TRABAJO FIN DE GRADO

INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Diseño Diseño y evaluación de un servicio OpenFlow de provisión de Calidad de Experiencia sobre Mininet

Autor
Cristian Alfonso Prieto Sánchez

Director
Juan José Ramos Muñoz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Julio de 2015

Diseño y evaluación de un servicio OpenFlow de provisión de Calidad de Experiencia sobre Mininet

Autor
Cristian Alfonso Prieto Sánchez

Director
Juan José Ramos Muñoz

Departamento de Teoría de la Señal, Telemática y Comunicaciones

—

Granada, Julio de 2015

Diseño y evaluación de un servicio OpenFlow de provisión de Calidad de Experiencia sobre Mininet

Cristian Alfonso Prieto Sánchez

Palabras clave: OpenDayLight, SDN, OpenFlow, Routing, Dijkstra, redes, Java, Controller, mininet, QoE, QoS.

Resumen

Las redes definidas por software (Software Defined Networking (SDN)) presentan un cambio de paradigma para las redes futuras, consistente en la separación del plano de control y el de transporte. Introducen grandes ventajas frente a las redes tradicionales. Por un lado los elementos de red podrán ser mucho más sencillos, sin que esto suponga pérdidas de capacidad en ningún caso. Por otro lado se conseguirá un aumento en flexibilidad y velocidad en la automatización de tareas de red debido a la centralización de estas tareas en un elemento único, el controlador.

Las grandes ventajas que presenta este nuevo paradigma ha atraído la atención de grandes fabricantes de la industria, que ya cuenten con sus propias soluciones para redes SDN. Esto no ha supuesto una privatización de conocimiento o un estancamiento de la tecnología, todo lo contrario, ya que ha impulsado las líneas de investigación sobre redes SDN en la comunidad científica e investigadora.

La transmisión de contenido multimedia supone uno de las mayores fuentes de tráfico de red en la actualidad, y se estima que tráficos como el de vídeo constituirán más del 75 % del tráfico de red en menos de 5 años. Estas transmisiones multimedia requerirán unas condiciones mínimas de transmisión para asegurar Calidad de Servicio (QoS).

En este trabajado se pretenden desarrollar servicios basados en SDN capaces de proporcionar Calidad de Experiencia (QoE) para aplicaciones multimedia, demostrando la capacidad de redes programables para satisfacer requisitos. Concretamente, se diseñará y desarrollará un protocolo de encaminamiento basado en QoS para distintos tipos de tráfico, que sea capaz de reaccionar rápidamente en caso de cambios en la topología de red. Para ello estudiaremos los conceptos que permiten la conformación de redes SDN, los protocolos de control y gestión desarrollados para esta tecnología y las principales implementaciones y entornos de desarrollo (privadas y abiertas) de controladores SDN.

El ahorro en costes es otro de los puntos que ha suscitado el interés de las grandes compañías por redes SDN. La solución presentada en este trabajo también aborda este aspecto, ya que automatiza la configuración de red, permitiendo al administrador ahorrar costes y tiempo en tareas de gestión de redes.

Para poder llevar a cabo la implementación de la solución propuesta, será asimismo necesario familiarizarse con el uso de varias herramientas de desarrollo de aplicaciones de red SDN (emulador de redes, entornos de desarrollo, etc) que permitan alcanzar los objetivos propuestos.

No se pretende dar una solución que abarque todos los conceptos y aspectos relacionados con calidad de servicio y de experiencia, pero sí demostrar que las nuevas redes programables añaden nuevas posibilidades y características que mejoran la gestión y automatización de red.

Diseño y evaluación de un servicio OpenFlow de provisión de Calidad de Experiencia sobre Mininet

Cristian Alfonso Prieto Sánchez

Keywords: OpenDayLight, SDN, OpenFlow, Routing, Dijkstra, redes, Java, Controller, mininet, QoE, QoS.

Abstract

The evolution of multimedia services cause new challenges for the service providers. How can they deal with this challenges? In the same direction mobile networks are 5G are evolving and developing new concepts for their network architecture. At the same time, service providers are looking for new methods to reduce costs of the monitoring and management of their networks. The SDN architecture is the answer for these questions for many reasons. If we focus on theoretical reasons, SDN allows to reduce CAPital EXpenditures (CAPEX) and OPEratix EXpense (OPEX) costs and also allows dynamic and faster management of the network than traditional architectures. To use the new architectures will be necessary a special design protocol which allows to manage the new networks. A lot of different protocols could be used, but the most important is OpenFlow protocol. OpenFlow is specially designed to get the best capacity of the new network architecture SDN so that is the principal reason to use this protocol. Additionally, other reasons to select OpenFlow is that it has open source implementations, and that it is supported by a number of important companies.

The support for the OpenFlow protocol will be very important to decide which controller we will choose for our solution. OpenDayLight is a new controller under the Linux Foundation which is supported for a lot of companies such as Cisco, Intel or HP among others. Also, it is necessary to put the focus on the open character of the project, and the active support given by the community of users.

OpenDayLight satisfies these necessary requisites. It is an Open Source controller with enough facilities to develop a network aplication for the controller. Throughout the project, we will study the concepts which make possible the development of new network applications in an SDN controller.

In this project we will focus on the QoE and QoS support that SDN networks can provide to developers. We will design a solution to fullfill this objetive. After this we will evaluate our proposal. The solution will use traffic engineering to evaluate the costs for each link (edge). It will also use the Dijkstra Algorithm to find the best path between two nodes. Finally, our solution will provide a way to delete the routes in switches if a broke path is detected (for example, a down link) so we will develope a solution for guaranteeing that the network will work after links outages.

Finally we will implement the solution using the JAVA programming language. We will describe the necessary steps to get a robust aplication which should work correctly.

We will evaluate the developed application by network emulation, defining several experimental scenarios. The evaluation tests will consider many differents points of view to be sure about the correct work under distinct working conditions. We will use different measures to estimate the subjective assessment of final users under different situations. These estimations will be obtained by applying different standardized theoretical models which will be studied along the document.

To end with, we will present the conclusions about this project in the last chapter, as well as the keys to continue studing and developing in future works.

Yo, **Cristian Alfonso Prieto Sánchez**, alumno de la titulación 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 75926381-T, 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: Cristian Alfonso Prieto Sánchez

Granada a 6 de Julio de 2015.

D. **Juan José Ramos Muñoz**, Profesor del Área de Telemática del Departamento Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado ***Diseño y evaluación de un servicio OpenFlow de provisión de Calidad de Experiencia sobre Mininet***, ha sido realizado bajo su supervisión por **Cristian Alfonso Prieto Sánchez**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 6 de Julio de 2015.

El director:

Juan José Ramos Muñoz

Agradecimientos

Dedicado a
Cada persona que me ha apoyado.
Que siempre me han hecho creer en mí.
Que nunca han dejado que me rinda.
Como no a Manu, el apoyo de todos estos meses
y en especial a ella que siempre está
y siempre saca lo mejor de mí.

Referencia de siglas y acrónimos

AAA Authentication, Authorization and Accounting

AD API Driven

API Application Programming Interface

APIC Application Policy Infrastructure Controller

ATM Asynchronous Transfer Mode

BGP Border Gateway Protocol

CAPEX CAPital EXpenditures

DHCP Dinamic Host Control Protocol

DOM Document Object Model

ForCES Forwarding and Control Element Separation

GENI Global Environment for Network Innovations

HD High Definition

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

ICMP Internet Control Message Protocol

IDE Integrated Development Environment

IETF Internet Engineering Task Force

IP Internet Protocol

IRIS Interconexión de Recursos InformáticoS

ITU Unión Internacional de telecomunicaciones

JAR Java ARchive

JSON JavaScript Object Notation

MD Model Driven

MOS Mean Opinion Score

MPEG Moving Picture Experts Group

MPLS MultiProtocol Label Switching

NAT Network Address Traslation

NB NorthBound

NETCONF Network Configuration Protocol

ODL OpenDayLight

ONF Open Networking Foundation

ONS Open Networking Summit

OPEX OPEratix EXpense

OSGi Open Services Gateway Initiative

OSPF Open Shortest Path First

PASITO Plataforma de Análisis de Servicios de Telecomunicaciones

PCE Path Computation Element

PCEP Path Computation Element Protocol

POM Project Object Model

QoE Calidad de Experiencia

QoS Calidad de Servicio

RCP Routing Control Platform

REST Representational State Transfer

RTP Real-time Routing Protocol

SAL Service Abstraction Layer

SB SouthBound

SDN Software Defined Networking

SO Sistema Operativo

SNMP Simple Network Management Protocol

TCP Transmission Control Protocol

TLS Transport Layer Security

TTL Time To Life

UDP Usser Datagram Protocol

UML Unified Modeling Language

VLC Video Local Area Network Client

VAN Virtual Application Networks

VoIP Voice Over IP

XML eXtensible Markup Language

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción y objetivos

Este primer capítulo se presentarán los conceptos básicos que se trabajarán a lo largo de esta memoria. En primer lugar se describirá el marco global donde se encuadra SDN (*Software Defined Networking*). Una vez presentada el área de aplicación del objeto de estudio de este trabajo, se presentarán las motivaciones que justifican la realización de este proyecto, y los objetivos que se pretenden alcanzar. Por último se describirá brevemente la estructura de la memoria para ayudar al lector a localizar con claridad los capítulos mayor interés.

1.1. Contexto

Si nos detenemos en la evolución futura de redes móviles nos topamos con las novedosas redes 5G. Redes que prometen una mayor capacidad y mejores capacidades para ofrecer servicios. Para alcanzar estas capacidades los proveedores de servicios necesitan adoptar nuevas arquitecturas de red, y SDN surge como una de las de mayor peso.

Estos proveedores buscan una constante evolución en sus redes, enfocada a conseguir mayor flexibilidad y mejoras de servicio sin aumentar costes, o incluso reduciéndolos. Existen dos tipos de costes asociados a las redes: el CAPEX (*CAPital Expenditure*) y el OPEX (*OPerational Expenditure*). En la sección 2.1 se realiza un estudio sobre como SDN puede ser el motor impulsor de las nuevas arquitecturas de red, como además se demuestra su aplicación en [2] entre otras que se estudiarán en los siguientes apartados.

Otro de los aspectos fundamentales en el impulso de redes SDN es el crecimiento de dispositivos con necesidades de acceso a la red. Este crecimiento está sustentado en dos pilares: desarrollo de zonas sin acceso e implantación de sistemas *wearables*. Teniendo en cuenta estos dos focos de crecimiento, según *Cisco* se puede estimar un crecimiento en el tráfico de red como el presentado en la figura 1.1 extraído de [3].

De esta figura podemos ver como el crecimiento de usuarios y tráfico seguirá una tendencia exponencial, con un crecimiento anual del 21 %, siendo todo un reto para la industria soportar tal cantidad de tráfico. Si ahora desglosamos el crecimiento por tipo de tráfico obtenemos los resultados que se aprecian en la figura 1.2. Podemos comprobar que el mayor crecimiento es el de tráfico en redes móviles (crecimiento en porcentaje, no en volumen), con un crecimiento de más del 60 % como se muestra en [3].

A partir de estas figuras surgen varias cuestiones ¿Qué tipo de tráfico será el que sufra un mayor aumento y sea más crítico? ¿Están las redes preparadas para soportar un incremento de más del 60 % en redes móviles? En caso de que no lo estuviesen, ¿qué soluciones existen?

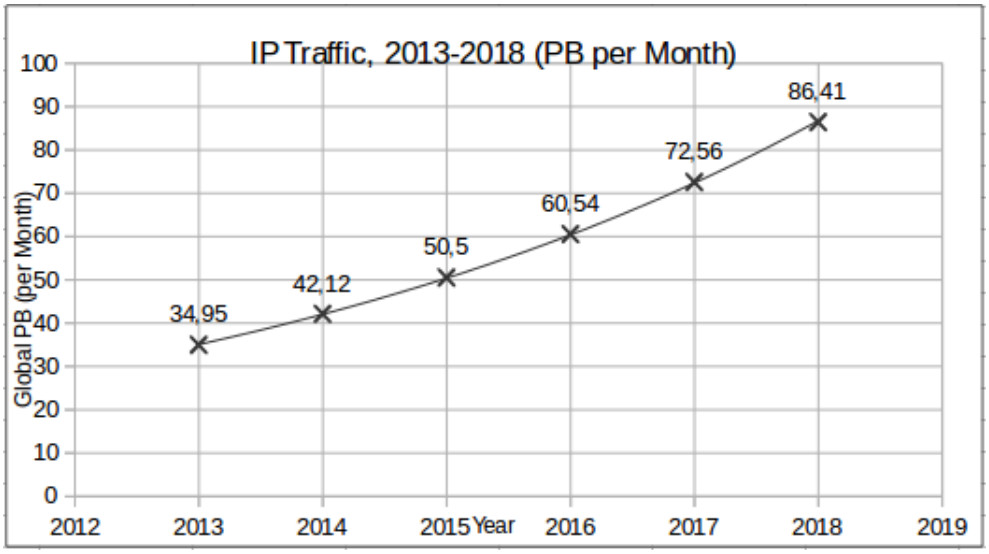


Figura 1.1: IP Traffic, 2013-2018 (PetaBytes por mes). Datos tomados de [3].

Conviene estacar que del tráfico generado por usuarios individuales (más del 80 % del total), uno de los mercados que experimentará mayor crecimiento será el de vídeo por Internet como se indica en la figura 1.3, demostrando la necesidad de replantear las soluciones actuales para servicios multimedia.

También es muy interesante para el estudio del contexto el artículo [4] que trata el crecimiento de tráfico en redes móviles, ya que son las que presentan un mayor aumento (aunque repetimos que no en volumen). En este segundo estudio (posterior al primero) se ajusta la previsión para el crecimiento de tráfico para este tipo de redes como se puede comprobar en la figura 1.4, pero en líneas generales se puede ver como la tendencia al alza se mantiene.

Uno de los resultados más interesantes y con mayor relevancia para constatar la importancia de redes SDN es la previsión de [4] sobre a los tipos de tráfico que generaran las redes móviles hasta 2019 como se muestra en la figura 1.5.

Estos resultados (junto al resto que se recogen en los citados artículos) presentan un contexto idóneo para pensar en la necesidad de evolución de las redes actuales, siendo SDN una de las soluciones de mayor relevancia a día de hoy. Se tratará más detenidamente las ventajas de usar SDN en la sección 2.1 pero conviene destacar que el éxito de esta tecnología está impulsado por beneficios teórico técnicos (poca inversión inicial, ahorro en costes, agilidad de red) y por casos de éxito real que demuestran las previsiones teóricas (Google [2]) o casos más concretos para redes multimedia (Verizon [5]).

Como último punto en el que situar este proyecto nos hacemos eco de una noticia del 9 de Junio de 2015 (se incluye por su relevancia para la justificación de la elección de SDN y la plataforma *OpenDayLight*). Esta noticia trata sobre la forma en la que una gran compañía como es Orange está haciendo uso de *OpenDayLight*. La noticia puede ser consultada en [6] y recoge los grandes beneficios que serán presentados en las secciones posteriores.

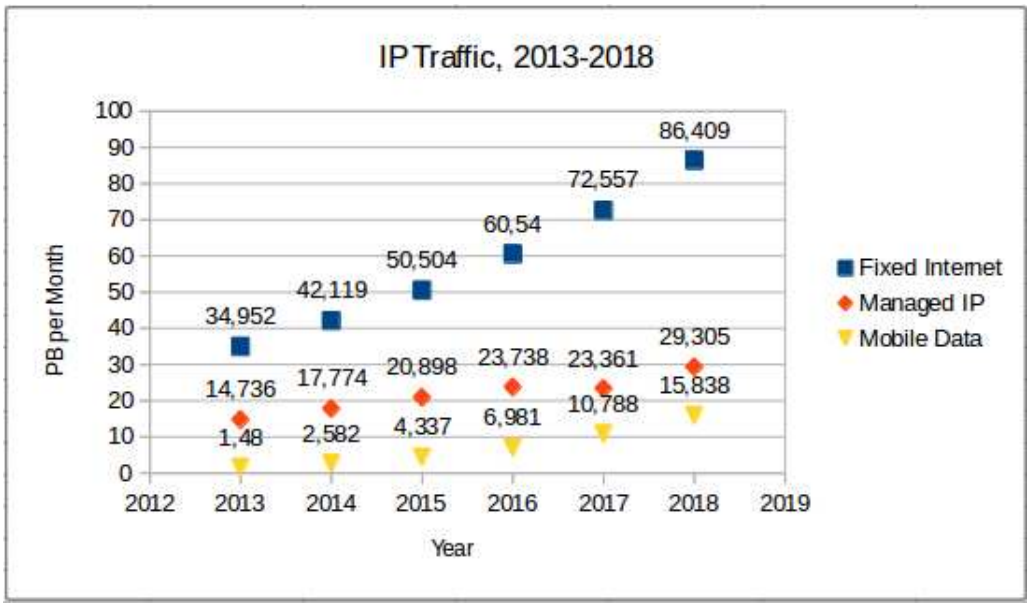


Figura 1.2: IP Traffic, 2013-2018 (PetaBytes por mes) por tipo de tráfico. Datos tomados de [3].

1.2. Motivación y objetivos

La justificación para la realización de este trabajo puede ser sacada directamente del contexto. Por un lado la necesidad de responder ante las demandas de tráfico de manera flexible y rápida, mejorando la gestión de estas redes y automatizándolas. Por otro lado la demanda de contenidos multimedia que precisan de unos fuertes requisitos (vídeo o VoIP precisan de un alto ancho de banda o latencias bajas) llevan a las grandes corporaciones del sector a buscar nuevas fórmulas y modelos que permitan satisfacer las demandas requeridas para ofrecer *calidad de servicio* (*QoS, Quality of Service*) que al final se convierte en una buena *calidad de experiencia* (*Quality of Experience, QoE*), es decir, la calidad que percibe el usuario final. Estos dos concetos se pueden definir de la siguiente manera:

- **QoS.** Definimos QoS como el rendimiento promedio de una red telemática. Cuantitativamente mide la calidad de los servicios en base a varios aspectos del servicio de red, tales como tasas de errores, ancho de banda, rendimiento, etc.
- **QoE.** Cuando se trata el término QoE se hace referencia a la calidad subjetiva que es percibida por un usuario final al disfrutar de un servicio ofrecido sobre una red telemática. Es una medida que se ve afectada por todos los elementos de la red extremo a extremo, que repercuten en la percepción final por parte del usuario.

Siendo SDN una de las soluciones con mayor prestigio y aceptación por parte de la industria, y estando aún en desarrollo continuo, se pretende en este trabajado alcanzar los siguientes objetivos:

- Revisar el estado de la técnica sobre redes SDN, protocolo *OpenFlow* y principales controladores.
- Revisar el estado de la técnica sobre emulación de redes SDN, *Mininet*.
- Diseñar una solución capaz de proveer QoE en redes SDN.

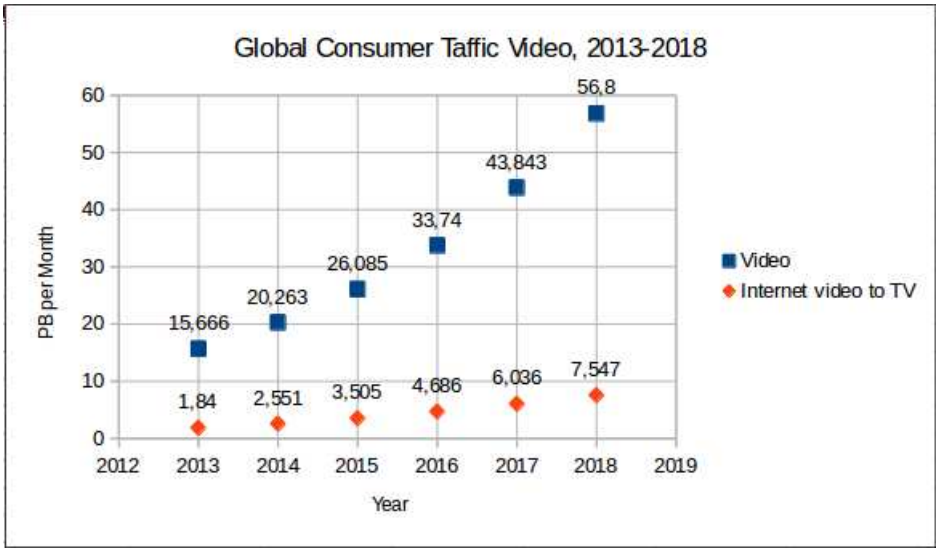


Figura 1.3: Global Consumer Video Traffic, 2013-2018. Datos tomados de [3].

- Implementar y evaluar dicha solución.

1.2.1. Alcance del proyecto

Se describe a continuación el alcance del proyecto. Este proyecto pretende ser un primer paso hacia el estudio de redes SDN y sus posibilidades mediante los siguientes puntos:

- Revisión del estado del arte. Se realizará un amplio estudio sobre las tecnologías implicadas y que son objeto de análisis en este proyecto.
- Estudio de herramientas SDN. Será necesario describir las herramientas que posibilitarán la emulación y evaluación de las capacidades de redes SDN.
- Diseño de la solución. Para alcanzar los objetivos propuestos es necesario el diseño de un algoritmo capaz de integrar las capacidades requeridas. Mediante la división en módulos se pretende encontrar una solución óptima para alcanzar los objetivos propuesto.
- Implementación de la solución. Para alcanzar los objetivos será necesario encontrar el modo de poder desarrollar e implementar una aplicación en un controlador para redes SDN.
- Evaluación de la solución. Una vez implementada la solución, se deberá buscar el mejor camino para evaluar ésta.

1.3. Bibliografía relevante

Esta sección pretende recoger las citas bibliográficas que se han considerado más importantes para la realización del proyecto.

- **Redes SDN** Las referencias más destacadas son:

  Definición SDN:

  [7] M. Boucadair and C. Jacquenet, "Software-defined networking: A perspective from within a service provider environment," 2014.

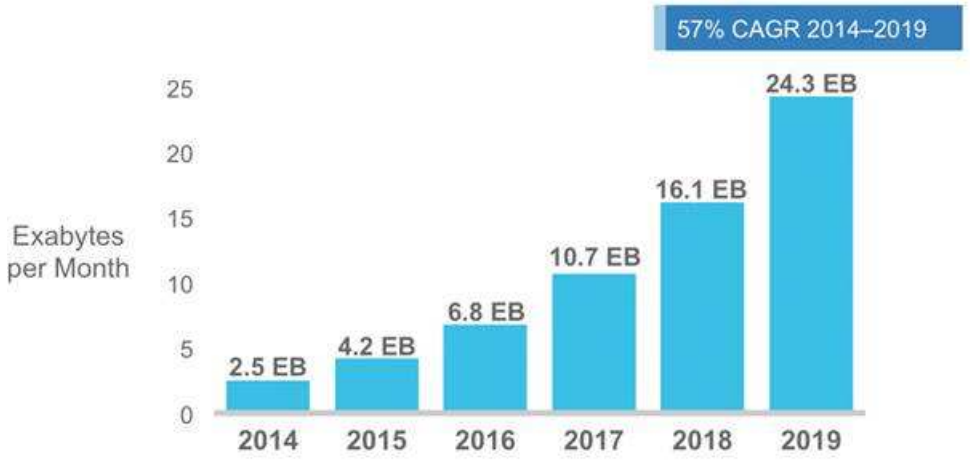


Figura 1.4: Mobile Data Traffic, 2014-2019 (ExaByte por mes). Imagen tomada de [4].

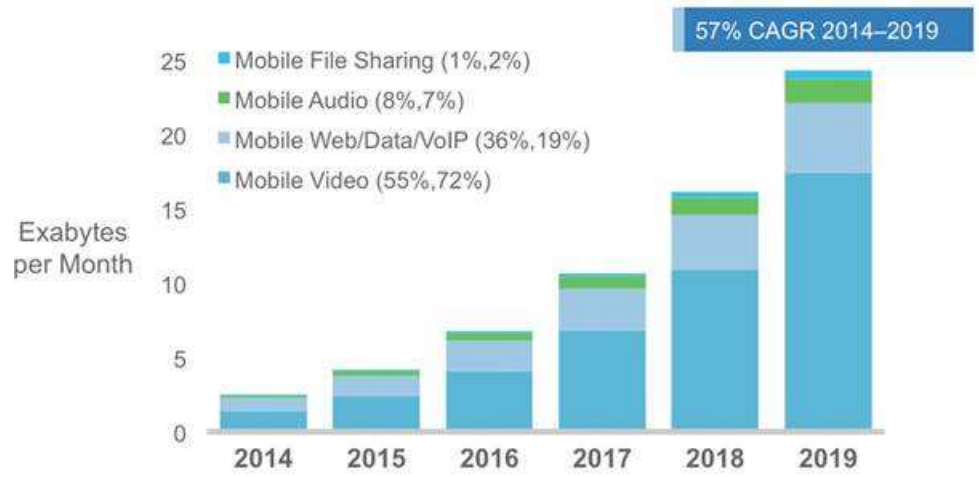


Figura 1.5: Generación de tráfico por tipos en redes móviles. Imagen tomada de [4].

Historia redes SDN:

[8] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

- ***OpenDayLight***

  Guía del desarrollador:

  [9] L. Foundation. (2015) Opendaylight developer guide. Último acceso 20 de Junio de 2015. [Online]. Available: `https://www.opendaylight.org/sites/opendaylight/files/Developer-Guide-Helium-SR2.pdf`

- **Obtención objetiva de calidad de audio sobre RTP**

  Recomendación ITU:

  [10] ITU, "El modelo e: un modelo informático para utilización en planificación de transmisión", 2014. [Online]. Available: `https://www.itu.int/rec/T-REC-G.107-201402-I/es`

- **Obtención objetiva de calidad de vídeo sobre RTP**

  Recomendación ITU:

  [11] ITU, "Opinion model for video-telephony applications," 2012. [Online]. Available: `http://www.itu.int/rec/T-REC-G.1070-201207-I/en`

1.4. Estructura de la memoria

En esta sección pretendemos presentar la estructura que tendrá la memoria, a fin de que esta sea lo más clara posible, facilitando al lector examinar la memoria de forma organizada y clara.

- Segundo Capítulo. Comenzaremos realizando un estudio sobre el estado del arte. Se describirán en él las tecnologías y conceptos en los que nos apoyaremos durante el desarrollo teórico/práctico del trabajo, haciendo especial hincapié en la razones que justifican el uso de tecnologías frente a otras.

- Tercer capítulo. En este capítulo se realiza una planificación temporal y una estimación de costes de toda la realización del trabajo.

- Cuarto capítulo. Introduciremos el entorno de desarrollo en el que se va a ejecutar todo el proyecto. Siendo este un capítulo de especial importancia para todo aquel que desee replicar o realizar otro proyecto de implementación sobre SDN. Trataremos todas las herramientas software que permitirán realizar las pruebas, así como algunos equivalentes en el mercado Hardware.

- Quinto capítulo. En el capítulo quinto nos centraremos en el diseño del sistema, describiendo los "módulos"que componen la solución propuesta de forma separada, con el fin de esclarecer cada uno de los diseños realizados, como por ejemplo la detección de cambios en la topología.

- Sexto capítulo. Describiremos la implementación que se ha llevado a cabo a partir del diseño del capítulo 4, prestando especial atención a aquellas partes de implementación que pueden resultar confusas, como el formato necesario para el código fuente.

- Séptimo capítulo. Se realizará una evaluación sobre la implementación del sistema. Por un lado evaluaremos las capacidades de los módulos por separado (asegurando el correcto funcionamiento de estos), y posteriormente realizaremos la evaluación del sistema final. Esta evaluación estará dividida en dos, evaluación objetiva (basada en datos como latencia o pérdidas de paquetes) y evaluación subjetiva (experiencia de usuario).

- Capítulo octavo. Por último en el capítulo séptimo se mostrarán las conclusiones extraídas tras el montaje y evaluación del sistema, así como se intentarán dar las claves que permitan continuar, mejorar o derivar en otras líneas de trabajo futuras.

Finalmente se pueden encontrar todas las referencias bibliográficas utilizadas tanto en la realización del proyecto, como las usadas para la realización de la memoria en particular además de un apéndice donde se aporta ayuda con las configuraciones de los diferentes elementos Software, además de código usado en el proyecto y algunos aspectos de configuración.

Como valor añadido a la memoria se incluye el enlace al blog: `www.aprendiendoodl.wordpress.com`. Este blog ha sido desarrollado por Manuel Sánchez López y por el autor de esta memoria, Cristian Alfonso Prieto Sánchez, con el objetivo de introducir al mundo SDN y ODL a todo aquel que esté dando sus primeros pasos con esta tecnología. Consideramos que la guía durante esta iniciación es muy importante debido a la falta de información concreta o falta de tutoriales (tecnología muy novedosa y en constante desarrollo). Por tanto emplazamos a aquel que a partir de la temática de este trabajo y el de Manuel Sánchez López sobre inspección de paquetes, sienta interés por estas novedosas tecnologías, visite el blog donde hallará ejemplos prácticos y explicaciones menos técnicas y más guiadas.

Capítulo 2

Estudio del estado del arte

En este capítulo vamos a describir exhaustivamente las tecnologías en las cuales está basado este trabajo. Comenzaremos con un análisis sobre redes SDN. En el siguiente punto analizaremos el protocolo más importante para redes SDN, *OpenFlow* y algunos otros que han surgido a partir del interés generado. Por último haremos un repaso de los principales controladores de redes SDN y centraremos la atención en el elegido para el desarrollo, *OpenDayLight*.

2.1. Software Defined Networking

Hasta ahora se han presentado los requisitos que deberán cumplir las redes futuras (crecimiento del número de usuarios, redes móviles o transmisión multimedia) y se han establecido los bases por las cuales podemos decir que las redes actuales serán incapaces de satisfacer la demanda de los usuarios, todo ello en la sección 1.1. El desafío de la industria consiste en idear un nuevo modelo de red capaz de responder a las necesidades futuras, y una de las alternativas más prometedoras son las redes definidas por software, SDN.

2.1.1. Historia y desarrollo

Las redes definidas por software representan un concepto relativamente novedoso que aún se encuentra en desarrollo. Sin embargo el concepto en si lleva años madurándose. A continuación se presenta la evolución de las primeras ideas relacionadas con SDN hasta ahora, según el orden presentado en [8].

En los años noventa se produjo el surgimiento de Internet para el gran público. El nacimiento de esta revolucionaria tecnología provocó un afán por la mejora de infraestructuras de red. Se pretendía poder gestionarlas de un modo mucho más efectivo, siendo uno de los objetivos hacerlas programables, siendo capaces de conmutar según las necesidades de cada momento.

El despliegue de Internet provocó un enorme desarrollo en las aplicaciones que a su vez se tradujo en la necesidad de diseñar nuevos protocolos de red para poder soportarlas. El problema residía en que cada protocolo debía ser estandarizado por la *Internet Engineering Task Force (IETF)*, siendo este un proceso demasiado lento para muchos investigadores.

Como alternativa, algunos grupos de investigación apostaron por un enfoque de apertura de control sobre las redes, permitiendo reprogramar las redes. Pero dado que las redes convencionales no eran ni son programables, surgieron las *redes activas*. Esta redes supusieron una aproximación radical hacia el control de red, basándose en una interfaz programable con capacidad para mostrar los recursos de cada nodo individual y capaz de aplicar acciones concretas para un grupo de paquetes definidos (flujos).

En contraposición, se encontraban muchos grupos que apostaban por la simplicidad en el núcleo de red, ya que según ellos esta parte es crítica para el éxito de Internet. Aún así la comunidad de redes activas propuso alternativas a los modelos tradicionales basados en Internet Protocol (IP) o Asynchronous Transfer Mode (ATM) (alternativas dominantes en la época). Surgieron dos modelos programables.

- **Modelo encapsulado.** Se propuso en este modelo encapsular las acciones que debía ejecutar el nodo, dentro de la propia información de cada paquete. De esta manera, porciones de código que se interpretaban en los nodos de red eran transportadas por los propios paquetes.

- **Modelo programable.** El código que debía ser ejecutado por cada nodo se generaba por aplicaciones o procesos externos al nodo.

El desarrollo *redes activas*, que no llego a extenderse ampliamente, dio lugar a tecnologías que hoy día se recogen y usan en SDN, como las funciones programables de los elementos de red, virtualización de red, por ejemplo.

A principios del nuevo milenio surge un nuevo concepto: la separación del plano de control (donde se encuentran las funciones de control de red) y de datos (funciones de reenvío de paquetes). El aumento del tráfico en Internet y la necesidad de redes manejables, predecibles y confiables llevaron a los investigadores a buscar nuevos enfoques para las funciones de gestión de red. Una de estas funciones era la ingeniería de tráfico, cuyas posibilidades se veían limitadas por los protocolos de enrutamiento convencionales, perdiendo capacidad.

La tarea de analizar y resolver problemas de configuración o controlar el comportamiento de enrutamiento de paquetes recaía durante esta época en *switches* y *routers*, una tarea demasiado compleja para los recursos con los que trabajaban. Fue esta razón la que derivó en la separación de estos planos y la que dio lugar a posteriores estándares y soluciones.

El crecimiento de Internet llevó a las empresas de equipos hardware a incluir lógica de reenvío de paquetes en estos equipos (separando así los dos planos). Mientras tanto los proveedores de servicio luchaban por gestionar las redes (crecientes) mientras añadían nuevos servicios para el usuario. En este contexto surgen varias innovaciones que serán importantes para el posterior desarrollo de SDN. Por ejemplo surgió una interfaz abierta entre los planos de datos y control, Forwarding and Control Element Separation (ForCES), estandarizada por la IETF o NetLink (proporciona la funcionalidad de reenvío de paquetes a nivel de núcleo Línux). Desde el punto de vista del control centralizado surgen *Routing Control Platform (RCP)*, arquitecturas *SoftRouter* (equipos programables) y el protocolo *Path Computation Element (PCE)*. Todos estos conceptos serán clave en el desarrollo futuro de SDN como se describe en [8].

En ese momento era necesario encontrar el modo de separar el plano de control y datos como en la imagen 2.1. El comienzo de este desarrollo estuvo centrado en la investigación de nuevas arquitecturas que permitieran el control lógico centralizado. Uno de los primeros proyectos (y de los de mayor importancia) fue el proyecto 4D[1]. En este proyecto se establecía la separación de la red en cuatro capas como se explica en [12]. Las cuatro capas que se contemplaron correspondían al *plano de datos* (encargado del reenvío basándose en las reglas, configurables), *plano de descubrimiento* (encargado de recoger estadísticas de tráfico y equipos en la red), el *plano de diseminación* (encargado de instalar reglas en los equipos que se ocuparán de aplicarlas) y el *plano de decisión* (encargado de tomar decisiones acerca del enrutamiento de paquetes en la red).

[1] http://www.cs.cmu.edu/ 4D/

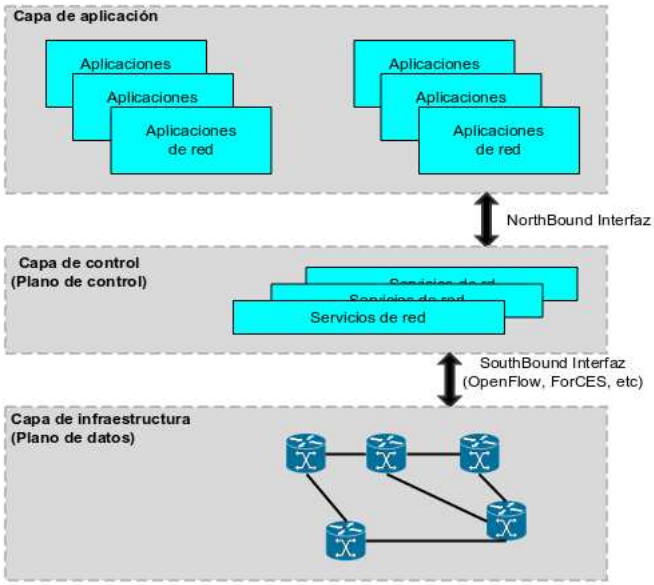


Figura 2.1: Arquitectura para la separación de planos de control y datos.

A partir de este trabajo, numerosos grupos comenzaron a seguir esta línea de investigación. El más relevante para nuestro proyecto, y uno de los más importantes, es el proyecto Ethane. Surgido en la universidad de Stanford, y cuyo desarrollo acabó siendo la base de *OpenFlow*. Para más información sobre los objetivos y resultados obtenidos en este proyecto se aconseja visitar [13].

El desarrollo total de *OpenFlow* como estándar abierto se produjo a mediados de la primera década del actual siglo, gracias a la experimentación en redes de gran escala (PlanetLab [14] o Emulab [15] entre otros), que dieron a lugar a un gran entusiasmo por parte de la comunidad investigadora a la hora de realizar experimentos en estos entornos. De este entusiasmo surgió el *Clean Slate Program*[2] enfocado en la investigación sobre redes universitarias y que daría lugar al estándar *OpenFlow* como hoy lo conocemos.

2.1.2. Definición SDN

Para definir SDN vamos a recurrir a la propia definición realizada por la Open Networking Foundation (ONF) en [16].

SDN es una arquitectura dinámica, manejable, rentable y adaptable, capaz de soportar el gran ancho de banda y la naturaleza dinámica de las aplicaciones actuales. Las características más importantes de SDN son:

- **Son redes programables.** El desacoplamiento entre las capas de control y aplicación permiten la programación directa de la red sin preocuparnos por las funciones de reenvío.

- **Permite un desarrollo ágil.** De igual manera que permite la programación dinámica de red, el desacoplamiento entre capas también permite que la modificación de flujos y funciones de red sea muy rápido y adaptable.

[2]Programa de investigación de la universidad de Stanford que trabaja en como debería ser rediseñado Internet desde 0 (borrón y cuenta nueva).

- **Permite la gestión centralizada.** La arquitectura SDN se basa en un control centralizado que mantiene una visión global de la red y que es el elemento que interactúa con todos los demás añadiendo, eliminando o modificando flujos.
- **Permite automatizar la programación de los dispositivos.** SDN permite a los administradores configurar programas escritos por ellos mismos (SDN no depende de Software propietario) que agilicen todo el proceso de configuración, optimización y administración de recursos de red.
- **Está basada en estándares abiertos.** Cuando SDN se implementa a través de estándares abiertos, ésta permite la simplificación de red al no depender de proveedores específicos, dispositivos de traducción o protocolos propietarios.

La figura 2.1 es una simplificación de la arquitectura SDN donde podemos comprobar la separación entre capas gracias al uso de protocolos de control gracias al uso de protocolos de control.

2.1.2.1. ¿Por qué SDN?

Para justificar la relevancia de SDN y su uso en este trabajo podemos recurrir a razones principales: la primera, los beneficios de usar SDN frente a otras tecnologías o arquitecturas. La segunda, el apoyo por parte de la industria del sector.

2.1.2.2. Beneficios teóricos.

Dentro de los beneficios de implantar SDN frente a otras arquitecturas existentes, encontramos dos visiones: beneficios económicos y beneficios técnicos.

Dentro de los beneficios económicos destacamos:

- **Reducción del CAPEX.** Permite la reutilización de hardware existente gracias a las compatibilidades mencionadas y a la *SouthBound* API, reduciendo de este modo el gasto inicial en instalación de equipos nuevos.
- **Reducción del OPEX.** Ahorro en costes operativos debido a la mejora en la gestión que introduce una arquitectura SDN. La mejora se ve reflejada en un mejor orden logístico más centralizado, y en ahorro de tiempo de gestión por parte de los administradores.

Desde el punto de vista técnico ya se han mencionado algunas de las ventajas de SDN, siendo las más importantes las referidas a la agilidad y flexibilidad que se gana usando esta arquitectura. Además las posibilidades futuras de desarrollo están garantizadas gracias al uso de la NorthBound API ya que permitirá el desarrollo e innovación en nuevas aplicaciones para SDN sin necesidad de cambios en la estructura de la arquitectura.

2.1.2.3. Apoyo de la industria.

SDN cuenta con el apoyo de grandes compañías, que además han conseguido implementar arquitecturas basadas en SDN que reducen la complejidad y coste de sus redes cableadas. Existen varios ejemplos, entre ellos el de Google, que ha implementado *OpenFlow* para sus dos redes troncales, obteniendo resultados muy positivos, como se puede comprobar en [2].

Otros casos de éxito son los de Verizon [5] (despliegue de red SDN para servicios de vídeo distribuido o el caso de ATT [17] que tiene un plan para la mejora de su arquitectura a través de virtualización y uso de SDN.

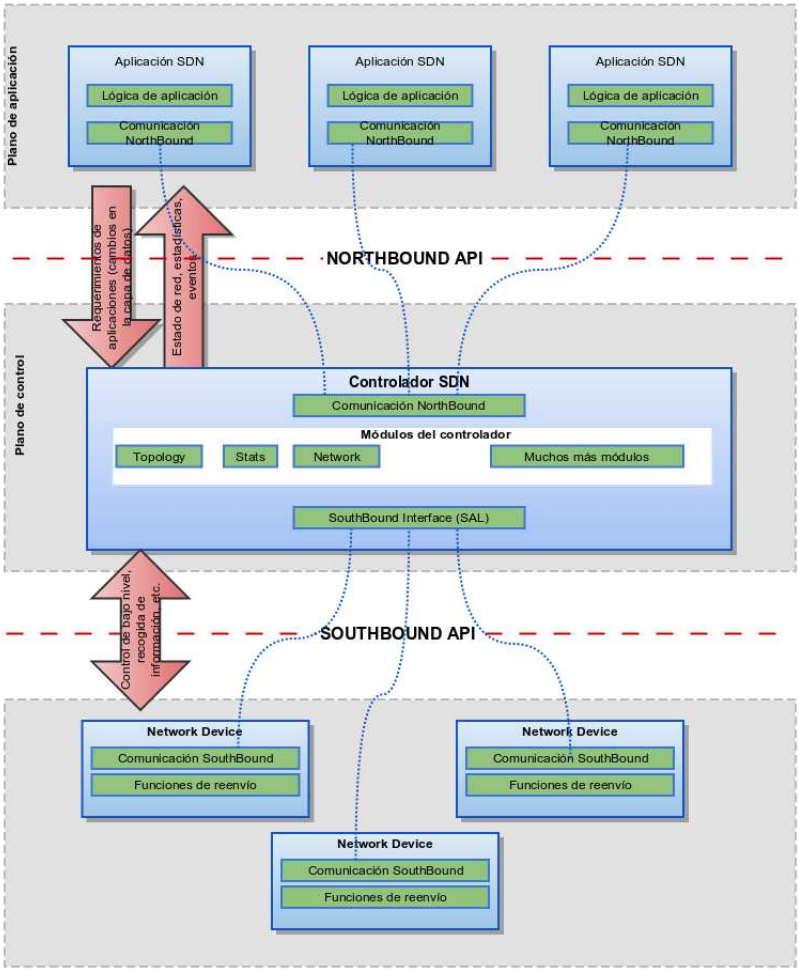


Figura 2.2: Visión esquemática de la estructura SDN. Imagen tomada de [18]

Estos son claros ejemplos del desarrollo y funcionalidad de SDN que justifican este trabajo. A continuación se describe la arquitectura que hace posible que SDN sea tan beneficiosa frente a otras soluciones.

2.1.2.4. Arquitectura y elementos en SDN.

En SDN todo se basa en el elemento central, el controlador. El controlador será el elemento que comunicará todas las capas mediante el uso de APIs. Para conseguir la separación de los planos de control y de datos, será necesario el uso de dos APIs. *Northbound* API (que comunica con la capa de aplicación) y *SouthBound* API (que comunica con los elementos al sur del controlador).

Para ver con más claridad como interactúan los elementos en SDN se incluye la figura 2.2, donde se puede comprobar perfectamente como todos los requerimientos de las aplicaciones del plano al norte, pasan por el controlador a través de la *Northbound* API, y que todas las instrucciones hacia elementos de red pasan por la *SouthBound* API (SDN Control-Data-Plane Interface en la imagen). Gracias al uso de la *SouthBound* API, el tipo de elemento de red sea indiferente para el controlador y las aplicaciones.

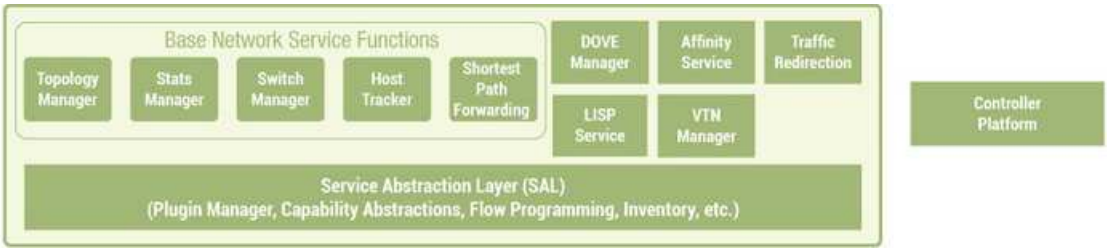


Figura 2.3: Capacidades del núcleo de *OpenDayLight*. Imagen extraída de `http://www.opendaylight.org/project/technical-overview`.

- **Controlador.** Como ya se ha comentado, el controlador SDN es el cerebro de la red. Este elemento se encuentra en el centro de todo y se comunica tanto con las aplicaciones (al norte) como con los elementos de transporte (*switches* y *routers*), al sur. Además en algunas soluciones se incluye la posibilidad de definir dominios de controladores SDN para hacer las redes SDN más escalables y manejables como se indica en [19].

  Un controlador SDN además estar capacitado para comunicarse a través de las APIs, suele incluir una serie de módulos que le permiten realizar diferentes tareas de red básicas: inventario de equipos conectados, colección de estadísticas de red u otras funciones. Además los proveedores de soluciones pueden añadir nuevas funciones en el núcleo del controlador según sus necesidades, siendo este uno de los puntos fuertes de la arquitectura SDN.

  Uno de los controladores más conocidos (y el que se usa en este proyecto) cuenta con los módulos que se muestran en la imagen 2.3, y que son la base a partir de la cual se pueden construir aplicaciones de red.

  SDN presenta además la posibilidad de aprovechar protocolos tradicionales de red. En este sentido están trabajando los miembros de uno de los grupos de IETF para conseguir que protocolos tradicionales (Open Shortest Path First (OSPF), MultiProtocol Label Switching (MPLS) o Border Gateway Protocol (BGP), entre otros) sean compatibles con SDN.

