ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Implementación de una plataforma redundante de control *SDN-IoT*

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Autor: José Daniel Padrón Pérez

Tutores: Dr. Carlos Miguel Ramírez Casañas

Fecha: 14 de julio de 2020

ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Implementación de una plataforma redundante de control *SDN-IoT*

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: Juan Domingo Sandoval González

Vocal **Secretario**

Fdo: Carmen Nieves Ojeda Guerra Fdo.: José Ramón Velázquez Monzón

Fecha: 14 de julio de 2020

Agradecimientos

En primer lugar quisiera agradecer a toda mi familia por haberme apoyado en todo momento en todas las decisiones que he tomado y por estar ahí cuando realmente lo necesitaba. A mí madre por siempre estar ahí y por la valentía de criar a un hijo prácticamente sola. A mis abuelos Celestino, Juana, Manuel y Kenia por criarme cuando era un niño y ayudarme a sentar las bases de lo que soy hoy como persona. A mis tíos Leo y Desi por ser como unos padres y amigos durante mi adolescencia y ayudarme a superar mis inseguridades. A Matías por comportarse como un padre durante todos estos años.

En segundo lugar, a toda la gente que he conocido estos años en la universidad. Comenzando por las primeras amistades que tuve: Fer, Carlitos, Manu, Antonio, Dayron y Miguel Miguel. Luego, siguió el Erasmus y aparecieron Jejeza, Cristian y Kekevin. Para acabar, este año he conocido más en detalle a otras geniales personas como: Alberto, Ale, Adán, Alex y Kevin.

Para finalizar, dar las gracias a mi tutor por haberme aconsejado para la realización de este trabajo y a la universidad por haberme brindado la oportunidad de realizar el Erasmus, ya que ha sido una de las mejores experiencias de mi vida.

Índice general

Índice de figuras

Índice de Tablas

Lista de acrónimos

API *Application Programming Interface*.

BLE *Bluetooth Low Energy*.

DDoS *Distributed Denial of Service*.

DPI *Deep Packet Inspection*.

EMS *Element Management System*.

ETSI *European Telecommunications Standards Institute*.

GUI *Graphical User Inerface*.

ICMP *Internet Control Message Protocol*.

IEEE *Institute of Electrical and Electronics Engineers*.

IETF *Internet Engineering Task Force*.

IoT *Internet of Things*.

ITU *International Telecommunication Union*.

JSON *JavaScript Object Notation*.

JVM *Java Virtual Machine*.

LLDP *Link Layer Discovery Protocol*.

MD-SAL *Model-Driven Service Abstraction Layer*.

MIT *Massachusetts Institute of Technology*.

NAS *Network Attached Storage*.

NAT *Network Address Translation*.

NBI *Northbound Interface*.

NFV *Network Function Virtualization*.

NFV-MANO *NFV-Management and Orchestration* .

NFVI *Network Functions Virtualization Infrastructure*.

ODL *OpenDayLight*.

OF *OpenFlow*.

OFM *OpenFlow Manager*.

ONF *Open Networking Foundation*.

OvSDB *Open virtual Switch Database*.

REST *Representational State Transfer*.

RFID *Radio-Frequency Identification*.

SAL *Service Abstraction Layer*.

SBI *Southbound Interface*.

SDN *Software Defined Networking*.

SNMP *Simple Network Management Protocol*.

STP *Spanning Tree Protocol*.

TLS *Transport Layer Security*.

TMN *Telecommunications Management Network*.

VIM *Virtualized Infrastructure Manager*.

VNF *Virtualized Network Functions*.

VNX *Virtual Networks over Linux*.

XML *Extensible Markup Language*.

YAML *YAML Ain't Markup Language*.

YANG *Yet Another Next Generation*.

Resumen

Tradicionalmente las redes han estado basadas en el uso de *switches*, *routers* y otros dispositivos físicos para su correcto funcionamiento. Debido al constante aumento de tráfico en la red provocado por la aparición de las numerosos dispositivos IoT y sumado a la inserción del concepto de *Cloud Computing* en red, las "redes tradicionales" han quedado obsoletas. La aplicación del concepto de *software defined networking* (SDN) o redes definidas por software solucionan este problema, ya que al estar basadas en el uso del *software* y no depender de un *hardware* específico, son más flexibles y permiten un mejor control y facilidad a la hora de asignar recursos.

En este proyecto se propone el estudio de las tecnologías SDN, NFV e IoT, ahondando más en la primera. Dentro de SDN, se estudia el protocolo que se ha establecido como un estándar de comunicación, OpenFlow. Seguidamente, se da paso al desarrollo de simulaciones en dos plataformas distintas (Mininet y GNS3) mediante el uso de dos controladores distintos (ODL y RYU), con la intención de estudiar las características de SDN y OpenFlow. Todo esto tiene como objetivo el desarrollo de una plataforma redundante de control SDN-IoT.

La plataforma consiste en la realización de un escenario de red SDN mediante el uso de dos controladores, un cliente IoT, un servidor y dos *switches* OF puros. A través del envío constante de información entre el dispositivo IoT y el servidor, se quiere demostrar la importancia de la redundancia y la automatización.

Los resultados finales dejan ver la importancia de la redundancia en SDN para *switches* OF puros y la importancia de la automatización en los procesos de red, para así evitar la dependencia humana de estos.

Abstract

Historically, networks have been based on the use of switches, routers and other physical devices for their proper functioning. Due to the constant increase in network traffic caused by the appearance of the numerous IoT devices and added to the insertion of the concept of networked Cloud Computing, "traditional networks" have become obsolete. The application of the concept of software defined networking (SDN) solves this problem, since they are based on the use of software and do not depend on specific hardware, they are more flexible and allow better control and facilitate the allocation of resources.

This project proposes the study of SDN, NFV and IoT technologies, going deeper into the first one. Within SDN, the protocol that has been established as a communication standard, OpenFlow, is examined. Then, it is followed by the development of simulations in two different platforms (Mininet and GNS3) by using two different controllers (ODL and RYU), aiming to study the characteristics of SDN and OpenFlow. All this has the intention of developing a redundant SDN-IoT control platform.

The platform consists of the implementation of a SDN network scenario through the use of two controllers, an IoT client, a server and two pure OF switches. By constantly sending information between the IoT device and the server, it is intended to demonstrate the importance of redundancy and automation.

The final results show the importance of redundancy in SDN for pure OF switches and the importance of automation in the network processes in order to avoid human dependence on them.

Parte I

MEMORIA

Capítulo 1

Introducción

1.1. Introducción

A lo largo de este documento, se ha desarrollado un Trabajo Fin de Grado (T.F.G.) centrado en el estudio, simulación e implementación física de tecnologías relacionadas con el diseño de la arquitectura de red. Estas son: SDN y NFV.

Junto a la implementación física de la red, se añade un dispositivo IoT que otorga sentido al título del trabajo.

1.2. Estado del arte

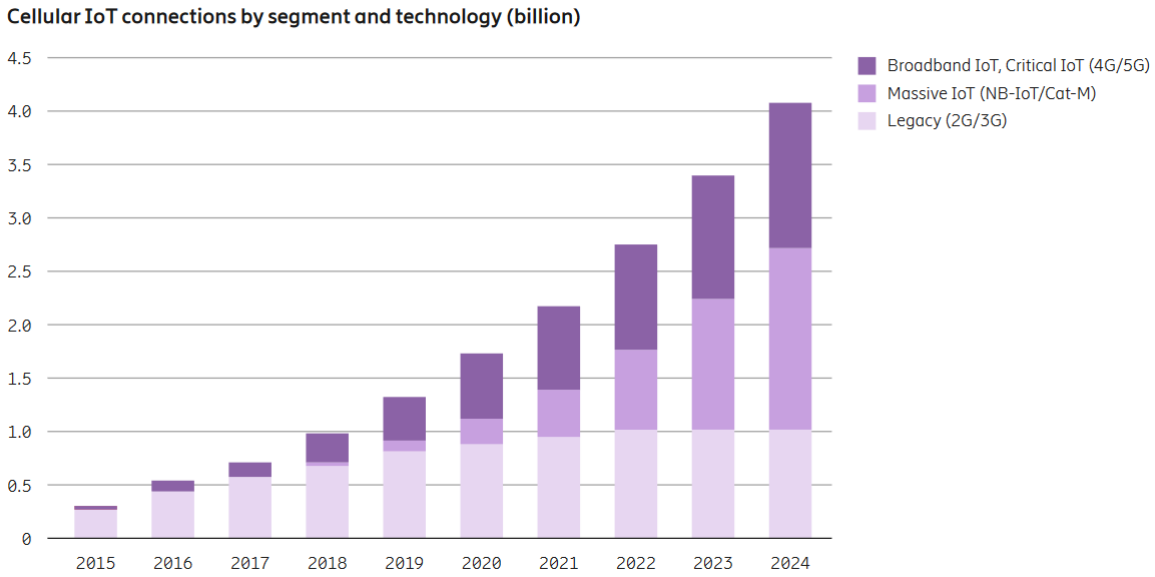
Para realizar un mejor análisis del estado del arte, se ha procedido a dividirlo en dos sub-apartados:

- Contexto actual
- Antecedentes

1.2.1. Contexo actual

Durante mucho tiempo las redes han estado basadas en el uso de switches, *routers* y otras infraestructuras físicas para su correcto funcionamiento. Estas se conocen como "las redes tradicionales". Debido a la constante evolución de la tecnología y con la aparición de las grandes redes de dispositivos IoT y el *Cloud Computing*, como se puede observar en la Figura 1.1 y Figura 1.2 , las redes tal y como se conocían hasta

Figura 1.1: Crecimiento de dispositivos IoT [3]



hoy, han quedado obsoletas. Esto se debe al gran volumen de datos y tráfico que las tecnologías mencionadas anteriormente generan. Es, por tanto, que las redes se han visto obligadas a soportar unas cargas de tráfico para las que no fueron diseñadas en su momento, lo cual hace que se tenga la necesidad de modificar su estructura.[1][2]

La solución a este problema está en el uso de las redes definidas por software o comúnmente llamadas por su acrónimo, SDN. La diferencia fundamental entre estas y las redes tradicionales reside en que como su propio nombre indica, las SDN están basadas en el uso del *software* y no en el *hardware* como sí lo están las redes tradicionales. Es por este motivo que las SDN son más flexibles ya que permiten a los administradores de red un mejor control y una mayor facilidad a la hora de asignar recursos.

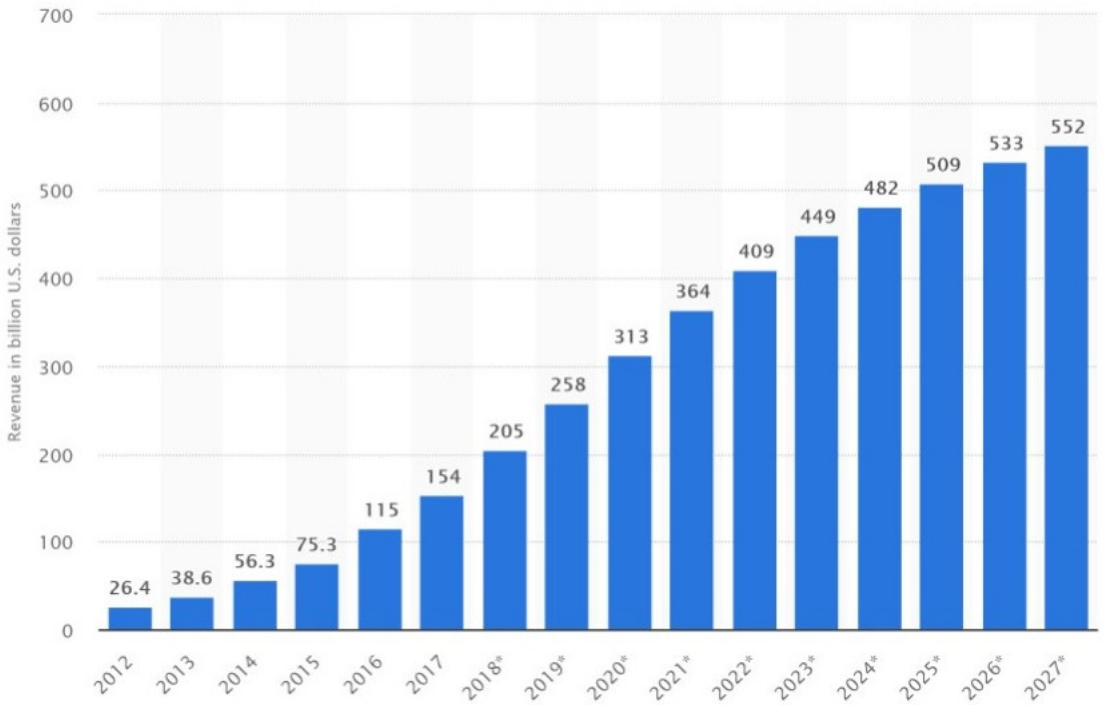
Las ventajas que tiene el uso de las SDN son las siguientes:

1. **Control centralizado de la red**: Ayuda a centralizar la gestión ofreciendo una perspectiva unificada de toda la red.

