TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE LAS TELECOMUNICACIONES

Implementación de movilidad en redes 5G

Simulación de movilidad con Software Defined Networking

Autor
Rafael González Callejas

Directores
Jorge Navarro Ortiz
Juan José Ramos Muñoz

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Septiembre de 2017

Implementación de movilidad en redes 5G.

Simulación de movilidad mediante Software Defined Networking.

Autor
Rafael González Callejas

Directores
Jorge Navarro Ortiz
Juan José Ramos Muñoz

Implementación de movilidad en redes 5G: Simulación de movilidad con Software Defined Networking

Rafael González Callejas

Palabras clave: *(Software Defined Network)* SDN, Movilidad, *Handover*, OpenDayLight, Mininet, *Framework*, OpenFlow, 5G, Controlador, Java.

Resumen

En un mundo marcado por el enorme crecimiento de los dispositivos con conexión a Internet, donde cada vez estamos más conectados y usamos los recursos de la red en la realización de las tareas más cotidianas, las futuras redes de comunicaciones deben tender hacia una mayor simplicación y flexibilidad, utilizando un *hardware* más sencillo e interoperable y menos costoso, invirtiendo en sistemas de gestión más sofisticados y eficientes que gestionen dicha demanda de recursos de la forma más inteligente posible.

En este sentido, tecnologías como las redes definidas por software se postulan como una alternativa en vías de desarrollo para satisfacer las nuevas necesidades y requerimientos del ecosistema de red. La filosofía detrás de las redes definidas por software es la de separar el plano de control del plano de datos de las redes de comunicaciones, de forma que toda la inteligencia de la red se centralice y se gestione a nivel *software* mientras los dispositivos de red se vuelven más simples y compatibles.

De la mano de esta centralización y gestión de los recursos de red a nivel *software*, OpenDayLight surge como un *framework* altamente mantenido y soportado por un consorcio de importantes fabricantes y operadoras de red, con las herramientas necesarias para gestionar todo el flujo de red, obtener estadísticas y facilitar el desarrollo de la inteligencia de red a través de aplicaciones y *plugins* específicamente diseñados para consumir los datos generados por los dispositivos de red y aplicar las reglas necesarias para generar el comportamiento que se desee conseguir.

Al hilo del desarrollo de estas tecnologías, OpenFlow aparece de la mano de la *(Open Networking Foundation)* ONF como un protocolo pensado para establecer la comunicación entre el plano de control y el plano de datos de una red, facilitando la instalación y borrado de flujos y la gestión de las tablas de flujo de los *switches* compatibles con esta tecnología.

En este contexto, este proyecto de fin de grado está destinado a estudiar y emular la gestión de movilidad en la que será la quinta generación de redes

móviles. Trataremos de emular el proceso de movilidad completo, desde el proceso de señalización, a través de aplicaciones Java, hasta la generación de un escenario usando elementos simulados con la herramienta Mininet, que nos permite utilizar *switches* compatibles con el protocolo OpenFlow y que está fuertemente integrada con OpenDayLight.

Por supuesto, desarrollaremos un controlador integrado en OpenDayLight que se encargará del descubrimiento e instalación de las rutas de encaminamiento de los paquetes intercambiados por nuestra red, a través de la instalación de flujos por medio del protocolo OpenFlow. Trabajaremos principalmente con Java como lenguaje de programación, si bien, la generación de la topología de red se realizará en Python, y utilizaremos *scripts bash* para la gestión de las tarjetas de red que actuarán como puntos de acceso.

Mobility implementation on 5G Networks: Mobility simulation with Software Defined Networking

Rafael González

Keywords: SDN (Software Defined Network), Mobility, Handover, OpenDayLight, Framework, OpenFlow, 5G, Controller, Java.

Abstract

Nowadays, with the exponential growth of devices consumming network resources, even for the most quotidian tasks, where the network claims for a more flexible and adaptative paradigm, which allows a reduction on hardware investment and pushes for a more centralized network management, the future fith generation of mobile communications is intended to develop a new philosophy of network management in order to deal with this requirements.

Given this scenario, technologies such as SDN (Sofware Defined Network) are being considered as an alternative to reach the goals represented by this new network requirements. SDN philosophy is based on the division between data plane and control plane, so that, network devices can be simpler, more interopeable and cheaper since network intelligence is centralized in software based applications.

According to this new paradigm, OpenDayLight arises as a versatile framework for network management, highly supported and mantained by a wide group of network manufacturers and operators. This technology is expected to be an alternative for network management, provided its capacity to extract network information so that an external application can be aware of network devices and topology and can also be able to install, delete and modify data flows depending on the desired network behaviour.

Since OpenDayLight is a very useful tool for network management, a protocol that allows OpenDayLight to communicate with network devices is required. OpenFlow is the first protocol developed for this purpose by the (ONF) Open Networking Foundation. OpenFlow is responsible for the communication between the controller application, where the business inteligence is implemented, and network devices where the actions ordered by the controller are applied.

Considering this evolution of network communications, this project will try to study and emulate a mobility scenario based on those new network

paradigms. We will use Mininet as a tool for simulating a network topology that allows us to test the handover procedure in a 5G network, which will be controlled by an application running over OpenDayLight.

During the project we will work mainly with Java, for developing all the applications and controller that take part on the simulation. We will also use Python to simulate our network topology, and bash scripts to configure network components such as wireless network interface cards.

Yo, **Rafael González Callejas**, alumno del Grado en Ingeniería de Tecnologías de Telecomunicación de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Rafael González Callejas

Granada a 12 de Septiembre de 2017.

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

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

Informan:

Que el presente trabajo, titulado ***Implementación de movilidad en redes 5G, Simulación de movilidad con Software Defined Networking***, ha sido realizado bajo su supervisión por **Rafael González Callejas**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

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

Los directores:

Jorge Navarro Ortiz **Juan José Ramos Muñoz**

Agradecimientos

A mi familia y amigos, que han supuesto un gran apoyo a lo largo de todos estos años, tanto en los buenos como en los malos momentos. A Bárbara, amiga y compañera durante este TFG y durante muchos años de carrera, a pesar de nuestros carácteres no podría haber encontrado una compañera mejor en este camino.

A mi madre, que ha rezado para que todo saliera bien antes de cada examen, ha sufrido mis desvelos, ha sido enfermera, psicóloga, profesora, taxista... pero sobre todo, mi ejemplo a seguir y mi amiga, sin ella nada de esto hubiera sido posible.

A Juanjo, y en especial a Jorge, por su dedicación, ayuda constante y predisposición, gracias a él este proyecto puede ver la luz.

A todos, gracias.

Índice de Acrónimos

AAA Authentication, Authorization and Accounting

AC Access Cloud

AD-SAL API Driven Service Abstraction Layer

ALTO Application Layer Traffic Optimization

AP Access Point

ARP Address Resolution Protocol

CapEx Capital Expenditure

CDMA Code Division Multiple Access

CDPI Control Data Plane Interface

CoAP Constrained Application Protocol

D-AMPS Digital Advanced Mobile Phone System

DAWN Dynamic Adhoc Wireless Network

DC Data Center

eNB evolved Node B

FTP File Transfer Protocol

GBP Group Based Policy

GSM Global System for Mobile Communications

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IP Internet Protocol

JAAS Java Authentication and Authorization Service

JMX Java Management Extensions

LACP Link Aggregation Control Protocol

LTE Long Term Evolution

MAC Media Access Control

MD-SAL Model Driven Service Abstraction Layer

NAT Network Adress Translation

NB North Bound

NC National Cloud

NEMO Network Mobility

NETCONF Network Configuration Protocol

NIC Network Intent Composition

NFV Network Function Virtualization

ONF Open Networking Foundation

OpEx Operating Expenditure

OWA Open Wireless Architecture

OSGi Open Services Gateway initiative

PCMM/COPS Packet Cable MultiMedia/Common Open Policy Service

PHS Personal Handyphon System

RAN Radio Access Network

RBAC Role-Based Access Control

RC Regional Cloud

REST Representational State Transfer

RPC Remote Procedure Call

RTT Round Trip Time

SAL Service Abstraction Layer

SB South Bound

SDN Software Defined Network

SDWN Software Defined Wireless Networking

SSH Secure Shell

STA Station

TCP Transmission Control Protocol

TDMA Time Division Multiple Access

TLS Transport Layer Security

TM Topology Manager

UE User Equipment

UDP User Datagram Protocol

WWWW World Wide Wireless Web

XML Extensible Markup Language

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

En este apartado vamos a tratar de aportar al lector un marco de referencia contextualizado sobre la situación actual de las redes de comunicaciones móviles. El objeto de este trabajo es estudiar el comportamiento de la futura red móvil 5G en escenarios de movilidad, para lo cuál es necesaria una amplia base en el funcionamiento (arquitectura, protocolos...) de los modelos previos. Es por ello que a lo largo de este proyecto vamos a tratar de analizar los modelos existentes, así como de describir y argumentar las razones que nos conducen a utilizar *Software Defined Network* (SDN) [1] para la emulación de un escenario de movilidad similar al que se implementaría en una red 5G.

La existencia de un software como OpenDayLight (controlador SDN) y de la herramienta Mininet (emulador de redes), que comentaremos en apartados posteriores, nos ofrece la posibilidad de emular escenarios reales, pudiendo configurarlos y gestionarlos de la forma que más se adapte a los requisitos que exija nuestro diseño. Si bien esto supone un nivel complejo de configuración, y una serie de obstáculos a la hora de realizar incluso las tareas más sencillas, también supone la capacidad de trabajar en una tecnología que se encuentra a la orden del día y que va a jugar un papel fundamental en el desarrollo de las próximas generaciones de redes móviles. Estos factores suponen el empujón necesario para emprender un proyecto de estas características.

1.2. Objetivos

El objetivo final de este proyecto es emular de la forma más exacta y efectiva posible un escenario de movilidad en una red móvil 5G. Para ello será necesario crear el escenario apropiado en un emulador de redes, configurar un controlador que gestione el envío de paquetes y la instalación de flujos de datos, y simular el comportamiento de una estación móvil que se conecte a nuestra red y realice el proceso de movilidad entre estaciones base, de acuerdo a los escenarios mostrados en [2].

Entre otras cosas, se pretende emular el comportamiento de un traspaso (*handover*) como el realizado en las redes móviles convencionales [3], aplicado al nuevo paradigma basado en SDN. Dado que no se dispone de estaciones base 5G y que sería muy costoso modificar la señalización entre un móvil y una estación base, se utilizarán puntos de acceso o Access Point (AP) Wi-Fi para emular dicho traspaso. Así, se simulará la señalización que aparece en [4] para forzar que un PC cambie su conexión de un AP a otro. Un controlador gestionará la modificación de los flujos necesaria en los switches SDN para que la conexión a Internet del PC se mantenga, y un *sniffer* (wireshark en nuestro caso) medirá la pérdida de paquetes en el proceso.

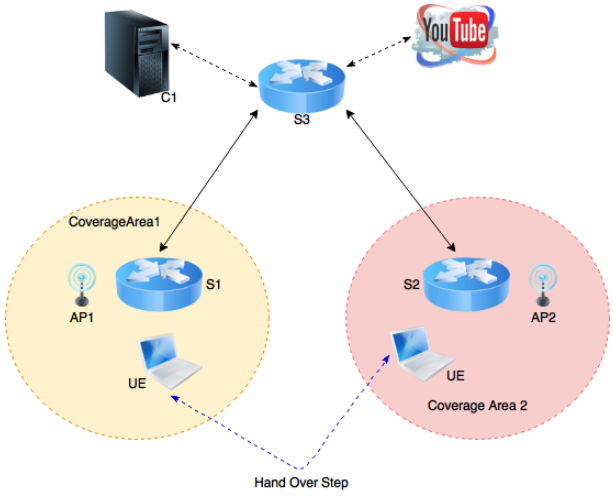


Figura 1.1: Simulación de movilidad a realizar. Figura realizada con draw.io.

La Figura 1.1 ejemplifica el proceso descrito en el párrafo anterior. En ella vemos con más detalle la topología que se pretende implementar, y el cambio entre puntos de acceso a realizar.

Como se aprecia, a grandes rasgos se pretende implementar una topología formada por tres switches (S1, S2 y S3), un controlador (C1), dos puntos de acceso (AP1 y AP2) y un PC que hará las veces de *User Equipment* (UE).

Este escenario nos permitirá realizar el cambio de celda entre los switches 1 y 2 manteniendo a su vez la conexión de datos en curso, e.g. para la descarga de un video de Youtube, a través del switch 3 o para la descarga de contenido a través de una sesión *File Transfer Protocol* (FTP).

En capítulos posteriores explicaremos con un mayor nivel de detalle el diseño e implementación de esta topología.

1.3. Contexto

En este apartado vamos a tratar de dotar al lector de una perspectiva global que le permita comprender el concepto entorno al que va a girar este proyecto, la gestión de movilidad en redes 5G. Para ello, en primer lugar se va a introducir el concepto de 5G, centrándose en los cambios fundamentales introducidos por esta tecnología con respecto a las redes 4G.

A continuación pasaremos a valorar la importancia del paradigma SDN en este tipo de redes, sus ventajas y las posibilidades que representa.

Finalmente, como resumen, se presenta la estructura que va a seguir el resto de la memoria, añadiendo un breve resumen de los conceptos contenidos en cada parte.

1.3.1. Introducción a la movilidad en redes 5G

Es evidente que en la última década las comunicaciones móviles han sufrido una evolución enorme, no sólo en el número de usuarios que hacen uso de ellas, creciendo exponencialmente, sino también en las prestaciones y las capacidades demandadas por el creciente número de aplicaciones, cada vez más sofisticadas y con requisitos de ancho de banda y latencias mucho más exigentes. En este sentido, las redes 4G suponen el último escalón del desarrollo de tecnologías de comunicaciones móviles tradicionales, que poco a poco dejará paso a un nuevo paradigma representado por la quinta generación de redes móviles.

La tecnología 5G es una tecnología orientada al consumidor, todavía en fase de desarrollo y prototipado, que se centrará en la simplificación de su arquitectura, la reducción de costes, y el soporte de diferentes tipos de servicio con requisitos muy heterogéneos.

No obstante, la optimización de la gestión de la movilidad, dejada a un lado en 4G, es uno de los principales retos de esta nueva generación, donde se han llevado a cabo experimentos en los que se alcanzan velocidades de hasta 1.2Gbps en una conexión donde el móvil receptor se movía a 100km/h [5]. Estos resultados son el fruto de una nueva perspectiva en lo que a gestión

de movilidad se refiere.

En este sentido, y en la línea del objetivo de este proyecto, 5G pretende una gestión inteligente del *handover* entre diferentes celdas, para lo cual el control de la señalización y de la gestión de los usuarios y las rutas se llevará a cabo a través de un controlador externo, que actuará como inteligencia para el resto de elementos de la red. Esta es la filosofía detrás de SDN, permitiendo así que las redes tengan una naturaleza mucho más dinámica.

En el siguiente apartado se debatirá la relevancia que el modelo planteado en SDN tiene en relación a los objetivos marcados en este proyecto.

1.4. Importancia del modelo SDN

Como ya hemos comentado, 5G pretende que los terminales móviles puedan seleccionar de forma inteligente a qué red de acceso inalámbrico van a conectarse en cada momento, incluso durante el desarrollo de una determinada tarea. En definitiva, la visión futura de las redes nos lleva a considerar la actual arquitectura de 4G como poco eficiente, puesto que hablamos de redes muy dinámicas, y la gestión de estos cambios con la arquitectura actual es muy costosa, con una alta carga computacional.

Esto nos lleva a ver la virtualización de redes como la alternativa más viable para sentar las bases de las nuevas lineas de desarrollo que las redes de telecomunicación deben seguir. Las redes definidas por *software*, junto con un *hardware* de red compatible con esta tecnología, permitirán una descentralización de la gestión de los recursos de red, haciendo posible que se compartan recursos de manera eficaz, controlada, y segura para los usuarios y para los sistemas.

Más adelante analizaremos esta tecnología en profundidad, pero en este punto, donde ha quedado patente el carácter cambiante de las redes de comunicaciones futuras, la posibilidad de gestionar todas las conexiones de los equipos que tengan acceso a nuestra red a nivel software nos permite diseñar e implementar sistemas que se encarguen del proceso de señalización, negociación, y establecimiento de las conexiones en un plano completamente aislado del plano de datos.

Esta es la gran ventaja que nos aportan las redes definidas por software, la flexibilidad y versatilidad de separar ambos planos, el de datos y el de control, teniendo así la posibilidad de definir el comportamiento de la red de forma dinámica a nivel software.

Una vez analizados brevemente algunos conceptos fundamentales para el resto del proyecto, vamos a pasar a definir la estructura de los capítulos siguientes. De forma breve y concisa explicaremos el contenido de cada uno

de ellos, sirviendo así de guía para el lector.

1.5. Estructura de la memoria

La memoria de este proyecto consta de un total de 10 capítulos, de acuerdo a lo recogido a continuación. En cada capítulo se tratará alguno de los aspectos fundamentales del desarrollo del proyecto:

1.5.1. Capítulo 1: Introducción

En este primer capítulo se hará una pequeña introducción los principales conceptos que van a formar parte de todo el proyecto. Si bien la mayoría de estos conceptos serán desarrollados en capítulos posteriores, su mención en este capítulo trata de situar al lector en el marco de referencia del proyecto, así como de justificar el uso de dichas tecnologías para nuestro fin.

1.5.2. Capítulo 2: Estado del arte

Este capítulo supone una revisión del resto de modelos para la gestión de movilidad en redes 5G. Aunque la mayor parte de estos modelos son sólo teóricos y no cuentan con una implementación que interaccione con elementos físicos reales como se pretende en este proyecto, nos dará una idea de hacia dónde se dirigen las principales investigaciones en este campo.

1.5.3. Capítulo 3: Planificación y estimación de costes

El capítulo de planificación y estimación de costes servirá para hacer una planificación inicial del tiempo que se invertirá en el trabajo y de todos los elementos que serán necesarios para su realización. De esta forma, podremos realizar una estimación de costes.

1.5.4. Capítulo 4: Requisitos de funcionalidad y decisiones de diseño

En este capítulo se analizarán y expondrán todas las decisiones tomadas tanto para la implementación del proyecto respecto a qué se pretende conseguir, como a las tecnologías usadas para conseguir los objetivos marcados, justificando dichas decisiones.

1.5.5. Capítulo 5: Tecnologías aplicadas

En este apartado continuaremos la linea marcada por el anterior, profundizando en las tecnologías que tienen una mayor implicación en el proyecto, como son SDN y el protocolo OpenFlow. Para cada una de estas tecnologías analizaremos tanto su estructura, como sus componentes y las funciones que desempeñan en nuestro proyecto, de forma que se adquiera una perspectiva global sobre la influencia de cada una de ellas en el desarrollo del mismo.

1.5.6. Capítulo 6: Herramientas utilizadas

De igual forma que en apartado anterior se incide en el análisis de las principales tecnologías que se utilizan en este proyecto, el capítulo seis hará lo mismo con las herramientas utilizadas para trabajar con dichas tecnologías, a saber, Mininet y OpenDayLight. Analizaremos las posibilidades que nos ofrecen estas herramientas y su aplicación para nuestro proyecto.

1.5.7. Capítulo 7: Diseño e implementación de un escenario de movilidad

El séptimo capítulo versará sobre el diseño y la implementación del controlador para gestionar la movilidad en redes 5G. A lo largo de este capítulo explicaremos en detalle las topologías diseñadas en Mininet para poder simular el escenario deseado, así como el comportamiento de las distintas aplicaciones diseñadas para emitir y recibir los mensajes de señalización de acuerdo lo recogido en [2].

1.5.8. Capítulo 8: Fase de pruebas y evaluacion

En este apartado realizaremos distintas pruebas destinadas a evaluar el rendimiento y funcionalidad de las soluciones implementadas en apartados anteriores, intentando caracterizar en la medida de lo posible el comportamiento de nuestro sistema de cara a la emulación del escenario de movilidad buscado.

1.5.9. Capítulo 9: Conclusión y vías futuras

Se realizará aquí una breve valoración sobre el desarrollo general del proyecto, los problemas que han surgido durante el mismo, la forma de solventarlos, y el nivel de satisfacción personal con el resultado obtenido. A continuación se analizarán las posibilidades futuras de este proyecto, cómo puede continuarse su desarrollo, y las formas en que puede ser ampliado.

1.5.10. Capítulo 10: Bibliografía

En el apartado de bibliografía se recogen todos las fuentes de las que se ha utilizado información tanto para la realización práctica como para la parte más teórica de este proyecto.

Capítulo 2

Estado del arte

El objetivo de este capítulo es dotar al lector de una amplia perspectiva sobre los estudios que se están realizando o se han realizado recientemente en el campo de la gestión de movilidad en redes 5G.

En los apartados siguientes se citarán varias publicaciones relevantes sobre la materia de la que es objeto este trabajo de fin de grado. Por tanto, para evitar la repetición de conceptos que ya se han explicado, o que se explicarán en apartados posteriores, nos centraremos en subrayar los puntos fuertes, las diferencias, y los puntos débiles, si los hay, de cada uno de estos estudios con respecto al que nosotros estamos llevando a cabo.

Esta sección finalizará con una breve conclusión sobre las aportaciones originales que este proyecto supone en el campo de la gestión de movilidad en las futuras redes 5G.

2.1. Comparación con otros modelos

Cuando uno comienza a buscar estudios sobre la gestión de movilidad en redes 5G se da cuenta de que existe una gran heterogeneidad en cuanto a los planteamientos que sirven como eje de cada estudio. Por ello, para este capítulo se han seleccionado algunos que reflejan de forma clara dichas diferencias.

2.1.1. Mininet-WiFi: Emulating Software-Defined Wireless Networks

El primero de los trabajos que vamos a analizar en este apartado es el que puede consultarse de forma íntegra en [6]. La idea de este proyecto es la de dotar a cualquiera que quiera realizar algún prototipado y análisis de

rendimiento en escenarios que usan redes inalámbricas, de una herramienta de simulación que permita recrear este tipo de situaciones.

Este proyecto, más que un análisis teórico pormenorizado de escenarios concretos, como el de movilidad, hace un uso extensivo de herramientas como Mininet para la experimentación con escenarios emulados a través de SDN y del protocolo OpenFlow. No se concentra, por tanto, en gestionar movilidad en redes 5G a través de estas herramientas, sino que más bien presenta una gran variedad de escenarios entre los que se encuentra uno en particular, que puede resultar muy interesante cuando se quiere realizar un simulación de movilidad en redes inalámbricas 5G íntegramente con elementos virtualizados.

En contraste con el trabajo que nosotros realizamos, este grupo de investigadores de la Universidad de Campinas basa su trabajo en el concepto de *Station* (STA) Wi-Fi y puntos de acceso AP virtualizados.

Uno de los principales conceptos que se introduce en este estudio es lo que ellos llaman *Software Defined Wireless Networking* (SDWN), que constituye el eje sobre el que gira el resto del proyecto, y que básicamente pretende ser una extensión del paradigma SDN pero adaptado a redes inalámbricas.

En el panorama actual, donde OpenFlow no da el soporte necesario para la utilización de protocolos Wi-Fi en la simulación de entornos de red, e incluso herramientas *open source* como OpenWRT presentan un conjunto muy limitado de recursos, la virtualización de todos los elementos de red supone una alternativa bastante interesante para aquellos que quieren trabajar con múltiples dispositivos, inalámbricos o no.

Sin entrar en detalle, en [6] se describen los elementos que sería necesario añadir a una topología basada en SDN para poder simular una red inalámbrica. Como puede verse en la Figura 2.1, tanto en el AP como en las *Station* (STA) se ha añadido una interfaz Wi-Fi. En capítulos posteriores comentaremos algunas de las posibilidades que se han usado en este trabajo de fin de grado para crear estas interfaces, y los problemas que ello ha conllevado.

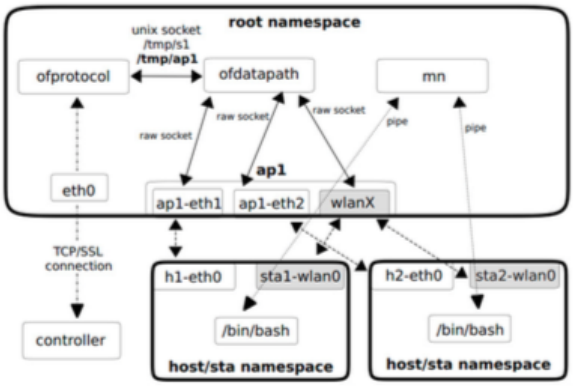


Figura 2.1: Componentes para una topología básica con interfaces Wi-Fi [6].

En concreto, en el caso de simulación de un escenario de movilidad, que al fin y al cabo es nuestro objeto de estudio, [6] propone un escenario básico formado por dos Station (STA) móviles y un AP fijo, cuyos parámetros pueden ser modificados por cualquier usuario que haga uso de las topologias definidas en Python que han creado con el fin de llevar a cabo esta simulación. Éstas pueden descargarse desde su repositorio en GitHub: `https://github.com/intrig-unicamp/mininet-wifi`.

Como vemos, en este modelo no se alcanza a simular un *handover*, considerándolo como el paso de una STA del área de cobertura de un AP al área de otro AP, sino que su pretensión va más encaminada a analizar cómo la distancia entre los puntos de acceso y las estaciones base afecta a la calidad de su enlace y por tanto aumenta las latencias en los *pings* entre ambos elementos de la red móvil.

Como ya hemos comentado, la principal ventaja de la implementación propuesta en [6] es que todo puede gestionarse de forma simulada. Sin embargo, el principal objetivo de este trabajo de fin de grado es la interacción con elementos reales, ya que a la hora de implementar un escenario como el que se presentará en capítulos posteriores, aparecen problemas de configuración y limitaciones adicionales que no pueden ser consideradas en un modelo simulado de estas características, y que provocan un replanteamiento en el diseño de nuestro controlador y nuestra topología de red.

Es por tanto un proyecto que no presenta una base teórica fuerte en cuanto al desarrollo y gestión que la movilidad en 5G conlleva, ya que no se desarrolla un protocolo de comunicación entre punto de acceso y estaciones conectadas a él, ni entre estaciones. Sin embargo, su utilidad como herramienta para el desarrollo de futuros estudios en este campo es innegable.

2.1.2. Handover Management in SDN-based Mobile Networks

El segundo de los estudios que vamos a analizar de entre los que se están desarrollando es este campo es el que aparece publicado en [7]. Tanto este estudio como el siguiente tienen como objetivo el análisis del *handover* tanto en redes existentes como en las futuras redes 5G.

En este estudio se describe de forma muy precisa la principal razón por la que es tan interesante el estudio del *handover* en las futuras redes 5G. Ante el creciente aumento de usuarios y dispositivos en las redes móviles, y el aún mayor crecimiento en la demanda de ancho de banda para el uso de servicios como servicios en la nube, o transmisión de vídeos de alta calidad, se tiende a incrementar la capacidad de la parte más crítica de las redes móviles, la *Radio Access Network* (RAN) a costa de reducir el tamaño de las celdas de cobertura de cada AP.

Esta disminución en el tamaño de las celdas provoca un aumento de carga desde el punto de vista de la movilidad, ya que significa que usuarios moviendose rápidamente pueden requirir de procesos muy rápidos de *handover* en periodos de tiempo muy cortos.

En redes *Long Term Evolution* (LTE), donde el único *handover* implementado es el de tipo *hard* (con filosofía *break-before-make*) conviene reducir al máximo el tiempo en que la conexión entre el UE y la estación base o *evolved Node B* (eNB) está rota.

A diferencia del estudio en el que se basa este nuestro proyecto y que analizaremos a continuación, en [7] se analiza el funcionamiento de la gestión de movilidad en las redes actuales, tanto desde el punto de vista matemático, para la toma de decisión sobre el momento óptimo de realización del cambio de celda, como desde el punto de vista de señalización. Sin embargo, es mucho más interesante para esta memoria el análisis del la gestión de *handover* en redes SDN.

En [7] el controlador es el que se encarga de realizar el reenvio de flujos a partir de parámetros como la dirección de origen o de destino, el puerto origen o destino, o bien el protocolo utilizado. Como veremos más adelante en nuestra propia implementación, estos parámetros son de vital importancia cuando se quieren instalar los flujos que permitan la comunicación entre diferentes elementos de una red formada por varios nodos, todo ello a través del protocolo OpenFlow. El controlador es quien puede modificar dinámicamente estos flujos, a partir de unas tablas que controlan los equipos y puertos que tiene asignados cada *switch*.

En cuanto a las propuestas que se realizan en este estudio sobre cómo puede aplicarse el paradigma SDN a la gestión de movilidad en redes 5G se ofrecen tres variantes:

2.1.2.1. SDN centralizado

En este modelo, toda la gestión de la red se deja a un único controlador lógico, con lo que se reduce el intercambio de información necesaria para realizar el *handover* cuando existen varios controladores y se pueden gestionar los flujos de datos de forma más eficiente, dada la visión global de la red que posee este controlador.

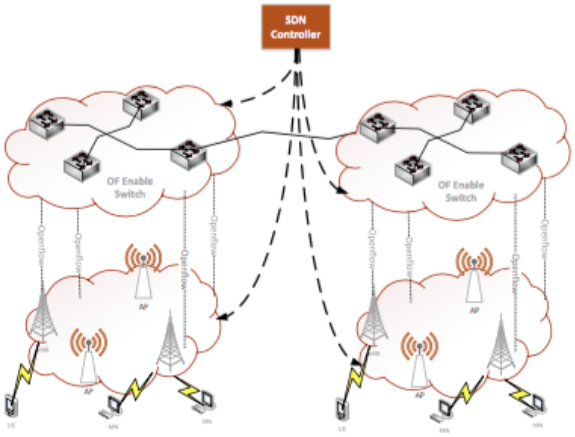


Figura 2.2: Aproximación *Centralized SDN* [7].

Los aspectos negativos de esta variante son la necesidad de enlaces de control, la poca escalabilidad, y la robustez del sistema, que confía todo su plano de control en un solo nodo lógico.

El esquema de esta aproximación es el que puede verse en la Figura 2.2, y es el diseño ideal cuando se tiene una red de pequeñas dimensiones con un número reducido de usuarios, pues permite un control total de forma sencilla del comportamiento de la red.

2.1.2.2. SDN semi-centralizado

Esta aproximación es una solución intermedia para redes de tamaño medio, pues es más robusta que la anterior propuesta, permitiendo dividir en dominios el plano de control de la red. Cada dominio podría considerarse como un subescenario con las características del diseño del apartado anterior.

La principal diferencia con el respecto al modelo anterior es que cada AP debe mandar en difusión la información sobre el dominio del que forma parte, de tal manera que si alguno de los AP recibe un informe de una estación móvil que indica que se necesita un *handover* debe reenviar este mensaje con la información pertinente sobre la sesión del usuario al AP del

dominio de destino.

Como vemos, el nivel de complejidad aumenta a medida que se incrementan el número de controladores, pero es la forma más eficiente de gestionar redes de un tamaño considerable, dotando al sistema de escalabilidad.

El esquema del escenario en este caso es el que puede verse en la Figura 2.3

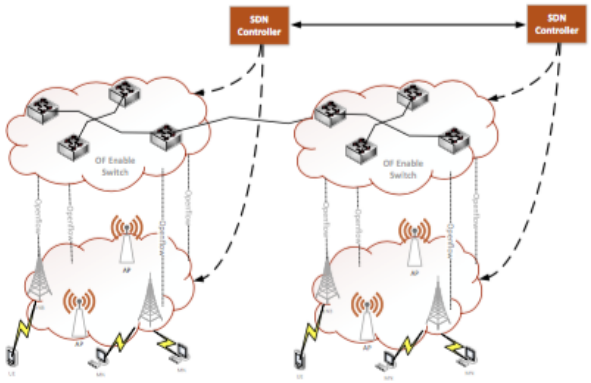


Figura 2.3: Aproximación *Semi-centralized* SDN [7].