  Con esto damos por finalizado la parte correspondiente al controlador SDN, destacando las posibilidades que ofrece, tanto desde el punto de vista de novedosas funciones, como con la compatibilidad con protocolos antiguos. Las funciones y módulos que incluya un controlador SDN serán claves para el desarrollo de aplicaciones, como se comprobará más adelante.

- ***Northbound* y *SouthBound* APIs.** Son las interfaces que permiten al controlador comunicarse con los elementos al norte y sur de éste.

  ***SouthBound* API.** Es la encargada de posibilitar la comunicación del controlador con los elementos de red. Permite hacer cambios dinámicos en los elementos para adecuarse a las necesidades en tiempo real. Además permite la independencia de los elementos subyacentes por parte del controlador, ya que este solo se debe ocupar de enviar instrucciones que esta API traduce y transmite al elemento destino al que se dirige la instrucción. Existen varias soluciones para conseguir estas capacidades y la de mayor relevancia por el momento es el ya comentado *OpenFlow*.

***Northbound* API.** Esta API es usada para facilitar la innovación mediante la comunicación del core del controlador con nuevas APIs. Gracias a la arquitectura SDN las nuevas APIs podrán incluir innovaciones que añadan funciones de red más potentes. Esta API es crítica debido al gran número de aplicaciones de diferente procedencia que se soportan a través de ella, pudiendo producirse conflictos o errores entre las aplicaciones si no se respeta el modelo definido en cada caso específico (cada controlador hará uso de diferentes APIs).

Al ser un componente tan determinante para SDN, la ONF decidió crear un grupo de trabajado centrado en la *Northbound* API, el grupo *NorthBound Working Group*. Este grupo se encarga de escribir código para la API, desarrollar prototipos y mostrar claramente las capacidades de dicha API.

2.1.3. Grupos de trabajo SDN

SDN es una arquitectura en movimiento y que evoluciona de forma constante. Las mejoras pueden ser de dos tipos: mejoras en la arquitectura o API de SDN o mejoras en las aplicaciones que posteriormente se comunican con el núcleo SDN. Muchos grupos de trabajo están desarrollando e innovando sobre SDN, algunos de estos ya han sido citados. Los más destacados son:

- **ONF.** La *Open Networking Foundation* sigue trabajando en la definición de SDN y ha publicado la versión 1.1 de la arquitectura general SDN [18].

- ***ONF Wireless and Mobile Working Group.*** Este grupo de trabajo (también de la ONF) se creó con la intención de investigar la adaptación de SDN y *OpenFlow* para su uso en redes de acceso celular y en transporte inalámbrico. Es un grupo de trabajo orientado a una de las aplicaciones futuras de SDN, redes móviles 5G.

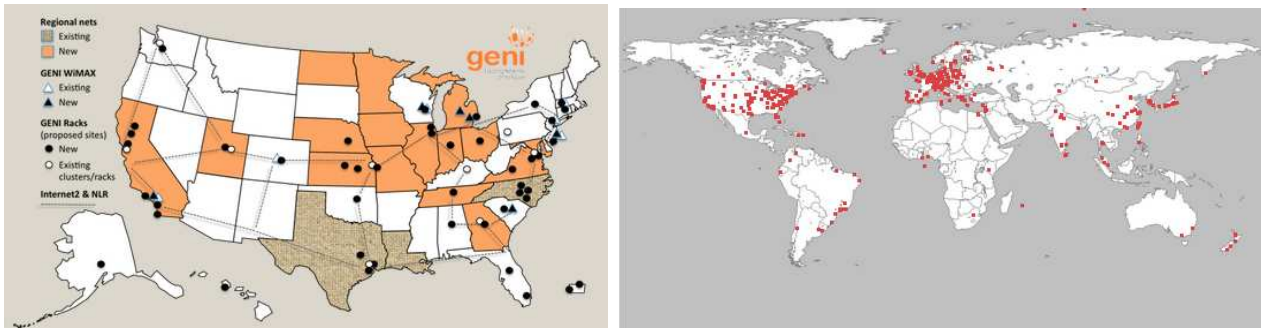
- **ONF *NorthBound Interface Working Group.*** Se ha hecho referencia a este grupo a lo largo de los apartados anteriores. Se encuentra enfocado en la investigación de aplicaciones SDN, y el estudio del funcionamiento y uso correcto de la *Northbound* API.

- **IETF *Software Driven Networks.*** Este grupo centra sus esfuerzos en el estudio de SDN dentro del marco IETF, completando huecos que consideran vacíos de la definición de SDN original. Proponen cambios, aportan soluciones y en definitiva estudian SDN desde otras perspectivas.

- **UC**[3] ***Software Defined Networking Activity Group.*** Este grupo de trabajo enfoca sus esfuerzos en proyectos de calidad de servicio y transmisión de datos multimedia sobre SDN. Los principales puntos de trabajo e innovación de este grupo son: estudio de casos de aplicación, análisis de capacidades SDN, definición de un framework para desarrollo SDN, definición de las APIs necesarias y definición de un programa de certificación SDN.

Estos son solo una pequeña parte de los grupos que en este momento se encuentran trabajando en la arquitectura SDN, quedando patente el interés generado por este nuevo paradigma de red.

[3]Unified Communications



(a) Red Geni. Imagen tomada de `http://www.geni.net/?page_id=2`

(b) Red PlanetLab. Imagen tomada de `https://www.planet-lab.org/`

Figura 2.4: Mapas de redes de experimentación.

2.2. *OpenFlow*

Antes de comenzar se debe destacar que SDN no es lo mismo que *OpenFlow*. *OpenFlow* es un estándar abierto y que SDN puede usar (existen alternativas) para gestionar la red. Lo que si se puede afirmar es que *OpenFlow* se está convirtiendo en el modelo estándar de implementación en redes SDN.

De acuerdo a la definición realizada por la ONF en [20], *OpenFlow* es la primera interfaz de comunicaciones definida entre las capas de control y de transporte en una arquitectura SDN. Permite el acceso directo y la manipulación de los elementos del plano de control, como *switches* o *routers*. Las tecnologías SDN basadas en *OpenFlow* están capacitadas para abordar el gran ancho de banda y la naturaleza dinámica de las aplicaciones actuales, permitiendo adaptar las redes a las necesidades en tiempo real, reduciendo significativamente las operaciones y la complejidad de gestión. En definitiva, *OpenFlow* posibilita casi todo aquello por lo cual las redes SDN son tan necesarias en el momento actual.

2.2.1. Historia y desarrollo

Los primeros pasos de OpenFLow están íntimamente ligados a los de SDN, como se ha comprobado en la sección referente a la historia de SDN, al final de la sección 2.1.1.

El crecimiento de las redes y el éxito de estas, presentó una gran oportunidad de trabajo para los investigadores (mayor importancia en este ámbito), sin embargo en este escenario realizar grandes avances o presentar resultados exitosos se antojaba complicado, ya que no existía ninguna manera práctica de experimentar y probar nuevos desarrollos en un entorno real o lo suficientemente cercano a la realidad para que pudieran considerar como válidos los resultados obtenidos. Como respuesta a esta necesidad surgieron diferentes propuestas. Por ejemplo la red Global Environment for Network Innovations (GENI)[4], serguió como una red pensada para facilitar la experimentación en nuevas arquitecturas de red mediante sistemas distribuidos. El concepto de funcionamiento se basaba en las redes programables que, usando virtualización, podían tratar paquetes de experimentos aislados entre sí y de forma simultánea.

[4] `http://www.geni.net/`

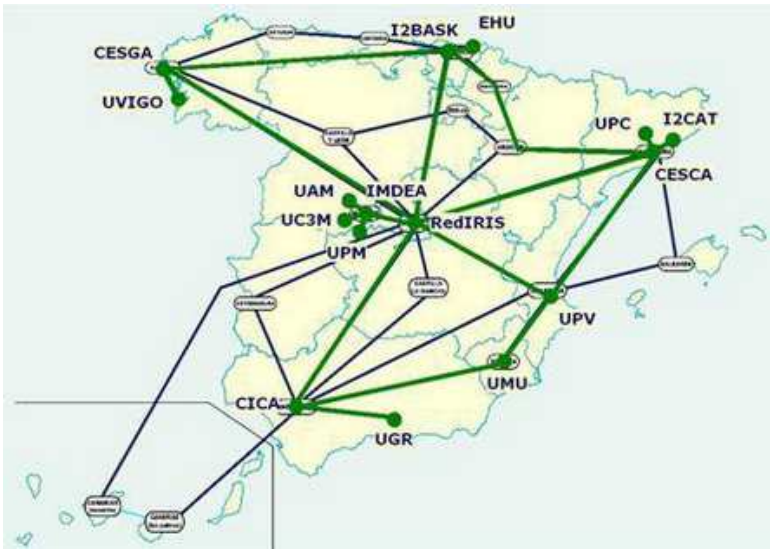


Figura 2.5: Mapa de Red PASITO. Imagen extraída de `http://www.rediris.es/jt/jt2006/RedIRIS10.jpg`

En España también surgieron redes pensadas para fomentar la innovación mediante la experimentación, como Plataforma de Análisis de Servicios de Telecomunicaciones (PASITO), un proyecto financiado por la secretaría del Estado, gestionado por Red.es y basado en la infraestructura de la Interconexión de Recursos InformáticoS (IRIS), para el desarrollo y la innovación sobre nuevos protocolos de Internet. El esquema de la red PASITO puede ser consultado en la figura 2.5.

A partir de estas redes surgieron numerosas posibilidades de desarrollo para la comunidad investigadora. En la universidad de Stanford se creó el Clean Slate Program que acabaría desembocando en el estándar *OpenFlow*. Pronto estas investigaciones llamaron la atención de importantes pesos pesados de la industria. Tanto es así que en la primera versión del estándar *OpenFlow* (versión 1.1, publicada el 28 de Febrero de 2011) estaban incluidos miembros de la talla de Google, Facebook o Microsoft, formando parte de este primer estándar. Posterior al lanzamiento de la primera versión, se hizo público que la ONF asumiría la responsabilidad de mantener el estándar a fin de conseguir una estandarización más global y eficaz.

Desde entonces hasta hoy se han ido presentando diferentes versiones del estándar *OpenFlow switch* hasta llegar a la más reciente (en Abril de 2015), la versión 1.5.1[5]. Por cuestiones de compatibilidad nosotros usaremos la versión 1.3.0 (versión más reciente soportada por *Mininet* [21]).

La gran acogida de *OpenFlow* no sólo se ve reflejada en el continuo proceso de innovación, también podemos contrastarla con el gran número de miembros adheridos a la ONF[6]. En total la ONF cuenta con más de 120 miembros, algunos de ellos de la importancia de Cisco, Google, Microsoft, Juniper, Brocade y otros muchos.

[5] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf`

[6] `https://www.opennetworking.org/our-members`

2.2.2. Alternativas. Redes programables.

Dentro de las alternativas (y a fin de no extendernos demasiado) vamos a destacar tan solo una de las más interesantes. El estándar OpFlex de Cisco, definido en [22]. Se destaca esta alternativa sobre el resto debido a la importancia de Cisco en el sector, y al trabajo ya realizado por esta compañía con el estándar *OpenFlow*.

- **Cisco OpFlex.** El objetivo de OpFlex es conseguir desarrollar un estándar que permita la aplicación de políticas de tráfico a través de *switches/routers* tanto físicos como virtuales, en un ecosistema con elementos de red de múltiples fabricantes.

  OpFlex usa una filosofía distinta a *OpenFlow* ya que opta por descentralizar la dependencia de red del controlador existente en *OpenFlow*. La justificación es que según los desarrolladores, el controlador actúa como cuello de botella en las redes programables. OpFlex propone la definición de políticas en el controlador, y el uso de este protocolo para comunicar estas políticas hacia los elementos de red. Estos elementos de red contarán con cierto nivel de inteligencia que les permitirá interpretar la política recibida por el controlador, dependiendo de la situación. Además se permite la comunicación bidireccional para recoger estadísticas, eventos y otra información que será usada para el ajuste de políticas.

  Para interpretar estas políticas será necesario un agente OpFlex embebido en los elementos de red para soportar el protocolo. Como resultado de esta necesidad, Cisco está trabajando para llevar OpFlex a la mayoría de plataformas del mercado. Microsoft, IBM, Red Hat, Canonical entre otras ya cuentan con agentes embebidos en sus sistemas para elementos de red.

OpFlex presenta ciertas ventajas frente a *OpenFlow*: virtualización o programabilidad de la red sin necesidad de comunicación del controlador. Por el contrario si enfocamos la comparativa con *OpenFlow* en torno a los costes, OpFlex sale perjudicado al necesitar de un agente embebido en cada elemento de red. Otro punto a favor de *OpenFlow* es el apoyo que de todos los miembros, mientras que OpFlex (aunque es abierto) aún cuenta con poco apoyo más allá del de la propia Cisco. Estas dos razones son muy importantes a la hora de elegir una solución u otra. Restaría realizar una comparativa de capacidades, aunque tanto *OpenFlow* como Opflex se encuentran en desarrollo, con lo que estas capacidades podrían variar (la primera versión OpFlex hasta la fecha fue lanzada en el último trimestre de 2014).

2.2.3. Funcionamiento *OpenFlow*

Openflow surge como solución a un problema. Separar distintos tipos de tráfico (producción e investigación) dentro de *switches* y *routers* pensados solo el transporte de datos. Uno de los objetivos es poder utilizar las tablas de flujo que ya implementan los *switches* para conseguir el objetivo. Tablas Network Address Traslation (NAT), firewall o QoS son comunes en todos los tipos de *switches*, y aunque dependan del tipo de fabricante, *OpenFlow* pretende explotar las características comunes entre todas las tablas. Con esta filosofía se permite la experimentación e innovación en redes globales, sin que el tráfico que circula por ellas se vea afectado.

Antes de comenzar con el desarrollo teórico del funcionamiento de *OpenFlow* se ha de resaltar que esta información está extraída de la especificación 1.3.5 [23] (por compatibilidad con otras herramientas empleadas), con lo que no se asegura que los conceptos en su totalidad se mantengan en futuras versiones de la especificación.

OpenFlow define dos elementos fundamentales: *switch* y Controlador.

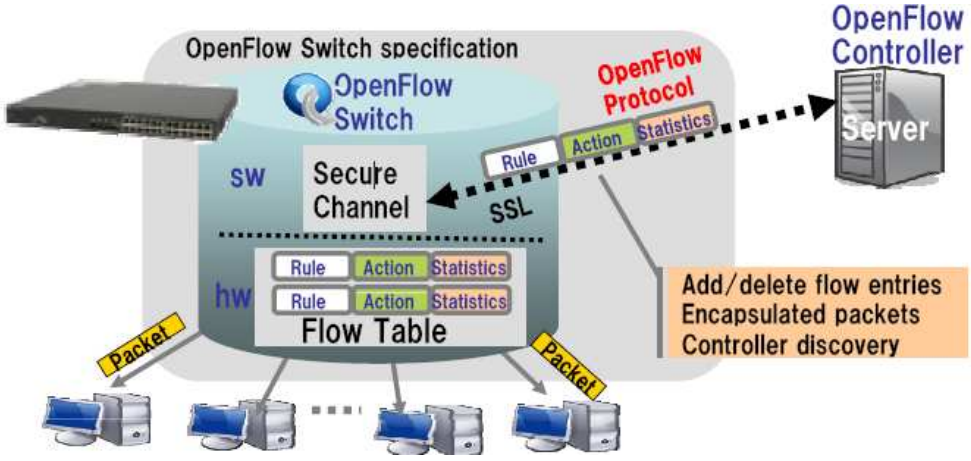


Figura 2.6: Esquema de comunicación *OpenFlow*. Imagen extraída de `https://ltgjamaica.wordpress.com/2012/04/29/software-defined-networking-first-look-at-openflow-30/`

- ***Switch.*** Encargado del procesamiento de paquetes de acuerdo a las reglas instaladas previamente por el controlador. Estas reglas son instaladas en la tabla de flujos del *switch*, que dependiendo del tipo *switch* (*OpenFlow-only* u *Openflow-hybrid*) serán tablas especificadas en el estándar *OpenFlow* directamente, o utilizará un mecanismo de clasificación no *OpenFlow*.

- **Controlador.** Elemento central de una red SDN y *OpenFlow*, capaz de evaluar el estado de red y añadir o eliminar flujos en los *switches OpenFlow*, de acuerdo a las aplicaciones instaladas en el controaldor. Está conectado a todos los *switches* que dependen de él, mediante un canal seguro establecido por el protocolo *OpenFlow*. Un controlador puede ser una simple aplicación ejecutándose en un PC que añade flujos de forma sistemática, o una aplicación mucho más compleja que reaccione de forma dinámica al estado de red, sin que esto suponga un cambio en el modo de trabajo de los elementos de red.

Una simplificación válida sobre el modo de operación de *OpenFlow*, es la que se aprecia en la imagen 2.6, donde a través del protocolo *OpenFlow*, el controlador se encarga de añadir, modificar o eliminar flujos. El *switch* se encarga de encaminar paquetes de acuerdo a las reglas de tráfico.

Para que esta comunicación se lleve a cabo será necesario hacer uso de múltiples elementos. Una breve descripción de todos estos conceptos puede ser consultada en el capítulo 3 (glosario) de [23]. A continuación se describen algunos de ellos.

2.2.3.1. Puertos *OpenFlow*

Son las interfaces de red definidas para el envío de paquetes entre *OpenFlow* y el resto de la red. Cada *switch OpenFlow* está conectado lógicamente (no necesariamente de forma física) con el resto, mediante sus puertos *OpenFlow*. Esta conexión lógica permite el envío de paquetes entre *switches OpenFlow*. Esta conexión debe respetar que el envío de paquetes de realice por los puertos *outPut*, y la recepcción por los puertos *ingress* de los *switches OpenFlow*.

Cuando se recibe un paquete, éste pasa por el proceso *pipeline* definido en 2.2.3.2. Este procesamiento *OpenFlow* decidirá que hacer con dicho paquete. Cada *switch OpenFlow* debe ser capaz de soportar tres tipos de puertos: físicos, lógicos y puertos reservados.

Los puertos físicos son aquellos que corresponden con una interfaz hardware física del *switch*. *OpenFlow* permite que los puertos físico realmente hagan uso de virtualización.

Otro tipo de puerto es el lógico. Este puerto no tiene porque corresponder con una interfaz física, sino que puede ser una abstracción de niveles más altos (por ejemplo para agregación de enlaces o túneles). Estos puertos deben poder encapsular paquetes (para identificación) y mapear varios puertos físicos por los cuales realizar acciones. El proceso de implementación de puertos lógicos debe ser transparente para *OpenFlow*, comunicándose con el protocolo como si un puerto físico se tratase.

A la hora de procesar solo existe una diferencia entre puertos físicos y lógicos, y es la inclusión de un campo extra llamado *Tunnel-ID* para identificación en los puertos lógicos. Por lo demás el procesamiento sigue el mismo flujo.

El tercer y último tipo de puerto corresponde a los reservados. Definidos por la especificación *OpenFlow* y usados para acciones de envío específicas, como transmisión de paquetes hacia el controlador, inundación de paquetes o reenvío usando métodos no *OpenFlow*. Dentro de los puertos reservados existen dos tipos: necesarios y no necesarios.

2.2.3.2. Tablas *OpenFlow*.

Las tablas *OpenFlow* son uno de los elementos más importantes del estándar, a continuación se describen los conceptos más importantes para poder trabajar con dichas tablas. La descripción completa puede ser encontrada en el capítulo 5 de [23].

- **Proceso *pipeLine*.** Este proceso es el que describe el mecanismo de *matching* y posterior toma de decisiones. Existen dos modos de trabajo diferentes dependiendo del tipo de *switches* (*Only-OpenFlow* u *OpenFlow-Hybrid*). Dado que vamos a trabajar con emulación de *switches OpenFlow* nos vamos a centrar en el proceso *Only-Openflow*.

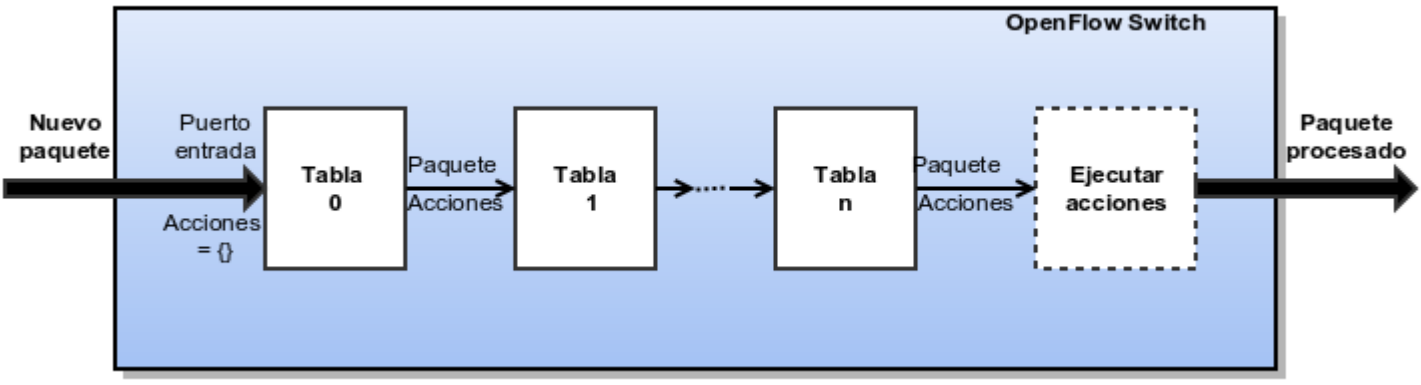
  El proceso *pipeline* de cada *switch OpenFlow* cuenta con una o más tablas de flujo, que a su vez contienen múltiples entradas de flujo. El proceso define como los paquetes interactúan con estas tablas de flujo de acuerdo al flujo de trabajo que se describe en la imagen 2.7. Un *switch* debe tener por tanto al menos una tabla de flujo (o equivalente si es un *switch* híbrido).

  Las tablas de flujo en un *switch OpenFlow* están enumeradas en orden ascendente, comenzando siempre por la tabla 0. Un paquete que comienza el proceso *pipeline* siempre comenzará siendo analizado de acuerdo a los flujos instalados en la tabla 0. Dependiendo del resultado obtenido en estaba tabla, el paquete continúa el proceso o finaliza.

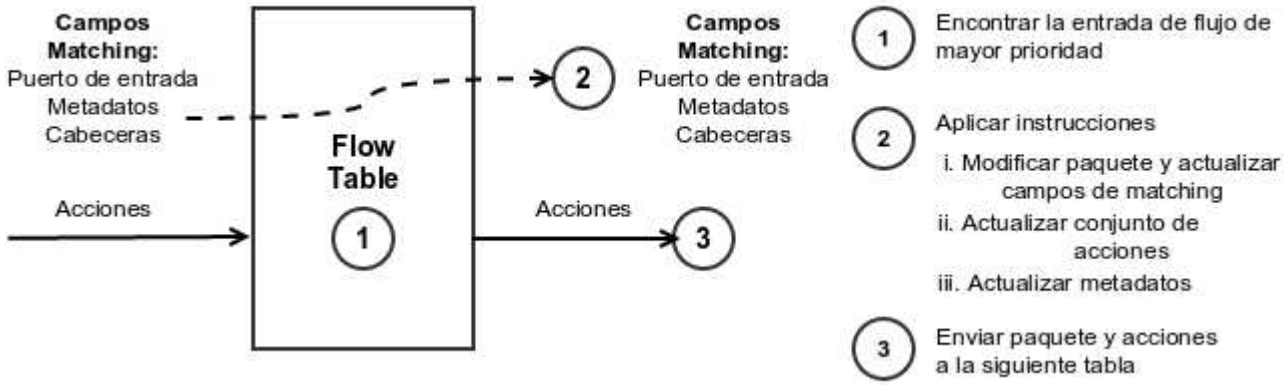
  El paquete procesado por una tabla de flujo siempre sigue el mismo flujo de operación:

  1. Se intenta realizar *matching* o emparejamiento del paquete con cada flujo en la tabla hasta encontrar una coincidencia, como se verá en 2.2.3.2. Hacer *matching* significa que todos los campos de un paquete coinciden con los valores especificados como *match* para un flujo.
  2. Si en la tabla no se ha encontrado ninguna entrada de flujo coincidente con el paquete, este pasará a la siguiente tabla. En caso de que si se encuentre dicha coincidencia, pueden darse dos situaciones.

     a) En el primer caso, una vez encontrada la coincidencia, el proceso *pipeline* se detiene en esta tabla, y se llevan a cabo las acciones definidas para los flujos coincidentes.



(a) Paso de paquetes por los múltiples *match* de las tablas de flujo



(b) Procesamiento de paquete en cada tabla

Figura 2.7: Esquema del procesamiento de paquetes a través del proceso *pipeline*.

b) En el segundo caso, si la entrada de flujo tiene definida una acción *Goto-Table*, el paquete debe seguir siendo procesado en otra tabla. En este segundo caso se requiere que siempre que se mande un paquete para ser procesado a otra tabla, el orden de esta tabla sea superior al de la primera (para evitar bucles).

3. En caso de que un paquete no encuentre ningún *match* en las tablas, es traspasado a una tabla denominada *missing table*, que decide que hacer con estos paquetes. Las opciones disponibles son: empezar de nuevo el *matching*, pero ignorando algunos campos para hacerlo más flexible, inundar el paquete en la red o mandarlo al controlador a través del canal de control.

Existen dos casos especiales para el procesamiento de paquetes: si no existe tabla *missing table*, los paquetes son directamente descartados. El segundo caso se da cuando los paquetes tienen un Time To Life (TTL) inválido, en esta caso son enviados al controlador.

- **Entradas de flujo.** Una tabla de flujo está compuesta por entradas de flujo que permitirán aplicar acciones para paquetes que coincidan con los *match* especificados. Una entrada de flujo consiste en los campos que se recogen en la tabla 2.1.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Tabla 2.1: Campos de una entrada de flujo.

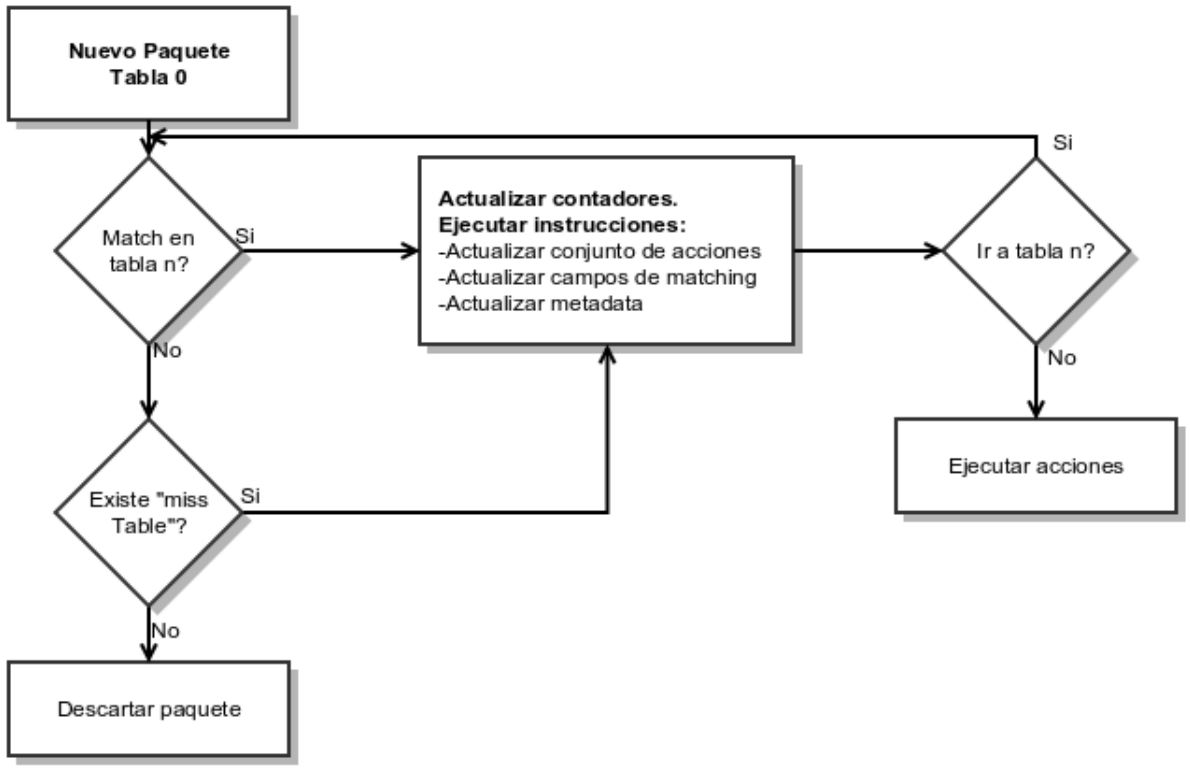


Figura 2.8: Procesamiento de paquetes para el *matching*.

Estos campos se encuentran definidos en [23]. Los más importantes para nosotros serán: campos *Match Fields* (permiten detectar los diferentes flujos, RTP, Transmission Control Protocol (TCP), etc.), *instructions* (indica que se hará con un paquete que coincida con el *matching*) y *timeouts* (definen el tiempo de validez para la entrada de flujo, *idle* o *hard*). Cada entrada de una tabla de flujo está identificada por la prioridad, y el *match* asociado. En caso de estar disponible en la implementación, existe una entrada con prioridad 0 y los campos del matching vacíos, esta es la conocida como la *missing table*.

- ***Matching.*** El proceso de *matching* se lleva a cabo para paquete que llega al *switch*. Está descrito en la figura 2.8 y consiste en consultar cada una de las tablas (por orden) y en función de los resultados obtenidos aplicar las instrucciones definidas.

  Podemos decir que existe *match* para un paquete siempre y cuando todos los campos del *match* definido, sean cumplidos por el paquete, sino el éste continuará el procesamiento. Por defecto los *swicthes OpenFlow* traen consigo una serie de campos implementados, pero para cumplir con la especificación solo se requieren los recogidos en la tabla 2.2.

  Existe la posibilidad de usar un mayor número de campos o incluir nuevos campos según las necesidades. Por ejemplo entre los campos por defecto se incluyen además algunos que permiten detección directa de etiquetas MPLS.

Campo	Descripción
OXM_OF_IN_PORT	Puerto de entrada. Puerto físico o lógico del *swtich*
OXM_OF_ETH_DST	Dirección Ethernet de destino. Puede usar cualquier máscara
OXM_OF_ETH_SRC	Dirección Ethernet de origen. Puede usar cualquier máscara
OXM_OF_ETH_TYPE	Tipo de paquete Ethernet, tras las etiquetas VLAN
OXM_OF_IP_PROTO	Protocolo IPv4 o IPv6
OXM_OF_IPV4_SRC	Dirección origen IPv4. Puede usar máscara de subred o no
OXM_OF_IPV4_DST	Dirección destino IPv4. Puede usar máscara de subred o no
OXM_OF_IPV6_SRC	Dirección origen IPv6. Puede usar máscara de subred o no
OXM_OF_IPV6_DST	Dirección destino IPv6. Puede usar máscara de subred o no
OXM_OF_TCP_SRC	Puerto origen TCP
OXM_OF_TCP_DST	Puerto destino TCP
OXM_OF_UDP_SRC	Puerto origen Usser Datagram Protocol (UDP)
OXM_OF_UDP_DST	Puerto destino UDP

Tabla 2.2: *Match Fields* requeridos por *OpenFlow*.

Para más información sobre el proceso de *matching* y ampliar detalles sobre los campos para *match*, se recomienda consultar la especificación [23], capítulos 5 y 7 respectivamente.

- **Contadores.** Los contadores están asociados a cada tabla de flujo, entrada de flujo, puerto, cola u otros elementos. Existe un gran número de contadores posibles, pero *OpenFlow* solo obliga el mantenimiento de algunos, (aunque existen muchos más opcionales). Los contadores obligatorios y de mayor relevancia son los relaciones con el número de entradas (para las tablas). La duración de cada entrada de flujo, paquetes transmitidos y paquetes recibidos por cada puerto.

- **Instrucciones.** Las instrucciones son ejecutadas cuando un paquete finaliza el proceso de *matching* de forma satisfactoria para una entrada de flujo. *OpenFlow* especifica una serie de instrucciones que han de ser soportadas por los *switches*, dejando otras instrucciones como opcionales. Las instrucciones más importantes son aquellas que escriben o borran acciones para cada entrada de flujo, y la instrucción que indica *Goto-Table*, para saltos entre tablas.

- ***TimeOuts.*** Los *timeOuts* están definidos para cada entrada de flujo. Existen dos tipos.

  - *IdleTimeOut*, es de tipo *soft* y permite que un flujo sea eliminado de la tabla de flujos tras la ausencia de paquetes coincidentes con el flujo durante el periodo especificado. Cada vez que un paquete hace *match* con esa entrada el contador de tiempo se reinicia.
  - El segundo tipo es *HardTimeOut*, este borrará el flujo siempre que pase el periodo de tiempo establecido, ocurra lo que ocurra (siempre y cuando el controlador no envíe una nueva instrucción que modifique el tiempo o comportamiento).

2.2.3.3. Acciones.

Cada vez que un paquete encuentra una entrada de flujo coincidente, se le asignan una o más acciones. En la especificación están recogidas una serie de acciones (no todas tienen que ser soportadas por los *switches*), de las cuales las más importantes para nosotros son las relacionadas con reenvío y descarte de paquetes.

- ***Output.*** Esta acción está asociada al reenvío de paquetes por un puerto *OpenFlow*.

- ***Drop.*** Cuando a un paquete se le asocia una acción *drop* tras el procesamiento, éste será descartado.

- ***Group.*** Esta acción conlleva el procesamiento del paquete como si fuera de un determinado grupo (consultar Type groupe [23]), permitiendo el procesamiento por grupos de paquetes. Esta definición está diseñada para aumentar la eficiencia de procesamiento.

2.2.3.4. *OpenFlow Channel* y *Control Channel.*

El canal *OpenFlow* es el que conecta lógicamente los *switches OpenFlow* con el controlador. A través de este canal, el controlador configurará y administrará cada *switch* (recibiendo paquetes o enviando órdenes). El canal de control *OpenFlow* puede estar constituido por un solo canal *OpenFlow*, conectado a un controlador, o por varios canales *OpenFlow*, conectados a varios controladores que comparten la administración del *switch*.

El canal *OpenFlow* exige que todos los mensajes que sean transportados por él, se envíen de acuerdo a la especificación *OpenFlow*, para conseguir la máxima compatibilidad. Este canal suele estar encriptado usando Transport Layer Security (TLS), aunque también existe la posibilidad de enviar los paquetes directamente sobre TCP.

El análisis de intercambio de mensajes no tiene cabida en este trabajo, pues no se pretende analizar el funcionamiento interno de *OpenFlow*, sino hacer uso de sus capacidades. Sin embargo es importante destacar que la comunicación entre *switch* y controlador, no tiene porque ser iniciada por ninguna de las dos partes. Por ejemplo un *switch* puede enviar mensajes de forma asíncrona para indicar eventos, sin que estos hayan sido solicitados por el controlador. También pueden existir mensajes simétricos de Hello o Echo por cualquiera de las dos partes, para mantener el canal de comunicación.

El capítulo 6 de [23] contiene todos los mensajes contemplados en el protocolo *OpenFlow* versión 1.3.5 (26 de Marzo de 2015), así como las posibilidades de comunicaciń entre ellos.

Con esto se da por concluido el apartado referente a *OpenFlow*. Se ha pretendido dar las claves de funcionamiento *OpenFlow*. También se ha intentado razonar y justificar el uso de *OpenFlow* como estándar de referencia para redes SDN, y haber descrito las claves teóricas que posteriormente nos permitirán desarrollar y alcanzar los objetivos de este trabajo.

2.3. Controladores SDN

Uno de los elementos troncales de la arquitectura SDN es el controlador, cerebro de toda la arquitectura de red que de él depende. Podemos dividir las diferentes alternativas actuales en dos grandes grupos: controladores de código abierto y controladores comerciales.

Para el desarrollador independiente, los más interesantes son los controladores abiertos, ya que en ellos podrá desarrollar libremente y contará con el apoyo de la comunidad. Algunos de los controladores comerciales también usan una filosofía de código abierto, con lo que la elección de uno frente a otros será "complicada".

En este trabajo se requiere el uso de controladores de código abierto que permitan el desarrollo sin coste alguno, se encuentren en desarrollo o en fases finales, y además cuenten con un fuerte apoyo de la comunidad, que permita resolver los problemas que se presenten.

Todas estas razones nos llevan a elegir un controlador de tipo abierto. Sin embargo a continuación se presentan las principales opciones comerciales para evaluar sus capacidades.

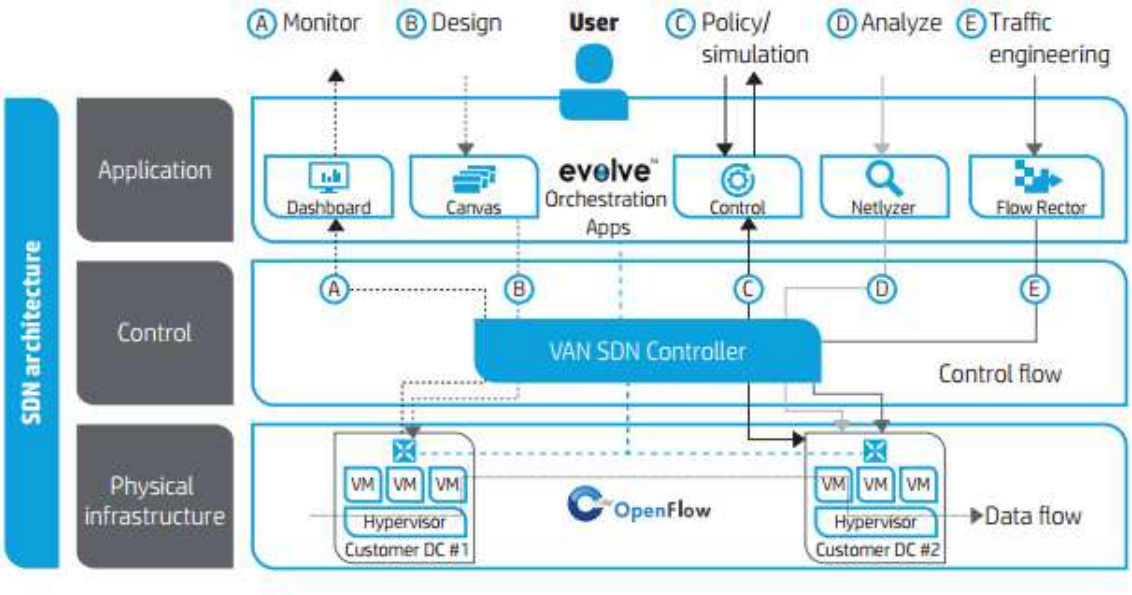


Figura 2.9: Arquitectura del controlador VAN. Imagen tomada de `https://sreeninet.wordpress.com/2014/08/09/sdn-openflow-commercial-applications-part-2/`

2.3.1. Controladores comerciales.

Este tipo de controladores se encuentran desarrollo (aunque algunos ya son encontrados en soluciones comerciales ofertadas). Surgen como respuesta a la necesidad de un desarrollo rápido y privado frente a los controladores abiertos. Empresas como Cisco y NEC por ejemplo son miembros importantes de algunos de los controladores de código abierto más importantes (como *OpenDayLight*), pero también desarrollan sus propias soluciones comerciales. A continuación se describen brevemente algunas de estas distribuciones comerciales.

- **Application Policy Infrastructure Controller (APIC).** [24]. Si se presentó OpFlex como alternativa a *OpenFlow*, se debe tratar el controlador paraeste paradigma diseñado por Cisco. En una infraestructura de red centralizada, APIC actúa como un único punto de control central. Proporciona una API central, un repositorio central de datos globales y un repositorio de políticas (consultar la sección referida a alternativas *OpenFlow* 2.2.2). APIC trabaja de forma diferente al resto de controladores SDN *OpenFlow*, ya que no se encarga de encaminar cada flujo de datos, sino que según el estado de red, instala políticas en los elementos de red, y serán estos los que se encarguen de tomar decisiones en función de estas políticas.

- **Virtual Application Networks (VAN).** [25]. Es el controlador desarrollado por *HP*. Este si que es un controlador desarrollado sobre *OpenFlow*. Sin entrar demasiado en el funcionamiento interno del controlador, podemos destacar que en el desarrollo de éste está centrado en: incluir seguridad en las comunicaciones del controlador con el resto de la red, administración centralizada, capacidad de automatización y otras capacidades inherentes a redes SDN. El esquema que presenta VAN no es diferente del usado por *OpenDayLight* u otros controladores, como se puede comprobar en la imagen 2.9.

- **VMware NSX.** NSX es un controlador desarrollado para la virtualización de redes. Este controlador permite tratar redes físicas como máquinas virtuales, sin importar que implementación física exista debajo. Además integra todas las capacidades propias de virtualización de redes, permitiendo a los clientes obtener todo el potencial de un centro de procesamiento de datos sin necesidad de redes físicas. Además cuenta con propiedades características de redes SDN, como la automatización o la naturaleza dinámica y ágil de éstas.

Estas son tres soluciones comerciales SDN, pero existen muchas más, que sin duda serán capaces de satisfacer las necesidades de los clientes a los que están dirigidas.

A continuación ponemos el foco en los controladores de código abierto.

2.3.2. Controladores de código abierto.

Dentro de los controladores de código abierto existen multitud de alternativas, algunas en fases de desarrollo muy tempranas y novedosas, y otras que ya son referentes en el mercado. En esta sección vamos a describir algunas de ellas apoyándonos en los resultados y conclusiones obtenidas en los artículos [26] y [27].

No podemos describir todas las alternativas disponibles así que nos hemos centrado en aquellas que consideramos conocidas y que aún se mantienen en desarrollo.

- **POX.** [28]. POX es un controlador hermano de uno de los pioneros, NOX[7]. POX surge como un intento para conseguir que aquellos usuarios que comienzan con SDN se encuentren con un entorno de desarrollo amigable. Este controlador se ha desarrollado sobre Python. POX como controlador que cuenta con dos métodos de desarrollo: el primero es una API basada en Python, el segundo método está basado en una API vía web que hace uso de JSON-RPC. Por último destacar que además cuenta con una interfaz gráfica vía web (como la mayoría de controladores) desde la cual se puede monitorizar la red y realizar otras operaciones. Como punto negativo hay mencionar que POX no cuenta con demasiado información ni manuales disponibles, siendo algo a valorar por aquellos que deseen comenzar con él.

- **Ryu.** [29]. Este controlador se define a si mismo como un proveedor de componentes software, con una API muy bien definida que permite a los desarrolladores crear nuevas aplicaciones de administración y control de forma sencilla. Uno de los puntos fuertes de Ryu es el soporte de varios protocolos para la administración de equipos, no solo *OpenFlow*. Ryu está desarrollado sobre Python. Como punto negativo debemos destacar la dependencia del S.O. sobre el que se ejecuta (solo se soporta Línux) así como la baja actividad que presentan sus *mailing list*[8], en comparación con las de ODL. Ryu cuenta con una *wiki* propia, con suficiente documentación como para considerarla de nivel medio, según [26]. Otro punto que se debe tener en cuenta es que Ryu no depende de miembros externos (como si *OpenDayLight*), con lo que se gana en agilidad y poder de decisión (de los usuarios) pero se pierde soporte y apoyo.

[7] `http://www.noxrepo.org/`
[8] `http://sourceforge.net/p/ryu/mailman/ryu-devel/`

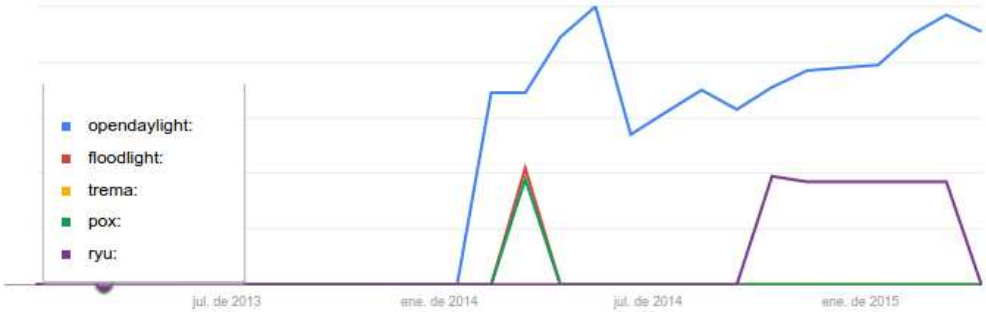


Figura 2.10: Interés generado por los diferentes controladores en Google entre 2013 y 2015. Datos tomados de GooGle Trends. Consultado en mayo 2015

- **Trema.** [30]. Más que un controlador en si, Trema presenta un *framework* de fácil uso, donde desarrollar tu propio controlador *OpenFlow* desde 0. Está escrito en Ruby y C, y pretende ser una alternativa real a controladores en fases de desarrollo avanzadas. Trema puede ser una opción para aquellos que deseen llevar a cabo un proyecto muy específico pues solo se implementarán los módulos necesarios (ganando en eficiencia). Sin embargo el trabajo por parte de los desarrolladores se verá aumentado al tener que diseñar un controlador completo. Trema no cuenta con interfaz gráfica (tendría que ser desarrollada para cada caso) y tampoco tiene soporte para Representational State Transfer (REST) API, capacidad muy demandada por la comunidad de usuarios. La documentación y flexibilidad que podemos encontrar por parte de Trema es buena, sin embargo en sus listas de correo el último tema abierto es del 3 de Marzo de 2015[9], no siendo un dato muy alentador para aquellos que decidan iniciarse con un controlador SDN.

- **FloodLight.** [31] FloodLight se puede presentar como la evolución del controlador Beacon (uno de los primeros controladores SDN que surgieron). Está desarrollado sobre JAVA y al igual que la gran mayoría de controladores posee una interfaz web. Además soporta el uso de API REST. Otro aspecto importante de Floodlight es la importante comunidad que lo apoya. Los desarrolladores participan de forma muy activa en la lista de correo, con lo que cualquier duda que pueda surgir para nuevos usuarios estará ya resuelta o se resolverá de forma rápida, lo cual como ya se ha comentado es imprescindible. Sin embargo presenta algunos inconvenientes, por ejemplo la dependencia de desarrollo de API REST. Además la relevancia de FloodLight (pese a contar con apoyos importantes como el de *C*anonical) ha ido perdiendo peso y apenas es relevante frente a ODL.

