la siguiente línea:

```
match.setField(MatchType.DL_TYPE, (short) 0x0800);  // IPv4 ethertype
```

En caso de no indicarlo, no será capaz de identificar los paquetes con ningún tipo de flujo y no se realizará ninguna acción.

E. Blog en WordPress.

Como uno de los últimos anexos no se nos puede escapar la realización de un *blog* por parte del autor de este proyecto y en colaboración con su compañero Cristian Alfonso Prieto Sánchez acerca de *OpenDayLight, Mininet* y todo lo que estos dos conceptos engloban. El *blog* se ha creado en *WordPress* y su contenido trata de ilustrar los conocimientos adquiridos durante el trabajo realizado con estas herramientas a fin de facilitar el trabajo a cualquier desarrollador que comience su aventura en este entorno de emulación de redes SDN.

El *blog* se ha titulado *"Aprendiendo OpenDayLight"* y su dirección *web es* https://aprendiendoodl.wordpress.com/ . En él se tratan diversos temas con relación a sendos proyectos y en él se explica cómo ponerse en marcha y algunos tutoriales para el desarrollo del controlador. En la siguiente figura podemos ver como se muestra nuestro *blog* al entrar.

Figura E.1. Portada del blog "Aprendiendo OpenDayLight".

La relevancia en la red de nuestro *blog* ha sido considerable, situándose decimoquinta en el navegador *google* al buscar la palabra "*opendaylight*" e incluso entre las diez primeras buscando "*mininet*". Muchos de nuestros *post* aparecen entre los primeros al buscar "*aprendiendo opendaylight*" o "*aprendiendo mininet*".

Atendiendo a las estadísticas que facilita *WordPress*, desde la creación de nuestro *blog* hemos tenido 1891 visitas de 426 visitantes diferentes a día 21 de junio de 2015 como muestra la figura siguiente.

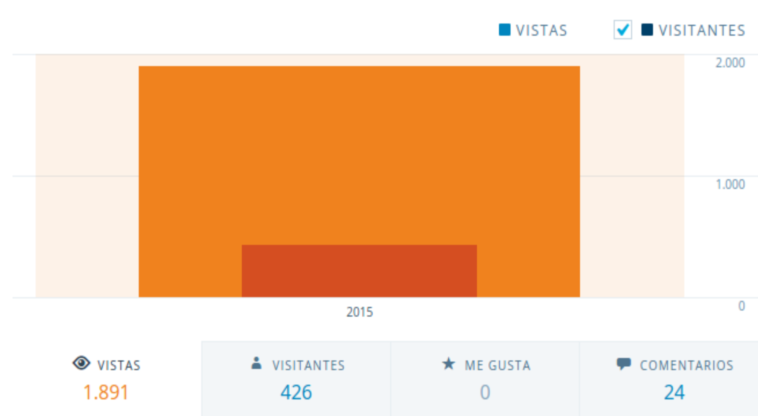


Figura E.2. Visitas y visitantes al blog aprendiendoodl.wordpress.com.

Esto refleja el gran trabajo realizado y su impacto en la comunidad que trabaja con *OpenDayLight* y *Mininet*.

Si nos centramos en los países desde donde se realizan las visitas podemos comprobar que en España es donde más interés genera nuestro *blog* como se muestra en la siguiente figura. Una de las principales causas es el idioma que se utiliza (español), ya que la mayor parte de la información encontrada en Internet está disponible en inglés. En la figura se muestran los países desde donde se han realizado más visitas, pero a esta lista se pueden añadir otros como Suecia, Argentina, Brasil o India.

Figura E.3. Países con más visitas al blog.

Este *blog* nos llena de satisfacción, ya que hemos podido comprobar que hay mucha gente interesada en nuestro trabajo y que realizamos una gran aportación a mucha gente, lo cual nos anima a seguir trabajando en su desarrollo.

F. Comparativa de controladores de red.

Existen varias alternativas en el mercado para actuar como controlador en una red SDN. En este anexo se pretende mostrar las más relevantes y justificar la elección de OpenDayLight como el controlador con el que hemos decidido trabajar. Para ello analizaremos controladores tanto de código abierto como de código privado.

Controladores *open source*.

A continuación se muestra una breve descripción de algunos de los controladores de código abierto más relevantes:

· *NOX*.

Uno de los primeros controladores de redes SDN fue *NOX* [53]: la mayoría de las aplicaciones de SDN y OpenFlow estaban implementadas con este controlador. Incluso Google utilizó este *software* para construir su propio controlador OpenFlow distribuido, denominado *ONIX*. No obstante, *NOX* dejó de utilizarse en 2010. A partir de entonces dejó de prestarse apoyo por parte de la comunidad y no hubo cambios importantes. En la siguiente imagen se muestra la evolución del soporte de la comunidad.