2.1.2.3. SDN jerárquica

En esta última propuesta se consigue dotar al plano de control de una visión global del sistema a la hora de tomar decisiones, como sucedía en el caso de SDN centralizada, pero se mantiene la escalabilidad de la propuesta semi-centralizada. Todo ello se consigue a través de dos niveles de controladores: los de la parte inferior, que se encargan de gestionar cada uno de sus dominios, y el de la parte superior que se encarga del intercambio de información para la gestión inter-dominio.

La principal ventaja de este diseño es que se reduce la carga computacional de los controladores de cada dominio, dejando las funciones de la gestión de información relativa a diferentes dominios al controlador maestro, a costa de introducir un nivel de complejidad mayor al de los dos casos anteriores.

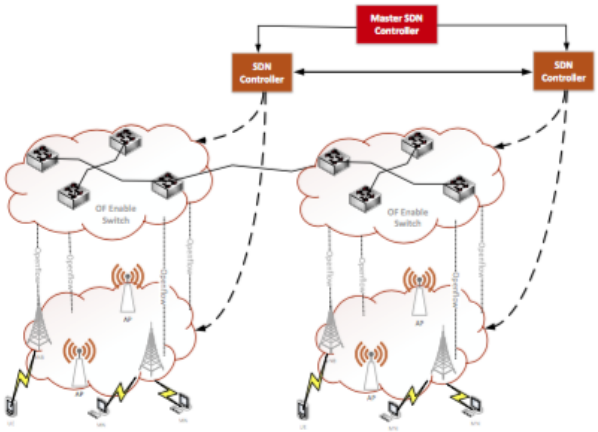


Figura 2.4: Aproximacion *hierarchical* SDN [7].

En este caso, la estructura del escenario sería la mostrada en la Figura 2.4, donde se pueden apreciar los dos niveles de controladores anteriormente mencionados operando sobre un escenario con dos dominios. No obstante, el caso ideal para esta configuración será el de un mayor número de dominios, donde el controlador maestro pueda generar unos mayores niveles de eficiencia.

2.1.3. Link-level Access Cloud Architecture Design Based on SDN for 5G Networks

Este estudio, realizado por miembros de la Universidad de Granada, puede consultarse en [4]. En general, presenta una estructura similar a la de los dos estudios anteriores. Sin embargo, en este caso se entra más en profundidad en el entramado de la arquitecutra de las futuras redes 5G, llegando a proponer un posible diseño basado en la división de la red en tres niveles jerárquicos, como puede verse en la Figura 2.5.

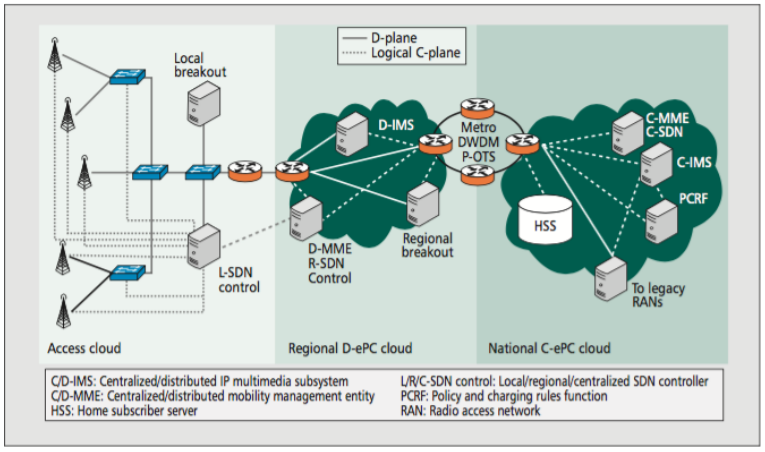


Figura 2.5: Estructura de una red 5G [4].

Como vemos, esta división se realiza implementando la nube de acceso o *Access Cloud* (AC) a partir de APs y *switches* que utilizan protocolo OpenFlow. El tráfico de varios Access Cloud (AC) se agrega a la *Regional Cloud* (RC), que está formada principalmente por Data Center (DC) y por dispositivos que actuan como enrutadores de capa 3, redirigiendo el tráfico de la red de acceso hacia el núcleo, así como de realizar funciones de movilidad.

Finalmente, el National Cloud (NC) se interconectan las diferentes Regional Clouds (RCs). En esta parte de la red es donde se ejecutan las funciones lógicas centralizadas, como la información de subcripción y cargo. En esta arquitectura de red, SDN se encarga de gestionar y orquestar todas las funciones del plano de control.

El detalle de la implementación de la nube de acceso, así como una descripción de todos los elementos que la componen se puede consultar en [4], si el lector así lo desea. Cabe destacar, sin embargo, el apartado en que se describe cómo se podría realizar un *handover* en esta AC, lo que en [4] aparece representado en la Figura 2.6.

Este proceso de *handover* será en el que vamos a basar la implementación de nuestro proyecto, de forma que será explicado en profundidad en apartados posteriores, a la par que se comentarán los detalles de su implementación.

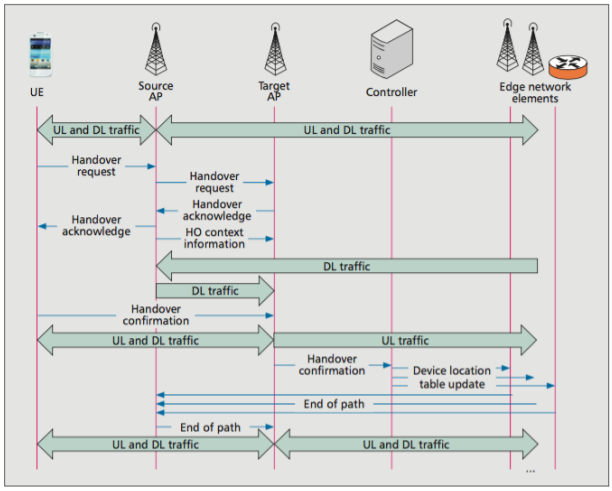


Figura 2.6: *Handover* basado en OpenFlow. Figura tomada de [4].

Concluye de esta forma el apartado de estado del arte, donde hemos tratado de llevar a cabo una amplia visión de los estudios más interesantes que se están llevando a cabo en este campo actualmente.

Como se ha podido comprobar, en su mayoria se tratan de propuestas teóricas, salvo en el caso de [6], donde se lleva a cabo un desarrollo práctico en su totalidad con elementos simulados.

Por estos motivos, este proyecto ofrece una alternativa aún no contemplada, la interacción del entorno simulado con elementos reales, como son las tarjetas de red que usaremos para implementar nuestros AP y el equipo que usaremos como UE y que suponen un nivel de configuración y complejidad de implementación que se apreciará en posteriores apartados.

Capítulo 3

Planificación temporal y presupuesto

En este apartado vamos a realizar una planificación temporal y económica del proyecto. Presentaremos la temporazalización del mismo a través de un diagrama de Gant (Figura 3.1) , basándonos en una estimación a priori del esfuerzo que va a requerir cada tarea, teniendo presente que ésta estimación puede variar en función de los requerimientos del proyecto y de las circunstancias personales del responsable del mismo.

Para la planificación económica haremos un inventario de todos los recursos, clasificándolos según el tipo en materiales (hardware), software y recursos humanos involucrados en el proyecto. Agruparemos todos estos recursos en tablas, en función de su categoría y estimaremos el coste de cada una de las partes, para facilitar así el posterior análisis que se realizará sobre el peso de cada una de estas partes en el coste total del proyecto.

Por último se realizará una estimación de costes, cruzando las estimaciones realizadas en los dos apartados previos, de manera que obtengamos un coste aproximado para la realización del proyecto que nos aporte una visión global sobre el coste de cada uno de los paquetes de trabajo y recursos que forman parte del proyecto.

3.1. Planificación temporal

Un resumen temporal de la planificación estimada para la realización de este proyecto a lo largo de un curso universitario se puede ver en la Figura 3.1. Como se observa, se plantea una planificación temporal con solapamiento entre tareas, puesto que muchas de ellas pueden realizarse simultaneamente, por ejemplo, comenzar la redacción del proyecto mientras

se realiza la fase de pruebas del mismo.

Un desglose más minucioso de cada tarea se presenta en el siguiente subapartado.

3.1.1. División del trabajo

En este apartado vamos a detallar las tareas que conforman cada parte del trabajo, de forma que todas las tareas queden bien definidas y enmarcadas en cada segmento temporal.

- **Búsqueda de información y documentación:** Esta tarea consiste en buscar información relacionada con la arquitectura que se está desarrollando para redes 5G, proyectos similares que se estén llevando a cabo, modelos de señalización de este tipo de procesos de movilidad, etc. También se incluye en esta etapa la búsqueda de información relacionada con las herramientas que posteriormente usaremos para el desarrollo de nuestro proyecto.

- **Familiarización con el entorno:** Esta tarea consiste en instalar un entorno de desarrollo en el que podamos llevar a cabo la mayor parte de las implementaciones necesarias para la construcción y funcionamiento del escenario. Se trata de un proceso costoso, pues requiere del uso de herramientas hasta ahora desconocidas para nosotros, complejas, y que requieren una profunda compresión.

- **Fase de diseño físico y lógico:** Este proyecto contiene una parte importante de diseño lógico, puesto que hay que plantear un escenario en el que se pueda realizar la movilidad entre dos AP de manera sencilla, para lo cual hay que tener en cuenta tanto los dispositivos reales con los que contamos, como la propia escalabilidad del modelo diseñado. Por otra parte, su implementación también requiere una serie de consideraciones que son analizadas durante esta fase.

- **Implementación del escenario:** Junto con la fase de familiarización con el entorno, esta es la fase que más tiempo se espera que ocupe, puesto que hay que trabajar con distintas tecnologías, programar tanto en JAVA como en Python, y hacer frente a los cambios de resideño que vayan surgiendo, así como a las soluciones de contingencia que sea necesario aplicar.

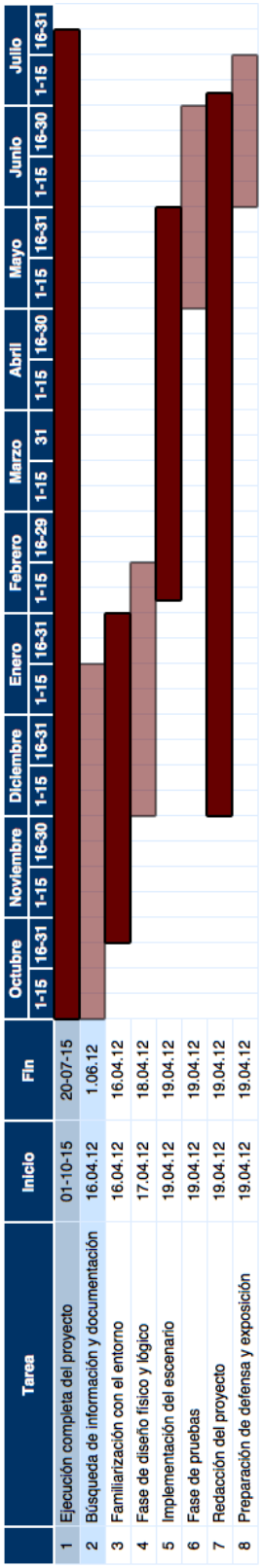
- **Fase de pruebas:** En esta fase se espera poder utilizar el escenario implementado y la inteligencia programada del controlador para establecer una sesión entre el UE y algún host interno de la red en la que

se descargue contenido y donde se pueda medir la pérdida de paquetes y se compruebe el mantenimiento de dicha sesión durante el proceso de movilidad.

- **Redacción del proyecto:** Al hilo de la implementación del proyecto y de su fase de pruebas, se llevará a cabo la fase de redacción del mismo. En este punto es importante remarcar que muchas de las partes que deben ser expuestas en un proyecto de esta magnitud pueden redactarse antes y durante su desarrollo, pues no variarán en función del desarrollo del mismo. Por tanto, el orden lógico para la redacción de esta memoria será comenzar por la introducción, objetivos y fundamento teórico, dejando para el final las partes de implementación, diseño lógico y físico y pruebas.

- **Preparación de defensa y exposición:** Finalmente, tras la concluir la redacción del proyecto, se procederá a preparar tanto la exposición para la defensa ante el tribunal, como la defensa que se realizará del mismo. Tras todo el trabajo realizado, esta fase debe reducirse a un proceso de síntesis y ordenación de ideas para exponer la mayor cantidad de información relevante posible, de forma clara y ordenada.

Una división del trabajo, en función del tiempo empleado en cada parte del mismo, sería la que puede verse en las tablas 3.1 y 3.2. Como vemos, las tareas más costosas son la redacción del proyecto y su implementación, ambas sujetas a muchos cambios y revisiones durante su desarrollo.



	Tarea	Inicio	Fin
1	Ejecución completa del proyecto	01-10-15	20-07-15
2	Búsqueda de información y documentación	16.04.12	1.06.12
3	Familiarización con el entorno	16.04.12	16.04.12
4	Fase de diseño físico y lógico	17.04.12	18.04.12
5	Implementación del escenario	19.04.12	19.04.12
6	Fase de pruebas	19.04.12	19.04.12
7	Redacción del proyecto	19.04.12	19.04.12
8	Preparación de defensa y exposición	19.04.12	19.04.12

Figura 3.1: Planificación temporal del proyecto. Figura realizada con draw.io

Tarea	**Duración (horas)**
Búsqueda de información y documentación	50
Familiarización con el entorno	70
Diseño físico y lógico	40
Implementación del escenario	100
Fase de pruebas	40
Redacción del proyecto	100
Preparación de exposición y defensa	20
Total	**430**

Cuadro 3.1: Tiempo empleado en la realización del proyecto.

Tutorización Jorge Navarro	
Tarea	**Duración (horas)**
Búsqueda de información y documentación	2
Familiarización con el entorno	4
Diseño físico y lógico	4
Implementación del escenario	6
Fase de pruebas	4
Redacción del proyecto	6
Preparación de exposición y defensa	2
Total	**28**

Cuadro 3.2: Tiempo de tutorización empleado en la realización del proyecto.

3.1.2. Planificación económica

En este apartado vamos a listar todos los recursos empleados en el proyecto. Para ello, vamos a distinguir entre recursos materiales, software y recursos humanos, y utilizaremos la estimación temporal realizada en los apartados anteriores para finalmente obtener un presupuesto aproximado al coste real del proyecto.

3.1.2.1. Recursos humanos

Este proyecto va a contar con la tutorización de D. Jorge Navarro Ortiz, en calidad de Profesor Contratado Doctor de la Universidad de Granada, en el Departamento de Teoría de la señal, Telemática y Comunicaciones. El valor del trabajo de tutorización realizado durante este proyecto se estimará en torno a 50 euros por cada hora.

Por otro lado, la mano de obra de un alumno del Grado en Ingeniería de Tecnologías de las Telecomunicación se estimará en 25 euros por cada hora

trabajada.

3.1.2.2. Recursos *hardware*

En la realización de este proyecto contaremos con los siguientes elementos, fundamentales tanto para la implementación del entorno como para la realización de las pruebas relativas al correcto funcionamiento de la lógica del controlador y de la topología:

- **Equipo de trabajo:** MacBook Pro 2010 con procesador Intel Core i7 de 2,8 GHz. Usado para la creación de las topologías básicas sobre las que se ejecuta el controlador, así como para la programación de la aplicación java que permite la simulación del *handover* y la utilización del entorno apropiado para las pruebas con el controlador.
- **Equipo para emulación:** Toshiba Satellite A660 13-q con procesador Intel Core i5. Usado para actuar como parte cliente en el proceso de *handover*, de tal forma que, a través de una máquina virtual con Ubuntu 16.04 LTS se ejecuten los scripts necesarios para la configuración de su interfaz inalámbrica y la aplicación JAVA necesaria para la señalización del UE.
- **Conexión a Internet:** Línea de acceso a internet de 50Mbps, necesaria para la realización de las pruebas y la medición de pérdida de paquetes durante el proceso de cambio de área de cobertura.
- **Adaptador Wi-Fi inalámbrico:** Tres adaptadores Wi-Fi USB inalámbricos Tenda N Dual de 300Mbps usados para crear dos puntos de acceso para que un dispositivo que actúe como UE dotado de otro adaptador pueda pasar de uno a otro y simular el cambio de celda.
- ***Hub* de puertos USB :** *Hub* para aumentar la capacidad de puertos USB del equipo destinado a la emulación del escenario, dado que se necesitan conectar varios adaptadores USB para la realización de las pruebas y el modelo usado sólo dispone de 2 puertos por defecto.

3.1.2.3. Recursos *software*

El *software* que va a formar parte de este proyecto va a ser en su totalidad *software* libre, lo cuál supone un importante ahorro en costes para la realización del mismo. Esta parte propicia también que el peso del coste económico del proyecto recaiga sobre todo en el componente humano, y en menor medida en el *hardware*.

Un breve listado de los elementos *software* que forman parte del proyecto sería:

- **Sistema operativo: Linux Ubuntu 14.04 de 64 bits :** Usando una máquina virtual para llevar a cabo todas las pruebas con el controlador y la topología definida.
- **IDE: Eclipse Neon :** Usado para la programación de la aplicación JAVA que permite realizar el salto de una celda de cobertura a otra y que conecta con el controlador para notificar ese cambio.
- **Lenguaje de programación principal JAVA :** Usado tanto para la implementación de la aplicación que simula el *handover* como para la programación de la inteligencia de nuestro controlador.
- **Lenguaje de programación secundario Python :** Usado para la implementación de las topologías que van a utilizarse para realizar las pruebas de redirección de flujo de datos cuando se produce un cambio de AP.
- **Controlador OpenDayLight :** Controlador elegido para gestionar el tráfico de flujos de información entre los distintos nodos que formarán parte de nuestra topología.
- **Mininet :** Herramienta de emulación de redes sencilla que nos será de gran utilidad para la emulación de nuestra topología.
- **Apache Maven :** *Framework* usado para la gestión de dependencias en proyectos de gran tamaño.
- **Wireshark :** *Sniffer* que se usará para la lectura de paquetes intercambiados por los distintos AP que formarán parte de nuestra red.

3.2. Estimación de costes

Con toda la información recopilada en apartados anteriores, podemos realizar una estimación económica del proyecto, analizando además cuales serán las partes que supondrán un mayor coste y proponiendo alternativas que permitan mejorar este resultado de cara a una posible implementación real.

3.2.1. Coste en recursos humanos

De acuerdo con las tablas 3.1 y 3.2, y dado que hemos estimado el trabajo realizado tanto por D. Jorge Navarro Ortiz en sus funciones de tutor del proyecto en 50 euros por hora, junto con los 25 euros por hora en los que se valora el trabajo realizado por un estudiante del grado de Ingeniería en

Tecnologías de las Telecomunicaciones, podemos resumir el total de gastos humanos en la tabla 3.3

Tarea	**Coste (Euros)**
Búsqueda de información y documentación	1350
Familiarización con el entorno	1950
Diseño físico y lógico	1200
Implementación del escenario	2800
Fase de pruebas	1200
Redacción del proyecto	2800
Preparación de exposición y defensa	600
Total	**11900**

Cuadro 3.3: Coste de la mano de obra para la realización del proyecto.

3.2.2. Coste en recursos *hardware*

En este apartado vamos a listar el precio de cada uno de los componentes comentados en el apartado de Recursos Hardware. El coste del ordenador será proporcional al tiempo usado con respecto al total de su ciclo de vida estimado. El resto de elementos sumarán su coste íntegro al presupuesto del proyecto.

Elemeto	**Unidades**	**Coste (Euros)**
MacBook Pro 2010	1	322.92
Toshiba Satellite	1	200
Conexión a internet	1	300
Adaptador WiFi USB	3	30
Hub USB	1	5
Total		**847.92**

Cuadro 3.4: Coste de los componentes hardware que forman parte del proyecto.

Notaremos que el coste de cada ordenador empleado se ha devengado a razón del periodo de tiempo en el cuál se plantea su utilización, para no desvirtuar el coste total del proyecto incluyendo el coste de dos equipos cuya única finalidad no es la realización de este proyecto.

3.2.3. Coste en recursos *software*

Como ya hemos comentado en apartados anteriores, todo el *software* utilizado para este proyecto es *open source*, y como tal, no supone coste

alguno para la realización del proyecto.

Uno de los principales motivos por los que se elige OpenDayLight como controlador, Mininet como herramienta de simulación de redes o Ubuntu como sistema operativo, aparte de porque son ampliamente explotados y constantemente actualizados, es su coste, que supone un factor diferencial con respecto a otros sistemas operativos o herramientas de características similares.

Ello, unido a la consideración de que parte de los equipos usados para este proyecto (i.e. ordenadores portatiles, o adaptadores WiFi USB) tienen un ciclo de vida superior a la duración del mismo, y que por tanto nos permite dividir el coste de los equipos entre su ciclo de vida total, resulta en un ajuste del coste total del proyecto que se acerca mucho más al coste real del mismo.

3.3. Análisis final de planificación y costes

Con toda la información desglosada en apartados anteriores reunida en este capítulo, podemos presentar gráficamente los datos obtenidos y exponerlos en este último apartado para, de un vistazo, resumir los aspectos más importantes de la planificación temporal y económica del proyecto.

En la Figura 3.2 vemos el porcentaje de tiempo total previsto para cada uno de los paquetes de trabajo definidos en el apartado 4.1.1. En este caso, el 48 % del tiempo total previsto para la realización del proyecto se destina a dos tareas fundamentales, la implementación del proyecto y su redacción.

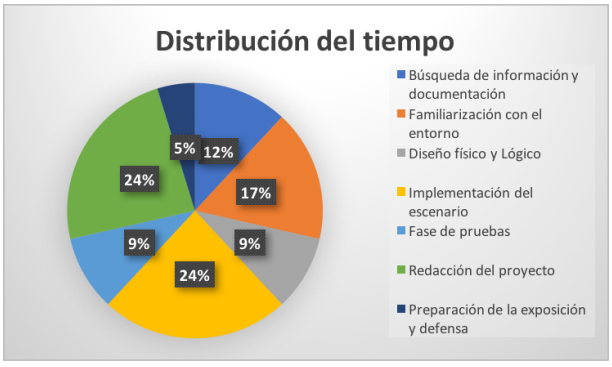


Figura 3.2: Distribución del tiempo de proyecto.

Por otro lado, en la Figura 3.3 podemos observar cómo, al contrario que en el caso anterior donde la distribución temporal está más repartida entre las distintas divisiones temporales, en este caso la práctica totalidad de los costes son debidos a la mano de obra, categorizada como recursos humanos.

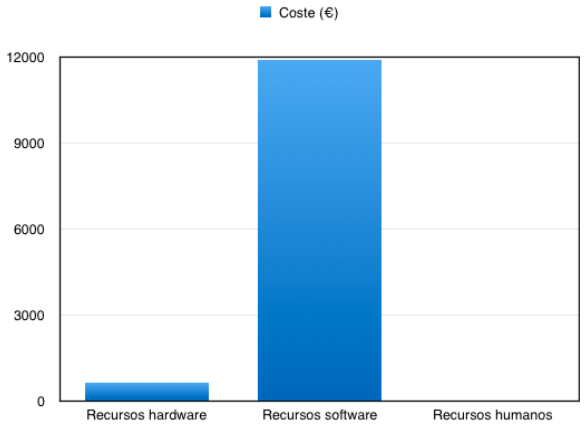


Figura 3.3: Distribución de costes del proyecto.

Como es lo lógico, en un caso práctico, el tiempo real de realización del proyecto sería menor, puesto que tanto la redacción, como las fases de diseño y de familiarización deben realizarse sólo la primera vez que se lleva a cabo este proyecto, con lo cuál los costes quedarían reducidos notablemente.

Capítulo 4

Requisitos de funcionalidad y decisiones de diseño

El siguiente capítulo va a constar de dos subsecciones: requisitos de funcionalidad y decisiones de diseño del proyecto. Ambos apartados están muy marcados por las limitaciones técnicas y las decisiones tomadas a nivel personal orientadas a solventar algún obstáculo técnico y a facilitar al máximo la ya difícil tarea de trabajar con herramientas hasta ahora desconocidas y con una tecnología en ciernes como es SDN.

Para plantear correctamente los siguientes apartados es importante diferenciar lo que se considerará en adelante como requisito de funcionalidad y lo que se considerará decisión de diseño:

- **Requisitos de funcionalidad :** Los requisitos de funcionalidad serán aquellos que vienen dados por la naturaleza del proyecto a realizar y sobre los cuales se va a plantear la solución que finalmente será desarrollada para conseguir los objetivos marcados. Un ejemplo de requisitos de funcionalidad aplicado al proyecto que nos ocupa sería la implementación del intercambio de señalización expuesto en [4], puesto que es una decisión que afecta al comportamiento de nuestro sistema.
- **Decisiones de diseño:** Las decisiones de diseño serán aquellas que nosotros mismos marcaremos antes o durante el desarrollo del proyecto. El uso de determinadas herramientas o lenguajes de desarollo, la topología implementada, etc, son ejemplos claros de decisiones de diseño, pues podríamos haber elegido cualquier otro sistema operativo o lenguaje, pero elegimos entre todos ellos uno en particular por razones ajenas a los aspectos puramente técnicos y funcionales del proyecto.

Una vez aclarado este punto podemos pasar a desglosar los subapartados

que forman parte de este capítulo. Comenzaremos por los requisitos de funcionalidad para acabar con las decisiones de diseño, sobre las que tenemos una mayor libertad de elección.

4.1. Requisitos de funcionalidad

Como hemos comentado al principio de esta sección, en este apartado nos vamos a centrar en lor requisitos de funcionalidad, que son aquellos que vienen marcados por la propia naturaleza del proyecto y por el comportamiento que queremos que nuestro sistema tenga de acuerdo a los recursos de los que disponemos.

Cuando se trata de un proyecto de estas características, a priori las posibilidades son infinitas, pero a la hora de ponerlo en prática, el *hardware* a nuestro alcance, el coste del proyecto y el tiempo que su implementación requiere son los factores que van a limitar de forma unilateral el planteamiento y posterior desarrollo de la solución final.

Sobre la premisa de reducir costes es lógico pensar que todo el software elegido para utilizar como herramientas que participen en la solución final será de licencia libre, ello no conduce a una solución única, puesto que el abanico de posibilidades dentro del *software* libre es muy amplio, lo cual nos permitirá aplicar otros criterios a la hora de realizar una elección. Estos criterios serán expuestos en el apartado siguiente, puesto que por tanto serán una decisión de diseño.

Las limitaciones *hardware* sí suponen un punto importante en este apartado, puesto que, como ya hemos comentado, son la piedra de toque sobre la que plantearemos el resto de soluciones de diseño.

Como es obvio, plantear una implementación real de este escenario queda fuera del objetivo de este proyecto, puesto que no contamos con todo el *hardware* necesario para su implementación, y la complejidad en la configuración del mismo hace que su realización quede fuera del alcance de un proyecto de la magnitud del que se desea realizar.

De igual manera, en pro de la sencillez, y teniendo en cuenta la libertad para configurar puntos de acceso, y no sólo eso, sino la compatibilidad con la mayoría de herramientas de *software* libre que se desarrollan, y dado que a lo largo de todos nuestros años de carrera, en las asignaturas dedicadas al estudio de redes de comunicaciones Linux ha sido el sistema operativo más ampliamente usado, su uso nos supondrá una mayor versatilidad y mejor capacidad de desarrollo de nuestro escenario, y por tanto, será el sistema operativo sobre el que desarrollemos todo lo referente a la simulación de nuestro escenario de movilidad.

Para la implementación del proceso de señalización de acuerdo a lo planteado en [4], usaremos JAVA, que será el mismo lenguaje utilizado para programar el controlador con el *framework* elegido y que además permite ejecutar dichas apliaciones en una gran variedad de sistemas operativos y es un lenguaje ampliamente usado para la comunicación a través del protocolo *User Datagram Protocol* (UDP) que se usará para el intercambio de mensajes entre *switches* y equipos. Su depuración es sencilla y nos permite observar el comportamiento del sistema en cualquier punto del mismo, resolviendo así todos los problemas que surjan durante el proceso de implementación, ahorrando tiempo y esfuerzo y no requiere de la instalación de *software* adicional, pues todos nuestros sistemas cuentan con el entorno necesario para la ejecución de dichas aplicaciones.

Estas decisiones están muy ligadas a las tomadas en el apartado siguiente, de requisitos no funcionales, puesto que en gran medida, un rastreo de todo el panorama de simulación y virtualización de redes nos conduce de manera inequivoca a usar herramientas desarrolladas en base UNIX, y por tanto, su elección es prácticamente obligatoria a la hora de trabajar con determinados controladores y herramientas de silumación de redes.

4.1.1. Decisiones de diseño

Dentro del bloque de decisiones de diseño vamos a englobar todo el resto de decisiones previas a la implementación del desarrollo que finalmente hemos llevado a cabo y que han dado sentido al comportamiento que se platea cuando se piensa en un escenario de movilidad.

Este apartado afecta en mayor medida a la parte *software* y de diseño del proyecto, pues como ya hemos comentado con anterioridad, las limitaciones *hardware* venían autoimpuestas por razones ajenas a criterios subjetivos de las personas que realizan este proyecto.

Al tratarse de un proyecto que consta de varias partes diferenciadas, muchas han sido las decisiones que se han tomado previas a su desarrollo, algunas de ellas basadas en decisiones técnicas, pero otras también por razones personales, puesto que la propia experiencia de los profesores del departamento, en este caso D. Jorge Navarro Ortiz, o la mía propia, hacen que la elección de algunas herramientas sea más favorable a la hora de agilizar su aprendizaje y manejo.

Los puntos principales sobre los que tomar decisiones de diseño y desarrollo son el controlador con el que trabajar, la herramienta con la que simular nuestra topología y la configuración de los puntos de acceso.

No se trata en este punto de analizar en detalle cada uno de los puntos citados en el párrafo anterior, sino más bien explicar el por qué de la elección

de cada uno de ellos. Teniendo esto en cuenta, vamos a comentar cada una de las elecciones:

- ***Framework* para el Controlador :** Para comenzar, una de las decisiones más importantes, pues marcará la programación de la inteligencia de la red, y el lenguaje en el que vamos a trabajar durante la mayor parte del proyecto será la de framework que se utilice para implementar el controlador. Existe una web (http://sdnhub.org/) donde podemos ver varias posibilidades para elegir un framework que se adecúe a nuestras necesidades. De entre todos ellos elegiremos OpenDayLight, puesto que es un controlador ampliamente utilizado, con soporte constante, y que ya ha sido utilizado por profesores del departamento de Teoría de la Señal, Telemática y Comunicaciones con anterioridad. Además funciona sobre JAVA, que es un lenguaje ya dominado, lo que supone una ventaja frente a otros controladores como POX, que funcionan sobre Python.

- **Herramienta de simulación de redes :** Muy ligado al controlador elegido, existe una imagen de linux con ubuntu 14.04 que integra tanto varios ejemplos realizados para un controlador basado en OpenDayLight como una herramienta de simulación de redes ligera y versátil que permite la implementación de topologías propias creadas mediante sencillos scripts en Python y cuyos *switches* son compatibles con el protocolo OpenFlow. Esta herramienta, Mininet, parece imponerse a otras alternativas comunes y muy conocidas como son Packet Tracer de Cisco y GNS3, que además o bien sólo tienen parte de su *software* gratuito, en versiones de prueba o para estudiante, o bien no son estables e integrables con los protocolos que queremos usar para nuestra simulación.