Uno de los aspectos más importantes para la decisión sobre que controlador usar es la relevancia actual de éste. En la imagen 2.10 se ve claramente la evolución de los distintos controladores presentados. Los datos que se presentan en la imagen 2.10 están limitados a términos de búsqueda relacionados con Redes, para garantizar la validez y concordancia de estos datos con el ámbito en el que trabajamos.

Por último se presenta en la tabla 2.3 un resumen sobre los controladores aquí presentados, incluyendo además el controlador *OpenDayLight* que se describe en la sección 2.3.3.

[9]Consultado el día 25/05/2015

	POX	Ryu	Trema	FloodLight	OpenDaylight
Interfaces	SB (OpenFlow)	SB (OpenFlow) +SB Management (OVSDB JSON)	SB (OpenFlow)	SB (OpenFlow) NB (Java & REST)	SB (OpenFlow & Others SB Protocols) NB (REST & Java RPC)
Virtualization	Mininet & Open vSwitch	Mininet & Open vSwitch	Built-in Emulation Virtual Tool	Mininet & Open vSwitch	Mininet & Open vSwitch
GUI	Yes	Yes (Initial Phase)	No	Web UI (Using REST)	Yes
REST API	No	Yes (For SB Interface only)	No	Yes	Yes
Productivity	Medium	Medium	High	Medium	Medium
Open Source	Yes	Yes	Yes	Yes	Yes
Documentation	Poor	Medium	Medium	Good	Medium
Language Support	Python	Python-Specific + Message Passing Reference	C/Ruby	Java + Any language that uses REST	Java
Modularity	Medium	Medium	Medium	High	High
Platform Support	Linux, Mac OS, and Windows	Most Supported on Linux	Linux Only	Linux, Mac & Windows	Linux
TLS Support	Yes	Yes	Yes	Yes	Yes
Age	1 year	1 year	2 years	2 years	2 Month
OpenFlow Support	OF v1.0	OF v1.0 v2.0 v3.0 & Nicira Extensions	OF v1.0	OF v1.0	OF v1.0
OpenStack Networking (Quantum)	NO	Strong	Weak	Medium	Medium

Tabla 2.3: Resumen controladores SDN. Tabla tomada de [26].

2.3.3. *OpenDayLight*

ODL es el controlador elegido para el estudio y desarrollo de los objetivos marcados en este trabajo. A lo largo de la sección anterior se han descrotp algunos de los puntos débiles del resto de controladores comparados, a fin de justificar el uso de *OpenDayLight*. A modo de resumen se enumeran a continuación las principales fortalezas de ODL:

1. Apoyo de la industria con más de 50 miembros implicados en el proyecto, algunos de ellos de gran relevancia como los que aparecen en la imagen 2.11.

2. Proyecto bajo la protección de la Línux Foundation y totalmente de código abierto.

3. Desarrollado totalmente en JAVA, con posibilidades de desarrollo de aplicaciones vía varias tecnologías (API REST, JAVA, Document Object Model (DOM). API).

4. Documentación extensa (aunque con deficiencias aún[10] para aquellos que no están familiarizados). Además cuenta con una comunidad muy activa y con un gran número de usuarios.

5. Proyecto en continuo desarrollo[11].

[10] A fecha 15/06/2015.

[11] Última versión estable *Lithium* lanzada el 29/06/2015.

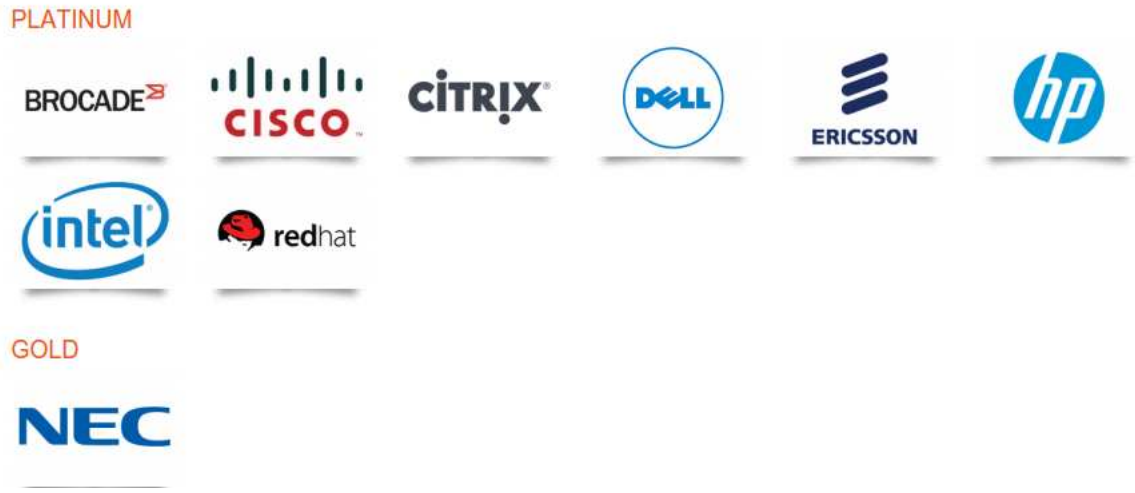


Figura 2.11: Miembros platino y oro del proyecto *OpenDayLight*. Imagen extraída de `http://www.opendaylight.org/`

Estas son solo algunas de las razones que justifican el uso de *OpenDayLight*, pero no todas, estamos obviando capacidades más técnicas que se irán desgranando a lo largo de esta sección.

2.3.3.1. Historia y desarrollo.

OpenDayLight surgió con el objetivo de acelerar la adopción de SDN, y crear una base sólida para las funciones de virtualización de red mediante, la colaboración de los pesos pesados del sector. El 8 de Febrero de 2013 se presentó en *SDN Central*[12] un consorcio de grandes empresas, bajo una estructura de fundación *open-source*. Posteriormente en Abril de 2013 (coincidiendo con el Open Networking Summit (ONS) de esa fecha), la Línux Foundation anunció la fundación de *OpenDayLight* como proyecto abierto, con el fin de acelerar la adopción, fomentar la innovación y crear un enfoque abierto y transparente. Dentro de los miembros fundadores se encontraban compañías de gran importancia, tales como Arista Networks, Big Switch, Brocade, Cisco, Citrix, Ericsson, HP, IBM, Juniper Networks, Microsoft, NEC, Nuage Networks, PLUMgrid, Red Hat y VMware.

Ante este anuncio las reacciones no fueron todo lo positivas como se podía esperar. Por un lado se encontraron aquellas voces que reaccionaron muy positivamente ante los objetivos del proyecto y su estructura abierta. Sin embargo también se dieron opiniones críticas en torno a los miembros del grupo, ya que muchos de estos no desarrollan ni venden hardware relacionado con este tipo de redes, con lo cual se desconfió de los objetivos del grupo, ya que miembros como Microsoft, VMware o Citrix entre otros no son considerados vendedores "históricosçomo si lo son Cisco o Juniper.

La primera versión de ODL (*Hydrogen*) fue lanzada en Febrero de 2014, incluyendo un controlador de código abierto, capacidades de virtualización y algunos *plugins*. Posteriormente han aparecido distintas versiones que incluían nuevas funcionalidades y mejoras. En Noviembre de 2014 se lanzó una nueva versión que incluía un entorno de trabajo mucho más amigable, basado en *Apache Karaf* y que se constituyó como *Helium*. La última versión estable es la ya mencionada SR3 que mantiene la estructura del proyecto *Helium*[13].

[12] `https://www.sdxcentral.com/articles/news/spotlight-on-daylight-sdn-consortium-open-source-controller/2013/02/`

[13] El 29 de Junio de 2019, se lanza una versión Lithium, la cual no ha podido ser analizada para esta memoria por el calendario de entrega.

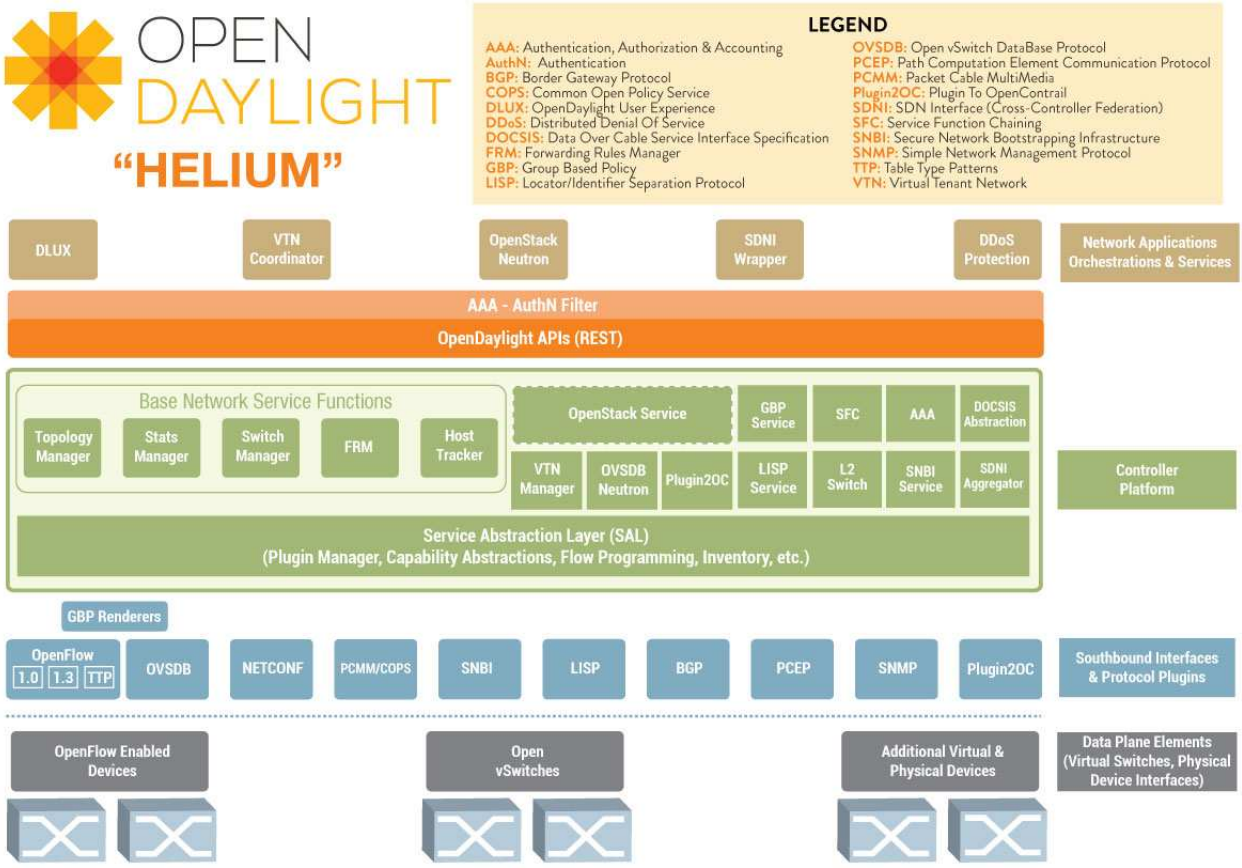


Figura 2.12: Vista técnica del controlador *OpenDayLight*. Imagen tomada de `http://www.opendaylight.org/project/technical-overview`.

Antes de finalizar, se debe destacar que cualquier cambio en *OpenDayLight* está sujeto a una votación por parte del comité de dirección técnica. El comité está limitado con un asiento/voto por cada miembro del proyecto, pero no todos los miembros tienen voto. Todos los miembros platino tienen un asiento en la junta independientemente de su colaboración, del resto de miembros, solo aquellos que son considerados importantes, de calidad o con proyectos centrados *OpenDayLight*, serán considerados para obtener un asiento (y por tanto voto) durante el año siguiente al nombramiento. Como conclusión se puede decir por tanto que *OpenDayLight* se asegura su independencia de un solo fabricante hardware. Sin embargo también es cierto que la mayoría del poder (votos) recae sobre los miembros platino.

2.3.3.2. Diseño

En la imagen 2.12 podemos encontrar la arquitectura del controlador en su última versión cuya estructura se mantiene como en la versión Hydrogen, aunque se añaden más módulos al núcleo del controlador y la capa de seguridad Authentication, Authorization and Accounting (AAA). Se distinguen tres capas en esta arquitectura:

1. **Aplicaciones de red e instrumentación.** En la capa de mayor nivel se encuentran las aplicaciones que se encargan del control y monitorización de la red subyacente. También se pueden encontrar en esta capa, soluciones que hagan uso de computación en la nube o servicios de virtualización de red. En esta capa podríamos decir que se encuentran aquellas aplicaciones que computan la ingeniería de tráfico de la red.

2. **Controlador.** Es la capa central y donde se manifiestan las abstracciones de SDN. El controlador cuenta con una serie de módulos implementados que permiten a las aplicaciones de la capa superior obtener datos e información sobre el estado de la red. Además cuenta con una serie de APIs que permiten el desarrollo de las aplicaciones superiores (DOM API, REST API o JAVA), mientras se implementan varios protocolos que permiten la comunicación con los elementos de red inferiores (*OpenFlow*, BGP, Path Computation Element Protocol (PCEP), Simple Network Management Protocol (SNMP) entre otros).

3. **Elementos de red físicos y virtuales.** La capa inferior está constituida por aquellos elementos de red que son programables mediante los protocolos implementados por el controlador. Gracias a la capa de abstracción del controlador, se pretende que todos los elementos (de cualquier fabricante) sean compatibles con el controlador *OpenDayLight*.

El controlador ODL está implementado en software, haciendo uso de su propia máquina virtual de JAVA. Por tanto su ejecución es independiente del SO (siempre y cuando los SO soporten JAVA), siendo compatible con la mayoría de SO del mundo.

El desarrollo de aplicaciones para ODL puede ser llevado a cabo por varias vías gracias a la *Northbound* API. Por un lado el controlador soporta el desarrollo mediante el uso del framework Open Services Gateway Initiative (OSGi)[14]. También soporta la comunicación bidireccional usando la REST API. OSGi será tratado con mayor profundidad en la sección 5. El uso de la API vía web permite ejecutar aplicaciones que no se encuentran en el mismo dominio de direcciones que el controlador, mientras que OSGi se usa para aquellas aplicaciones que si están en el mismo dominio de direcciones. Haciendo uso de cualquiera de los dos métodos, las aplicaciones usan el controlador para recabar información sobre el estado de red, monitorización, etc. Con esta información ejecutan algoritmos inteligentes que posteriormente les permitirán (mediante el uso del controlador de nuevo), la instalación de nuevas reglas para la red subyacente.

En el núcleo del controlador se encuentran una serie de módulos que posibilitan la recolección de estadísticas, visión global de red, estudio de capacidades, etc. Estos módulos son los que permiten a las aplicaciones superiores tener información actualizada para hacer uso de los algoritmos inteligentes. Aparte de estos servicios en el núcleo del controlador, también se pueden encontrar servicios orientados a seguridad y otras extensiones que permiten un mejor control.

Por último destacar que el controlador es totalmente independiente de los elementos de red y protocolos inferiores gracias al uso de la capa de Service Abstraction Layer (SAL), que se encarga de exponer trasladar las peticiones de las aplicaciones de capas superiores hacia abajo, transformando estas peticiones dependiendo del protocolo de comunicación con los dispositivos de red (ya sea *OpenFlow*, BGP-LS o el resto de los soportados).

Se ha descrito brevemente funcionamiento de *OpenDayLight* y como se debe encarar el desarrollo de aplicaciones posteriores a fin de conseguir los objetivos propuestos y realizar un estudio sobre la funcionalidad y capacidad de redes SDN.

En la sección 5 se tratará con mayor profundidad el modo de desarrollo sobre ODL y el funcionamiento interno de este.

[14] `http://www.osgi.org/Main/HomePage`

Con esta sección se da por terminado el capítulo dedicado al estado del arte, donde hemos realizado un estudio teórico de las tecnologías implicadas en este trabajo, a fin de conseguir una mejor comprensión que posteriormente permita realizar el desarrollo marcado en los objetivos.

Capítulo 3

Planificación y costes

Este capítulo describe la planificación temporal y de costes para la realización de este proyecto. La planificación intentará acoger todas las fases de preparación, desarrollo y finalización del proyecto, con el objetivo de al menos igualar la carga de trabajo especificada para la realización de un proyecto de estas características. De igual modo se ha intentado eliminar en la planificación de costes todos aquellos costes prescindibles.

3.1. Planificación temporal

Para la planificación temporal se ha optado por dividir la carga de trabajo en las siguientes actividades.

1. Revisión bibliográfica. El objetivo de esta primera fase consistirá en la recopilación y comprensión de toda la información que pueda ser de utilidad para el trabajo. Se pretenden adquirir y recopilar aquellos conocimientos útiles para el desarrollo del trabajo. Por ejemplo, en esta fase se intentará adquirir toda la información posible sobre redes SDN.

2. Familiarización con el entorno de trabajo. Esta segunda fase se ha diseñado para aglutinar todo el trabajo previo a la implementación del sistema. Por ejemplo, tiempo de configuración del sistema operativo, primeros pasos con mininet o familiarización con éste.

3. Familiarización con el entorno de desarrollo. Para la tercera fase del proyecto se han escogido las tareas de preparación previa a la implementación de la solución. Estudiar las alternativas para la programación de aplicaciones para el controlador, elección de herramientas de desarrollo y ejecución de ejemplos.

4. Diseño de la solución. Durante el desarrollo de esta fase se pretende diseñar una solución óptima para el problema propuesto, describiendo exhaustivamente los módulos en los cuales se dividirá la solución.

5. Descripción de protocolos y algoritmo de encaminamiento. Fase diseñada para el estudio de los protocolos que posteriormente encaminados. En esta fase se incluyen los aspectos de diseño e implementación del algoritmo de encaminamiento elegido.

6. Desarrollo de aplicaciones guiadas. Fase diseñada para la realización de ejemplos guiados para familiarizarnos con el desarrollo de aplicaciones.

7. Implementación de la solución. En la séptima fase del proyecto se pretende acometer la implementación del sistema, así como la solución de posibles problemas derivados de esta implementación

Fase	**Carga de trabajo (horas)**
Revisión bibliográfica	100
Familiarización con entorno de trabajo	120
Familiarización con entorno de desarrollo	20
Estudio de protocolos y encaminamiento	15
Propuesta y diseño de solución	20
Desarrollo de aplicaciones guiadas	5
Implementación de la solución propuesta	180
Evaluación de la solución propuesta	20
Elaboración de memoria técnica	90

Tabla 3.1: Planificación temporal del proyecto

8. Evaluación de la solución propuesta. En la octava fase se llevarán a cabo diferentes evaluaciones, dividas por módulos, para comprobar el correcto funcionamiento de la solución diseñada. Además se reserva en esta fase carga de trabajo para realizar la evaluación sobre la solución completa, es decir, todos los módulos juntos.

9. Elaboración de la memoria técnica del proyecto. En esta última fase se incluye todo lo relacionado con la realización del documento técnico. Se incluyen el tiempo de preparación y conocimiento del lenguaje Látex y la realización técnica del proyecto.

Esta planificación tiene su equivalencia en duración (horas) en la tabla 3.1.

3.2. Estimación de costes

Esta sección vamos a desglosar los recursos utilizados y el coste que estos implican. Dividiremos los recursos utilizados en: recursos humanos, recursos software y recursos hardware.

3.2.1. Recursos Humanos

Los recursos humanos estám divididos en dos: Cristian Alfonso Prieto Sánchez (alumno/investigador) y D. Juan José Ramos Muñoz (profesor/tutor). Estimamos que el sueldo de un graduado del nivel del alumno que ha realizado el proyecto (recién titulado) puede estar en torno a 15 euros la hora. El sueldo medio de un profesor/doctor se estima en los 30 euros/hora.

Con estas suposiciones, suponiendo que el profesor/tutor ha realizado un trabajo aproximado de 50 horas (40 horas de tutorización y 20 de revisión del trabajo del alumno) obtenemos un coste total:

$$Costehumano = 570horasEst \cdot 15€/hora + 60horasProf \cdot 30€/hora = 10{,}350€$$

Recurso	Período de uso	Coste
Equipo personal	12 meses	750 €
Acceso Internet	12 meses	25€ · 12
Acesso IEEE (artículos académicos)	12 meses	Gratuito (convenio universitario)

Tabla 3.2: Estimación de costes en recursos hardware.

3.2.2. Recursos software

En la planificación de costes se ha intentado minimizar el coste total, abogando por el uso de herramientas gratuitas y libres. El coste asociado es de 0 euros, al usar las herramientas que se enumeran a continuación. Además se ha necesitado hacer uso del software *Text-Live* y *Texmaker* para la realización de esta memoria técnica.

- Ubuntu 14.04.
- Compilador Látex para Línux y editor Texmaker.
- *Mininet*.
- OpenJDK 7.
- *Wireshark*.
- Compilador JAVA Maven.
- IDE ATOM.
- Controlador *OpenDayLight*.
- Software emisión RTP.
- *Mausezahn* y *VLC*.
- Captura de vídeo software. *SimpleScreenRecorder*.

3.2.3. Recursos hardware

Serán necesarios los recursos físicos especificados en la tabla 3.2.

3.2.4. Presupuesto final

En total se obtiene un coste total de 11.400 €para la realización completa del proyecto, donde se han incluido todos los recursos necesarios, tanto humanos como software y hardware.

Capítulo 4

Diseño de la solución

En este capítulo se definirán los elementos que posteriormente conformarán la solución propuesta. Para ello se detallará el diseño software que regirá el comportamiento del sistema, dividiendo en secciones los tres componentes de la propuesta que se han considerado de mayor importancia (algoritmo de encaminamiento con calidad de servicio, recogida de estadísticas de red, instalación de rutas y recuperación antes cambios de topología).

4.1. Introducción

La solución diseñada en este proyecto se basa en aprovechar la potencia que ofrece disponer de un controlador SDN centralizado, que tiene una visión global de toda la red, para poder desplegar rutas que satisfagan los requisitos de calidad de experiencia (QoE) de varios flujos multimedia.

Concretamente, se ha diseñado un protocolo de encaminamiento de *estado de enlace*, para generar las rutas por cada flujo multimedia que llegue a la red SDN. Para ello, el protocolo de encaminamiento toma como costes distintas métricas, que dependen del tipo de flujo para el que se crea la ruta. Así, tras identificar el tipo de un nuevo flujo, se calculará y configurará una ruta a su destino, que minimice los parámetros de red que degradan la calidad que percibirá el destinatario.

La solución propuesta se compone de varios elementos, que también han sido debidamente diseñados:

- Clasificación de flujos paquetes (Sec. 4.2). Para poder seleccionar la ruta que mejor se ajuste a los requisitos de QoE de un flujo de paquetes, es necesario identificar a qué tipo de aplicación corresponde dicho flujo.

- Algoritmo de encaminamiento (Sec. 4.3). Una vez reconocido el tipo de flujo, es necesario ejecutar un algoritmo que proporcione una ruta hacia el destino del flujo. Los parámetros de QoS de los enlaces que componen dicha ruta debería maximizar la calidad que el usuario final al que va destinado el flujo percibirá. Para ello hace falta estimar previamente cuál es el coste de cada enlace para el algoritmo.

- Definición de la matriz de costes por enlace (Sec. 4.3.2). El algoritmo seleccionado para construir las rutas para cada flujo necesita una representación de la red (qué *switches* y qué enlaces existen). Esta representación será la matriz de costes. Para calcular dicha matriz, es necesario definir una métrica de calidad que se base en los parámetros objetivos de cada enlace, y que tenga en cuenta el tipo de flujo que debe seguir dicha ruta.

- Estimación de los parámetros de calidad de servicio por enlace (Sec. 4.4). Para extraer los parámetros objetivos de calidad de cada enlace, es necesario definir cómo calcularlos a partir de la información que proporciona cada *switch*.

Cada uno de estos elementos se describen a continuación.

4.2. Clasificación de paquetes

Dentro del desarrollo de este trabajo se necesita evaluar el desarrollo con tipos de tráfico acordes a las necesidades y objetivos que se han planteado hasta ahora. A fin de conseguir este objetivo, se considerarán los tipos de flujo siguientes: flujos Transmission Control Protocol (TCP), [32]) o Usser Datagram Protocol (UDP), [33]), vídeo sobre Real-time Routing Protocol (RTP), [34]), voz sobre Real-time Routing Protocol (RTP), y tráfico Internet Control Message Protocol (ICMP), [35]).

Hay que tener en cuenta que, aunque los *switches OpenFlow* pueden clasificar los paquetes por varios campos de la trama *Ethernet* que reciben, este análisis está limitado por los campos definidos en la especificación *OpenFlow*, [23]. Por ello, aprovechando que cuando llega el primer paquete de un flujo nuevo se le consulta al controlador qué hacer con él, será el controlador, que sí puede procesar en detalle el paquete recibido, el que identifique de qué tipo se trata.

En esta sección se tratará la detección de cada uno de estos tipos de tráfico. Si bien, algunos tipos de flujos pueden ser fácilmente identificables mirando las cabeceras del protocolo IP, ([36]), otros como los flujos de vídeo necesitan que ser analizados con más detalle para poder determinar su tipo.

La toma de decisiones del algoritmo de clasificación se lleva a cabo de acuerdo al diagrama mostrado en 4.1, y está orientado a minimizar la carga en el controlador en función de la carga de tráfico que supone cada uno de estos tipos. Esta minimización pretende aumentar la eficiencia del controlador para mejorar el servicio ofrecido. Atendiendo al diagrama 4.1, primero se diferencian los paquetes de tipo RTP, primero vídeo y posteriormente audio. En tercer lugar se situaría la detección de tráfico TCP y por último ICMP. Esta organización está basada en la suposición de requerimientos de ancho de banda de cada tipo de tráfico, es decir, hemos supuesto que en la red el número de paquetes RTP será mucho mayor al resto, con lo que será más probable que un nuevo paquete que llega al controlador sea de tipo RTP vídeo. Lo que se pretende conseguir es que un paquete de vídeo no deba pasar por el proceso de identificación de tipo ICMP o RTP.

4.2.1. Identificación de tráfico TCP y UDP

La identificación de este tipo de tráfico se puede extraer del campo *identificador de protocolo* de la cabecera de los datagramas IP [36]. La API (*Application Programming Interface*) JAVA del controlador *OpenDayLight* hace esta identificación automáticamente. Por ello, la decisión sobre si el tipo de paquete es UDP o TCP se realizará mediante métodos proporcionados por la *API* usada, ([37]).

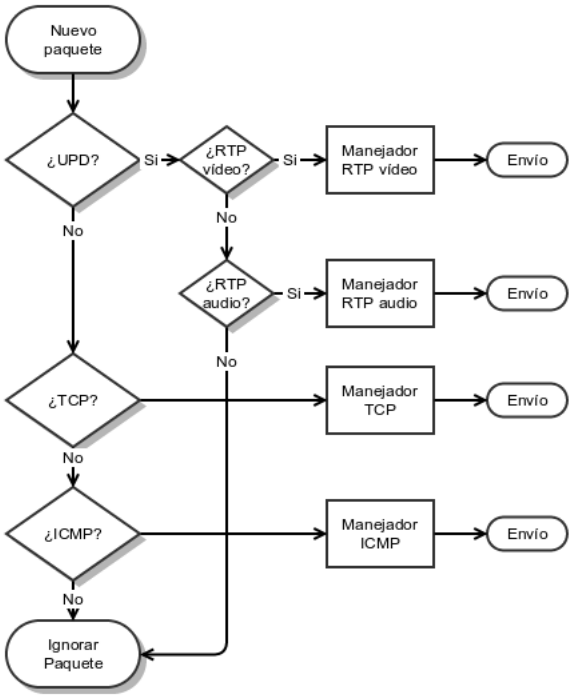


Figura 4.1: Procedimiento de clasificación de los flujos de paquetes.

4.2.2. Identificación de vídeo sobre RTP

Para identificar tipos de flujo de vídeo es necesario analizar la cabecera RTP y comprobar que el campo de *tipo de carga* (*Payload Type*, [34]), corresponda con el de alguno de los codificadores de vídeo que consideraremos en el proyecto. Además, para seleccionar con más precisión los flujos de aplicaciones concretas, se comprobará también el puerto al que va destinado. En nuestro caso, los paquetes de vídeo se envían al puerto 5004. Con estas dos condiciones, se puede crear una regla de emparejamiento en las tablas de los *switches OpenFlow* para identificar el flujo de vídeo detectado, que incluya sólo las direcciones IP y puertos del paquete. Una vez que se han instalado los flujos en los *switches*, estos encaminarán los paquetes correspondientes consultando en sus cabeceras tan solo el protocolo de transporte (UDP), el puerto de destino, además de las direcciones de origen y destino IP. El proceso de detección que se sigue es el siguiente:

1. Lo primero que se comprueba es si se trata de tráfico UDP.
2. Una vez comprobado que estamos ante tráfico UDP, se llama a la función que determinará a cuál de los tipos admitidos corresponde.
3. La función para vídeo comprueba dos requisitos:
   a) Primero comprueba que el puerto de destino es el 5004.
   b) A continuación comprueba los campos señalados en la figura 4.2, analizando si tienen el código estandarizado para vídeo RTP sobre Moving Picture Experts Group (MPEG) [38]. El primer valor consultado debe corresponder a la versión, que actualmente tiene valor 2. El segundo campo que se consulta es el correspondiente al tipo de carga, que para MPEG es el 33[1].

[1]Especificación RTP. `http://www.ietf.org/proceedings/48/I-D/avt-rtp-mp2t-00.txt`

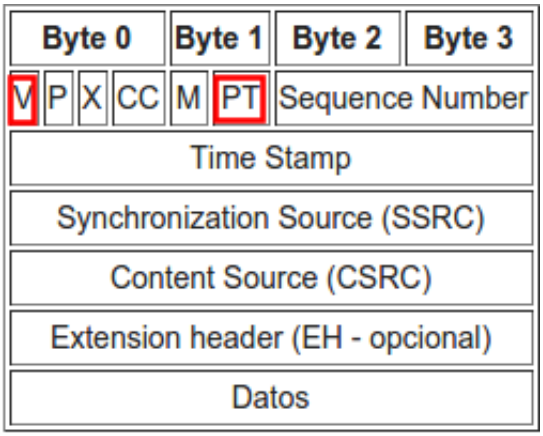


Figura 4.2: Cabecera RTP. Imagen tomada de `https://tools.ietf.org/html/rfc3550`.

4.2.3. Identificación de voz sobre RTP

La detección de paquetes de voz sigue un proceso muy parecido a los paquetes de vídeo. Sin embargo, en este caso, el puerto de destino al cual se envían los paquetes es el puerto 30000. Además, el tipo de carga útil de los paquetes RTP coincide con un estándar para voz, el G.711 definido por la Unión Internacional de telecomunicaciones (ITU) para la transmisión de voz [39]. Esta codificación coincide con el tipo 0 en RTP, definida en el RFC 3551[2].

4.2.4. ICMP

ICMP es un tipo de tráfico distinguido tanto por los *switches* OpenFlow como por el controlador *OpenDayLight* (incluyendo constantes de identificación ICMP), con lo que la detección de este tipo de tráfico no va más allá de la llamada a un método de la JAVA API que devolverá si el tipo de tráfico es ICMP o no.

4.3. Protocolo de encaminamiento

Uno de los problemas y a la vez objetivos de nuestro proyecto es el diseño de un sistema capaz de conseguir la mejor ruta dependiendo de los requisitos de QoS de varios tipos de tráfico. En esta sección se pretenden dar las claves a partir de las cuales se consigue calcular el camino más apropiado (más rápido, con menos pérdidas o más corto), según el tipo de tráfico a encaminar. Esto se conseguirá a través de la aplicación de dos elementos: un algoritmo de encaminamiento, y una función de cálculo de coste de rutas.

4.3.1. Algoritmo de encaminamiento

En una red SDN es el controlador el que toma las decisiones relacionadas con la infraestructura de red. Estas decisiones se toman de forma lógicamente centralizada, ya que además en el controlador dispone de la información de la topología de red. Por ello, es adecuado elegir un algoritmo de encaminamiento centralizado que aproveche el conocimiento completo de la topología de red para crear rutas de menor coste. Concretamente, nuestro diseño contempla aplicar el algoritmo de *Dijkstra* para identificar las rutas óptimas.

[2] `https://tools.ietf.org/html/rfc3551`.

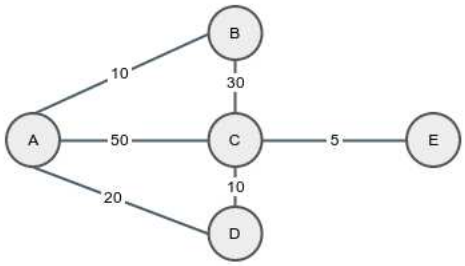


Figura 4.3: Grafo de una topología de red con costes asignados a los enlaces.

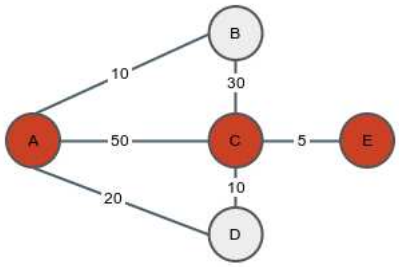


Figura 4.4: Solución aplicación del algoritmo sin consideraciones de costes.

Dijkstra es un algoritmo diseñado para determinar el camino más corto (o de menor coste) entre dos vértices de un grafo. *Dijkstra* no es el único algoritmo de encaminamiento existente, pero si que es uno de los más conocidos, usados e implementados, lo cual es una ventaja a la hora de aplicarlo en nuestro desarrollo. Otro algoritmo adecuado a nuestro problema sería el de *Bellman-Ford* [40]. El inconveniente de usar el algoritmo de *Bellman-Ford* es que tiene un tiempo de procesamiento mayor, lo cual lo hace una opción menos apta para nuestro diseño. Otros algoritmos como los de *Johnson* [41] o *Floyd-Warshall* [42] también podrían ser aplicados en esta solución. Sin embargo, estos dos algoritmos se diferencian en que en lugar de devolver el camino más corto entre dos vértices de un grafo, devuelven los caminos más corto entre todas las parejas de vértices de la red. Estos algoritmos parten con ventaja en redes estáticas, pero dado que nuestro desarrollo pretende demostrar la viabilidad de utilizar SDN para reconfigurar rápidamente una red cambiante, no parece muy lógico usar algoritmos más lentos que tendrán que ser recalculados tras cada cambio. Una vez justificado el uso de este algoritmo sobre otros, conviene entender el modo de funcionamiento.

La idea en la que se basa es explorar todos los caminos más óptimos que parten desde un vértice origen hacia el resto, descartando las rutas que no son prometedoras. Una vez hallados todos los caminos más óptimos, el algoritmo se detiene, obteniendo como resultado el camino más óptimo desde un vértice origen a todos los demás. El ejemplo expuesto a continuación ilustra el modo de funcionamiento del algoritmo.

Partamos de un grafo de red con un coste asociado a cada enlace como en la figura 4.3.

El objetivo será encontrar el camino de menor coste entre los nodos A y E. Si se usase el algoritmo de *Dijkstra* (o cualquier otro) sin ponderar el coste de cada enlace, es decir, suponiendo que todos los enlaces tienen el mismo coste, obtendríamos el camino de la figura 4.4, A-C-E, que no es el de menor coste, sino el más corto.

Usando *Dijkstra* pesado con costes que estén relacionados con los parámetros de QoS oportunos para cada tipo de tráfico, se obtendría un camino mejor. Por ejemplo, si en el caso presentado en la figura 4.3 los costes de cada enlace correspondieran a su retardo de transmisión, para una aplicación con requisitos de tiempo real, obtendríamos la ruta óptima de mínimo retardo, A-D-C-E. Un desarrollo paso a paso del algoritmo se puede encontrar en el apéndice B.1, donde se pueden comprobar paso a paso cada una de las iteraciones que nos llevan hasta el resultado deseado.

Otra de las características de *Dijkstra* que puede ser útil para nuestro diseño, es que es capaz de aplicarse sobre grafos dirigidos. Es decir, es capaz de hallar el mejor camino considerando enlaces con dos sentidos, lo que (aunque en nuestro desarrollo actual no tenga un gran peso), es muy útil en redes que no tienen características simétricas en los enlaces de subida y de bajada.

De cualquier manera, para poder aplicar el algoritmo de encaminamiento, es necesario definir los costes de los enlaces de la red mediante una matriz de costes. Dichos costes dependerán del tipo de tráfico que deba seguir la ruta calculada por *Dijkstra*.

4.3.2. Estimación de la matriz de costes

La estimación de costes de enlaces es uno de los puntos de mayor relevancia en este trabajo. Esta estimación permite un encaminamiento personalizado para cada tipo de flujo. Sin esta estimación no se repartirían los flujos a lo largo de distintas rutas de la topología de red, y siempre se escogería el camino más corto, sin distinguir tipos de flujo, se escogería siempre el mismo camino. Como solución se ha optado por el cálculo de una matriz de costes por cada tipo de flujo, como se explica a continuación.

La matriz de costes se recalcula cada vez que se actualizan las estadísticas de la clase. Esta matriz se actualiza con los nuevos pesos asociados a cada enlace. El procedimiento que se sigue para rellenar la matriz es el descrito en la figura 4.5, y sigue los siguientes pasos:

1. Se recorre cada enlace de la topología. Esta información eśta previamente almacenada en una matriz de enlaces, en la que se especifica por cada pareja de nodos, la existencia de un enlace. Por ejemplo, la posición (1,2) de la matriz contiene el enlace que va desde el nodo 1 al 2.

2. Según la posición que estemos recorriendo en el momento actual pueden pasar varias cosas. En las posiciones de la diagonal de la matriz (1-1, 2-2, ..., $i - i$), se asigna un peso 0 al coste. En las posiciones donde no hay enlace asignado, se asigna coste indeterminado, indicando que no existe enlace entre nodos. Por último en caso de que sí exista enlace se aplica la ecuación 4.1, que devuelve un valor diferente dependiendo del tipo de flujo que se esté tratando.

$$C_i[Edge] = \alpha_i \cdot C_{latency}[Edge] + \beta_i \cdot C_{jitter}[Edge] + \gamma_i \cdot C_{load}[Edge] + \sigma_i \cdot C_{loss}[Edge] \quad (4.1)$$

Donde $C_i[Edge]$ representa el coste del enlace $Edge$ para el tipo de tŕafico i, $C_{latency}[Edge]$ la componente del coste debido a retardo, $C_{jitter}[Edge]$ la componente del coste debido a la variación de retardo entre paquetes (*jitter*), $C_{load}[Edge]$ la componente del coste debido al consumo de capacidad del enlace, y $C_{load}[Edge]$ la componente del coste relativo al porcentaje de pérdidas de paquetes del enlace $Edge$. Cada componente está pesado con un factor que pondera la importancia de dicha componente sobre el coste final, y dependen del tipo de tráfico que demanda la nueva ruta. Estas funciones están descritas a continuación, subsección 4.3.3.

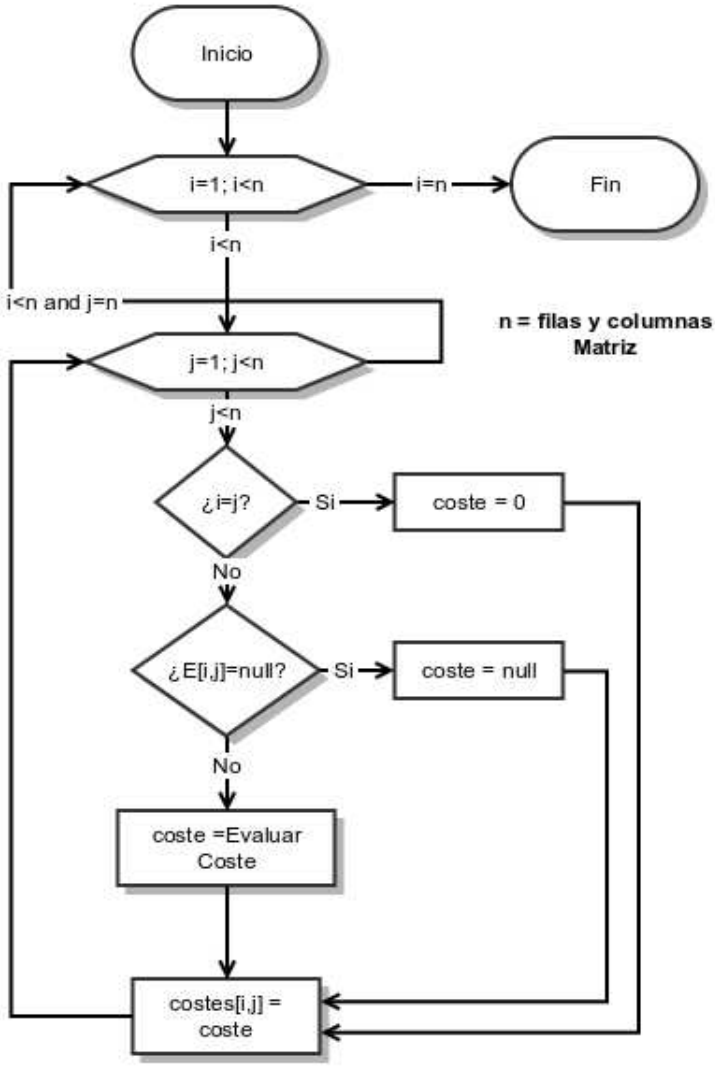


Figura 4.5: Diagrama de flujo para la construcción de matriz de costes.

La asignación de los valores α_i, β_i, γ_i y σ_i se ha realizado en la sección 7, donde a partir del estudio sobre los diferentes flujos, y resultados obtenidos, se han estimado los que consideramos mejores valores para cada tipo de tráfico (i). Los distintos componentes del coste han de estimarse previamente. A este paso se le ha denominado: estimación de estadísticas de red.

4.3.3. Descripción de métricas de coste

Para la evaluación de costes de cada enlace se usan 4 funciones diferentes, cada una relacionada con el tipo de evaluación que realiza: función de evaluación de latencias ($C_{latency}[E]$), función de evaluación de *jitter* ($C_{jitter}[E]$),función de evaluación de pérdidas de paquetes ($C_{loss}[E]$) y función de evaluación de carga (($C_{load}[E]$)). Estas cuatro funciones se normalizan para que devuelvan valores de coste entre 0 y 100. Una vez definidas estas funciones de coste, es necesario ajustar los pesos α, β, γ y σ de cada uno de dichos componentes para obtener el coste total estimado para cada tipo de tráfico.

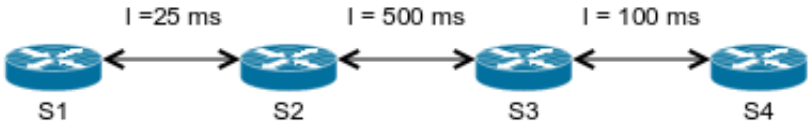


Figura 4.6: Ejemplo de topología para el cálculo de costes asociados a latencias.

4.3.3.1. Función de evaluación de latencias

Gracias a la medida de latencias descrita en la sección 4.4.1 y evaluada en el apartado 7.1.3, podemos ahora realizar una evaluación del coste asociado a los enlaces en función de la latencia.

Hay que tener en cuenta que es necesario normalizar la función de costes para que su rango esté entre de 0 a 100. Eso implica que hay que determinar qué valores de retardo tendrán asociado el valor 0 y cuáles el 100. Ya que esta función se utiliza para comparar los costes de varios caminos en el algoritmo de encaminamiento. Por ello, se normalizará la salida de esta (y siguientes) funciones de coste teniendo en cuenta la latencia mínima y máxima.

Para este cálculo será usada la latencia media estimada por el módulo de recolección de estadísticas diseñado. La latencia mínima encontrada en la red tendrá asociado un coste de 1. También se tendrá en cuenta la latencia máxima encontrada en la red pues será necesario acotar el coste resultante entre los límites establecidos (1 y 100), siendo en este caso 100 el coste máximo.

El cálculo asociado al resto de retardos se debe calcular teniendo en cuenta la normalización descrita. En este trabajo se han barajado dos posibilidades para calcularlas: ajuste exponencial o ajuste lineal de costes. A continuación se describe una topología sobre la que se ejemplificará el cálculo de costes mediante ambas alternativas

Topología de ejemplo. Se hará uso de la topología de la figura 4.6 para ilustrar el cálculo de costes para latencias. En esta topología podemos visualizar 4 *switches* conectados uno tras de otro, por tanto tendremos tres enlaces que evaluar. Estos enlaces se configuran para tener los siguientes retardos $l_{sw_i - sw_j}$:

$$l_{S1-S2} = 25ms$$
$$l_{S2-S3} = 500ms$$
$$l_{S3-S4} = 100ms$$

En esta topología, las latencias mínimas y máximas son 25ms y 500ms respectivamente. Estos valores hay que tenerlos en cuenta, pues los costes se calcularán considerando esos valores como umbral inferior y superior, respectivamente.

Correspondencia entre latencias y costes. Para el cálculo de costes en función de la latencia se propone hacer uso de un ajuste lineal que distribuya el coste entre las latencias máximas y mínimas registradas en la red. Se puede usar otro tipo de ajuste, por ejemplo exponencial, pero por simplicidad y eficiencia nos hemos decantado por uno de tipo lineal.

Para un ajuste lineal, el cálculo se lleva a cabo de la siguiente manera:

$$C_{latencia}[E] = a + b \cdot d \tag{4.2}$$

donde d es la latencia estimada en el enlace y medida en milisegundos, y a y b deben obtenerse del siguiente sistema de ecuaciones:

$$\begin{aligned} a \cdot d_{min} + b &= 1 \\ a \cdot d_{max} + b &= 100 \end{aligned} \tag{4.3}$$

Por ejemplo, para calcular el coste asociado al enlace de la topología presentada en 4.6, donde $d_{min} = 25$ y $d_{max} = 500$. Despejando del anterior sistema obtenemos que $a = 0{,}208$ y $b = -4{,}211$. Por tanto, para el enlace S3-S4 obtenemos que $C_{latencia}[S3 - S4] = -4{,}2 + 0{,}208 \cdot 100 = 16{,}6$

Se debe establecer una diferencia mínima entre las latencias mínimas y máximas, a fin de limitar el impacto del coste. Por ejemplo, en una red donde todos los enlaces tengan una latencia asociada igual, una variación menor a un milisegundo podría suponer una diferencia de coste de 99 unidades, siendo en este caso una mala medida. Por tanto se establece un mínimo de diferencias que será capaz de cubrir ampliamente el aumento de latencias debidas al tiempo de procesamiento por parte del controlador. A partir de los resultados observados experimentalmente en pruebas previas, se propone usar una diferencia mínima de 5 ms.