2. **Mayor seguridad**: Al tener la red centralizada en el controlador SDN, en caso de realizarse un ataque a su controlador como objetivo principal, la red estará asegurada si éste lo está.

3. **Ahorro en costes operativos**: Muchas de las tareas de administración de red

Figura 1.2: Crecimiento del *Cloud Computing* [2]



pueden ser automatizadas y centralizadas. Obteniendo una mayor reducción en costes operativos.

4. **Ahorros en *hardware* y reducción de gastos**: El controlador SDN es capaz de mandar instrucciones al *hardware* básico ya instalado para que cumpla nuevas funciones, consiguiendo así que estos aparatos sean reutilizables.

5. **Mayor QoS (*Quality of Service*)**: Esto se obtiene gracias a la posibilidad de gestionar el tráfico de datos. Por ello es más sencillo tener un mayor control de priorización del tráfico y, por tanto, una mayor calidad de servicio. Por ejemplo para el tráfico VoIP se puede dirigir y automatizar el tráfico en función de las condiciones de la red.

Estas ventajas hacen que las SDN se consoliden como una alternativa real y se estén implantando cada día más.[4]

1.2.2. Antecedentes

El proyecto continúa la línea de investigación sobre SDN con el trabajo previo[5]. Éste básicamente ha consistido en la implementación de una red con arquitectura SDN

de manera virtual y física.

Desarrollo virtualizado

Para llevar acabo la arquitectura SDN de manera virtualizada se han usado dos entornos para las simulaciones:

- **VNX**
- **Mininet**

Con la herramienta **VNX** se creó una red virtualizada en la que los sistemas finales estaban basados en Linux y los *routers* en VyOS[1]. La conexión entre todos los dispositivos fue posible gracias al uso de *bridges* Ethernet que utilizan interfaces virtuales de tipo *veth*.

Haciendo uso de la herramienta **Mininet** se realizaron simulaciones con tres topologías de red diferentes:

1. Topología en árbol
2. Topología en estrella
3. Topología lineal

Para cada una de ellas se hicieron dos simulaciones, distinguiendo entre una y otra simulación el controlador SDN utilizado. En el primer caso fue RYU y en el segundo, ODL. La diferencia principal entre ambos es el modo de ejecución, ya que RYU está basado en Python y ODL corre sobre una JVM. En las simulaciones se pudo observar la utilización del protocolo OpenFlow entre el controlador y el *switch* o los *switches* de la red. El análisis del tráfico fue posible gracias al uso de la herramienta Wireshark, capaz de capturar los paquetes enviados.

Desarrollo físico

Para el desarrollo de la arquitectura SDN de forma física los dispositivos usados fueron una Raspberry Pi 4, dos Raspberry Pi 3, un *switch* OpenFlow Zodiac FX y un ordenador portátil. Las funciones que se les asignaron dentro de la red a cada uno de ellos fueron las siguientes:

[1]VyOS es un sistema operativo de red de código abierto basado en Debian GNU/Linux

- El ordenador y la Raspberry Pi 4 se usaron indistintamente como controlador SDN.
- Las dos Raspberry Pi 3 fueron usadas como dispositivos finales.
- El Zodiac fue utilizado como un *switch*.

Una vez definidas las funciones, los dispositivos se conectaron mediante cables Ethernet a los distintos puertos del *switch*, estando en el puerto 4 el controlador. Esto ha de ser así ya que el fabricante del *switch* lo establece.

Realizadas las conexiones, se instaló en el controlador tanto RYU como Faucet, ambos harán las veces de gestor del plano de control.

Finalmente, se analizó tanto el tráfico de paquetes y las tablas de flujo de OpenFlow, como la conectividad entre los distintos dispositivos tras insertar diversas órdenes en la tabla de flujos del *switch*.

1.3. Objetivos

Este T.F.G tiene como principal objetivo el estudio de la tecnología SDN, tanto su arquitectura, como los protocolos que son propios de ella.

Los objetivos que se proponen son:

- **O1**. Analizar la tecnología SDN y tecnologías relacionadas como NFV e IoT.
- **O2**. Estudiar los distintos protocolos y controladores usados en SDN.
- **O3**. Construir arquitecturas SDN en entornos simulados.
- **O4**. Estudiar el lenguaje de programación Python.
- **O5**. Emular una aplicación práctica de arquitectura SDN mediante la creación de una plataforma SDN-IoT.

1.4. Estructura de la memoria

La presente memoria consta de: siete capítulos, valoración presupuestaria y anexos. A continuación se describe brevemente el contenido de cada capítulo:

- El **capítulo 1** realiza una introducción en la que se tratan tanto el contexto actual de las redes como los antecedentes en los que se basa el proyecto. Además, se incluyen los objetivos propuestos para la realización del trabajo.

- El **capítulo 2** tiene como objetivo principal analizar desde un punto de vista teórico las tecnologías SDN, NFV e IoT.

- El **capítulo 3** incluye los protocolos y controladores presentes en SDN . Dentro de los protocolos se centra en la descripción más detallada de OpenFlow, debido a que este es el utilizado en el desarrollo del proyecto.

- El **capítulo 4** abarca las simulaciones de arquitecturas SDN usando Mininet y distintos controladores; tales como: ODL y RYU.

- El **capítulo 5** incluye las simulaciones de arquitecturas SDN usando GNS3 y distintos controladores; tales como: ODL y RYU. Además, se describe la tecnología *docker* usada por GNS3.

- El **capítulo 6** recoge la aplicación práctica de una arquitectura SDN y la implementación de manera conjunta con un dispositivo IoT.

- El **capítulo 7** aborda las conclusiones y las líneas futuras del trabajo y la tecnología SDN.

Finalmente, se realiza un presupuesto en el que se desglosa lo que costaría realizar el proyecto y se añaden anexos a modo de aclaración de los distintos capítulos.

Capítulo 2

Fundamentos teóricos: SDN, NFV e IoT

2.1. Introducción

Este capítulo aborda una serie de conceptos importantes del desarrollo del T.F.G.. Para comenzar, trata el concepto de SDN, pieza fundamental en el transcurso del trabajo, puesto que se va a utilizar en la red a desarrollar. Luego, se describe la tecnología NFV, ya que, pese a no implementarla en el proyecto es digna de mención y tiene una estrecha relación con SDN. Para finalizar, desarrolla el concepto de IoT para posteriormente aplicarlo a SDN.

2.2. SDN

2.2.1. Definición

Según la ONF, SDN se define como la separación física del plano de control del plano de datos, donde un plano de control puede controlar múltiples dispositivos [6]. De otro modo, según se especifica en [7], SDN es el conjunto de técnicas usadas para facilitar el diseño, entrega y funcionamiento de los servicios de red de una manera determinista, dinámica y escalable. En definitiva, tenemos que si unimos ambas, se llega a una conclusión en la que SDN establece como una de las técnicas, la separación del plano de control del plano de datos.

2.2.2. Limitaciones de las redes tradicionales

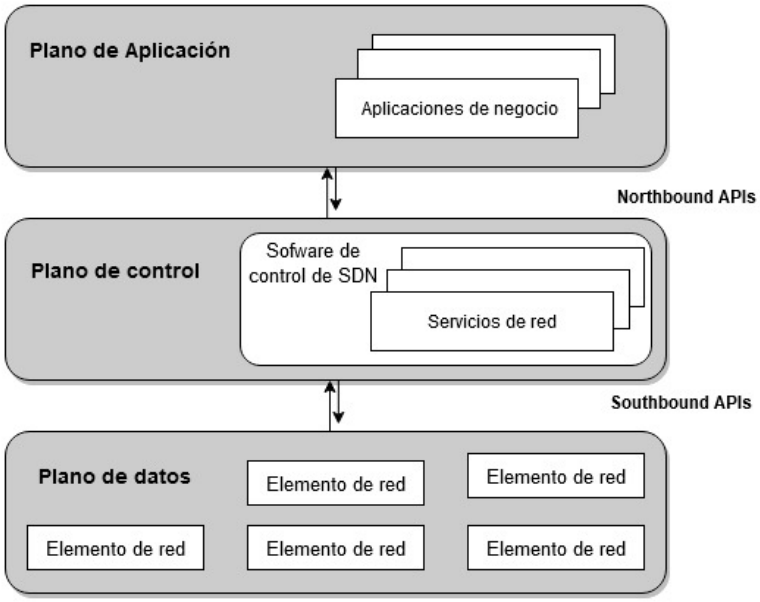
Como se comentó previamente en la sección 1.2.1, las redes tradicionales han quedado desfasadas en tiempo y forma respecto de lo que demandan las necesidades actuales. Esto se debe, a que a diferencia de las SDN, sus principales características son una desventaja para el desarrollo y creación de nuevas aplicaciones.[8]

- **Son rígidas**: Se comportan en base al dictamen de los fabricantes de estos dispositivos.
- **Limitan la investigación y el desarrollo**: Los dispositivos son opacos de cara al consumidor, no se pueden realizar grandes avances sin tener en cuenta a los fabricantes de los mismos.
- **Lentitud para actualizar**: Al tener que configurar cada uno de los elementos de la red individualmente, una actualización en la red puede llevar una gran demora de tiempo.
- **Grandes recursos de mantenimiento**: Al no estar la red centralizada en un único dispositivo, se necesita un gran número de personal para realizar las gestiones de mantenimiento.

	SDN	**Redes tradicionales**
Limitan la investigación y el desarrollo	No. Están basadas en *Open Source*	Sí. Se ciñen al dictamen de los fabricantes.
Tiempo de actualización	Rápido. Está centralizada en el controlador.	Lento. La red está distribuida y hay que cambiar la función de cada dispositivo de manera individual.
Separa el plano de datos y de control	Sí. Lo que hace que se tenga un control general de las funciones de los dispositivos.	No. Lo que hace que cada dispositivo tenga su propio mecanismo de control.

Tabla 2.1: Comparación SDN vs Redes tradicionales

Figura 2.1: Arquitectura SDN



2.2.3. Arquitectura

Como se ha mencionado en el apartado 2.2.1, SDN se basa en el concepto de la separación del plano de control del de datos. Por esta razón, existirá tanto un dispositivo que actúe de controlador, como otro que esté controlado. El controlador es capaz de manejar a los dispositivos controlados a través de una interfaz. Normalmente, estas comunicaciones se realizan a través de las llamadas Nortbound y Southbound APIs, ya sean abiertas o propietarias. [9]

Como se observa en la Figura 2.1, SDN se divide en tres planos, cada uno de ellos con una función.

- **Plano de datos**: Está compuesto por uno o más dispositivos de red. Es el responsable del manejo de paquetes en el *datapath*. Dentro de sus capacidades están el reenvío y descarte de los mismos. Esto se lleva acabo siguiendo las directrices marcadas por el controlador.

- **Plano de control**: Está formado por uno o más controladores SDN. Es el encargado de las acciones a realizar sobre los paquetes que circulan en la red. Estas órdenes son trasladadas desde el controlador SDN hasta los elementos de red, para posteriormente ser ejecutadas por estos últimos.

- **Plano de aplicación**: En él residen las aplicaciones capaces de proveer de ser-

vicios a los usuarios que gestionan la red. Estas se comunican con el controlador a través de las *Northbound* APIs usando el estilo REST.

A continuación, se describen con más detalle cada uno de los planos mencionados anteriormente.