- **Configuración de puntos de acceso:** La conexión a la red a la que van a conectarse los UE puede implementarse de dos formas distintas, como una red descentralizada ad-hoc o como un punto de acceso tradicional usando herramientas como iwconfig. En este caso, no existe un criterio más válido o lógico que otro, ambas soluciones son válidas, sin embargo, por meros motivos de facilidad de configuración, y tras realizar varias pruebas con ambas configuraciones, se ha decidido que configurar cada tarjeta inalámbrica como un punto de acceso tradicional da un resultado más estable, por lo que a razón del resultado, será éste el método que usemos para implementar nuestras interfaces inalámbricas.

Capítulo 5

Tecnologias aplicadas

Una vez analizados los requisitos de nuestro proyecto, dedicaremos los capítulos quinto y sexto al análisis y comprensión de las principales tecnologías y herramientas implicadas en el desarrollo del mismo.

En concreto, en este capítulo que nos ocupa vamos a analizar dos tecnologías que van muy de la mano con las dos herramientas analizadas en el siguiente, pues ambas herramientas son las más populares y utilizadas en ambos campos, a saber: las redes definidas por *software* SDN con Mininet, y el protocolo OpenFlow con el *framework* OpenDayLight.

Vamos a empezar haciendo un pequeño análisis de cuáles son los fundamentos de las redes definidas por software.

5.1. Redes Definidas por *Software*

Las redes definidas por *software* tienen como objetivo aprovechar las mejoras en tecnologías basadas en la nube para aplicarlas al ámbito de las redes de telecomunicaciones, consiguiendo una respuesta rápida y eficaz a los requirimientos cada vez más cambiantes del propio negocio de los sistemas de comunicación.

SDN permite aunar un gran número de tecnologías de red diseñadas para hacer la red más flexible y ágil, y las adapta para que soporten la infraestructura de virtualización y almacenamiento de los data centers actuales. El resultado último de la filosofía de las redes definidas por software es la capacidad de definir, implementar y gestionar de una forma desacoplada con respecto al plano de datos y por consiguiente, favoreciendo el desarrollo de una infraestructura más simple, con un plano de control más versátil y programable.

5.1.1. Arquitectura SDN

En función del proveedor, existen muchas y muy diversas arquitecturas de redes definidas por *software* compitiendo entre sí. Sin embargo, todas tienen en común unos conceptos y unos elementos básicos. Sobre todos ellos prevalece la idea de extraer y separar la lógica del controlador de los dispositivos *Software* de red.

En general, cualquier red basada en *software* cuenta con los siguientes elementos básicos:

- **Controlador :** Es el cerebro de la red, tiene una visión completa de toda la red y permite a los administradores de la misma indicar a los sistemas de control cómo se maneja el flujo de datos de la red.

- **API *SouthBound* :** Las *SouthBound* APIs son las encargadas de enviar información a los nodos de la red, generalmente switches y routers.

- **API *NorthBound* :** Las *NorthBound* APIs son el enlace del controlador con la aplicación que contiene toda la inteligencia de negocio. En ella, un cliente puede dar forma al tráfico de la red que quiere gestionar.

La Figura 5.1 muestra la visión general de cómo la arquitectura de una red definida por software debe ser según la *Open Networking Foundation* (ONF).

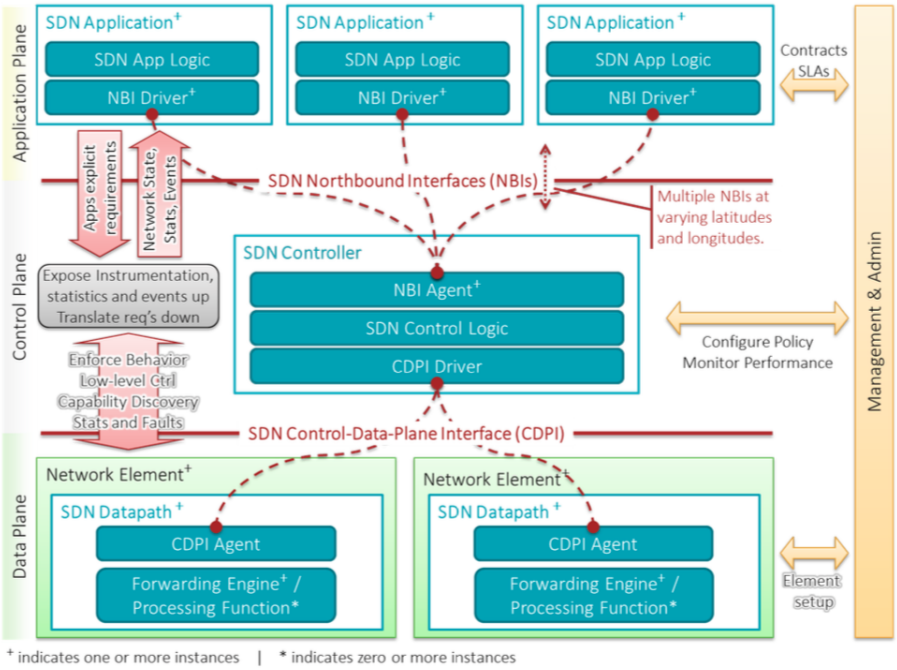


Figura 5.1: Visión general de la arquitectura SDN según la ONF.

Como se puede comprobar, la estructura comentada anteriormente es la base de la arquitectura, a la que se han añadido algunos componentes para acercarse más a lo que en realidad sería una arquitectura SDN completa. Estos elementos son:

- **SDN *Application* :** Una aplicación SDN es un programa que se comunica explícitamente con el controlador para indicar el comportamiento que desea que tenga la red y los requerimientos que necesita. Además de esto, para poder gestionar mejor el flujo de información en la red, la aplicación puede recibir una visión abstracta de la red proporcionada por el controlador. Las aplicaciones están formadas por dos elementos fundamentales, un bloque de lógica (representado por el SDN *Application Logic*) y uno o más *NorthBound Inferface Drivers* (NBI Driver).

- **SDN *Datapath* :** Se trata de un dispositivo lógico de red que se encarga de dar visibilidad y control a través de sus capacidades de reenvío y procesado de datos. Su representación lógica puede abarcar todos o una parte de los recursos físicos de la red. Los SDN *Datapath* están compuestos por un *Control Data Plane Interface* (CDPI), un conjunto de uno o más motores de reenvío de tráfico y opcionalmente funciones de procesado de tráfico. Un dispositivo físico de red puede

contener varios *Datapaths*, de igual modo que un sólo *Datapath* puede estar definido a lo largo de muchos dispositivos físicos.

- **SDN *Control Data-Plane Interface* :** Se trata de una interfaz definida entre entre el controlador y un *Datapath* que se encarga de controlar de forma programática el reenvío de operaciones, capacidad para notificar eventos y reporte de estadísticas de red. Uno de los puntos más importantes de SDN es que los *Control Data Plane Interface* (CDPI) se implementan de forma abierta, son multiproveedor y altamente interoperables.

- ***Interface Drivers & Agents* :** Cada interfaz es implementada por un par *driver-agent* en el que el agente representa la parte que se comunica con los dispositivos de red, la *SouthBound* y el driver representa la parte de aplicación, la *NorthBound*.

- ***Management & Administration* :** El plano de gestión (*management*) se encarga de controlar las estadísticas de red, que se gestionan de forma más eficiente fuera del plano de la aplicación. Algunas de las tareas que se monitorizan en esta parte son la gestión de la relación entre la parte proveedora y la parte cliente, la asignación de recursos al cliente, la puesta en marcha de equipos, coordinación de la alcanzabilidad de los dispositivos y sus credenciales entre dispositivos lógicos y físicos, etc. Cada entidad de negocio tiene sus propias entidades de gestión.

5.1.2. Beneficios del uso de SDN

Uno de los principales beneficios que supone las redes definidas por *software* sobre las redes tradicionales es que las aplicaciones pueden tener conciencia de la red, al contrario que las aplicaciones tradicionales (que sólo pueden describir sus requerimientos de forma explícita y directa a través de varios procesos que carecen de mecanización) y que las redes tradicionales (que no ofrecen una manera dinámica de expresar el amplio rango de requerimientos de una aplicación y que no exponen su información y el estado de la red a las aplicaciones que la están usando).

Por otra parte, la ya comentada centralización del plano de control y su separación del plano de datos permite que el controlador de una red SDN resuma el estado de la red para una aplicación y traduzca los requerimientos de dicha aplicación a reglas de bajo nivel.

Estos dos factores ofrecen una serie de beneficios que se pueden resumir en los siguientes puntos:

- **Programación directa :** El hecho de que el plano de control esté

separado del plano de datos permite que la red sea configurada programaticamente por herramientas de automatización como OpenStack, Chef, o Puppet.

- **Reducción del *Capital Expenditure* (CapEx) :** Las redes definidas por software permiten sustituir el equipamiento de red diseñado para un propósito particular y sustuirlo por hardware estandarizado, pudiendo adoptar se esa forma un modelo escalable de crecimiento en el que se puedan contratar más recursos en la nube en la medida en que los requerimientos de nuestra red van creciendo.

- **Reducción del *Operating Expenditure* (OpEx) :** Es posible desarrollar algoritmos para controlar la red y todos los elementos que la componen, facilitando el diseño, desarrollo, gestión y escalado de redes. La capacidad de automatizar el aprovisionamiento y la orquestación de los recursos de una red incrementa su disponibilidad y confiabilidad reduciendo el tiempo de gestión y eliminando el factor de error humano.

- **Distribución flexible y ágil :** El modelo de redes definidas por software favorece la organización y la rápida implementación de aplicaciones, servicios e infraestructuras.

- **Abierto a innovación :** De igual manera que se favorece la distribución, también se impulsa la creación de nuevos tipos de aplicaciones, servicios y modelos de negocio que generen flujos de ingresos y aporten valor a la red.

5.2. Protocolo OpenFlow

De igual manera que hemos analizado la arquitectura de las redes definidas por *software*, vamos a echar ahora un vistazo al protocolo OpenFlow. Ambas tecnologías suelen ir de la mano, pues OpenFlow es uno de los primeros estándares *de facto* definidos para SDN. Su objetivo principal era el de permitir la comunicación directa entre el plano de datos y el controlador, de manera que un sistema pudiera adecuarse mejor a los requirimientos cambiantes del negocio.

La arquitectura general de OpenFlow, como se puede ver en la Figura 5.2 es bastante sencilla. Un switch, o cualquier dispositivo de red que abstraiga su plano de control a través de OpenFlow estará logicamente compuesto por una serie de tablas de flujo, una tabla grupal, y un canal con el que se comunicará con el controlador.

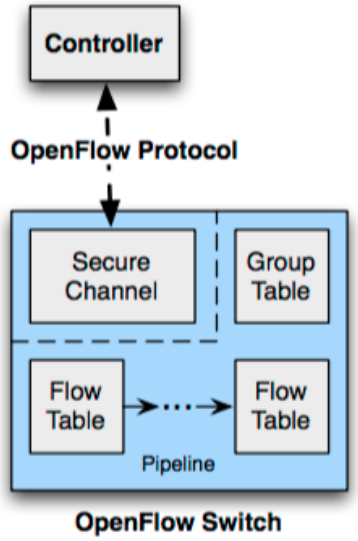


Figura 5.2: Arquitectura OpenFlow. Imagen obtenida de [8]

El principal objetivo de las tablas de flujo y la tabla grupal es el de realizar la búsqueda y el reenvío de paquetes. Cada tabla de flujo en un switch contiene un conjunto de entradas de flujo, que a su vez están compuestas por una serie de campos, contadores e instrucciones que se aplicarán a los paquetes que coincidan con las condiciones aplicadas a esa entrada de flujo.

Es posible que en un switch haya más de una tabla de flujo, en ese caso, se establece un orden de prioridad de tablas, de forma que un paquete irá comprobando si existe una coincidencia en alguna de las entradas de una tabla de flujo, de la más a la menos prioritaria, de manera que cuando se encuentre coincidencia se ejecutará la acción que se indique en esa entrada de la tabla de flujos. Si se ha pasado por una tabla de flujo y no se ha encontrado ninguna coincidencia, el destino de dicho paquete dependerá de la configuración del switch, pudiendo ser enviado al controlador, borrado, o avanzar a las siguientes tablas de flujo.

Las acciones que se pueden realizar sobre un paquete describirán hacia dónde se debe reenviar el paquete, alguna posible modificación sobre el paquete, procesamiento sobre la tabla grupal, y procesamiento de *pipeline*, que permite a los paquetes ser enviados a las siguientes tablas de flujo para un proceso de búsqueda de coincidencias más amplio.

En general, las entradas de flujo de una tabla de flujos redirigirán la salida a un puerto. Este puerto generalmente es un puerto físico, pero también puede ser un puerto virtual definido en el switch o reservado. Los puertos reservados pueden describir acciones específicas, como por ejemplo, el reenvío del paquete al controlador, inundación, o reenvío del paquete me-

diante un protocolo distinto. De igual forma, los puertos virtuales definidos por el switch pueden especificar grupos de agregación de enlaces, túneles o interfaces de *loopback*.

En los subapartados siguientes analizaremos los dos componentes principales de la arquitectura de OpenFlow, las tablas de flujos, y el canal, para comprender mejor cómo funciona OpenFlow a nivel de gestión del encaminamiento de los paquetes en una red, que es la parte que nos concierne en este proyecto. En [9] se puede consultar en mayor profundidad el funcionamiento de los elementos que forman parte del protocolo OpenFlow.

5.2.1. Tablas OpenFlow

Las tablas de flujo, como hemos comentado en la introducción a este apartado están compuestas sencillamente por entradas de flujo, que suelen tener la siguiente estructura:

Match Fields	Counters	Instructions

Cuadro 5.1: Estructura de una entrada en una tabla de flujo.

Como vemos, constan de tres componentes fundamentales, los *Match Fields* son los campos a partir de los cuales se van a buscar coincidencias en las tablas de flujo. Suelen ser tanto las cabeceras de los paquetes como los puertos de entrada a un switch o las direcciones *Media Access Control* (MAC) de destino de un paquete. Los contadores se actualizan cuando se ha buscado coincidencias en los paquetes en una entrada de una tabla de flujo y las acciones indican los procedimientos que se han de aplicar a los paquetes una vez se ha encontrado una coincidencia en las tablas.

Los switches que trabajan con OpenFlow pueden ser de dos tipos, híbridos o sólo OpenFlow, es lo que se conoce como *OpenFlow-Hybrid* y *OpenFlow-Only*. Los switches que son *OpenFlow-Only* sólo soportan operaciones de tipo OpenFlow y todos los paquetes se procesan por el OpenFlow *pipeline*.

5.2.1.1. *Pipeline processing*

El procesado de un paquete a través de un switch OpenFlow involucra a muchas tablas de flujo, cada una de las cuales contiene muchas entradas de flujo que definen cómo los paquetes interactúan con las tablas de flujo. Para hacernos una idea de cómo funciona este proceso, la Figura 5.3 resume el proceso de pipeline de un paquete.

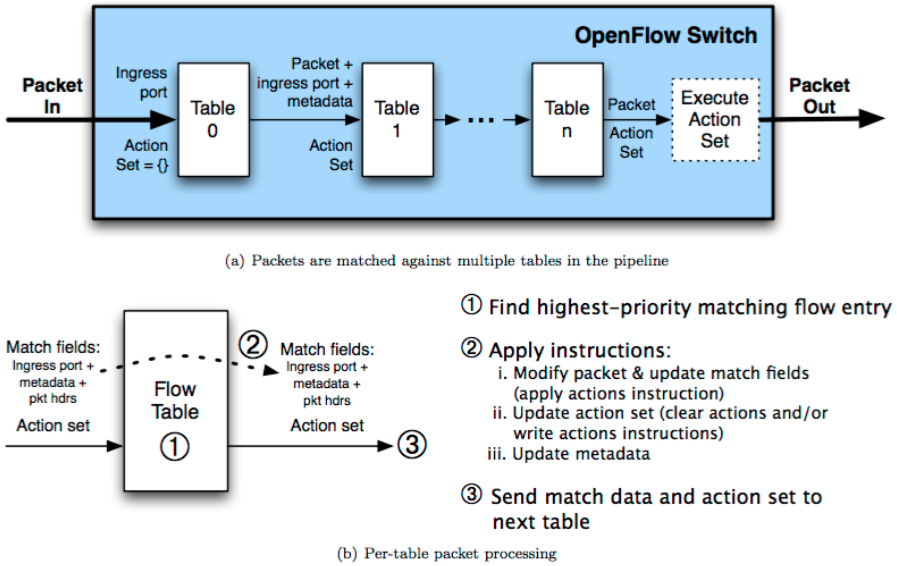


Figura 5.3: Flujo de un paquete a través del *Processing Pipeline* [9].

Echando un vistazo a la figura podemos apreciar que las tablas se secuencian empezando por la cero, de tal forma que el proceso de *matching* las recorrerá en orden ascendente, por tanto, siempre se buscarán coincidencias a un paquete en la tabla de flujo cero en primer lugar. Se puede pasar por otras tablas independientemente del orden si la salida de la entrada de flujo de la primera tabla donde se haya encontrado una coincidencia así lo indica.

Una entrada de flujo sólo puede dirigir un paquete hacia otra tabla de flujo en el caso de que el número de secuencia asignado a ésta sea superior. Si una entrada de flujo no dirige el paquete hacia ninguna otra, el *pipeline processing* terminará en esa tabla de flujo y el paquete se procesará de acuerdo a la acción o conjunto de acciones que tenga asignadas.

5.2.1.2. *Group Tables*

Una tabla grupal representa un grupo de entradas. Esta agrupación permite a OpenFlow desarrollar métodos adicionales de reenvío, por ejemplo, seleccionar un determinado grupo o todos los switches que existan en la red.

La estructura que tiene una entrada grupal es como sigue:

Group Identifier	Group Type	Counters	Action Buckets

Cuadro 5.2: Estructura de una entrada en una tabla de flujos grupal

En este caso, una entrada de flujo está compuesta por: un identificador

de grupo, de 32 bits sin signo que identifica unívocamente al grupo; un tipo de grupo, que identifica la semántica seguida en ese grupo; contadores que se actualizan cuando un paquete es procesado por un grupo; cubos de acciones, que son grupos ordenados de acciones con sus parámetros asociados.

Los tipos de grupos definidos para las tablas grupales son los siguientes:

- ***All*:** Ejecuta todos los cubos del grupo, se usa para reenvío *multicast* o *broadcast*. El paquete se clona por cada cubo, de manera que si uno de los cubos dirige el paquete por el puerto de entrada, el clon de elimina, la única manera de enviar un paquete por el puerto de entrada es añadiendo un nuevo cubo con una acción de salida por un puerto virtual específico, el OFPP_ IN_ PORT.

- ***Select*:** Ejecuta un cubo de los del grupo. Se elige el cubo en función de un algoritmo que se computa a nivel de switch. Toda la lógica tras la elección del algoritmo es externa a OpenFlow. Si un puerto especificado en alguno de los cubos se cae, el switch puede restringir la selección de cubos al grupo restante de cubos.

- ***Indirect*:** Ejecuta el único cubo definido para ese grupo. Permite que muchos flujos u otros grupos apunten a un identificador grupal común, de esta forma se consigue una convergencia más rápida y efectiva. Este grupo es equivalente al grupo *All* si éste tuviera un sólo cubo definido.

- ***Fast failover*:** Ejecuta el primer cubo activo. Cada cubo de acciones en este grupo apunta a un puerto o a un grupo que controla si ese cubo está activo. Permite al switch cambiar el reenvío sin necesidad de un mecanismo de *round-robin*s.

5.2.1.3. Proceso de *Matching*

Cada entrada de flujo, como ya hemos visto, contiene un valor específico, o de tipo *ANY* para coincidir con cualquier criterio y poder así determinar la acción que se va a realizar sobre un determinado paquete. Si los switches soportan máscaras arbitrarias en la interfaz de origen o en la de destino, o en la IP de origen o la de destino, se pueden usar estos campos para especificar un criterio de coincidencia mucho más específico.

La Figura 5.4 contiene todos los criterios que pueden usarse para el proceso de matching de las entradas de una tabla de flujo:

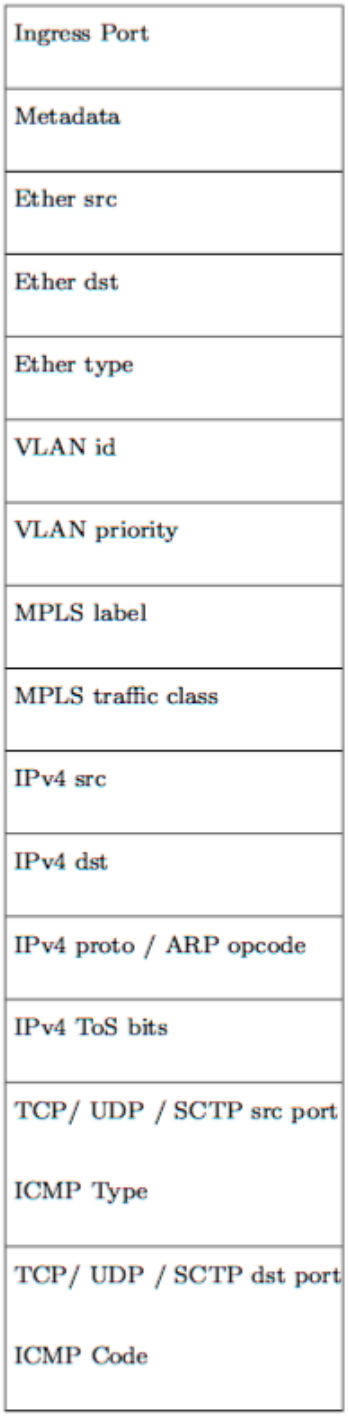


Figura 5.4: Criterios de *Matching* en OpenFlow [9].

Cuando se recibe un paquete, el switch lleva a cabo una serie de funciones, definidas en el diagrama de flujo de la Figura 32. Lo primero que se lleva a cabo en el switch es un proceso de búsqueda de coincidencias en la primera tabla de flujos, para posteriormente, en caso de no encontrar ningún campo coincidente, en función de la configuración del pipeline processing se pasará a otra tabla o se ejecutará una acción sobre el paquete.

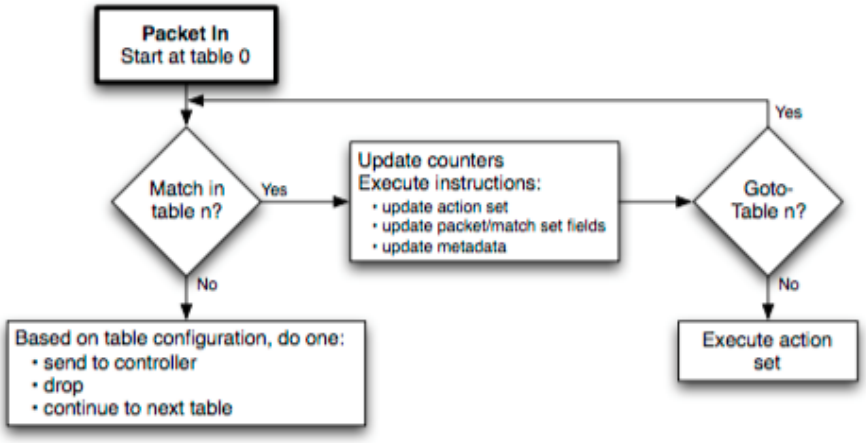


Figura 5.5: Diagrama de flujo del proceso de *matching* en OpenFlow. Imagen obtenida de [9]

Una vez que se ha encontrado una coincidencia, el switch deberá aplicar el conjunto de instrucciones indicadas en la entrada de flujo y posteriormente actualizará el contador correspondiente asociado a la entrada de flujo más prioritaria que haya tenido coincidencia con el paquete.

Dichos contadores deben ser mantenidos por cada tabla, flujo, puerto, cola, grupo y cubo. Los contadores de OpenFlow se pueden implementar en software y pueden ser mantenidos a través de contadores hardware con un rango más limitado de valores. La duración de cada contador hace referencia a la cantidad de tiempo que un flujo lleva instalado en un switch.

5.2.1.4. Acciones

Una entrada de flujo contiene un conjunto de instrucciones que se ejecutan cuando un paquete coincide con dicha entrada. Las instrucciones provocan un cambio en los paquetes, en el conjunto de acciones, o en el proceso de pipeline. Existen cinco tipo de instrucciones en OpenFlow:

- ***Apply-Actions*:** Aplica la acción o acciones correspondientes de forma inmediata y sin realizar ningún cambio sobre el conjunto de acciones. Esta instrucción se usa para modificar un paquete entre dos tablas o para realizar muchas acciones del mismo tipo.
- ***Clear-Actions*:** Borra todas las acciones de un conjunto de manera inmediata.
- ***Write-Actions*:** Incluye las acciones indicadas dentro de un conjunto ya existente de acciones. Si alguna de las acciones que se quiere incluir ya existe en ese conjunto, se sobreescribe, si no, se añade.
- ***Write-Metadata metadata / mask*:** Sobreescribe los metadatos que tengan la máscara a uno. Se le pasa como parámetro unos metadatos y su correspondiente máscara, de tal forma que sólo los bits indicados serán modificados.
- ***Goto-table next-table-id*:** Indica la próxima tabla a la que saltar en el proceso de pipeline.

El conjunto de instrucciones asociado a una entrada de flujo sólo puede contener una acción de cada tipo de entre las anteriores. Además, el orden de ejecución de dichas acciones será también el indicado en los puntos anteriores, de tal forma que por ejemplo, un *Clear-Actions* siempre se ejecutará antes que un *Write-Actions*.

Por su parte, el conjunto de acciones asociadas a cada paquete se encuentra vacío por defecto. Una entrada de flujo puede modificar este conjunto de acciones usando una instrucción de tipo *Write-Action*. Cuando el conjunto de instrucciones no contiene ninguna de tipo *Goto-Table* el proceso de *pipeline* se detiene y se ejecutan las acciones que se hayan definido.

De igual forma que en un conjunto de instrucciones no puede haber dos instrucciones del mismo tipo, en un conjunto de acciones no puede haber un mismo tipo de acción repetida dos veces. También, como sucede con las instrucciones, las acciones tienen un determinado orden de ejecución, especificado en los siguientes puntos:

- ***Copy TTL inwards*:** Se aplica una acción *copy TTL inwards* al paquete.
- ***Pop*:** Aplica acciones "*all tag pop*" al paquete.
- ***Push*:** Aplica acciones "*all tag push*" al paquete.
- ***Copy TTL outwards*:** Aplica una acción "*copy TTL outwards*" al paquete.

- ***Decrement TTL*:** Aplica una acción "*decrement TTL*" al paquete.
- ***Set*:** Aplica todas las acciones "*set-field*" al paquete.
- ***Qos*:** Aplica todas las acciones *QoS*, por ejemplo "*set_ queue*" al paquete.
- ***Group*:** Si existen un grupo de acciones definido, se aplican las acciones a los cubos del grupo, en el orden especificado en esta lista.
- ***Output*:** Si no existe un grupo, el paquete se reenvía por el puerto especificado a través de esta acción.

Todas las acciones que aparecen en esta lista van a ser analizadas a continuación, cuando definamos los tipos de acciones que existen en OpenFlow, es importante conocer el orden porque independientemente de la secuencia en que estas órdenes sean añadidas, siempre se ejecutarán de igual forma, respetando la prioridad marcada por la lista anterior.

La única forma que se define en OpenFlow de ejecutar un conjunto de acciones en un orden diferente es mediante la creación de una lista de acciones. Las acciones especificadas por una lista serán aplicadas inmediatamente, y acompañarán a la instrucción *Apply-Actions* y al mensage *Packet-out*.

Un switch no tiene por qué soportar todos los tipos de acciones, hay acciones que son obligatorias para todos los switches, aquellas marcadas como "*Required Actions*". Cuando un switch se está conectando a un controlador le indica las "*Optional Actions*" que soporta.

A continuación vamos a resumir las acciones especificadas en OpenFlow, poniendo fin a la sección de tablas de flujo y pasando a analizar el canal de comunicación entre el controlador y el switch.

Acción *Output*

La acción de *Output* reenvía un paquete al puerto especificado en la entrada de flujo. Un switch OpenFlow debe soportar el reenvío a puertos físicos y a puertos virtuales definidos por el switch. Además de forma obligatoria (*Required Action*), un switch debe soportar el reenvío a los siguientes puertos virtuales:

- ***ALL*:** Reenvía el paquete a todos los puertos estándar excepto el puerto de entrada y los puertos configurados como puertos de no reenvío.
- ***CONTROLLER*:** Encapsula y envía el paquete al controlador.
- ***TABLE*:** Manda el paquete a la primera tabla de flujo definida, de manera que pueda ser procesado por el *pipeline*.

· ***IN_ PORT*:** Envía el paquete por el puerto de entrada.

Opcionalmente (*Optional Actions*) el switch puede soportar el reenvío a los siguientes puertos virtuales reservados:

· ***LOCAL*:** Envía el paquete a la pila de enrutado local del switch, de esta forma se permite que entidades remotas interaccionen con el switch a través de OpenFlow.

· ***NORMAL*:** Indica que el paquete se va a procesar por los mecanismos tradicionales, no a través del procesado del *pipeline* de OpenFlow. Si el switch no puede enviar paquetes desde el *pipeline* de OpenFlow al *pipeline* normal del switch debe indicarlo al conectarse al controlador.

· ***FLOOD*:** Realiza una inundación del paquete hacia todos los puertos estándar salvo el puerto de ingreso y los puertos que estén en estado bloqueado.

El hecho de que parte de las acciones de Output sean opcionales se debe a la ya comentada existencia de dos tipos de switch soportados por el protocolo OpenFlow, los híbridos y los que sólo usan el protocolo OpenFlow. Son éstos últimos los que no pueden soportar las acciones de tipo *NORMAL* ó *LOCAL*.

Acción *Set-Queue*

Esta acción de tipo *Optional Actions* asigna el id de cola de un paquete, de forma que cuando un paquete es reenviado desde un swith a un determinado puerto, este id determinará cuál de las colas de ese puerto será la que se encargará del reenvío del paquete. El comportamiento de la cola con respecto al reenvío dependerá de la propia configuración de la cola y suele usarse para dar QoS a nivel básico.

Acción *Drop*

Esta acción, de tipo *Required Actions* no puede representarse por medio de ningún acción en sí, si no que es el resultado de un paquete cuyo conjunto de acciones no indica ninguna salida. Por ejemplo, un conjunto de instrucciones vacías resulta en un *Drop*, o un cubo de acciones vacío durante el proceso de pipeline, o después de ejecutar una instrucción de tipo *Clear-Actions*.

Acción Group

Igualmente de tipo *Required Actions*, envía el paquete para que sea procesado por un grupo. El procesado del paquete dependerá de la propia configuración del grupo.

Acción Pop-Tag/Push-Tag

Algunos switches pueden implementar la opción de añadir o eliminar cabeceras. Una cabecera que se añada lo hará en la posición más a la izquierda de las cabeceras existentes siempre siguiendo un orden preestablecido que puede verse en la siguiente tabla:

Ethernet	VLAN	MPLS	ARP-IP	TCP/UDP/SCTP (IP Only)

Cuadro 5.3: Orden seguido para los campos de cabecera en OpenFlow

Acción Set-Field

Esta acción de tipo *Optional Actions* se utiliza para modificar los respectivos valores de los campos de cabecera de un paquete.