4.3.3.2. Función de evaluación de *jitter*.

Uno de los problemas que más afectan a aplicaciones multimedia es el *jitter*. Una de las primeras definiciones de esta propiedad se puede encontrar en [43], donde se define como la variación en el tiempo en la llegada de los paquetes, causada por congestión de red, perdida de sincronización o por las diferentes rutas seguidas por los paquetes para llegar al destino. El *jitter* es por tanto uno de los problemas más complicados de paliar/solucionar en redes multimedia, con lo que elegir el mejor camino (en cuanto a *jitter*) en ciertos tipos de tráfico tendrá un gran valor.

La detección de *jitter* se lleva a cabo usando las medidas temporales y medias de latencia analizadas en la sección 4.4.1. Esta detección tiene ciertas complicaciones, pues como se comprobó existen variaciones que no dependen únicamente del enlace en las medidas, sino que también dependen del tiempo de procesamiento en el controlador. Otro inconveniente de estas medidas está provocada por la relación entre *jitter* y latencias. Por ejemplo, un *jitter* de un milisegundo en una red con latencias promedio de 10 milisegundos tendría una relevancia muy importante, sin embargo encontrarnos este mismo *jitter* de un milisegundo en redes con latencias de 500 milisegundos tendrá una importancia mucho menor, sin embargo, si solo se consideran *jitter* mínimo y máximo para el cálculo, se obtendrá el mismo coste para el enlace en ambos casos.

Para solucionar estos dos problemas se establece:

1. Limitar una diferencia mínima entre los valores máximos y mínimos de *jitter*, a partir de la cual comenzar a calcular costes estándares (si la diferencia fuera menor, los enlaces tendrían un coste asociado al *jitter* igual para todos).

2. Escalar el coste del *jitter* en función de las latencias máximas y mínimas encontradas, de modo que un *jitter* muy grande en una red de latencias pequeñas (y cercanas), será mucho más significativo en cuanto a coste que un *jitter* grande en una red lenta y de grandes diferencias.

Con estas consideraciones se ha decidido calcular el *jitter* como la ecuación 4.4:

$$jitter = |latencia_{instantanea} - latencia_{promedio}| \tag{4.4}$$

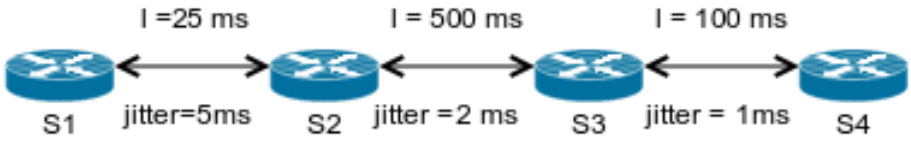


Figura 4.7: Topología para el cálculo de costes de *jitter* en enlaces.

Con el *jitter* por enlace calculado sería el momento se conseguir el coste en función de cada enlace. Se hará un ajuste lineal del mismo tipo que se realizó en el cálculo de costes para latencias, sin embargo en este ajuste se tendrá en cuenta el factor a (pendiente) del ajuste de latencias. Este factor servirá para escalar el coste obtenido por el *jitter* a valores que estén en consonancia a las latencias obtenidas en la red.

Por ejemplo, en la topología de la figura 4.7:

Siguiendo el mismo procedimiento descrito para el ajuste de latencias obtendremos los valores: $a_{jitter} = 24{,}75$, $b_{jitter} = -23{,}75$, combinados con el valor $a_{latencia} = 0{,}208$, obtenido anteriormente, devuelven un coste asociado $C_{jitter} = 5{,}356$

Con este ajuste se consigue normalizar el coste asociado al *jitter* en función de las latencias encontradas en la red. El coste asociado al *jitter* no estará acotado entre 1 y 100 al depender de la pendiente de la recta de ajuste para latencias, pero siempre mantendrá un valor relacionado al estado de la red.

4.3.3.3. Función de evaluación de pérdidas.

Para poder evaluar las pérdidas en la red, se usará el porcentaje de estas en un enlace. Este porcentaje equivaldrá al coste de forma directa, con lo que estará limitado entre 0 y 100. Para obtener el valor asociado al porcentaje de pérdidas se hará uso de las estadísticas recogidas en la sección 7.1.2.1, y se usará la ecuación 4.5[3].

$$C_{loss}(E) = \frac{sentBytes(E) - receivedBytes(E)}{sentBytes(E)} \cdot 100 \tag{4.5}$$

4.3.3.4. Función de evaluación de carga.

Esta es una de las funciones más difíciles de ponderar y describir ya que no tenemos medidas sobre el ancho de banda disponible en un enlace, y tan solo podremos estimar el ancho de banda consumido por este. De este modo no podemos garantizar asociar un mayor coste a un enlace con menor ancho de banda disponible, sino que asociaremos el mayor coste al enlace con mayor uso en la red. [4].

Para evaluar la carga se ha añadido al procesado de estadísticas una medida de tiempo que permita un cálculo de ancho de banda consumido. Se hace uso de la ecuación 4.6.

$$BW(E) = \frac{sentBytest1(E) - sentBytest0(E)}{t1 - t0} \tag{4.6}$$

[3]Se usarán los bytes enviados o recibidos según la orientación del enlace. Por ejemplo para el enlace 2-1 se usarán los bytes enviados por el puerto correspondiente al enlace S2-S1 en el nodo 2, y los recibidos por el puerto correspondiente a dicho enlace en el nodo 1.

[4]En la implementación final no se usa este coste al no garantizar medidas "útiles"

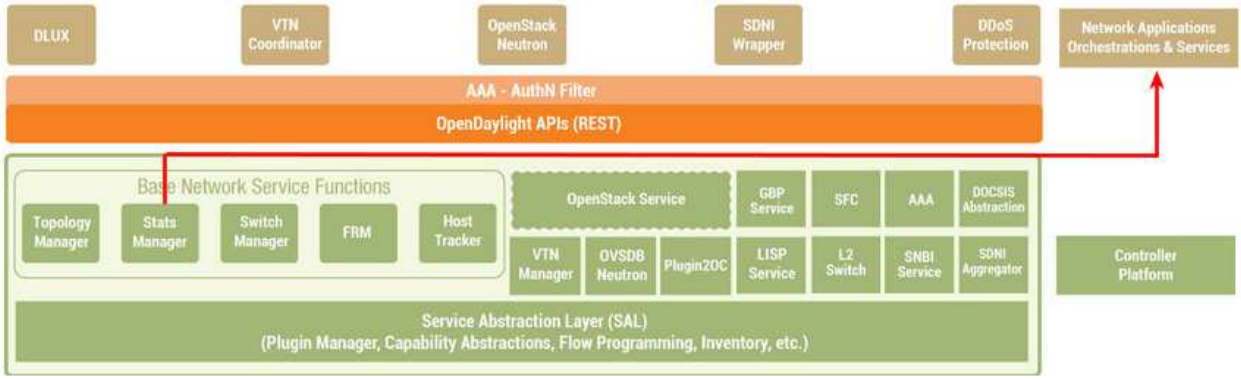


Figura 4.8: Obtención de estadísticas del módulo situado en el núcleo de ODL.

Esta ecuación devuelve unos valores estimados en Bps. Estos valores se deberán transformar a una medida más acorde a las velocidades de transmisión actuales (MBps). A partir del ancho de banda consumido se realizará un ajuste linea como los realizados para las funciones anteriores (con costes mínimo 1 y máximo 100), que permita distribuir el coste asociado al ancho de banda.

Ya que no se ha implementado en la solución final, el coeficiente σ_i tiene asignado el valor fijo 0.

4.4. Estimación de estadísticas de red

El controlador *OpenDayLight* cuenta en su núcleo con un recolector de estadísticas que almacena información sobre todos los elementos de red susceptibles de contar con estadísticas de red (paquetes enviados por un puerto, bytes recibidos por un *switch*, paquetes que han coincidido con un flujo en un *switch*, etc.), [9]. Haciendo uso de este almacén de estadísticas conseguiremos obtener datos que se usarán para la construcción de los costes asociados a cada enlace. En la figura 4.8 se puede ver dónde se implementa en *OpenDayLight* el módulo de gestión de estadísticas, que puede ser consultado desde las aplicaciones de red que se desarrollen.

Sin embargo, hay algunas estadísticas que, aunque teóricamente deberían estar disponibles en el controlador *OpenDayLight*, en la práctica no es así, por lo que nuestra solución ha tenido que implementarlas. En el desarrollo se ha hecho uso de tres estadísticas principales (de entre todas las posibles). A continuación se detalla la estrategia de obtención de dichos parámetros.

4.4.1. Latencia y *jitter*

Uno de los parámetros que puede tener impacto en las aplicaciones interactivas de red es la *latencia* extremo a extremo. Es decir, el tiempo que transcurre desde que un paquete es enviado hasta que llega al destino. En esa latencia influyen los retardos que se producen en cada *switch* y cada enlace. En este apartado se describe cómo se calculan los retardos por enlace en nuestra solución.

La latencia como tal no es una medida estadística que se haya obtenido a partir del módulo ODL, sino que se ha obtenido a partir de diversas medidas de retardos de paquetes. Como tal existe una propiedad en los enlaces (recogida dentro de las propiedades de los enlaces en redes SDN) que es la llamada *Latency*. Sin embargo al intentar extraer esta propiedad mediante los métodos proporcionados en la API de JAVA, no se ha obtenido valor alguno, con lo que, siendo una medida necesaria para evaluación de costes, se ha optado por medir nuestras propias estadísticas temporales. Se espera que este inconveniente se solucione en versiones futuras del controlador que implementen estas estadísticas de forma nativa.

Para la medida de estadísticas (no solo las de latencia), es necesario añadir un proceso de aprendizaje por el cual se almacenan valores estadísticos para todos los enlaces disponibles durante un periodo de entrenamiento inicial. La duración de este periodo de aprendizaje se ha definido en el capítulo 7 a partir de los experimentos. De este modo se establece un período de tiempo (paquetes) durante el cual todos los enlaces transmiten paquetes y se almacenan las estadísticas correspondientes a cada enlace. El proceso seguido para la obtención de estadísticas de retardos es el que se enumera a continuación:

1. Durante la fase de entrenamiento en la que se obtienen las estadísticas de retardos, los *switches* reenvían cada paquete al controlador. Si en el controlador se identifica como un tipo de flujo de los considerados en la solución (ICMP, Vídeo, Voz o TCP), se almacena junto a una marca temporal e información del puerto de entrada justo a su llegada. También se incluye una marca de tiempo a su salida del *switch*.

2. El controlador conoce qué enlaces existen. En la información de los enlaces se incluyen los identificadores de los puertos y *switches* que conectan. Si cuando llega un paquete i al controlador se comprueba que existe una marca de tiempo asociado al paquete y al enlace por el que entra al *switch*, se calcula la diferencia entre el instante de tiempo en el que se envió por el puerto de salida ($t_{i,out}$) y el tiempo en el que se recibe por el puerto de entrada ($t_{i,in}$). A continuación, el paquete almacenado se elimina de la memoria del controlador. En caso de no estar almacenado se continua el procesamiento normal y a la salida (cuando es enviado por algún puerto), se almacena junto a la marca temporal $t_{i,out}$.

   Por cada paquete i, estas medidas de latencia se estiman de acuerdo a la ecuación 4.7:

$$latency_i = t_{i,in} - t_{i,out} \tag{4.7}$$

3. Una vez calculada la diferencia temporal, estos datos son enviados a una función que los inserta en dos matrices que serán las que posteriormente se usen para la evaluación de costes. Una matriz contiene la media de los últimos valores de latencia para cada enlace, mientras que la otra solo contiene el último valor. El número de valores para el cálculo de la media se ha definido en el capítulo 7.

El diagrama de flujo de la figura 4.9 muestra el proceso explicado anteriormente.

Nótese que mediante este procedimiento se obtiene una medida experimental, lo que no es exactamente la latencia de un enlace, como se muestra en las figuras 4.10 y 4.11. El tiempo medido que nuestra solución obtiene es igual al de la ecuación 4.8. Para conseguir resultados más exactos, el procedimiento intenta no contabilizar el retardo de procesamiento dentro del controlador.

$$latency'_i = t_{C-1} + t_{1-2} + t_{2-C} \tag{4.8}$$

Donde t_{C-1} es la latencia sufrida por un paquete para ir desde el nodo 1 al controlador. t_{1-2} corresponde a la latencia sufrida por un paquete para ser transmitido a través del enlace entre S1 y S2. Por último, t_{2-C} es el retardo experimentado por un paquete para ir desde el controlador hacia el nodo 2.

En la sección 7.1.3 se estiman los retardos adicionales a la latencia como tal. Sin embargo es preciso remarcar que estos tiempos afectarán por igual a todos los enlaces (suponemos que todos los nodos están conectados al controlador con enlaces de iguales características), con lo que la variación respecto a los cálculos que se realizaran con la latencia ideal, debe ser mínima.

Como se ha comentado anteriormente, estas medidas serán ponderadas en la función de evaluación del coste por enlace en función del tipo de tráfico.

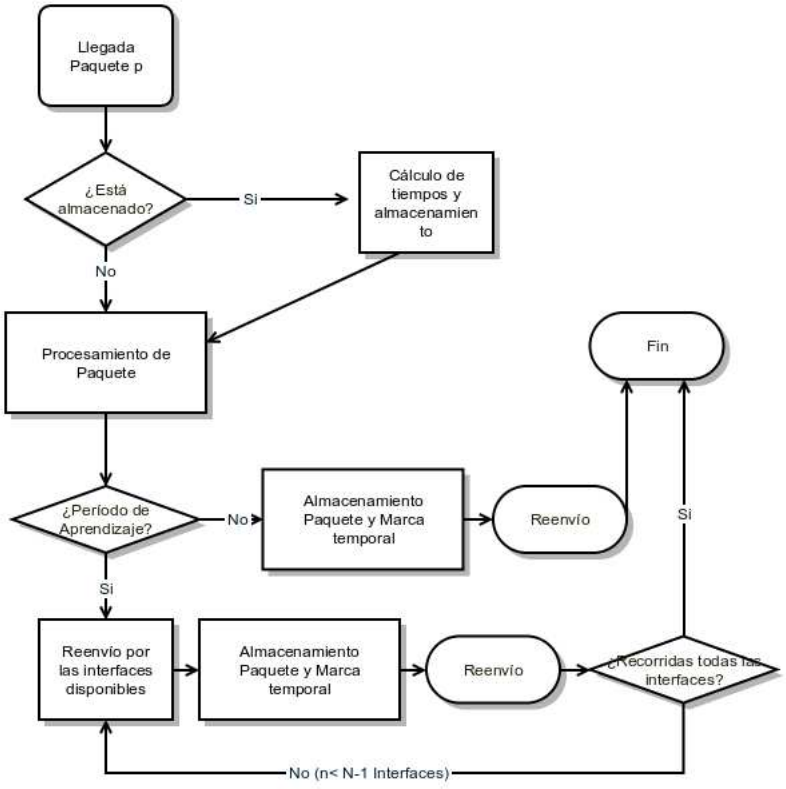


Figura 4.9: Diagrama de flujo del procesamiento de paquetes para la obtención de latencias en enlaces.

4.4.2. Pérdidas de paquetes

La recogida de estadísticas para pérdidas (y cualquier estadística relacionada con peticiones al módulo *statisticsManager*) se hará de forma periódica de acuerdo a un tiempo establecido t_{update}. El valor apropiado de t_{update} se estudiará con más detenimiento en la sección sobre actualización de la topología. Las pérdidas se representarán mediante un porcentaje, calculado como se explica a continuación.

Cada vez que se cumple el período de actualización t_{update} establecido, se consulta cuántos paquetes han sido enviados y recibidos por cada uno de los dos *NodeConnector* en cada enlace. Tenga en cuenta que un enlace tiene un *TailConnector* y un *HeadConnector*. Por ejemplo, el enlace 1-2 tendrá como *HeadConnector* un *Connector* del nodo 2, y como *TailConnector* un *Connector* del nodo 1. Una vez recogidas las estadísticas, se almacenan en un mapa de enlaces que posteriormente será usado por los diferentes manejadores para la estimación de costes asociados. Cada enlace tendrá asociado por tanto un mapa donde se habrán almacenado las diferentes estadísticas (en este caso especialmente interesantes la cantidad de paquetes y bytes enviados por el *TailConnector* y los recibidos por el *HeadConnector* para la estimación del porcentaje de pérdidas).

Cada vez que se cumple el período de actualización, se lleva a cabo esta actualización de estadísticas y se pasa a los manejadores correspondientes (clases para procesar cada tipo de flujo) toda la información actualizada en forma de "matriz de mapas", donde se tiene un mapa de estadísticas para cada enlace. Posteriormente, estos datos son procesados de acuerdo a la necesidad de cada tipo de flujo y manejador.

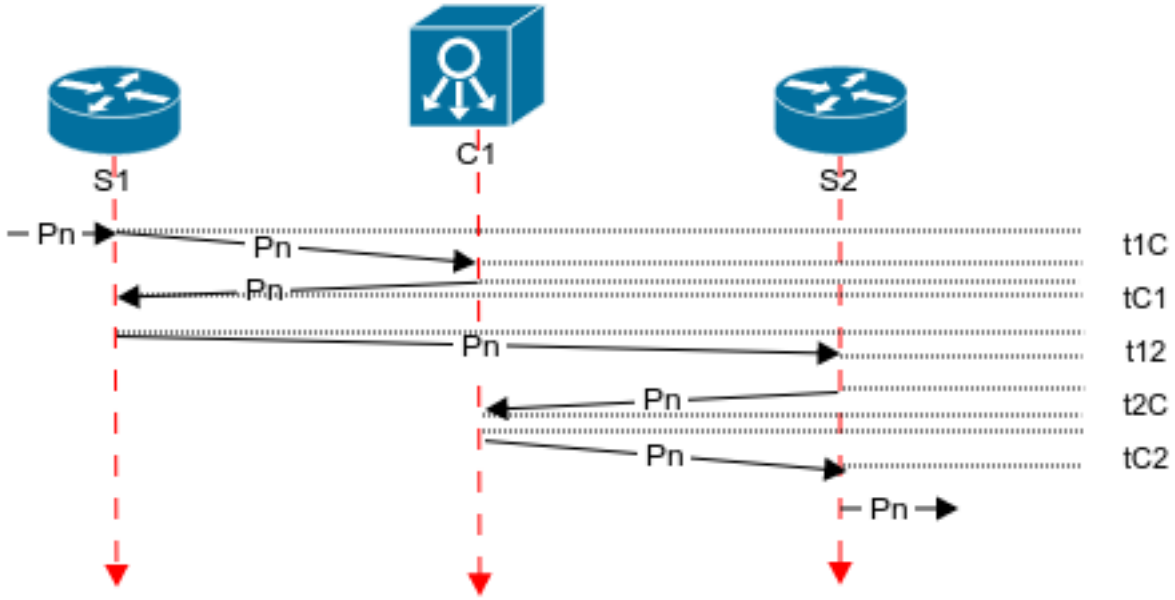


Figura 4.10: Intercambio de paquetes durante la medida de latencias.

4.4.3. Carga en los enlaces

La carga en los enlaces se define como el ancho de banda transportado por éstos. Esta se obtiene a partir de los mismos datos recogidos para la estimación de pérdidas, es decir, la cantidad de bytes y paquetes enviados y recibidos por cada puerto de cada *switch*. Además de los datos estadísticos, se hace uso de un mapa que contiene los valores máximos y mínimos de cada estadística. Por ejemplo, se almacena el máximo número de paquetes enviados por un *connector*. De este modo podemos estimar la carga soportada por un enlace y paliar la falta de información referente a la capacidad de los enlaces (que esperamos incluya las próximas versiones del controlador).

Con estos datos y estadísticas se construirá posteriormente la matriz de costes asociada a cada tipo de flujo y que permitirá un encaminamiento óptimo por tipo de flujo.

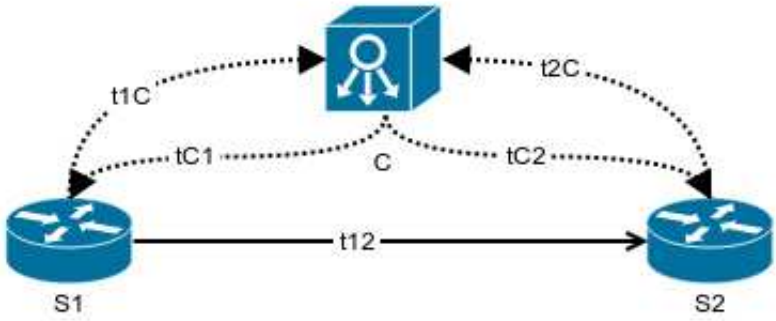


Figura 4.11: Intercambio de paquetes durante la medida de latencias.

4.5. Configuración de las rutas en los *switches*

Haciendo uso del algoritmo *Dijkstra* somos capaces de obtener el mejor camino entre dos nodos, de acuerdo a la métrica que elijamos. Los flujos que coincidan con nuestras especificaciones deberán seguir el camino precalculado. Las redes con *switches OpenFlow* implican la necesidad de instalar una regla por cada flujo en los nodos que componen el camino. En nuestra solución, esto se llevará a cabo de acuerdo a los pasos descritos a continuación, y plasmados en la figura 4.13.

1. Localización de los extremos. La API usada para el desarrollo provee métodos que nos permiten obtener los nodos a los cuales están conectados los *hosts*. Estos métodos devuelven un mapa donde cada *host*[5]. Gracias a la IP destino y origen, obtendremos los nodos en los que estamos interesados.

2. Obtención del camino. Con los nodos origen y destino consultaremos el mapa de caminos (caminos que ya han sido calculados y almacenados). En caso de estar el camino guardado ya, pasamos al punto 4. Si el camino no está guardado, haciendo uso del *Dijkstra* implementado, calcularemos el mejor camino entre los dos nodos.

3. Reordenación de camino. Uno de los problemas que se ha detectado es que a veces el algoritmo devuelve el camino sin ordenar, por ejemplo, para ir del nodo 1 al 4 nos puede devolver la sucesión de nodos: 1-2, 3-2, 4-3. Por ello se ha diseñado un procedimiento para reordenar los enlaces de las rutas. La reordenación de caminos sigue el flujo del diagrama presentado en la figura 4.12. A partir de esta lógica se obtiene la secuencia de nodos consecutivos, lo que posibilita la instalación de flujos.

   La reordenación de caminos se ha diseñado de una forma muy esquemática.

   a) Se busca el primer nodo del camino. Buscaremos el primer nodo del camino (nodo origen) en el camino devuelto por *Dijkstra*, una vez encontrado el nodo, añadiremos el enlace asociado al camino definitivo y lo eliminaremos de la lista devuelta por *Dijkstra*.

   b) Mientras queden enlaces en la lista, buscaremos aquellos cuyo *connector* de cola, coincida con el *connector* de cabecera del enlace anterior. En caso de no encontrar ningún enlace con dichas características se descarta el procedimiento y se devuelve un camino vacío.

   c) Por último, una vez no quedan enlaces en la lista devuelta por *Dijkstra*, se comprueba que el último enlace contiene el *connector* correspondiente al nodo destino. En caso afirmativo el camino definitivo estará completo. Si esta última condición no se cumpliese devolveríamos un camino vacío.

4. Instalación de flujos en los extremos. Una vez tenemos el camino ordenado, procedemos a instalar flujos para los extremos, es decir, instalaremos reglas de flujo para los nodos que conectan a los *hosts* implicados (direcciones IP).

5. Instalación de flujos en los nodos del camino. Se recorrerá el camino obtenido y se instalarán flujos en los nodos que reenviarán paquetes por el camino. Los flujos instalados dependerán del tipo de flujo, pues los parámetros que afectan al *match*, así como las acciones varían entre tipos de tráfico.

[5]Cada *host* tendrá asociada su dirección IP

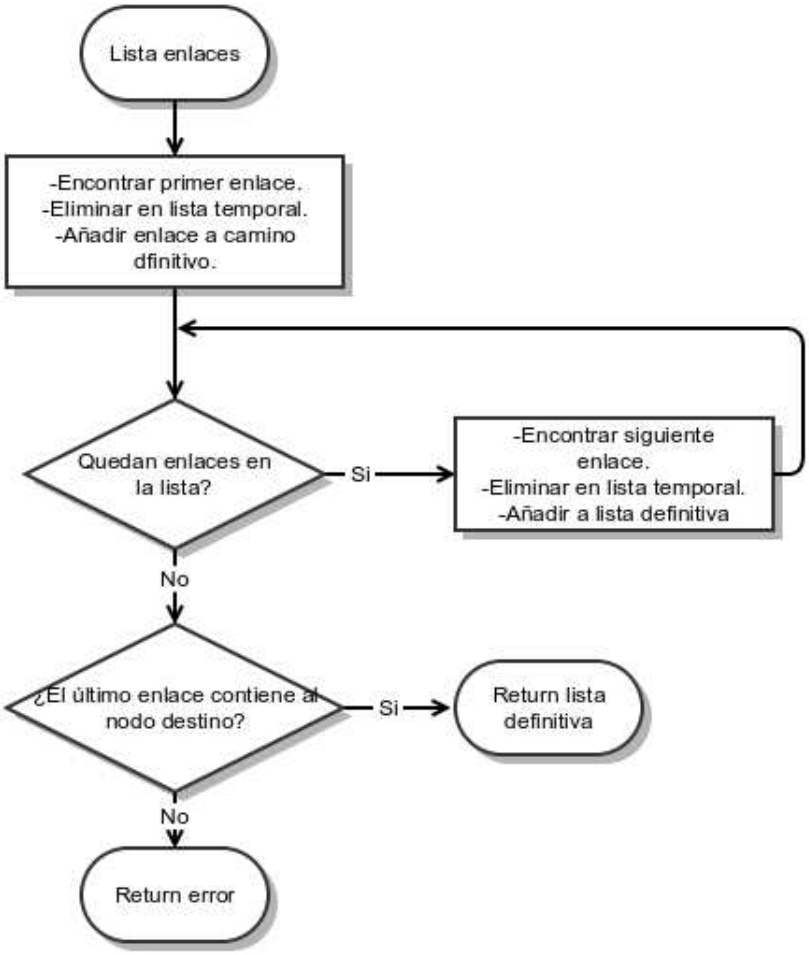


Figura 4.12: Diagrama de flujo para la reordenación de caminos.

Los flujos son instalados por orden y en el mismo proceso. Se planteó la instalación de flujos por separado (cada vez que un nodo lo pidiera), pero se concluyó que este es el método más rápido ya que existe un menor intercambio de mensajes y menor carga para la red.

4.6. Actualización de topología

En esta sección se describe el proceso que consigue responder ante cambios en la red tales como caídas de enlaces o nodos. Es uno de los pilares en los cuales se apoya todo el proceso para proveer de QoE y QoS a las aplicaciones que usan los usuarios. Para conseguir el objetivo, el procedimiento diseñado se divide en dos fases: detección y actualización de rutas.

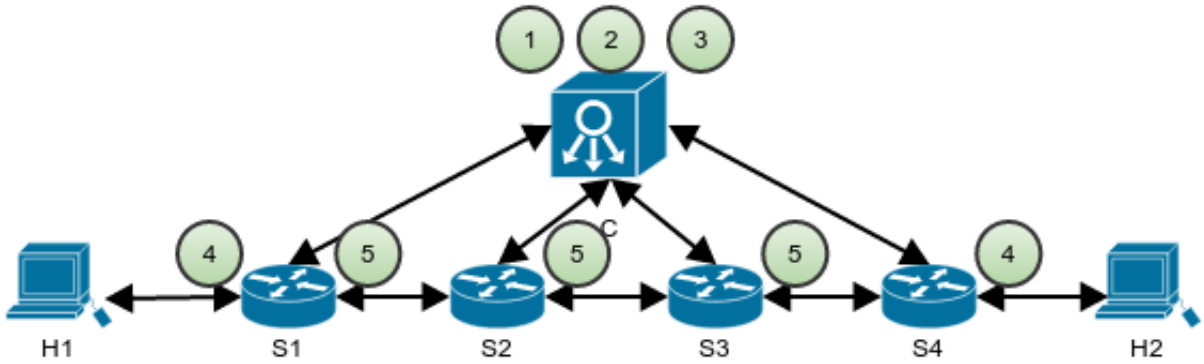


Figura 4.13: Proceso de instalación de flujos a lo largo de un camino.

4.6.1. Detección de cambios en la topología

Mediante el procedimiento de detección de cambios de la topología de red, nuestra solución es capaz de lanzar el procedimiento que actualice la información sobre la red, y recalcular las rutas que sean necesarias. Estos cambios comprenden la modificación de las características de los enlaces, la caída de nodos, etc. Si bien el propio controlador *OpenDayLight* puede detectar los cambios en la topología (existe un módulo capaz de avisar a las aplicaciones de red de cambios en la topología), se decidió prescindir de esta posibilidad, ya que nuestro diseño contempla el tratamiento de los flujos en el mismo módulo donde se reciben los paquetes.

La detección se lleva a cabo mediante consultas sucesivas al estado de la red. El tiempo entre consultas es el mismo tiempo que el de actualización de estadísticas, t_{update}. El valor de este tiempo de actualización se propone como 100 ms, pudiendo ser modificado según las necesidades de red o usuario. La propuesta está basada en:

- Un período de 100 ms garantiza la comprobación en un tiempo menor al que se tarda en actualizar todos los enlaces (como se comprobará en el capítulo 7.

- No se pretende sobrecargar el controlador eligiendo un tiempo menor, puesto que podría darse una situación en la cual se están actualizando estadísticas, mientras se detectan cambios en la topología, provocando dos accesos a un dato al mismo tiempo. Suponemos que con esta elección de tiempos se asegura que el tiempo de procesamiento es mucho menor a la actualización, evitando posibles problemas.

Por supuesto este valor podrá ser modificado en cualquier momento.

Desde la parte de detección es donde se detecta cuándo y dónde se produce el cambio. Tras la detección se llama a la función que manejará este cambio. La detección sigue el flujo de presentado en la figura 4.14.

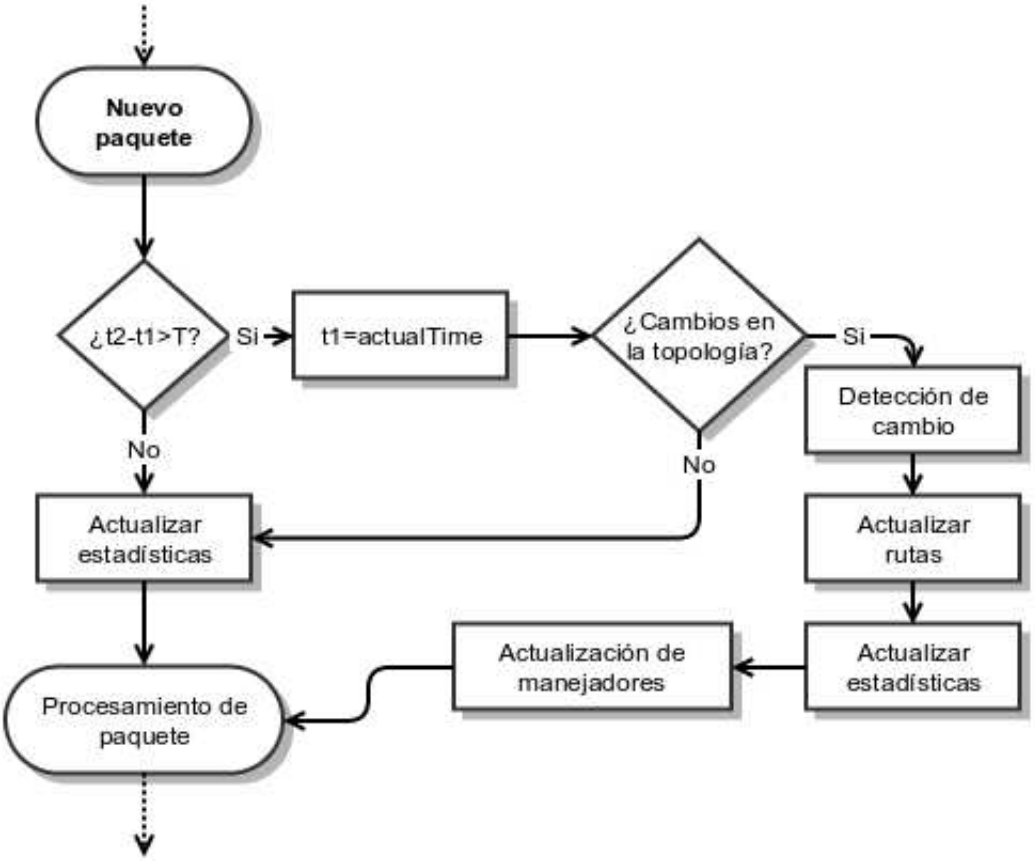


Figura 4.14: Diagrama de flujo para la detección de cambios en topología.

4.6.2. Actualización de rutas

Cada vez que es detectado un cambio en la topología se llama al procedimiento de actualización de rutas. Dentro de la actualización podríamos tener dos caminos que seguir. Por un lado se podría intentar actualizar cada ruta dando un camino alternativo para los flujos, y por otro se podría simplemente eliminar las rutas afectadas. Se ha optado por la segunda solución por dos razones: no conservar flujos que llevan tiempo sin usarse y es posible que no se vuelvan a usar. La segunda razón esta relacionada con la eficiencia de la solución, pues la carga que supondría tener que procesar eliminar flujos afectados y creación de nuevos, podría suponer una pérdida de paquetes importante. Aún así esta última tesis no se ha comprobado pues suponía una carga de trabajo muy elevada junto a la programación de la segunda solución. No por ello deja de ser una razón adecuada para la elección de la segunda solución.

De la solución elegida surgen las tres variantes que se comparan en la tabla 4.1:

Como se extrae de la tabla 4.1 se debe escoger la solución 3, borrado por tipo de flujo afectado, ya que es la de mayor eficiencia en cuanto a conservar flujos útiles e implica una comunicación menor con *switches*, evitando retardos provocados por la comunicación del controlador con estos. El resto de soluciones se intentaron usar para hacer una comparativa de resultados, pero en la versión actual del controlador ha sido imposible usarlas.

La solución elegida está descrita a continuación, además se puede consultar el diagrama de flujo en la imagen 4.15.

Capacidad/Solución	Borrado completo de flujos	Borrado completo de camino afectado	Borrado por tipo de flujo de camino afectado
Rapidez	Muy rápido. Solo recorre los nodos y elimina flujos.	Medio. Recorre los nodos afectados y elimina todos los flujos. Consulta todos los tipos de tráfico.	Baja. Recorre los nodos afectados y elimina los flujos afectados. Necesidad de comprobar cada flujo.
Eficiencia	Baja. Elimina flujos que no están afectados.	Media/Baja. Puede eliminar flujos que no están afectados.	Alta. Sólo elimina flujos afectados.
Facilidad de programación	Muy sencillo.	Media.	Difícil. Necesidad de comprobar cada flujo por cada tráfico.
Funcionalidad	Imposible de usar. Error con función para eliminar todos los flujos de un nodo.	Imposible de usar. Error con función para eliminar todos los flujos de un nodo.	Completa. Elimina los flujos afectados.

Tabla 4.1: Comparativa soluciones propuestas para actualización de ruta.

1. Se recibe el enlace que ha cambiado según la detección (cuando un nodo cae, todos sus enlaces son detectados y pasan por este proceso de forma individual).
2. Por orden se buscan caminos afectados por cada uno de los 4 tipos de tráfico afectados.
3. Por cada tipo de tráfico:
   a) Se comprueba si existen caminos afectados.
   b) En caso positivo se recorren los nodos del camino (suponemos estos caminos ordenados, al haber sido almacenados tras una reordenación).
   c) En cada nodo se comprueban los flujos instalados buscando flujos coincidentes con los que se deben eliminar. Por cada flujo se comprueba si su acción de envío coincide con el enlace que corresponda del camino. Se podría intentar eliminar solo el enlace afectado, pero en ese caso surgen problemas con el cálculo de caminos extremo a extremo al haber flujos instalados ya.
4. Finalmente se devuelve el control al programa de detección que actualizará mapas y estadísticas de de los manejadores.

Se da por terminado este capítulo donde se ha descrito cada uno de los elementos que posteriormente implementaremos en el sistema para poder alcanzar los objetivos del proyecto.

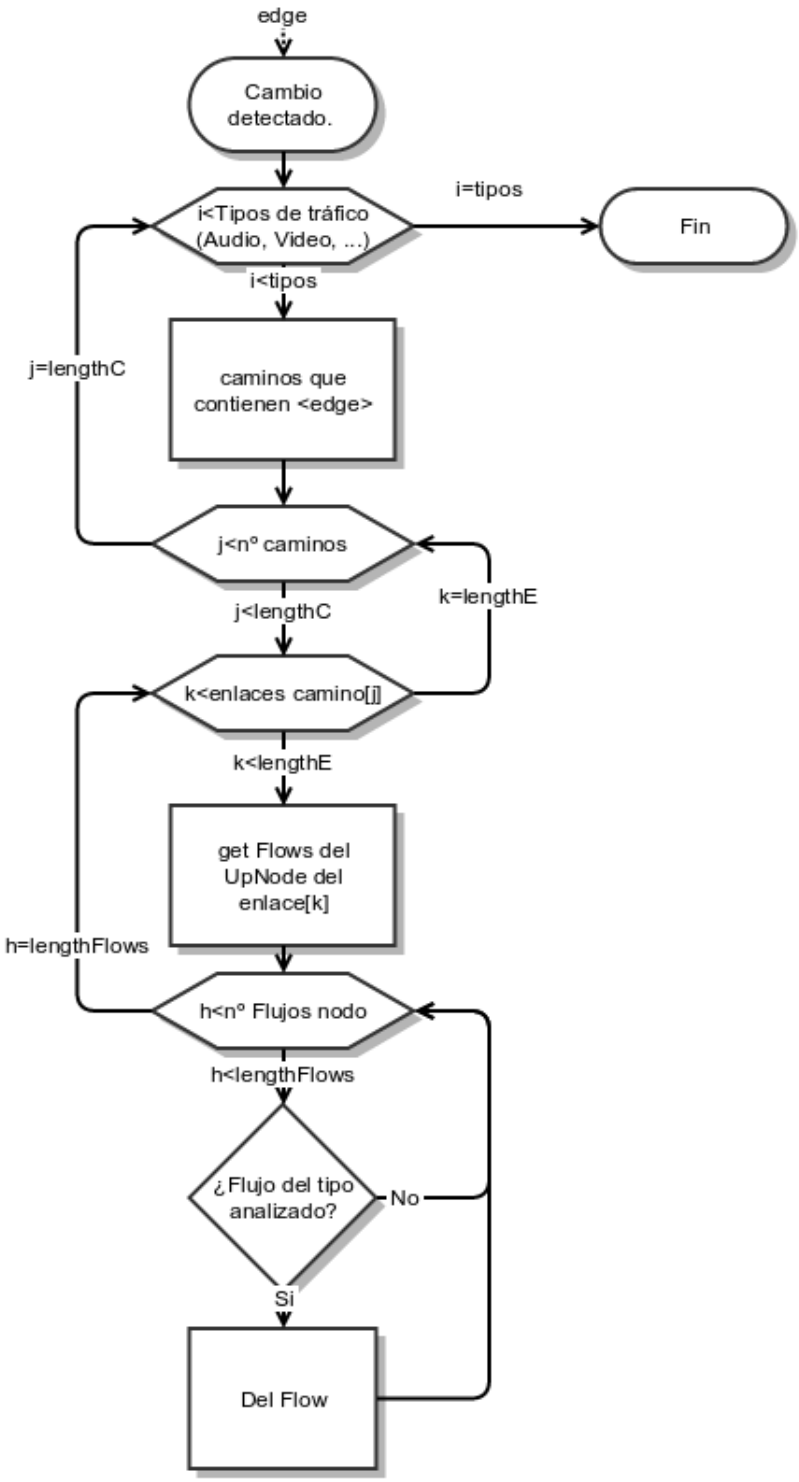


Figura 4.15: Diagrama de flujo del proceso de actualización de topología

Capítulo 5

Entorno de desarrollo

A lo largo de este capítulo se pretende introducir todos los elementos que formarán y permitirán la ejecución final de la solución propuesta. Comenzaremos describiendo el entorno en el cual se ejecutará dicha solución. Continuaremos describiendo la herramienta que posibilita que en un solo equipo puedan ser emuladas redes de gran complejidad, mininet. Al ser una parte troncal de este trabajo dedicaremos un apartado completo a la aplicación. Por último se tratará el entorno de ejecución de *OpenDayLight*, así como las herramientas que permiten desarrollar y ejecutar aplicaciones para este controlador.

5.1. Entorno de ejecución

Para agilizar el desarrollo, la ejecución y las pruebas del sistema desarrollado, se ha optado por integrar todos los elementos del entorno en un sólo equipo informático. Dicho equipo deberá cumplir ciertas condiciones. La de mayor importancia es que se trate de un equipo con SO Línux (Ubuntu [44] y experimentalmente Fedora [45]), o que el SO nativo sea capaz de ejecutar alguno de los sistemas permitidos (mediante el uso de virtualización). Esta condición viene impuesta por el modelo de *switch* SDN elegido, *Open vSwitch*, soportado por las versiones más recientes de Ubuntu (a partir de la 12.04) y a partir de la versión 20 por *F*edora.

5.1.1. Comparativa

Existen dos SO sobre los cuales podremos desarrollar y ejecutar el sistema de forma nativa. A continuación se comparán las dos posibilidades existenetes, para finalmente tomar una decisión.

De los sistemas basados en Línux, Ubuntu y *F*edora son dos de los más populares actualmente. La colaboración por parte de la comunidad en ambos casos, y tener dos grandes compañías (*C*anonical y Red Hat respectivamente) como soporte, han hecho que sean alternativas para los SO tradicionales para usuarios. Una de las ventajas principales de Ubuntu (para casos específicos como el de este trabajo) es la estabilidad de software. Esta ventaja radica en la mentalidad de los desarrolladores: mientras que en *F*edora se prima la innovación y la mejora continua, *C*anonical prefiere seguir un camino más estable y de menor desarrollo, a fin de introducir mejoras testadas exhaustivamente. Para un desarrollo a largo plazo se prefiere un SO estable.

Otra de las razones que pesarán en la decisión es el apoyo de la comunidad. Contando los dos sistemas con un gran número de usuarios activos y desarrolladores, el sistema mantenido por *C*anonical, tradicionalmente (y actualmente) cuenta con una comunidad mayor, con lo que enfrentarse a cualquier problema con el sistema tendrá una mayor probabilidad de éxito de ser resuelto.

Por último se ha de destacar que el emulador de redes que se usará, mininet [21], está perfectamente integrado en Ubuntu, mientras que *F*edora cuenta tan sólo con soporte experimental. Esta razón también tiene un gran peso sobre la decisión final, que no es otra que usar una versión estable de Ubuntu. En este caso se ha optado por la versión Ubuntu 14.04 *LTS*. Entre las diferentes versiones posibles de Ubuntu, elegimos la última versión *LTS*, que cuenta con un soporte de tres años, mientras que el resto de versiones (no *LTS*), cuentan con soporte de tan solo 6 meses, de acuerdo con [46][1].

Como se ha comentado en la introducción del capítulo, existen dos opciones para ejecutar el sistema elegido sobre un equipo: virtualización o instalación nativa.

- **Virtualización.** En el contexto en el que nos situamos, la virtualización hace referencia a la creación (mediante uso de software) de una versión virtual de un algún recurso, como puede ser un equipo hardware, recursos de red o un sistema operativo.

  A lo largo de este texto se ha hecho referencia antes a este concepto, refiriéndonos sobre todo a la virtualización de elementos de red. Ahora el término hace referencia a ejecutar un SO sobre otro. En definitiva, usar Ubuntu sobre un SO Windows o OS X.

  Una de las ventajas de usar virtualización contra instalaciones frescas es la posibilidad de ejecutar una misma imagen virtual en cualquier equipo, sin tener que realizar costosos procesos de instalación y configuración. Esta es una gran ventaja, sobre todo si ya existen imágenes preparadas para fines específicos. Por ejemplo existen imágenes listas para ejecutar mininet y el controlador sin necesidad de configuración extra.

  Otro de los aspectos a descartar es el respaldo del sistema, *backups*. Será muy cómodo hacer copias de seguridad de estas imágenes al solo necesitar copiar los archivos que guardan la configuración de la maquina virtual. En un SO instalado de forma nativa, el proceso es mucho más costoso en cuanto a tiempo se refiere.

  Para la tarea que nos ocupa (bien distinta de virtualización de recursos red como podría ser un servidor), se ha de destacar que el equipo donde será ejecutada dicha virtualización tendrá unos recursos limitados. Esto podría originar problemas ya que además de virtualizar un SO, se deberán emular redes y en estas redes se deberá emular una alta carga de envío de paquetes, pudiendo llegar a consumir los recursos del equipo. Por esta última razón (que además influye en la fluidez de trabajo sobre el equipo), se ha decidido realizar una instalación nativa de Ubuntu, donde se han instalado, configurando y personalizando todos los elementos que compondrán el sistema. En el apéndice A.1 se han recogido aquellos aspectos que consideramos claves para la configuración del sistema.

Por las razones que se han ido desglosando en esta sección (estabilidad y rendimiento), se ha decidido usar una instalación nativa del SO Ubuntu. Para la configuración se han seguido los pasos recogidos en el apéndice A.1. Esto no indica ningún tipo de incompatibilidad con cualquier otra de las diferentes posibilidades, ya que este sistema podrá ser montando de igual modo con cualquiera de los métodos recomendados por *Mininet*[2].

5.2. *Mininet*

Mininet es una herramienta que permite crear redes virtuales, con todos sus elementos en una sola máquina. Además permite interactuar fácilmente con las redes creadas a través de línea de comandos. Incluso permite el desarrollo y ejecución de redes diseñadas para mininet en hardware real, multiplicando de este modo las posibilidades ofrecidas por el software.