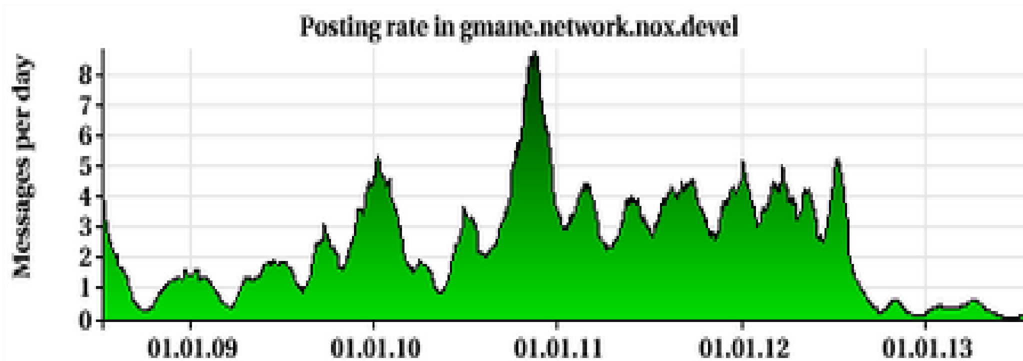


Figura F.1. Mensajes enviados por día comunidad a la NOX.

NOX está escrito en C++ y su entorno de desarrollo es complejo. Además, no soporta topologías con bucles. Como solución se puede activar STP (*Spanning Tree Protocol*), pero este enfoque es claramente contradictorio a la esencia de SDN. En cuanto a la documentación, es prácticamente inexistente. *NOX* suministra una interfaz gráfica de usuario basada en Python, la cual se muestra en la siguiente figura.

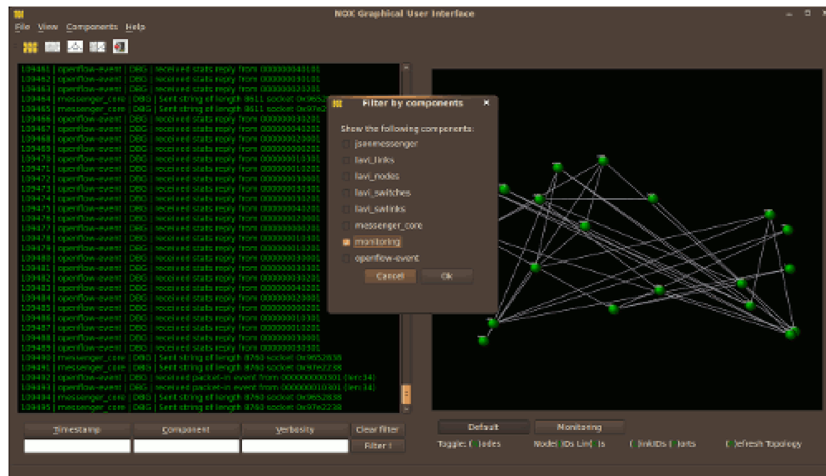


Figura F.2. Interfaz gráfica de NOX

En resumen, *NOX* no es recomendable si nuestro fin es el de desarrollar a partir de él, sobre todo teniendo en cuenta las variedad de alternativas en el mercado actual.

· *POX*.

POX [54] nace a partir de *NOX*. Ambos tienen el soporte de la misma organización. Tiene una lista de correo muy activa, ya que casi todos los recién llegados a OpenFlow comienzan con este controlador. *POX* ofrece una *API* web limitada y una colección de tamaño moderado de los manuales en su *wiki*. Está escrito en Python y ofrece una *API* de Python para programadores en este lenguaje. Debido a que Python es un lenguaje interpretado, *POX* reduce el tiempo empleado en el desarrollo y despliegue, sobre todo en comparación con C++ basado en *NOX*. Por otra parte, además de apoyar la *GUI NOX*, *POX* también proporciona una interfaz gráfica de usuario basada en *web* como se aprecia en la siguiente imagen.

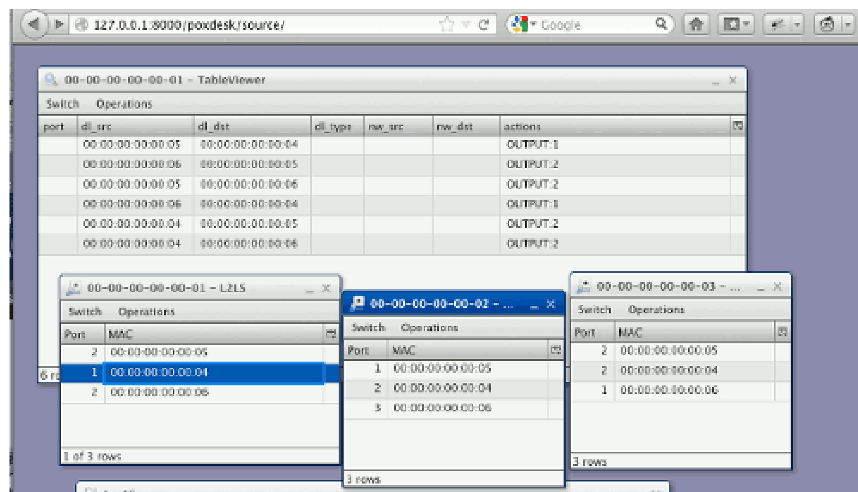


Figura F.3. Interfaz gráfica de POX.

· ***Helios***.

Helios [55] es una extensión basada en C de los controladores OpenFlow sobre *NEC (Network Enabled Capabilities)*, orientada a investigadores. También facilita la realización de experimentos integrados.