Plano de datos

El plano de datos es el responsable de lidiar con las acciones relacionadas con el flujo de paquetes en la red. Los elementos de red que lo forman (como por ejemplo los *switches*) son los encargados de realizar dichas acciones. Estos deben tener los suficientes recursos físicos para poder asegurar una buena disponibilidad, calidad, seguridad y conectividad en la red.

Al estar separados el plano de datos y el de control, los elementos de red no son autónomos, es decir, no toman decisiones propias para los paquetes que reciben. Esto se debe a que dependen de las instrucciones recibidas desde el controlador (enviar, descartar, reenviar por otros puertos, etc.) a través de la SBI para realizar dichas acciones. Sin embargo, el controlador puede configurar los elementos de red para que actúen de manera autónoma ante fallos en la red o para protocolos como LLDP, STP o ICMP. [10]

Plano de control

En primer lugar, como se observa en la Figura 2.1, el plano de control se encuentra entre el plano de datos y el plano de aplicación. Es por este motivo que actúa como nodo de comunicación entre los dos. Esta comunicación se efectúa a través de interfaces, para la comunicación con el plano de datos, se lleva acabo a través de la llamada *Southbound Interface* (SBI), en cambio para la comunicación con el plano de aplicación, se lleva acabo a través de la *Northbound Interface* (NBI).

En segundo lugar, conviene resaltar que el plano de control está formado por uno o varios controladores, siendo este el elemento principal dentro de la arquitectura SDN. El controlador no es más que una aplicación ejecutada en un servidor y tiene la función de actuar como una especie de sistema operativo, ya que es capaz de comunicarse con aplicaciones y elementos de red. Una comunicación estándar podría ser la siguiente:

- El administrador de red abre una aplicación. Una vez dentro, cambia parámetros relacionados con el flujo de paquetes en la red y aplica el descarte de paquetes ICMP. Al salvar esa configuración, la aplicación envía los datos a través de la NBI al controlador mediante una llamada a una API. Cuando el controlador recibe los datos, los envía a través de la SBI mediante el uso de una Soutbound API a los elementos de red asociados. Una vez estos reciben la información del controlador, empiezan a aplicar la tabla de flujo recibida. Por tanto, ahora cuando un PC realice un *ping* a otro, no recibirá respuesta, pero en cambio, si realiza una conexión usando cualquier otro protocolo si la recibirá.

Es por este motivo que SDN se considera una red centralizada, ya que, desde un único controlador se pueden cambiar los flujos de tráfico en la red en base a las políticas decididas por el administrador. Esto dota de dinamismo y flexibilidad a SDN, en caso de un cambio urgente, no es necesario ir elemento a elemento cambiando las reglas para el tráfico de paquetes. [10]

Para finalizar, tanto los distintos controladores, como protocolos de comunicaciones entre los distintos planos, se explican más en detalle en el Capítulo 3.

Plano de aplicación

El plano de aplicación es el que se encuentra en lo más alto de la arquitectura SDN y en él residen todas las aplicaciones usadas. Como bien se comentó, estas aplicaciones utilizan REST APIs (el formato de estas puede ser variado: JSON, XML, YAML, etc.) para comunicarse con el controlador, y viceversa. Al ser la comunicación en ambos sentidos, las aplicaciones pueden mandar órdenes al controlador, recibir información y generar estadísticas e informes sobre lo que ocurre en la red. Ejemplos de las funciones de aplicaciones usadas son las siguientes:

- **Servicios de seguridad**. Es muy importante reducir los riesgos y actuar rápidamente en caso de producirse un ataque. Aplicaciones usadas en este aspecto son:[11]

    1. **F5's BIG-IP**. Escanea la red en busca de diferentes tipos de amenazas y cuando la encuentra se comunica con el controlador. Le informa de la situación y el controlador posteriormente envía los flujos necesarios al elemento

de red por donde entra esa amenaza para descartar esos paquetes en su entrada.

2. **Radware Defenseflow**. Es usado para las amenazas de tipo DDoS. Se escanean los *switches* en busca de actividad maliciosa (esta se detecta en base a estadísticas), en caso de que sea detectada, automáticamente se transmite esta información al controlador para que actualice las tablas de flujo y se redirijan los paquetes a otra aplicación, DefensePro IDS. Esta filtra el tráfico malicioso y deja pasar únicamente al que no suponga una amenaza.

- **Monitorización de la red**. Es importante escanear la red para actuar en caso de fallos, hoy, donde las redes son más complejas, esta puede ser una ardua tarea. Ejemplos de aplicaciones que ayudan a ello son:[11]

  1. **Big Tap Monitoring Fabric**. Esta se divide en varias capas, cada una de ellas con una función. La capa de filtrado se encarga de filtrar el tráfico. La capa de servicio se utiliza para las modificaciones de paquetes. La capa entrega envía los paquetes filtrados a las herramientas de monitoreo. Finalmente, el controlador de Big Tap programa el monitoreo.
  2. **Microsoft DEMon**. Existen puertos de monitorización dentro de los *switches*. Estos envían los datos recibidos por los puertos a las herramientas de monitoreo. Una vez allí, estos son analizados en búsqueda de cualquier anomalía.

- **Gestión de ancho de banda**. Con este tipo de aplicaciones, los operadores aseguran a sus usuarios una óptima conexión en cualquier instante de la red. Estas monitorean el ancho de banda requerido por los usuarios, para así poder establecer flujos que lo aseguren. Ejemplos de aplicaciones que ayudan a ello son:[12]

  1. **Kemp**. Proporciona un balance de carga adaptativo basado en la cantidad de tráfico recibido por los *switches*. Normalmente, la toma de decisiones está basada en las características requeridas según las capas 4 a 7 (modelo OSI).
  2. **Realstatus hyperglance**. Muestra la topología de red y la información relacionada con el flujo a tiempo real. Al mismo tiempo, permite encontrar

el camino más corto entre 2 *hosts* y modificarlo en caso de que el actual no sea el más óptimo.

2.2.4. Seguridad

En este apartado se aborda un aspecto muy importante en SDN, la seguridad. Para su análisis, se analizan dos puntos:

1. Los problemas asociados, divididos a nivel de comunicación y de componentes.
2. Las directrices que se han de seguir para la solución de estos problemas, así como las limitaciones que generan.

Problemas asociados

Según [13], los problemas de seguridad presentes en SDN se pueden dividir en dos: los problemas a nivel de comunicación y de componentes. Se comenzará por la descripción del primero.

Por un lado, los problemas derivados de la comunicación, se observan dos niveles: *Northbound* y *Southbound*.

En primer lugar, a nivel *Northbound* se tienen dos problemas principales:

- La ausencia de confianza y una débil autenticación en la comunicaciones entre controlador y aplicación. Esto puede llevar a un *spoofing*, que viene a ser la suplantación de los mensajes API transmitidos entre estos elementos.
- Una mala autorización. Esta puede causar un acceso inapropiado o de carácter malicioso a las aplicaciones.

En segundo lugar, a nivel *southbound* se tienen tres problemas principales:

- La ausencia de comunicaciones cifradas entre el controlador y los elementos de red. Esto puede llevar a escuchas y suplantaciones en las comunicaciones.
- La ausencia de métodos de confianza y la débil autenticación puede llevar a ataques del tipo *Man-In-The-Middle* y *spoofing*. Esto provoca que el atacante pueda capturar los flujos de la red y actuar en consecuencia.

- Una mala autorización. Esto puede causar un acceso inapropiado y por tanto, que el atacante entre, cambie las políticas de flujo en el controlador y las envíe a los *switches*, para que estos las ejecuten y reenvíen los paquetes a donde desee.

Por otro lado, los problemas derivados por parte de los componentes se organizan en tres partes: la aplicación, el controlador y el elemento de red, en este caso se trata de un *switch*.

En primer lugar, para las aplicaciones existen tres problemas asociados.

- La existencia de aplicaciones maliciosas o que no procedan de fuentes fiables. Esto puede generar una gran vulnerabilidad y comprometer a la totalidad de la red.
- La vulnerabilidad de la API. En caso de que la API fuese vulnerable, el atacante podría acceder, crear sus propias políticas SDN y cambiar el tráfico en la red a su favor.
- La falta de aislamiento de la aplicación. Esto puede dar lugar a reglas de flujo inconsistentes que afecten al normal funcionamiento de la red.

En segundo lugar, para el controlador se observan cuatro problemas principales.

- El controlador puede sufrir un ataque de denegación de servicio. En este caso se pondría en jaque a toda la red, ya que el controlador dejaría de funcionar.
- La implantación de controladores corruptos por parte del atacante. En este caso, él podría crear entradas en las tablas de flujo de los elementos de red, teniendo así un control completo de esta.
- *Controller Hijacking*, es decir, el secuestro del controlador. En este caso el atacante tendría pleno control sobre la red al tener todo el acceso al controlador.
- Los errores de análisis relacionados con el lenguaje de codificación (XML,JSON,YAML). Esto podría generar nuevas vulnerabilidades en el controlador.

Finalmente, para el elemento de red, en este caso el *switch*, se destacan tres problemas.

- La posibilidad de un ataque de denegación del servicio. En este caso, el atacante podría inundar las tablas de flujo del *switch* mediante un envío masivo de reglas de flujo.
- El estado activo del modo escucha. En caso de que el *switch* tuviera activado este modo, se le da la posibilidad al atacante de entrar sin credenciales.
- La inconsistencia en las reglas de flujo. Esto puede llevar a una indisponibilidad e inestabilidad en la red.

Soluciones

Desde un punto de vista general, las soluciones pasan por dotar de disponibilidad, integridad y privacidad a todos los recursos e información presentes en la red. Para ello y según [14], se han de seguir los siguientes criterios:

- **Asegurar el controlador**. Puesto que es el elemento principal de la red, el acceso a él necesita estar fuertemente controlado.
- **Proteger el controlador**. En caso de que el controlador se cayera, la red correría peligro. Es por tanto que necesita una buena protección.
- **Generar confianza**. El hecho de proteger las conexiones en la red es un tema crítico. Es por ello que las aplicaciones instaladas en el controlador y los elementos asociados a él deben provenir o ser de fuentes fiables.
- **Crear una política robusta.** Se necesita de un sistema que compruebe que el controlador está realizando en todo momento lo que se desee que haga.
- **Realizar análisis forenses y correcciones**. En caso de que ocurriese un incidente, se ha de investigar el porqué y una vez averiguado, solucionarlo y proteger al sistema en caso de que surgiera de la misma incidencia.

Dentro de estos criterios, se puede indagar de una manera más específica en las soluciones. Según [13], estas son los siguientes:

- El uso de cifrado TLS en todas las comunicaciones, tanto northbound como southbound. Esto ayuda a eliminar las escuchas y el *spoofing*. Además, añade la autenticación mutua, lo que hace que se valide cada elemento y se evite la introducción de controladores y *switches* maliciosos en la red.

- El endurecimiento de las políticas del controlador. De esta manera se reducen las vulnerabilidades y se puede monitorizar e identificar cualquier actividad sospechosa de ser maliciosa.

- Redirigir el tráfico a *middleboxes* (*firewalls*, DPI boxes, NATs)

- Desplegar nuevas reglas de flujo de manera reactiva.

Pero estas soluciones también generan unas consecuencias. Estas son:

- Al ser la implementación de TLS bastante compleja, muchos de los vendedores de SDN no lo proporcionan. Además, hasta la fecha no hay ningún modelo predefinido de implementación como podría ser la Infraestructura de Clave Pública (PKI) o la Privacidad Bastante Buena (PGP). La falta de una estructura predefinida hace que se disminuya administrabilidad y extensibilidad de una red SDN.

- El problema principal del endurecimiento de las políticas del controlador es que es general y depende del modo de implementarlas y del sistema operativo asociado a él.

- La redirección del tráfico a *middleboxes* es una solución momentánea, ya que al hacerlo se eliminan las principales ventajas de SDN, escalabilidad y flexibilidad.

- El despliegue de reglas de flujo de manera reactiva no es fácil de configurar, depende de las aplicaciones instaladas y no tiene un estándar a seguir. Además, el hecho de despliegue de las reglas de este modo puede generar una inundación en las tablas y como consecuencia, llevar a una denegación del servicio.