[1] Siempre haciendo referencia a las versiones de escritorio, *desktop*.

[2] `http://mininet.org/download/`

	Instalación nativa	Virtualización software
Rendimiento	El más alto posible	Limitado a los recursos prestados por el SO origen
Facilidad de configuración	Necesidad de configuración	Imágenes preparadas para funcionar desde el primer uso
Backups	Lento	Rápido
Compatibilidad	Limitado a equipos compatibles con el SO	Total (siempre y cuando el anfitrión pueda ejecutar software de virtualización)
Facilidad de instalación del SO	Fácil	Muy fácil

Tabla 5.1: Comparativa entre instalación y virtualización de SO

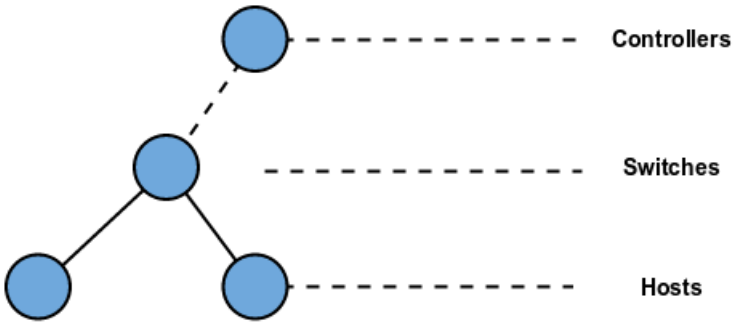


Figura 5.1: Topología creada con el comando *sudo mn*.

5.2.1. Historia y desarrollo

Mininet está basado en los procesos de virtualización mediante el espacio de nombres en Línux [47]. Gracias al espacio de nombres, Bob Lantz y Brandon Heller desarrollaron el primer prototipo de mininet, la versión 1.0. A partir de esta primera versión, gracias al esfuerzo de James Page (y el apoyo de *C*anonical y *Big Switch*), se creó el sistema de paquetes que permite la instalación de mininet en Ubuntu de forma sencilla. Este fue un gran paso que permitió acercar mininet a todos los usuarios.

Al ser un proyecto de tipo *OpenSource*, la colaboración estuvo garantizada desde el principio, consiguiendo mayor difusión, mayor aceptación y un mejor reporte de fallos y soporte. Con el esfuerzo de la comunidad y apoyos de varias organizaciones (Stanford u ON.lab por ejemplo), el desarrollo de mininet fue rápido, y se fueron sucediendo las diferentes versiones. La versión actual es la versión 2.2.1.

Mininet es desarrollado por contribuidores y miembros del núcleo. Los desarrolladores principales son seis, mientras que entre los contribuidores, podemos encontrar más de 27. Todo esto sin contar a todos aquellos usuarios de la comunidad que reportan fallos y/o intentan arreglar dichos fallos. proponiendo cambios a los miembros desarrolladores.

Mininet es otra muestra del crecimiento fugaz de los proyectos de tipo abierto, gracias al apoyo de la comunidad.

5.2.2. Comparativa

Mininet no es la única herramienta para emulación de redes, existen alternativas. En esta subsección vamos a comparar mininet con *EstiNet* [48], apoyándonos en los resultados presentados [49]. *EstiNet* es una herramienta que permite tanto emular como simular redes.

Un simulador es software que predice el comportamiento de una red de acuerdo a unos parámetros y modelos definidos. Los parámetros estarán, normalmente, definidos por el usuario, para adaptar el comportamiento simulado al comportamiento real que se desee. La simulación podrá simular fallos o cambios de red, en tiempo real o planificados, entre otras muchas posibilidades. Existe un gran número de simuladores que permiten predecir el comportamiento de casi cualquier protocolo de red en una situación definida. La predicción se lleva a cabo mediante el uso de modelos matemáticos sobre: fuentes de datos, canales, elementos y protocolos.

En contraposición a la simulación se encuentra la emulación. En emulación se pretende parecer una red real ejecutando, el mismo código que ejecutaría el dispositivo emulado, permitiendo así obtener datos y eventos ïdénticos.[a] los que sucederían en una red real. Las redes son imperfectas, aparecen eventos no previstos continuamente, afectando al resultado de los datos obtenidos. Emulando se pretende que esta parte aleatoria, que no puede ser controlada (solo estimada) en simulación, esté prevista y sea recogida en tiempo real.

Como resumen podemos destacar que simulación es una buena solución (básica a día de hoy para probar nuevas tecnologías), capaz de dar resultados fiables en entornos controlados. Además simulación es una solución muy escalable al poder ser todo controlado en un mismo equipo. Emulación introduce la probabilidad de que no todo vaya según lo previsto, al ser una ejecución en tiempo real, que recoge todos los eventos posibles en una red. Existen otras muchas características que podrían ser comparadas entre emulación y simulación, pero para nosotros, la fidelidad a la realidad (y la posibilidad de reproducir fallos o errores), es la más importante a la hora de comprobar el funcionamiento de la aplicación que queremos evaluar.

Para emular redes existen diferentes soluciones. En el artículo [49] se comparan *EstiNet* y mininet.

5.2.2.1. Emuladores de red.

Existen dos emuladores principales de red, *Mininet* y *EstiNet*. Mientras el primero de ellos es de tipo abierto y colaborativo, el segundo es una solución privada y cerrada. Esta será una razón importante para que se haya elegido *Mininet* como herramienta de emulación.

Los dos emuladores tienen una filosofía muy diferente. En EtsiNet cada vez que se genera un paquete en un *host*, este paquete es procesado por un *socket* Línux real del tipo *socket/TCP/IP*. Tras ser procesado por el *socket*, se crea un túnel que conecta con la emulación o simulación, y que procesa paquetes que se transportan sobre protocolos por debajo de IP. Estas capas de red emuladas, simulan todos los efectos que podría sufrir el paquete en una red real antes de enviarlo al destino. Los *sockets* Línux están integrados en el *kernel* del SO. Este modo de trabajo permite que cada *switch* emulado, pueda establecer una conexión TCP real con los controladores *OpenFlow* reales, imitando de forma fiel el comportamiento del protocolo *OpenFlow*.

Mininet por su parte recurre a la virtualización de elementos para conseguir la emulación de red. Para poder distinguir interfaces de tablas de *routing* y demás elementos, hace uso del espacio de nombres definido en Línux desde la versión 2.2.26 del *kernel*. Para la virtualización de *switches OpenFlow*, *Mininet* usa *Open vSwitch*, integrado totalmente en el *kernel* de Línux. Las conexiones *ethernet* también están virtualizadas gracias al *kernel*. Cada paquete inyectado en la red es procesado por la pila de protocolos de Línux, otorgando realismo al proceso. El único problema que puede encontrar *Mininet* es el de escalabilidad, ya que todos los elementos virtuales comparten la misma CPU, siendo en algunos casos muy complicado para el planificador de Línux cumplir todos los requisitos. En las últimas versiones de *Mininet* se incluye la posibilidad de conectar la red por alguna interfaz con el exterior, con lo que parte de la red podría ser emulada en otro equipo, solucionando este problema.

Según los resultados obtenidos en [49], se puede decir que el emulador EtsiNetm obtiene resultados muy precisos hasta que el número de elementos se vuelve demasiado elevado (961), a partir de este número de elementos, los resultados se desvían de los resultados esperados. El simulador EtsiNet obtiene unos buenos resultados en lo referente a escalabilidad y precisión, sin embargo la simulación ocupa un mayor tiempo que la emulación, además los recursos usados son mayores. Por último *Mininet* muestra un buen comportamiento, sin embargo (según el estudio), devuelve resultados inconsistentes bajo algunas condiciones de red (carga alta o casos especiales).

Los resultados obtenidos en el estudio [49] dan EtsiNet un paso por delante de *Mininet*. Sin embargo, la comunidad que apoya este último, la ayuda que puede ser encontrada y ser de tipo abierto, hacen de *Mininet* la mejor opción para el desarrollo de este proyecto.

Otra medida interesante para comparar las dos herramientas, es la que se obtiene al comparar los resultados obtenidos en Google a 18 de Junio de 2014. La búsqueda de los términos *EstiNet* y *Mininet* deja una comparativa sobre la relevancia de cada uno. El primero obtiene 16200 resultados en 0.3 segundos según Google. *Mininet* por su parte consigue 467.000 resultados en 0.4 segundos. Esta diferencia de casi 30 veces más resultados de *Mininet* frente a *EstiNet*, es una muestra de la aceptación de uno y otro.

Por último se quiere destacar que el blog desarrollado en paralelo a este trabajo, `http://www.aprendiendoodl.wordpress.com`, se sitúa en el puesto 15 de los resultados arrojados por Google para el término de búsqueda *Mininet*, siendo un dato interesante para evaluar el alcance que puede llegar a tener este proyecto.

5.2.3. Diseño de redes

Para el diseño de redes, *Mininet* cuenta con una API [50], escrita en Python, que permite diseñar y ejecutar casi cualquier tipo de red. Existen numerosos ejemplos por la red de uso, además de los que se encontrarán en este trabajo por lo que no vamos a tratar en profundidad como definir redes en *Mininet*.

5.3. *OpenDayLight*

En la sección 2.3.3 se ha descrito *OpenDayLight* de forma teórica, es el momento de comenzar a trabajar con la implementación software del controlador. En esta sección van a ser tratadas las versiones existentes del controlador, justificaremos la elegida, y finalmente haremos hincapié en las posibilidades de desarrollo de aplicaciones para éste.

Figura 5.2: Algunas de las clases de la *Mininet* Python API. Imagen tomada de [50]

5.3.1. Distribucciones y versiones

Desde la primera versión hasta hoy, el proyecto *OpenDayLight* ha ido lanzando nuevas versiones de su controlador. En tan solo dos años, el proyecto ha lanzado más de 5 versiones estables finales, lo que es una clara muestra del interés generado en torno al proyecto. Las versiones sucesivas han ido añadiendo características al núcleo de ODL, mejoras de funcionalidad y otras innovaciones. A continuación se han resumido brevemente las versiones disponibles para *OpenDayLight*.

5.3.1.1. Versiones Hydrogen.

Fueron las primeras versiones del controlador *OpenDayLight*, se lanzaron tres versiones diferentes a la vez: básica, virtualización y proveedor de servicios. Cada una de estas versiones estaba destinado a proyectos concretos como su propio nombre indica. Todas estas versiones están basadas en los siguientes conceptos:

- **Maven.** Maven permite a *OpenDayLight* una fácil automatización a la hora de construir aplicaciones. A través del fichero pom.xml de Maven, ODL sabe que módulos debe cargar e iniciar. Se describirá exhaustivamente en la sección 5.3.2.
- **OSGi.** OSGi es un framework que permite la carga dinámica de bundles y paquetes (archivos JAR). Además permite enlazar *bundles* para el intercambio de información. Se describirá ampliamente en la sección 5.3.2.
- **JAVA interfaces.** Son usadas para el intercambio de información y para la especificación de eventos de escucha. A través de JAVA se definen funciones *call-back*, que permitirán que posteriormente las aplicaciones sean llamadas cuando sucedan eventos, o que las propias aplicaciones generen eventos.
- **REST APIs.** APIs *Northbound* que permitirán a las aplicaciones la comunicación y obtención de información con el controlador (vía web).

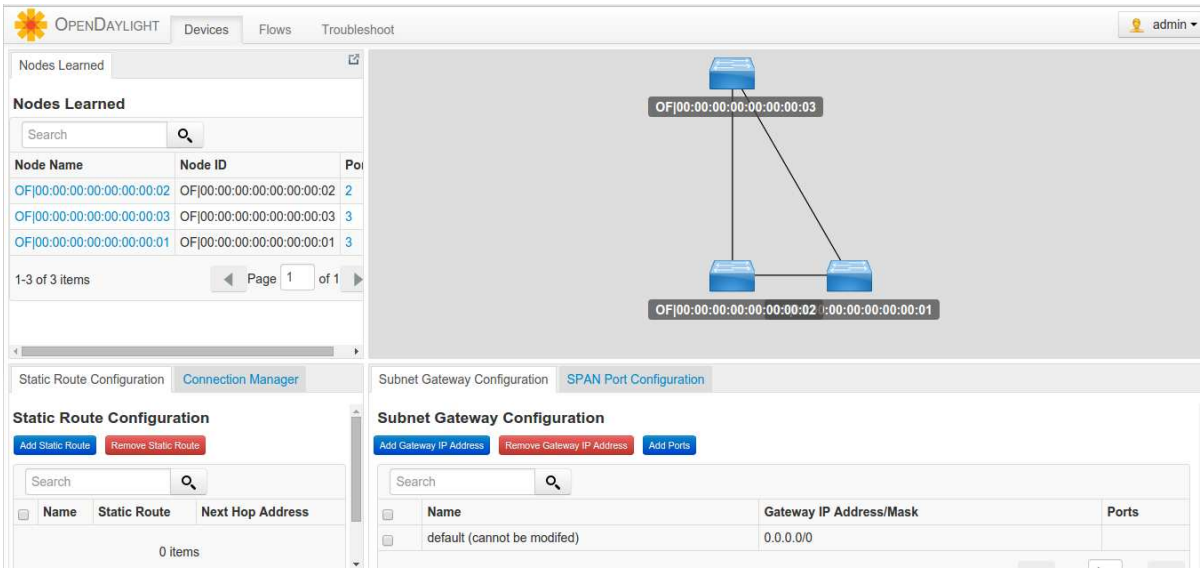


Figura 5.3: Interfaz web versión base ODL

Estas primeras versiones incluían ya un buen número de proyectos (ampliados en las siguientes), para facilitar y posibilitar el uso de capacidades de red, como por ejemplo control de la capa física u obtención de información acerca del estado de red. Además se incluyó desde el principio compatibilidad con las versiones 1.0 y 1.3 de *OpenFlow*. Entre las características iniciales también se incluyó una interfaz web que permitía el control de red de forma gráfica, como se aprecia en la imagen 5.3.

OpenDayLight siguió trabajando en el núcleo del controlador, lanzando sucesivas versiones de éste. Una de estas versiones fue elegida para realizar el proyecto, la versión base 0.2.2[3]. Se eligió esta versión por dos motivos: primero, al momento de comenzar este trabajo se acababa de lanzar la versión Helium (que describiremos a continuación), que suponía un cambio respecto a las anteriores, con lo que la información sobre esta versión era escasa y un tanto difusa. La segunda razón es que la versión elegida ofreció un buen funcionamiento desde el principio con los módulos propuestos en el trabajo en ese momento, lo cual marcó nuestra elección[4].

5.3.1.2. Helium.

Heliumfues la apuesta de ODL para hacer su controlador más accesible y cómodo al usuario. El núcleo del controlador siguió siendo el mismo que en Hydrogen (con las mejoras propias de las sucesivas versiones), pero en lo referente a la interfaz se produjo una revolución. Helium apostó por integrar karaf [51]. Karaf proporciona un contenedor de aplicaciones y componentes que se pueden implementar al mismo tiempo y de forma dinámica. Basado en OSGi, proporciona facilidad de uso cuando se usan un gran número de componentes y módulos.

[3]https://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/integration/distributions-base/0.2.2-SNAPSHOT/

[4]Existe una versión posterior a la 0.2.2, 0.3.0, que fue descartada por problemas con los módulos para instalar flujos.

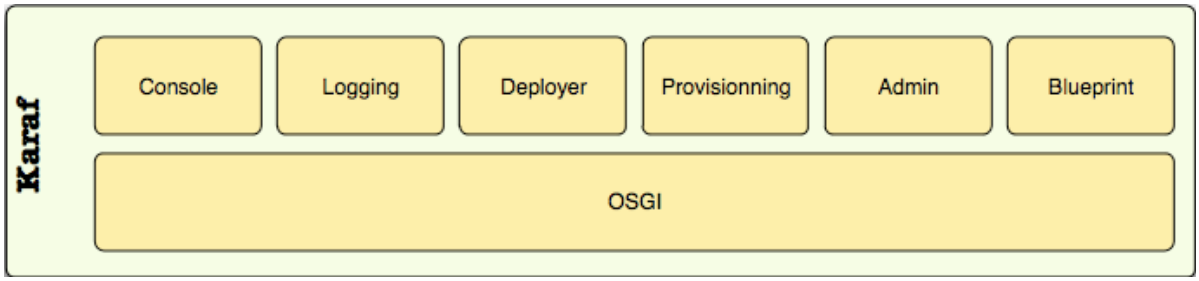


Figura 5.4: Esquema básico de karaf.

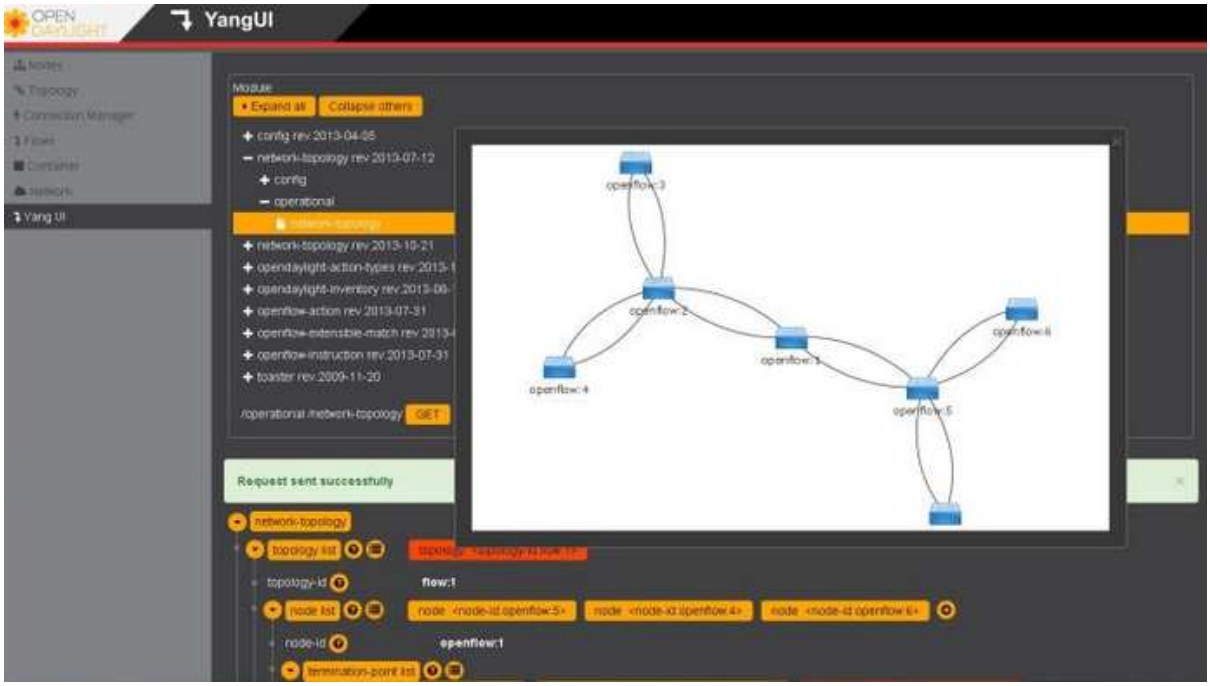


Figura 5.5: Interfaz web ODL con la versión Helium.

Además de este cambio en la forma de mostrarse al usuario, con Helium se lanzaron una serie de mejoras, como la inclusión de la interfaz web DLUX (una versión muy mejorada de la anterior interfaz), que permite un gran número de opciones de control y administración de red, como se aprecia en la figura 5.5. Muchos de los componentes del núcleo *OpenDayLight*, y muchos aplicaciones siguen en continuo desarrollado y pueden ser instaladas y actualizadas de forma sencilla gracias al uso de karaf.

Helium ha tenido un desarrollo muy rápido y se han ido sucediendo las versiones de forma fugaz (en menos de 6 meses han sido lanzadas 4 versiones del controlador), mejorando en todas ellas diferentes funcionalidades.

5.3.1.3. Lithium

El día 29 de Junio de 2015 se lanzó una nueva versión del controlador *OpenDayLight*. Debido a lo ajustado del calendario no se ha realizado un estudio sobre los cambios que introduce esta nueva versión, sin embargo se cree necesaria la inclusión de esta nueva versión en el documento.

Como ya se ha comentado, se ha elegido para trabajar la versión base 0.2.2 en este proyecto. Esto no debería suponer un impedimento (en teoría) para poder usar la aplicación desarrollada en este trabajo con cualquier otra versión *OpenDayLight*. Sin embargo pueden existir problemas de compatibilidad con las actualizaciones de algunos de los módulos usados. Una de las vías de trabajo futuras podría ser intentar desarrollar y/o adaptar los módulos desarrollados para las primeras en las nuevas versiones ODL, que serán más eficientes y tendrán un mayor número de funcionalidades.

5.3.2. Desarrollo

Existen varias opciones para llevar a cabo el desarrollo de una aplicación (o cualquier otra funcionalidad) en *OpenDayLight*. En las primeras versiones del controlador se usó una abstracción conocida como API Driven (AD)-SAL. AD-SAL, que permitía el desarrollo de aplicaciones para el controlador, sin preocuparse por el protocolo que controlará la red SDN. De este modo se posibilitó el desarrollo aplicaciones independientes del tipo de elemento de red, que posteriormente recibía las instrucciones generadas por las aplicaciones. En las primeras versiones del controlador esta capa de abstracción permitía obtener información acerca de los vecinos de un elemento de red, o las propiedades de una interfaz de cualquier elemento, entre otras muchas características.

Posteriormente se desarrolló Model Driven (MD)-SAL, una mejora de la anterior capa de abstracción. La abstracción por modelos se presentó como una oportunidad para unificar todas las APIs (*SouthBound* y *Northbound*), y las estructuras de datos usados en los componentes de un controlador SDN. Para describir las estructuras de datos se propuso un lenguaje de modelado, YANG [52]. Con este lenguaje de modelado se permitió modelar todas las estructuras de datos y funcionalidades proporcionadas por los componentes del controlador, definir relaciones entre elementos y modelar todos los componentes de un sistema. YANG es un lenguaje de naturaleza eXtensible Markup Language (XML). Utilizando un esquema de lenguaje simplificado como éste, se simplificó el desarrollo de componentes y aplicaciones para el controlador. Un desarrollador podía describir un módulo definiendo tan solo el esquema, y crear, a partir de dicho esquema, la funcionalidad mediante alguno de las APIs permitidas.

En la tabla 5.2 se resaltan las diferencias entre las posibilidades.

Para desarrollar una aplicación para el controlador usaremos MD-SAL, por todas las ventajas que genera. Para el desarrollo emplearemos algunas de las APIs definidas en [9].

5.3.2.1. DOM API [54].

DOM es una interfaz de programación para documentos HyperText Markup Language (HTML) y XML. Facilita una representación estructurada del documento y define de qué manera los programas pueden acceder para modificar estructura, estilo y contenido. DOM es una representación del documento como un grupo de nodos y objetos estructurados que tienen propiedades y métodos. Esencialmente, conecta las páginas web a *scripts* o lenguajes de programación.

Una página web es un documento. Éste documento puede mostrarse en la ventana de un navegador o también como código fuente HTML. En ambos casos, es el mismo documento. DOM proporciona otras formas de permite otra manera, guardar y manipular este mismo documento. DOM es una representación completamente orientada al objeto de la página web para ser modificado por *scripts* o código progamado.

Como API, DOM se desarrolló posteriormente a las primeras versiones de *OpenDayLight*, como respuesta a las necesidades planteadas por la comunidad. *OpenDayLight* evoluciona en función de las necesidades de los usuarios.

AD-SAL	MD-SAL
Las APIs de abstracción solicitan rutas entre consumidores y proveedores. Las adaptaciones se definen de forma estática al compilar.	Las APIs de abstracción solicitan las rutas mediante la definición de modelos, y las adaptaciones son realizadas en tiempo de ejecución.
Es necesaria una REST API dedicada para cada NorthBound (NB) y SouthBound (SB) plugin	Proporciona una REST API común para acceder a datos y funciones definidos en modelos
Provee servicios de adaptación para plugins	No provee de servicios de adaptación para plugins
Las peticiones de routing están basadas en el tipo de plugin. Cuando un plugin NB solicita una operación en un nodo dado, el plugin SB se encarga de encaminar a dicho nodo	Las solicitudes en MD-SAL se realiza de forma dinámica ya que los datos del nodo se exportan de forma automática al SAL
AD-SAL no comparte datos	Los datos de modelos definidos por plugins pueden ser intercambiados entre consumidores y proveedores a través del almacenamiento en MD-SAL
Los servicios en AD-SAL suelen proveer funciones síncronas y asíncronas	MD-SAL se enfoca hacia APIs asíncronas, permitiendo sin embargo bloquear procesos para posibilitar comunicaciones síncronas

Tabla 5.2: Diferencias entre las capas de abstracción AD-SAL y MD-SAL. Tabla tomada de [53]

5.3.2.2. REST API.

Cuando hablamos de REST nos referimos a cualquier interfaz entre sistemas que utilice directamente HyperText Transfer Protocol (HTTP) para la comunicación en cualquier formato (XML, JavaScript Object Notation (JSON), etc). En *OpenDayLight* REST se base en RestConf, [55] y [56], que es un protocolo implementado en el SAL de *OpenDayLight*. Los datos están definidos en YANG. El almacenamiento está definido por Network Configuration Protocol (NETCONF), usando *Restconf* (protocolo sobre HTTP). Lo que hace *Restconf*, es describir como mapear una especificación YANG para una interfaz REST.

La REST API proporciona una interfaz simplificada adicional para el uso de NETCONF, a través del protocolo *Restconf*. *Restconf* permite acceder a los *datastore* definidos por YANG en el controlador. Existen dos almacenes de datos, configuración y operacional.

En *OpenDayLight* la escucha de peticiones se hace a través del puerto 8080 (HTTP). Cuenta con las siguientes operaciones disponibles: OPTIONS, GET, PUT, POST, DELETE. Estas peticiones podrán ir en formato JSON o XML, siempre de acuerdo con el modelo YANG definido.

Uno de los problemas a la hora de usar REST API es su carácter pro activo, ya que no existirán funciones de aviso capaces de informar sobre eventos de red en tiempo real (no existen métodos de escucha soportados, al menos encontrados durante este trabajo), con lo cual no se podrá reaccionar de forma automática ante cambios en la red.

Figura 5.6: Logo maven

5.3.2.3. JAVA API.

JAVA es un lenguaje de programación orientado a objetos muy extendido. Gracias a los servicios de abstracción implementados en *OpenDayLight*, se permite que cada proveedor desarrolle su propia API para la programación. Las APIs de este tipo también incluyen funcionalidades para acceder a los datos mediantes REST, sin embargo (por conocimiento del lenguaje JAVA) se ha optado por realizar un desarrollo totalmente en JAVA sin hacer uso del resto de APIs. Haremos uso del JAVAdoc proporcionado por Cisco [37] (está definido para la adaptación de Cisco del controlador *OpenDayLight*, DevNet), que recoge todas las clases disponibles desarrolladas para el controlador. *OpenDayLight*, y las que además ha definido Cisco para su adaptación del controlador.

Se ha elegido para el desarrollo el uso de la JAVA API proporcionada por Cisco, por conocimiento del lenguaje de programación y por la posibilidad programar flujos de manera tipo reactiva, algo que es básico en recuperación de errores para QoE, uno de los objetivos de este trabajo.

5.4. Java

JAVA es un lenguaje de programación orientado a objetos, que puede ser ejecutado en cualquier SO gracias a la máquina virtual de JAVA. Es esta una de las principales razones por las que *OpenDayLight* está desarrollado sobre este lenguaje, la compatibilidad. Siendo JAVA uno de los lenguajes de programación que se han estudiado a lo largo de la preparación universitaria del autor, parece lógica su elección como lenguaje para programar aplicaciones ODL.

También se tiene en cuenta que las APIs de JAVA proporcionan las funcionalidades necesarias para alcanzar los objetivos requeridos en el proyecto.

Una vez escrito el código será necesario compilarlo para su posterior ejecución. Para esta tarea *OpenDayLight* especifica como compilador Maven.

5.4.1. Maven

Maven [57] se inició como una herramienta para simplificar la compilación de proyectos JAVA. Se pretendió crear un estándar que: definiera de forma clara en que consistía el proyecto, mantuviera organizada toda la información relacionada con éste y simplificara el modo en que el proyecto depende del resto de librerias o módulos (a través de los archivos Java ARchive (JAR)). El resultado es una herramienta que facilita el trabajo diario de los desarrolladores, ahorrando tiempo en la definición y compilación de proyectos, y simplificando las relaciones entre estos.

Una buena definición de lo que Maven ha conseguido se puede encontrar en [57]. Permitir al desarrollador comprender todo un proyecto completo (relaciones de clases por ejemplo), en el menor tiempo posible, mediante las características de Maven:

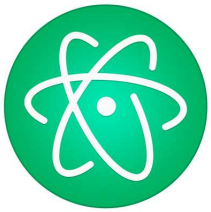


Figura 5.7: Logo de ATOM IDE

1. Facilitar el proceso de compilación.
2. Proporcionar un sistema de compilación uniforme para todos los proyectos.
3. Proporcionar información de calidad sobre el proyecto.
4. Proveer de unas líneas de buenas prácticas para el desarrollo.
5. Permitir nuevas características en los proyectos, de forma transparente.

Mientras estemos desarrollando un módulo ODL comprobaremos la utilidad de algunos de estos puntos, por ejemplo el último, ya que cada vez que hagamos un cambio y compilemos el código, automáticamente esta nueva característica se instalara en el controlador sin necesidad de ninguna acción extra por nuestra parte. También veremos que cualquier error en la programación del código será reflejado al intentar compilar, proporcionando una información concreta y precisa para intentar resolver el problema.

La programación para el compilador Maven hará uso del archivo Project Object Model (POM).XML. Este archivo es la unidad fundamental de trabajo Maven ya que en él se describen todos los detalles del proyecto, como la configuración o los repositorios donde se han de buscar librerías para el proyecto. También se especifican los parámetros de compilación (como donde instalar el JAR resultante) o donde se encuentran los archivos fuente. También incluye la posibilidad de definir archivos test que comprueben la validez del código implementado, ahorrando tiempo en el proceso de pruebas. También se especifica aquí información del proyecto, como la versión, descripción, autores, etc.

El archivo POM.XML es esencial para el desarrollo. Para facilitar su comprensión se incluye en el apéndice C.1 el resultante del desarrollo de este proyecto. No se incluye todo el codigo del proyecto debido a su extensión, pero podrá ser consultado en el repositorio del proyecto[5].

5.4.2. Atom

A la hora de desarrollar un proyecto de gran extensión se hace imprescindible el uso de un asistente para la programación o IDE. Existen muchos asistentes disponibles que además permiten la integración completa de Maven (como es el caso de eclipse [58], que además es uno de los más extendidos y completos). Sin embargo en la realización de este trabajo se ha optado por un IDE más ligero (en cuanto a número de complementos y posibilidades) como es Atom [59]. Atom es un asistente de desarrollo ligado a github (repositorio en el que está alojado el proyecto), con lo que la integración con el servidor de repositorios es total.

[5] `https://github.com/alfonsito92/serverVideoApp`

Otro de los aspectos por los que se ha elegido este IDE, es que no es necesario configurarlo para la compatibilidad con Maven, sino que simplemente nos podremos dedicar a programar el código. Además su simplicidad a la hora de instalar y usar en el SO elegido, permite que no sea necesario preocuparse por otro aspectos ajenos al desarrollo de este trabajo.

Por último al no integrar Maven ha sido necesario un mayor esfuerzo en cuanto al desarrollo se refiere, lo que a la larga ha permitido un mejor aprendizaje sobre el funcionamiento y características de la aplicación desarrollada.

Por supuesto se puede usar cualquier otro IDE e incluso existen guías de configuración, [60] para eclipse.

Haciendo uso de las herramientas y software descrito en este capítulo, se desarrollará todo el sistema que se describe en el capítulo 4.

Capítulo 6

Implementación de la solución

En el capítulo anterior nos hemos centrado en definir los elementos lógicos que compondrán el sistema. En este capítulo nos ocuparemos de describir la implementación del entorno de emulación y de la solución (tanto programación del bundle ODL como la topología y configuración de *Mininet*). Todas las herramientas que usaremos a continuación han sido configurados como se recoge en el apéndice A.1, A.2 y A.3.

6.1. Configuración del entorno de red SDN sobre *Mininet*

La solución propuesta se implementrá en una red SDN con *switches OpenFlow* y controlador *OpenDayLight*. Para que tanto el desarrollo como la evaluación sean ágiles y repetibles, se usa *Mininet* para desplegar una red emulada con todos los elementos descritos. Las topologías de red que se evaluarán serán diseñadas con la API de Python, tal como se describió en la sección 5.2.3. En los diseños nos deberemos ocupar de definir los elementos que formarán parte de la red (*hosts* y *switches*), así como indicar a qué controlador ha de administrar los *switches* (cada *switch* puede estar conectado a más de un controlador).

6.2. Descripción de los elementos de la red SDN implementada

Dentro de los equipos que compondrán la solución propuesta encontramos tres diferentes: *hosts*, *switches OpenFlow* y controlador. *Hosts* y *switches OpenFlow* estarán emulados gracias a *Mininet*, indicando la dirección IP del controlador (en nuestro caso estará ubicado en el mismo equipo físico).

6.2.1. *Hosts*

Los *hosts* son equipos finales emulados por *Mininet*, que podrán ejecutar cualquier comando o aplicación que esté disponible en el sistema Línux sobre el que se están ejecutando. Estos *hosts* podrán actuar como servidores web (HTTP) o clientes de vídeo *streaming*, como ocurre en las capturas 6.1 y 6.2.

```
root@Cristian-PC:~/mininet/custom# python -m SimpleHTTPServer 80 &
[1] 4168
root@Cristian-PC:~/mininet/custom# Serving HTTP on 0.0.0.0 port 80 ...
```

Figura 6.1: Ejemplo servidor web en *host* de *Mininet*

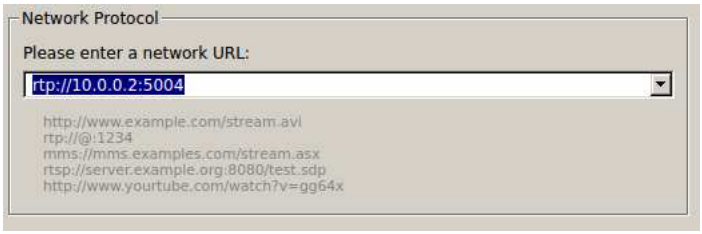


Figura 6.2: Ejemplo cliente RTP en *host* de *Mininet*

Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	OpenFlow	84	Type: OFPT_HELLO
127.0.0.1	127.0.0.1	OpenFlow	84	Type: OFPT_HELLO
127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_FEATURES_REQUEST
127.0.0.1	127.0.0.1	OpenFlow	100	Type: OFPT_FEATURES_REPLY
127.0.0.1	127.0.0.1	OpenFlow	200	Type: OFPT_PACKET_IN
127.0.0.1	127.0.0.1	OpenFlow	200	Type: OFPT_PACKET_IN
127.0.0.1	127.0.0.1	OpenFlow	238	Type: OFPT_PACKET_IN
127.0.0.1	127.0.0.1	OpenFlow	208	Type: OFPT_PACKET_IN
127.0.0.1	127.0.0.1	OpenFlow	84	Type: OFPT_HELLO
127.0.0.1	127.0.0.1	OpenFlow	84	Type: OFPT_HELLO
127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_FEATURES_REQUEST
127.0.0.1	127.0.0.1	OpenFlow	100	Type: OFPT_FEATURES_REPLY
127.0.0.1	127.0.0.1	OpenFlow	238	Type: OFPT_PACKET_IN

Figura 6.3: Ejemplo de comunicación *OpenFlow* entre *switches Mininet* y controlador.

El número de *hosts* disponibles dependerá de los definidos en la topología que hayamos diseñado. En la mayoría de nuestras pruebas habrá 2 (cliente y servidor), sin embargo podrán coexistir tantos como sean necesarios (es posible que a partir de cierto número de equipos, los resultados obtenidos varíen con los teóricos como se estima en [49]).

6.2.2. Controlador

El controlador ODL será ejecutado en la misma máquina física donde estará corriendo *Mininet*. Este controlador contará la aplicación diseñada ya instalada. También se eliminarán aquellos *bundles* que manejen paquetes, para evitar incompatibilidades. En el apéndice A.3 se realiza una guía sobre la configuración del controlador ODL.

La comunicación entre *switches OpenFlow* (siguiente subsección), se llevará a cabo haciendo uso del protocolo *OpenFlow*, de acuerdo a la figura 6.3. El controlador estará encargado de recibir los eventos que emitan los *switches*, comunicarlo a los módulos y aplicaciones instalados, y comunicar hacia los equipos las instrucciones requeridas por estas aplicaciones y módulos.

El controlador además será el encargado de recolectar (de forma automática) todas las estadísticas e información acerca del estado de la red, que posteriormente usaremos para la implementación del sistema.

6.2.3. Elementos intermedios. *Switches OpenFlow*

Los *switches OpenFlow* empleados por *Mininet* estarán encargados de conformar la red de interconexión y además conectar con los *hosts* finales. El encaminamiento en esta red vendrá dado por las indicaciones que el controlador comunique a los *switches*. Como se ha estudiado en la sección 2.2, estos *switches* contarán con tablas y flujos para realizar el encaminamiento.

Los *switches* deberán comunicarse constantemente con el controlador, para actualizar estadísticas e información del estado de red, además en caso de eventos desconocidos (o instrucciones que requieran comunicación con el controlador) también deberán informar al controlador.

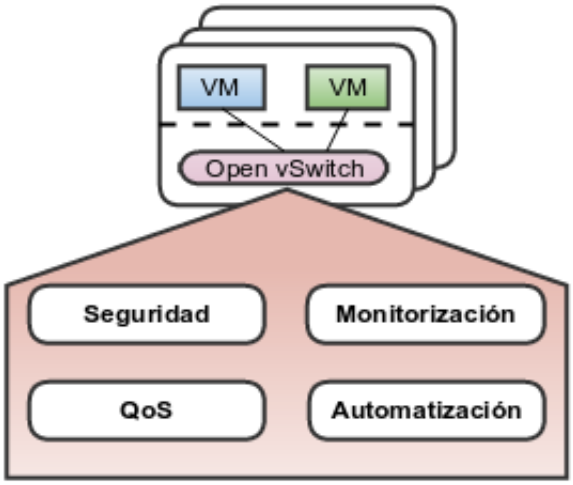


Figura 6.4: Esquema de Open vSwitch.

Estos equipos estarán emulados gracias a virtualización. El software que usa *Mininet* para virtualizar *switches* y *routers* es *Open vSwitch* [61]. *Open vSwitch* es un conmutador virtual multicapa diseñado para automatizar la red mediante el uso de protocolos como *OpenFlow*, pero sin dejar de lado otro tipo de protocolos de gestión. Otra de las características de su diseño es su filosofía de implementación real, ya que puede usarse en múltiples servidores físicos reales como los distribuidos por *vmWare*.

6.2.4. Red de interconexión

La red de interconexión hace referencia a los enlaces entre *switches OpenFlow*. Estos enlaces son provistos por la implementación en el *kernel* de Línux. Los enlaces serán de tipo Ethernet (no se ha estudiado por ahora la posibilidad de uso de enlaces de fibra óptica u otros tipos). De estos enlaces se podrá configurar un gran número de parámetros, como latencia, probabilidad de pérdidas o ancho de banda, entre otras.

El uso de este tipo de enlaces permitirá modificar la red en la que se evalúa el sistema, a fin de comprobar el comportamiento de la solución propuesta en diversos escenarios.

6.2.5. Generación de escenarios en *Mininet*

Para la evaluación del sistema se usarán topologías que variarán en función de los resultados requeridos en cada caso. Una de las más usadas será la topología 6.5, ya que cuenta con varios caminos para llegar al destino, con diferente número de saltos y permitirá comprobar la correcta evaluación de costes del capítulo 7.

En esta topología (y en otras con diferente número de saltos), variaremos los parámetros de cada enlace y obtendremos los resultados requeridos. El código utilizado para esta topología básica se encuentra en el apéndice B.2.1, con los siguientes parámetros:

$$BW = 10Mbps$$
$$latency = 10ms$$
$$loss(\,\%) = 0$$

Siguiendo este esquema se construirán todas las topologías necesarias para evaluar la solución propuesta.

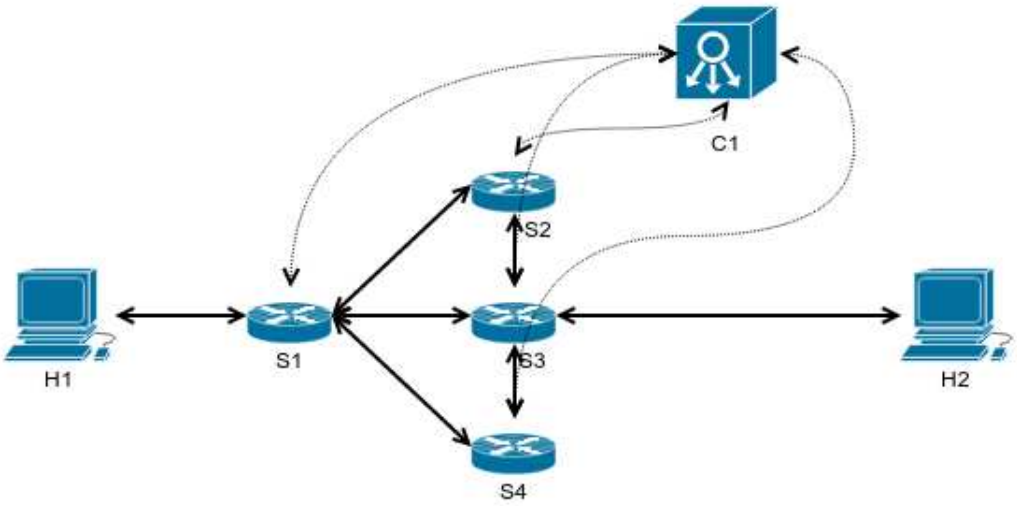


Figura 6.5: Topología básica para pruebas.

```
2015-06-22 12:21:07.001 CEST [remote-connector-processing-executor-3] INFO  o.o.
c.s.c.netconf.NetconfDevice  - RemoteDevice{controller-config}: Netconf connecto
r initialized successfully

osgi>
osgi>
```

Figura 6.6: Terminal mientras se está ejecutando el controlador ODL.

6.3. OpenDayLight

Para ejecutar el controlador tan solo lo descargaremos del enlace correspondiente[1]. Una vez descargado y descomprimido tan solo será necesario ejecutarlo y comprobar el correcto funcionamiento de todos los componentes. En caso de existir algún problema con la ejecución se aconseja consultar el apéndice A.3.

Si la ejecución es correcta tendremos en terminal una consolo OSGi (o karaf si es una versión superior[2]), y podremos ejecutar una topología de *Mininet* (como la topología básica del apéndice B.2.1), indicando que el controlador está en la dirección 127.0.0.1 (mismo equipo). Si todo va bien y se realiza la conexión podremos realizar Ping entre los *hosts* de la topología como en la imagen 6.7.

Además podremos comprobar que se han instalado flujos *OpenFlow*, de igual manera que se realiza en la imagen 6.8.

[1]Para la versión elegida 0.2.2 el enlace es `https://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/integration/distributions-base/0.2.2-SNAPSHOT/`

[2]Para obtener los mismos resultados con karaf será necesario instalar módulos adicionales para el manejo de paquetes

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=22.4 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=20.2 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=20.3 ms
```

Figura 6.7: Prueba de transmisión de ping con el controlador ODL.