· ***Maestro***.

Maestro [56] es una extensión basada en Java del controlador OpenFlow desarrollada por la universidad *Rice*. Tiene soporte para multihebra y está dirigida a investigadores. Es de código abierto y utiliza Java. No está limitado a ser un controlador de OpenFlow.

· ***Beacon*.**

Este controlador [57] es fácil de implementar; clonamos el repositorio, importamos el proyecto en Eclipse y lo ejecutamos. Si acabamos de modificar alguna parte del código simplemente volvemos a ejecutarlo. Incluso ofrece tutoriales en su *web*. Además, cuenta con un foro, donde David Erickson, el autor de *Beacon*, responde activamente. Por otro lado, el código se implementa de forma clara.

Este controlador fue el primero en incluir una interfaz de usuario de una gran calidad y está escrito en Java, lo cual supone un cambio con respecto a los controladores anteriores.

Como hemos comentado anteriormente, *Beacon* está estrechamente ligado a Eclipse tal que incluso se admite *Maven* [58]. El gran inconveniente es que *Beacon* no permite manejar topologías con bucles o dispositivos accesibles por múltiples puertos del *switch*.

En resumen, *Beacon* ofrece una base de código compacto y expresivo además de una buena interfaz de usuario y un gran apoyo de la comunidad. No obstante, Eclipse y el soporte de topología en estrella únicamente (es decir, topologías sin bucles) son condiciones muy restrictivas.

· ***Floodlight*.**

El código base de *FloodLight* [59] es prácticamente el mismo que el de *Beacon*. Este controlador si apoya las topologías con bucles, además de dominios no OpenFlow y dispositivos accesibles por múltiples puertos del *switch*, lo cual mejora notablemente a *Beacon*. La mayoría de los desarrolladores de OpenFlow trabajan en redes *BigSwitch* y participan directamente en las listas de correo, por lo que el apoyo de la comunidad es también muy notable.

Por otra parte, este controlador implementa un sistema de módulos que evita que haya que parar la ejecución para incluir una funcionalidad nueva. *Floodlight* expone casi toda su funcionalidad a través de una *API REST* y existen muchas utilidades prácticas para tareas comunes tales como rutas estáticas y ruta de extremo a extremo. Tiene una interfaz de usuario basada en *web* y una interfaz gráfica de usuario basada en Java, llamada *Avior*. No obstante, esta interfaz tiene algunos errores. *Floodlight* también se puede ejecutar como una

red *backend* para *OpenStack* utilizando el *plugin* Quantum. Por último decir que es uno de los proyectos de controlador más documentados del mercado. En la siguiente figura mostramos la interfaz de *Floodlight* que facilita su utilización.

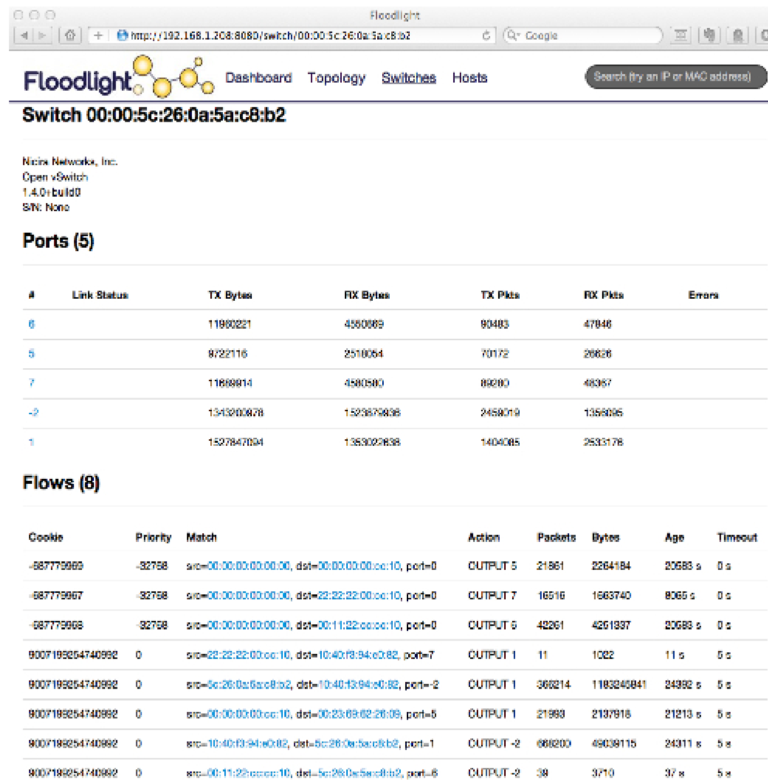


Figura F.4. Interfaz gráfica de Floodlight.

· *OpenDayLight*.

OpenDayLight [11] es un proyecto de Linux apoyado por la industria. Su código base, escrito en Java, es más claro en comparación con *Floodlight*. Hay algunas características en *OpenDayLight* que no están presentes en *Floodlight* como, por ejemplo, el soporte de múltiples islas con bucles. En las primeras etapas del proyecto se optó por favorecer el código aportado por los desarrolladores de Cisco basado en redes *BigSwitch* presentes en *Floodlight*. En la lista de correo se puede observar fácilmente el dominio de desarrolladores de *Cisco* en la misma, en la cual se evidencia un gran apoyo de la comunidad a este proyecto.