2.3. NFV

2.3.1. Definición

Debido a la aparición de nuevas tecnologías, la red sigue teniendo un crecimiento continuo y los operadores de red necesitan adaptarse comprando nuevos *middleboxes* (*firewalls*, *proxys*, monitores de QoS, DPIs, etc.). El problema principal de esto, es que estos di positivos son caros, necesitan de un personal especializado, consumen mucha energía, no pueden realizar otras funciones y tienen un corto tiempo de vida. Al mismo tiempo, todo esto genera que la red sea poco flexible y difícilmente escalable. Aquí es donde surge NFV. [15]

Según la ETSI describe en [16], la virtualización de funciones de red (NFV) tiene la intención de cambiar la manera en la que los operadores de red establecen la arquitectura de red. Para ello se usa la virtualización de los elementos presentes en ella, es decir, con NFV se propone la abstención del uso de dispositivos físicos, *routers*, *firewalls*, DPIs, *proxys*, ... e implementar las funciones de estos en grandes servidores. De esta manera se consigue que funciones que antes se realizaban en hardware específico se realicen ahora en el hardware estándar de un servidor. Esto supone una gran ventaja, ya que dota a la red de una gran escalabilidad y flexibilidad, puesto que el tiempo y coste de instalación de estos servicios es mínimo.

2.3.2. Arquitectura

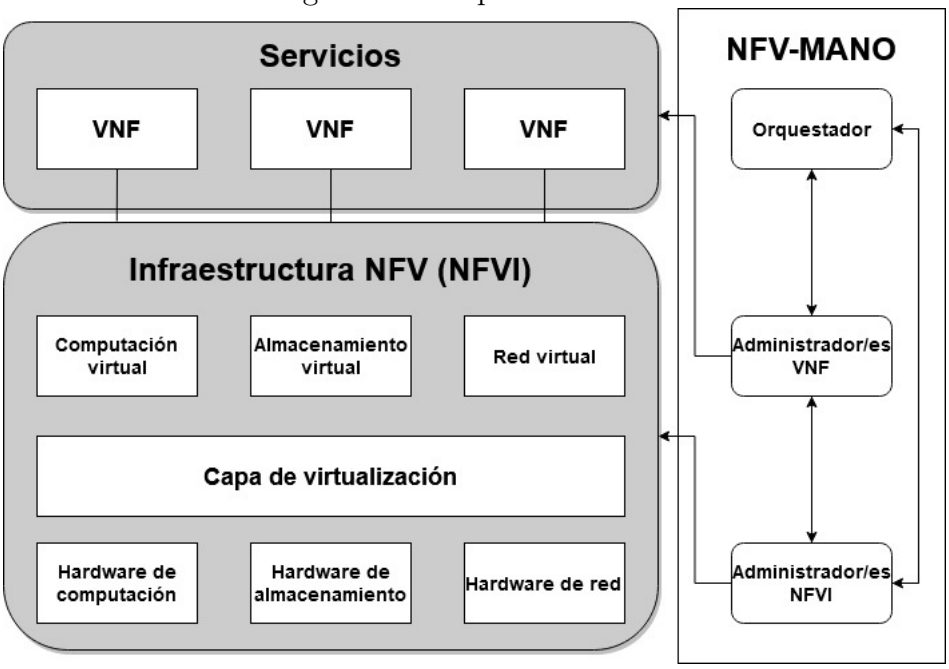
Según [17], dentro de la arquitectura en NFV existen tres elementos principales: las VNFs o funciones de red virtualizadas, la NFVI o infraestructura de red virtualizada y la NFV-MANO o el administrador y orquestador de NFV.

Es notable destacar que en cuanto al elemento de las VNFs, según [15], se le llama servicios. Este no es más que el conjunto de varias VNF. De aquí en adelante, se usará esta nomenclatura. El esquema de esta arquitectura se puede observar en la Figura 2.2

Servicios

Como se comentó en el punto anterior, los servicios son el conjunto de VNFs. Estas pueden ser implementadas en una o más máquinas virtuales. Cada VNF es la

Figura 2.2: Arquitectura NFV



implementación software de una determinada función de red y generalmente suelen ser administradas por un agente llamado EMS o Sistema de Gestión de Elementos. Este es el responsable de su creación, configuración, monitorización, seguridad y rendimiento [15]. Todo esto se puede observar de manera gráfica la Figura 2.3

Infraestructura NFV (NFVI)

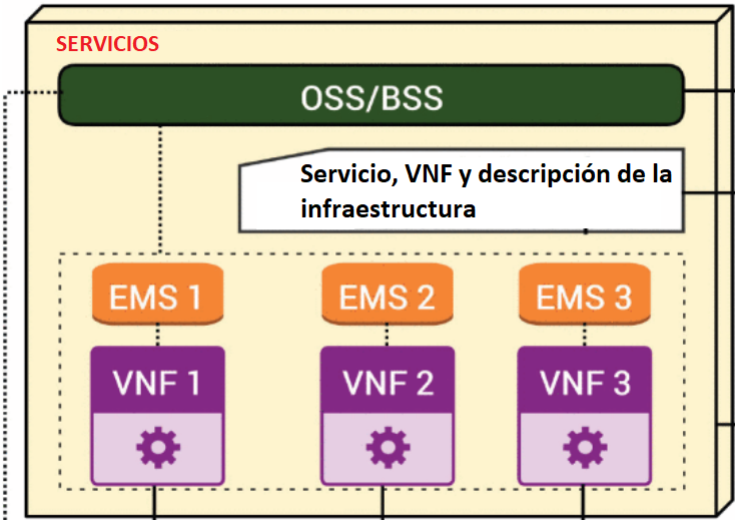
Según define la ETSI en [17], NFVI es el conjunto de recursos físicos y virtuales que posibilitan la correcta ejecución, administración y despliegue de las VNF. Dentro de la infraestructura NFV existen dos tipos de recursos, los físicos y virtuales.

En primer lugar, en los recursos físicos, se distinguen tres tipos:

- **Recursos de computación**. Tienen que ver con los elementos requeridos para el procesado de datos (ej. número de CPUs)
- **Recursos de almacenamiento**. Pueden ser compartidos (mediante el uso de NAS) o individuales (presentes únicamente en el servidor).
- **Recursos de red**. Son los referentes a las funciones de conmutación, tipos de enlaces, número de tarjetas de red, etc.

Destacar que los recursos de computación y los de almacenamiento normalmente son puestos en común.

Figura 2.3: Servicios NFV. [15]



En segundo lugar, los recursos virtuales. Donde se distinguen la capa de virtualización, la computación virtual, el almacenamiento virtual y la red virtual.

Por un lado, la capa de virtualización, que normalmente suele ser un hipervisor, se encarga de la abstracción de los recursos hardware separándolos de manera lógica para cada VNF, haciendo así que sus recursos virtuales sean independientes entre sí.

Por otro lado, el resto de recursos virtuales (almacenamiento, memoria, red) son los que tienen asignados cada una de las máquinas virtuales sobre las que se ejecutan las VNF. Todo esto hace que desde el punto de vista de la VNF estos elementos parezcan un único elemento con las características requeridas.

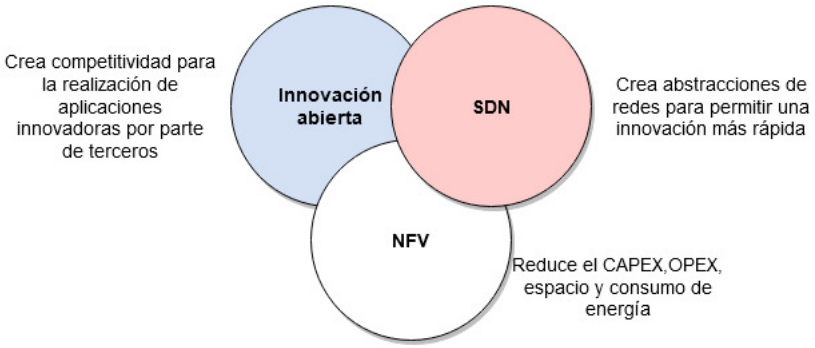
NFV-MANO

El orquestador y administrador de NFV está dividido en tres elementos principales: el administrador de la infraestructura NFV (VIM), el administrador de VNF y el orquestador.

Primeramente, el VIM según [17], es el encargado de gestionar y controlar los recursos hardware asignados a las distintas VNF. Para llevarlo acabo ejecuta las siguientes tareas:

- **Gestión de recursos**. Se encarga de llevar un inventario del software y los recursos de computación, almacenamiento y red. Paralelamente, ejecuta la distribución

Figura 2.4: Relación NFV-SDN [16]



de las máquinas virtuales en los hipervisores, así como la gestión de sus recursos para poder dar o quitar en función de lo que se necesite.

- **Llevar operaciones con distintos fines.** Para la visibilidad y gestión de la infraestructura NFV, el análisis de problemas de rendimiento y la recolección de datos sobre fallos en la infraestructura.

En segundo lugar, el administrador de VNF es el encargado de la gestión del ciclo de vida de las distintas VNF.

Finalmente, el orquestador es el responsable de automatizar todos los procesos llevados a cabo en la infraestructura de NFV.

2.3.3. Relación con SDN

SDN y NFV son tecnologías distintas y no dependientes una de la otra, pero que a la hora de implementar una red basada en el uso de estas dos, supone una gran mejora respecto de usar una o ninguna de las dos.

Por un lado, al SDN separar los planos de datos y control, genera una mejora en el rendimiento y desarrollo de la red y facilita las labores de mantenimiento de la misma.

Por otro lado, NFV provee de la infraestructura sobre la que SDN va a ejecutarse. La relación entre ambas se observa en la Figura 2.4.

Al ser ambas tecnologías de carácter abierto, no se depende de ningún vendedor para desarrollarlas, lo hace que la labor para la investigación, el desarrollo y la posterior

innovación sean más fáciles de llevar acabo ya que cualquiera que esté interesado puede efectuar esa labor.

A continuación, para una mejor comprensión de lo tratado anteriormente, se presenta un ejemplo de aplicación de estas tecnologías. Este ejemplo está sacado de [18].

Ejemplo de implementación de arquitectura SDN-NFV

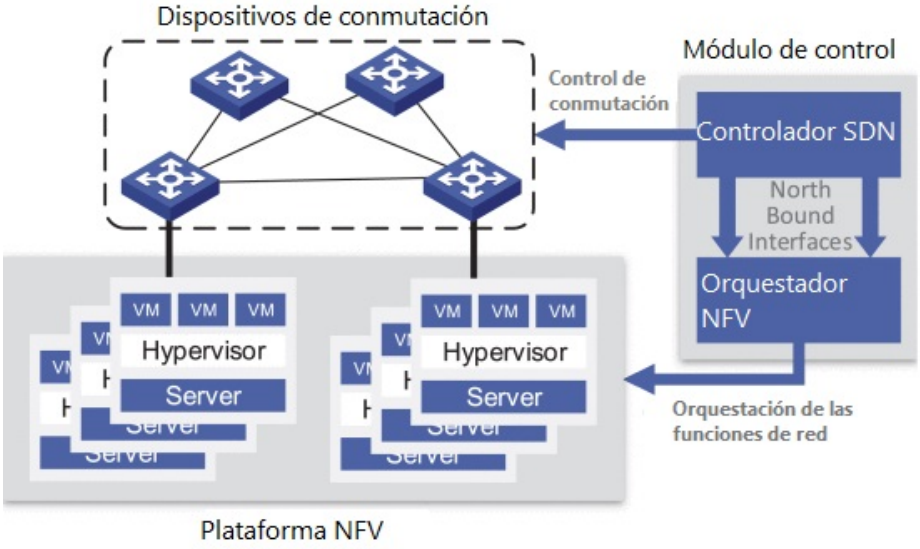
En primer lugar, la arquitectura consiste en el uso de tres elementos: dispositivos de conmutación (p.ej: *switches*), un módulo de control y una plataforma NFV. La distribución de los elementos se puede observar en la Figura 2.5.

Segundo, la lógica de la conmutación de paquetes, como era previsible, está determinada por el controlador SDN siendo este quien establezca las tablas de flujo dentro de los distintos *switches*. Para esta comunicación, como se ha mencionado con anterioridad en el apartado 2.2.3, se hace uso de la interfaz southbound, junto con uno de los protocolos establecidos, por ejemplo, OpenFlow.