```
cristian@Cristian-PC:~$ sudo ovs-ofctl dump-flows s1 -O OpenFlow13
[sudo] password for cristian:
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=18.565s, table=0, n_packets=4, n_bytes=392, priority=1,ip,
nw_dst=10.0.0.2 actions=output:3
 cookie=0x0, duration=20.954s, table=0, n_packets=3, n_bytes=294, priority=1,ip,
nw_dst=10.0.0.1 actions=set_field:00:00:00:00:00:01->eth_dst,output:1
```

Figura 6.8: Flujos instalados en S1 con el controlador ODL.

Si alguna de estas pruebas no resultase satisfactoria, en el apéndice A.4 pueden encontrarse diversas soluciones a problemas frecuentes. En caso no de resolver el problema, puede optar por visitar el blog del proyecto `http://www.aprendiendoodl.wordpress.com` y consultar las diferentes entradas, o dejar un comentario para que le intentemos ayudar.

Las aplicaciones preinstaladas que posibilitan la instalación de flujos (y que pueden causar conflictos con la desarrollada por no nosotros) son: *simpleForwarding* y *loadBalancer*. Podremos parar estos dos módulos (será necesario hacerlo cada vez que se inicie el controlador) o desinstalarlos por completo (permanente). Para parar los módulos tendremos que ejecutar (dentro de la consola OSGi):

ss *bundleName*

Lo cual nos devolverá el número de la aplicación. Con este número:

stop *bundleNumber*

Si queremos desinstalar los módulos podremos ir a la carpeta *plugins* (del controlador descargado), y eliminar los *bundles* necesarios. En esta carpeta será donde posteriormente instalaremos nuestra propia aplicación.

6.4. Implementación en java

Para desarrollar la aplicación que nos permita alcanzar los objetivos propuestos, usaremos el lenguaje JAVA. Como compilador se usará Maven. En esta sección vamos a describir como se ha implementado el código, como se compila con Maven y como instalar la aplicación en el controlador.

6.4.1. Desarrollo Bundle Maven para ODL

El desarrollo de un bundle para ODL usando Maven se divide en dos partes: creación del archivo POM.XML y creación los archivos de código fuente JAVA. Dado que los ficheros de código fuente son muy extensos, no se ha querido sobrecargar esta memoria con ellos. Pueden ser consultados en el repositorio `https://github.com/alfonsito92/serverVideoApp`. Si se incluye el fichero POM en el apéndice C.1.

Para facilitar la descripción de todas las clases implementadas, en la figura 6.9 podemos apreciar las relaciones entre clases que se han definido[3][4].

Además en el repositorio se incluye el JAVAdoc generado con todos los métodos (públicos y privados) que se han usado. El archivo generado es el *index.html* que se encuentra dentro de la carpeta *doc*.

[3]No se han incluido todas las clases ni métodos para facilitar la visión relación entre la clase principal y el resto.

[4]El resto de clases cuentan con los mismos métodos públicos (con diferente nombre) que la clase *RTPRouting*.

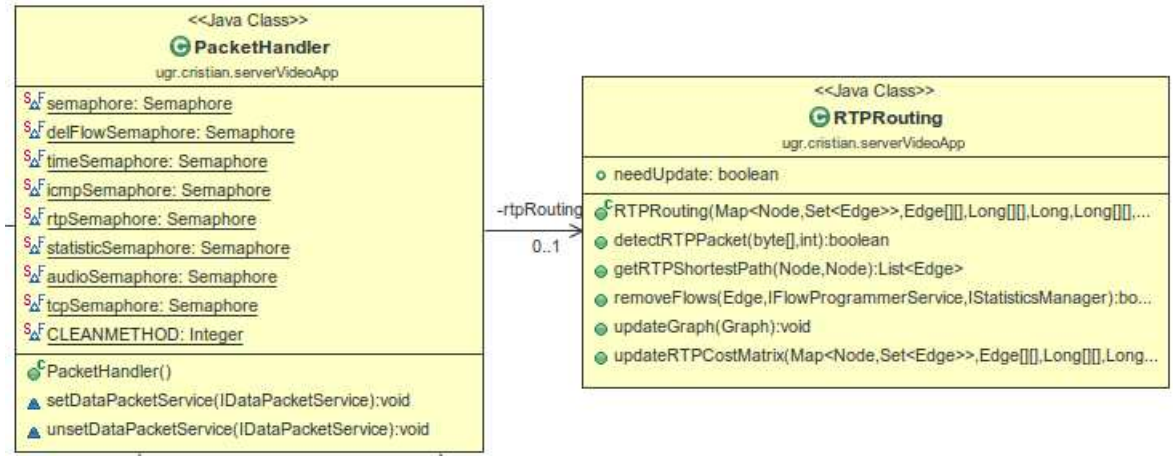


Figura 6.9: Relación UML entre PacketHandler y RTPRouting.

En este diagrama de clases se puede observar como se han añadido semáforos a la clase principal *PacketHandler.java*. Estos semáforos evitarán que se accedan a datos al mismo tiempo, para evitar por ejemplo que intentemos instalar flujos en un *switch* al mismo tiempo, o que se intente eliminar un flujo mientras se están leyendo.

Los archivos código fuente de JAVA para *OpenDayLight* se dividen en dos grandes grupos, activador y manejador. En [62] existen varias guías y ejemplos sobre como comenzar con el desarrollo de componentes OSGi para ODL.

El activador es el fichero donde se indica al controlador la clase manejadora y las relaciones con el resto de módulos *OpenDayLight*, como por ejemplo que la aplicación implementará la *interfaz de escucha de eventos de nuevo paquete*[5], o que hará uso de interfaces para consultas de estadísticas o instalación de flujos. En el apéndice C.2 se encuentra el fichero JAVA donde está definido el activador para este proyecto.

En nuestro caso será necesario implementar la funcionalidad para recibir eventos de nuevos paquetes, además deberemos hacer uso de:

- ***IDataPacketService.*** Esta interfaz permite la decodificación de paquetes (entrar dentro del paquete y analizarlo). Nos permitirá comprobar el tipo de paquetes.

- ***ISwitchManager.*** Con esta interfaz podremos consultar el estado actual de la red o las propiedades de un *switch*. Será muy importante a la hora de actualizar la topología en nuestra aplicación.

- ***IFlowProgrammerService.*** Clase que permitirá eliminar e instalar flujos en los diferentes nodos de la red mediante el protocolo *OpenFlow*.

- ***IStatisticsManager.*** Esta última clase será la encargada de permitir la recogida de estadísticas periódicamente, básica para la evaluación de costes.

Gracias al uso de estas 4 clases y a la implementación de *IListenDataPacket*, conseguiremos desarrollar la aplicación requerida. *IListenDataPacket* estará implementada en la clase principal *PacketHandler*.

[5]La interfaz *IListenDataPacket* permite implementar las acciones que realizará la aplicación cada vez que llegue un nuevo paquete al controlador.

6.4.1.1. *PacketHandler*

PacketHandler es la clase que define las acciones a realizar cada vez que llega un nuevo paquete al controlador. Debe ocuparse del cálculo de latencias, recogida y actualización de estadísticas, actualización de estadísticas e inicialización y llamada del resto de clases. Para facilitar esta tarea se ha dividido el código en un gran número de métodos que pueden ser consultados en el repositorio del trabajo. En el apéndice C.3.1 se puede encontrar la implementación de *IListenDataPacket*, que está descrita en la sección 6.4.1.2.

Además de la implementación y los métodos necesarios para el correcto funcionamiento de la aplicación, esta clase se ocupa de establecer las acciones cada vez que se inicia y para la aplicación, así como la inicialización de las interfaces descritas en la sección anterior.

En nuestra solución se propone que cada vez que se llame a la función *start* de *PacketHandler* se reinicien todos los parámetros que son usados durante la ejecución, para evitar que se almacenen datos de ejecuciones anteriores.

6.4.1.2. *IListenDataPacket*

En la implementación de esta interfaz es donde se realizan todas las llamadas a funciones que se realizan cada vez que un nuevo paquete es recibido. Se sigue un orden de trabajo muy esquemático, resumido en el diagrama de flujo de la figura 6.10, y que explicaremos a continuación. Cada una de las operaciones que están reflejadas en este diagrama seguirán el comportamiento descrito en el capítulo 4.

Los pasos que sigue la implementación son:

1. Actualización de topologías y estadísticas. Es lo primero que se ha de hacer. Se ha implementado de acuerdo al diseño realizado en la sección 4.15. Primero se comprueba si existen cambios en la topología, en caso afirmativo se eliminan los flujos afectados y se reinician las clases auxiliares con estadísticas y mapas actualizados. En caso negativo se actualizan estadísticas y latencias en los objetos auxiliares ya inicializados. La implementación la podemos encontrar en el apéndice C.3.2.

2. Comprobación de latencias. Se comprueba si el paquete está guardado en el mapa de latencias, en caso afirmativo se calcula la latencia y se introduce esta en las matrices y mapas especificados de acuerdo al diseño del capítulo 4. La implementación puede ser encontrada en el apéndice C.3.3.

3. Se comprueba el tipo de paquete gracias a los objetos correspondientes. En caso de ser un tipo desconocido el paquete será ignorado, si es uno de los conocidos, pasará al manejador que corresponda (ICMP, TCP, RTP Vídeo o RTP Audio).

4. Los manejadores de paquetes se encargan de calcular la ruta entre los dos nodos extremos (donde están los *hosts* conectados), gracias a la implementación de *Dijkstra* en cada objeto. Una vez devuelta la ruta, los manejadores se ocupan de instalar los flujos. La implementación del manejador RTP Vídeo se puede consultar en el apéndice C.4. En caso de cualquier error se intentará recuperar (reinicializando los objetos si es un error al obtener propiedades como el camino resultado), o se ignorará directamente el paquete.

   Si todo ha ido según lo previsto, el paquete se reenviará por la interfaz correspondiente (que sabremos gracias al cálculo de rutas) y se devolverá el atributo *PacketResult.CONSUME*, que indica que se ha procesado correctamente el paquete.

Para poder realizar este proceso en el tercer punto ha sido necesario hacer uso del algoritmo *Dijkstra* para el cálculo de caminos.

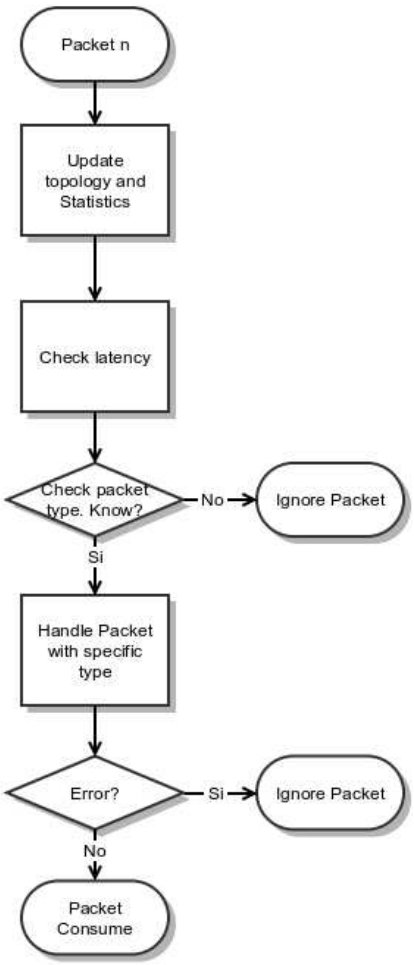


Figura 6.10: Diagrama de flujo de la implementación *IListenDataPacket*.

6.4.2. Obtención de las estadísticas de latencias

Las medidas de latencias se obtienen en nanosegundos, lo que permitirá realizar medidas precisas sobre los enlaces (que tendrán latencias superiores). Para estar seguros de que las funciones que devuelven valores en nanosegundos están funcionando de forma correcta, se ha realizado un estudio sobre la granularidad y resolución del equipo donde se está ejecutando el sistema. Para ello se ha consultado el JAVAdoc de la función *nanotime()* [63]. Además se ha consultado la resolución del reloj del sistema mediante llamadas del sistema Línux, obteniendo 1 ns. Como se explica en [63], la función *nanotime()* devuelve el tiempo actual del sistema con resolución de nanosegundos, pero esto no garantiza que entre dos llamadas consecutivas se obtenga el valor preciso del tiempo transcurrido, pues lo que se obtiene es un cálculo a partir de los ciclos del procesador. Incluso se podrían obtener valores negativos. Para comprobar el correcto funcionamiento se ha calculado una media con 20 llamadas consecutivas a la función *nanotime()* y se ha obtenido una media de 1.1 us de diferencia entre llamadas consecutivas. En ningún caso se ha superado una diferencia de 3us en llamadas sucesivas. Podemos concluir que la función utilizada, sin tener una precisión de 1ns entre llamadas, si que proporciona (en este equipo) precisión y granularidad suficiente para las medidas de latencia que se usarán posteriormente.

6.4.3. Implementación del algoritmo de *Dijkstra*

El algoritmo de *Dijkstra* está implementado en un gran número de lenguajes de programación dada su importancia. JAVA no es una excepción y existen múltiples implementaciones del algoritmo. Para nuestro trabajo se ha optado por hacer uso de la implementación que puede ser encontrada en [64], y que permite ser inicializada a partir de un mapa de nodos y enlaces (como el que devuelve directamente el controlador *OpenDayLight*) y un *Transformer*. El *Transformer* es una funcioón que a para cada enlace devuelve el coste asociado. La mayor dificultad en este apartado ha sido la de entender el concepto *Transformer*. En el apéndice C.4.1 podemos encontrar la construcción del *Transformer* para la clase RTP Vídeo.

Una vez inicializado *Dijkstra*, conseguir el camino más corto es simplemente hacer una petición al objeto *Dijkstra*, que devolverá el camino más corto. Es posible que el camino esté desordenado, por lo que será necesario hacer uso de una función para reordenarlo y asegurar que los nodos sean consecutivos (lo que permitirá la instalación secuencial de flujos en el camino). A continuación se incluye la implementación del algoritmo de reordenación de caminos.

6.4.3.1. Reordenación de caminos

De acuerdo al diseño de esta reordenación, se ha implementado una función capaz devolver un camino perfectamente consecutivo a partir de un camino desordenado. No se incluye la implementación de este método pues entendemos que a partir del diagrama de flujo de la figura 4.12 se comprende perfectamente el modo de operación del método.

6.4.4. Clases auxiliares

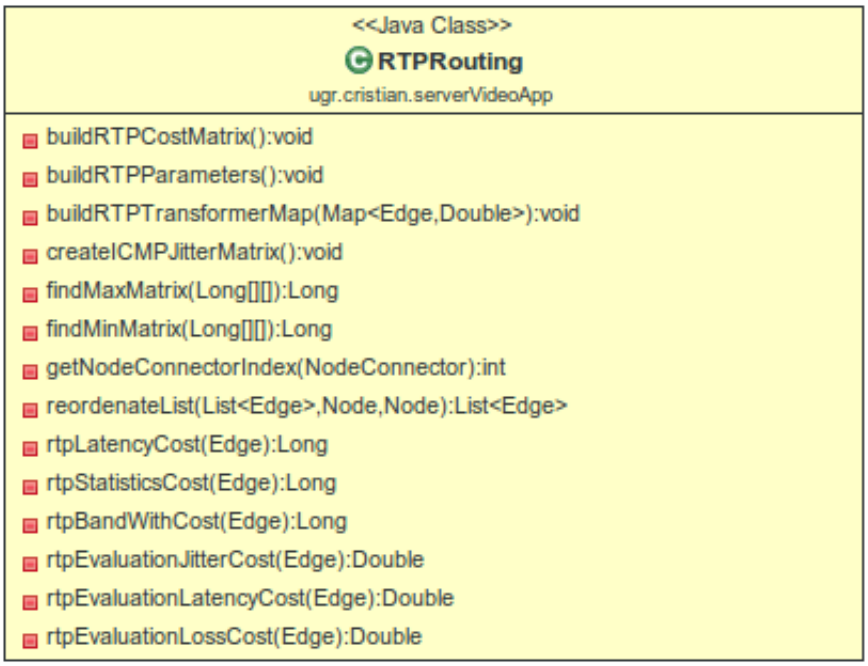
Por último se describen las clases auxiliares. Cada clase corresponde a uno de los protocolos soportados por la aplicación, a fin de mantener una mejor organización sobre el código escrito. Cada una de estas clases cuenta con el mapa de estadísticas global y con las matrices de latencias que posteriormente usarán para el cálculo de costes. Cuentan con un constructor (es idéntico para todas) y varios métodos públicos que permiten actualizar estadísticas y solicitar el camino más corto entre dos nodos. También se ha incluido un método para detección de tipo de flujo. Aparte de estos métodos, se incluyen varios más para agilizar el precesamiento interno (construcción de costes, inicialización del algoritmo *Dijkstra*, reordenación de caminos, etc). No se ha incluido la implementación completa de estas clases en el documento. Estas implementaciones podrán ser consultadas en el repositorio del proyecto.

En las figuras 6.11(a) y 6.11(b) podemos consultar los métodos públicos y privados implementados en las clases.

Todas las clases tendrán unos métodos muy similares a estos que permitirán calcular todos los parámetros necesarios para la posterior obtención del mejor camino posible entre dos nodos.

Con esto damos por concluido el capítulo de implementación, donde se ha presentado parte del código final de la aplicación diseñada. El código no incluido en apéndices puede ser consultado en el repositorio `https://github.com/alfonsito92/serverVideoApp`.

(a) Métodos públicos de la clase RTPRouting



(b) Métodos privados de la clase RTPRouting

Figura 6.11: Métodos implementados en la clase RTPRouting.

Capítulo 7

Evaluación de la solución

En esta capítulo se van a presentar los resultados obtenidos durante la evaluación del diseño e implementación propuestos. Para ello, se han llevado a cabo distintos experimentos sobre una red SDN emulada. Dichos experimentos comprenden: evaluación de los procedimientos de recogida de estadísticas, estimación de latencias, estimación de costes y cálculo de caminos para flujos de distinto tipo. Finalmente se evaluará la solución propuesta en un caso de aplicación real.

El capítulo estará dividido en cuatro secciones para facilitar el acceso a la información. Estas secciones son:

1. Evaluación de los módulos de recolección de parámetros de red, sección 7.1. Esta sección está dedicada a comprobar la validad de los datos recogidos sobre la red, estadísticas de enlaces y latencias de estos.

2. Evaluación del cálculo de costes, sección 7.2. A lo largo de esta sección se ajustarán los parámetros que marcan el peso de cada una de estas funciones. Por último se realizará una evaluación sobre la idoneidad de estos parámetros.

3. Evaluación del algoritmo de encaminamiento, sección 7.3. Realizaremos en esta sección un estudio sobre el algoritmo de encaminamiento, para comprobar que en la mayoría de casos se elige el camino más óptimo por cada tipo de flujo (de acuerdo a los parámetros definidos en la sección anterior).

4. Evaluación sobre la solución, sección 7.4. Se describen en esta sección los métodos escogidos para la evaluación objetiva de la solución implementada. Se realizarán diversas medidas (latencia, pérdidas, *jitter*, etc) sobre paquetes a través de la red. A partir de estas medidas se aplicarán diversos modelos para la estimación de la calidad de experiencia subjetiva del usuario final.

7.1. Evaluación de los módulos de recolección de parámetros de red

Las primeras pruebas realizadas se centran en la obtención de las estadísticas de parámetros de red, como son los retardos por enlace y las probabilidades de pérdidas correspondientes, para comprobar que estos módulos funcionan correctamente.

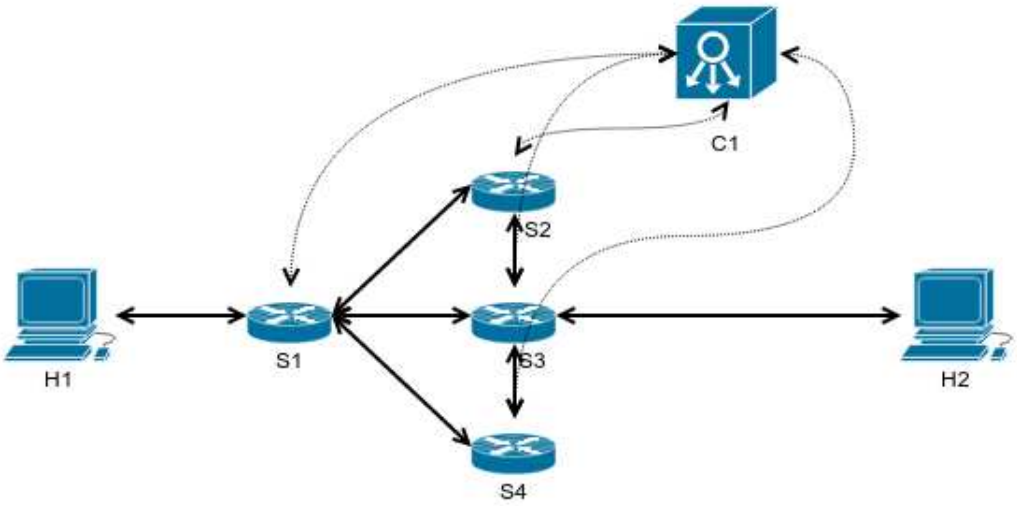


Figura 7.1: Topología para pruebas.

7.1.1. Descripción del entorno experimental

Las pruebas descritas a continuación se llevarán a cabo sobre la topología presentada en la figura 7.1. Las propiedades concretas de los enlaces se asignarán en cada experimento. Todos los *switches* que actúan en la topología soportan *OpenFlow*, al ser equipos emulados por *Mininet*. La configuración de estos equipos y enlaces se ha hecho de acuerdo al ejemplo que se puede consultar en el apéndice B.2.1 como ya se describió en la sección 6.2.5.

Como se puede observar, esta topología presenta dos equipos finales. El tráfico será generado por H1, mientras que H2 recibirá paquetes, en caso de que estos consigan ser encaminados por la red. Estos equipos finales estarán conectados entre si gracias a la red de interconexión intermedia, compuesta por cuatro *switches OpenFlow*, conectados entre si por enlaces *Ethernet* (los parámetros de los enlaces serán modificados en función del experimento a realizar). Estos *switches* dependerán del controlador C_1, donde estará instalada la solución propuesta que se encargará de encaminar paquetes.

En esta topología se ha elegido una distribución en la que para encaminar nos encontramos con soluciones con distinto número de saltos, a fin de demostrar que el algoritmo no siempre encaminará por el camino con menor número de saltos, sino que elegirá el camino más óptimo.

7.1.2. Estadísticas de tráfico

En este experimento se trata de comprobar si el módulo de recogida de estadísticas de paquetes enviados implementado funciona correctamente. Es decir, de si el número de paquetes y bytes enviados por cada enlace coincide con el tráfico real enviado.

	Bytes transmitidos	**Bytes recibidos**
Enlace S3-S1 t0	252	252
Enlace S3-S1 t1	2065	2163

Tabla 7.1: Estadísticas capturadas para el enlace S3-S1.

Para ello se envían paquetes ICMP entre los nodos H1 y H2 durante 10 segundos con la herramienta de red *ping*. Esta herramienta envía un mensajes ICMP del tipo *echo request* a un dispositivo IP, y espera a recibir la respuesta de dicho dispositivo, un mensaje ICMP del tipo *echo reply*. La tasa de envío será un paquete cada segundo, hasta un máximo de 10. En la práctica el número de paquetes enviados podrá variar ya que el experimento finalizará cuando se hayan recibido 10 confirmaciones de *ICMP echo reply* positivas, no cuando se hayan enviado los 10 paquetes ICMP. Previamente se instalará una ruta entre H1 y H2 que siga el mínimo número de enlaces, y comprobaremos que el mapa de estadísticas generado por nuestro módulo coincide con el patrón de paquetes enviados. En esta prueba la topología no varía durante el experimento.

El estudio se centrará en realizar medidas de las estadísticas en el enlace entre los nodos S1-S3, en ambos sentidos. Se tomará en un instante de inicial t_0 una captura de las estadísticas recogidas por el módulo implementado. A continuación se iniciará la captura de paquetes en la interfaz del nodo S1 mediante la herramienta *Wireshark*. En ese instante se comenzará a enviar mensajes ICMP durante 10 segundos. Pasado ese periodo, en el instante t_1, se detendrá la captura de *Wireshark*, y se tomará una captura sobre las estadísticas en ese instante.

Finalmente se comparará la diferencia de paquetes y cantidad de datos registrados en los instantes t_1 y t_0. Si el número de bytes enviados por las interfaces que reporta *Wireshark*, coincide con el identificado por los módulos implementados, podremos decir que la obtención de estadísticas se realiza de modo correcto.

7.1.2.1. Resultados experimentales

Lo resultados de este primera prueba pueden verse en la tabla 7.1, donde se muestran los valores registrados y capturados. Como puede apreciarse, tras el paso del periodo $t_1 - t_0$, y el envío de 10 mensajes ICMP, (figura 7.2(a)), se registran 2065 bytes enviados desde el nodo S3 al S1, y 2163 del nodo S1 al S3.

Sin embargo, en *Wireshark* se han detectado 16 mensajes ICMP recibidos en S3 (de H1 hacia H2, es decir de S1 a S3), y 13 mensajes ICMP transmitidos hacia S1, como se ilustra en la captura de pantalla de la figura 7.2(b). Analizando detenidamente las trazas de tráfico de red, se llega a la conclusión de que varios de estos paquetes se detectan al haber sido inundados para el entrenamiento de la red. En el caso en que no se inundara la red, se deberían detectar sólo los 10 paquetes. Es por ello que existe diferencia entre los bytes registrados en el módulo de recolección de estadísticas y los capturados mediante *Wireshark*. Además, se ha comprobado que existe tráfico de fondo de protocolos de gestión y mantenimiento de la red, como Dinamic Host Control Protocol (DHCP) e ICMP, en enlaces por los que no se ha enviado tráfico alguno. Se observa que este tráfico es simular para todas las interfaces, como se comprueba en las capturas de la consola del controlador mostradas en las figuras 7.3(a) y 7.3(b) (alrededor de 700B de diferencia entre los instantes t_1 y t_0), que es un valor próximo al exceso de bytes obtenidos en la tabla de resultados. Se trata pues de un incremento de 70B/s, por lo que concluimos que la recogida de estadísticas, aún sin ser exacta, es lo suficientemente buena como para poder ser usada en el cálculo de costes.

```
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=349 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.32 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.145 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.155 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.198 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.165 ms
64 bytes from 10.0.0.2: icmp_seq=7 ttl=64 time=0.160 ms
64 bytes from 10.0.0.2: icmp_seq=8 ttl=64 time=0.164 ms
64 bytes from 10.0.0.2: icmp_seq=9 ttl=64 time=0.151 ms
64 bytes from 10.0.0.2: icmp_seq=10 ttl=64 time=0.104 ms
^C
--- 10.0.0.2 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9004ms
rtt min/avg/max/mdev = 0.104/35.176/349.191/104.672 ms
```

(a) Captura de las ejecución de la herramienta *ping* para la validación de estadísticas

Address	Packets	Bytes	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes
10.0.0.1	29	2 842	16	1 568	13	1 274
10.0.0.2	29	2 842	13	1 274	16	1 568

(b) Suma total de paquetes capturados con *Wireshark* durante el experimento.

Figura 7.2: Paquetes enviados(desde *Mininet*) 7.2(a) y capturados con *Wireshark* 7.2(b).

```
(OF|2@OF|00:00:00:00:00:00:00:03->OF|2@OF|00:00:00:00:00:00:00:02)
, Transmits Bytes=[252, 252], Receive Bytes=[5116, 4774], Receive
```

(a) Estadísticas en t_0 en el enlace S3-S2.