Por otro lado, *OpenDayLight* sigue un modelo de controlador que además de OpenFlow, puede introducir otros protocolos. Esta faceta diferencia significativamente a *OpenDayLight* de otros controladores y le permite utilizar *switches* que emplean los protocolos de control de propiedad no OpenFlow. En la siguiente figura se muestra el apoyo recibido por muchas de las empresas importantes en el desarrollo de OpenFlow a este controlador:

Figura F.5. Lista de empresas que ofrecen apoyo al controlador OpenDayLight.

En cuanto a la información referente a este controlador, hay que destacar la extensa *wiki* [60] de la que se nutre, en la cual podemos encontrar multitud de referencias a diferentes aspectos tanto de implementación como de arquitecturas o características del controlador.

OpenDayLight también ofrece una interfaz vía *web* bastante depurada y con la que se puede trabajar fácilmente con el controlador. En la siguiente figura se muestra la interfaz.

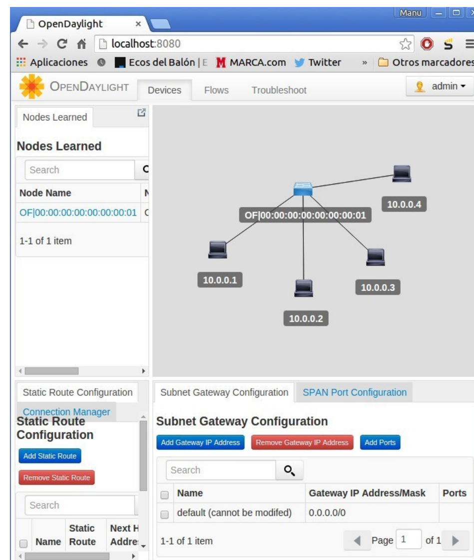


Figura F.6. Interfaz web de OpenDayLight.

En la siguiente tabla se muestra una comparativa a modo de resumen entre los controladores más destacados de los vistos anteriormente:

Característica	***NOX***	***POX***	***Beacon***	***Floodlight***	***OpenDaylight***
Lenguaje de programación	C++	Python	Java	Java	Java
Lenguaje soportado por el controlador	C, C++, Python	Python	Java	Java, Python	Java
¿Está siendo desarrollado?	No	Si	mantenido	Si	Si
¿Tiene comunidad activa?	No	Si	Si	Si	Si
¿Fácil de instalar?	No	Si	Si	Si	Si
¿Fácil de programar?	No	SI	Si	Si	Si
¿Está documentado?	No	Si	Si	Si	Si
***¿REST API*?**	No	Si (limitado)	Si	Si	Si
¿Interfaz?	Python+QT 4	Python+QT4, Web	Web	Java, Web	Web
¿*Hosts* con múltiples accesos?	No	No	No	Si	Si
¿Topologías con bucles?	No	Si (via STP)	No	Si	Si
¿Conexiones no-OpenFlow?	No	No	No	Si	Si
Soporte OpenFlow	OF v1.0	OF v1.0	OF v1.0	OF v1.0	OF v1.0 – OF v1.3
¿Conexiones no-OpenFlow con bucles?	No	No	No	No	Si
¿Capa de abstracción sobre protocolos *south-bound*?	No	No	No	No	Si
Soporta *OpenStack Quantum*?	No	No	No	Si	Si

Tabla F.1. Comparativa de controladores de redes SDN.

Controladores comerciales.

Existen también un gran número de controladores SDN comerciales disponibles en el mercado. Algunos de ellos están basados en *software* mientras que otros se basan en el *hardware*.

· *Big Switch Big Cloud Fabric.*

Este controlador [61] combina una serie de características que permiten comprender el reemplazo de las nuevas redes SDN por las redes tradicionales. En este caso, en el *switch* físico se hace correr una copia de *Switch Light*, el cual es un tipo de sistema operativo *Big Switch*.

La funcionalidad del controlador se provee a partir de *Big Switch Big Cloud Fabric,* el cual realiza todas las funciones del plano de control haciendo uso del protocolo OpenFlow. Esta solución también ofrece soporte para *switches* virtuales a través del producto antes mencionado de la compañía *Switch Light vSwitch*.

· *Plexxi Big Data Fabric*.

El controlador se llama *Plexxi Control* [62] y funciona junto con el *switch Plexxi* para tomar las decisiones en el plano de control de la red. Lo que diferencia a *Plexxi* es que el *hardware* lo realiza la propia compañía.

· HP *Virtual Application Networks (*VAN*) SDN Controller/Virtual Cloud Networking (*VCN*).*

HP VAN trabaja junto a *OpenStack* para proporcionar de un entorno de red virtual. La aplicación HP VCN extiende las capacidades de *OpenStack* proponiendo una serie de mejoras, incluyendo un mejor control de *Open vSwitch*, soporte de máquinas virtuales, soporte de *routers* de distribución virtual etcétera. Las redes SDN que utilizan este controlador trabajan con diferentes *switches* OpenFlow de HP.