5.2.2. Canal OpenFlow

La última parte del protocolo OpenFlow que resta por analizar es la que representa a la interfaz que conecta cada switch OpenFlow con el controlador. A través de esta vía, el controlador recibe eventos del switch, genera salidas para los paquetes y en general, gestiona el funcionamiento de los switches.

Todos los mensajes enviados a través de este canal deben seguir el formato especificado por el protocolo OpenFlow y suelen estar encriptados usando *Transport Layer Security* (TLS), aunque puede usarse directamente sobre *Transmission Control Protocol* (TCP).

5.2.2.1. Mensajes del protocolo OpenFlow

En el protocolo OpenFlow se utilizan tres tipos de mensajes, los de tipo controlador-switch, los asíncronos y los simétricos, cada uno con sus subtipos propios. Los mensajes controlador-switch son usados por el controlador para ver y gestionar el estado de los switches. Los de tipo asíncronos son enviados por el switch para actualizar el controlador con respecto al estado de los switches. Los simétricos son iniciados tanto por el controlador como por el switch, y no necesitan realizar una solicitud previa.

5.2.2.1.1 Mensajes Controlador-switch

Dentro de los mensajes controlador-switch tenemos los siguientes subtipos de mensajes:

- ***Features*:** Este mensaje se utiliza para que el controlador pueda pedir información sobre las capacidades de un switch. El switch al recibir

este mensaje debe responder al controlador con un mensaje en el que especifique sus capacidades. Se suelen utilizar para el establecimiento del canal OpenFlow.

- ***Configuration*:** Modifica y encola los parámetros de configuración de un switch.
- ***Modify-State*:** Estos mensajes modifican el estado de los switches. El propósito principal de estos flujos es añadir, eliminar o modificar flujos o grupos de flujos o modificar las propiedades de un puerto del switch.
- ***Real-State*:** Colecciona información de las estadísticas del switch.
- ***Packet-out*:** El controlador usa estos mensajes para enviar un paquete como salida a través de un determinado puerto del switch y para reenviar paquetes recibidos a traves de un *Packet-in*. Los mensajes de tipo *Packet-out* deben contener un paquete completo, o un identificador que referencie a un paquete almacenado en el switch.
- ***Barrier*:** Este tipo de mensaje es usado por el controlador para asegurarse de que las dependencias de un mensaje se han cumplido o para recibir notificaciones de operaciones completadas.

5.2.2.1.2 Mensajes asíncronos

Los mensajes asíncronos son enviados por los switches sin que el controlador los haya reclamado. Hay varias situaciones en las que un switch puede enviar un mensaje a un controlador, por ejemplo, para notificar la llegada de un mensaje, un cambio en el estado del switch o un error. Estos son los mensajes asíncronos más comunes:

- ***Packet-in*:** Por cada paquete que no tiene coincidencia con ninguna entrada de las tablas de flujo, un mensaje de *packet-in* debe ser enviado al controlador. Para todos los mensajes dirigidos al puerto virtual del controlador, se envía un evento de *packet-in* al controlador. Los switches que tienen capacidad de almacenamiento pueden enviar parte de la cabecera de los paquetes en los *packet-in*, así como un identificador de buffer, de forma que el controlador pueda indicar al switch cuando disponer los paquetes de un determinado *buffer*. Los switches que no tienen capacidad de almacenamiento deben enviar el paquete entero al controlador como parte del evento *packet-in*.
- ***Flow-Removed*:** Cuando un flujo es añadido a una tabla de flujos de un switch, un temporizador se asigna a dicha entrada para indicar

cuándo se debe borrar ese flujo por un periodo excesivo de inactividad. De igual forma otro temporizador se activa para indicar cuándo se debe borrar dicha entrada, incluso si ha habido actividad durante dicho periodo. Los mensajes de tipo *Flow-removed* son enviados por el switch para indicar al controlador que un flujo ha expirado.

- ***Port-Status*:** Se espera que el switch envíe estos mensajes al controlador cuando se produzca un cambio en el estado de un puerto del switch.

- ***Error*:** Son mensajes que el switch utiliza para indicar al controlador que se ha producido algún tipo de error.

5.2.2.1.3 Mensajes simétricos

Los mensajes simétricos son enviados tanto por switch como por controlador, sin necesidad de solicitud previa. Existen tres tipos principales de mensajes simétricos:

- ***Hello*:** Este tipo de mensaje se intercambia entre controlador y switch durante el inicio de la conexión.

- ***Echo*:** Son mensajes de petición y respuesta, que pueden ser iniciados por el controlador o por el switch y que deben recibir otro mensaje echo como respuesta. En general estos mensajes se usan para medir la latencia o el ancho de banda del canal, así como para verificar que el par se encuentra activo.

- ***Experimenter*:** Estos mensajes se utilizan por OpenFlow para dar una forma estándar a los switches de ofrecer servicios adicionales dentro del espacio de mensajes del protocolo.

5.2.2.2. Establecimiento de la conexión

Para que se establezca una conexión entre el switch y el controlador, es necesario disponer de una dirección IP configurable a nivel de usuario, al igual que un puerto. Con esos dos elementos conocidos, el switch puede iniciar una conexión de tipo *Transport Layer Security* (TLS) o *Transmission Control Protocol* (TCP).

Una vez establecida la conexión, cada extremo de la misma debe enviar un mensaje de tipo *hello* con la versión del protocolo más alta que soporten cada uno, la versión usada será la más baja enviada entre los dos participantes de la conexión.

Si la versión negociada es soportada por ambos participantes, entonces la conexión se mantiene y comienza el intercambio de información entre los extremos. Si por el contrario ocurre algún fallo durante el proceso de establecimiento, el receptor debe responder con un con un mensaje de error indicando el tipo del fallo y un código que indica que el fallo se ha producido por incompatibilidad.

5.2.2.3. Interrupción de la conexión

En el caso de que tenga lugar un *timeout* en un mensaje *echo*, o se pierda la conexión TLS, o ocurra cualquier otro evento de desconexión que provoque la pérdida de contacto del switch con el controlador, el switch debe tratar de conectar con controladores de *backup*. El orden en el que el controlador debe contactar a estos controladores no está establecido por el protocolo.

Hay dos modos que el switch puede adoptar cuando pierde la conexión con el controlador: "*fail secure mode*" ó "*fail standalone mode*". En el primero de los modos, el único cambio con respecto al funcionamiento normal del switch es que los mensajes que fueran dirigidos al controlador ahora serán eliminados. En el segundo caso, el switch procesará todos los paquetes como si se tratase de un switch *ethernet*, a través el puerto establecido para dicho comportamiento, OFPP_ NORMAL.

Cuando se consigue establecer de nuevo la conexión con un controlador, éste tiene la opción de borrar las entradas de flujo restantes.

5.2.2.4. Encriptado

Para el encriptado de mensajes entre el switch y el controlador, el switch, durante el establecimiento de la conexión debe iniciar una conexión de tipo *Transport Layer Security* (TLS), que normalmente suele estar en el puerto 6633. En ese momento, tanto switch como controlador se autenticarán mediante el intercambio de certificados firmados con clave privada.

Cada switch debe poder configurarse por el usuario, para lo cual se necesitará un certificado de controlador o un certificado de switch.

5.2.2.5. Gestión de mensajes

En cuanto a la gestión de los mensajes, OpenFlow ofrece control en el envío y el procesado de mensajes, pero no asegura su ordenación, ni el envío de mensajes de acknowledgment.

En lo referente al envío de mensajes, su envío es garantizado en OpenFlow, con la excepción de un fallo completo de la conexión, en cuyo caso el

controlador no podrá asumir nada acerca del estado del switch.

El procesado de mensajes debe realizarse en los switches para cualquier mensaje y de forma completa, pudiendo generar una respuesta en función del mensaje. Si un switch no puede procesar por completo un mensaje, éste debe ser enviado al controlador con un mensaje de error. Cuando se trata de un packet-out, el procesado del mensaje no garantiza que el paquete salga del switch, entran en juego factores como su filtrado por el procesado de OpenFlow debido a una congestión en el switch, o por políticas de calidad de servicio, o puede ser enviado a un puerto bloqueado o incorrecto.

La ordenación se puede controlar a través del uso de mensajes de tipo *barrier*. Si no hay mensajes de este tipo, los switches pueden reordenar de forma arbitraria los paquetes para conseguir un mejor rendimiento. Es por esto que los controladores no dependen del ordenamiento de los paquetes, y los flujos se pueden añadir en un orden distinto del que fueron recibidos en los mensajes de modificación de flujos. No obstante, cuando se envía un mensaje de tipo *barrier*, hay varios factores que permiten ordenar el procesado de los paquetes:

- Los mensajes previos a un mensaje de tipo *barrier* deben ser completamente procesados antes del procesado de dicho mensaje, incluyendo el envío de mensajes de respuesta o error.
- Después de que todos los mensajes hayan sido procesados, se puede procesar el mensaje *barrier* y se debe generar una respuesta del mismo tipo.
- Los mensajes que lleguen tras el mensaje barrier serán procesados una vez terminado el procesado del mensaje *barrier*.

Con esto concluye el apartado destinado a la comprensión del funcionamiento de las tecnologías que van a suponer la base de nuestro proyecto, las redes definidas por *software* y el protocolo OpenFlow para la gestión de la comunicación entre los elementos de red y el controlador.

En el siguiente capítulo analizaremos las herramientas utilizadas para hacer uso de estas tecnologías, Mininet, para la virtualización de redes, y OpenDayLight para gestionar las operaciones relacionadas con el controlador.

Capítulo 6

Herramientas utilizadas

En este capítulo vamos a tratar las dos herramientas fundamentales que son la base sobre la que se desarrollará todo el proyecto: Mininet, que soportará todas las funciones de virtualización de redes y OpenDayLight, una plataforma muy versátil que entre otras cosas se utiliza para la automatización de servicios, virtualización, optimización, visualización y control de redes, siendo ésta última la funcionalidad que más nos interesa para el desarrollo de nuestro proyecto.

Ambas herramientas son ampliamente utilizadas, y la documentación y ejemplos de uso son muy abundantes, siendo una de las principales razones para la elección de ambas herramientas que además, como ya hemos comentado en apartados anteriores, suelen usarse muy de la mano para la implementación de sistemas de control de flujo en redes virtualizadas.

6.1. Mininet

6.1.1. ¿Por qué usar Mininet?

Mininet es una herramienta de simulación de redes a bajo nivel que, de forma sencilla y rápida, nos permite simular topologías de red típicas con un número reducido de hosts y switches, permitiéndonos además elegir entre varios tipos de controladores para gestionar los flujos de encaminamiento entre switches.

Además de esto, una de las principales ventajas de Mininet es que cada *host* y cada *switch* se puede configurar a través de una ventana de comandos que tiene toda la potencia de la terminal de Linux, por lo que podemos usar y ejecutar cualquier orden que ejecutaríamos en Linux. Este punto es fundamental, pues más adelante implementaremos una aplicación cliente-

servidor que será ejecutada en los *switches* y que simulará el proceso de señalización de *handover* en nuestra red 5G. Contar con la capacidad de ejecución de aplicaciones JAVA será por tanto un requisito fundamental en cualquier herramienta de simulación de redes que se quisiera usar para realizar nuestro proceso de *handover*.

6.1.2. Usando Mininet

Para comprender de forma clara y sencilla cómo funciona Mininet, la mejor opción es seguir alguno de los muchos tutoriales básicos que podemos encontrar fácilmente en internet. En [10] encontramos un tutorial en el que, apartado por apartado, se realiza un recorrido por todas las características principales de Mininet, de las cuales recogemos algunas de las más importantes de forma resumida en la tabla 6.1:

Adicionalmente, en la Figura 6.1 y la Figura 6.3 vemos dos ejemplos diferenciados de cómo crear una topología en Mininet. El primero de ellos, que aparece en la Figura 6.1 muestra la creación de una topología sencilla de cuatro *switches* y cuatro *hosts*, uno por switch, en la que los *switches* están conectados entre sí de manera lineal, dicha topología puede apreciarse de forma clara en la figura 6.2.

```
ubuntu@sdnhubvm:~/Desktop/TFG_INFO/componentes_topologia_final/topologia mininet[13:10]$ sudo mn --test pingall --topo linear,4
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller

*** Starting 4 switches
s1 s2 s3 s4 ...
*** Waiting for switches to connect
s1 s2 s3 s4
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Figura 6.1: Creación y testeo de conexión con una red básica en Mininet.

Este primer ejemplo es una buena muestra de la sencillez de Mininet a la hora de crear topologías comunes, con una cantidad reducida de nodos, de forma prácticamente instantánea. Como se puede observar, con un sólo comando podemos implementar dicha topología, representada en la Figura 6.2:

Al mismo tiempo, hemos probado la conectividad entre todos los nodos, cerciorándonos de que la topología funciona adecuadamente.

Funcionalidad	**Comandos**	**Funciones especificas**
Información general y de topologia	>help	Obtiene información de los comandos que podemos usar en Mininet
	>dump	Obtiene información de todos los nodos que forman parte de nuestra topología
Información en un nodo. Comandos aplicados a cada nodo, (*switches* o *hosts*)	>h1 ifconfig -a	Muestra la configuración de los interfaces de un nodo, en este caso en el *host* 1
	>s1 ps -a	Muestra todos los procesos ejecutándose en un nodo, en este caso *switch* 1
	>h1 *ping* -c 1 h2	Comprueba la conectividad entre dos nodos, en este caso los *hosts* 1 y 2
	>*pingall*	Util para comprobar que todos los nodos de nuestra topología son accesibles
Opciones de configuración de elementos de nuestra topología	– link [bw, delay, loss, max_queue_size, use_htb]	Configurar ancho de banda (Mbit), *delay*, pérdida de paquetes en %, tamaño de colas y uso de *Hierarchical Token Bucket*
Selección del tipo de *switch*	– switch [user, ovsk]	Crear topologías con switches de tipo user-space u Open vSwitch
Ejecutar comandos python	>py	Permite ejecutar funciones definidas en python.
Cargar topologías personalizadas	– custom <topologia.py >	Carga un escenario cuya topología ha sido previamente implementada en python.

Cuadro 6.1: Comandos útiles en Mininet

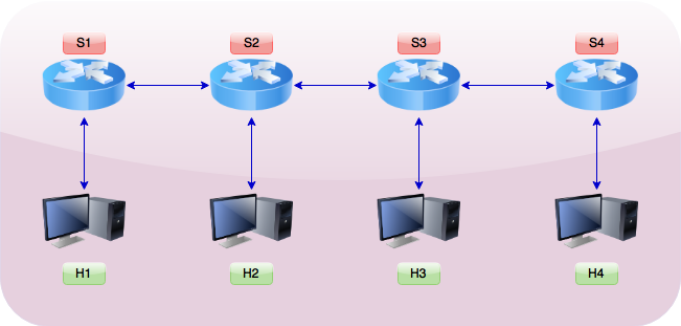


Figura 6.2: Topologia lineal con cuatro hosts. Diagrama realizado con Draw.io

Un caso algo más completo es el que se puede observar en la Figura 6.3. Quizá éste sea el caso que más nos interese de cara al desarrollo del proyecto. Previamente hemos mencionado que en Mininet podemos cargar topologías customizadas haciendo uso de la opción *-custom «topologia.py»*, sin embargo, una opción que suele dar mejores resultados es ejecutar la topología a través de la terminal de comandos lanzando directamente el script de Python que hayamos creado.