Tercero, dentro de la plataforma NFV se encuentran los servidores, sus asociados hipervisores y las distintas máquinas virtuales. Al igual que en anteriores apartados, el servidor en este caso es el que aporta los recursos físicos, el hipervisor se encarga de virtualizarlos y asignarlos de manera lógica a cada una de las máquinas virtuales. Luego, cada una de estas máquinas cumple cierta función de *middlebox* dentro de la red, como puede ser actuar de *proxy* o de cortafuegos.

Finalmente, el módulo de control se encarga de obtener tanto la topología como las políticas de reenvío de paquetes de la red. Una vez obtenidas, efectúa las consecuentes operaciones, como pueden ser la asignación de funciones de red a cada una de las máquinas virtuales y la traducción de las políticas en el establecimiento de enlaces de comunicación optimizados. En este caso, el orquestador se encarga de la asignación de las funciones de red a cada una de las máquinas virtuales y el controlador, inserta en estas y en los dispositivos de conmutación las reglas para el reenvío de paquetes.

Figura 2.5: Ejemplo de arquitectura SDN-NFV [18]



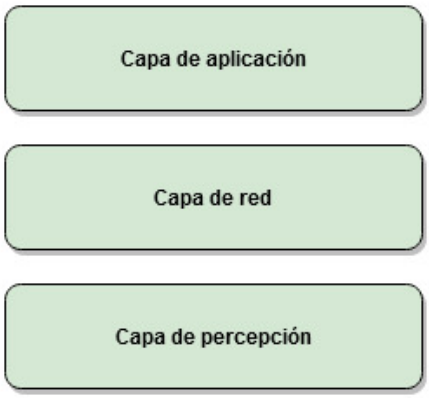
2.4. IoT

2.4.1. Introducción

El primer concepto de IoT surge en el año 1999 cuando dos investigadores del MIT, David Brock y Sanjay Sarma, estudiaron la posibilidad de añadir etiquetas RFID *low-cost* a productos, con el objetivo de localizarlos durante la cadena de suministro. A estas etiquetas se añadiría un número de serie que a posteriori sería guardado en una base de datos accesible desde Internet. Es preciso comentar que hasta ese momento en las etiquetas solo se guardaba información relacionada con el producto o con el contenedor donde viajaban. Este hecho hizo que por primera vez se pudieran asociar objetos de estas características a Internet. Después, las empresas comenzaron a emplear ese método y de esa manera sabían automáticamente cuando llegaba el producto. [19]

Años más tarde, varias organizaciones como la ETSI, la ITU, el IEEE, la IETF, etc. trabajaron en la estandarización y en la definición del concepto de IoT o internet de las cosas. Cada organización lo define a su manera, haciendo una síntesis de cada una de ellas, el IoT se puede definir como la conexión de todo tipo de dispositivos a lo que se conoce comúnmente como Internet. Estos dispositivos pueden ser coches, juguetes, *smartphones*, sensores, edificios, etc.

Figura 2.6: Arquitectura IoT. Modelo de 3 capas [20]



2.4.2. Arquitectura

Respecto de la arquitectura utilizada para redes IoT, [20] destaca el uso de dos modelos sobre los demás, el básico o de tres capas y el de cinco de capas.

Modelo de tres capas o básico

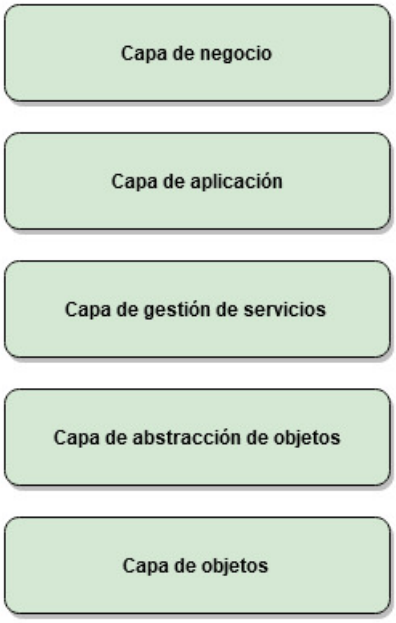
Es el que más destaca para la descripción de la arquitectura de una red en IoT. Se puede observar su composición en la Figura 2.6. La descripción de las funciones de cada una de las tres capas es la siguiente:

- **Capa de percepción**. Está dividida a su vez en dos partes: el nodo de percepción y la red de percepción. El nodo de percepción está compuesto por todos los dispositivos físicos conectados a la red (sensores, GPS, tags RFID, etc.). La red de percepción se encarga de la recolección de la información a través del control de los objetos; y del trasporte de la información hacia la capa de red. [21]

- **Capa de red**. También llamada capa de transmisión, puesto que su función es la de recibir los datos de la capa de percepción, procesarlos y enviarlos a la capa de aplicación o viceversa. Las tecnologías que hacen posible dicha función son Zigbee, 3G, Wi-Fi, Infrarrojos, Bluetooth, etc. [21]

- **Capa de aplicación**. Esta que permite la comunicación directa con los usuarios. En ella residen todas las aplicaciones presentes para los distintos servicios demandados (salud, domótica, seguridad, automoción, etc.).

Figura 2.7: Arquitectura IoT. Modelo de 5 capas [20]



Modelo de cinco capas

Es el utilizado en la actualidad, ya que el modelo básico se comienza a quedar corto para las distintas tecnologías que surgen cada año. Este nace la como combinación de dos modelos, el TCP/IP y el TMN [20]. Se puede observar su composición en la Figura 2.7. La descripción de las funciones de cada una de las cinco capas es la siguiente:

- **Capa de objetos**. Es la capa más baja dentro de la arquitectura y tiene como funciones principales la recolección y el procesado de la información obtenida (humedad, temperatura, aceleración, etc.) por parte de los distintos dispositivos IoT. Aquí es donde se generan todos los datos que pasarán a formar parte de lo que se conoce como *Big Data*.

- **Capa de abstracción de objetos**. Ejerce la función de intermediario entre la capa de objetos y la de gestión de servicios. Se encarga de que el envío de la información se haga a través de canales seguros. Para ello se utilizan tecnologías como RFID, Wi-Fi, Zigbee, BLE, etc.. Paralelamente, también se encarga del manejo de las funciones relacionadas con el *Cloud Computing* y la gestión de datos. [22]

- **Capa de gestión de servicios**. Es la capa intermedia de la arquitectura y se encarga de recibir cantidades masivas de datos por parte de la capa de abstracción

de objetos para posteriormente, guardarlos, procesarlos y analizarlos. Asimismo, su función principal es tomar las decisiones correspondientes para ofrecer al cliente el servicio oportuno.

- **Capa de aplicación**. En esta capa se ofrece al cliente diferentes tipos de servicios según lo demandado. Aquí entran los conceptos de *smart home*, *city*, *energy*, etc. [22]

- **Capa de negocio**. Esta es la capa más alta de la arquitectura, en ella se efectúa toda la gestión relacionada con todos los servicios y actividades de IoT. Por ejemplo, en esta capa se llevan acabo labores de diseño, análisis, implementación, evaluación y monitorización de las necesidades de la aplicación IoT. [22]

2.4.3. Relación con SDN

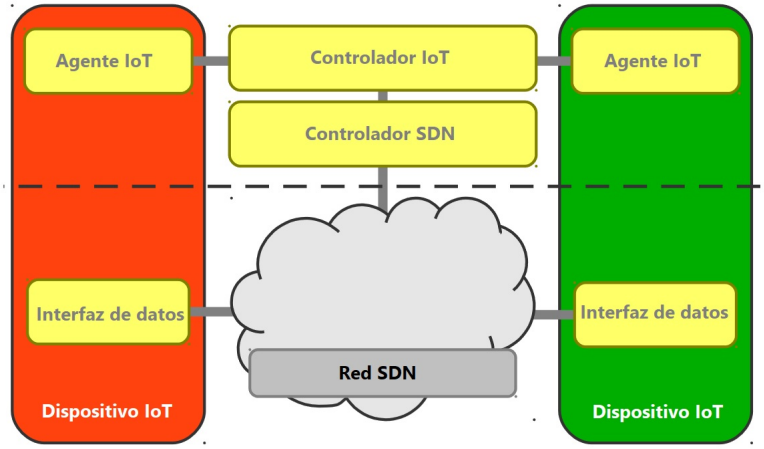
Las redes IoT están formadas por un gran número de dispositivos, que a su vez, generan un gran volumen de tráfico de datos. Según estima Ericsson en su informe de movilidad anual, este año 2020 habrán alrededor de 1.800 millones de dispositivos IoT conectados a la red [3]. Las arquitecturas de red tradicionales no pueden hacer frente al gran dinamismo de las redes IoT y ahí es donde entra SDN. Como ya se ha comentado, SDN es ampliamente flexible y dinámica, en gran medida se debe a que separa los planos de datos y de control, lo que hace que la red sea altamente programable y adaptable. Algunos de los beneficios que supone el uso de SDN según [23] son:

- **Prevención de cuellos de botella**. Esto es posible debido a que la red está centralizada en el controlador SDN. Este tiene una visión general de la red y, por tanto, puede redirigir el tráfico con el fin de realizar un enrutamiento óptimo.

- **Simplificación de procesos.** Los procesos análisis de información y toma de decisiones son simplificados en gran medida, gracias a que SDN provee de herramientas al entorno IoT para ello.

- **Mejora de prestaciones en materia de seguridad**. Gracias al aprovechamiento de características presentes en el controlador SDN se pueden implementar mecanismos de seguridad de manera más fácil.

Figura 2.8: Ejemplo de arquitectura SDN-IoT [24]



Ejemplo de implementación de arquitectura SDN-IoT

A continuación, se presenta una propuesta de arquitectura SDN-IoT realizada por [24]. Como se puede observar en en la Figura 2.8, esta se compone de dispositivos IoT, un controlador IoT, un controlador SDN y el resto de la red SDN.

En primer lugar, los dispositivos IoT. En ellos destacan dos elementos principales: la interfaz de datos y el agente IoT.

1. **Interfaz de datos**. Perteneciente al plano de datos, se encarga de la transmisión de la información en paquetes según el protocolo establecido. Es importante destacar que antes de la realización de dicha transmisión, envía al agente IoT la dirección o identificador del destinatario.

2. **Agente IoT**. Perteneciente al plano de control, se comunica tanto con la interfaz de datos como con el controlador IoT. Se encarga de enviar al controlador IoT su dirección para que este la almacene. Además de esta dirección, cada vez que se desea enviar un paquete, también envía al controlador la que recibe de la interfaz de datos.

En segundo lugar, el controlador IoT. Perteneciente al plano de control, se comunica tanto con el agente IoT como con el controlador SDN. Tiene la función de almacenar la dirección o identificador perteneciente a cada uno de los agentes IoT. De esta manera, cuando un agente IoT le envíe la dirección del destinatario, debe buscarla en sus tablas. Una vez encontrada, mediante algún algoritmo calcula la ruta que debe seguir

el paquete y genera las reglas para su reenvío. Realizado esto, comunica dichas reglas al controlador SDN.

En tercer lugar, el controlador SDN. Este se comunica con la red SDN y el controlador IoT. Tiene la función de implementar en los dispositivos de conmutación de la red las reglas para el reenvío de paquetes que recibe del controlador IoT.

Finalmente, la red SDN. En ella se encuentran el resto de dispositivos de la red, entre los que destacan los dispositivos de conmutación (p. ej. *switches*) capaces de encaminar los paquetes para que lleguen a su destino.

Capítulo 3

Protocolos y controladores en el entorno SDN

3.1. Introducción

En este capítulo se abordan una serie de cuestiones relacionadas con las comunicaciones y los distintos controladores en SDN. En primer lugar, se exponen distintos protocolos usados en las *Southbound* APIs, haciendo inciso en el protocolo más destacado, OpenFlow. Seguidamente, se analizan las principales características de las *Northbound* APIs. Para finalizar, se describen las funciones y propiedades de un controlador SDN. Al mismo tiempo, se analizan las características de una serie de controladores que se usan para el desarrollo de las simulaciones.