· *Juniper Contrail*.

Es una versión compatible con *Open Contrail*, y está dirigido tanto a empresas como a proveedores de servicios. En las redes de empresas, *Contrail* [63] puede implementarse en la nube y en una plataforma de automatización, como *OpenStack*, y provee de una capa de virtualización de red la cual incluye soporte de conmutación, enrutado y servicios de red sobre la capa física. Además ofrece funcionalidades NFV (*Network Functions Virtualization*).

Como arquitectura, *Contrail* se constituye por dos componentes clave: el controlador *Contrail* y el *vRouter Contrail*.

Este controlador permite la configuración, control y análisis dentro de la red:

- El componente de configuración acepta peticiones para la asignación de recursos de la red.

- El componente de control interactúa con los elementos de red y dirige el aprovisionamiento de red hacia una máquina virtual utilizando el protocolo XMPP (no OpenFlow).
- El componente de análisis recopila, almacena, correlaciona y analiza la información de los elementos de red.

· Cisco *Application centric Infrastructure* (ACI)/*Application Policy Infrastucture Controller* (APIC).

Cisco ofrece un par de soluciones diferentes para redes SDN. La que parece prosperar más es la *Application Centric Infrastucture* (ACI) [64]. Esta solución es similar a la propuesta por *Plexxi*.

Se fundamenta en el *switch* "*Cisco Nexus 9000*". Estos *switches* utilizan una arquitectura particular, la cual hace uso de tecnologías para calcular la trayectoria de envío para diferentes *hosts* conectados. Este método de reenvío está pensado para ser completamente transparente para los *hosts* conectados y se realiza prácticamente en su totalidad a través del *hardware*.

El controlador *Application Policy Infrastucture Controller* (APIC) [65] es el controlador *software* central para esta solución. Está pensado para definir aplicaciones utilizadas para clasificar el tráfico mediante diferentes técnicas, incluyendo las tramas sin etiquetar, IEEE 802.1Q, VXLAN o NVGRE.

ACI tampoco hace uso de OpenFlow para la comunicación con el *switch*. Esta comunicación se realiza utilizando el protocolo Cisco OpFlex, aunque soporta también OpenFlow.

La clara ventaja de OpenDayLight sobre este tipo de controladores es sin duda el código abierto y la libertad para desarrollarlo, además del soporte que ofrece toda la comunidad que trabaja con este controlador.

Estadísticas de búsqueda de google trend.

Hemos realizado un estudio de la repercusión que ha tenido el controlador OpenDayLight a lo largo de los últimos años recogiendo estadísticas de búsqueda en Internet.

La evolución de OpenDayLight es totalmente creciente desde el año 2013 como puede apreciarse en el siguiente grafo:

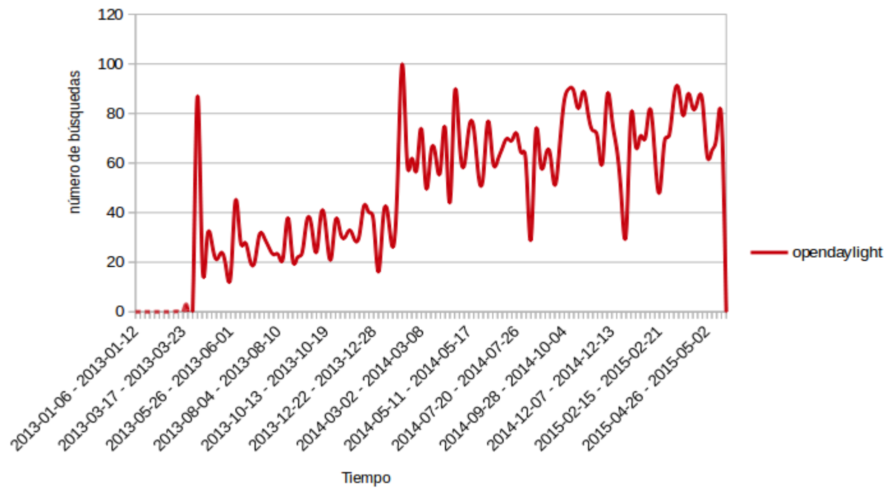


Figura F.7. Evolución de búsqueda de "OpenDayLight" mediante google.

También podemos ver cuáles son las búsquedas principales de OpenDayLight:

Búsquedas principales de opendaylight	
sdn opendaylight	100
opendaylight controller	95
OpenFlow	95
openstack opendaylight	75
opendaylight cisco	45
opendaylight install	40
mininet	40
opendaylight mininet	40
opendaylight java	40
opendaylight wiki	35
opendaylight tutorial	30
opendaylight project	30

Tabla F.2. Búsquedas principales de OpenDayLight

En contraposición, la búsqueda de otros controladores de código abierto relacionados con estas redes fue infructuosa, siendo su repercusión bastante baja en cuanto a su relación con SDN.

G. Topologías para *Mininet*.

En este anexo se incluyen algunas topologías que se han considerado relevantes, ya que nos proporcionan diferentes vías para conectar la red creada a Internet y, además, ofrecen una idea de cómo programar cualquier otro tipo de red utilizando Python. Pasamos ahora a presentar estas topologías.