```
ubuntu@sdnhubvm:~/Desktop/TFG_INFO/componentes_topologia_final/topologia mininet[13:06]$ sudo python 3switches-3host-2ap.py
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding switch s1
*** Adding wlan0 a switch 1
*** Checking wlan0
Error setting wlan0 up: wlan0: ERROR while getting interface flags: No such device
 *** Adding switch s2
*** Adding wlan2 a switch 1*** Checking wlan1
Error setting wlan1 up: wlan1: ERROR while getting interface flags: No such device
 *** Adding switch s3
*** Adding hosts*** Adding links***Starting network
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
***Starting network
*** Starting CLI:
mininet> dump
<Host h1: h1-eth0:192.168.1.1 pid=3408>
<Host h2: h2-eth0:192.168.1.2 pid=3415>
<Host h3: h3-eth0:192.168.1.3 pid=3420>
<OVSSwitch s1: lo:127.0.0.1,wlan0:None,s1-eth2:None,s1-eth3:None pid=3387>
<OVSSwitch s2: lo:127.0.0.1,wlan1:None,s2-eth2:None,s2-eth3:None pid=3395>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=3403>
<RemoteController c0: 127.0.0.1:6633 pid=3380>
mininet>
```

Figura 6.3: Simulación de una topología propia.

En este caso estamos ejecutando una versión muy similar a la que será la versión definitiva de nuestra topología elegida para simular el cambio de celda de un UE, una red formada por tres *switches* con topología en árbol de dos niveles, siendo el nodo raíz el *switch* 3.

A grandes rasgos estas son las posibilidades que nos ofrece Mininet de ca-

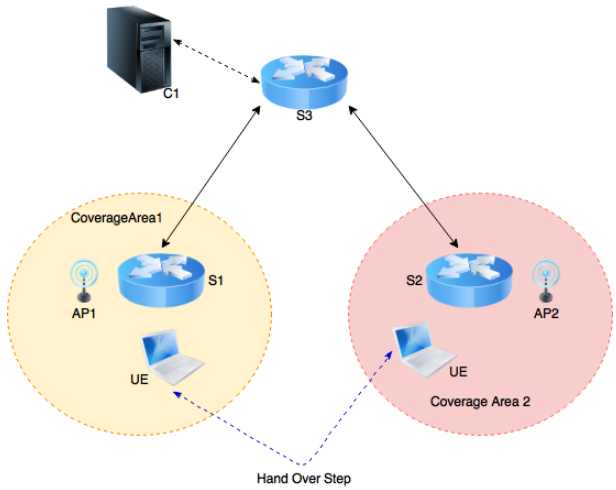


Figura 6.4: Escenario generado a partir de topología propia.

ra al desarrollo de nuestra solucion final. En la siguiente subsección veremos qué otras herramientas para simulación se han considerado, y las ventajas y desventajas que éstas ofrecen con respecto a Mininet.

6.1.3. Comparacion con GNS3

Existen otras muchas herramientas para la simulacion de redes, algunas de ellas, como *Packet Tracer* de Cisco, o GNS3 cuentan con una interfaz de desarrollo mucho más potente que la de Mininet, pero presentan otras desventajas que las convierten en alternativas menos atractivas a la hora de ser usadas como herramientas para la simulación de nuestro escenario.

En este apartado vamos a mostrar cómo sería el desarollo de una topología sencilla con GNS3, puesto que *Packet Tracer* ofrece una versión de estudiantes con capacidades algo limitadas, lo que quizás sea menos interesante de cara a la simulación de escenarios más complejos que pudieran desarrollarse.

GNS3 es una herramienta de simulación cuya principal ventaja es la cantidad de elementos de red distintos con los que trabaja. En [11] se puede ver una lista de todos los sistemas operativos con los que puede trabajar y todos los fabricantes de los distintos elementos de red que se pueden emplear en las simulaciones.

Por defecto, al descargar GNS3 tanto en Windows como en Mac OSX, se descargan algunos paquetes que permiten implementar, entre otras cosas equipos virtuales que nos permitirán acceder a una pequeña lista de funciones propias de GNS3 para la configuración de las redes.

La Figura 6.5 nos muestra un pequeño ejemplo de cómo se ve la interfaz de GNS3 y de los distintos comandos que pueden usarse con los paquetes más básicos descargados:

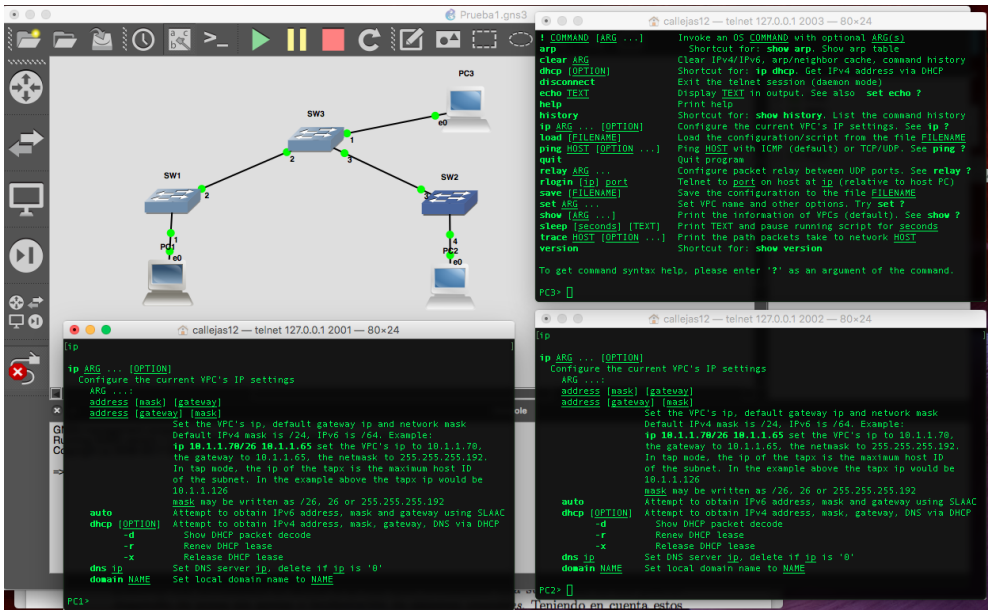


Figura 6.5: Interfaz GNS3 para la simulacion de redes.

De manera rápida y sencilla se pueden arrastrar los elementos de red que queramos, elegir las interfaces a través de las cuales éstos se conectan y realizar configuraciones básicas.

El principal problema de GNS3 de cara a su empleo en nuestro proyecto es que a priori no está integrado con switches que sean compatibles con OpenFlow, por lo que necesaritaríamos realizar configuraciones más complejas de las requeridas en Mininet para crear una topología con todas las características que necesitamos e integrada con el framework OpenDayLight que funcionen bajo el protocolo OpenFlow.

Es por este motivo por el cuál Mininet, como ya hemos comentado anteriormente, es la elección que mejor se adapta a las necesidades de simulación de nuestro proyecto.

6.2. OpenDayLight

Una vez introducida la herramienta que se usará para la simulación de topologías de nuestro proyecto, estamos listos para hablar del controlador que se va a utilizar para dotar de inteligencia a los nodos de nuestra red de

forma que puedan redirigir los flujos de datos.

Este controlador será OpenDayLight, de nuevo por su total compatibilidad con Mininet a la hora de comunicarse con los *switches* de nuestra red, además de por su potencia y el soporte de una gran comunidad de desarrolladores.

6.2.1. ¿Qué es OpenDayLight?

OpenDayLight es un proyecto Linux que cuenta con el apoyo de empresas tan importantes como CISCO, Redhat, Intel, HP, Dell, o Ericsson entre otras, y que se centra en desarrollar un *framework* abierto para trabajar con tecnologías relacionadas con las redes definidas por software SDN y la virtualización de redes *Network Function Virtualization* (NFV).

Como puede apreciarse en la Figura 6.6, existen varias filosofías a la hora de crear un proyecto basaco en OpenDayLight, ordenados de izquierda a derecha de menor a mayor nivel de integración con el protocolo OpenFlow, podemos ver las formas en las que OpenDayLight puede integrarse en un sistema para ofrecer una solución adecuada a las necesitades del proyecto.

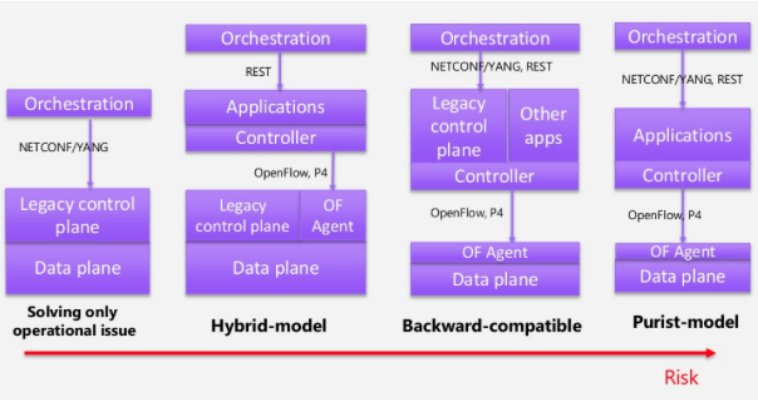


Figura 6.6: Espectro de filosofías SDN abarcadas por OpenDayLight. Imagen obtenida de [12]

En el apartado de arquitectura OpenDayLight describiremos muchos elementos que aparecen en la Figura 6.6, por el momento nos interesa comprender que OpenDayLight ofrece un marco flexible y adaptable de trabajo, con el que integrar un controlador de flujos en sistemas que a priori podrían no estar pensados para tales fines, cediendo parte o la totalidad del plano de control a través del protocolo OpenFlow a un controlador virtualizado a nivel *software* que se encargará de redirigir los flujos de datos de la forma que a nosotros más nos interese.

El primer paso a la hora de llevar a cabo un proyecto con OpenDayLight, será por tanto decidir qué paradigma se va a aplicar, qué nivel de integración, de tal forma que podamos elegir con criterio la distribución de OpenDayLight que encaje mejor en nuestros requisitos y hacer el mejor uso posible de las herramientas que se ponen a nuestra disposición.

6.2.2. Distribuciones de OpenDayLight

Al tratarse OpenDayLight de un *framework* abierto, y altamente mantenido por sus desarrolladores dadas sus amplias aplicaciones industriales, existen muchas y muy variadas versiones de OpenDayLight, cada una de las cuales supone una evolución de la anterior y añade nuevas herramientas y utilidades.

Por ello, en este apartado está dedicado a aclarar las posibilidades de cada una de las distribuciones, analizar sus características y suponer una pequeña guía para que un usuario que quiere empezar a desarrollar su aplicación pueda decidir qué distribución se adapta mejor a sus necesidades.

6.2.2.1. Alternativas de instalación

Antes de todo, este breve apartado detalla las posibilidades de instalación que la plataforma OpenDayLight pone a nuestra disposición para poder hacer uso del *framework* de la manera que nos pueda resultar más sencilla.

- **Instalación desde ZIP:** Esta es la forma más sencilla de instalar OpenDayLight en nuestro equipo. En este caso, al descomprimir el ZIP nos encontraremos con todos los ficheros principales bajo el directorio "/opendaylight". Entre estos ficheros están:
  - ***run.sh*:** Lanza la aplicación en los sistemas operativos en los que OpenDayLight es compatible, excepto en Windows.
  - ***run.bat*:** Lanzador de OpenDayLight para sistemas Windows.
  - ***version.properties*:** Indica la versión de OpenDayLight que estamos montando en nuesto sistema.
  - ***configuration*:** Contiene los ficheros con la configuración inicial con la que se va a arrancar nuestro controlador.
  - ***lib*:** Contiene las principales librerías de JAVA de las que hace uso el *framework*.
  - ***plugins*:** Contiene *plugins Open Services Gateway initiative* (OSGi) de OpenDayLight.

- **Instalación desde RPM:** Podemos usar el repositorio *yum* para instalar OpenDayLight, para lo cual debemos seguir los pasos:
  - **Instalación del repositorio *yum*:** Para ello basta con ejecutar el comando "sudo apt-get install yum" en nuestra terminal, en el caso de trabajar con Linux.
  - **Descargar el fichero de repositorio:** En [13] se puede ver la ruta del fichero de repositorio de *yum* para OpenDayLight.
  - **Instalar el fichero:** Haciendo uso del comando "sudo rpm -Uvh opendaylight-release-0.1.0-2.fc19.noarch.rpm"
  - **Instalar OpenDayLight:** En alguna de las versiones disponibles en este repositorio, a saber:
    - Edición base: "sudo yum install opendaylight".
    - Edición de virtualización: "sudo yum install opendaylight-virtualization"
    - Service Provider: "sudo yum install opendaylight-serviceprovider"
- **Imagen virtualizable:** Esta será la opción utilizada en nuestro proyecto, y en nuestro caso usaremos una imagen disponible en SDNHub [14]. Los pasos para usar una imagen virtualizada que contenga OpenDayLight son sencillos, similares a los que se seguirían para utilizar cualquier imagen de un sistema operativo:
  - **Configuración:** Para el correcto funcionamiento de OpenDayLight en nuestra máquina virtual es importante que dotemos a la misma de al menos 2GB de RAM, y que activemos *Network Adress Translation* (NAT) para que nuestra imagen tenga acceso a internet.

Existen algunas posibilidades más a la hora de hacer uso de OpenDayLight, como la utilización de un contenedor como Docker, o su implementación en *clusters*, pero ambos modelos se alejan demasiado del objetivo del proyecto, en cualquier caso, si se quiere consultar más información sobre cómo intalar o usar OpenDayLight en cualquier plataforma, en [15] se puede consultar cualquier detalle.

6.2.2.2. *Hydrogen*, la distribución base

Esta distribución de OpenDayLight data de principios de 2014, se trata del primer lanzamiento de este *framework*, y por ello, cuenta con los paquetes más básicos y un número limitado de herramientas, entre las que se encuentran:

- ***Clustering Manager*:** Gestiona la caché compartida entre todas las instancias de los controladores.

- ***Container Manager*:** Controla los recursos de las red y el *Network Slicing* [16].

- ***Switch Manager*:** Se hace cargo de la información de los dispositivos que forman parte del South Bound (SB).

- ***Statistics Manager*:** Colecciona información de todo el sistema.

- ***Topology Manager*:** Construye la topología de la red.

- ***Host Tracker*:** Rastrea todos los dispositivos conectados al sistema.

- ***Forwarding Rules Manager*:** Instala los flujos en los dispositivos del SB.

- ***ARP Handler*:** Gestiona los mensajes ARP del sistema.

- ***Forwarding Manager*:** Instala las rutas y rastrea el proximo salto en la topología de red construida.

- **OpenFlow *Plugin*:** Se encarga de la interacción con OpenFlow y los dispotivos que lo implementan.

- **NetConf *Plugin*:** Gestiona la interacción con lo *switches* NetConf.

En esta primera distrución solo se incluyen los protocolos OpenFlow, NetConf y OVSDB en la interfaz SB. La arquitectura de esta distribución base puede observarse en su totalidad en la Figura 6.7

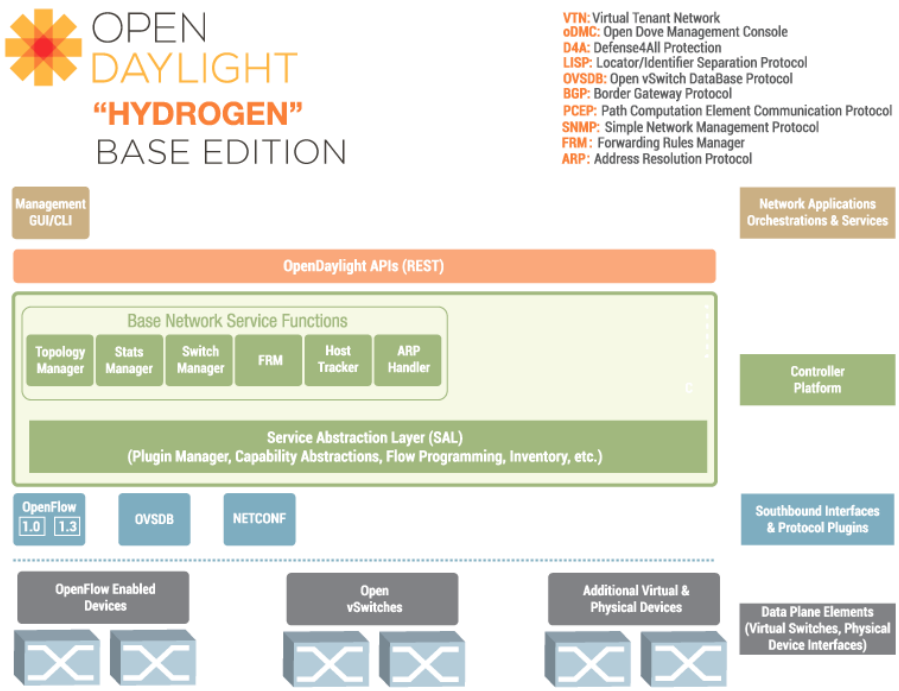


Figura 6.7: Arquitectura de la distrución Hydrogen [17].

6.2.2.3. *Hellium*

Hellium es la segunda versión del *framework* OpenDayLight, que incluye algunas mejoras importantes con respecto a la versión anterior, como por ejemplo la integración con Karaf, un proyecto de Apache que tiene como objetivo crear un contenedor ligero que permita la integración con módulos que pertenezcan a tecnologías muy distintas.

Entre las principales características de Karaf encontramos las siguientes:

- **Despliegue en caliente:** Basta con arrastrar un fichero en el directorio de Karaf para que éste detecte el tipo de fichero de que se trata e intente desplegarlo.
- **Consola:** Karaf cuenta con una consola con las mismas caracteristicas que una consola de Linux desde la cual se pueden controlar todos los aspectos del contenedor.
- **Configuración dinámica:** Karaf también cuenta con un conjunto de comandos específicos de su tecnología y especialmente pensados para gestionar su propia configuración. Además todos los ficheros de configuración de Karaf están agrupados en un mismo directorio, de

manera que un cambio en alguna de sus configuraciones es rápidamente detectado y cargado.

- **Sistema avanzado de *logging*:** En Karaf se implementan los sistemas más populares de logging, como slf4j o log4j. Se use el sistema que se use, Karaf centralizará la configuración en un solo fichero.

- **Provisionamiento:** Esta tecnología cuenta con un gran número de URL en las que instalar nuestras aplicaciones, e incluye el concepto de "Karaf feature" que es una manera de describir nuestra aplicación propia de Karaf.

- **Gestión:** Apache Karaf es un contenedor listo para empresas, por lo que contiene muchos indicadores y operaciones de gestión a través de *Java Management Extensions* (JMX).

- **Uso remoto:** Karaf incluye un servidor *Secure Shell* (SSH) que permite acceder a la consola remotamente. De igual manera puede accederse a la capa de gestión.

- **Seguridad:** Esta tecnología cuenta además con un completo *framework* de seguridad basado en *Java Authentication and Authorization Service* (JAAS) y cuenta con un mecanismo basado en *Role-Based Access Control* (RBAC) para el acceso a través de terminal y de Java Management Extensions (JMX).

- **Múltiples instancias:** Todas ellas pudiendo ser gestionadas desde una única instancia raíz.

- **Frameworks OSGi:** Karaf no está estrictamente acoplado a un solo *framework* OSGi, si bien funciona con Apache Felix *Framework*, puede cambiarse de manera sencilla para funcionar con, por ejemplo, Equinox.

Para consultar más sobre estas u otras características de Karaf, en [18] y [19] podemos encontrar información interesante.

Finalmente, en la Figura 6.8 podemos ver el diagrama de la arquitectura de esta distribución de OpenDayLight.

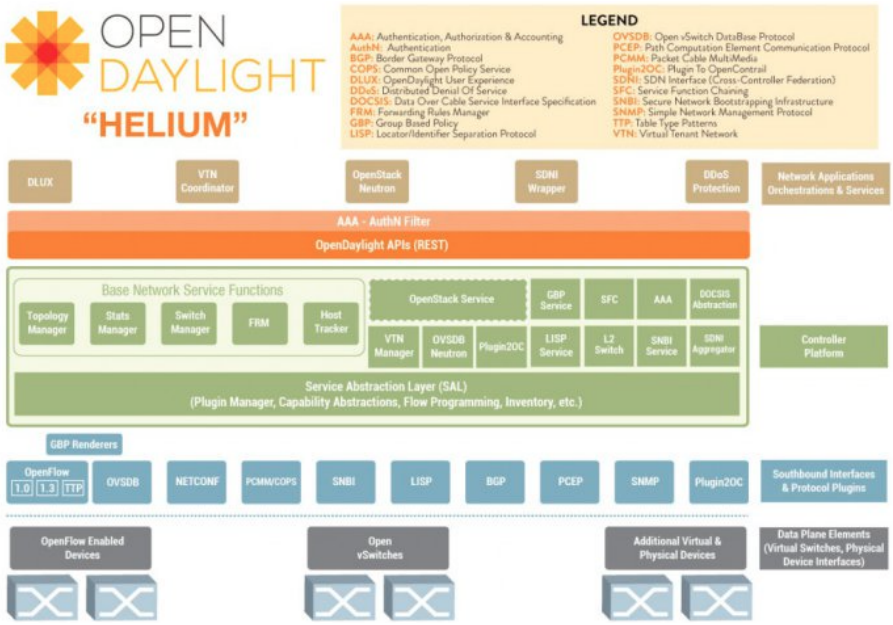


Figura 6.8: Arquitectura de la distrución *Hellium* [17].

6.2.2.4. *Lithium*

Lithium en la siguiente versión de OpenDayLight, lanzada a partir de junio de 2015. En este caso, el diagrama en el que vemos su arquitectura es el que aparece en la Figura 6.9:

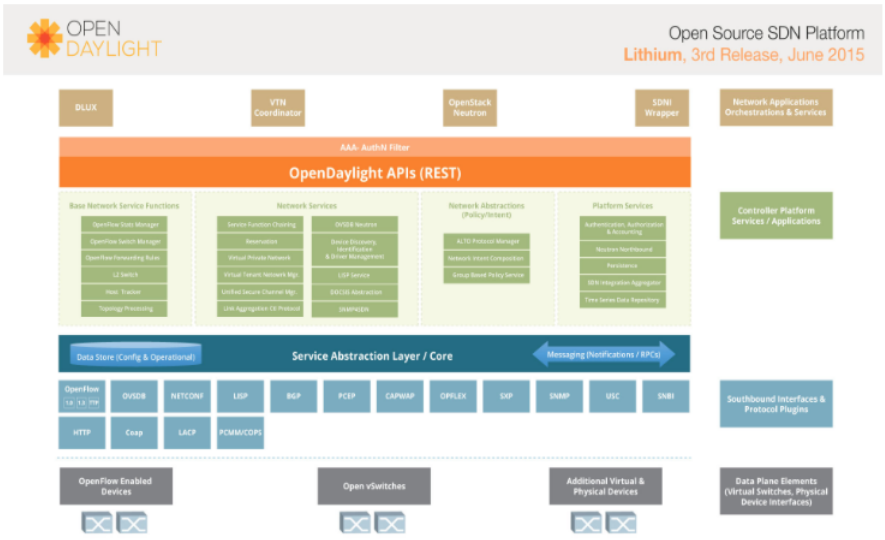


Figura 6.9: Arquitectura de la distrución *Lithium* [17].

En esta distribución se añaden nuevos módulos en el SB, entre los que se encuentran: *Hypertext Transfer Protocol* (HTTP), *Constrained Application Protocol* (CoAP), *Link Aggregation Control Protocol* (LACP), y *Packet Cable MultiMedia/Common Open Policy Service* (PCMM/COPS). Además introduce mejoras en su sistema de authenticación *Authentication, Authorization and Accounting* (AAA).

Si atendemos a los diagramas anteriores, se añaden nuevos módulos a las API's REST, que ahora pasa a dividirse en cuatro categorias: *Base Network Service Functions*, *Network Services*, *Network Abstraction* y *Platform Services*.

6.2.2.5. *Beryllium*

Beryllium es la cuarta distribución de OpenDayLight. En este caso, la distribución presenta grandes mejoras en rendimiento, funcionalidad y escalabilidad con respecto a las versiones anterior.

Algunos de los factores más reseñables que sirven para evidenciar estas mejoras son los siguientes:

- **Rendimiento:** En la distribución *Beryllium* se permite que varias instancias de OpenDayLight actuen como un sólo controlador, lo que resulta en una capacidad de análisis y *testing* mucho mayor.
- **Facilidad de adopción:** Se integran nuevas características para facilitar la interoperabilidad en entornos multiproveedor, a través de actualizaciones de su arquitectura de microservicios y de proyectos como NetIDE para el acceso independiente a recursos de red.
- **Nuevos modelos de abstracción:** OpenDayLight incluye un amplio rango de configuraciones para generar políticas e implementación de cualquier plataforma o controlador, a través de cuatro métodos: *Network Mobility* (NEMO), *Application Layer Traffic Optimization* (ALTO), *Group Based Policy* (GBP) y *Network Intent Composition* (NIC).
- **Amplio conjunto de casos de uso:** OpenDayLight cuenta con un gran número de casos de uso, tanto tradicionales como punteros, pensados para proveedores de servicios y redes empresariales. Las mejoras en esta versión de OpenDayLight potencian sobre todo su uso en la nube y en la virtualización de redes.

En la Figura 6.10 vemos la arquitectura para esta distribución de OpenDayLight, en la que podemos observar la inclusión de algunos de los módulos comentados en los apartados anteriores.

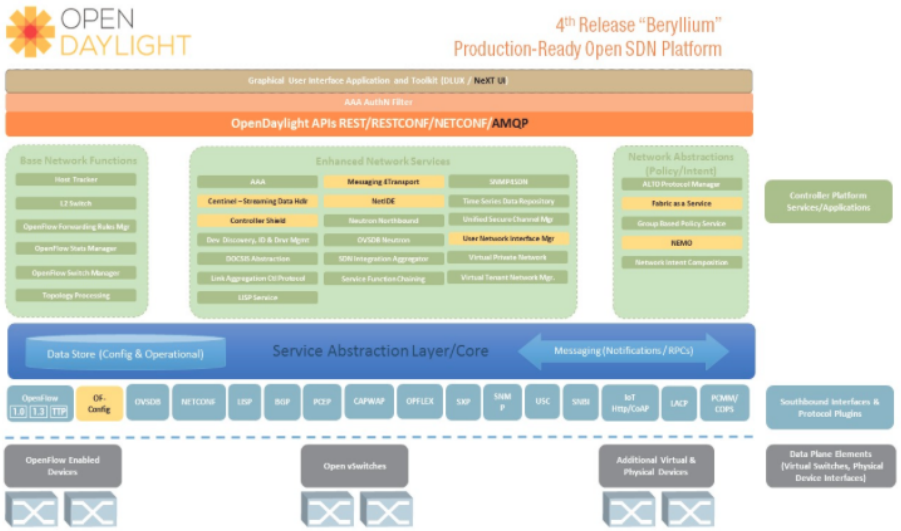


Figura 6.10: Arquitectura de la distrución *Beryllium* [17].

Existen versiones más recientes de OpenDayLight, por ejemplo la distribución *Boron*, pero no vamos a tratarla en este apartado, pues para nuestra implementación usaremos la distribución *Beryllium* que viene integrada con la imagen virtual con la que vamos a trabajar y que cuenta con los recursos suficientes para implementar el controlador que necesitamos.

6.2.3. Arquitectura de OpenDayLight

En apartados anteriores hemos visto los diferentes diagramas que representan la arquitectura que conforma el *framework* de OpenDayLight, con todos los módulos, API's y *plugins*. En este apartado vamos a tratar de profundizar un poco en algunos de los componentes de esta arquitectura que son piezas fundamentales para la realización de nuestro proyecto, si bien existen otros muchos componentes que aún no siendo fundamentales en este proyecto, tienen su utilidad en la implementación de otras aplicaciones.

La idea base de OpenDayLight es crear un *framework* multiproveedor, que pueda ser usado en cualquier sistema operativo, siempre que éste soporte JAVA, que es la base sobre la que está construido. En una perspectiva más general, tenemos el siguiente esquema para nuestro framework:

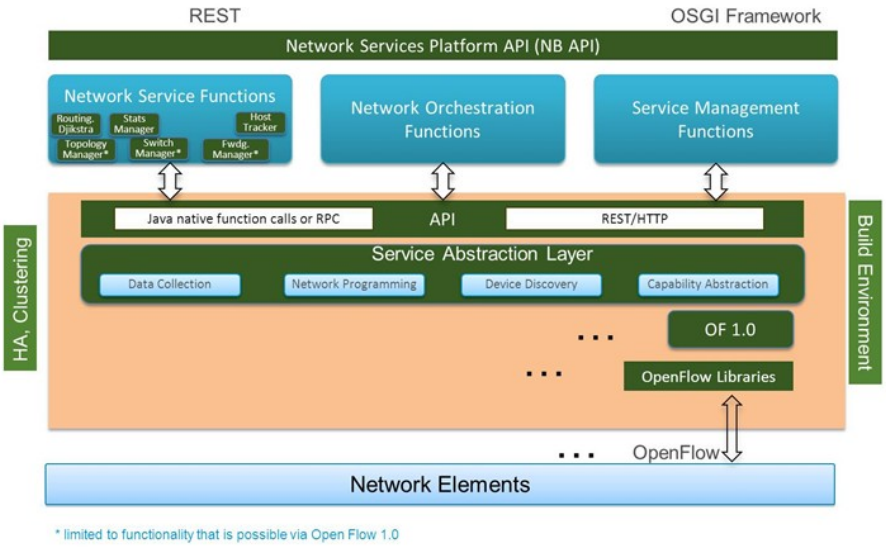


Figura 6.11: Visión general del framework OpenDayLight [20].

Como puede verse en la figura 6.11, obtenida de [20], podemos dividir el *framework* en tres grandes bloques: la SB, la *North Bound* (NB) y la capa de abstracción *Service Abstraction Layer* (SAL). A grandes rasgos, la SB está compuesta por distintos protocolos, entre ellos, y principalmente útil para nuestro proyecto OpenFlow13. Los fabricantes incluyen en el *framework* sus protocolos a través de *plugins*.

Estos *plugins* a su vez están conectados a la capa de abstracción Service Abstraction Layer (SAL) que se encarga de crear las peticiones que los módulos de la *North Bound* (NB) esperan recibir, independientemente del protocolo que se esté usando entre el controlador y los módulos de la SB. Esta abstracción es posible gracias a que el controlador cuenta con un *Topology Manager* (TM) que se encarga de recoger información sobre los dispositivos que se encuentran en el dominio del controlador, por ejemplo, las capacidades de cada dispositivo, su alcanzabilidad y disponibilidad, etc.

En la North Bound (NB) el controlador despliega sus API's para que puedan ser utilizadas por cualquier aplicación que quiera acceder a ellas. Para ello la NB soporta el *framework Open Services Gateway initiative* (OSGi), que está pensado para trabajar con aplicaciones que se encuentren en el mismo espacio de direcciones que el controlador, así como una API REST para aplicaciones que no se encuentren en el mismo dominio que el controlador.

OpenDayLight por tanto supone un *framework* especialmente pensado para que las aplicaciones reúnan toda la inteligencia de la red, puedan obtener estadísticas de red de forma sencilla, y puedan orquestar las nuevas reglas que quieran implementar fácilmente haciendo uso de los módulos que

OpenDayLight pone a su disposición.

6.2.3.1. MD-SAL y comparación con AD-SAL

Como hemos visto en la sección anterior, gran parte de la complejidad del *framework* que representa OpenDayLight se encuentra en la capa de abstracción que conecta los dispositivos de red de la SB con las aplicaciones de la NB. Vamos ahora a analizar el modelo *Model Driven Service Abstraction Layer* (MD-SAL) y a compararlo con su predecesor, *API Driven Service Abstraction Layer* (AD-SAL) para conocer mejor la filosofía que hay detrás de esta capa del controlador.

En las Figuras 6.11 y 6.12, podemos ver las filosofías AD-SAL y MD-SAL respectivamente:

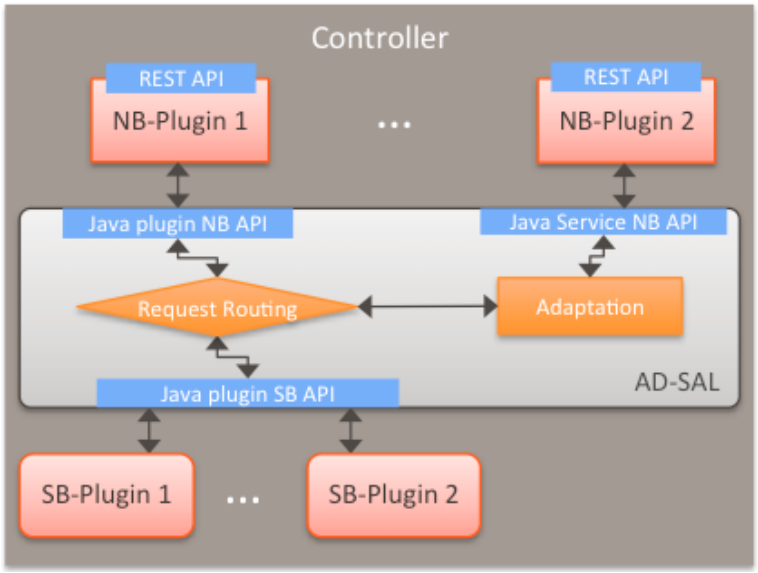


Figura 6.12: Arquitectura AD-SAL [21].

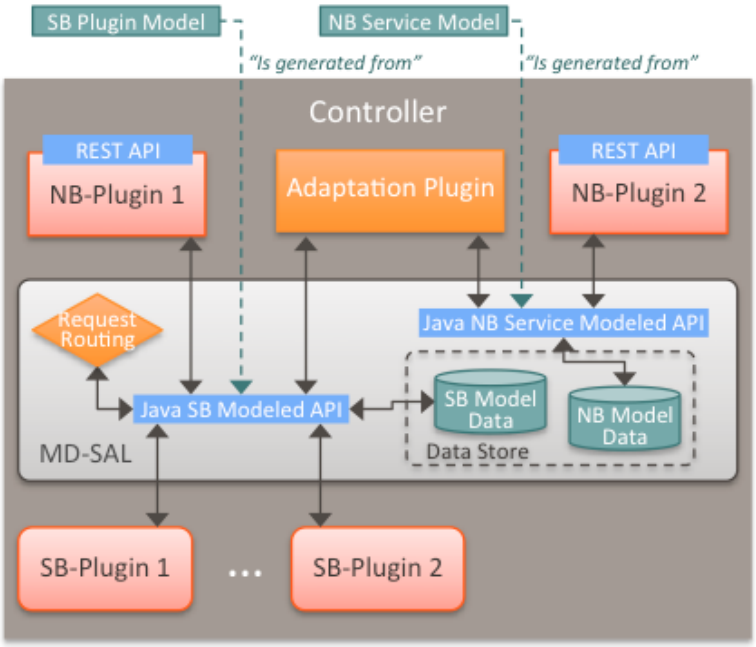


Figura 6.13: Arquitectura MD-SAL [21].

En AD-SAL se ofrece por una parte enrutamiento de peticiones de los servicios abstractos de la NB a los plugins de la SB y también adaptación de servicios, para los casos en los que los protocolos de los servicios de la API NB sean distintos a los de la API de la SB.

En la Figura 6.12 se puede ver un ejemplo en el que la petición de un servicio de la NB es enrutado a través de AD-SAL hacia dos *plugins* de la SB. En ese caso se está partiendo de la base de que los protocolos de ambas API's son los mismos. No ocurriría así en el caso del *Plugin*2 de la NB.

La filosofía de enrutado de peticiones en AD-SAL se sustenta en el hecho de que la capa de abstración SAL conoce las instancias de los nodos que sirven cada *plugin*, de tal forma que cuando un *plugin* de la NB quiere hacer una operación sobre un nodo específico, la petición se redirige hacia el plugin correspondiente el cuál dirigirá la petición al nodo deseado.

En el caso de MD-SAL, la diferencia con respecto a AD-SAL radica en que únicamente se proporciona la infraestructura para soportar la adaptación de servicios, pero no se incluye ningún módulo que realice dicha adaptación, como sí ocurre en AD-SAL.

La adaptación de peticiones entre servicios con diferentes protocolos desde el punto de vista de MD-SAL se realiza a través de un *plugin* que consume y envía información por medio de API's a la capa de abstracción. La labor de un *plugin* de adaptación es básicamente realizar una traducción modelo

a modelo entre dos API's

El enrutado de servicios en Model Driven Service Abstraction Layer (MD-SAL) se realiza tanto por tipo de protocolo como por intancia del nodo, puesto que en este modelo la información de las instancias de los nodos se exporta desde los *plugins* a la capa de abstracción, que contiene un *Data Store*. Concretamente, en la Figura 6.13 dicha información se almacenaría en el *SB Model Data*.

A nivel práctico la diferencia fundamental entre AD-SAL y MD-SAL es que en MD-SAL las API's generadas a partir de los modelos son más sencillas y funcionalmente equivalentes a sus versiones basadas en AD-SAL. Además, el modelo MD-SAL permite almecenar información sobre los modelos generados por los plugins, y los proveedores y consumidores pueden intercambiar la información que se almacena en la capa de abstración.

Otra gran diferencia reseñable es que en AD-SAL existía un mapeo 1:1 entre API's de la NB y SB incluso para aplicaciones con el mismo protocolo. Esto desaparece en MD-SAL, donde si dos *plugins* tienen el mismo protocolo, pueden compartir API, actuando ésta a la vez como API consumidora y API proveedora.

Existen otras pequeñas diferencias entre MD-SAL y AD-SAL, aparte de las ya mencionadas, y que recogemos en la Tabla 6.2 a modo de resumen.

6.2.4. RESTCONF y REST API

Como ya hemos comentado en apartados anteriores, para la comunicación con los *plugins* de la SB así como con los de la NB se utiliza la REST API. Vamos a analizar un poco la estructura y el funcionamiento de esta API.

En primer lugar es necesario definir qué es RESTCONF. Se trata de un protocolo basado en Representational State Transfer (REST) que funciona sobre HTTP y que se utiliza para acceder a datos que han sido definidos por YANG, usando datastores definidos en *Network Configuration Protocol* (NETCONF).

Esta definición introduce otros dos conceptos importantes para comprender qué es y como funciona RESTCONF:

- **YANG :** Se trata de un lenguaje usado para modelar datos, notificaciones y Remote Procedure Call (RPC). En general, YANG se utiliza para generar modelos de datos que son posteriormente pasados a MD-SAL incluyéndolos en ficheros de configuración y cargándolos con Karaf. Con estos datos MD-SAL genera dos de sus *datastores*, *Config Data Store* y *Operational Data Store*.

MD-SAL	AD-SAL
El enrutado de peticiones entre consumidores y proveedores se define en los modelos y la adaptación de datos se realiza a través de *plugins*.	El enrutado de peticiones entre proveedores y consumidores y la adaptación de datos se definen estáticamente en tiempo de compilación.
Se permite que aplicaciones de la SB y NB compartan API generada a partir de un modelo.	Incluso si una aplicación de la SB y otra de la NB tienen un mareo 1:1 y comparten protocolo cada una necesita su propia API.
Hay REST API's comunes para acceder a los datos y funciones definidas en los modelos.	Hay API's dedicadas para cada *plugin* de la *Southbound* y de la *Northbound*.
Se encarga del enrutado de peticiones y cuenta con la infraestructura necesaria para soportar la adaptación de servicios, pero no se encarga de la adaptación.	Se encarga tanto del enrutado de peticiones como de la adaptación de servicios si una API de servicio de la NB tiene un protocolo diferente a su correspondiente API en la SB.
El enrutado de peticiones se realiza según el tipo de protocolo y el tipo de instancia de nodo, ya que la información sobre las instancias se exporta de los *plugin* a la SAL.	El enrutado se basa en el tipo de *plugin*. SAL sabe qué instancia de nodo es servida por qué *plugin*.
Se puede almacenar la información definida por los *plugins*. Proveedores y consumidores pueden intercambiar información a través de MD-SAL *Storage*.	En AD-SAL no existen estados, no se no se almacena información de los *plugins*.
Sólo hay API's asíncronas, pero cuentan con métodos que permiten bloquear las llamadas a métodos hasta que el objeto de entrada a otra llamada se ha recuperado, de forma que una API asíncrona puede usarse en aproximaciones síncronas y asíncronas.	Los servicios tienen una versión asíncrona y otra sincrona de cada API.

Cuadro 6.2: Diferencias entre AD-SAL y MD-SAL

- **NETCONF :** *Network Configuration Protocol* es un protocolo de gestión de red desarrollado por el *Internet Engineering Task Force* (IETF) que se encarga de ofrecer mecanismos de instalación, configuración y borrado de la configuración de dispositivos de red. Este protocolo usa una condificación de la información basada en Extensible Markup Language (XML) para la información de configuración y los mensajes del protocolo.

RESTCONF ofrece una capa de simplicación de NETCONF que permite la compatibilidad con la abstracción para dispositivos orientados a recursos. Además, RESTCONF permite el acceso a las *datastores* definidas por el controlador en la capa de abstracción. Las operaciones soportadas por la RESTCONF son: *OPTIONS*, *PUT*, *GET*, *POST* y *DELETE*.

Como ejemplo de instalación de flujos en la REST API, las Figuras 6.14 y 6.15 muestran el proceso que se seguiría para el borrado y la adición de flujos en un controlador OpenDayLight:

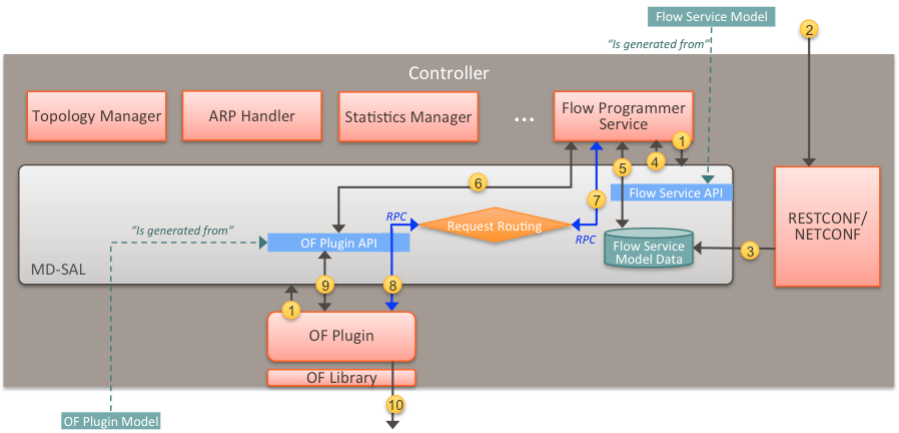


Figura 6.14: Proceso de adición de flujos en la REST-API [21].

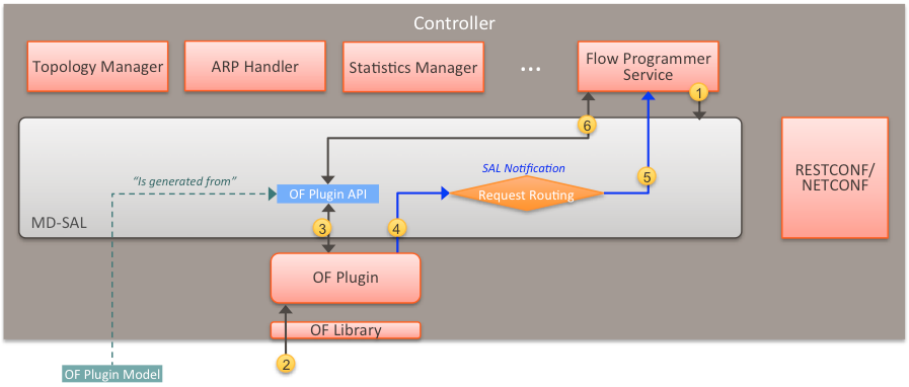


Figura 6.15: Proceso de borrado de flujo en la REST-API [21].

Profundizando un poco más en cada proceso podemos comprender mejor cómo funciona el mecanismo de instalación de flujos de la REST API.

Proceso de adición de flujos.

En la Figura 6.14 podemos observar un ejemplo en el que una aplicación externa instala un flujo por medio de la NB REST API del controlador. El proceso, como se aprecia en dicha figura, consta de 10 etapas, que son las siguientes:

- **Paso 1 :** En primer lugar, cuando el controlador y el *plugin* se inician suceden dos eventos: El *Flow Programmer Service* se registra en la capa MD-SAL para la configuración de flujos y la notificiación de información. El *plugin* OpenFlow registra una implementación de un RPC de tipo "AddFlow" también en la capa de abstracción. Hay que considerar que dicha RPC está definida en el modelo del *plugin* OpenFlow.
- **Paso 2 :** En el segundo paso, un cliente genera una petición de tipo *add* a través de la REST API del controlador. Dicha aplicación aporta todos los parámetros necesarios para realizar la llamada a REST.
- **Paso 3 :** Toda la información enviada en la petición de la aplicación cliente es deserializada y se crea un nuevo flujo en el módulo *Flow Service Model Data*. Una vez que la información llega a este módulo, la aplicación cliente recibe una respuesta exitosa.
- **Paso 4 :** El *Flow Programmer Service*, que está registrado para recibir notificaciones cuando se produzcan cambios de datos en el *Flow Service* recibe una notificación que es generada por MD-SAL.
- **Paso 5 :** El *Flow Programmer Service* lee el flujo de tipo *add* recién añadido y realiza una operación de este tipo.

- **Paso 6 :** En algún punto de la operación de adición de flujo el *Flow Programmer Service* necesitará indicarle al *plugin* OpenFlow que añada el flujo en el dispositivo adecuado. Para ello, el *Flow Programmer Service* usa la API generada por el *plugin* OpenFlow para crear un parámetro de entrada en la RPC del *plugin*.

- **Paso 7 :** El *Flow Programmer Service* obtiene entonces la instancia del servicio sobre el que quiere realizar la llamada para realizar el proceso de adición de flujo. La capa de abstracción en ese momento dirige la petición al *plugin* adecuado.

- **Paso 8 :** Se realiza la llamada RPC indicandose el tipo "*AddFlow*" de la llamada, de forma que el método que implementa éste proceso es invocado.

- **Paso 9 :** La implementación del proceso de adición de flujos usa la API del *plugin* OpenFlow para leer los valores necesarios a su entrada.

- **Paso 10 :** Finalmente, el proceso de adición de flujo es realizado y se envía una modificación de flujo al *switch* correspondiente.

Proceso de borrado de flujos.

En la figura 6.15 se puede ver el proceso de borrado de flujos, que, como puede comprobarse, tiene una complejidad menor que el proceso de instalación de flujos. De igual manera que en el ejemplo vamos a desglosar los pasos necesarios para realizar un borrado de flujos:

- **Paso 1 :** El *Flow Programmer Service* registra la petición de borrado una vez que tanto el controlador como los *plugins* están activos.

- **Paso 2 :** Un paquete OpenFlow de tipo "Flow Deleted" llega al controlador. Esta petición es recibida por la OpenFlow *Library* a través de la conexión TCP/TLS que mantiene con el *switch* que envía la petición y pasa dicha petición al plugin OpenFlow.

- **Paso 3 :** El *plugin* OpenFlow parsea los datos que recibe en la petición y los usa para crear una petición de borrado que envía a la capa de abstracción.

- **Paso 4 :** Una vez que se ha enviado la petición del *plugin* a la capa SAL, en ésta se conduce la notificación hasta los consumidores registrados en el sistema, que en este caso, es el *Flow Programmer Service*.

- **Paso 5 :** El *Flow Programmer Service* recibe y procesa dicha petición.

- **Paso 6 :** Una vez recibida, se usa la API generada por el *plugin* OpenFlow correspondiente para obtener los datos de la petición recibida y se realiza el proceso de borrado.

Con esto concluye el apartado de herramientas, en el que hemos hecho un análisis de las principales funcionalidades y arquitectura de las dos piezas basicas en la implementacion de nuestro escenario de movilidad.

En el siguiente apartado veremos cómo se ha diseñado e implementado cada uno de los componentes necesarios para emular un escenario de movilidad. Posteriormente se dedicará otro capítulo a probar y analizar los resultados obtenidos en las pruebas realizadas durante el proceso de emulación del escenario de movilidad.

Capítulo 7

Diseño e implementación de un escenario de movilidad

En el capítulo que nos ocupa, vamos a abordar el diseño de todas las piezas que forman el puzle de la simulación de un *handover* entre dos puntos de acceso de la que podría ser la futura estructura de red sobre la que se implemente la quinta generación de redes móviles.

Como iremos viendo durante el desarrollo de esta sección, la simulación de un cambio de celda a otra involucra muchos elementos distintos, muchas aplicaciones que deben interactuar entre sí, siguiendo un determinado flujo, y que deben funcionar de manera coordinada como conjunto, sincronizados y de forma que el comportamiento final sea el fijado en nuestros objetivos.

Cuando analizemos el controlador que va a gestionar los flujos de datos de la red, barajaremos dos alternativas que resultan válidas aunque bastante diferentes en cuanto a filosofía de implementación, pero que nos permitirán desarrollar una fase de pruebas mucho más interesante.

Sin más dilaciones vamos a comenzar a analizar la topología de la red escogida y los diferentes elementos que van a formar parte de ella.

7.1. Diseño e implementación de la topología de red

Cuando comenzamos a pensar en el diseño de la posible solución que nos permita realizar una simulación de movilidad, el primer factor que se nos viene a la mente es la topología de nuestra red. Como ya hemos visto en apartados anteriores, donde hemos analizado el funcionamiento de esta herramienta, Mininet cuenta con una serie de escenarios de ejemplo, que podemos utilizar para realizar nuestras primeras pruebas, ver cómo instalar

los flujos de datos para que los elementos de la red se comuniquen entre sí, y analizar cómo se comporta la distribución de interfaces dentro de cada *switch*.

Uno de las primeras tareas será, por tanto, el diseño de una topología que permita tener dos puntos de acceso en *switches* distintos para que un equipo, actuando como UE pueda iniciar una comunicación con el *switch* donde se encuentra el AP al que el UE está conectado, y dar comienzo así al intercambio de mensajes que terminará con el equipo conectado al punto de acceso situado en el *switch* de destino.

Para facilitar las explicaciones siguientes, vamos a recordar la estructura del escenario que se pretende implementar:

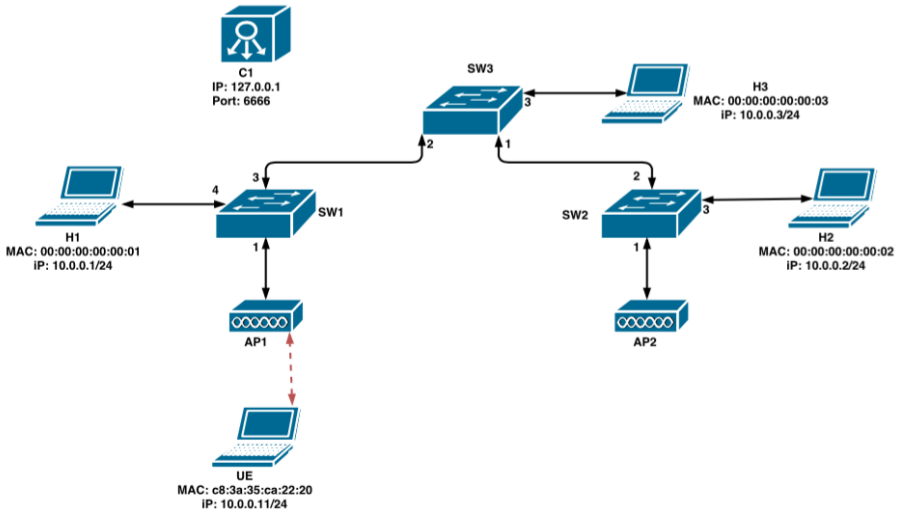


Figura 7.1: Diseño de topología de la red

En la Figura 7.1 podemos ver todos los componentes del escenario. Es importante comentar algunos aspectos relativos a la emulación del escenario. Por ejemplo, algunos elementos existirán físicamente (como es el caso del UE), mientras que otros serán simulados a través de la herramienta Mininet.

Los elementos físicos que formarán parte de la red serán:

- **UE :** Un ordenador portátil Toshiba Satellite A660-13q que hará las veces de equipo que se conecta a unos de los puntos de acceso a través de una tarjeta Wi-Fi inalámbrica Tenda N Dual y realiza la parte del proceso de señalización correspondiente al UE. Contará con una máquina virtual Linux, con la versión de Ubuntu 16.04 LTE, dado que la gestión y configuración de las conexiones de red es mucho más

sencilla en un sistema operativo con base UNIX.

- **AP1 y AP2 :** Estos puntos de acceso se configurarán en las interfaces de los switches 1 y 2 simulados en Mininet y serán dos tarjetas Wi-Fi inalámbricas Tenda N Dual, al igual que la usada para el UE.

El resto de elementos de la red se simulará a través de Mininet, para lo cual usaremos una topología de red customizada que hemos desarrollado en Python y que puede consultarse en el Anexo A de esta memoria.

Una vez diseñada la topología, el último paso para poder probarla y comprobar la conectividad entre todos los elementos de la topología será planificar la instalación de los flujos que permitan que todos los elementos puedan comunicarse entre sí. Para ello, lo primero que necesitamos es, en la consola de mininet, ejecutar el comando "*net*" y comprobar la distribución de los puertos en cada *switch* de nuestra red.

Si observamos la Figura 7.1 comprobaremos que junto a cada enlace de cada switch aparece el número de puerto correspondiente.

La Figura 7.2 muestra cómo se ve la distribución de puertos en cada *switch* a través de la consola de Mininet:

```
ubuntu@sdnhubvm:~/Desktop/TFG_INFO/componentes_topologia_final/topologia
*** Adding controller
*** Adding switch s1
*** Adding wlan0 a switch 1
*** Checking wlan0
*** Adding switch s2
*** Adding wlan1 a switch 1
*** Checking wlan1
*** Adding switch s3
*** Adding eth0 a switch 3
*** Adding hosts*** Adding links***Starting network
*** Configuring hosts
nat0 h1 h2 h3
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
***Starting network
*** Starting CLI:
mininet> net
nat0 nat0-eth0:s1-eth2
h1 h1-eth0:s1-eth4
h2 h2-eth0:s2-eth3
h3 h3-eth0:s3-eth3
s1 lo:  wlan0:  s1-eth2:nat0-eth0 s1-eth3:s3-eth2 s1-eth4:h1-eth0
s2 lo:  wlan1:  s2-eth2:s3-eth1 s2-eth3:h2-eth0
s3 lo:  s3-eth1:s2-eth2 s3-eth2:s1-eth3 s3-eth3:h3-eth0
c0
mininet>
```

Figura 7.2: Estructura de la red generada por Mininet

Un par de detalles que serán importantes a la hora de realizar el cambio de celda y que deberemos tener en cuenta será el hecho de que las interfaces

Wi-Fi siempre se colocan por defecto en la interfaz con menor índice. En nuestro caso siempre se encuentran en el puerto 1 de cada *switch*.

El *switch 1* tiene una diferencia fundamental con respecto al resto de switches, y es que tiene una interfaz que realiza Network Adress Translation (NAT) para permitir que toda la red pueda hacer *ping* hacia Internet. Para ello, habrá que instalar una serie de flujos adicionales y configurar una pasarela en cada switch. En nuestro caso tendrá la dirección IP 10.0.0.6, si bien, en su configuración por defecto, Mininet da a esta interfaz la IP 10.0.0.1 que coincide con la IP del *Host1* de nuestra red.

Para que los elementos de nuestra red puedan comunicarse entre sí, por ejemplo a través de un *ping*, necesitaremos instalar los flujos en los *switches* que permitan que los paquetes se encaminen a los puertos correctos. Las tablas siguientes muestran, por *switch*, los flujos que se deben instalar para que todos los elementos de la red tengan visibilidad entre sí:

Switch	Criterios de match	Prioridad	Acción
Switch 1	dl_type=0x0806	-	ALL
	dl_type=0x0800	10	2
	dl_dst=00:00:00:00:00:01	-	4
	dl_dst=00:00:00:00:00:02	-	3
	dl_dst=00:00:00:00:00:03	-	3
	dl_dst=c8:3a:35:ca:22:20	-	1
Switch 2	dl_type=0x0806	-	ALL
	dl_type=0x0800	10	2
	dl_dst=00:00:00:00:00:01	-	2
	dl_dst=00:00:00:00:00:02	-	3
	dl_dst=00:00:00:00:00:03	-	2
	dl_dst=c8:3a:35:ca:22:20	-	2
Switch 3	dl_type=0x0806	-	ALL
	dl_type=0x0800	10	2
	dl_dst=00:00:00:00:00:01	-	2
	dl_dst=00:00:00:00:00:02	-	1
	dl_dst=00:00:00:00:00:03	-	3
	dl_dst=c8:3a:35:ca:22:20	-	2

Cuadro 7.1: Flujos para el escenario de movilidad

Los criterios de *match* dl_type=0x0806 y dl_type=0x0800 hacen referencia a flujos de tráfico Address Resolution Protocol (ARP) y TCP/IP. Este segundo se utiliza para que, cuando se hagan peticiones a la dirección de NAT, los flujos se redirijan correctamente hacia el *switch* 1. La prioridad 10 indica que es menos prioritario que el resto de flujos del *switch*. Por tanto, cuando se quiera realizar un *ping* a alguna de las direcciones de los *hosts*,

coincidentes con las direcciones MAC de las entradas de flujo, estos flujos serán más prioritarios y se aplicará su acción relacionada.

Una vez que hemos instalado estos flujos, podemos comprobar que existe conectividad entre todos los elementos de la red. Para ello, vamos a realizar una serie de *pings* entre elementos de la red. La Figura 7.3 muestra *pings* realizados en Mininet, entre los hosts virtuales H1, H2 y H3. En la Figura 7.4 vemos un *ping* desde nuestro equipo cliente al *host* H1, comprobando así que el equipo cliente está correctamente conectado al AP1.

Ambas figuras además ponen en evidencia dos hechos que parecen lógicos: el primer *ping* para cada flujo de datos tiene una latencia mucho mayor que los *pings* sucesivos y la latencia entre el equipo conectado al AP1 y los equipos de la red implementada en Mininet es mucho mayor que la latencia entre equipos de la red Mininet.

```
mininet>
mininet> h1 ping -c 2 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=0.583 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.081 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.081/0.332/0.583/0.251 ms
mininet> h2 ping -c 2 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.373 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.065 ms

--- 10.0.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.065/0.219/0.373/0.154 ms
mininet> h3 ping -c 2 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.484 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.070 ms

--- 10.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.070/0.277/0.484/0.207 ms
mininet> h2 ping -c 2 10.0.0.11
PING 10.0.0.11 (10.0.0.11) 56(84) bytes of data.
64 bytes from 10.0.0.11: icmp_seq=1 ttl=64 time=52.7 ms
64 bytes from 10.0.0.11: icmp_seq=2 ttl=64 time=12.8 ms

--- 10.0.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 12.824/32.803/52.783/19.980 ms
mininet>
```

Figura 7.3: Test de conexión entre equipos en Mininet

Al principio, se producirá una inundación de paquetes ARP que provocará que la latencia sea mayor mientras que, a medida que va pasando

el tiempo, la red pasará a un estado estacionario en el que los tiempos de latencia, como resultado, serán mucho menores.

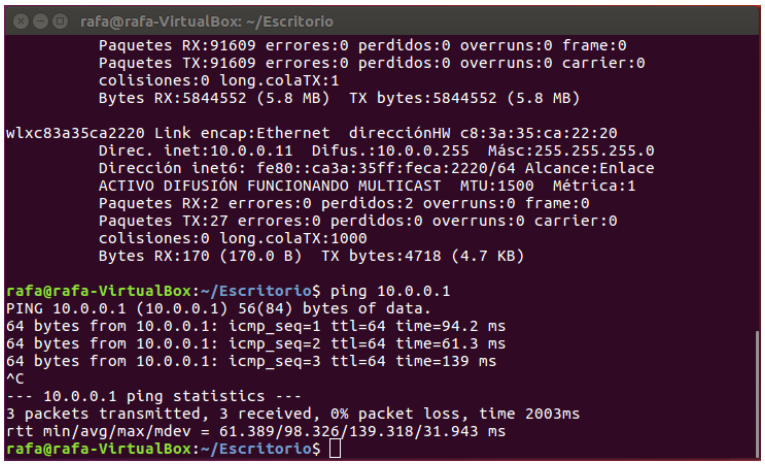


Figura 7.4: Test de conexión entre equipos Mininet y externos

De igual forma, si observamos todos los flujos instalados en los switches, veremos cuántas coincidencias ha tenido cada entrada de flujo, lo que da muestras de la transmisión inicial de paquetes de tipo ARP:

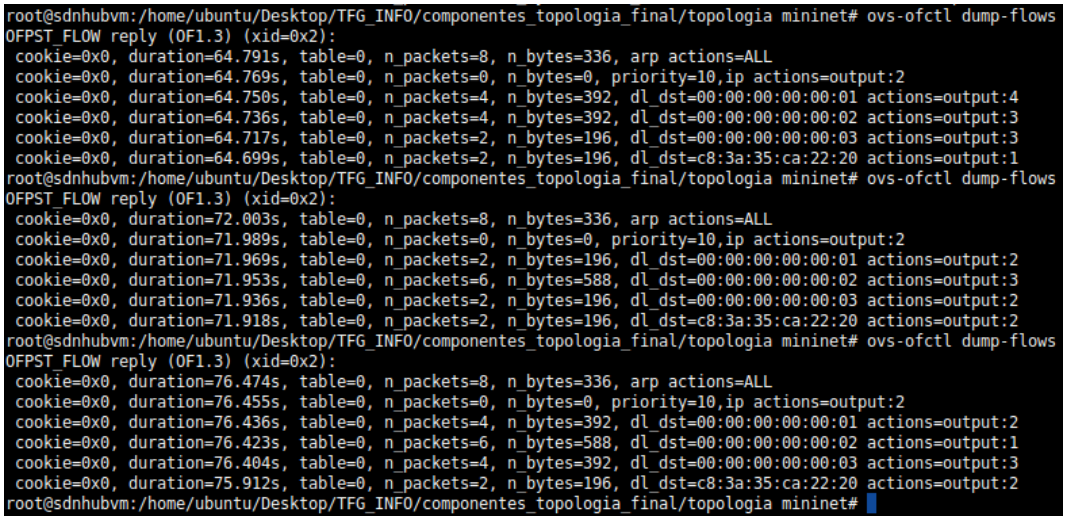


Figura 7.5: Matching entre flujos iniciales del escenario

De esta forma, una vez comprobada la funcionalidad del escenario, podemos pasar al resto de componentes de la simulación. En las subsecciones siguientes nos centraremos en analizar tanto el diseño y desarrollo del protocolo de señalización, como el desarrollo de funciones auxiliares, elementales

a la hora de gestionar procesos tales como la conexión de los puntos de acceso, la configuración de red del cliente y su mecanismo de conexión a cada uno de los puntos de acceso.

7.2. Diseño e implementación del proceso de señalización

El proceso de señalización representa otra de las piezas fundamentales, y que más quebraderos de cabeza nos ha dado a la hora de implementar una solución práctica para un escenario de movilidad. Como ya hemos comentado anteriormente, en [4] se sientan las bases sobre las que se va a trabajar para conseguir una simulación lo más fiel posible al esquema que aparece en la Figura 7.6, que recordamos ahora para facilitar al lector el seguimiento del desarrollo subsiguiente.

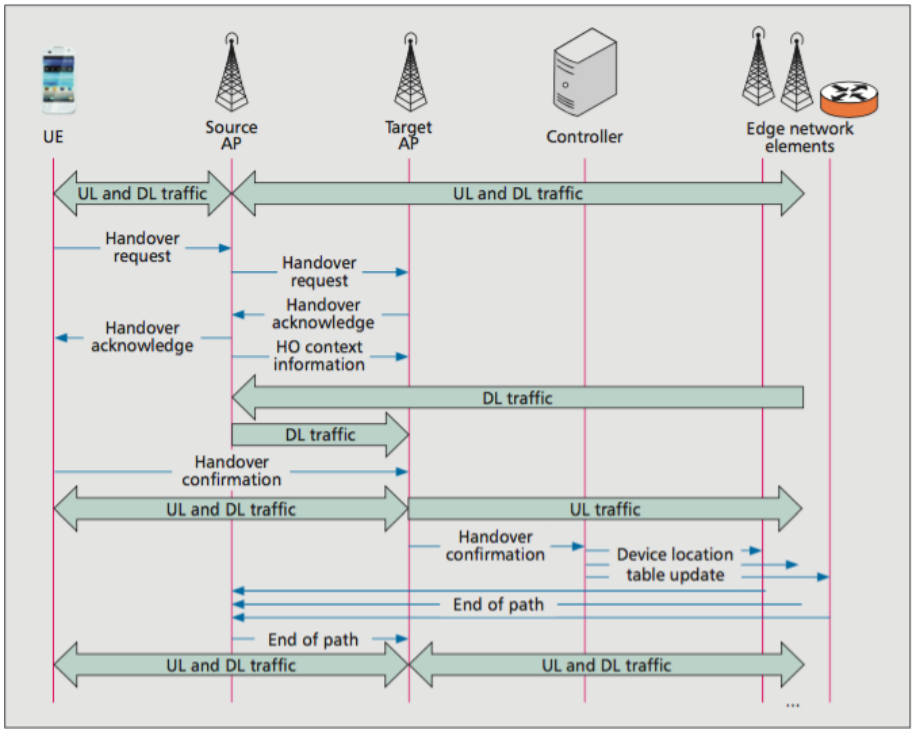


Figura 7.6: Protocolo de señalización para *Handover* en redes 5G [4].

Para realizar el proceso de señalización, vamos utilizar JAVA como lenguaje de programación. La idea básica es crear aplicaciones servidor en las estaciones base, que estén costantemente escuchando mensajes, hasta que una aplicación cliente, representada por el UE inicie el proceso de señaliza-

ción y con ello se inicie el proceso de cambio de celda.

Aplicando la topología de nuestra red al diagrama de la Figura 7.6, podemos hacernos una idea de la distribuciones de roles de los elementos de nuestra red en el proceso de señalización:

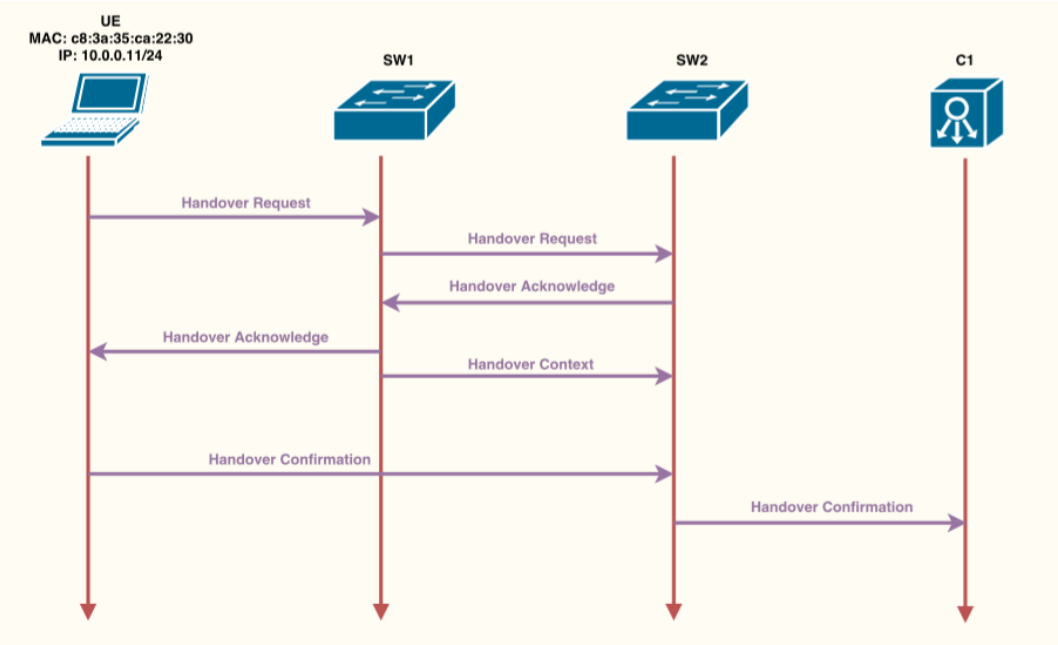


Figura 7.7: Protocolo de señalización para *handover* en redes 5G

Como se ve, el ordenador Toshiba, correrá en su máquina virtual de Ubuntu la aplicación cliente que dará inicio a la conexión. El *switch* 1 albergará la aplicación servidor que representa la parte de la comunicación correspondiente al *source AP* y el *switch* 2 lanzará la aplicación servidor que hará las veces de *target AP*. Finalmente, el controlador OpenDayLight estará costantemente escuchando en busca de mensajes que le indiquen que se debe realizar la instalación de un flujo.

Adicionalmente, crearemos una aplicación servidora que realice una instalación manual de los flujos necesarios para que toda la red se conecte y que también actuará como servidor para, de manera alternativa al controlador integrado en OpenDayLight, permitir al UE realizar un cambio de celda sin necesidad de procesamiento por parte de OpenDayLight. Esta solución nos permitirá, en primer lugar, probar de forma más sencilla el correcto funcionamiento de todo el proceso de señalización y, en segundo lugar, medir el retardo que los elementos hardware de nuestra red generan en el sistema. Así se puede aíslar la carga que posteriormente supondrá el procesamiento de OpenDayLight, pudiendo así realizar medidas más precisas.

Todo es posible gracias a que Mininet permite que cada *switch* tenga su propia terminal de comandos, con los mismos programas que se tengan en el sistema operativo base donde se haya instalado Mininet. De esta forma podemos compilar y ejecutar aplicaciones JAVA como en cualquier otro sistema.

Cada una de las aplicaciones es una *mainClass* de JAVA, que puede ejecutarse por separado sin depender de ninguna clase auxiliar. Para comprender mejor la implementación del proceso de señalización, podemos analizar el comportamiento de cada una de las clases de JAVA que componen dicho proceso, explicando su comportamiento y la estructura de los mensajes que se envían.

7.2.1. Implementación en JAVA

Como ya hemos visto, vamos a necesitar cuatro clases distintas, una aplicación cliente y tres servidoras que estén constantemente escuchando mensajes del resto de los elementos de la red. Todas las aplicaciones, ya sean cliente o servidor, tienen una estructura bastante similar en la que sólo variará la forma en la que procesamos los mensajes recibidos.

El procesado de los paquetes, así como la estructura de los mismo en cada aplicación será expuesta en las secciones siguientes, de forma que quede lo más clara posible la implementación de la señalización de la cara a la emulación del *handover*.

7.2.1.1. Cliente para el User Equipment

El comportamiento general de la aplicación cliente comienza iniciando un proceso de comunicación con el *source* AP. Para ello, utilizararemos datagramas *User Datagram Protocol* (UDP) puesto que la comunicación es secuencial y los mensajes del mismo tipo van secuenciados, así que no necesitamos que haya sincronización entre la petición y la respuesta.

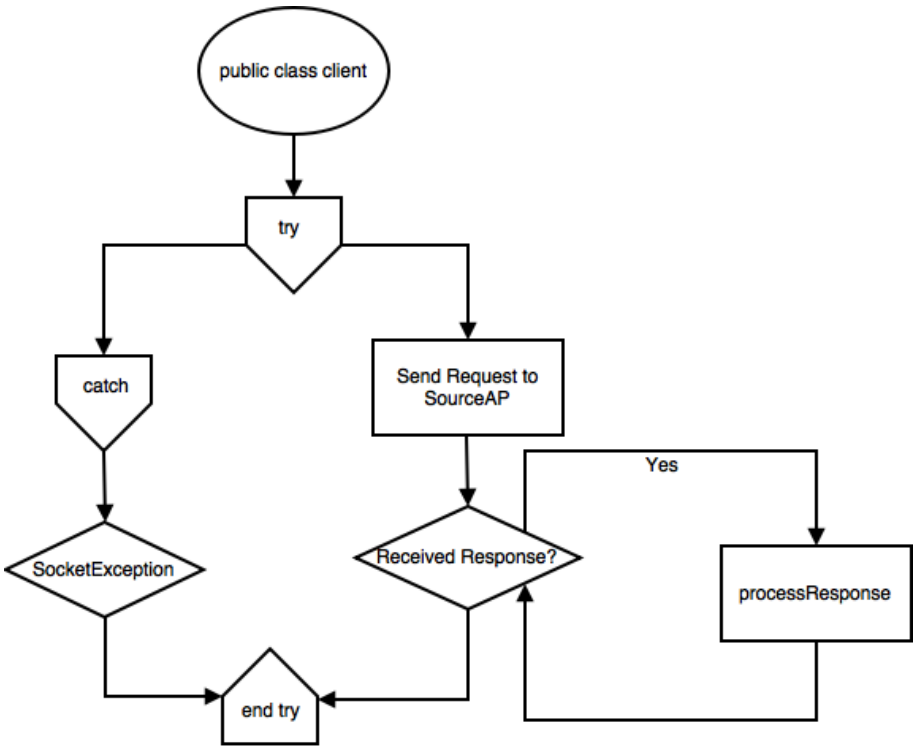


Figura 7.8: Diagrama de flujo de la aplicación cliente

Una vez que el cliente ha iniciado la comunicación se queda esperando a recibir la primera y única respuesta que va a recibir, que será un *Handover Acknowledge* de parte del *source AP*. Cuando se recibe la respuesta, en cada aplicación existe una función auxiliar que parsea el mensaje, procesa la información y genera un mensaje de respuesta en cada caso.

Además de esto, cada aplicación cuenta con una serie de funciones auxiliares que se encargan, entre otras cosas, de convertir los *Strings* que representan cada mensaje a bytes para ser enviados dentro de la parte de datos de los datagramas UDP y ejecutar comandos de la terminal de Linux desde la aplicación.

Estas funciones, que son bastante sencillas y comunes para todas las aplicaciones, tienen la siguiente forma:

```
1 //Function that converts Byte codified information to String:
2 public static String[] byteParser(byte[] bytes){
3         String newString = new String(bytes,StandardCharsets.
              UTF_8);
4         String[] stringArray = newString.split(",");
5         return stringArray;
6 }
```

```
1 //Function that converts a String to Byte codified information:
2 public static byte[] stringToByte(String string){
3         byte[] bytes = string.getBytes(StandardCharsets.UTF_8);
4         return bytes;
5 }
```

```
1  //Function that executes a command with sudo permission:
2  public static void executeCommand(String command) throws
       IOException, InterruptedException{
3          List<String> commands = new ArrayList<>();
4          String[] arguments = command.split(" ");
5          commands.add("sudo");
6          for (String item : arguments){
7                  commands.add(item);
8          }
9          ProcessBuilder pb = new ProcessBuilder(commands);
10         System.out.println("CONTROLLER:: Ejecutando comandos..."
               );
11         Process process = pb.start();
12         int errCode = process.waitFor();
13         if(errCode == 0){
14                 System.out.println("CONTROLLER:: Comando
                       ejecutado con exito");
15         }else{
16                 System.out.println("CONTROLLER:: Error al
                       ejecutar el comando");
17         }
18 }
```

Como entradas, al lanzar la aplicación, el cliente debe recibir su dirección *Media Access Control* (MAC), su dirección Internet Protocol (IP) y la dirección *Internet Protocol* (IP) del *source AP*. En un escenario real, como se expone en [4], este proceso se realizaría a nivel MAC, siendo innecesario el uso de parámetros en el proceso de señalización, pero dada la complejidad que esa implementación conllevaría, adaptamos el proceso para facilitar la emulación.

Los mensajes enviados por la aplicación cliente son los siguientes:

- ***Handover Request*:** Contiene un identificador de mensaje, seguido de un número de secuencia, además de la propia dirección IP y MAC del UE. Este mensaje es transmitido desde el cliente hacia su estación de cobertura, en este caso, directamente a la aplicación servidor alojada en el *switch* 1.
- ***Handover Confirmation*:** Este mensaje contiene el identificador de mensaje, el número de secuencia del mensaje, la dirección MAC y la dirección IP del cliente e irá dirigido al *target* AP, por lo cual,

previamente el *source* AP indica al UE la dirección IP del *target* AP, como veremos en la estructura de ese mensaje.

Una vez que se ha enviado el *Handover Confirmation* al *target* AP, el cliente ejecutará el comando que permitirá que se pase de estar conectado al AP1 a estar conectado al AP2. Este proceso se ejecutará a través de un script *Bash*, que mostraremos más adelante y que se ejecutará a través de la función auxiliar "*executeCommand()*" que se muestra más arriba.

7.2.2. Servidores para el *source* AP y el *target* AP

Por su parte, para el resto de aplicaciones servidoras tendremos la estructura que aparece en la Figura 7.9, que es realmente muy similar al diagrama anterior, con la salvedad de que en este caso no se envía ninguna petición sino que directamente se escucha la llegada de algún mensaje:

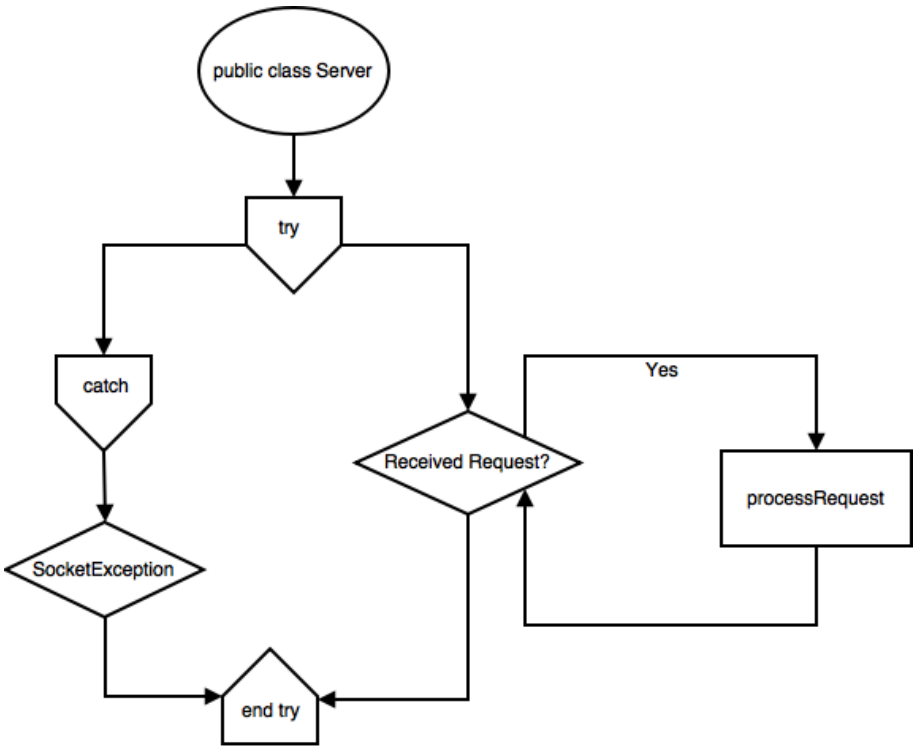


Figura 7.9: Diagrama de flujo de las aplicaciones servidor

En este caso, la función que procesará la llegada de datagramas tendrá una complejidad mayor, sobre todo en el caso del *source* AP, que recibe y envía una cantidad mayor de mensajes. El resto de funciones serán muy

similares a las usadas por el cliente, ya que son funciones auxiliares que no dependen de la lógica detrás de la señalización.

En el caso del *source* AP, que iniciaremos con la IP correspondiente al *target AP*, los mensajes a enviar serán los siguientes:

- ***Handover Request*:** Este mensaje será enviado hacia el *target* AP cuando se reciba un *Handover Request* procedente del UE. Básicamente, se leerán las direcciones MAC e IP que previamente había enviado el UE y se reenviarán al *target AP* con el mismo identificador de mensaje pero incrementando el número de secuencia.
- ***Handover Acknowledge*:** De igual manera que el *Handover Request* supone un reenvío de mensaje, este mensaje será un reenvío hacia el UE de un mensaje procedente del *target* AP. La diferencia en este caso radica en que al mensaje inicial enviado por el *target* AP, que sólo contiene un id y un número de secuencia, se le añadirá la IP del *target* AP para que el UE pueda posteriormente enviar el *Handover Confirmation* que hemos comentado anteriormente. Esta IP no es necesario que sea enviada por el *target* AP, porque el *source* AP puede extraerla del propio datagrama recibido como la IP origen del mensaje.
- ***Handover Context Information*:** Este mensaje, que está sin definir, se utilizaría en caso de necesitar el envio de información extra, de capacidades de los *switches*, o en escenarios de mayor complejidad, para indicar parámetros adicionales. Sin embargo, en nuestro escenario no es necesario que se envíe este tipo de información, por lo que sólo contiene un identificador.

Para el caso del *target* AP no es necesario incluir ningún parámetro inicial, basta con comenzar a escuchar en el puerto elegido y esperar a que llegue el primer mensaje procedente del *source* AP. Esta aplicación recibirá varios mensajes, pero enviará solamente dos:

- ***Handover Acknowledge*:** Este mensaje es respuesta al *Handover Request* recibido a desde el *source* AP. Cuando el *target* AP recibe este mensaje, se almacena la MAC del UE para posteriormente enviar un mensaje al controlador indicando que dicho equipo quiere realizar un cambio de celda. Como en el resto de mensajes, el *Handover Acknowledge* incluirá un *id* de mensaje y un número de secuencia y no añadirá ninguna información más, puesto que el resto se completará cuando el *source* AP reciba el mensaje y lo reenvíe al UE.
- ***Handover Confirmation*:** En este caso se ha recibido un mensaje directamente desde el UE, en el que se indica su dirección IP y su

MAC. El *target* AP, que ya ha almacenado la MAC del UE, comprueba que la recibida ahora coincide con la almacenada y traslada dicha información al controlador para que se puedan realizar las modificaciones pertinentes en los flujos que permitan que el UE siga siendo visible por toda la red.

En el momento en el que el *target* AP ha recibido el *Handover Confirmation*, el UE ya ha comenzado el proceso de cambio de celda. Este proceso es algo más costoso computacionalmente que el proceso de borrado e instalación de nuevos flujos del controlador integrado en OpenDayLight, lo cual permite que el proceso de cambio de celda sea prácticamente transparente para la red.

7.2.3. Controlador integrado en OpenDayLight

La filosofía sobre la que va a implementarse el controlador es la basada en la implementación de un *learning-switch* con OpenDayLight. Esta idea parte de la base de explotar todas las posibilidades que se nos ofrece en nuestro sistema dicho *framework*, donde podemos aprender todos los flujos necesarios para la comunicación inicial de todos los elementos de la red, así como recibir paquetes cuyo destino sea a priori desconocido e instalar flujos para esos paquetes de forma que puedan llegar a su destino.

En este caso, el funcionamiento del controlador es algo más complejo, pues tendrá una fase inicial de descubrimiento de la red. Para que esta fase sea posible, una vez iniciada nuestra topología de red, deberemos variar los flujos inciales instalados, para que, por defecto, cada *switch* que reciba un paquete con una dirección MAC de destino desconocida, lo redirija al controlador.

Estos flujos se pueden instalar con los siguientes comandos:

```
1 ovs-ofctl add-flow -OOpenFlow13 s1 dl\_type=0x0806,priority=2,
      actions=output:all
2 ovs-ofctl add-flow -OOpenFlow13 s2 dl\_type=0x0806,priority=2,
      actions=output:all
3 ovs-ofctl add-flow -OOpenFlow13 s3 dl\_type=0x0806,priority=2,
      actions=output:all
4 s1 ovs-ofctl add-flow tcp:127.0.0.1:6634 -OOpenFlow13 priority
      =1,action=output:controller
5 s2 ovs-ofctl add-flow tcp:127.0.0.1:6635 -OOpenFlow13 priority
      =1,action=output:controller
6 s3 ovs-ofctl add-flow tcp:127.0.0.1:6636 -OOpenFlow13 priority
      =1,action=output:controller
```

Los flujos que establecen la conexión con el controlador deben instalarse desde la consola de Mininet, a diferencia de todos los que habiamos insta-

lado hasta ahora, que podían instalarse desde cualquier terminal de nuestra máquina virtual base. Además, cada conexión establecida debe ir a un puerto distinto, de esta forma el controlador puede diferenciar los *switches* conectados a él.

Por otro lado, el controlador funcionará de acuerdo al diagrama de flujo de la Figura 7.10. La filosofía sobre la que funciona la aplicación que actúa como controlador, si bien algo más compleja, es la misma que la aplicada para las aplicaciones servidoras que hemos visto en apartados anteriores.

Tendremos una aplicación integrada en OpenDayLight que activaremos una vez iniciemos Karaf. En esta aplicación, tras instalar los flujos en cada *switch* que enlazan con el controlador, esperará la llegada de un paquete que contendrá información relativa a la dirección MAC origen y destino del mensaje, así como del identificador del switch del que procede dicho paquete.

Con todo ello, crearemos un mapa con una tabla por cada *switch* que contendrá por *switch*, direcciones MAC y puerto de salida para cada paquete en función de la dirección MAC de destino, de modo que identifiquemos si un flujo ha cambiado y podamos realizar una modificación de flujos, o bien, una instalación de un nuevo flujo que no teníamos almacenado en nuestra tabla.

Si no se encuentra puerto de salida, se llamará a una función que realizará una inundación del paquete en todos los puertos salvo el puerto de entrada. Si por el contrario se encuentra una salida para la MAC de destino para el *switch* del que se haya recibido el paquete en nuestro mapa de tablas se procederá a instalar el flujo en los *switches* y a generar el paquete de salida hacia el *switch*.

Esta solución, unida a una pequeña modificación en la aplicación que gestiona el *target* AP, permitirá realizar un cambio de celda de forma transparente. La clave en este caso reside en, conocida la MAC origen del UE que va a realizar el cambio de celda, realizar un borrado de los flujos relativos a ese equipo en todos los *switches*, de forma que cuando se quiera conectar de nuevo con el UE en su nueva celda, el *target* AP no tendrá flujo definido para la MAC asociada al UE y tendrá que intercambiar mensajes con el controlador, que detectará diferencias entre los flujos almacenados para dicha MAC con respecto a los recibidos ahora, de forma que modificará los flujos y los volverá a instalar creando la nueva ruta hacia el UE.

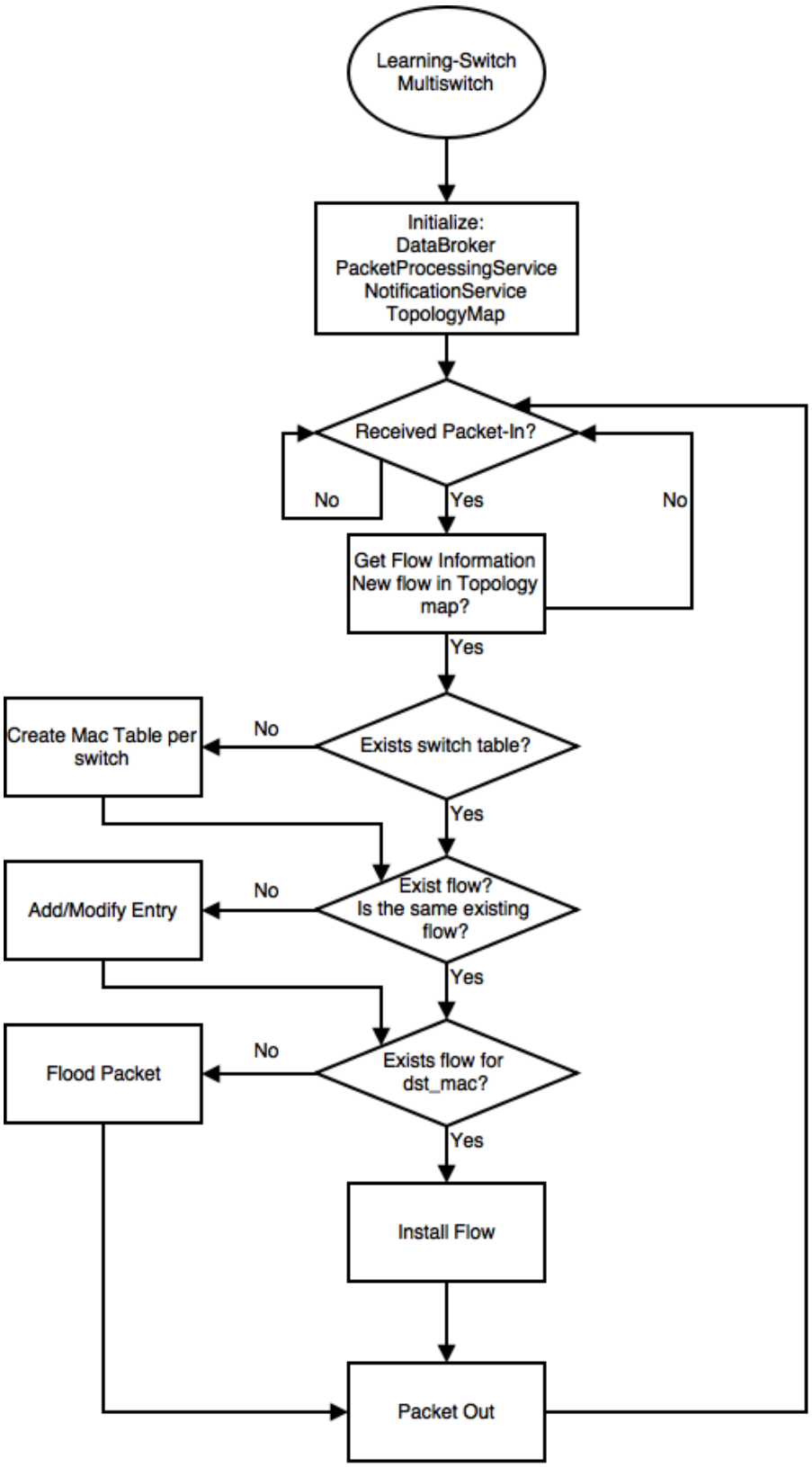


Figura 7.10: Solucion basada en OpenDayLight

7.2.4. Instalador de flujos

Una solución parcial para simular el comportamiento del controlador a expensas del protocolo OpenFlow es la representada por una aplicación servidora que realice una instalación manual de flujos a través de comandos del protocolo OpenFlow. En este caso, la estructura de la aplicación es bastante similar a la de las aplicaciones servidoras analizadas anteriormente, con alguna modificación.

Incidimos de nuevo en el hecho de que esta solución será utilizada para realizar medidas de referencia que caractericen el retardo creado por el hardware usando para la emulación de nuestro escenario, orientado por tanto, a la realización más exacta de medidas y análisis de rendimiento del sistema.

La Figura 7.11 muestra la estructura básica de dicha aplicación:

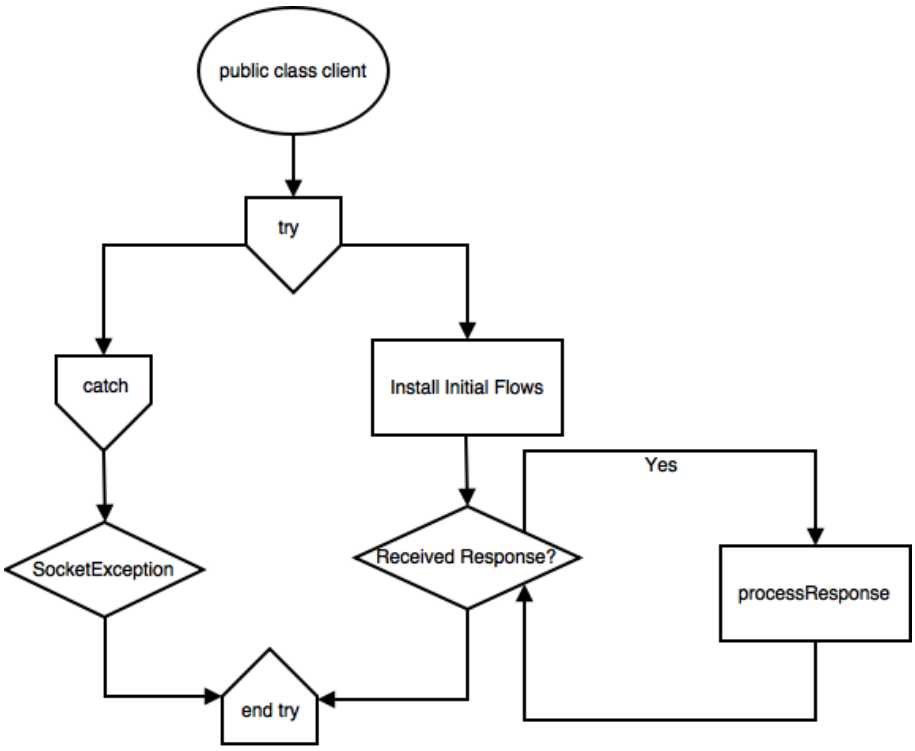


Figura 7.11: Diagrama de flujo de la aplicación controlador

Como vemos, en el caso de esta aplicación, habrá una parte del código dedicada a la instalación de los flujos iniciales, para lo cual se leerá un fichero en formato CSV que contiene todos los flujos iniciales que se deben instalar para que la red sea funcional. Estos flujos contienen información fundamental para el instalador, ya que por Media Access Control (MAC)

indican el puerto de salida de cada *switch*. De esta forma, a la vez que el controlador instala los flujos a través de la función "*executeCommand()*" que hemos visto anteriormente, también se va creando un mapa de la topología que relaciona *switch* con MAC de destino y con puerto de salida.

Cuando el controlador recibe el *Handover Confirmation* de parte del *target* AP, lee la dirección MAC del cliente, que estará almacenada en el *switch* 1. Además, conoce la ruta en cada *switch* para ir al *switch* 2 y también sabe que las interfaces Wi-Fi correspondientes a los puntos de acceso se localizan siempre en el menor puerto del *switch*.

Con esta información, el controlador primero borra en todos los *switches* las reglas que cumplieran la condición de que la MAC de destino fuera la del UE, para después instalar en cada *switch* una nueva regla que indique el nuevo camino para llegar al UE.

Con todo ello, una vez finalizado el proceso el UE podrá seguir comunicandose con normalidad con el resto de la red, como veremos en la fase de pruebas.

7.3. Funciones auxiliares

Ya hemos analizado la parte encargada de crear la topología y la parte que permite realizar el proceso de señalización, pero nos quedan algunos flecos que gestionar y que son igualmente necesarios para conseguir que el proceso se complete correctamente.

En primer lugar, necesitamos iniciar karaf para activar los módulos correspondientes a la aplicacion controladora que hemos implementado y poder realizar la instalación de los flujos, y en general, el proceso de *handover*. Basta con ejecutar el comando:

```
1 cd SDNHub\_Opendaylight\_Tutorial/distribution/opendaylight-
    karaf/target/assembly
2 sudo ./bin/karaf
```

Una vez iniciado, podemos iniciar nuestra topologia en Mininet como se ha visto en apartados anteriores. Ahora bien, en esta topología, al *switch* 1 se le asigna la interfaz *wlan0* y al *switch* 2 la correspondiente interfaz *wlan1*, pero dichas interfaces no están configuradas. Para configurarlas, cada interfaz tiene asociado un script *bash* y un fichero de configuración como los siguientes:

```
1 #!/bin/bash
2 #Script bash for Access Point configuration
3 INTERFACE=$1
```

```
cp /etc/hostapd/hostapd1.conf.backup /etc/hostapd/hostapd1.conf
sed -i 's/wlan0/'${INTERFACE}'/g' /etc/hostapd/hostapd1.conf
ifconfig ${INTERFACE} 10.0.0.4/24
hostapd -d /etc/hostapd/hostapd1.conf
```

```
#Configuration file
interface=wlan0
driver=nl80211
ssid=sdn_ap1
hw_mode=g
channel=1
```

El *script bash* recibe como parámetros el nombre de la interfaz que el sistema le asigne a nuestra tarjeta Wi-Fi, crea una copia de seguridad y, en caso de que haya cambiado la interfaz asignada a la tarjeta con respecto a la anterior ejecución, sustituye en el fichero de configuración el nombre de la interfaz. Finalmente, se le asigna una IP a la interfaz y se carga la configuración del punto de acceso a través del paquete *hostapd*, que hemos descargado previamente.

De igual manera, en nuestro equipo que actúa como UE necesitamos un par de *scripts* para realizar la configuración iniciar la configuración inicial de la tarjeta Wi-Fi y para conectarnos a uno u otro punto de acceso en función de lo que queramos en cada momento.

```
#!/bin/bash
#Script para configurar la interfaz wifi
if [ "$#" -ne 1 ]; then
        echo "Sintaxis: $0 <interfaz wifi>"
        exit 1
fi
INTERFACE=$1
sudo ifconfig ${INTERFACE} 10.0.0.11/24 up
sudo route add default gw 10.0.0.4
```

```
#!/bin/bash
#Script para conectarse de forma persistente a un Access Point
if [ "$#" -ne 1 ]; then
        echo "Sintaxis: $0 <interfaz wifi>"
        exit 1
fi
INTERFACE=$1
# Desconectamos (por si estuviese conectado previamente)
sudo iw ${INTERFACE} disconnect
STATUS="checking"
while [ "${STATUS}" != "done" ]
do
        desconectado=`iwconfig ${INTERFACE} | grep "Not-
```