3.2. Southbound APIs

Las *Southbound* APIs son usadas a través de la SBI para establecer las comunicaciones entre el controlador y los distintos elementos de conmutación presentes en la red. Como se ha comentado anteriormente, su función principal es la de insertar reglas de flujo en las tablas de los dispositivos del plano de datos, consiguiendo así un control centralizado de las políticas de red.

De entre los distintos protocolos establecidos dentro las *Southbound* APIs, destaca uno, OpenFlow, un protocolo ideado en 2012 por la ONF, impulsora del desarrollo de SDN. OpenFlow se utiliza hoy en día como protocolo de comunicación estándar en

SDN y es por lo que se profundizará más adelante en sus características. Además de OpenFlow, existen otros protocolos de comunicación, algunos se apoyan en el uso de él y otros no. Como ejemplos de estos protocolos se tienen los siguientes:

- **OvSDB**. Es un protocolo implementado para comunicaciones que usan Open vSwitch[1] debido a que estos permiten el uso combinado junto a OpenFlow. Dentro de las cualidades más destacadas de OvSDB según [25] están:

  1. La configuración del conjunto de controladores al que debe conectarse un datapath OF[2].
  2. La creación, modificación y eliminación de puertos en el datapath OF.
  3. La creación, modificación y eliminación de colas
  4. La configuración de políticas de QoS para posteriormente asignarlas a las colas.

- **OF-Config**. Según lo especificado en [26], tiene como misión principal el posibilitar la configuración remota de *switches* OF. Para ello se vale del uso de dos conceptos:

  1. *OpenFlow Logical Switch*. Se define como la abstracción lógica de un *switch* OF. En él, OF-Config realiza las operaciones necesarias para que este se pueda comunicar con un controlador OpenFlow a través del protocolo OF.
  2. *OpenFlow Capable Switch*. Tiene la intención de ser equivalente a un elemento de red, ya sea físico o virtual, con la capacidad de alojar uno o más *OpenFlow Logical Switch* gracias a la división lógica de sus recursos OF, como pueden ser los puertos o las colas. El protocolo OF-Config es el encargado de realizar la asociación de estos recursos de manera dinámica, asegurándose de que cada uno de los distintos *OpenFlow Logical Switch* sea capaz de gestionar sin problemas de sus recursos asignados.

[1] Es un *switch* virtual bajo la licencia de código abierto Apache 2.0. Permite redirigir tráfico entre distintas máquinas virtuales y también con la red física.

[2] El datapath de OpenFlow es la línea de procesado de datos de los flujos OpenFlow.

- **OpFlex**. Es un protocolo ideado y diseñado por la empresa estadounidense Cisco en 2014 y su uso está orientado para arquitecturas SDN que implementen el control declarativo[3]. Al igual que hará OpenFlow, se basa en la abstracción de las políticas de red de los distintos dispositivos de conmutación en favor del controlador. Para la transferencia hacia los dispositivos de las distintas políticas dictadas por el controlador, se usan tanto JSON como XML. Además, cualquier dispositivo, virtual o físico puede implementarlo, lo que genera una buena interoperabilidad entre los dispositivos de distintos vendedores.[27]

- **NetConf**. Según se puede extraer de [28], NetConf es un protocolo de configuración de red, estandarizado y creado por la IETF en 2006. Surge como necesidad de los operadores para facilitar la configuración de los distintos dispositivos de red, ya que las redes suelen estar integradas por equipos de distintos fabricantes y debido a que la mayoría de estos se configuran a través de la línea de interfaz de comandos (CLI), normalmente difieren en la manera de ejecutar las órdenes.

  NetConf unifica la manera en la que se administran estos dispositivos (realizar acciones de red determinadas, definir métodos para configurar sus bases de datos, etc.) y para ello se vale del lenguaje de modelización YANG, capaz de definir el contenido de las órdenes enviadas a través de NetConf.

3.2.1. OpenFlow

Introducción

Ideado en 2009 en la Universidad de Stanford en su primera versión, OpenFlow surge como un proyecto para el desarrollo de un protocolo de comunicación para SDN. Dos años más tarde, junto con la creación de la ONF y la posterior adquisición del proyecto por su parte, OpenFlow se actualiza a la versión 1.1, con ligeros cambios sobre la versión anterior, y a finales de año lanza la versión 1.2.. Al año siguiente, en junio de 2012, se lanza la versión del protocolo 1.3, con grandes cambios respecto

[3]El controlador declara las necesidades de la aplicación y a su vez informa a los distintos dispositivos de red como han de hacer para cumplirlas. Llegados a este punto, el controlador se desentiende y son los dispositivos los que con su lógica interna llevan a acabo las distintas acciones necesarias para satisfacer la demanda de la aplicación.

a las tres anteriores y con el objetivo de convertirse en una versión estable para así poder comercializarse [29]. Hasta la fecha han surgido variaciones de la 1.3 y se han implementado las versiones 1.4 y 1.5, pero se desarrollarán los conceptos teóricos de la 1.3, ya que se considera la versión estándar y son capaces de entenderla tanto los *switches*, como los controladores que se van a usar en el desarrollo de las simulaciones y de la aplicación práctica.

A continuación, se desarrollan las principales características de la versión 1.3 del protocolo OpenFlow según el documento especificativo de la ONF [30].

Características y tipos de *switches*

De entre los tipos de *switches* presentes en OpenFlow existen dos variantes: los puros y los híbridos.

Por un lado, los *switches* OF puros solo soportan operaciones OpenFlow, es decir, todos los paquetes son procesados por la *pipeline* de OpenFlow y no pueden ser procesados de otro modo.

Por otro lado, los *switches* OF híbridos soportan tanto las operaciones OpenFlow como las operaciones de un *switch* tradicional, es decir los paquetes pueden ser procesados tanto por una como por otra *pipeline*. Dado el caso, un paquete puede ser procesado por la *pipeline* de OpenFlow y luego ser enviado a través de unos puertos reservados a la de procesado tradicional.

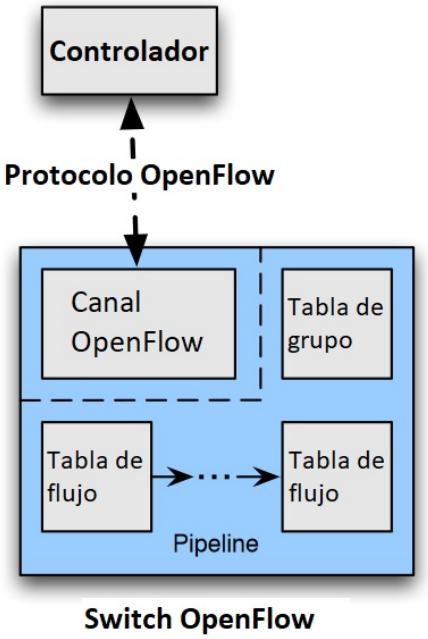
Al mismo tiempo, ambos comparten una serie de elementos que caracterizan a los *switches* OF y que se observan en la Figura 3.1.

Primeramente, el controlador puede insertar, actualizar y eliminar entradas de flujo en las tablas de flujo. Estas acciones son transmitidas al *switch* mediante el uso del protocolo y el canal OpenFlow.

Seguidamente, las tablas de flujo están compuestas por varias entradas de flujo, que a su vez se componen de: *match fields*, contadores y un conjunto de instrucciones para aplicar a los paquetes que hacen *match*.

Finalmente, la tabla de grupo está formada por entradas de grupo, que a su vez están formadas por listas de *action buckets* o conjunto de acciones, que se especifican dependiendo del grupo.

Figura 3.1: Principales componentes de un *switch* OpenFlow [30]



Tipos de puertos

Los puertos OpenFlow son interfaces a través de las cuales se transmiten los paquetes a través de los distintos elementos presentes en la red, en este caso *switch* y controlador. Dentro de este protocolo se especifican tres tipos de puertos: físicos, lógicos y reservados.

En primer lugar, los puertos físicos hacen mención a los intrínsecos del *switch*, es decir sus propios puertos *hardware*.

En segundo lugar, los puertos lógicos son aquellos que suponen una abstracción de los hardware mediante métodos no OpenFlow, como puede ser mediante el uso de VLAN, *tunnels*, *loopback*, etc.

Finalmente, los puertos reservados son definidos por el estándar OpenFlow y ejemplos de estos son los siguientes:

- **ALL**. Representa el conjunto de puertos que puede usar un *switch* para retransmitir un paquete.
- **CONTROLLER**. Hace referencia al puerto que comunica al controlador OF con el canal OF.
- **IN-PORT**. Representa el puerto de entrada del paquete.

- **LOCAL**. Es el puerto mediante el que se accede a la gestión interna del *switch*. Además, es el que se conecta físicamente al controlador, ya que permite a entidades remotas interactuar con sus servicios de red a través de la red OF.

En caso de ser un *switch* híbrido también existen:

- **NORMAL**. Representa las acciones no OpenFlow que realiza el *switch*, es decir, a través de este puerto se realiza *switching* tradicional.
- **FLOOD**. Es el puerto que a través del cual se inunda la red de manera tradicional.

Tablas de flujo

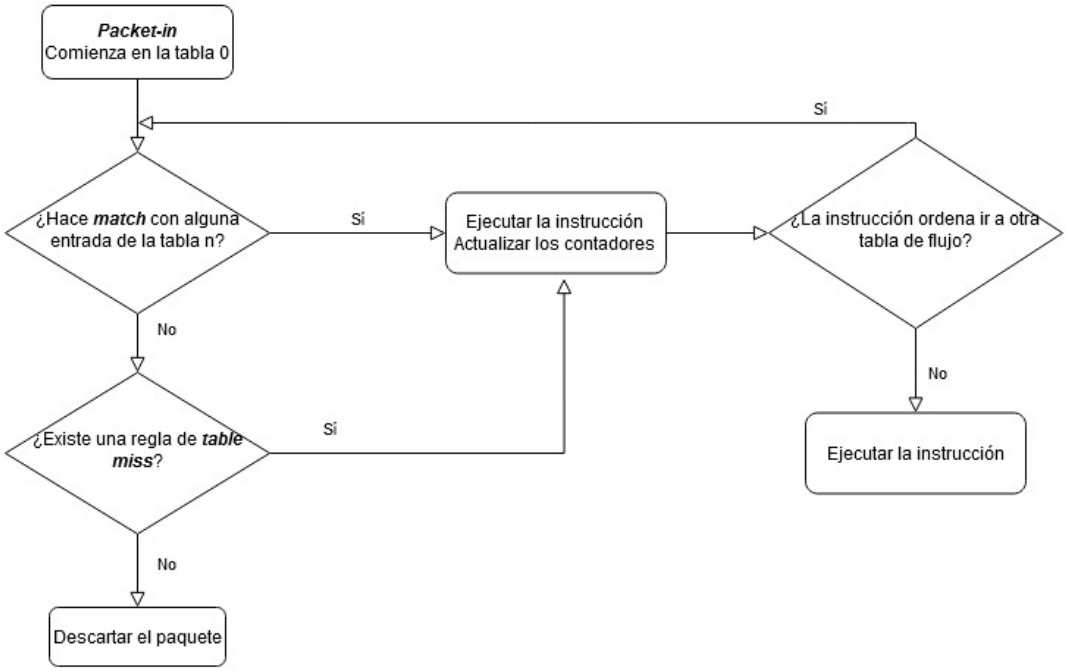
Las tablas de flujo no son más que un conjunto de entradas, en la versión 1.3 de OpenFlow pueden coexistir n tablas, siendo la primera la tabla 0 y n-1 la última tabla. Los paquetes son procesados por las entradas de una única tabla de flujo. Cada entrada de flujo tiene los elementos que se observan en la Tabla 3.1 y que aquí se describen:

Match fields	Prioridad	Contadores	Instrucciones	*Timeouts*	*Cookies*

Tabla 3.1: Elementos de una entrada de flujo

- ***Match fields***. Es usado para seleccionar paquetes que cumplan los requisitos marcados en este campo. En esta versión de OF los *matches* se pueden hacer en base a varios criterios, p. ej.: El puerto de entada, la dirección IPv4 o IPv6 de destino o de la fuente, el puerto de destino TCP o UDP, protocolos IPv4 o IPv6, etc.
- **Prioridad**. Es un número que va del 0 al 65535, siendo ordenada esta de mayor a menor, es decir una regla con un número de prioridad más alto se ejecuta por delante de una que tenga un número menor. Es importante mencionar que la prioridad 0 también se conoce como *table miss*.
- **Contadores**. Se actualizan por paquete que coincide, p. ej. por cada regla se observan el numero de bytes totales que se han procesado.