Topología formada por tres *switches* y tres *host* con NAT:

```
#!/usr/bin/python

"""
Copyright (C) 2015 Manuel Sanchez Lopez

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have getTransmitErrorCountd a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
"""

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSSwitch, Controller, RemoteController
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info, lg
from mininet.link import TCLink, Intf
from subprocess import call

from mininet.nodelib import NAT
from mininet.topolib import TreeNet



"""
                          __________
                         /          \
           h1   nat0-----|  INTERNET |
           |     |       \__________/
           s1-----
     ______|_______
     |             |
   s2              s3
     |             |
   h2              h3
"""

def myNetwork():

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

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

    info( '*** Add switches\n')
    s1 = net.addSwitch('s3', cls=OVSSwitch, mac='00:00:00:00:00:06', protocols='OpenFlow13')
```

```
    s2 = net.addSwitch('s2', cls=OVSSwitch, mac='00:00:00:00:00:05', protocols='OpenFlow13')
    s3 = net.addSwitch('s1', cls=OVSSwitch, mac='00:00:00:00:00:04', protocols='OpenFlow13')

    info( '*** Add hosts\n')
    h1 = net.addHost('h3', cls=Host, ip='10.0.0.3', mac='00:00:00:00:00:03', defaultRoute='via
                     10.0.0.4')
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', mac='00:00:00:00:00:02', defaultRoute='via
                     10.0.0.4')
    h3 = net.addHost('h1', cls=Host, ip='10.0.0.1', mac='00:00:00:00:00:01', defaultRoute='via
                     10.0.0.4')
    # La ruta por defecto de los host es la de NAT (10.0.0.4 en este caso)

    info('*** Add NAT\n')
    net.addNAT().configDefault()


    info( '*** Add links\n')
    net.addLink(s1, s2, bw=10, delay='0.2ms')
    net.addLink(s1, s3, bw=10, delay='0.2ms')
    net.addLink(s1, h1, bw=10, delay='0.2ms')
    net.addLink(s2, h2, bw=10, delay='0.2ms')
    net.addLink(s3, h3, bw=10, delay='0.2ms')


    info( '*** Starting network\n')
    net.build()
    net.start()
    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()


    info( '*** Starting switches\n')
    net.get('s3').start([c0])
    net.get('s2').start([c0])
    net.get('s1').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
    net.stop()

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

Topología formada por dos *hosts* y un *switch* con un *bridge* creado para la conexón a internet:

```
#!/usr/bin/python

"""
Copyright (C) 2015 Manuel Sanchez Lopez

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have getTransmitErrorCountd a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
"""
```

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSSwitch, Controller, RemoteController
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info, lg
from mininet.link import TCLink, Intf
from subprocess import call

from mininet.nodelib import NAT
from mininet.topolib import TreeNet
"""
               bridge
        __________________
 h2 ----|                  |
        |--s1 ----wlan0---|---INTERNET
 h1 ----|_________________|
"""

def myNetwork():

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

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

    info( '*** Add switches\n')
    s1 = net.addSwitch('s1', cls=OVSSwitch, mac='00:00:00:00:00:04', protocols='OpenFlow13')
    #s1 = net.addSwitch('s1', ip='10.0.0.10')

    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', mac='00:00:00:00:00:01')
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', mac='00:00:00:00:00:02')

    info( '*** Add links\n')
    net.addLink(s1, h1, bw=10, delay='0.2ms')
    net.addLink(s1, h2, bw=10, delay='0.2ms')

    info( '*** Add bridge\n')
    Intf('eth1',node=s1)

    info( '*** Starting network\n')
    net.build()
    net.start()

    info( '*** Starting controllers\n')
    for controller in net.controllers:
        controller.start()

    info( '*** Starting switches\n')
    net.get('s1').start([c0])

    info( '*** Post configure switches and hosts\n')

    CLI(net)
    net.stop()

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

Bibliografía

[1] Wikipedia, «Redes definidas por Software,» [En línea]. Available: http://es.wikipedia.org/wiki/Redes_definidas_por_software .

[2] Wikipedia, «Deep Packet Inspection,» [En línea]. Available: https://en.wikipedia.org/wiki/Deep_packet_inspection.

[3] S. R. Santamaría, «Mecanismos de controls de las comunicaciones en la internet el futuro a través de openflow,» [En línea]. Available: http://repositorio.unican.es/xmlui/bitstream/handle/10902/1165/Sergio%20Rodriguez%20Santamaria.pdf?sequence=1.

[4] Cisco Corporation, «"Cisco visual networking index: forecast and methodology, 2012-2017",» [En línea]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html.

[5] CISCO, «CCNP Self-Study: Understanding and Implementing Quality of Service in Cisco Multilayer Switched Networks,» [En línea]. Available: http://www.ciscopress.com/articles/article.asp?p=170743.

[6] Google, «Tendencias de búsqueda Google Trends,» [En línea]. Available: http://www.google.es/trends/.

[7] YouTube, «YouTube,» [En línea]. Available: https://www.youtube.com/.

[8] YouTube, «Estadísticas oficiales de YouTube,» [En línea]. Available: http://www.ciscopress.com/articles/article.asp?p=170743.

[9] Ignacio Gavilán, «Slideshare: Fundamentos de SDN,» [En línea]. Available: http://es.slideshare.net/igrgavilan/20130805-introduccion-sdn.

[10] Open Networking Foundation, «Software Defined Networking Definition,» [En línea]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition.

[11] OpenDayLight, «OpenDayLight,» [En línea]. Available: http://www.opendaylight.org/.

[12] Mininet, «Mininet,» [En línea]. Available: http://mininet.org/.

[13] IETF, «Type of Service in the Internet Protocol Suite,» [En línea]. Available: https://tools.ietf.org/html/rfc1349.

[14] Open Networking Foundation, «SDN-resources: OpenFlow,» [En línea]. Available: https://www.opennetworking.org/sdn-resources/openflow.

[15] OpenFlow, [En línea]. Available: http://archive.openflow.org/wp/learnmore/.

[16] P. Goransson, C. Black, Elsevier y Morgan Kauffman, Software Defined Networks: A Comprehensive Approach, 2014.

[17] Open Networking Foundation, «OpenFlow Switch Specification,» 25 Junio 2012. [En línea]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[18] Xen, «Xen Project,» [En línea]. Available: http://www.xenproject.org/.

[19] Linux, «Kernel Virtual Machine,» [En línea]. Available: http://www.linux-kvm.org/page/Main_Page.

[20] ORACLE, «VirtualBox,» [En línea]. Available: https://www.virtualbox.org/.

[21] PROXMOX, «PROXMOX Virtual Environment,» [En línea]. Available: https://pve.proxmox.com/wiki/Main_Page.

[22] OpenStack, «OpenStack,» [En línea]. Available: https://www.openstack.org/.

[23] Open QRM Enterprise, «OpenQRM,» [En línea]. Available: http://www.openqrm-enterprise.com/.

[24] Open Nebula, «OpenNebula,» [En línea]. Available: http://opennebula.org/.

[25] oVirt, «Open Your Virtual Datacenter,» [En línea]. Available: http://www.ovirt.org/Home.

[26] ntop, «nDPI,» [En línea]. Available: http://www.ntop.org/products/deep-packet-inspection/ndpi/.

[27] BRO, «The Bro Network Security Monitor,» [En línea]. Available: https://www.bro.org/.

[28] SNORT, «SNORT,» [En línea]. Available: https://www.snort.org/.

[29] The fast mode, «Deep Packet Inspection Vendors & Products,» [En línea]. Available: http://www.thefastmode.com/deep-packet-inspection-vendors.

[30] F5, «Policy Enforcement Manager,» [En línea]. Available: https://f5.com/products/service-provider-products/policy-enforcement-manager.

[31] Slideshare, «F5 solutions for service providers,» [En línea]. Available: http://www.slideshare.net/BAKOTECH/f5-solutions-for-service-providers.

[32] Alcatel-lucent, «7750 Service Router - Mobile Gateway,» [En línea]. Available: https://www.alcatel-lucent.com/products/7750-service-router-mobile-gateway.

[33] Bivio, «Bivio products,» [En línea]. Available: http://www.bivio.net/products/b7000/.

[34] EstiNet Technologies, «EstiNet,» [En línea]. Available: http://www.estinet.com/.

[35] S.-Y. Wang, «IEEEXplore,» [En línea]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6912609.

[36] Mozilla, «DOM,» [En línea]. Available: https://developer.mozilla.org/es/docs/DOM.