```
            Associated" | wc -l`
14          if [ "$desconectado" -eq 1 ]; then
15                  echo "Correctamente desconectado..."
16                  STATUS="done"
17          else
18                  echo "Esperando 0.5 seg para comprobar la
                        desconexion de nuevo (STATUS=${STATUS})..."
19                  sleep 0.5
20          fi
21  done
22  sudo rfkill unblock all
23  sudo iwconfig ${INTERFACE} essid sdn_ap1
24  sleep 1
25  # Comprobamos que se ha conectado, o esperamos en caso contrario
        .
26  STATUS="checking"
27  while [ "${STATUS}" != "done" ]
28  do
29          desconectado=`iwconfig ${INTERFACE} | grep "Not-
                Associated" | wc -l`
30          if [ "$desconectado" -ne 1 ]; then
31                  echo "Conectado!"
32                  STATUS="done"
33          else
34                  echo "Desconectado, esperando 1 seg para
                        reintentar la conexion..."
35                  sudo rfkill unblock all
36                  sudo iwconfig ${INTERFACE} essid sdn_ap1
37                  sleep 1
38          fi
39  done
```

Con todo ello, ya tenemos todos los componentes necesarios para realizar una emulación completa de un *handover*. En el siguiente apartado analizaremos el funcionamiento de las soluciones implementadas y realizaremos una pequeña comparación entre ambas, tanto en rendimiento como en escalabilidad.

Capítulo 8

Fase de pruebas y evaluación

Una vez implementado el escenario, es la hora de comprobar su funcionalidad y eficiencia. El hecho de tener dos implementaciones distintas nos permitirá analizar el caso ideal, donde la sobrecarga de datos enviados es la mínima posible (caso del instalador manual de flujos) y compararla con la implementación de un controlador en OpenDayLight.

La solución basada en la instalación de flujos no supone una alternativa al controlador integrado en OpenDayLight, por factores tales como la escalabilidad, el aprendizaje de rutas de encaminamiento a través del procesado de paquetes, o simplemente, por no estar integrado en la arquitectura SDN. Nos permite, no obstante, hacernos una idea del comportamiento ideal que debe tener un sistema como el que queremos desarrollar y ser capaces de medir cuánto retardo añade una solución basada en OpenDayLight al propio retardo generado por los elementos que usamos en nuestra red para realizar el cambio de celda, por ejemplo, el tiempo que tardan las tarjetas Wi-Fi en conectarse.

8.1. Prueba de funcionalidad: señalización y topología

Tanto en el caso del instalador de flujos como en del controlador integrado en OpenDayLight y basado en un *learning-switch* para redes con múltiples *switches*, el proceso de señalización será similar. Por tanto, una de las primeras comprobaciones que debemos hacer es la del correcto funcionamiento de la red y de las aplicaciones para comprobar que la señalización se produce correctamente.

Los pasos para la realización de una prueba pasan por iniciar todos los componentes que forman parte de la emulación, a saber: tarjetas Wi-Fi,

configuradas con los *scripts* mostrados en el capítulo anterior, la topología de Mininet, y las aplicaciones que actuan como servidores, clientes e instalador de flujos o controlador, dependiendo del caso.

La distribución de interfaces de la red creada será que se aprecia en la Figura 44:

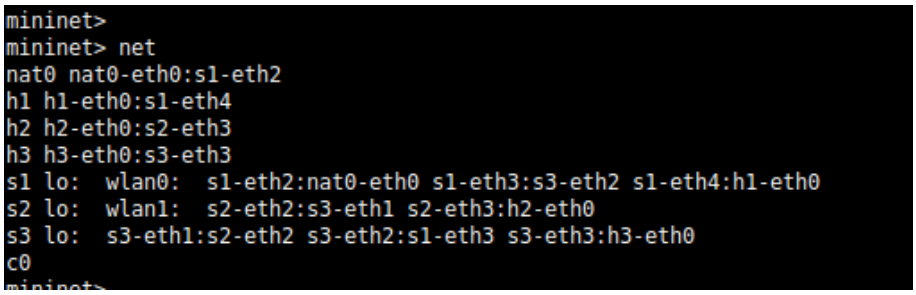


Figura 8.1: Topología de red del escenario implementado

Una vez montado el entorno, el siguiente paso será el de configurar nuestras interfaces, para poder elegir la IP a la que va a apuntar cada *switch* o cliente, de forma que puedan enviarse, a través de datagramas UDP, mensajes relativos a la señalización e intercambiar la información necesaria para ejecutar correctamente el cambio de celda. En caso de dudas sobre qué dirección IP es la usada por alguno de los equipos que integran nuestra red, la solución pasa por realizar un *ping* desde ese equipo hacia cualquier otro y analizar las trazas con Wireshark. En todo este apartado, Wireshark será nuestro mejor aliado para comprobar el funcionamiento y rendimiento del sistema.

Con independencia de la solución que estemos ejecutando, en nuestro sistema siempre tendremos una aplicación correspondiente al *source*s AP ejecutada en el *switch* 1 y escuchando el primer mensaje del UE, otra aplicación correspondiente al *target* AP ejecutándose en el *switch* 2, esperando el primer mensaje del *source* AP y una aplicación cliente que representa al UE y que será la parte proactiva de la señalización.

Tendremos por tanto dos aplicaciones escuchando cada una en su correspondiente dirección IP y puerto, que una vez comience el proceso de señalización (cuando lancemos la aplicación correspondiente al UE) se intercambiarán una secuencia de mensajes como la que aparece en la Figura 8.2:

Time	Source	Destination	Protocol	Length	Info
190.737026	10.0.0.11	10.0.0.40	UDP	75	3333 → 4444 Len=31
190.737343	10.0.0.11	10.0.0.40	UDP	75	3333 → 4444 Len=31
190.737347	10.0.0.11	10.0.0.40	UDP	75	3333 → 4444 Len=31
190.741477	10.0.0.6	10.0.0.6	UDP	1044	4444 → 5555 Len=1000
190.744913	10.0.0.6	10.0.0.6	UDP	47	5555 → 4444 Len=3
190.753931	10.0.0.6	10.0.0.11	UDP	56	4444 → 3333 Len=12
190.753941	10.0.0.6	10.0.0.11	UDP	56	4444 → 3333 Len=12
190.754097	10.0.0.6	10.0.0.11	UDP	56	4444 → 3333 Len=12
190.754676	10.0.0.6	10.0.0.6	UDP	47	4444 → 5555 Len=3
190.822189	10.0.0.11	10.0.0.6	UDP	75	3333 → 5555 Len=31
190.822207	10.0.0.11	10.0.0.6	UDP	75	3333 → 5555 Len=31
190.822210	10.0.0.11	10.0.0.6	UDP	75	3333 → 5555 Len=31
190.822915	127.0.0.1	127.0.0.1	UDP	65	5555 → 6666 Len=21

Figura 8.2: Traza de mensajes de señalización

En este caso, se ha usado para el ejemplo el instalador de flujos, que escucha en el puerto 6666. En el caso del controlador integrado en OpenDayLight, esos mensajes se enviarán a la ruta por defecto instalada en cada *switch*.

Los datagramas UDP intercambiados llevan los mensajes que se envían todos los elementos de la red codificados en bytes, uno por cada caracter usado en el *String* que representa dicho mensaje. Por ello, no podemos seguir la secuencia basándonos en la información intercambiada. Sin embargo, sí podemos comprobar el desarrollo de la secuencia de mensajes sabiendo que el UE escucha en el puerto 3333, el *source* AP lo hace en el 4444, el *target* AP en el 5555 y el instalador de flujos en el 6666.

Para una trazabilidad más clara, las Figuras 8.3, 8.4 y 8.5 muestran, respectivamente, la respuesta a los eventos recibidos en el *source* AP, en el *target* AP y en el instalador:

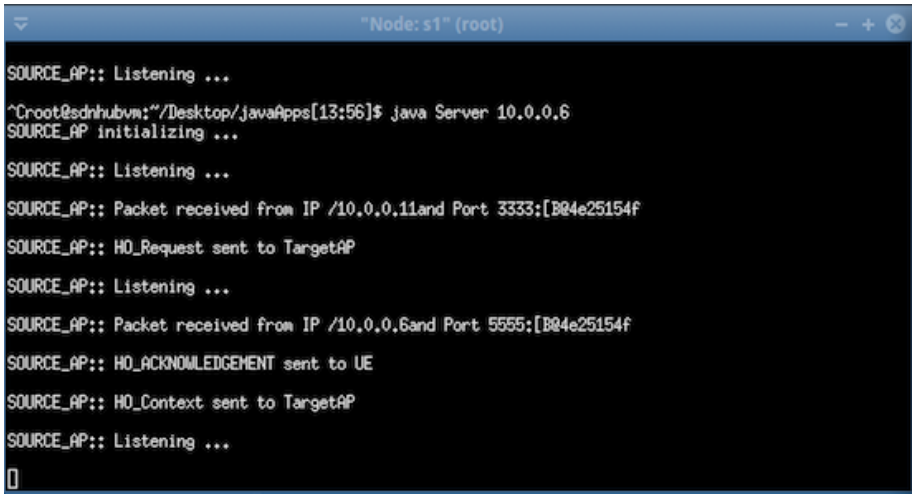


Figura 8.3: Eventos recibidos en el *source* AP

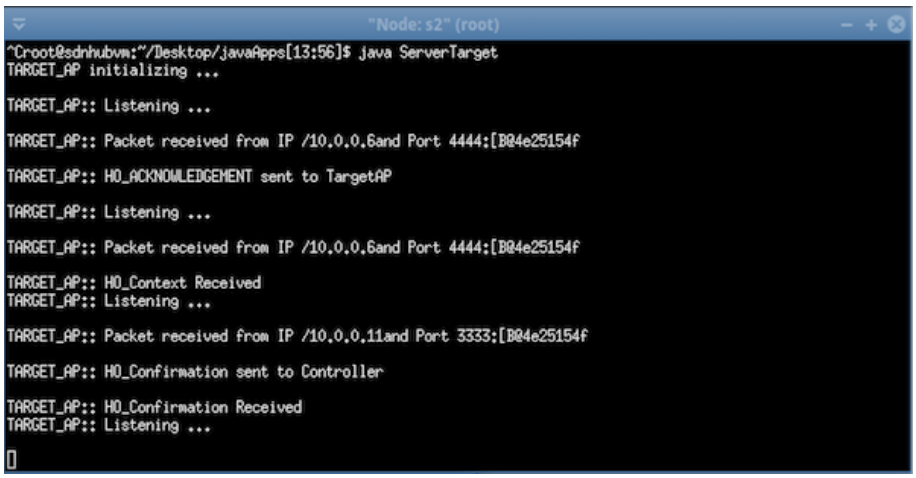


Figura 8.4: Eventos recibidos en el *target* AP

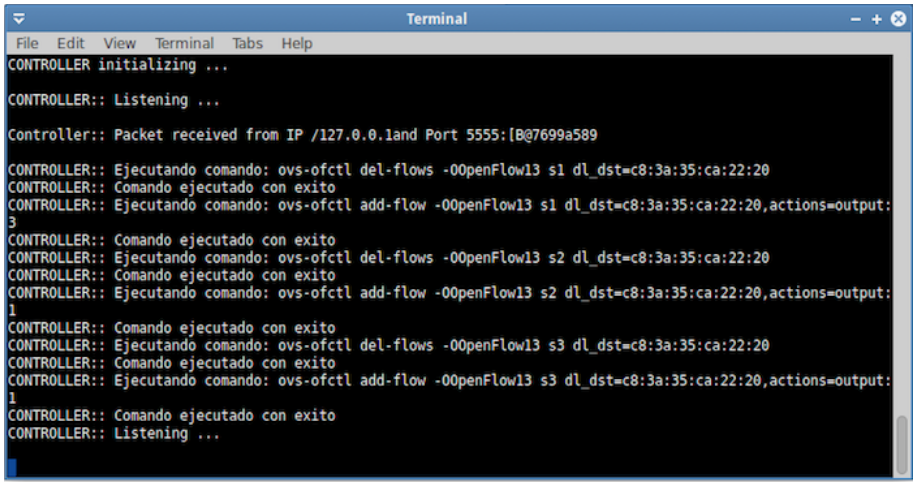


Figura 8.5: Eventos recibidos en el instalador

De igual forma, el UE recibe, procesa, y ejecuta las acciones correspondientes, como vemos en la Figura 8.3:

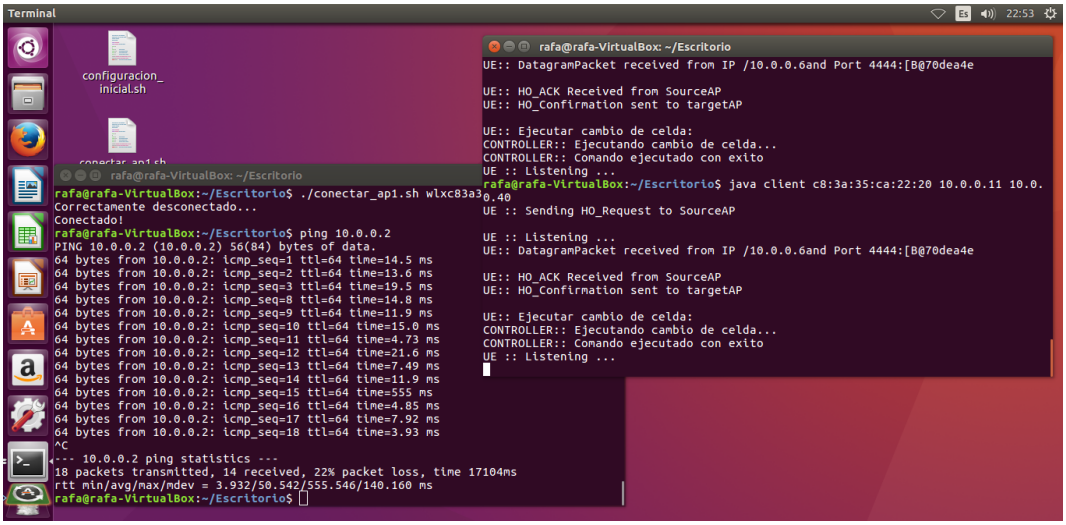


Figura 8.6: Eventos recibidos en el instalador

Comprobamos así que nuestra señalización funciona de la forma esperada, que todos los elementos de la red pueden comunicarse entre sí a través de una IP y un puerto conocidos, y que por tanto, puede intercambiar información relativa a la red, sus direcciones IP o sus direcciones MAC, para poder controlar el reenvío de flujos.

8.2. Pruebas de rendimiento y funcionalidad

En un entorno controlado, como el que estamos desarrollando, parece lógico diseñar unas pruebas que no introduzcan factores externos que puedan modificar los resultados obtenidos. Por ello, en este apartado vamos a llevar a cabo dos pruebas que nos permitan medir la velocidad de reenvío de paquetes y de descarga de ficheros durante el proceso de cambio de celda.

Las siguientes secciones están destinadas a analizar el rendimiento y funcionalidad del sistema, identificando los momentos en los que tiene lugar el cambio de AP y viendo su influencia en el envío de paquetes.

8.3. *Ping* entre UE y *hosts*

El tipo más básico de pruebas que podemos realizar es el *ping* entre equipos. Teniendo conectado el UE al AP1 y ejecutando un cambio al AP2. Durante el proceso, habrá un pequeño intervalo de tiempo en el que los paquetes se pierdan, hasta que se establezca la conexión con el AP de destino.

Es dificil determinar cuánta incidencia tiene en ello el proceso de señalización y cuánta es debida a las limitaciones hardware de nuestro escenario. Para ello, la implementación que usa el instalador de flujos nos servirá de referencia, puesto que marcará el tiempo de referencia con el que comparar para obtener el tiempo que se añade debido al procesado de paquetes.

Realizaremos una serie de tres simulaciones por cada solución, tanto para el instalador de flujos como para el controlador, de forma que obtengamos la media y veamos cuánto se incrementa el *Round Trip Time* (RTT) a partir de las estadísticas que nos ofrece el *ping*. Los resultados se reflejan en las Figuras 8.7 y 8.8

Iteración	RTT mínimo (ms)	RTT medio (ms)	RTT máximo (ms)	Desviación media (ms)	Paquetes perdidos
1	3,543	12,783	53,004	12,099	5
2	3,886	10,577	27,291	6,397	4
3	4,782	13,469	52,607	13,156	5
Media	**4,070**	**12,276**	**44,301**	**10,551**	**4,667**

Figura 8.7: Instalador de flujos: Resultados de *ping*.

Iteración	RTT mínimo (ms)	RTT medio (ms)	RTT máximo (ms)	Desviación media (ms)	Paquetes perdidos
1	6,826	71,172	1044,504	189,640	5
2	4,978	39,472	63,264	14,452	7
3	26,469	52,148	105,896	19,594	6
Media	**12,758**	**54,264**	**404,555**	**74,562**	**6,000**

Figura 8.8: Controlador: Resultados de *ping*.

Como vemos, en el caso de la solución basada en el controlador integrado con OpenDayLight, se añade un retardo del orden de 3 veces superior al caso ideal, representado por los resultados obtenidos para el caso de instalador de flujos. Si bien la pérdida de paquetes es bastante similar, aumentando en uno sólamente.

De igual forma, podemos comprobar el comportamiento del *ping* a lo largo del tiempo, analizando los paquetes correspondientes al protocolo *Internet Control Message Protocol* (ICMP), donde podremos observar el intervalo en el que no pueden enviarse paquetes debido al cambio de celda para ambos casos:

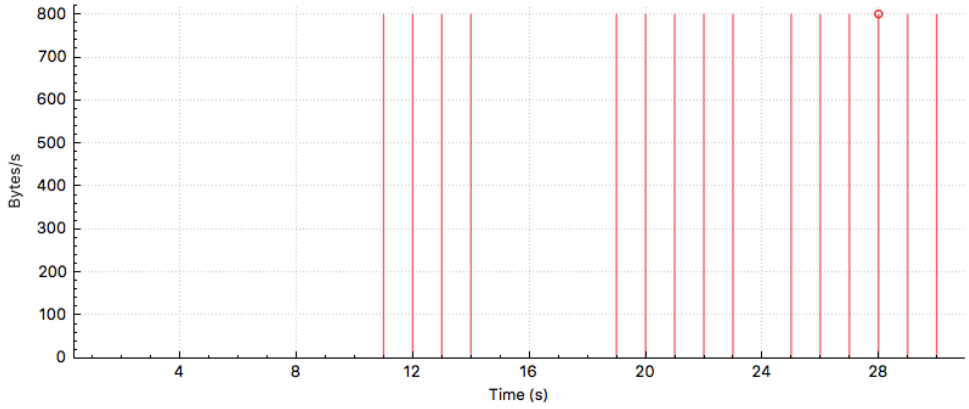


Figura 8.9: Instalador de flujos: *Handover* durante *ping*.

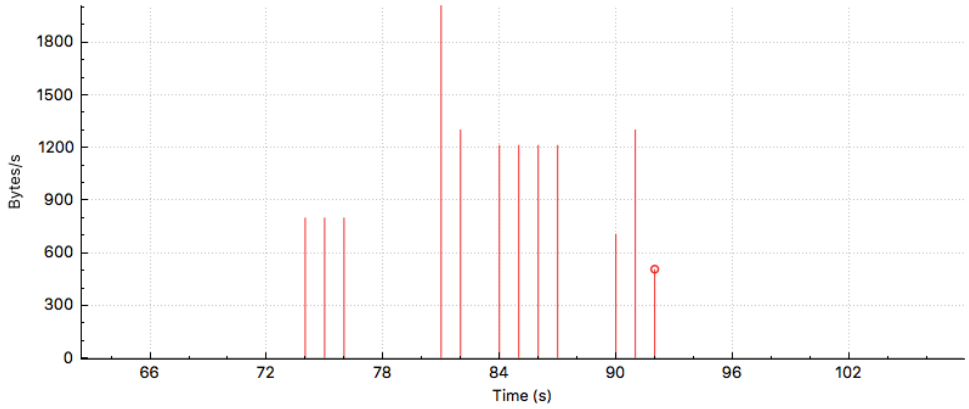


Figura 8.10: Controlador: *Handover* durante *ping*.

En este caso, el comportamiento del *ping* sólo se ve afectado durante una franja temporal muy acotada, en la que la pérdida de paquetes es mínima, y no se aprecian diferencias notables entre el comportamiento ideal y el comportamiento del sistema integrado en OpenDayLight, salvo por el envío de una mayor cantidad de información.

8.4. FTP entre UE y *hosts*

Una segunda prueba algo más compleja, dedicada a evaluar cómo afecta el controlador con OpenDayLight a la transferencia de ficheros, será el intercambio de un archivo generado con datos aleatorios entre el UE y el *host* 3. En este caso, haremos una prueba sin realizar *handover*, y otra realizando

handover para cada solución, de forma que tratemos de localizar el momento en que dicho cambio de AP tiene lugar y veamos si influye en el desarollo del resto de la transferencia.

En las Figuras 8.11, 8.12 y 8.13 vemos el resultado de las pruebas realizadas para los tres casos comentados:

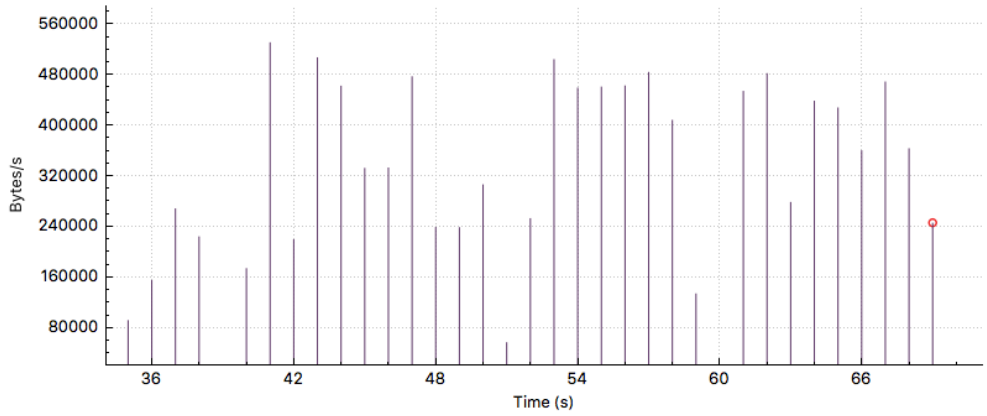


Figura 8.11: FTP sin *handover*

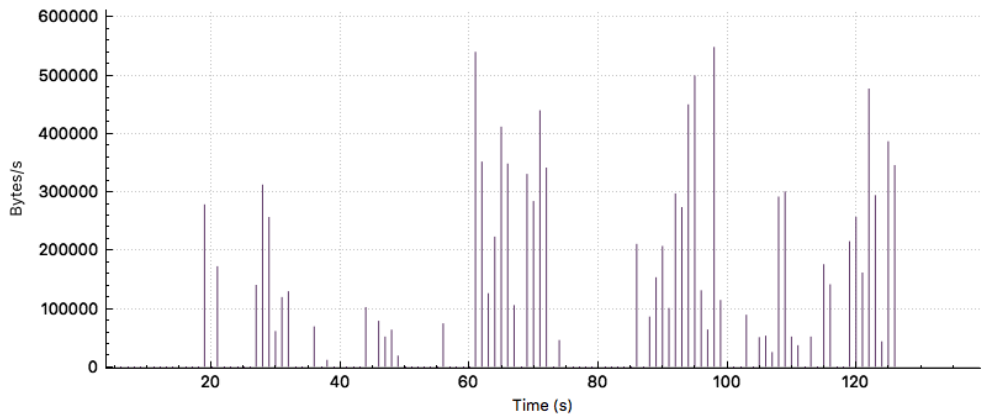


Figura 8.12: Instalador de flujos: *Handover* durante FTP

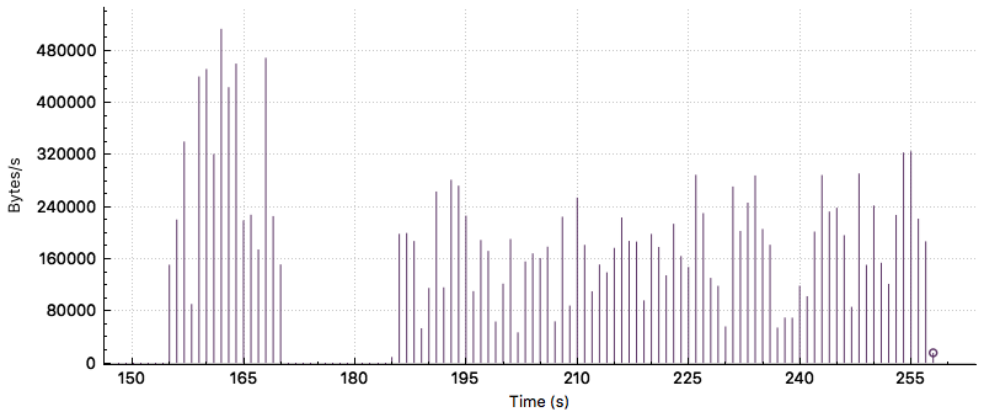


Figura 8.13: Controlador: *Handover* durante FTP

Hay que tener en cuenta, que la velocidad de transferencia entre un punto de acceso y otro puede variar en función de las condiciones de cada punto de acceso. Durante los experimentos realizados hemos podido comprobar que la velocidad de transferencia de un fichero mediante FTP a través del AP2 es ligeramente menor que la velocidad de transferencia alcanzada por el AP1, lo cual se ve claramente en el caso del *handover* realizado a través del controlador, puesto que hay un periodo lo suficientemente largo de transferencia a través del AP1 como para poder comparar dicha velocidad con la alcanzada por el AP2.

En el caso básico, donde no se realiza ningún cambio del celda sino que solamente se realiza una conexión FTP a través del AP1, el intercambio de ficheros se realiza con total normalidad. Observamos cómo fluctúa la velocidad de transferencia, entre los 80KBytes y los 500KBytes, sin ninguna interrupción aparente del proceso en ningún punto de la transmisión.

Los dos casos más interesantes son los representados por las Figuras 8.12 y 8.13. Analicemos primero el comportamiento en el caso ideal, en el que realizamos un cambio de punto de acceso con el instalador manual de flujos. En este caso, la descarga del fichero comienza durante el segundo 20, pasando rápidamente a realizar el cambio de celda, en torno al segundo 22. Vemos cómo se detiene el flujo de mensajes FTP durante un periodo bastante reducido de tiempo, entorno a unos 6 segundos, tiempo tras el cuál se continúa con el flujo normal de intercambio de paquetes. Parece que durante el resto de la conexión hay algunos momentos en los que apenas existe intercambio de información, lo cual nos hace sospechar que el tiempo durante el cuál la conexión está interrumpida puede camuflarse con una caída en la transmisión de paquetes.

En el caso del controlador integrado con OpenDayLight la transferencia comienza en torno al segundo 155. En este caso, los picos de transferencia

más altos se alcanzan en el intervalo en el que se está realizando la transferencia a través del AP1, para, en el segundo 171 realizar un cambio de celda, que se nota claramente en el flujo de paquetes intercambiados entre *host3* y UE. Como vemos, el intercambio de paquetes permanece interrumpido durante un periodo de tiempo más prolongado que en el caso anterior, en torno a 15 segundos, tiempo durante el cuál el controlador procesa los mensajes recibidos desde los *switches*, que no saben ahora hacia dónde dirigir los paquetes que tienen como destino la MAC del UE.

La transferencia en este caso se realiza en 102,43 segundos, un tiempo algo menor que el registrado en el caso anterior, por lo que podemos concluir que el tiempo de transferencia no se ve afectado por la utilización de este controlador con respecto al caso ideal representado por el instalador de flujos, si no que más bien dependerá del hardware empleado y de las condiciones de la red en el momento en el que se realiza la conexión.

8.5. Evaluación de resultados

Tras la realización de las pruebas mostradas en el apartado anterior, se pueden obtener varias conclusiones interesantes, que vamos a recoger en este apartado.

En primer lugar, en términos de eficiencia, los resultados obtenidos para las diferentes pruebas realizadas indican que el rendimiento obtenido cuando se utiliza un controlador con una solución basada en OpenDayLight son muy similares a los obtenidos para el caso ideal. En un principio podría pensarse que el procesamiento de paquetes realizador por el controlador podría percibirse de forma negativa en el rendimiento del escenario, pero los resultados demuestran que su utilización es prácticamente imperceptible y se ve enmascarada por el resto de elementos hardware y de red que afectan al rendimiento del sistema.

La pérdida mínima de paquetes supone que se pueda simular un escenario de movilidad de forma transparente para el usuario, que es ajeno al proceso de modificación de flujos que se está desarrollando. Las conexiones activas entre el UE y el resto de elementos de la red se mantienen como tal y la descarga de ficheros se completa de igual manera que en una sesión sin movilidad.

En términos de escalabilidad, un sistema basado en OpenDayLight es la solución definitiva, puesto que gestiona la instalación de tráfico de forma transparente, independientemente del número de *switches* que compongan nuestra red, mientras que el instalador de flujos iría aumentando su complejidad de forma paralela al crecimiento de la red. Es por ello que desde un principio sólo hemos tratado el instalador de flujos como una referencia

de cara a la realización de medidas y comprobaciones de funcionalidad del sistema.

La consideración de escalabidad es de vital importancia, pues permite ahorrar, como se ha comentado en los primeros capítulos de esta memoria, costes en equipamiento específico de un fabricante de telecomunicaciones y construir un hardware genérico y más sencillo que ceda el control de la red a una inteligencia exterma centralizada sin penalizar a la red en modo alguno.

Si bien son muchos los conceptos sobre los que hay que seguir trabajando, y mucho desarrollo el que queda por realizar para aplicar esta filosofía a un entorno de red real, este proyecto supone una pequeña muestras de las posibilidad que nos ofrecen las redes definidas por *software* y el framework *OpenDayLight* para la gestión de movilidad en las futuras redes de telecomunicaciones.

Capítulo 9

Conclusiones y vías futuras

Realizado el trabajo, tras mucho estudio, muchas pruebas, muchos fallos, y muchos quebraderos de cabeza, este apartado está destinado a analizar el desarrollo del proyecto, analizar las contribuciones del mismo al resto de trabajos que se desarrollan en la misma linea de investigación y experimentación, y sentar las bases para posibles trabajos futuros que supongan una ampliación o mejora del que nos ocupa.

9.1. Conclusiones

Antes de empezar esta aventura, lo único que teníamos claro era el objetivo que se deseaba conseguir: partir de un estudio teórico sobre cómo debía desarrollarse y gestionarse la movilidad en una red 5G y conseguir realizar una emulación utilizando para ello el concepto tantas veces mencionado a lo largo de esta memoria de SDN.

Trabajar con herramientas como Mininet y OpenDayLight implica una fase de comprensión y una abstracción a conceptos complejos como la virtualización de redes y su interacción con elementos físicos reales que en ocasiones ha resultado muy costosa en cuanto a tiempo y esfuerzo se refiere. A su favor hay que decir que siempre que necesitas realizar algún cambio, o implementar alguna función que no conoces, la documentación es abundante y no faltan los foros en los que gente como nosotros comete los mismos errores y se plantea las mismas preguntas, lo cual es a la vez tranquilizador y una importante ayuda.

A la vista de los resultados, las ventajas obtenidas en el empleo de un controlador implementado a través de OpenDayLight parecen claras frente a otras formas de abarcar la solución al problema de la movilidad en las futuras redes 5G. Acogiéndonos a los objetivos del proyecto, conseguir una implementación parecida a la planteada al comienzo del proyecto, usando

varias soluciones para ello y creando las muchas aplicaciones y *scripts* necesarios para llevar a cabo el proceso de movilidad es motivo de satisfacción.

Además de hacer hincapié en el hecho de alejarnos del plano teórico para llevar a cabo una emulación de la gestión de movilidad, también se ha planteado la estructura que pueden tener los mensajes en el posible protocolo de señalización utilizado para este proceso pues, como ya comentamos, en los estudios en los que se basa este proyecto se define la secuencia de mensajes, pero no el contenido de los mismos.

Si bien, no es hasta llegado el momento de la implementación cuando de verdad eres consciente de la información que necesitas y de la forma en que esa información debe llegar a los distintos elementos de la red para que sea útil y permita a cada aplicación cumplir con su cometido.

Podemos concluir con que este proyecto es una primera aproximación a un escenario de simulación real, con los recursos limitados de los que disponemos, y teniendo en cuenta que son nuestros primeros pasos con una tecnología que tiene una complejidad notable y un potencial mucho mayor. Encontramos a lo largo de la memoria un equilibrio entre carga práctica y teórica, pues son muchos los detalles de implementación que se debían explicar, de igual forma que muchos los conceptos que se consideran necesarios para facilitar al lector el acercamiento a los conceptos sobre los que se levanta este proyecto.

9.2. Vías futuras

En pro de la continuidad de este proyecto en futuros trabajos relacionados con la movilidad en redes 5G, se propone una implementación en un entorno real, si bien sería un proyecto de una magnitud enorme, que se enriquecería de una colaboración entre varios alumnos para posibilitar la configuración de todos los equipos involucrados en la consecución del proceso de cambio de estación base.

Una implementación real nos daría una idea mucho más exacta de la viabilidad de la aplicación de las redes definidas por software en entornos controlados, donde se pudiera hacer un estudio detallado del rendimiento del procesado de los paquetes enviados y de los beneficios obtenidos del empleo del *framework* OpenDayLight.

Otra posible linea de estudio sería la de unir este proyecto con otros proyectos realizados en el departemento de Teoría de la Señal, Telemática y Comunicaciones. Por ejemplo, el realizado por Bárbara Valera, que trata de reducir el impácto de la inundación de determinados tipos de paquetes en redes definidas por software controladas igualmente por un controlador implementado en OpenDayLight. Sería muy interesante aplicar ese filtrado de

paquetes a nuestro proyecto, de manera que se redujera la carga de paquetes enviada por el controlador y se favoreciera el incremento del rendimiento en escenarios de movilidad.

9.3. Valoración personal

Un trabajo de esta envergadura supone un desafío para cualquier alumno que está terminando sus estudios en ingeniería de telecomunicación. Al fin y al cabo, lo principal que se espera de un ingeniero, de cara al mundo laboral, es que sea capaz de familiarizarse de forma rápida con tecnologías nuevas, que sea flexible y adaptable y que busque soluciones a las posibles contingencias que surjan durante la realización de un proyecto.

Este trabajo de fin de grado ofrece todas esas posibilidades y desafíos. A veces ha resultado frustrante ver que faltaban muchas piezas por encajar, mucho por implementar, y muchos cambios por hacer sobre lo ya implementado. Finalmente, como en cualquier objetivo que nos fijamos, las cosas tienden a funcionar, con mayor o menor esfuerzo.

En mi caso personal, este proyecto se ha prolongado más de lo deseado, al solaparse con la incorporación al mundo laboral, que ha supuesto otra vía de esfuerzo y aprendizaje paralela, y que también ha tenido connotaciones positivas, respecto a la mejora experimentada en conocimientos de programación y de utilización de sistemas Linux.

Recordaré este proyecto como una experiencia enriquecedora, tanto tecnolgica como formalmente hablando, pues es la única oportunidad que tenemos durante la carrera de acercarnos a realizar un estudio pormenorizado de un proyecto, atendiendo a la estructura y contenidos que deben acompañar a la parte más práctica a la que todos estamos acostumbrados.

Apéndice A

Anexo A: Topología en Mininet

Presentamos en este anexo la topología usada para la emulación del escenario de movilidad descrito en varios capítulos de esta memoria.

En esencia, se trata de un script realizado en Python que importa una serie de paquetes de Mininet necesarios, entre otras cosas, para dotar a los equipos de la capacidad de ejecutar comandos con la versatilidad de cualquier sistema Linux que tan útil nos va a resultar de cara a la simulación del handover.

Tambien se importan paquetes para implementar NAT, indicar el tipo de controlador usado, el tipo de switches, las interfaces y los enlaces empleados, etcétera.

La topología crea un escenario con tres switches conectados en forma de árbol, con un controlador remoto, y con interfaces inalámbricas conectadas a los dos switches del nivel más bajo del árbol y que serán configuradas como puntos de acceso en cada switch.

Se puede apreciar de igual manera la asignación de direcciones IP y MAC de la topología, que se intenta que sea lo más sencilla posible para faciliar la implementación y prueba de la conexión entre todos los equipos.

```
1 from mininet.net import Mininet
2 from mininet.cli import CLI
3 from mininet.node import Controller, RemoteController,
      OVSController
4 from mininet.node import CPULimitedHost, Host, Node
5 from mininet.node import OVSSwitch, Controller, RemoteController
6 from mininet.node import IVSSwitch, OVSKernelSwitch
7 from mininet.log import setLogLevel, info, error
8 from mininet.link import Intf, TCLink
```

```
9  from mininet.util import quietRun
10 from mininet.nodelib import NAT
11 from mininet.topolib import TreeNet
12
13 import re
14 import sys
15
16 def checkIntf( intf ):
17     "Make sure intf exists and is not configured."
18     if ( ' %s:' % intf ) not in quietRun( 'ip link show' ):
19         error( 'Error:', intf, 'does not exist!\n' )
20         exit( 1 )
21     ips = re.findall( r'\d+\.\d+\.\d+\.\d+', quietRun( 'ifconfig
           ' + intf ) )
22     if ips:
23         error( 'Error:', intf, 'has an IP address,' 'and is
               probably in use!\n' )
24         exit( 1 )
25
26 def myNetwork():
27         net=Mininet(controller=RemoteController, switch=
               OVSKernelSwitch, ipBase='10.0.0.0/8', listenPort
               =6634, link=TCLink)
28
29         info( '*** Adding controller\n' )
30         c0 = net.addController( name='c0', controller=
               RemoteController, protocols='OpenFlow13', ip='
               127.0.0.1')
31
32         info( '*** Adding switch s1\n' )
33         s1 = net.addSwitch( 's1', cls=OVSSwitch, mac='
               00:00:00:00:00:10', protocols='OpenFlow13' )
34         info( '*** Adding wlan0 a switch 1\n' )
35         intfName= 'wlan0'
36         info( '*** Checking', intfName, '\n' )
37         #checkIntf( intfName )
38         Intf( intfName, node=s1 )
39
40         info( '*** Adding switch s2\n' )
41         s2 = net.addSwitch( 's2', cls=OVSSwitch, mac='
               00:00:00:00:00:20', protocols='OpenFlow13' )
42         info( '*** Adding wlan1 a switch 1\n' )
43         intfName= 'wlan1'
44         info( '*** Checking', intfName, '\n' )
45         #checkIntf( intfName )
46         Intf( intfName, node=s2 )
47
48         info( '*** Adding switch s3\n' )
49         s3 = net.addSwitch( 's3', cls=OVSSwitch, mac='
               00:00:00:00:00:30', protocols='OpenFlow13' )
50         info( '*** Adding eth0 a switch 3\n' )
51         #Intf( 'eth1', node=s3 )
52
53         net.addNAT().configDefault();
```

```
54
55          info( '*** Adding hosts')
56          h1 = net.addHost( 'h1', mac='00:00:00:00:00:01', ip='
                10.0.0.1/24', defaultRoute='via 10.0.0.4' )
57          h2 = net.addHost( 'h2', mac='00:00:00:00:00:02', ip='
                10.0.0.2/24', defaultRoute='via 10.0.0.4' )
58          h3 = net.addHost( 'h3', mac='00:00:00:00:00:03', ip='
                10.0.0.3/24', defaultRoute='via 10.0.0.4' )
59
60          info('*** Adding links')
61          net.addLink( s3, s2 )
62          net.addLink( s3, s1 )
63          net.addLink( s1, h1 )
64          net.addLink( s2, h2 )
65          net.addLink( s3, h3 )
66
67          info('***Starting network\n')
68          net.build()
69          net.start()
70
71          info('***Starting network\n')
72          for controller in net.controllers:
73                  controller.start()
74          net.get('s1').start([c0])
75          net.get('s2').start([c0])
76          net.get('s3').start([c0])
77          CLI(net)
78          net.stop()
79
80  if __name__ == '__main__':
81          setLogLevel( 'info' )
82          myNetwork()
```

Bibliografía

[1] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[2] A. F. Cattoni, P. E. Mogensen, S. Vesterinen, M. Laitila, L. Schumacher, P. Ameigeiras, and J. J. Ramos-Munoz, "Ethernet-based mobility architecture for 5g," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, Oct 2014, pp. 449–454.

[3] J. Vehanen, "Handover between lte and 3g radio access technologies: Test measurement challenges and field environment test planning," G2 Pro gradu, diplomityö, 2011. [Online]. Available: http://urn.fi/URN:NBN:fi:aalto-201207022686

[4] P. Ameigeiras, J. J. Ramos-munoz, L. Schumacher, J. Prados-Garzon, J. Navarro-Ortiz, and J. M. Lopez-soler, "Link-level access cloud architecture design based on sdn for 5g networks," *IEEE Network*, vol. 29, no. 2, pp. 24–31, March 2015.

[5] Y. Kim, H.-Y. Lee, P. Hwang, R. K. Patro, J. Lee, W. Roh, and K. Cheun, "Feasibility of mobile cellular communications at millimeter wave frequency," *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 3, pp. 589–599, 2016.

[6] R. R. Fontes, S. Afzal, S. H. B. Brito, M. A. S. Santos, and C. E. Rothenberg, "Mininet-wifi: Emulating software-defined wireless networks," in *Network and Service Management (CNSM), 2015 11th International Conference on*, Nov 2015, pp. 384–389.

[7] S. Kukliński, Y. Li, and K. T. Dinh, "Handover management in sdn-based mobile networks," in *2014 IEEE Globecom Workshops (GC Wkshps)*, Dec 2014, pp. 194–200.

[8] "Openflow overview," accedido 15-05-2017. [Online]. Available: URL{https://noviflow.com/the-basics-of-sdn-and-the-openflow-network-architecture/}

[9] I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov, "Security in software defined networks: A survey," vol. 17, pp. 1–1, 01 2015.

[10] "Mininet walkthrough," accedido 23-05-2017. [Online]. Available: URL{http://mininet.org/walkthrough}

[11] "Gns3 documentation," accedido 23-05-2017. [Online]. Available: URL{https://docs.gns3.com/1FFbs5hOBbx8O855KxLetlCwlbymTN8L1zXXQzCqfmy4/index.htmlssssssss}

[12] "Netvirt project," accedido 13-07-2017. [Online]. Available: URL{https://www.slideshare.net/sdnhub/opendaylight-app-development-tutorial}

[13] "Opendaylight yum repository," accedido 15-07-2017. [Online]. Available: URL{https://nexus.opendaylight.org/content/repositories/opendaylight-yum-fedora-19-x86_64/rpm/opendaylight-release/0.1.0-2.fc19.noarch/opendaylight-release-0.1.0-2.fc19.noarch.rpm}

[14] "Opendaylight vm installation," accedido 05-07-2017. [Online]. Available: URL{http://sdnhub.org/tutorials/openflow-1-3/}

[15] "Opendaylight set-up," accedido 15-07-2017. [Online]. Available: URL{https://wiki.opendaylight.org/view/Release/Hydrogen/Base/Installation_Guide}

[16] "Network slicing concept," accedido 16-07-2017. [Online]. Available: URL{https://www.ericsson.com/en/networks/topics/network-slicing}

[17] "Odl releases index," accedido 16-07-2017. [Online]. Available: URL{https://www.opendaylight.org/technical-community/getting-started-for-developers/downloads-and-documentation}

[18] "Karaf overview," accedido 16-07-2017. [Online]. Available: URL{https://karaf.apache.org/manual/latest/overview.html}

[19] "Karaf overview," accedido 16-07-2017. [Online]. Available: URL{http://karaf.apache.org/documentation.html}

[20] "Netvirt project," accedido 17-07-2017. [Online]. Available: URL{https://wiki.opendaylight.org/view/OpenDaylight_Controller:Architectural_Framework}

[21] "Opendaylight architecture overview," accedido 11-07-2017. [Online]. Available: URL{http://docs.inocybe.com/dev-guide/content/_opendaylight_controller_md_sal_faqs.html}

[22] L. Barreto Flórez, "Telefonía móvil 2g," Universidad Incca de Colombia, Tech. Rep., 2012.

[23] E. Martínez, "La evolución de la telefonía móvil," *articulo publicado en la revista RED*, 2001.

[24] J. Scourias, "Overview of the global system for mobile communications," *University of Waterloo*, vol. 4, 1995.

[25] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.

[26] "Genius project," accedido 17-07-2017. [Online]. Available: URL{https://wiki.opendaylight.org/view/Genius:Main}

[27] "Netvirt project," accedido 17-07-2017. [Online]. Available: URL{https://wiki.opendaylight.org/view/NetVirt}