```
(OF|2@OF|00:00:00:00:00:00:00:03->OF|2@OF|00:00:00:00:00:00:00:02)=
, Transmits Bytes=[987, 1183], Receive Bytes=[6838, 6642], Receive
```

(b) Estadísticas en t_1 en el enlace $3-2$.

Figura 7.3: Diferencia sobre bytes transmitidos entre t_0 (a) y t_1 (b) en enlace $3-2$.

7.1.2.2. Efecto de tráfico de fondo

Se ha detectado que la cantidad de datos capturados que no pertenecen al tráfico ICMP es significativa durante el periodo evaluado. Para comprobar la validez de los resultados obtenidos, y que el número de paquetes de tráfico de fondo no contemplados, no afecta demasiado al módulo de recolección de estadísticas, se ha realizado el estudio que se presenta en la tabla 7.2. En este experimento se repite el escenario de la sección anterior, con la excepción de que se envían de paquetes de vídeo sobre RTP.

	Bytes transmitidos t_0	**Bytes transmitidos t1**	**Diferencia bytes periodo** $t_1 - t_0$	**Bytes detectados *Wireshark* (RTP)**	**Diferencia bytes enviados-detectados**
Enlace S1-S3	31 603 084	70 612 838	39 009 754	39 002 530	7 224

Tabla 7.2: Estadísticas recogidas enlace S1-S3 para el envío de vídeo sobre RTP.

Latencia media (ms)	S1	S2	S3	S4
S1	0	51.9ms	303.2ms	102.0ms
S2	70.6ms	0	203.1ms	—
S3	303.5ms	203.1ms	0	152.8ms
S4	129.8ms	—	153.0ms	0

Tabla 7.3: Latencias medias obtenidas en ms. Primera prueba.

La diferencia entre la cantidad de paquetes y de datos identificados con *Wireshark* y los recogidos en las estadísticas en este caso es mucho menos relevante que la detectada con tráfico ICMP en la subsección 7.1.2.1. En este caso, la duración del experimento ($t_1 - t_0$) es de es de aproximadamente un minuto. En este caso, el error incurrido supone apenas un 0,00185 % del total de los datos transmitidos por el enlace $S1 \rightarrow S3$.

Con la revisión de estos resultados se puede afirmar que la obtención de estadísticas se realiza de forma correcta y permitirá el cálculo de costes de forma correcta y sin añadir error o desviación en los datos.

7.1.2.3. Errores en la estimación de la probabilidad de pérdida de paquetes

Se ha detectado que las estadísticas muestran una diferencia muy significativa entre los bytes recibidos y los enviados, medidos en los extremos de un mismo enlace, siendo superiores los bytes recibidos. A la hora de evaluar las pérdidas esto supone un obstáculo, debido a que estos paquetes (no controlados por el usuario) aumentan la diferencia en estadísticas. No obstante, la influencia de esta disimilitud entre datos recibidos y enviados no es relevante una vez que la red cuenta con una carga de tráfico elevada. Esto se da en cuanto transcurren unos segundos de envío de vídeo RTP. Se ha observado que este problema provoca falsas estimaciones de coste en los primeros instantes de red. Finalmente se ha optado por aumentar el período de entrenamiento en la red (se requieren 100 paquetes por cada nodo en la red) al inicio de esta, para conseguir un mayor número de muestras, y de este modo aquellos enlaces con una alta probabilidad de pérdidas son detectados.

7.1.3. Estimación de latencias

Una de las medidas más complejas y a la vez necesaria para la evaluación de pesos y costes en nuestra solución, es la referente a latencias. Siguiendo el proceso descrito en la subsección 4.4.1, se van a obtener las latencias entre enlaces en la topología de la figura 7.4, que sigue una estructura muy parecida a la topología 7.1, pero modificando los parámetros de latencia (para poder comprobar las correctas medidas). Nótese que los enlaces que comunican los *hosts* H1 y H2 con la red no se modifican, pues no son parte de los cálculos de costes ni de caminos de red.

En estas pruebas, el módulo recolector de estadísticas de latencias recopila 10 valores para calcular la media de la latencia. Se ha comprobado que a partir de este número de repeticiones, los valores de retardo se estabilizan. El número de repeticiones puede configurarse fácilmente.

En este escenario se ha realizado una prueba que sobrecarga la red, ya que se configura cada *switch* para que reenvíe los paquetes recibidos por todos los puertos salvo el de entrada. Esto implica la medida de paquetes en una situación de estrés, y comprobando la validez de los resultados. Hay que tener en cuenta que esta sobrecarga de tráfico se traduce en espera en las colas de los dispositivos de red, y posibles retardos adicionales.

Los resultados de esta prueba se pueden ve en la tabla 7.3, como valores medios. Una de las capturas de estos experimentos pueden consultarse en las figuras 7.5(a) y 7.5(b).

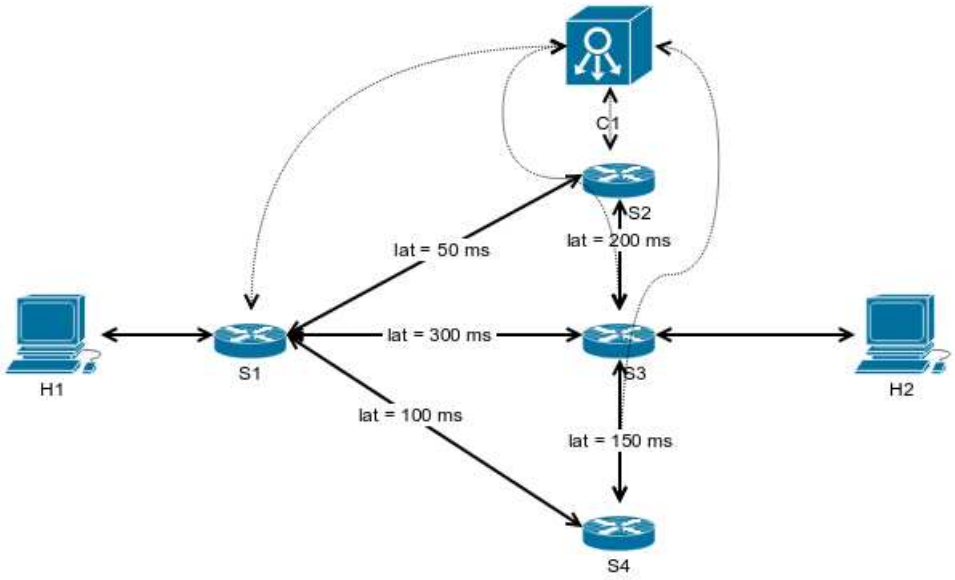


Figura 7.4: Topología de prueba para la medida de latencias.

Latencia instantánea	**S1**	**S2**	**S3**	**S4**
S1	0	73.63	326.755	120.76
S2	124.84	0	230.685	—
S3	292.52	213.84	0	156.29
S4	158.63	—	187.84	0

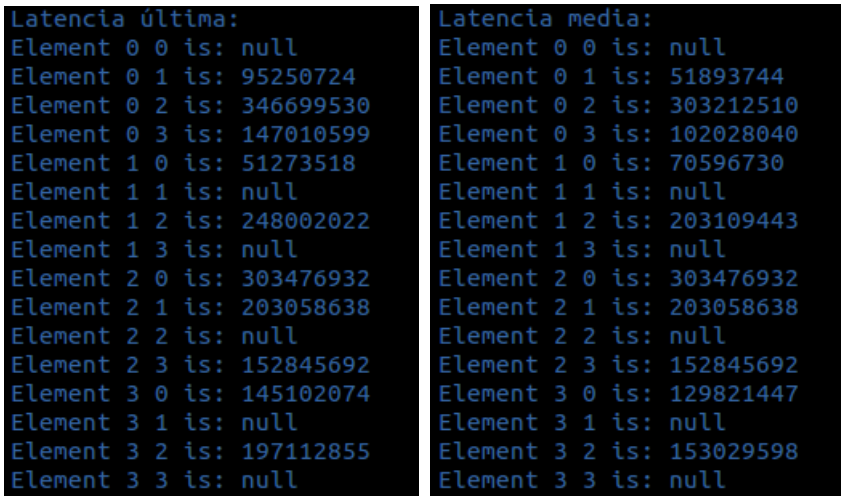
Tabla 7.4: Media de latencias instantáneas. 5 experimentos.

De los resultados obtenidos se puede destacar rápidamente que existe diferencia entre la latencia definida en la topología y la obtenida por nuestras medidas. Esta diferencia se sitúa en torno a los 3.5 milisegundos. Esta variación se describe en la sección 4.4.1. Para estimar con exactitud este valor se han realizado varios experimentos y se ha calculado el valor medio de latencia. Estos resultados se recogen en las tablas 7.4 para latencias instantáneas, y 7.5.

Otro de los resultados destacables es que las latencias medidas en los caminos de "vuelta", son mayores a lo esperado (enlaces S2-S1 y S4-S1), estos resultados están debidos a la sobrecarga en el *switch* S1, producida por el procesamiento de paquetes que provienen del resto de enlaces y que han de ser enviados al controlador, lo cual provoca un retardo al enviar los paquetes hacia el controlador y que éste obtenga la latencia. Será uno de los puntos a tener en cuenta para la evaluación, de ahí que sea necesario el uso de valores medios de latencia.

Latencia media	**S1**	**S2**	**S3**	**S4**
S1	0	53.02	301.972	102.796
S2	57.52	0	203.44	—
S3	297.516	200.83	0	148.236
S4	107.794	—	152.978	0

Tabla 7.5: Media de latencias medias. 5 experimentos.



(a) Última medida de latencias en nanosegundos.

(b) Latencias medias medidas en nanosegundos.

Figura 7.5: Latencias instantáneas 7.5(a) y media 7.5(b) sobre la topología 7.4 en ns.

A partir de la latencia media se calcula la variación media respecto al valor ideal, resultando ser esta variación a 3.46 ms.

1. La medida de latencia obtenida (exceptuando enlaces con gran sobrecarga), es muy próxima a los valores teóricos, añadiendo un tiempo de procesamiento por parte del controlador en torno a 3.46 ms en cada enlace.

2. Las condiciones de sobrecarga son soportadas por los enlaces hasta cierto punto, siendo una posible fuente de error la sobrecarga de enlaces entre *switch* y controlador, que puede provocar un aumento de las latencias obtenidas y que debe ser considerada en el proceso de evaluación.

3. En diferencias grandes entre la latencia de los enlaces este método puede ser usado con garantías. Sin embargo cuando las diferencias se reducen y nos movemos en márgenes estrechos (menos de 10 ms con mucha carga de tráfico), la fiabilidad de las medidas decae, y aunque los tiempos de procesamiento pueden ser estimados y corregidos (o simplemente ignorados ya que afectan por igual para todos los enlaces), el retardo provocad por sobrecarga será muy significativo.

4. Se espera la mejora de los módulos del controlador ODL que permitan obtener la latencia de los enlaces de forma precisa y sin depender de métodos o módulos externos, como en este caso. Al ser *OpenDayLight* un controlador en desarrollo, se espera que estos módulos se encuentren ya implementados en futuras versiones del software.

7.2. Ajuste experimental de los coeficientes α_i, β_i, γ_i y σ_i.

Estos 4 valores son los que determinan la importancia (peso) de cada una de las funciones de coste. Gracias a estos cuatro parámetros, cada enlace tendrá un coste diferente dependiendo del tipo de tráfico que se esté evaluando. Así por ejemplo podremos dar prioridad a enlaces con una baja probabilidad de pérdidas para flujos TCP, o con una latencia menor para flujos VoIP.

En el desarrollo del proyecto se ha decidido establecer que la suma de los cuatro parámetros no ha de ser igual a ningún valor específico, sino que se puede adaptar según cada caso. Esta decisión es una propuesta que puede ser modificada dependiendo de las necesidades de cualquier usuario o red. A partir de estas premisas se ha llegado a las siguientes conclusiones.

Por último destacar que σ_i no ha sido tenido en cuenta, pues la función de evaluación de carga no se ha implementado, con lo que directamente asignamos un valor 0 a este parámetro.

7.2.1. Requisitos de QoS para el tráfico considerado

- **Audio RTP (VoIP).** La transmisión de paquetes de audio en una red. es una de las más estrictas en cuanto a calidad de servicio. Se deben cumplir ciertas condiciones para que la comunicación se pueda considerar aceptable. Estas consideraciones se han presentado en diferentes publicaciones y están resumidas en la tabla 7.6 extraída de [65].

	Latencia	**Jitter**	**Pérdidas**
Máximo permitido	150 ms one-way	20 ms	<2 % Dependiendo del códec

Tabla 7.6: Requerimientos VoIP QoS

  A partir de estos datos concluimos:

  1. En comunicaciones VoIP la importancia de la latencia es muy importante, siendo junto a las pérdidas el factor de mayor importancia (el *jitter* podrá ser compensado en redes con latencias bajas).
  2. El umbral máximo de pérdidas dará un coste 5 (5 % de pérdidas), mientras que la latencia se moverá en valores que no podrán ser controlados, pero que variarían el coste de acuerdo a los umbrales máximos y mínimos.
  3. Será necesario aumentar la importancia (peso) del coste de pérdidas frente al resto.

- **TCP.** TCP engloba demasiados servicios como para poder realizar una estimación igual a la realizada para VoIP. Existen demasiados servicios diferentes que van sobre este protocolo como para estimar unos requerimientos mínimos. Lo que si es seguro es que TCP es un protocolo donde la pérdida de paquetes afectará en el rendimiento final de forma considerable. En menor medida suelen ser importantes *jitter* y latencia.

- **ICMP.** El protocolo ICMP ha sido usado para comprobar: la conectividad entre nodos de red, obtener medidas de latencia experimentales en la red y realizar comprobaciones de caminos. Sobre este protocolo no se establecen medidas de calidad de servicio o restricciones temporales, así como el *jitter* tampoco afecta demasiado a la calidad de experiencia.

- **RTP Vídeo.** Este protocolo de transporte permite enviar *streaming* de vídeo en tiempo real, siendo esta en si una restricción. En [1] se recogen algunos de los requerimientos mínimos para envío de *streaming* de vídeo. Se ha de tener en cuenta que el códec utilizado marcará enormemente los requerimientos (sobre todo ancho de banda), pero debido a que se ha descartado la evaluación de costes en función del ancho de banda, no se tendrá en cuenta este aspecto, pudiendo implementarse posteriormente. El resumen de requerimientos está recogido en la tabla 7.7.

	Latencia	**Jitter**	**Pérdidas**
Máximo permitido	<10 s	<2 s	<2 % Dependiendo del códec

Tabla 7.7: Requerimientos QoS en transmisión de vídeo según [1]

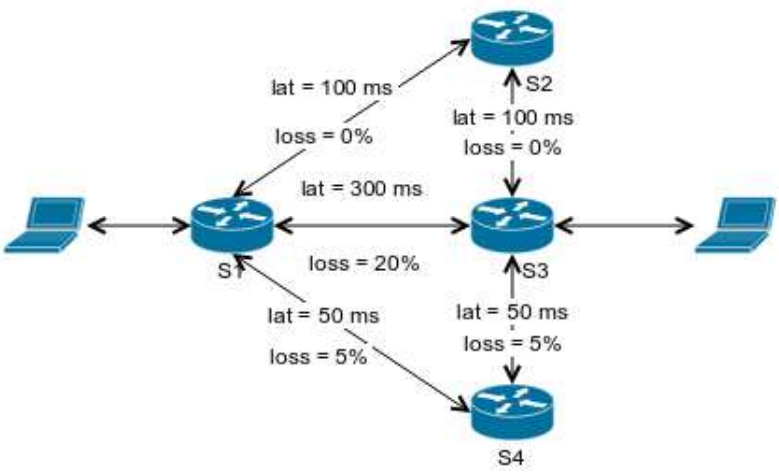


Figura 7.6: Topología para pruebas *Dijkstra*.

Estas restricciones se combinarán con el estudio teórico de la subsección 7.2.1.1 para estimar establecer el valor de los parámetros en cada caso.

7.2.1.1. Ajuste de las variables α_i, β_i y γ_i

Para el estudio de estas variables usaremos la topología de la figura 7.6, que se describe a continuación.

- **Descripción del entorno experimental.**

  La topología 7.6 se ha diseñado para la evaluación del algoritmo *Dijkstra*.

  En esta topología (que tiene la misma estructura que la ya descrita 7.1), se han seleccionado los siguientes parámetros para forzar que la diferencia de costes sea elevada.

$$l_{S1-S2} = 100ms$$
$$loss_{S1-S2}(\,\%) = 0\,\%$$
$$l_{S1-S3} = 300ms$$
$$loss_{S1-S3}(\,\%) = 20\,\%$$
$$l_{S1-S4} = 50ms$$
$$loss_{S1-S4}(\,\%) = 5\,\%$$
$$l_{S2-S3} = 100ms$$
$$loss_{S2-S3}(\,\%) = 0\,\%$$
$$l_{S4-S3} = 50ms$$
$$loss_{S4-S3}(\,\%) = 5\,\%$$

Los enlaces son simétricos. Es decir, S2-S1 tiene los mismos parámetros de transmisión que S1-S2.

- **Estudio de variables**

  Se ha realizado un estudio sobre la elección de caminos en la topología 7.6, dejando el *jitter* de todos los caminos a un valor fijo 5, con lo que su coste asociado (devuelto por la función de evaluación) será el mínimo seleccionado (por defecto 5 en todos los casos).

  Además se ha descartado el cálculo sobre el ancho de banda (por las razones ya explicadas). Los resultados los podemos consultar en las tablas 7.7(a), 7.7(b) y 7.7(c). En estas tablas se han calculado los costes y el camino elegido para la topología 7.6 con los valores α β γ que se indican en las tablas. En cada tabla se recoge el camino mínimo para llegar desde el nodo 1 al nodo 3. El estudio ha finalizado en el momento en el que se ha observado una tendencia clara.

  Podemos observar que cuando el peso de la latencia es importante, el coste devuelto a la evaluación de pérdidas (pese a tener una mayor importancia), apenas tiene relevancia. Además se debe tener en cuenta que se han establecido unas pérdidas del 5 % (superiores al máximo teórico que hemos visto en la sección 7.2.1), con unas pérdidas inferiores la relevancia sería aún menor.

7.2.1.2. Estimación de valores α_i, β_i y γ_i

A partir de los resultados obtenidos podremos establecer que para que las pérdidas tengan relevancia en el cálculo del mejor camino, deben equipararse a los costes que devuelven las evaluaciones de latencia y *jitter*. Según el caso de necesidad (pérdidas menores al 2 % en RTP por ejemplo), será necesario que el valor γ sea varias veces el valor del resto de parámetros.

- **ICMP.** Este protocolo no exige requisitos de calidad de servicio, con lo que se propone unos pesos estándar para cada caso::

$$\alpha_{icmp} = 1$$
$$\beta_{icmp} = 1$$
$$\gamma_{icmp} = 5$$

  Los valores elegidos permiten equilibrar que el camino elegido sea uno donde tanto la latencia como *jitter* como pérdidas tengan una importancia parecida.

- **TCP.** El protocolo TCP por si mismo no requiere calidad de servicio puesto que es muy general, pero existen multitud de protocolos que se transmiten vía TCP y que pueden requerir caminos muy específicos. Al no haber realizado una disección de protocolos más exhaustiva, se han establecido unos parámetros que den una mayor importancia a las pérdidas (TCP es muy sensible a pérdidas) y menor a *jitter*. Se propone:

$$\alpha_{tcp} = 0{,}5$$
$$\beta_{tcp} = 0{,}1$$
$$\gamma_{tcp} = 5$$

- **RTP Vídeo.** De la evaluación de calidad de transmisión de vídeo *streaming* se deduce que el factor de mayor importancia es la probabilidad de pérdidas. Además se establece que el *jitter* tendrá mayor importancia según sea cercano al valor de latencia (un *jitter* de 100 ms en una red con latencia 150 ms será crítico). La latencia se admite en cierta medida en este tipo de envío de paquetes. Se propone.

(a) Primera tabla de cálculo de costes y caminos con α β γ variantes.

alpha, beta, gamma	Camino mínimo	Coste
0,5---0,5----0,5	n4-n3	11
0,5---0,5----1	n4-n3	16
0,5---0,5----2	n2-n3	25,8
0,5---0,5----3	n2-n3	25,8
0,5---0,5----4	n2-n3	25,8
0,5---0,5----5	n2-n3	25,8
0,5---1----0,5	n4-n3	16
0,5---1----1	n4-n3	21
0,5---1----2	n2-n3	30,8
0,5---1----3	n2-n3	30,8
0,5---1----4	n2-n3	30,8
0,5---1----5	n2-n3	30,8
0,5---2----0,5	n4-n3	26
0,5---2----1	n4-n3	31
0,5---2----2	n2-n3	40,8
0,5---2----3	n2-n3	40,8
0,5---2----4	n2-n3	40,8
0,5---2----5	n2-n3	40,8
0,5---3----0,5	n4-n3	36
0,5---3----1	n4-n3	41
0,5---3----2	n2-n3	50,8
0,5---3----3	n2-n3	50,8
0,5---3----4	n2-n3	50,8
0,5---3----5	n2-n3	50,8

(b) Segunda tabla de cálculo de costes y caminos con α β γ variantes.

alpha, beta, gamma	Camino mínimo	Coste
1---0,5----0,5	n4-n3	12
1---0,5----1	n4-n3	17
1---0,5----2	n2-n3	26,6
1---0,5----3	n2-n3	26,6
1---0,5----4	n2-n3	26,6
1---0,5----5	n2-n3	26,6
1---1----0,5	n4-n3	16
1---1----1	n4-n3	21
1---1----2	n2-n3	30,8
1---1----3	n2-n3	30,8
1---1----4	n2-n3	30,8
1---1----5	n2-n3	30,8
1---2----0,5	n4-n3	26
1---2----1	n4-n3	31
1---2----2	n2-n3	40,8
1---2----3	n2-n3	40,8
1---2----4	n2-n3	40,8
1---2----5	n2-n3	40,8
1---3----0,5	n4-n3	36
1---3----1	n4-n3	41
1---3----2	n2-n3	50,8
1---3----3	n2-n3	50,8
1---3----4	n2-n3	50,8
1---3----5	n2-n3	50,8

(c) Tercera tabla de cálculo de costes y caminos con α β γ variantes.

alpha, beta, gamma	Camino mínimo	Coste
2---0,5----0,5	n4-n3	14
2---0,5----1	n4-n3	19
2---0,5----2	n4-n3	29
2---0,5----3	n4-n3	39
2---0,5----4	n4-n3	49
2---0,5----5	n4-n3	59
2---1----0,5	n4-n3	19
2---1----1	n4-n3	24
2---1----2	n4-n3	34
2---1----3	n4-n3	54
2---1----4	n4-n3	64
2---1----5	n4-n3	74
2---2----0,5	n4-n3	24
2---2----1	n4-n3	29
2---2----2	n4-n3	39
2---2----3	n4-n3	49
2---2----4	n4-n3	59
2---2----5	n4-n3	69
2---3----0,5	n4-n3	29
2---3----1	n4-n3	34
2---3----2	n4-n3	44
2---3----3	n4-n3	54
2---3----4	n4-n3	64
2---3----5	n4-n3	74

Tabla 7.8: Resultados del estudio teórico de cálculos de caminos.

Matriz costes ICMP	**S1**	**S2**	**S3**	**S4**
S1	0.0	134.59	134.47	78.57
S2	17.78	0	146.69	—
S3	92.08	77.88	0	42.61
S4	4.98	—	101.27	0.0
Camino elegido	Camino elegido: S1-S4-S3			Coste: 179.84

Tabla 7.9: Matriz de costes para ICMP con $\alpha_{icmp} = 1$ $\beta_{icmp} = 1$ y $\gamma_{icmp} = 5$

$$\alpha_{vídeo} = 0{,}1$$
$$\beta_{vídeo} = 0{,}5$$
$$\gamma_{vídeo} = 10$$

- **RTP VoIP.** De las restricciones para calidad de VoIP se extrae la necesidad de controlar los tres parámetros en gran medida. Para que unas pérdidas máximas del 2 % sean equiparables a los costes de latencia, se ha decidido que γ sea 10 veces el valor α y β. Se propone:

$$\alpha_{voip} = 1$$
$$\beta_{voip} = 1$$
$$\gamma_{voip} = 10$$

Con la evaluación de estos valores completada, podemos proceder a la evaluación del algoritmo *Dijkstra* con esta selección de pesos (que afectarán directamente en la evaluación de costes).

7.3. Evaluación del algoritmo de encaminamiento

En esta sección se va a comprobar si se realiza la elección del mejor camino en un escenario de red emulada. Comprobaremos que el algoritmo de encaminamiento elige el mejor camino. Los parámetros α_i β_i y γ_i tomarán los valores propuestos en la sección anterior.

Se comprobará que se elige el mejor camino en cuanto a coste (y que este camino sea el óptimo de acuerdo a las estimaciones tomadas en la sección 7.2).

Empezaremos evaluando los resultados con la topología 7.6. Nótese que El *jitter* no puede ser controlado en el emulador, con lo que los resultados obtenidos pueden no coincidir con los teóricos. Una vez evaluada esta, cambiaremos las tasas de pérdidas entre los enlaces S1-S4 y S4-S3 del 5 % al 2 % para evaluar el comportamiento del algoritmo de encaminamiento. Tras el período de entrenamiento se han obtenido los resultados que se muestran en las tablas 7.9, 7.10, 7.11 y 7.12.

En estas pruebas algunos enlaces obtienen costes muy bajos, esto es debido a que se han realizado con el envío de *streaming* de vídeo (para aumentar los bytes en los enlaces), solo en la dirección S1-S3.

Se puede comprobar como en los casos donde las pérdidas tienen una importancia muy superior a la latencia, se escoge el camino de menores pérdidas, mientras que cuando la latencia tiene una importancia elevada se escoge el camino más rápido. En ningún caso se escoge el camino de menor número de saltos puesto que no es el mejor para ninguno de los protocolos.

Matriz costes TCP	**S1**	**S2**	**S3**	**S4**
S1	0.0	36.51	110.32	59.42
S2	2.01	0	33.35	—
S3	84.80	25.15	0	18.53
S4	2.09	—	35.27	0.0
Camino elegido	Camino elegido: S1-S2-S3			Coste: 69.86

Tabla 7.10: Matriz de costes para TCP con $\alpha_{tcp} = 0{,}5$ $\beta_{tcp} = 0{,}1$ y $\gamma_{tcp} = 5$

Matriz costes RTP Vídeo	**S1**	**S2**	**S3**	**S4**
S1	0.0	13.46	202.83	103.98
S2	1.68	0	14.67	—
S3	165.33	7.79	0	21.24
S4	0.50	—	58.08	0.0
Camino elegido	Camino elegido: S1-S2-S3			Coste: 28.13

Tabla 7.11: Matriz de costes para vídeo sobre RTP con $\alpha_{vídeo} = 0{,}1$ $\beta_{vídeo} = 0{,}5$ y $\gamma_{vídeo} = 10$

Se modifican ahora las propiedades de los enlaces de acuerdo a la topología 7.7 para evaluar los cambios.

Los resultados obtenidos (solo el camino elegido), están recogidos en la tabla 7.13.

Para comprobar que se elige el mejor camino (siempre y cuando se hayan recogido las estadísticas y latencias de forma correcta), se ha repetido el experimento 5 veces. A continuación se muestran los resultados de 5 experimentos individuales para audio, tabla 7.14, y vídeo, tabla 7.15.

En estas tablas se comprueba que los costes varían bastante, cuando para un mismo camino en teoría deberían mantenerse. Esta desviación es provocada por varias situaciones:

1. En algunos casos se detectan latencias elevadas para algunos enlaces, con lo que el máximo se sitúa por encima de los valores reales, disminuyendo el valor asociado al coste de latencia de un enlace. A partir de la detección de este problema se decidió aumentar el número de valores para el cálculo de los valores de latencia media.

Matriz costes RTP audio	**S1**	**S2**	**S3**	**S4**
S1	0.0	134.59	238.35	129.16
S2	17.78	0	146.69	—
S3	174.25	77.88	0	51.55
S4	4.98	—	126.51	0.0
Camino elegido	Camino elegido: S1-S4-S3			Coste: 255.67

Tabla 7.12: Matriz de costes VoIP sobre RTP con $\alpha_{voip} = 1$ $\beta_{voip} = 1$ y $\gamma_{voip} = 10$

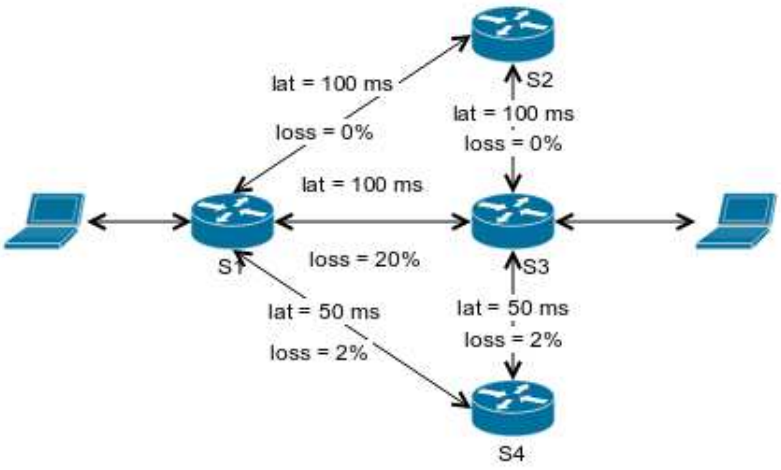


Figura 7.7: Segunda topología para realizar evaluación sobre el algoritmo de encaminamiento.

Camino elegido ICMP	S1-S4-S3	Coste: 110.54
Camino elegido TCP	S1-S2-S3	Coste: 15.90
Camino elegido Vídeo	S1-S2-S3	Coste: 21.28
Camino elegido Audio	S1-S4-S3	Coste: 123.08

Tabla 7.13: Resultados obtenidos en segunda topología para evaluación del algoritmo *Dijkstra*.

2. En ocasiones las pérdidas no se detectan de forma correcta (por los problemas que se han ido describiendo a lo largo del documento), con lo que el cálculo de costes está desviado respecto al valor real esperado.

3. El *jitter* no está controlado en la red, provocando en algunos casos desviaciones de costes.

A pesar de estos inconvenientes se puede comprobar que en la mayoría de los casos se escoge el camino más óptimo, o al menos uno cercano (descartando en todos los casos el camino más corto y menos óptimo).

En esta sección se ha demostrado de forma práctica que los pesos elegidos para cada función dan un peso equilibrado, priorizando la menor probabilidad de pérdidas siempre que sea posible (y necesario), sin dejar de evaluar parámetros como la latencia o el *jitter*.

Nº prueba	**Camino**	**Coste**
1	S1-S4-S3	52.4
2	S1-S4-S3	146.9
3	S1-S2-S3	87.8
4	S1-S4-S3	109.0
5	S1-S4-S3	117.08

Tabla 7.14: Resultados aplicación de algoritmo *Dijkstra* sobre flujos RTP VoIP en 7.7.

Nº prueba	Camino	Coste
1	S1-S2-S3	14.4
2	S1-S4-S3	81.4
3	S1-S2-S3	85.14
4	S1-S4-S3	88.85
5	S1-S2-S3	16.19

Tabla 7.15: Resultados aplicación de algoritmo *Dijkstra* sobre paquetes RTP vídeo en 7.7.

7.4. Evaluación sobre la solución completa

En esta sección se realizará la evaluación del sistema completo utilizando flujos de de audio y vídeo. Evaluaremos mediante parámetros objetivos y estimaciones de la calidad subjetiva los resultados obtenidos.

7.4.1. Descripción del entorno experimental

Se han diseñado dos topologías para la evaluación, mostradas en las figuras 7.8 y 7.9. La probabilidad de pérdida de paquetes en ambos escenarios será 0 %. Sin embargo, se configurarán dos caídas de enlaces durante las emisiones para evaluar cómo la solución implementada reacciona a las modificaciones de la red.

7.4.1.1. Topologías de red

A lo largo de la evaluación de la solución vamos a usar dos topologías, 7.8 y 7.9. La primera es una versión modificada de la topología 7.1, con lo que no será necesario volver a describirla. Se ha descartado el uso de probabilidad de pérdidas en los enlaces, con lo que los únicos parámetros definidos son:

$$latencia_{S1-S2} = 10ms$$
$$latencia_{S1-S3} = 80ms$$
$$latencia_{S1-S4} = 20ms$$
$$latencia_{S2-S3} = 40ms$$
$$latencia_{S4-S3} = 35ms$$

Como segunda topología se ha elegido una de mayor complejidad pues combina un gran número de equipos con un gran número de posibles caminos. En esta topología intervienen 13 *switches* conectados por enlaces de latencia variable. La probabilidad de pérdidas en los enlaces se ha mantenido a 0 para poder evaluar las pérdidas introducidas por una caída de enlace. Todos los *switches* de la topología serán controlados por el controlador C_1[1]. Por último, en la topología contamos con dos equipos finales, encargados de enviar y recibir el tráfico.

Para comparar los resultados obtenidos en la topología 7.9 con los ideales, se deben calcular los mejores caminos (considerando que el flujo de paquetes será siempre de izquierda a derecha y fijándonos solo en la latencia) para esta topología. Todos los posibles caminos están recogidos en la tabla 7.16, destacando el mejor y peor de todos ellos.

[1]No aparece en la topología, pero todos los *switches* estarán conectados al controlador.

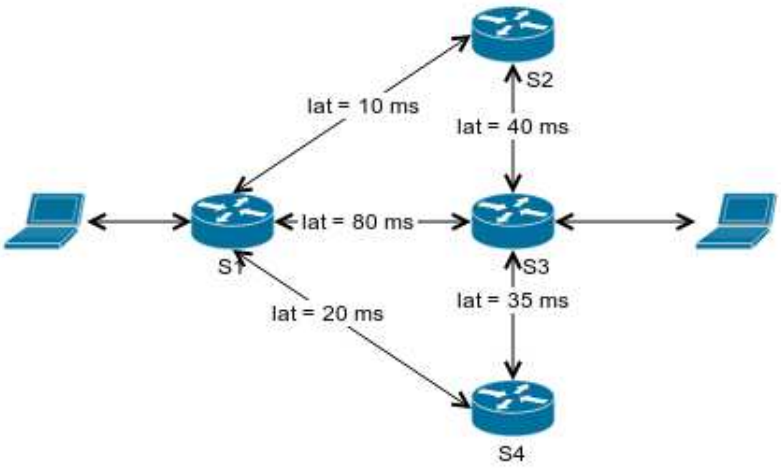


Figura 7.8: Primera topología para la evaluación del sistema.

Camino	Latencia
S1-S2-S6-S11-S13	**75 ms**
S1-S2-S4-S12-S13	80 ms
S1-S3-S4-S12-S13	80 ms
S1-S3-S4-S11-S13	85 ms
S1-S2-S4-S11-S13	85 ms
S1-S2-S8-S11-S13	90 ms
S1-S3-S5-S11-S13	105 ms
S1-S3-S7-S12-S13	110 ms
S1-S3-S10-S11-S13	145 ms
S1-S3-S9-S12-S13	150 ms
S1-S2-S10-S11-S13	150 ms
S1-S3-S10-S12-S13	165 ms
S1-S2-S10-S12-S13	**170 ms**

Tabla 7.16: Posibles caminos con el mínimo número de saltos en la topología presentada en la figura 7.9, y las latencias asociadas.

7.4.1.2. Generación del tráfico

Para la generación de audio se ha valorado el uso de diferentes herramientas, decantándonos finalmente por la herramienta *mausezahn* [66], por su facilidad de uso y compatibilidad con Ubuntu. *Musezahn* es una herramienta libre diseñada para la generación rápida de paquetes. Está diseñada para soportar el envío de un gran tipo de paquetes, incluso tipos que no han sido definidos, ya que cuenta con un modo de envío donde cada byte del paquete puede ser especificado. Como generador de paquetes de audio RTP, *mausezahn* directamente se encarga de usar paquetes VoIP, como posteriormente comprobaremos con *Wireshark*.

Para la generación del tráfico de vídeo se utilizará la herramienta Video Local Area Network Client (VLC), un reproductor multimedia de software libre que permite emitir flujos de vídeo en distintos formatos y sobre distintos protocolos. En nuestro caso usaremos RTP como protocolo de trasporte.

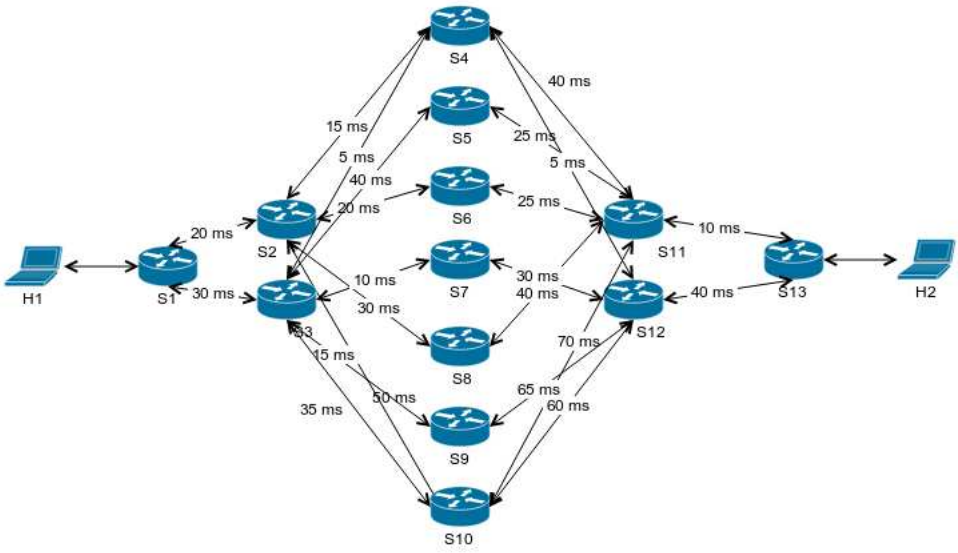


Figura 7.9: Segunda topología para la evaluación del sistema.

7.4.1.3. Medición objetiva de la QoE de flujos de audio y vídeo

La evaluación de QoE que percibe un usuario es una tarea compleja, que suele llevarse a cabo mediante encuestas de opinión sobre un gran número de sujetos a los que se les presenta el resultado, en nuestro caso, de la transmisión de los flujos multimedia que han de evaluarse. La opinión de los usuarios suele medirse en una escala predefinida común para todos los participantes del experimento. En nuestro caso hemos optado por hacer uso de la escala MOS, recogida en la tabla 7.17, y definida por la ITU-T en la recomendación P.800 [67]. Nótese que para cada calidad se asocia un número del 1 al 5, que es lo que debería indicar el sujeto experimental.

MOS	**Calidad**	**Percepción de problemas**
5	Excelente	Imperceptibles
4	Buena	Algunos pero sin provocar descontento
3	Regular	Descontento aceptable
2	Mediocre	Descontento
1	Mala	Imposible usar

Tabla 7.17: Tabla de valores MOS y su equivalencia.

A partir de esta definición podremos estimar la calidad de la solución propuesta.

En nuestro trabajo, para la evaluación subjetiva de las rutas calculadas para cada tipo de tráfico, al no disponer de suficientes sujetos experimentales para realizar una encuesta de satisfacción, se ha optado por utilizar dos medidas de calidad estandarizadas. Para audio (VoIP) se utilizará el Emodel, y para evaluar la calidad de vídeo, la recomendación G.1070.

Recomendación G.107 para la evaluación de la calidad de audio Para la evaluación de la calidad percibida por el usuario en transmisiones de audio, usaremos la recomendación de la ITU [10], donde se describe un modelo para la estimación de calidad en transmisión deaudio. Además nos apoyaremos en el artículo [68], donde se propone un modelo simplificado para el códec de audio usado, en este caso G.711.

Para el cálculo de la calidad necesitaremos hacer uso de las ecuaciones :

$$MOS = 1 + 0{,}0{,}35 \cdot R + R \cdot (R - 60) \cdot (100 - R) \cdot 7 \cdot 10^{-6} \tag{7.1a}$$

$$R \approx 94{,}2 - 0{,}024 \cdot d + 0{,}11 \cdot (d - 1773) \cdot H \cdot (d - 177{,}3) - I_e \tag{7.1b}$$

$$I_e(G{,}711, random) \approx 30 \cdot ln(1 + 15 \cdot e) \tag{7.1c}$$

$$H(x) = \begin{cases} 0 & \text{si} \quad x < 0 \\ 1 & \text{si} \quad x \geq 0 \end{cases} \tag{7.1d}$$

Donde d será la latencia media de los paquetes y e la probabilidad de pérdidas de paquete calculada.

Recomendación G.1070 para la evaluación de la calidad de vídeo Evaluar la calidad de transmisión será un reto ya que deberemos asumir ciertas suposiciones. Usaremos la recomendación de la ITU [11], donde se describe un modelo para la estimación de calidad en transmisión de vídeo. Además nos apoyaremos en el artículo [69] donde se recogen correcciones para formatos de vídeo de mayor calidad. Se debe suponer:

- El formato de envío de vídeo será MPEG-2.
- La calidad máxima de envío será estándar (SD), aunque en realidad algunos de los vídeos cuentan con una mayor calidad.
- Se asume que la probabilidad de pérdidas dependerá solo de las caídas provocadas en los enlaces.
- Suponemos que todos los vídeos tienen una alta tasa de movimiento (afectará a la elección de coeficientes).

Las ecuaciones con las que calcularemos la calidad de vídeo son: 7.2a, 7.2b y 7.2c.

$$V_q = 1 + I_c e^{\frac{P_{plv}}{D_{Pplv}}} \tag{7.2a}$$

$$I_c = v_3 \left(1 - \frac{1}{1 + \left(\frac{Br_V}{v_4} \right)^{v_5}} \right) \tag{7.2b}$$

$$D_{Pplv} = v_{10} + v_{11} e^{-\frac{Fr_V}{v_8}} + v_{12} e^{-\frac{Br_V}{v_9}} \tag{7.2c}$$

Doonde P_{Plv} es la probabilidad de pérdidas de paquete (en %), B_{rV} el *Bit Rate* necesario para el vídeo y F_{rV} los *frames* por segundo del vídeo.

Para poder hacer uso de estas ecuaciones, se deben elegir los coeficientes para éstas. En este caso se eligen:

Prueba	Datos enviados	Datos perdidos	Latencia media	Jitter Promedio	MOS Scale EModel
1	720 000 B 4500 Paquetes	11 520 B 72 Paquetes	56 ms	42 us	3.82
2	720 000 B 4500 Paquetes	15 840 B 99 Paquetes	61.2 ms	25 us	3.72
3	720 000 B 4500 Paquetes	11 840 B 74 Paquetes	55 ms	29 us	3.80
4	720 000 B 4500 Paquetes	8 480 B 53 Paquetes	63.3 ms	32 us	3.86
5	720 000 B 4500 Paquetes	11 680 B 73 Paquetes	60 ms	29 us	3.80

Tabla 7.18: Resultados obtenidos tras el envío de paquetes durante 90 segundos (2 caídas de enlace intermedias) sobre la topología presentada en la figura 7.8.

$$v_3 = 4$$
$$v_4 = 1{,}088$$
$$v_5 = 1{,}56$$
$$v_8 = 0{,}7846$$
$$v_9 = 85{,}15$$
$$v_{10} = 1{,}32$$
$$v_{11} = 539{,}48$$
$$v_{12} = 356{,}6$$

Estos coeficientes serán aplicados en las ecuaciones 7.2a, 7.2b y 7.2c.

7.4.2. Evaluación objetiva de la calidad de transmisión

Para la evaluación objetiva se va a generar tráfico de tipo RTP voz y RTP vídeo. Realizaremos una evaluación independiente para cada tipo de tráfico.

7.4.2.1. Evaluación objetiva de la calidad RTP VoIP

Para la topología 7.8, y una transmisión de paquetes de 64 kbits/s, el equivalente a una sesión de VoIP con el códec G.711 [39]. Se han realizado diversas pruebas, emulando los eventos que se detallan a continuación.

A los 30 segundos de transmisión se emula la caída del enlace S1-S2, a los 60 segundos se ha recuperado el enlace S1-S2 y se ha emulado la caída del enlace S1-S4 (o el que se haya elegido para el envío). A los 90 segundos se ha detenido el envío de paquetes. En la tabla 7.18 se presentan los resultados obtenidos usando el modelo presentado en [68] para la evaluación del modelo *EModel* y parámetros para el *códec* G.711. y la topología 7.8.

En las pérdidas se han incluido aquellos paquetes que han llegado fuera de orden (fuera del espacio temporal asumible). Además se incluyen os paquetes que se pierden al inicio, ya que la instalación del primer flujo de envío (cuando llegan nuevos paquetes), no es inmediata.

Se realizan las mismas pruebas sobre la topología 7.9.

Por último el MOS medio para cada experimento se incluye en la tabla 7.20.

Estos resultados contrastan con el valor máximo que se podría obtener en la topología 7.8 en un caso ideal, que está calculado en la tabla 7.21. De igual modo se ha calculado para la topología 7.9 en la tabla 7.22.

Prueba	Datos enviados	Datos perdidos	Latencia media	Jitter Promedio	MOS Scale EModel
1	720 000 B 4500 Paquetes	17 280 B 108 Paquetes	85 ms	120 us	3.68
2	720 000 B 4500 Paquetes	23 840 B 149 Paquetes	82 ms	114 us	3.58
3	720 000 B 4500 Paquetes	12 480 B 78 Paquetes	92 ms	102 us	3.75
4	720 000 B 4500 Paquetes	22 720 B 142 Paquetes	98 ms	97 us	3.58
5	720 000 B 4500 Paquetes	17 600 B 110 Paquetes	85 ms	110 us	3.67

Tabla 7.19: Resultados obtenidos tras el envío de paquetes durante 90 segundos (2 caídas de enlace intermedias) sobre la topología presentada en la figura 7.9.

	MOS Medio
Escenario 1	3.8
Escenario 2	3.65

Tabla 7.20: Valores MOS para los experimentos sobre audio

La latencia ideal corresponde a 60 segundos por el camino de 50 ms y 30 segundos por el de 55 ms.

La latencia ideal corresponde a 60 segundos por el camino de 75 ms y 30 segundos por el de 80 ms.

Podemos comprobar que los resultados ideales para estas dos topologías no se alejan de los resultados obtenidos en la evaluación. La disminución de calidad percibida se debe a: desviación de la latencia obtenida frente a la ideal (tiempos de procesamiento), elección de caminos no óptimos (en algunos casos no se ha escogido el mejor camino disponible) y pérdida de paquetes en la actualización de topologías e inicio.

Otra comprobación posible es la que se obtiene a partir de los resultados 7.19. Si prestamos atención a las latencias obtenidas, comprobaremos que son cercanas a los mejores caminos calculados en la tabla 7.16, lo cual es un indicador del buen funcionamiento del cálculo de costes y funcionamiento del algoritmo *Dijkstra*.

Por último se incluye un cálculo para el tiempo sin servicio en ambos casos. El tamaño de cada paquete (para calcular la tasa de transmisión), es de 160B, de la especificación G. 711 [39] también se obtiene la tasa de transmisión en bits, que es igual a 64.000 kbps. Con estos datos se han calculado los resultados que aparecen en las tablas 7.23 y 7.24.

Podemos comprobar que un mayor número de saltos en los caminos de transporte (mayor número de nodos) influirá negativamente en los resultados obtenidos por la solución propuesta. Se deja para el capítulo 8 la valoración sobre estos resultados.

Latencia	51.67 ms
Pérdidas	0 %
Jitter	0 ms
MOS	4.10

Tabla 7.21: Resultados ideales (óptimos) MOS para la topología de la figura 7.8.

Latencia	76.67 ms
Pérdidas	0 %
Jitter	0 ms
MOS	4.07

Tabla 7.22: Resultados ideales (óptimos) MOS en la topología 7.8

Topología 1	**Paquetes perdidos**	**Rate (paquetes/segundo)**	Caídas de enlace	**Tiempo sin servicio (segundos)**
Prueba 1	72	50	2	0.72
Prueba 2	99	50	2	0.99
Prueba 3	74	50	2	0.74
Prueba 4	53	50	2	0.53
Prueba 5	73	50	2	0.73
Media	75	50	2	0.74

Tabla 7.23: Tiempo sin servicio para la topología 7.8 para una caída de enlace

7.4.2.2. Evaluación objetiva de la calidad de transmisión de RTP vídeo

Para esta evaluación se ha optado por emitir vídeo vía *streaming* de alta calidad (High Definition (HD)). Recogeremos estadísticas y a partir de estas se realizará una evaluación de calidad de transmisión. Se emitirán tres formatos de vídeo, con diferentes calidades, y se comprobarán los resultados. Los vídeos a emitir tienen las características de la tabla 7.25.

Para la emisión de estos vídeos haremos uso de la herramienta libre y gratuita VLC.

A partir de las expresiones para la medida de la calidad del vídeo (recomendación ITU-T G.1070) descritas en la sección 7.4.1.3 y las estadísticas de red detectadas, podremos estimar la calidad de vídeo percibida.

Para la recolección de información se han establecido 5 sesiones de *streaming* de vídeo, provocando varias caídas en instantes aleatorios del enlace de transmisión*qué enlace exactamente?*. Los resultados recolectados están resumidos en las tablas 7.26, 7.27 y 7.28.

Para los tipos de vídeo escogidos, en un caso ideal (0 % de probabilidad de pérdidas), se obtendría una calidad de 5 en la escala MOS. Con lo que los resultados que obtengamos podrían llegar a tener un máximo de calidad igual a 5. Se de ha de tener en cuenta que todos los vídeos necesitan un período inicial para la instalación del primer flujo, lo que conllevará algunas pérdidas extra (en torno a 5 paquetes en las medidas realizadas). En vídeos de larga duración estas pérdidas no son significativas, sin embargo cuando los vídeos son muy cortos (como en el primer ejemplo), tendrán un peso importante en los resultados obtenidos del modelo.

Topología 2	**Paquetes perdidos**	**Rate (paquetes/segundo)**	Caídas de enlace	**Tiempo sin servicio (segundos)**
Prueba 1	108	50	2	1.08
Prueba 2	149	50	2	1.49
Prueba 3	78	50	2	0.78
Prueba 4	142	50	2	1.42
Prueba 5	110	50	2	1.1
Media	118	50	2	1.17

Tabla 7.24: Tiempo sin servicio para la topología 7.9 para una caída de enlace

	Vídeo 1	**Vídeo 2**	**Vídeo 3**
Duración	52 s	5 min 58 s	2 min 28 s
Resolución	400x226	1280 x 720	1920 x 1080
Códec	H.264	H.264	H.264
fps	30	24	25
Tasa de bits	248 + 96 kbps	1529 + 191 kbps	2013 + 125 kbps

Tabla 7.25: Propiedades vídeos usados evaluación calidad RTP vídeo.

Video 1	**Paquetes enviados**	**Paquetes recibidos**	**Pérdidas (%)**	**Rate (bps)**	**Frames por segundo**	**Calidad MOS**
1	2201	2175	1.18	344 000	30	2.63
2	2199	2183	0.73	344 000	30	3.30
3	2198	2174	1.09	344 000	30	2.75
4	2203	2194	0.41	344 000	30	3.94
5	2200	2197	0.14	344 000	30	4.61

Tabla 7.26: Resultados calidad vídeo 1 RTP. 1 caída de enlace.

A partir de estos resultados se han obtenido los valores medios para MOS, en la tabla 7.29.

Para obtener la calidad percibida por el usuario final se recurre de nuevo a la tabla 7.17, observando como la calidad obtenida, sin ser la mejor, tampoco llega nunca a situarse por debajo de valores que generarían descontento en el usuario.

Adelantando conclusiones, podemos decir que a una mayor calidad de vídeo (mayor *rate* de transmisión), se obtendrán mayores pérdidas y peor calidad de experiencia según el modelo, debido a la mayor tasa de transmisión.

Se calcula ahora la pérdida media de paquetes para vídeo, y el tiempo sin servicio. Se incluyen en estas pérdidas medias por enlace las pérdidas iniciales ya comentadas anteriormente, obteniendo los resultados de la tabla 7.30.

Para hacer estos cálculos se ha comprobado con *Wireshark* que la carga útil de un paquete RTP vídeo es de 1328B como indica la captura 7.10. Se están capturando paquetes Ethernet con una longitud total de 1370B. RTP se encuentra encapsulado en UDP y a su vez en IP, provocando que la carga útil sea la que que se puede comprobar en 7.10.

Finalmente se comprueba que en todos los casos el tiempo sin servicio es inferior a un segundo. Un usuario puede tolerar una pérdida de servicio pequeña siempre y cuando el resto del servicio sea bueno (buena calidad de vídeo y pocas pérdidas continuas). En el capítulo 8 se presentan las conclusiones a partir de estos datos.

Video 2	**Paquetes enviados**	**Paquetes recibidos**	**Pérdidas (%)**	**Rate (bps)**	**Frames por segundo**	**Calidad MOS**
1	63 051	62 650	0.64	1 720 000	24	3.47
2	63 042	62 593	0.71	1 720 000	24	3.33
3	63 050	62 574	0.67	1 720 000	24	3.40
4	63 038	62 464	0.90	1 720 000	24	3.02
5	63 037	62 565	0.75	1 720 000	24	3.26

Tabla 7.27: Resultados calidad vídeo 2 RTP. 4 caídas de enlace.

Video 3	Paquetes enviados	Paquetes recibidos	Pérdidas (%)	Rate (bps)	Frames por segundo	Calidad MOS
1	32 121	32 036	0.30	2 128 000	25	4.20
2	32 121	31 768	1.09	2 128 000	25	2.74
3	32 121	31 874	0.79	2 128 000	25	3.20
4	32 121	21 878	0.78	2 128 000	25	3.22
5	32 120	32 018	0.63	2 128 000	25	3.48

Tabla 7.28: Resultados calidad vídeo 3 RTP. 2 caídas de enlace.

Vídeo	MOS medio
1	3.45
2	3.30
3	3.37

Tabla 7.29: Valores MOS medios para cada vídeo.

7.4.3. Evaluación subjetiva de la calidad de transmisión

Las imágenes que aparecen a continuación están extraídas de vídeos bajo licencia *Creative Commons* y tanto su distribución como uso está permitido y amparado por esta licencia. La película ha sido descargada de la página `https://peach.blender.org/`.

La evaluación subjetiva se realizará con la emisión de *streaming* de vídeo HD a través de la topología 7.8. Durante la emisión del *streaming* se forzarán dos caídas aleatorias del enlace por el cual se transmite y se evaluarán los posibles inconvenientes percibidos por el usuario.

Las capturas 7.11 y 7.12 muestran lo que el usuario visualiza en el momento de la caída.

En estas capturas se aprecian pérdidas, pero no una parada de servicio como tal (pantalla en negro o pérdida de información). Adicionalmente se ha comprobado que ocurre en casos como caída durante un cambio de escena o escena con mucho movimiento. El resultado para cambio de escena se traduce en una mayor pérdida de información para el usuario, como se comprueba en la captura 7.13.

En una escena de alto movimiento, se detectan mayores pérdidas, pero sin llegar a suponer una pérdida de información importante para el el espectador, como ocurre en la captura 7.14.

Se comprueba que una caída durante un cambio de escena es mucho más crítica que en cualquier otra situación. Para paliar este efecto se puede optar por aumentar el *búffer* de reproducción y aplicar técnicas de recuperación, o usar técnicas de reordenación de paquetes para prevenir errores de ráfaga. Estos desarrollos escapan al alcance del presente trabajo por lo que se proponen como líneas de trabajo futuras.

A parte de los inconvenientes visuales, también se ha detectado el problema que se describe a continuación.

Vídeo	Paquetes perdidos	Rate (Paquetes/segundo)	Caídas de enlace	Paquetes perdidos por caída	Tiempo sin servicio (s)
1	16	32	1	16	0.48
2	464	160	4	116	0.72
3	233	197	2	117	0.58

Tabla 7.30: Tiempo sin servicio a partir de la estimación de paquetes perdidos.

```
▸ Frame 38: 1370 bytes on wire (10960 bits), 1370 bytes captured (10960 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▸ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.2 (10.0.0.2)
▸ User Datagram Protocol, Src Port: 56377 (56377), Dst Port: 5004 (5004)
▾ Data (1328 bytes)
```

Figura 7.10: Carga útil paquete RTP vídeo.

Figura 7.11: Captura momento de caída 1 en transmisión streaming vídeo RTP. Captura extraída de la película "Big Buck Bunny.

- Desincronización de audio. En las primeras pruebas realizadas (antes de optimizar el código) se detectaron casos en los que se perdió el audio, provocando una pérdida de información muy importante. Tras la optimización del código la frecuencia de estos casos es muy baja, sin embargo puede suceder. Para resolver este inconveniente se podría optar por otras herramientas para recepción y emisión. Esta alternativa queda como línea de trabajo futura.

Figura 7.12: Captura momento de caída 2 en transmisión streaming vídeo RTP. Captura extraída de la película “Big Buck Bunny.

Figura 7.13: Captura momento de caída de enlace durante cambio de escena. Captura extraída de la película “Big Buck Bunny.

Figura 7.14: Captura momento de caída de enlace en escena con alto movimiento. Captura extraída de la película “Big Buck Bunny.

Capítulo 8

Conclusiones y líneas de trabajo futuras

A lo largo de este capítulo se pretenden exponer alcanzadas tras la realización del proyecto. Estarán divididas en tres grupos: conclusiones técnicas sobre SDN, conclusiones técnicas sobre las herramientas utilizadas (controlador y *Mininet*) y conclusiones sobre la solución propuesta (a partir de los resultados obtenidos).

Además se propondrán una serie de líneas de trabajo futuras que se pueden acometer a partir de la información recolectada tras la realización del presente trabajo.

8.1. Conclusiones

En este trabajo se ha diseñado un sistema de provisión de calidad de experiencia para usuarios en redes SDN, para evaluar las ventajas que ofrecen estas nuevas redes. Para ello se ha llevado a cabo una revisión del estado del arte sobre SDN y *Openflow*. También se han descrito los principales controladores disponibles para redes SDN.

Se ha diseñado e implementado en el controlador *OpenDayLight* (ODL) una aplicación capaz de elegir el camino óptimo dependiendo de cada tipo de flujo, y de recuperar la red ante posibles caídas de enlaces en ésta. El diseño de la solución contempla elementos para: recogida de estadísticas de red (como latencias o bytes transmitidos en cada enlace), evaluación de costes por enlace, encaminamiento inteligente de flujos mediante clasificación de paquetes, detección de cambios y recuperación de errores.

Finalmente se han evaluado esta solución mediante experimentación en diversos entornos, para garantizar su correcto funcionamiento. Los resultados obtenidos muestran que es posible la administración automática de la red gracias al uso del controlador SDN. Los resultados obtenidos muestran una rápida recuperación de errores, dando un tiempo sin servicio menor a un segundo en cada caída, mejorando los resultados que se obtendrían usando algoritmos típicos de redes tradicionales, como OSPF. No se tienen en cuenta sistemas de alta disponibilidad, ya que la aplicación desarrollada no puede compararse en ese marco.

Para organizar el resto de las conclusiones, estas se han dividido en tres grupos, según la temática sobre la cual se enfoca cada conclusión.

8.1.1. Conclusiones sobre SDN y protocolo *OpenFlow*

Las conclusiones del primer grupo están enfocadas en redes SDN y el protocolo *OpenFlow*.

8.1.1.1. Conclusiones sobre SDN

- Las redes SDN suponen una revolución tecnológica necesaria y por la cual están apostando los grandes fabricantes hardware y software de la industria.
- El interés generado por esta nueva tecnología está consiguiendo que el impulso y desarrollo de ésta esté avanzando a una gran velocidad, existiendo ya soluciones finales que usan esta tecnología.

8.1.1.2. Conclusiones sobre *OpenFlow*

- Existen diferentes alternativas diseñadas para redes SDN, pero es *OpenFlow* el protocolo dominante en este ámbito.
- *OpenFlow* se encuentra en desarrollo. Además cuenta con el apoyo de grandes compañías de la industria que posibilitan el rápido avance de este protocolo.
- Existen casos de éxito de la aplicación del protocolo *OpenFlow* en situaciones reales, por ejemplo, [2].

8.1.2. Conclusiones sobre *Mininet* y *OpenDayLight*

El segundo grupo de conclusiones está centrado en las implementaciones utilizadas para controlador y emulador de redes.

8.1.2.1. Conclusiones sobre *Mininet*

- *Mininet* es una herramienta muy potente capaz de emular redes con un gran número de nodos (siempre que el equipo donde se ejecute sea capaz de ello).
- Existen ventajas en resultados obtenidos con emulación sobre emulación, lo cual convierte mininet en una potente e interesante herramienta para temas didácticos o de experimentación.
- El diseño de topologías en *Mininet* es muy cómodo gracias a la *Python* API. Además existen editores gráficos que permiten dibujar una topología y obtener el código para *Mininet*.

8.1.2.2. Conclusiones sobre controlador *OpenDayLight*

- El carácter abierto de *OpenDayLight*, así como el apoyo de pesos pesados en la industria ha supuesto un gran impulso en este, que junto al apoyo de la comunidad lo convierte en una de las soluciones más atractivas.
- Las diferentes posibilidades de desarrollo de aplicaciones para el controlador ODL presentan una buena oportunidad para desarrolladores de cualquier tipo. Sin embargo la rápida evolución del controlador provoca cierto desconcierto y falta de información para estos desarrolladores que están dando sus primeros pasos.

8.1.3. Conclusiones sobre los resultados

La evaluación sobre la solución desarrollada se dividió en dos: resultados objetivos y resultados subjetivos. A partir de los resultados hemos destacado las siguientes conclusiones:

8.1.3.1. Resultados objetivos

Se transmitieron paquetes de audio RTP y se provocaron errores en la red (caídas de enlace). Fueron evaluadas dos topologías, una simple y otra compleja. Todos los datos obtenidos se han conseguido con transmisión simple a través de la red, sin algoritmos para recuperación de pérdidas, o intercalado de paquetes para evitar pérdidas de ráfagas.

En la segunda evaluación objetiva se envío *streaming* de vídeo sobre la red 7.8, obteniendo los resultados recogidos en la sección 7.4.2.2.

Evaluación sobre resultados RTP audio

- Gracias al modelo simplificado de EMODEL presentando en [68] se han obtenido en ambas topologías valores de escala MOS. Estos valores se concentran en torno al valor 3.8 (nivel de calidad *bueno*) en una topología simple y 3.7 (nivel de calidad *bueno*) en una topología compleja.

- Si extrapolamos los datos a tiempo sin servicio estos datos, dependiendo de la complejidad de la topología, se comprueba que cada caída de enlace provoca un tiempo sin servicio de 0.75 segundos en una topología simple. Si la complejidad de la topología aumenta, el tiempo sin servicio aumenta hasta los 1.20 segundos.

- En una llamada telefónica estar un segundo sin servicio puede suponer un inconveniente para el usuario, pero en ningún caso equivalente a lo que supone una caída de enlace en redes actuales (sin tener en cuenta sistemas de alta disponibilidad que abusan de recursos de red para conseguirlo), donde una caída de enlace supone en muchos casos la finalización del servicio temporalmente.

- Se ha conseguido encaminar (en la mayoría de los casos) de forma rápida por aquellos caminos considerados como óptimos en cada caso.

- Las fuertes restricciones temporales para VoIP provocan que las pérdidas registradas sean mayores, ya que en audio realmente no se pierden tantos paquetes (al usar un *rate* de transmisión bajo), sin embargo las caídas de enlace provocan que muchos de estos paquetes lleguen fuera del orden temporal, sumándose en este caso a las pérdidas detectadas.

- La automatización es absoluta en la red, pues se consiga encaminar en tiempo real y reaccionar ante caídas de enlaces, de manera automática.

Evaluación sobre resultados RTP vídeo

- Estos modelos no contemplan calidades de vídeo HD o *full* HD, con lo que los resultados alcanzados no reflejan con exactitud la calidad real percibida.

- Los resultados obtenidos se concentran en torno al valor 3.4, que es una calidad entre buena y regular, según la escala MOS.

- La calidad calculada se aleja de los máximos esperados debido al número de caidas introducidas durante la experimentación, instalación inicial de flujos, y al alto *rate* de transmisión requerido por las aplicaciones de *streaming* de vídeo.

- Atendiendo a los resultados calculados para el tiempo sin servicio, dependiendo de la calidad del víde,o se han obtenido tiempos entre 0.5 segundos y 0.75 segundos.

- A pesar de requerir una velocidad de transmisión, el *streaming* de vídeo sufre menor tiempo sin servicio que el registrado para audio. Esto es debido a las fuertes restricciones respecto a la latencia para los paquetes de audio.

8.1.3.2. Resultados subjetivos para vídeo

Además de los resultados objetivos se ha incluido un pequeño apartado mostrando los diferentes eventos que pueden suceder durante una caída de enlace.

- El tiempo sin servicio en *streaming* de vídeo es aproximadamente igual al presentado en los resultados objetivos, sin embargo en casos donde la caída se produce en un cambio de escena, el usuario detecta la caída y la valoración es negativa. Cuando la caída se produce durante el transcurso de una escena el impacto es mucho menor ya que pese a encontrar pérdidas, la reproducción no para en ningún momento.

- En algunos casos se ha detectado pérdida de audio durante una caída debida a de sincronización, provocando una valoración muy negativa por parte del usuario. Este aspecto debe ser corregido en futuras implementaciones (cambio de método de transmisión o mejora del sistema).

- Las escenas con mucho movimiento pueden suponer un problema para la valoración del destinatario, pero en ningún suponen pérdidas de información, con lo que la valoración del usuario seguiría siendo positiva.

8.2. Líneas de trabajo futuras

A partir del desarrollo realizado se han detectado y establecido las siguientes líneas de trabajo. Se han organizado de acuerdo a su temática.

8.2.1. Líneas teóricas

- Estudio de aplicaciones reales de redes SDN. Se ha estudiado en este trabajo las ventajas de este tipo de redes, así como algunas aplicaciones reales. Sin embargo es de gran interés estudiar la evolución de estas redes, así como su aplicación a redes 5G, entre otras.

- Estudio sobre protocolos alternativos. En este trabajo se ha descrito por encima el protocolo OpFlex. Será de gran interés seguir de cerca la evolución del protocolo *OpenFlow* así como los nuevos protocolos que puedan surgir y cuyas características puedan suponer una mejora frente a este.

- Estudio sobre la evolución del controlador *OpenDayLight* y otros controladores. A día 29 de Junio de 2015, el controlador *OpenDayLight* ha presentado una nueva versión, lo cual no es sino otra muestra del grado de desarrollo que está siguiendo este y otros controladores.

8.2.2. Líneas prácticas

- Mejoras sobre la implementación. Se han detectado problemas en la implementación, como la estimación de pérdidas. Será de gran interés la mejora de la aplicación que permita disminuir el tiempo de entrenamiento provocado por estas pérdidas.

- Adaptación de la aplicación a las nuevas versiones del controlador *OpenDayLight*. Estas nuevas versiones probablemente incluyan mejoras de rendimiento del controlador, así como un mejor funcionamiento. Estas nuevas versiones probablemente incluyan soluciones para los problemas encontrados en la obtención de latencias y ancho de banda del sistema.

- Implementación de métodos para no depender de nuevos paquetes que lleguen al controlador. Existen implementaciones de clases que son llamadas cuando ocurren eventos en la red y que pueden detectar caídas o situaciones de sobrecarga en enlaces, siendo estas clases de mayor utilidad que la implementada en este trabajo.

- Implementación de un sistema de protección frente a ráfagas, para evitar pérdidas de información importantes en situaciones extremas, como los cambios de escena.

- Mejoras de eficiencia que permitan disminuir los tiempos de recuperación de enlace, lo que afectará directamente en los valores MOS obtenidos.

- Por último se propone la integración de esta aplicación (u otras versiones) en una aplicación real para comprobar el funcionamiento de está en este escenario, así como para comprobar si el uso de redes SDN supone una ventaja real y una mejora de funcionamiento.

Apéndice A

Configuración del entorno de trabajo

En estos apéndices se recogen documentos para configuración y trozos de código para una mejor comprensión de lo realizado en este proyecto.

A.1. Configuración Ubuntu

En el trabajo se ha optado por una instalación fresca de Ubuntu 14.04. Para el uso de Ubuntu y todas las herramientas del sistema solo existe un requisito, que la versión del *kernel* soporte Open vSwitch. Las últimas versiones del *kernel* de Línux integran Open vSwitch.

La instalación básica de Ubuntu puede ser encontrada en cualquier lugar con lo cual no se incluye aquí una guía para ello.

Para la realización del trabajo se han instalado las siguientes herramientas extra:

- ***Wireshark.***
- **Text-Live y Texmaker.** Editor y compilador para el lenguaje Látex, para la realización de la memoria técnica del proyecto.
- **Java.** Ubuntu por defecto incluye la versión openJDK. Es posible que éste use la versión 8, para la perfecta compatibilidad con *OpenDayLight* será necesario desinstalar la versión 8 e instalar la versión 7.
- **Maven.** Necesario instalar el compilador maven.