Figura 3.2: Diagrama de flujo del proceso de paquetes en OF [30]



- **Instrucciones**. Acciones que se realizan con los paquetes. Entre estas están por ejemplo el mandar el paquete por un cierto puerto, el descartar un paquete, ir a una cierta tabla de flujo, etc.
- ***Timeouts***. Es el tiempo máximo o tiempo de inactividad para que la entrada expire de la tabla de flujos.
- ***Cookies***. Datos opacos cuyos valores son elegidos por el controlador. Pueden ser usados por el controlador para realizar estadísticas.

Teniendo en cuenta todo esto, mediante el diagrama de flujo presente en la Figura 3.2 se explica el proceso que siguen los paquetes en OpenFlow.

Mensajes OpenFlow

A través del canal OpenFlow se envían mensajes que comunican al controlador con el *switch* y viceversa, dentro de estos se distinguen tres tipos de mensajes: controlador-*switch*, simétricos y asíncronos.

En primer lugar, los mensajes del tipo controlador-*switch*, son aquellos que como su nombre indica, son enviados por parte del controlador al *switch* y pueden o no requerir una respuesta de este. Dentro de estos destacan los siguientes:

- ***Features***. Se suele enviar justo tras el establecimiento del canal OpenFlow. Tras recibirlo el *switch*, este responde enviando sus características (versión de OF, nº de tablas soportadas, etc.) al controlador.
- ***Configuration***. Con este tipo de mensajes el controlador es capaz de establecer y consultar parámetros de configuración del *switch*.
- ***Packet-out***. Envía órdenes al *switch* para retransmitir un paquete por un puerto específico.
- ***Modify-State***. Se envían para gestionar el estado del *switch*, como por ejemplo: añadir, borrar o insertar entradas de flujos en las tablas.
- ***Read-State***. Tienen la función de recolectar información (estadísticas, capacidades, configuración) presente en el *switch*.

En segundo lugar, los mensajes simétricos, son enviados tanto por el controlador como por el *switch* sin una solicitud previa. Dentro de estos destacan:

- ***Hello***. Este mensaje se envía en el establecimiento de la conexión.
- ***Echo***. Necesita de una respuesta o *reply*, y es enviado para verificar que sigue existiendo conexión entre ambos dispositivos.

Por último, los mensajes asíncronos, son enviados por parte del *switch* al controlador, sin que este les solicite su envío. Dentro de estos se encuentran:

- ***Packet-in***. Se envía cuando el *switch* no sabe gestionar el paquete recibido. Esto tiene que estar especificado en su tabla del flujo.
- ***Flow-Removed***. Informa al controlador del borrado de una tabla o una entrada de flujo en el *switch*.
- ***Port-status***. Informa de cambios en el estado de un puerto.
- **Error**. Tiene la misión de notificar la existencia de cualquier tipo de problema.

Todos estos mensajes son enviados mediante el uso del protocolo TCP a través del puerto 6633 y en caso de necesitar el cifrado de mensajes, se establecería la conexión mediante el protocolo TLS.

Interrupción en la conexión

En el supuesto caso de que la conexión entre el *switch* y el controlador cayera, existen dos modos de actuación: *fail secure mode* y *fail standalone mode*.

- ***Fail secure mode***. Consiste en seguir implementando las reglas de las tablas de flujo presentes en el *switch* hasta que estas expiren tras un determinado tiempo o *timeout*. Una vez expiren, todos los paquetes recibidos por el *switch* son descartados.

- ***Fail standalone mode***. Suele aplicarse en los *switches* híbridos, ya que en este modo todo los paquetes son reenviados al puerto NORMAL. De esta manera los paquetes se reenvían a través de la red de manera tradicional.

3.3. Northbound APIs

La *Northbound Interface* es clave dentro de SDN, puesto que es capaz de generar la abstracción necesaria para la programación de la red. Como se ha comentado previamente, es la responsable de comunicar al plano de control con el plano de aplicación mediante el uso de APIs, más concretamente *Northbound* APIs. Al contrario que en la interfaz *Southbound* donde se tiene OpenFlow, aquí no existe un protocolo o API de comunicación estandarizado y es por esta razón que en 2012 la ONF decide crear un grupo llamado *NBI Working Group*, con la misión de desarrollar un método de comunicación estándar. Hasta la fecha dicha misión ha dado resultados negativos hasta la fecha, ya que aún existe esa ausencia. Pese a ello, para la implementación de las *Northbound* APIs, actualmente existen dos líneas de actuación: *intent-based* y *controller-based*. [31]

- ***Intent-based***. Se basan en el uso de tecnologías estandarizadas para realizar las comunicaciones, como por ejemplo pueden ser las REST APIs o Java APIs.

- ***Controller-based***. Se basan en la implementación de APIs propias para cada software controlador, es decir que si el controlador es X usará su X API asociada únicamente a él.

Para el desarrollo de las simulaciones se usarán las *intent-based*, puesto que estas son las que implementan los controladores ODL y RYU.

3.4. Controladores

Como se ha dicho más arriba, el controlador se integra dentro del plano de control, siendo el encargado de la gestión de la red a través del control de los flujos. Este no es más que un software integrado dentro de un servidor y es por ello que existen diversos modelos, como los presentes en la Tabla 3.2. Pese a esta cantidad de software y poder diferenciarse en múltiples aspectos, todos ellos comparten las funciones principales de un controlador, como por ejemplo lo son la gestión de los elementos de red, gestión del tráfico, estadísticas, etc. Las diferencias más notables entre todos ellos son el lenguaje usado para su desarrollo, el protocolo o protocolos que soportan en la SBI y las APIs usadas a través de la NBI. [31]

Controlador	**Lenguaje de programación**	**SBI**	**NBI**
OpenDayLight	Java	OF v1.0 - v1.3, OvSDB, SNMP, NETCONF	REST, REST-CONF, NET-CONF
Floodlight	Java	OF v1.0 - v1.3	REST, Java RPC, Quantum
ONOS	Java	OF v1.0 - v1.3	REST, Neutron
RYU	Python	OF v1.0 - v1.5, OF-CONFIG, NETCONF	REST, Quantum
POX	Python	OpenFlow v1.0	ad-hoc
NOX	C++	OpenFlow v1.0	ad-hoc
Beacon	Java	OpenFlow v1.0	adhoc

Tabla 3.2: Lista de algunos controladores presentes en el mercado [31]

Dentro de los distintos controladores, se analizan en profundidad en los siguientes apartados OpenDayLight y RYU, puesto que son los que se van a utilizar en las simulaciones.

3.4.1. OpenDayLight

OpenDayLight es un proyecto software de código abierto ideado y creado por *The Linux Foundation* en abril de 2013 con el objetivo de promover las tecnologías SDN y NFV.

Hasta la fecha se han lanzado 12 versiones, siendo lanzada la primera en Febrero de 2014 y la última en Marzo de este presente año. Como curiosidad, se observa que todas las versiones tienen el nombre de elementos de la tabla periódica y están lanzadas según el orden de estos en dicha tabla. La versiones son las siguientes:

Versión	**Fecha de lanzamiento**
Hydrogen	Febrero 2014
Helium	Octubre 2014
Lithium	Junio 2015
Beryllium	Febrero 2016
Boron	Noviembre 2016
Carbon	Junio 2017
Nitrogen	Septiembre 2017
Oxygen	Marzo 2018
Fluorine	Agosto 2018
Neon	Marzo 2019
Sodium	Septiembre 2019
Magnesium	Marzo 2020

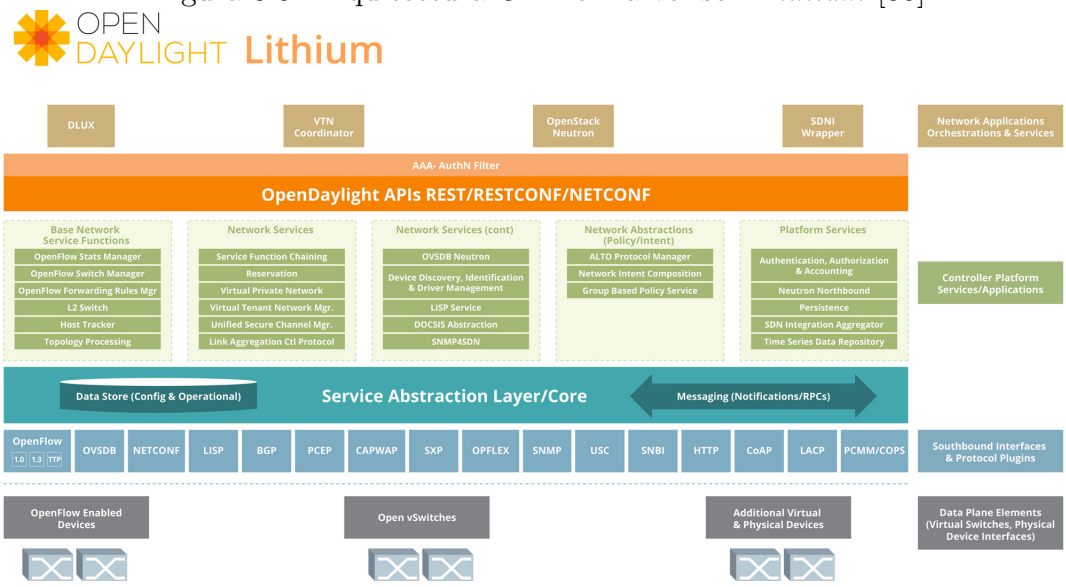
Tabla 3.3: Versiones de ODL [32]

El desarrollo de las posteriores simulaciones estará centrado en el uso de la versión 3 o también llamada versión Lithium, cuya arquitectura se observa en la Figura 3.3.

Arquitectura

La versión Lithium, al igual que las posteriores, se basa en el uso del MD-SAL. En este, los elementos y aplicaciones de red son tratados como objetos o modelos que luego son procesados por la SAL. Al mismo tiempo, la organización de la arquitectura se organiza por capas.

Figura 3.3: Arquitectura ODL en la verisón *Lithium* [33]



En primer lugar, los elementos de red presentes, como pueden ser los dispositivos OpenFlow, los Open vSwitches y otros dispositivos tanto físicos como virtuales, forman el conjunto del plano de datos.

En segundo lugar, protocolos como OpenFlow, OvSDB, NETCONF y SNMP forman parte de las herramientas y protocolos usados por la SBI.

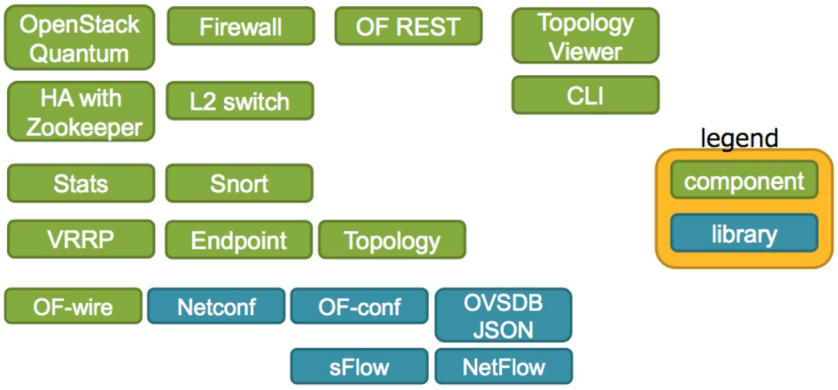
En tercer lugar, la SAL es una capa usada como mecanismo de intercambio de datos entre modelos YANG que representan dispositivos de red y las aplicaciones. Los modelos YANG aportan una descripción generalizada de las capacidades de los elementos de red o aplicaciones sin tener que conocer explícitamente las especificaciones de ellos. Dentro del SAL, los modelos se definen simplemente por sus respectivos papeles en una interacción dada. [34]

En cuarto lugar, servicios de OpenFlow como el gestor del *switch*, de estadísticas o reglas de flujo, servicios de red como VPN, y políticas de seguridad, se alojan en la capa de controlador.