[37] SDN tutorials, «OpenDaylight, Netconf, Restconf & YANG,» [En línea]. Available: http://sdntutorials.com/opendaylight-netconf-restconf-and-yang/.

[38] Cisco, «Java API para OpenDayLight,» [En línea]. Available: https://developer.cisco.com/media/XNCJavaDocs/org/opendaylight/controller/sal/utils/IPProtocols.html.

[39] OpenDayLight, «HOW ORANGE IS USING OPENDAYLIGHT,» [En línea]. Available: http://www.opendaylight.org/blogs/2015/06/how-orange-using-opendaylight.

[40] OpenDayLight, «Repositorios de OpenDayLight: Distribución Base,» [En línea]. Available: https://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/controller/distribution.opendaylight/.

[41] Apache, «Apache karaf,» [En línea]. Available: http://karaf.apache.org/.

[42] OpenDayLight, «OpenDayLight Wiki: OpenDayLight DLUX: DLUX Karaf Feature,» [En línea]. Available: https://wiki.opendaylight.org/view/OpenDaylight_DLUX:DLUX_Karaf_Feature.

[43] OpenDayLight, «Wiki OpenDayLight: OpenDaylight Controller:MD-SAL:FAQ,» [En línea]. Available: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ.

[44] SDN Tutorials, «Difference Between AD-SAL & MD-SAL,» [En línea]. Available: http://sdntutorials.com/difference-between-ad-sal-and-md-sal/.

[45] A. Marqués, «Conceptos sobre APIs REST,» [En línea]. Available: http://asiermarques.com/2013/conceptos-sobre-apis-rest/.

[46] Postman, «Postman,» [En línea]. Available: www.getpostman.com.

[47] OpenDayLight, «Wiki OpenDayLight: Editing OpenDaylight OpenFlow Plugin:End to End Flows:Example Flows,» [En línea]. Available: https://wiki.opendaylight.org/view/Editing_OpenDaylight_OpenFlow_Plugin:End_to_End_Flows:Example_Flows.

[48] SDN Hub, [En línea]. Available: http://sdnhub.org/.

[49] Mininet, «GitHub: Mininet-VM-Images,» [En línea]. Available: https://github.com/mininet/mininet/wiki/Mininet-VM-Images.

[50] OpenDayLight, «Distribución Helium de OpenDayLight,» [En línea]. Available: http://www.opendaylight.org/software/downloads.

[51] Apache, «Maven,» [En línea]. Available: http://maven.apache.org.

[52] Wikipedia, «OSGi,» [En línea]. Available: wikipedia.org/wiki/osgi.

[53] NOX, «About NOX,» [En línea]. Available: http://www.noxrepo.org/nox/about-nox/.

[54] NOX, «About POX,» [En línea]. Available: http://www.noxrepo.org/pox/about-pox/.

[55] NEC, «Helios Controller,» [En línea]. Available: http://www.nec.com/.

[56] Maestro, «Maestro Platform,» [En línea]. Available: https://code.google.com/p/maestro-platform/.

[57] OpenFLow, «Beacon Controller,» [En línea]. Available: https://openflow.stanford.edu/display/Beacon/Home.

[58] C. Álvarez, «¿Qué es Maven?,» [En línea]. Available: http://www.genbetadev.com/java-j2ee/que-es-maven.

[59] Project FloodLight, «FloodLight Controller,» [En línea]. Available: http://www.projectfloodlight.org/floodlight/.

[60] OpenDayLight, «OpenDayLight Wiki,» [En línea]. Available: https://wiki.opendaylight.org/view/Main_Page.

[61] Big Switch, «Big Cloud Fabric,» [En línea]. Available: http://www.bigswitch.com/sdn-products/big-cloud-fabrictm.

[62] Plexxi, «Plexxi Control,» [En línea]. Available: http://www.plexxi.com/products/plexxi-control/.

[63] Juniper, «Juniper Contrail,» [En línea]. Available: http://www.juniper.net/us/en/products-services/sdn/contrail/.

[64] Cisco, «Cisco Application Centric Infrastructure,» [En línea]. Available: http://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html.

[65] Cisco, «Cisco Application Policy Infrastucture Controller,» [En línea]. Available: http://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html.

[66] R. Manríquez Peralta, «Emulacion de SNDs usando Mininet».

[67] Cisco Corporation, «The Zettabyte Era—Trends and Analysis,» [En línea]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html.

[68] Alexa Corporation, «The yop 500 sites on the web,» [En línea]. Available: http://www.alexa.com/topsites.

[69] OpenDayLight, «OpenDayLight Wiki,» [En línea]. Available: https://wiki.opendaylight.org.

[70] Wikipedia, «Open vSwitch,» [En línea]. Available: https://es.wikipedia.org/wiki/Open_vSwitch.

[71] ntop, «nDPI,» [En línea]. Available: http://www.ntop.org/products/deep-packet-inspection/ndpi/.

[72] SEO basico, «Las páginas más visitadas y populares del mundo,» [En línea]. Available: http://www.seobasico.com/blog/.

[73] «How to use MiniEdit, Mininet's graphical user interface,» [En línea]. Available: http://www.brianlinkletter.com/how-to-use-miniedit-mininets-graphical-user-interface/.

[74] F5, «F5 products,» [En línea]. Available: https://f5.com/products/service-provider-products/policy-enforcement-manager.

[75] Panorama audiovisual, «El consumo de vídeo bajo demanda en España se equipara ya en España al de tv convencional,» [En línea]. Available: http://www.panoramaaudiovisual.com/2014/10/08/el-consumo-de-vídeo-bajo-

demanda-en-espana-practicamente-se-equipara-ya-en-espana-al-de-television-convencional/.

[76] ORACLE, «Administración de Oracle Solaris: interfaces y virtualización de redes,» [En línea]. Available: http://docs.oracle.com/cd/E26921_01/html/E25833/gfkbw.html.

[77] Open Networking FOundation, «Acerca de OpenFlow,» [En línea]. Available: http://archive.openflow.org/wp/learnmore/.

[78] The Guardian, «A history of media streaming and the future of connected TV,» [En línea]. Available: http://www.theguardian.com/media-network/media-network-blog/2013/mar/01/history-streaming-future-connected-tv.

[79] tom'sIT PRO, «A Guide To Software Defined Networking (SDN) Solutions,» [En línea]. Available: http://www.tomsitpro.com/articles/software-defined-networking-solutions,2-835-2.html.

[80] Open Networking Foundation, [En línea]. Available: https://www.opennetworking.org/index.php.

[81] Amazon, «Precio de productos F5,» [En línea]. Available: https://aws.amazon.com/marketplace/seller-profile?id=74d946f0-fa54-4d9f-99e8-ff3bd8eb2745.

[82] P. Goransson, C. Black, Elsevier y M. Kauffman, «Software Defined Networks: A Comprehensive Approach,» 2014.