  ```
  1 sudo apt-get install maven
  ```

  No instalar la versión 2. No será necesario configurar maven salvo que sea necesaria alguna librería externa. Podrá encontrarse fácilmente una guía con una simple búsqueda en Google del tipo: "repositorio externo Maven".
- Instalación de *github*. Muy útil para la descarga de código del proyecto o de cualquier lugar (por ejemplo *Mininet*).

  ```
  1 sudo apt-get install git
  ```
- Se aconseja usar un entorno de trabajo con varios escritorios disponibles, es una cuestión de gustos personales, pero facilitará la transición entre ventanas de terminal y código.

A.2. Configuración *Mininet*

Instalar *Mininet* en Ubuntu es una tarea que puede resultar muy sencilla (siempre y cuando se esté familiarizado con el SO). Existen dos caminos, hacer uso de los paquetes para Ubuntu o compilar el código fuente.

En este caso se ha optado por usar la compilación de código fuente, ya que la última versión de *Mininet* no estaba disponible en los repositorios oficiales en el momento de su instalación.

La instalación sigue los siguientes pasos:

```
1 git clone git://github.com/mininet/mininet
2
3 cd mininet
4
5 ./util/install.sh -a
```

Aceptar en cualquier caso. Es posible que la instalación de algún problema de permisos. En ese caso ejecutar el último comando como super usuario.

Para comprobar el correcto funcionamiento se podrá ejecutar:

```
1 sudo mn --test pingall
```

En caso de que no funcione puede existir algún problema con la instalación de *Mininet*. Se recomienda reiniciar y en caso de no funcionar eliminar y reinstalar *Mininet*. En caso de no funcionar la única opción será reinstalar Ubuntu ya que es posible que exista alguna incompatibilidad generada por algún otro software.

A.3. Configuración controlador *OpenDayLight*

La gran ventaja de *OpenDayLight* es que los paquetes descargados vienen preconfigurados para uso directo del controlador. Como ya se ha comentado a lo largo del trabajo se ha optado por el uso de la versión base 0.2.2, pero existen varias versiones posteriores que aumentan las funcionalidades de ésta.

La ejecución de *OpenDayLight* puede presentar dos problemas principales. Que no conecten los *switches* emulados en *Mininet*. En este caso será necesario reiniciar la topología y controlador.

El segundo problema está provocado por las incompatibilidades con la versión 8 de JAVA, con lo que se recomienda el uso de la versión 7.

También se quiere destacar que para poder ver los logs que se incluyen en el código (información sobre el estado), este debe ser activado en el controlador. Por ejemplo para activar los log.trace de los paquetes ugr será necesario ejecutar (en la consola del controlador):

```
1 osgi> setLogLevel ugr trace
```

```
1 osgi> setLogLevel ugr debug
```

A.4. Problemas frecuentes

Muchos de los problemas que se pueden presentar ya han sido presentados en los diferentes apartados de configuración, se resumen aquí todos ellos.

- *Mininet* no realiza el test de pingall.

  Reiniciar, en caso negativo reinstalar *Mininet*. Si aún así sigue sin funcionar se deberá reinstalar Ubunutu.

- *Mininet* no conecta al controlador *OpenDayLight*.

  1. Comprobar que estamos indicando a *Mininet* la dirección correcta del controlador (si está en el mismo equipo será 127.0.0.1).
  2. Comprobar que el controlador se está ejecutando (visitar en un navegador localhost:8080).
  3. Reiniciar la topología
  4. Reiniciar controlador y topología.

  Si tras esto no se corrige el problema se deberá descargar de nuevo la versión del controlador o descargar otra.

- La aplicación desarrollada se instala pero no se ejecuta en el controlador.

  Comprobar que las librerías externas que se usen están instaladas (por ejemplo en nuestro indicar correctamente en el pom.xml esta librería).

  Intentar arrancar manualmente el bundle en la consola OSGi:

  ```
  osgi> start '''bundle number'''
  ```

  En caso de no conseguir funcionar, este intento devolverá el error concreto.

- La aplicación no hace nada o no instala flujos.

  Esto puede deberse a incompatibilidad con la versión del controlador. Para solucionarlo podemos descargar otra versión (anterior) del controlador o intentar adaptar la aplicación, siendo esta segunda opción de un mayor interés para el desarrollador.

Apéndice B

Diseño de topologías y algoritmo de encaminamiento

B.1. Ejemplo algoritmo *Dijkstra* paso a paso.

En esta sección se incluye paso a paso un ejemplo para el algoritmo *Dijkstra*. No se incluye explicación teórica debido a que entendemos que las imágenes que a continuación se presentan son lo suficientemente explicativas. Se pretende calcular el camino más corto (menor coste) para llegar del nodo A al nodo E.

Finalmente se han recorrido todos los caminos posibles y se ha obtenido que el camino de menor coste es el A-D-E.

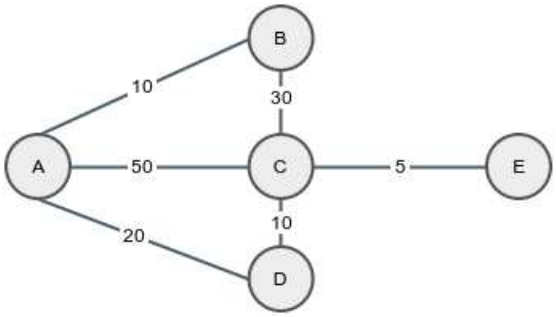
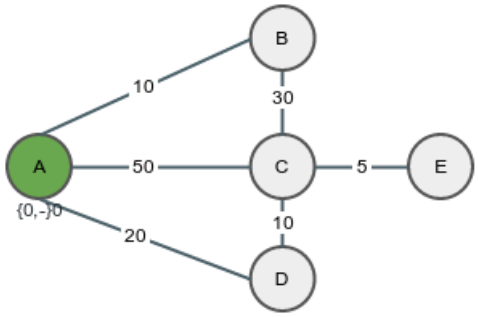
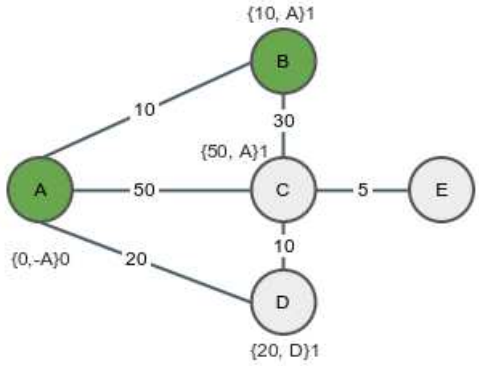


Figura B.1: Topología para ejemplo *Dijkstra*.

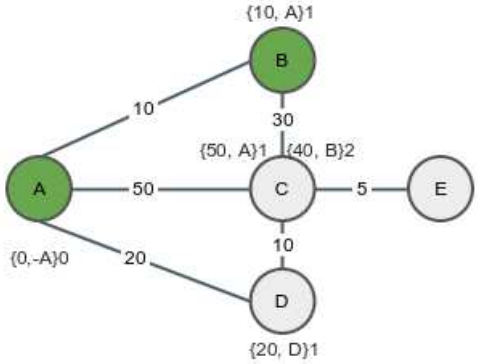


(a) Primer paso algoritmo *Dijkstra*.

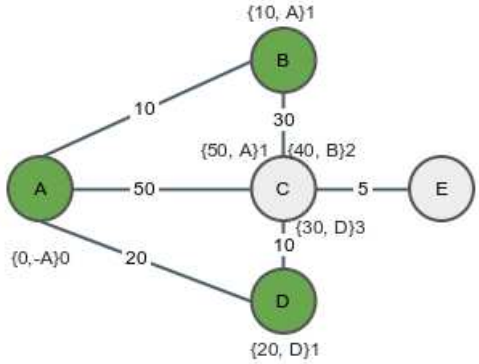


(b) Segundo paso algoritmo *Dijkstra*.

Figura B.2: Cálculo de algoritmo *Dijkstra*.



(a) Tercer paso algoritmo *Dijkstra*.



(b) Cuarto paso algoritmo *Dijkstra*.

Figura B.3: Cálculo de algoritmo *Dijkstra*.

B.2. Diseño de topologías Python API

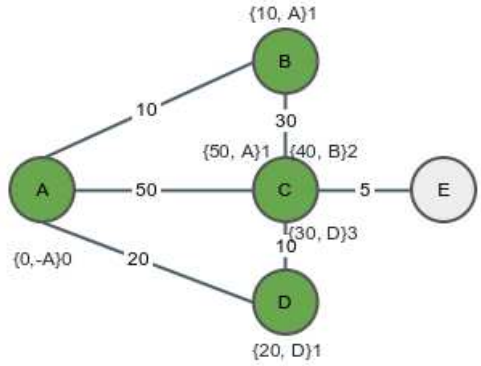
Haciendo uso de la Python API y [50] se han desarrollado las topologías usadas en este trabajo. A continuación se muestra una de ellas como ejemplo.

B.2.1. Topología básica

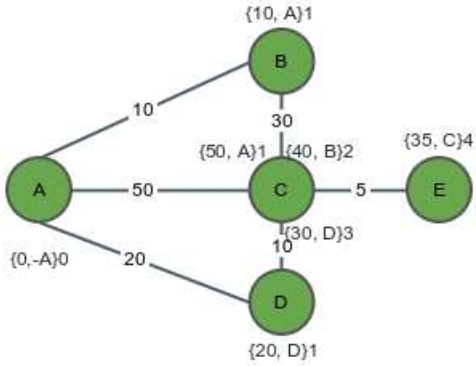
A continuación encontramos la definición en Python de la topología B.5.

Importante: Para introducir código fuente Python en LaTeX se ha usado la información presentada en [70].

```
1 '''
2 Copyright (C) 2015  Cristian Alfonso Prieto Sanchez
3
4 This program is free software: you can redistribute it and/or
    modify
5 it under the terms of the GNU General Public License as
    published by
6 the Free Software Foundation, either version 3 of the License,
    or
```



(a) Quinto paso algoritmo *Dijkstra*.



(b) Último paso algoritmo *Dijkstra*.

Figura B.4: Cálculo de algoritmo *Dijkstra*.

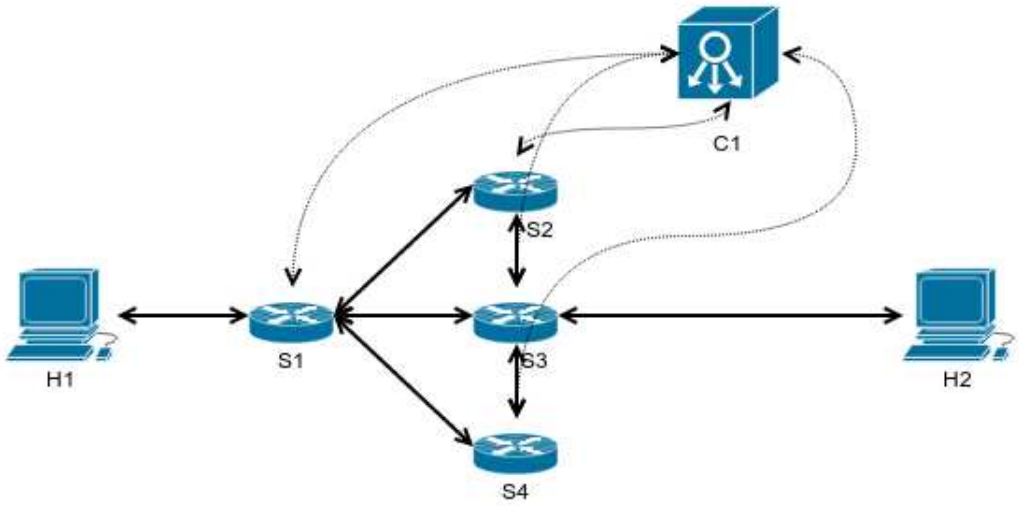


Figura B.5: Topología básica.

```
7  (at your option) any later version.
8
9  This program is distributed in the hope that it will be useful,
10 but WITHOUT ANY WARRANTY; without even the implied warranty of
11 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 GNU General Public License for more details.
13
14 You should have getTransmitErrorCountd a copy of the GNU General
       Public License
15 along with this program. If not, see <http://www.gnu.org/
       licenses/>.
16 '''
17
18
19 #!/usr/bin/python
20
```

```
21 from mininet.net import Mininet
22 from mininet.node import Controller, RemoteController,
      OVSController
23 from mininet.node import CPULimitedHost, Host, Node
24 from mininet.node import OVSKernelSwitch, UserSwitch
25 from mininet.node import IVSSwitch
26 from mininet.cli import CLI
27 from mininet.log import setLogLevel, info
28 from mininet.link import TCLink, Intf
29 from subprocess import call
30
31 def myNetwork():
32
33     net = Mininet( topo=None,
34                    listenPort=6633,
35                    build=False,
36                    ipBase='10.0.0.0/8')
37
38     info( '*** Adding controller\n' ) #Define the
          remoteController and the parameters like IP or protocol
39     c0=net.addController(name='c0',
40                          controller=RemoteController,
41                          protocol='tcp', protocols='OpenFlow13',
42                          port=6633)
43
44     info( '*** Add switches\n') #Define all the necessary
          Switches and the physical address
45     s1 = net.addSwitch('s1', cls=OVSKernelSwitch, mac='
          00:00:00:00:00:03', protocols='OpenFlow13')
46     s2 = net.addSwitch('s2', cls=OVSKernelSwitch, mac='
          00:00:00:00:00:04', protocols='OpenFlow13')
47     s3 = net.addSwitch('s3', cls=OVSKernelSwitch, mac='
          00:00:00:00:00:05', protocols='OpenFlow13')
48     s4 = net.addSwitch('s4', cls=OVSKernelSwitch, mac='
          00:00:00:00:00:06', protocols='OpenFlow13')
49
50     info( '*** Add hosts\n') #Define final Hosts and ther IPs
          and Mac
51     h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute
          =None, mac='00:00:00:00:00:02', protocols='OpenFlow13')
52     h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute
          =None, mac='00:00:00:00:00:01', protocols='OpenFlow13')
53
54     info( '*** Add links\n') #Add the links with their
          parameters
55     h1s1 = {'bw':1000,'loss':0,'delay':0}
56     net.addLink(h1, s1, cls=TCLink , **h1s1)
57     s1s2 = {'bw':10,'loss':0, 'delay':'10ms'}
58     net.addLink(s1, s2, cls=TCLink , **s1s2)
```

```
59      s1s3 = {'bw':10, 'loss':0, 'delay':'10ms'}
60      net.addLink(s1, s3, cls=TCLink , **s1s3)
61      s1s4 = {'bw':10, 'loss':0,'delay':'10ms'}
62      net.addLink(s1, s4, cls=TCLink , **s1s4)
63      s2s3 = {'bw':10,'loss':0,'delay':'10ms'}
64      net.addLink(s2, s3, cls=TCLink , **s2s3)
65      s4s3 = {'bw':10,'loss':0,'delay':'10ms'}
66      net.addLink(s4, s3, cls=TCLink , **s4s3)
67      s4h2 = {'bw':1000,'loss':0,'delay':0}
68      net.addLink(s3, h2, cls=TCLink , **s4h2)
69
70
71      info( '*** Starting network\n')
72      net.build()
73      info( '*** Starting controllers\n')
74      for controller in net.controllers:
75          controller.start()
76
77      info( '*** Starting switches\n')
78      net.get('s1').start([c0])
79      net.get('s2').start([c0])
80      net.get('s3').start([c0])
81      net.get('s4').start([c0])
82
83      info( '*** Post configure switches and hosts\n')
84
85      CLI(net)
86      net.stop()
87
88  if __name__ == '__main__':
89      setLogLevel( 'info' )
90      myNetwork()
```

Apéndice C

Código fuente de la aplicación desarrollada

En este apéndice se incluyen algunos de los archivos que se han usado para el código fuente de la aplicación desarrollada.

C.1. pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>ugr.cristian.serverVideoApp</groupId>
  <artifactId>serverVideoApp</artifactId>
  <packaging>bundle</packaging>
  <version>2.0-SNAPSHOT</version>
  <name>serverVideoApp</name>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.7</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Import-Package>
            *
            </Import-Package>
            <Export-Package>
              ugr.cristian.serverVideoApp
            </Export-Package>
            <Bundle-Activator>
```

```
27              ugr.cristian.serverVideoApp.Activator
28            </Bundle-Activator>
29          </instructions>
30          <manifestLocation>${project.basedir}/META-INF</manifestLocation>
31          <buildDirectory>/home/cristian/base/opendaylight/plugins</
                buildDirectory>
32        </configuration>
33      </plugin>
34    </plugins>
35  </build>
36  <dependencies>
37      <dependency>
38       <groupId>org.opendaylight.controller.thirdparty</groupId>
39       <artifactId>net.sf.jung2</artifactId>
40       <version>2.0.1</version>
41     </dependency>
42       <dependency>
43         <groupId>org.opendaylight.controller</groupId>
44         <artifactId>sal</artifactId>
45         <version>0.7.0</version>
46       </dependency>
47       <dependency>
48         <groupId>org.opendaylight.controller</groupId>
49         <artifactId>switchmanager</artifactId>
50         <version>0.7.0</version>
51       </dependency>
52       <dependency>
53         <groupId>org.opendaylight.controller</groupId>
54         <artifactId>topologymanager</artifactId>
55         <version>0.4.3-SNAPSHOT</version>
56       </dependency>
57       <dependency>
58         <groupId>org.opendaylight.controller</groupId>
59         <artifactId>statisticsmanager</artifactId>
60         <version>0.5.2-SNAPSHOT</version>
61       </dependency>
62       <dependency>
63         <groupId>junit</groupId>
64         <artifactId>junit</artifactId>
65         <version>3.8.1</version>
66         <scope>test</scope>
67       </dependency>
68    </dependencies>
69    <repositories>
70       <!-- OpenDaylight releases -->
71       <repository>
72         <id>opendaylight-mirror</id>
73         <name>opendaylight-mirror</name>
74         <url>http://nexus.opendaylight.org/content/groups/public/</url>
```

```
75        <snapshots>
76            <enabled>false</enabled>
77        </snapshots>
78        <releases>
79            <enabled>true</enabled>
80            <updatePolicy>never</updatePolicy>
81        </releases>
82      </repository>
83      <!-- OpenDaylight snapshots -->
84      <repository>
85        <id>opendaylight-snapshot</id>
86        <name>opendaylight-snapshot</name>
87        <url>http://nexus.opendaylight.org/content/repositories/opendaylight.
              snapshot/</url>
88        <snapshots>
89            <enabled>true</enabled>
90        </snapshots>
91        <releases>
92            <enabled>false</enabled>
93        </releases>
94      </repository>
95    </repositories>
96  </project>
```

C.2. Activator.java

```
1  /**
2  Copyright (C) 2015  Cristian Alfonso Prieto Sanchez
3
4  This program is free software: you can redistribute it and/or
       modify
5  it under the terms of the GNU General Public License as
       published by
6  the Free Software Foundation, either version 3 of the License,
       or
7  (at your option) any later version.
8
9  This program is distributed in the hope that it will be useful,
10 but WITHOUT ANY WARRANTY; without even the implied warranty of
11 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 GNU General Public License for more details.
13
14 You should have received a copy of the GNU General Public
       License
15 along with this program. If not, see <http://www.gnu.org/
       licenses/>.
16 */
17
18 package ugr.cristian.serverVideoApp;
19
20 import java.util.Dictionary;
21 import java.util.Hashtable;
22
23 import org.apache.felix.dm.Component;
24 import org.opendaylight.controller.sal.core.
       ComponentActivatorAbstractBase;
25 import org.opendaylight.controller.sal.flowprogrammer.
       IFlowProgrammerService;
26 import org.opendaylight.controller.sal.packet.IDataPacketService
       ;
27 import org.opendaylight.controller.sal.packet.IListenDataPacket;
28 import org.opendaylight.controller.sal.routing.IRouting;
29 import org.opendaylight.controller.switchmanager.ISwitchManager;
30 import org.opendaylight.controller.topologymanager.
       ITopologyManager;
31 import org.opendaylight.controller.statisticsmanager.
       IStatisticsManager;
32
33 import org.slf4j.Logger;
34 import org.slf4j.LoggerFactory;
35
36 public class Activator extends ComponentActivatorAbstractBase {
```

```
37      private static final Logger log = LoggerFactory.getLogger(
            PacketHandler.class);
38
39
40      /**
41       * Function called when the activator starts just after some
42       * initializations are done by the
43       * ComponentActivatorAbstractBase.
44       *
45       */
46      public void init() {
47      }
48
49      /**
50       * Function called when the activator stops just before the
51       * cleanup done by ComponentActivatorAbstractBase
52       *
53       */
54      public void destroy() {
55
56      }
57
58
59      /**
60       * Function that is used to communicate to dependency
             manager the
61       * list of known implementations for services inside a
             container
62       *
63       * @return An array containing all the CLASS objects that
             will be
64       * instantiated in order to get an fully working
             implementation
65       * Object
66       */
67      public Object[] getImplementations() {
68          log.trace("Getting Implementations");
69
70          Object[] res = { PacketHandler.class };
71          return res;
72      }
73
74      /**
75       * Function that is called when configuration of the
             dependencies
76       * is required.
77       *
78       * @param c dependency manager Component object, used for
79       * configuring the dependencies exported and imported
```

```
80      * @param imp Implementation class that is being configured,
81      * needed as long as the same routine can configure multiple
82      * implementations
83      * @param containerName The containerName being configured,
            this allow
84      * also optional per-container different behavior if needed,
            usually
85      * should not be the case though.
86      */
87
88     public void configureInstance(Component c, Object imp,
           String containerName) {
89         log.trace("Configuring instance");
90
91         if (imp.equals(PacketHandler.class)) {
92             // Define exported and used services for
                   PacketHandler component.
93
94             Dictionary<String, Object> props = new Hashtable<
                   String, Object>();
95             props.put("salListenerName", "myPacketHandler");
96
97             // Export IListenDataPacket interface to receive
                   packet-in events.
98             c.setInterface(new String[] {IListenDataPacket.class
                   .getName()}, props);
99
100            // Need the DataPacketService for encoding, decoding
                   , sending data packets
101            c.add(createContainerServiceDependency(containerName
                   ).setService(
102                    IDataPacketService.class).setCallbacks(
103                    "setDataPacketService", "
                       unsetDataPacketService")
104                    .setRequired(true));
105
106            // Need SwitchManager service for enumerating ports
                   of switch
107            c.add(createContainerServiceDependency(containerName
                   ).setService(
108                    ISwitchManager.class).setCallbacks(
109                    "setSwitchManagerService", "
                       unsetSwitchManagerService")
110                    .setRequired(true));
111
112            // Need FlowProgrammerService for programming flows
113            c.add(createContainerServiceDependency(containerName
                   ).setService(
114                    IFlowProgrammerService.class).setCallbacks(
```

```
115                     "setFlowProgrammerService", "
                           unsetFlowProgrammerService")
116                     .setRequired(true));
117
118             //Need a StatisticsManagerService to try to resolve
                   new challenges.
119             c.add(createContainerServiceDependency(containerName
                   ).setService(
120                     IStatisticsManager.class).setCallbacks(
121                     "setStatisticsManagerService", "
                           unsetStatisticsManagerService")
122                     .setRequired(true));
123
124             //Need a TopologyManagerService to try to obtain map
                    with the topology
125             c.add(createContainerServiceDependency(containerName
                   ).setService(
126                     ITopologyManager.class).setCallbacks(
127                     "setTopologyManagerService", "
                           unsetTopologyManagerService")
128                     .setRequired(true));
129         }
130
131     }
132 }
```

C.3. PacketHandler.java

En esta sección se incluyen algunas trazas código de la clase PacketHandler.java.

C.3.1. Implementación IListenDataPacket

```
1  @Override
2    public PacketResult receiveDataPacket(RawPacket inPkt) {
3      //Once a packet come the Topology has to be updated
4
5      timeSemaphore.tryAcquire();
6      t2 = System.currentTimeMillis();
7      if((t2-t1) > UPDATETIME){
8        updateTopology();
9        t1 = System.currentTimeMillis();
10     }
11     timeSemaphore.release();
12
13     //First I get the incoming Connector where the packet came
         .
14     NodeConnector ingressConnector = inPkt.
         getIncomingNodeConnector();
15     Packet pkt = dataPacketService.decodeDataPacket(inPkt);
16     log.trace("The packet came from " + ingressConnector + "
         NodeConnector");
17
18     //Now we obtain the node where we received the packet
19     Node node = ingressConnector.getNode();
20     log.trace("The packet came from " + node + " Node");
21
22     if(!hasHostConnected(ingressConnector)){EOL25
23       Edge upEdge = getUpEdge(node, ingressConnector);
24       if(upEdge != null){
25         calculateLatency(upEdge, pkt);
26       }
27     }
28
29     if(pkt instanceof Ethernet) {
30       //Pass the Ethernet Packet
31       Ethernet ethFrame = (Ethernet) pkt;
32       byte[] srcMAC_B = (ethFrame).getSourceMACAddress();
33       long srcMAC = BitBufferHelper.toNumber(srcMAC_B);
34       byte[] dstMAC_B = (ethFrame).getDestinationMACAddress();
35       long dstMAC = BitBufferHelper.toNumber(dstMAC_B);
36       Object l3Pkt = ethFrame.getPayload();
37
38       if(l3Pkt instanceof IPv4){
39         IPv4 ipv4Pkt = (IPv4)l3Pkt;
40         InetAddress srcAddr = intToInetAddress(ipv4Pkt.
             getSourceAddress());
```

```
41            InetAddress dstAddr = intToInetAddress(ipv4Pkt.
                 getDestinationAddress());
42            Object l4Datagram = ipv4Pkt.getPayload();
43
44
45            if(l4Datagram instanceof UDP){
46
47              UDP udpDatagram = (UDP) l4Datagram;
48              int clientPort = udpDatagram.getSourcePort();
49              int dstPort = udpDatagram.getDestinationPort();
50
51
52              byte[] udpRawPayload = udpDatagram.getRawPayload();
53
54              if(this.rtpRouting.detectRTPPacket(udpRawPayload,
                   dstPort)){
55                return handleRTPPacket(inPkt, srcAddr, srcMAC_B,
                     ingressConnector, node, dstAddr, dstMAC_B,
                     dstPort);
56              }
57              else if(this.audioRouting.detectAudioPacket(
                   udpRawPayload, dstPort)){
58                return handleAudioPacket(inPkt, srcAddr, srcMAC_B,
                      ingressConnector, node, dstAddr, dstMAC_B,
                     dstPort);
59              }
60
61            }else if(l4Datagram instanceof TCP){
62              TCP tcpDatagram = (TCP) l4Datagram;
63              int clientPort = tcpDatagram.getSourcePort();
64              int dstPort = tcpDatagram.getDestinationPort();
65
66              return handleTCPPacket(inPkt, srcAddr, srcMAC_B,
                   ingressConnector, node, dstAddr, dstMAC_B,
                   dstPort);
67            }else   if(l4Datagram instanceof ICMP){
68
69                return handleICMPPacket(inPkt, srcAddr, srcMAC_B,
                     ingressConnector, node, dstAddr, dstMAC_B);
70            }
71          }
72        }
73        return PacketResult.IGNORED;
74      }
```

C.3.2. updateTopology

```
/**
    *Function that is called when is necessary update the
       current Topology store
    */

    synchronized private void updateTopology(){

      Map<Node, Set<Edge>> edges = this.topologyManager.
         getNodeEdges();


      if(this.nodeEdges.isEmpty() || !this.nodeEdges.equals(
         edges) || this.nodeEdges == null){

        MAXFLOODPACKET = 100*this.nodeEdges.size();

        this.packetTime.clear();
        this.edgePackets.clear();
        this.edgeBandWith.clear();
        this.minBandWith=0L;
        this.edgeMapTime.clear();
        flood=0;

        if(this.nodeEdges!=null && edges!=null){
          removeOldFlow(edges);
        }

        this.nodeEdges = edges;

        buildEdgeMatrix(edges);

        log.trace("The new map is " + this.nodeEdges);
        resetLatencyMatrix();
        createTopologyGraph();

        updateEdgeStatistics();

        if(first){
          resetRoutingProtocols();
          first=false;
        }

        log.debug("The topology has been updated");

      }

        updateEdgeStatistics();
```

```
45          this.icmpSemaphore.tryAcquire();
46          this.icmpRouting.updateStandardCostMatrix(this.nodeEdges
                , this.edgeMatrix, this.latencyMatrix, this.
                minLatency,
47          this.mediumLatencyMatrix, this.minMediumLatency, this.
                edgeStatistics, this.maxStatistics);
48          this.icmpSemaphore.release();
49
50          this.tcpSemaphore.tryAcquire();
51          this.tcpRouting.updateTCPCostMatrix(this.nodeEdges, this
                .edgeMatrix, this.latencyMatrix, this.minLatency,
52          this.mediumLatencyMatrix, this.minMediumLatency, this.
                edgeStatistics, this.maxStatistics, this.edgeBandWith
                , this.minBandWith);
53          this.tcpSemaphore.release();
54
55
56          this.rtpSemaphore.tryAcquire();
57          this.rtpRouting.updateRTPCostMatrix(this.nodeEdges, this
                .edgeMatrix, this.latencyMatrix, this.minLatency,
58          this.mediumLatencyMatrix, this.minMediumLatency, this.
                edgeStatistics, this.maxStatistics, this.edgeBandWith
                , this.minBandWith);
59          this.rtpSemaphore.release();
60
61          this.audioSemaphore.tryAcquire();
62          this.audioRouting.updateAudioCostMatrix(this.nodeEdges,
                this.edgeMatrix, this.latencyMatrix, this.minLatency,
63          this.mediumLatencyMatrix, this.minMediumLatency, this.
                edgeStatistics, this.maxStatistics, this.edgeBandWith
                , this.minBandWith);
64          this.audioSemaphore.release();
65      }
```

C.3.3. Comprobación de latencias

```
1  /**
2    *This function is called when a Packet come and is necessary
         to calculate the latency
3    *@param edge The associate edge
4    *@param packet The packet which came just now
5    */
6
7    private void calculateLatency(Edge edge, Packet packet){
8
9      Set<Packet> temp = this.edgePackets.get(edge);
10     if(checkSetPacket(packet, temp)){
11       temp.remove(packet);
12       this.edgePackets.remove(edge);
13       this.edgePackets.put(edge, temp);
14       Long t2 = System.nanoTime();
15       Long t1 = returnPacketTime(edge, packet);
16
17       if(t1!=null){
18         Long t = t2-t1;
19         updateLatencyMatrix(edge, t);
20         if(minLatency == null){
21           minLatency=t;
22         }
23         else if(minLatency> t){
24           minLatency=t;
25         }
26       }
27
28     }
29
30   }
```

C.4. Implementación manejador RTP Vídeo

```
/**
    *Function that is called when a RTP Packet needs to be
       Handled
    *@param inPKt The received Packet
    *@param srcAddr The src IP Address
    *@param srcMAC_B The src MAC Address in bytes
    *@param ingressConnector The connector where the packet came
    *@param node The node where the packet have been received
    *@param dstAddr The dst IP Address
    *@param dstMAC_B The dst MAC Address in bytes
    *@param dstPort The RTP dstPort that identify the protocol
    *@return result The result of handle the RTP Packet
    */

    private PacketResult handleRTPPacket(RawPacket inPkt,
       InetAddress srcAddr, byte[] srcMAC_B,
    NodeConnector ingressConnector, Node node, InetAddress
       dstAddr, byte[] dstMAC_B, int dstPort){

      Packet pkt = dataPacketService.decodeDataPacket(inPkt);
      NodeConnector egressConnector=null;
      PacketResult result = ---;

      if(flood<MAXFLOODPACKET){
        this.flood++;
        floodPacket(inPkt, node, ingressConnector);
      }else{

        NodeConnector tempSrcConnector = findHost(srcAddr);
        Node tempSrcNode = tempSrcConnector.getNode();
        NodeConnector tempDstConnector = findHost(dstAddr);
        Node tempDstNode = tempDstConnector.getNode();

        Map<Node, Node> tempMap = new HashMap<Node, Node>();
              tempMap.put(tempSrcNode, tempDstNode);

              List<Edge> definitivePath = new ArrayList<Edge>();

        try{
          this.rtpSemaphore.tryAcquire();
          definitivePath = this.rtpRouting.getRTPShortestPath(
             tempSrcNode, tempDstNode);
          this.rtpSemaphore.release();
        }
        catch(RuntimeException badDijkstraRTP){
          log.info("Impossible to obtain the best Path.");
```

```
43            log.info("If the problem persist please update your
                topology (link s1 s2 down and up for example)");
44
45            this.rtpSemaphore.tryAcquire();
46            this.rtpRouting = new RTPRouting(this.nodeEdges, this.
                edgeMatrix, this.latencyMatrix, this.minLatency,
47            this.mediumLatencyMatrix, this.minMediumLatency, this.
                edgeStatistics, this.maxStatistics, this.g, this.
                edgeBandWith, this.minBandWith);
48            this.rtpSemaphore.release();
49
50            return PacketResult.IGNORED;
51          }
52
53          if(definitivePath != ---){
54
55            egressConnector = installRTPListFlows(definitivePath,
                srcAddr, srcMAC_B, dstAddr, dstMAC_B, node,
56            tempSrcConnector, tempDstConnector, dstPort);
57
58
59            if(!hasHostConnected(egressConnector)){
60              Edge downEdge = getDownEdge(node, egressConnector);
61              if(downEdge!= ---){
62                putPacket(downEdge, pkt);
63              }
64            }
65
66            if(egressConnector!= ---){
67            //Send the packet for the selected Port.
68              inPkt.setOutgoingNodeConnector(egressConnector);
69              this.dataPacketService.transmitDataPacket(inPkt);
70            }else{
71              floodPacket(inPkt, node, ingressConnector);
72            }
73            result = PacketResult.CONSUME;
74
75          }else{
76            log.trace("Destination host is unrecheable!");
77            result = PacketResult.IGNORED;
78          }
79        }
80
81        return result;
82      }
```

C.4.1. Construcción Transformer RTP Vídeo

```
1         /**
2       *Function that is called when is necessary to build the
           transformer rtp for Dijkstra
3       */
4
5       private void buildRTPTransformerMap(final Map<Edge, Double
           > rtpEdgeCostMap2){
6
7         this.costRTPTransformer = new Transformer<Edge, Number
             >(){
8           public Double transform(Edge e){
9
10            return rtpEdgeCostMap2.get(e);
11          }
12        };
13
14      }
```

Bibliografía

[1] R. A. Cacheda, D. C. García, A. Cuevas, F. J. G. Castano, J. H. Sánchez, G. Koltsidas, V. Mancuso, J. I. M. Novella, S. Oh, and A. Pantò, "Qos requirements for multimedia services," in *Resource Management in Satellite Networks*. Springer, 2007, pp. 67–94.

[2] Google@OpenFlow. (2013) Google @ openflow. Último acceso 10 de Junio de 2015. [Online]. Available: http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf

[3] C. V. Networking. (2014) Cisco visual networking index: Forecast and methodology, 2013–2018. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html

[4] ——. (2015) Global mobile data traffic forecast update, 2014–2019 white paper. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

[5] Verizon. (2012) Adaptation sdn: Progress update. Último acceso 10 de Junio de 2015. [Online]. Available: http://opennetsummit.org/archives/apr12/elby-tue-sdn.pdf

[6] C. N. S. Jamil Chawki and O. A. G. M. Standards Manager, Orange. (2015) How is orange using opendaylight. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.opendaylight.org/blogs/2015/06/how-orange-using-opendaylight

[7] M. Boucadair and C. Jacquenet, "Software-defined networking: A perspective from within a service provider environment," 2014.

[8] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[9] L. Foundation. (2015) Opendaylight developer guide. Último acceso 20 de Junio de 2015. [Online]. Available: https://www.opendaylight.org/sites/opendaylight/files/Developer-Guide-Helium-SR2.pdf

[10] ITU, "El modelo e: un modelo informático para utilización en planificación de transmisión," 2014. [Online]. Available: https://www.itu.int/rec/T-REC-G.107-201402-I/es

[11] ——, "Opinion model for video-telephony applications," 2012. [Online]. Available: http://www.itu.int/rec/T-REC-G.1070-201207-I/en

[12] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4d approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 1–12.

[14] U. de Priceton. (2007) Planetlab. Último acceso 30 de Junio de 2015. [Online]. Available: https://www.planet-lab.org/

[15] U. de Utah. (2015) Emulab. Último acceso 30 de Junio de 2015. [Online]. Available: https://www.emulab.net/

[16] O. N. Foundation, "Software-defined networking (sdn) definition," https://www.opennetworking.org/sdn-resources/sdn-definition, 2013.

[17] ATT. (2013) Att launches supplier domain program 2.0. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.att.com/gen/press-room?pid=24817&cdvn=news&newsarticleid=37013

[18] ONF. (2014) Sdn architecture overview 1.1. Último acceso 10 de Junio de 2015. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN-ARCH-Overview-1.1-11112014.02.pdf

[19] sdxCentral. (2013) What are sdn controllers? Último acceso 10 de Junio de 2015. [Online]. Available: https://www.sdxcentral.com/resources/sdn/sdn-controllers/

[20] O. N. Foundation. (2015) Openflow definition by onf. Último acceso 10 de Junio de 2015. [Online]. Available: https://www.opennetworking.org/sdn-resources/openflow

[21] M. Team. (2015) Mininet. Último acceso 18 de Junio de 2015. [Online]. Available: http://mininet.org/

[22] IETF. (2015) Cisco opflex. Último acceso 10 de Junio de 2015. [Online]. Available: https://tools.ietf.org/html/draft-smith-opflex-00

[23] O. N. Foundation. (2015) Openflow switch specification, version 1.3.5. Último acceso 10 de Junio de 2015. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.5.pdf

[24] CISCO. (2015) Cisco controller apic. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.cisco.com/c/en/us/products/cloud-systems-management/application-policy-infrastructure-controller-apic/index.html

[25] HP. (2015) Hp controller van. Último acceso 10 de Junio de 2015. [Online]. Available: http://h17007.www1.hp.com/us/en/networking/products/network-management/HP_VAN_SDN_Controller_Software/index.aspx#.VXA2pq3tmko

[26] R. Khondoker, A. Zaalouk, R. Marx, and K. Bayarou, "Feature-based comparison and selection of software defined networking (sdn) controllers," in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on.* IEEE, 2014, pp. 1–7.

[27] S. A. Shah, J. Faiz, M. Farooq, A. Shafi, and S. A. Mehdi, "An architectural evaluation of sdn controllers," in *Communications (ICC), 2013 IEEE International Conference on.* IEEE, 2013, pp. 3504–3508.

[28] NOXRepo.org. (2015) Pox controller. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[29] R. Team. (2015) Ryu controller. Último acceso 10 de Junio de 2015. [Online]. Available: http://osrg.github.io/ryu/

[30] T. group. (2015) Trema c and ruby framework. Último acceso 10 de Junio de 2015. [Online]. Available: http://trema.github.io/trema/

[31] P. Floodlight. (2015) Floodlight controller. Último acceso 10 de Junio de 2015. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[32] J. Postel, "Transmission control protocol," 1981.

[33] U. D. Protocol, "Rfc 768 j. postel isi 28 august 1980," *Isi*, 1980.

[34] V. Jacobson, R. Frederick, S. Casner, and H. Schulzrinne, "Rtp: A transport protocol for real-time applications," 2003.

[35] S. Deering, "Icmp router discovery messages," 1991.

[36] J. Postel, "Internet protocol," 1981.

[37] Cisco. (2014) Cisco xnc 1.5.0 api. Último acceso 21 de Junio de 2015. [Online]. Available: https://developer.cisco.com/media/XNCJavaDocs/overview-summary.html

[38] M. Nilsson, "The audio/mpeg media type," 2000.

[39] ITU, "711"modulación por impulsos codificados (mic) de frecuencias vocales," *Series G: Codificación de las señales analógicas". CCITT Libro naranja (Ginebra 1976)*, 1993. [Online]. Available: https://www.itu.int/rec/T-REC-G.711/es

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms 2nd edition," pp. 588–592, 2001.

[41] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[42] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press y McGraw-Hill, 1990, ch. 22, pp. 558–565.

[43] D. Berriman, "What is jitter?" *Hi-Fi News and Record Review*, vol. 41, no. 3, pp. 64–5, 1996.

[44] Canonical. (2015) Ubuntu. Último acceso 18 de Junio de 2015. [Online]. Available: http://www.ubuntu.com/

[45] R. Hat. (2015) Fedora project. Último acceso 18 de Junio de 2015. [Online]. Available: https://getfedora.org/es/

[46] U. s. T. Canonical. (2015) Long term support, lts, ubuntu. Último acceso 4 de Julio de 2015. [Online]. Available: https://wiki.ubuntu.com/LTS

[47] M. Team. (2015) Mininet. Último acceso 18 de Junio de 2015. [Online]. Available: http://mininet.org/credits

[48] E. technologies. (2015) Estinet 9.0. Último acceso 18 de Junio de 2015. [Online]. Available: http://www.estinet.com/products.php?lv1=13&sn=13

[49] S.-Y. Wang, "Comparison of sdn openflow network simulator and emulators: Estinet vs. mininet," in *Computers and Communication (ISCC), 2014 IEEE Symposium on.* IEEE, 2014, pp. 1–6.

[50] M. Team. (2015) Mininet python api reference manual. Último acceso 18 de Junio de 2015. [Online]. Available: http://mininet.org/api/index.html

[51] A. S. Foundation and A. K. Team. (2015) Apache karaf. Último acceso 18 de Junio de 2015. [Online]. Available: http://karaf.apache.org/

[52] M. Björklund, "Yang-a data modeling language for netconf," Internet Draft (work in progress)¡draft-ietf-netmod-yang-07. txt¿, Tail-f Systems, Tech. Rep., 2009.

[53] S. Tutorials. (2015) Difference between ad-sal and md-sal. Último acceso 20 de Junio de 2015. [Online]. Available: http://sdntutorials.com/difference-between-ad-sal-and-md-sal/

[54] M. D. Network and individual contributors. (2015) What is the dom? Último acceso 21 de Junio de 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[55] S. Tutorials and Kanika. (2015) Opendaylight, netconf, restconf and yang. Último acceso 21 de Junio de 2015. [Online]. Available: http://sdntutorials.com/opendaylight-netconf-restconf-and-yang/

[56] ——. (2015) What is restconf? Último acceso 21 de Junio de 2015. [Online]. Available: http://sdntutorials.com/what-is-restconf/

[57] A. S. Foundation. (2015) What is apache maven? Último acceso 21 de Junio de 2015. [Online]. Available: https://maven.apache.org/what-is-maven.html

[58] T. E. Foundation. (2015) Eclipse. Último acceso 21 de Junio de 2015. [Online]. Available: https://eclipse.org/

[59] G. Inc. (2015) Atom. Último acceso 21 de Junio de 2015. [Online]. Available: https://atom.io/

[60] L. Foundation. (2015) Configure eclipse for opendaylight development. Último acceso 21 de Junio de 2015. [Online]. Available: https://wiki.opendaylight.org/view/GettingStarted:_Eclipse

[61] O. vSwitch. (2014) What is open vswitch? Último acceso 21 de Junio de 2015. [Online]. Available: http://openvswitch.org/

[62] P. S. Cristian Alfonso and S. L. Manuel. (2015) Blog aprendiendo opendaylight. Último acceso 22 de Junio de 2015. [Online]. Available: http://aprendiendoodl.wordpress.com

[63] Oracle and affiliates. (2013) Jacadoc for system. Último acceso 25 de Junio de 2015. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime%28%29

[64] SourceForge. (2009) Clase dijkstrashortestpath. Último acceso 23 de Junio de 2015. [Online]. Available: http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/algorithms/shortestpath/DijkstraShortestPath.html

[65] M. F. Finneran, *Voice over WLANS: The complete guide.* Newnes, 2011.

[66] C. Ltd. and Ubuntu. (2010) Mausezahn software for ubuntu. Último acceso 5 de Julio de 2015. [Online]. Available: http://manpages.ubuntu.com/manpages/karmic/man1/mz.1.html

[67] ITU, "P.800. methods for subjective determination of transmission quality," *Series P: Telephone Transmission Quality; Methods for Objective and Subjective Assessment of Quality*, 1996. [Online]. Available: https://www.itu.int/rec/T-REC-P.800-199608-I/es

[68] J. J. Ramos-Munoz and J. M. Lopez-Soler, "Efficient allocation for voip packet losses detection services in programmable and active networks," in *Autonomic and Autonomous Systems and International Conference on Networking and Services, 2005. ICAS-ICNS 2005. Joint International Conference on.* IEEE, 2005, pp. 52–52.

[69] J. Joskowicz and J. Ardao, "Enhancements to the opinion model for video-telephony applications," in *Proceedings of the 5th International Latin American Networking Conference.* ACM, 2009, pp. 87–94.

[70] W. Daniel. (2013) Syntax highlighting for python scripts in latex documents. Último acceso 22 de Junio de 2015. [Online]. Available: http://widerin.net/blog/syntax-highlighting-for-python-scripts-in-latex-documents