En quinto lugar, dentro de la NBI se observan, como se comentó en la Tabla 3.2, las distintas APIs usadas por ODL, tales como REST, RESTCONF y NETCONF, usadas para el intercambio de información entre las aplicaciones y el controlador.

Finalmente, aplicaciones gráficas como DLUX y plataformas de infraestructura como servicio como OpenStack Neutron se encuentran dentro de la capa de aplicaciones de red y servicios de orquestación.

Figura 3.4: Componentes y librerías en RYU [35]



3.4.2. RYU

Ryu, que significa flujo en japonés, es un *open source framework* SDN que hace las veces de controlador. Creado y desarrollado por la empresa nipona de telecomunicaciones NTT, se distingue de los demás controladores por su agilidad y flexibilidad. La primera viene dada por el hecho de ser un *framework* SDN en lugar de ser un gran controlador con un montón de propósitos, y la segunda la consigue gracias a versatilidad de sus *Northbound* APIs.

Ryu está basado en el uso de componentes y librerías junto con APIs bien definidas para hacer más fácil la labor de crear nuevas aplicaciones de gestión y control de red a los desarrolladores. De esta manera, los desarrolladores pueden modificar componentes existentes, crear nuevos y posteriormente combinarlos para crear una nueva aplicación. Las librerías y distintos componentes presentes se observa en la Figura 3.4. [35]

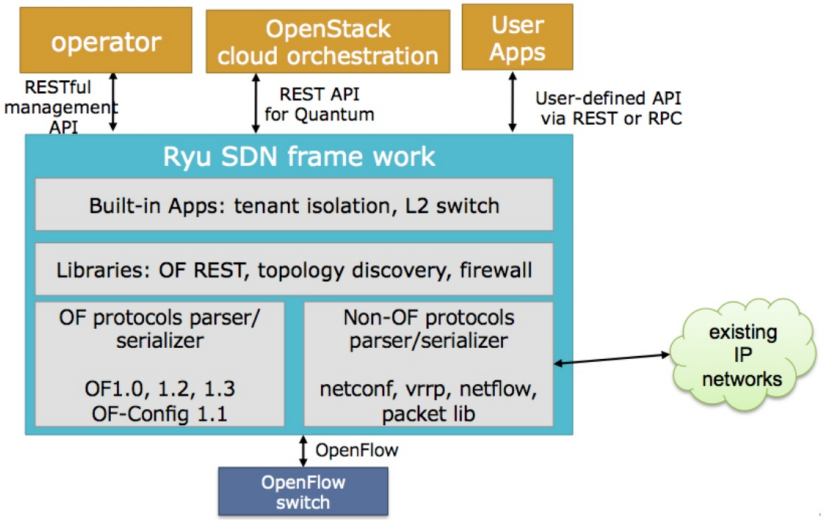
Arquitectura

En cuanto a la arquitectura definida en RYU, existen tres partes diferenciales: los elementos de red, el *framework* SDN y las aplicaciones. Todo esto se observa en la Figura 3.5.

En primer lugar, en cuanto a los elementos de red, principalmente estos son *switches* OpenFlow, pero pueden existir otros. Los protocolos soportados por la SBI son: OpenFlow en las versiones 1.0 - 1.5, NETCONF, OF-Config, etc.

En segundo lugar, el *framework* SDN, que ejerce de controlador y está formado por las librerías y componentes mencionadas anteriormente y presentes en la Figura 3.4.

Figura 3.5: Arquitectura RYU [35]



Finalmente, las aplicaciones, entre las que se encuentran un operador para la red, un orquestador y las aplicaciones creadas por el usuario.

Capítulo 4

Realización de pruebas en entorno simulado. Mininet

4.1. Introducción

Mininet es una herramienta *software* de emulación de red que permite la creación de *hosts*, enlaces, *switches* y controladores virtuales. Dentro de sus características, destacan los hechos de que los *hosts* están basados en sistemas Linux y los *switches* son capaces de soportar OpenFlow. Esto hace que se puedan realizar arquitecturas SDN grandes y personalizables.

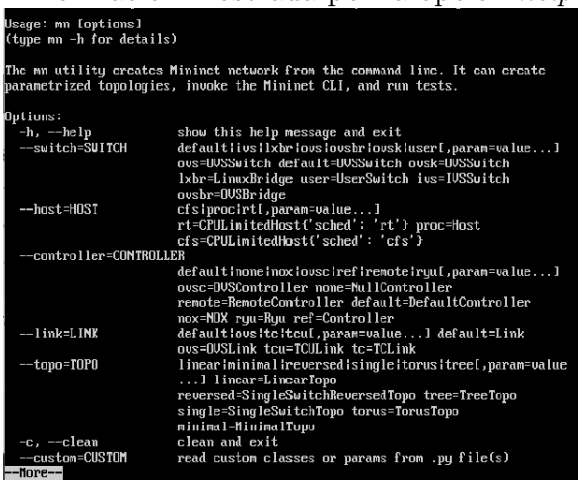
A lo largo de este capítulo se desarrollan varias simulaciones, usando para cada una de ellas los controladores ODL y RYU. Estas tienen como objetivo principal analizar de manera práctica las características de SDN y OpenFlow estudiadas en los Capítulos 2 y 3, respectivamente.

Al mismo tiempo, todo esto no sería posible sin la herramienta de captura de tráfico de paquetes, Wireshark, utilizada para visualizar e inspeccionar los mensajes OpenFlow enviados a través de la red.

4.2. Comandos Mininet

Tras instalar Mininet según los pasos seguidos en el Anexo A y antes del desarrollo de las simulaciones, se tiene que tener en cuenta el significado de una serie de comandos esenciales. Para un mejor análisis se dividen en cuatro grupos:

Figura 4.1: Información mostrada por la opción *help* en Mininet



- Comandos básicos
- Comandos de topología
- Comandos de *hosts*
- Comandos OpenFlow

4.2.1. Comandos básicos

Son considerados comandos básicos aquellos que son de ámbito general y que sirven para mostrar cierta información puntal. Dentro de estos destacan los siguientes:

En primer lugar, el comando *help*. Como su nombre bien indica, muestra una ayuda de las distintas opciones disponibles dentro de Mininet. Esta comando se puede ejecutar dos formas y el resultado de su ejecución se observa en la Figura 4.1.

Opción 1

```
$ sudo mn --help
```

Opción 2

```
$ sudo mn -h | more
```

En segundo lugar, el comando *clean*. Finaliza posibles procesos previos que se encuentren en ejecución y limpia el entorno de Mininet. De esta manera se asegura que el siguiente proceso a ejecutar va a ser el único en la cola. Se puede expresar de dos formas:

Figura 4.2: Información mostrada por la opción *nodes* en Mininet

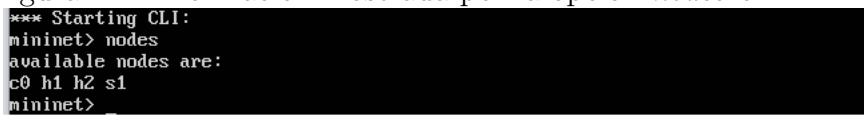
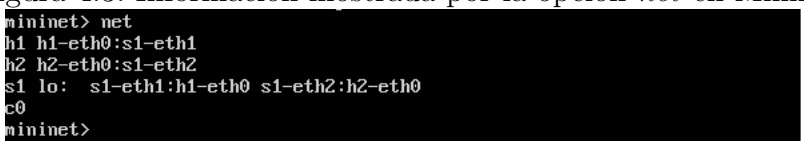


Figura 4.3: Información mostrada por la opción *net* en Mininet



Opción 1

```
$ sudo mn --clean
```

Opción 2

```
$ sudo mn -c
```

En tercer lugar, atendiendo a los comandos que se ejecutan una vez entrado al entorno de Mininet, destacan los siguientes:

Primero, la opción *nodes*. Muestra todos los nodos presentes en la red. El resultado de esta operación se observa en la Figura 4.2.

```
mininet> nodes
```

Segundo, la opción *net*. Muestra la manera en la que se conectan los dispositivos. El resultado de esta operación se muestra en la Figura 4.3.

```
mininet> net
```

Tercero, la opción *dump*. Presenta las direcciones IP y los pid de cada dispositivo. El resultado de esta operación se muestra en la Figura 4.4.

```
mininet> dump
```

Figura 4.4: Información mostrada por la opción *dump* en Mininet

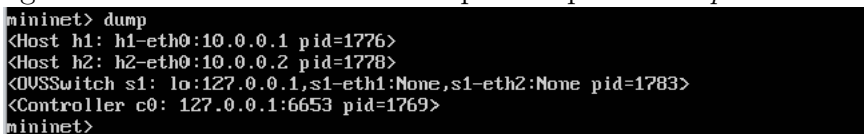
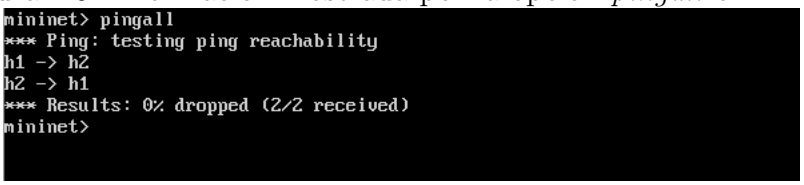


Figura 4.5: Información mostrada por la opción *pingall* en Mininet



Último, la opción *pingall*. Permite realizar un *ping* entre todas las máquinas presentes en la red. El resultado de esta operación se muestra en la Figura 4.5.

```
mininet> pingall
```

4.2.2. Comandos de topología

En Mininet existen distintos tipos de topologías para el diseño de las redes, de entre las que destacan: *minimal*, *single*, *linear* y *tree*. Además de esto, también se puede especificar si el controlador es remoto o no.

Minimal

En la topología *minimal* únicamente existe un *switch*, en este caso s1, y dos *hosts* que se conectan a él, h1 y h2. Para crear una topología de este tipo se ejecuta el siguiente comando:

```
$ sudo mn
```

Single

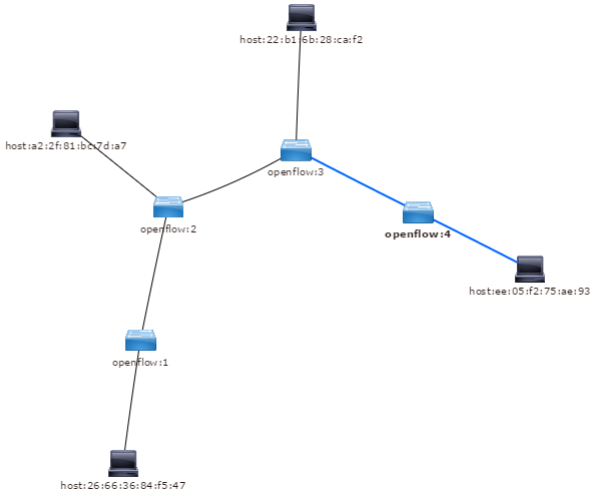
Para la topología *single* únicamente existe un *switch* (s1). Los *hosts* que se conectan a él, lo hacen de tal manera que el *host* 1 se conecta al puerto 1, el 2 al puerto 2 y así sucesivamente. En este caso, al contrario que para *minimal*, el número de *hosts* no está especificado. Para crear una topología de este tipo se ejecuta el siguiente comando:

Ej.: 4 *hosts*

```
$ sudo mn --topo=single,4
```

Existe otra variación de *single* llamada *reversed*, que, al contrario de *single*, conecta cada *host* al puerto contrario a su número. Es decir, el *host* 1 se conecta al puerto n,

Figura 4.6: Topología *linear* 4 *hosts*



el *host* n al puerto 1, el 2 al n-1 y así sucesivamente. Para crear una topología de este tipo se ejecuta el siguiente comando:

Ej.: 4 *hosts*

```
$ sudo mn --topo=reversed,4
```

Linear

La topología *linear* consiste en tener el mismo número de *switches* que de *hosts*. Cada *switch* tiene dos enlaces, uno a un *host* y otro a un *switch*. La estructura de esta topología se observa en Figura 4.6. El comando a ejecutar para crearla es el siguiente:

Ej.: 4 *hosts*

```
$ sudo mn --topo=linear,4
```

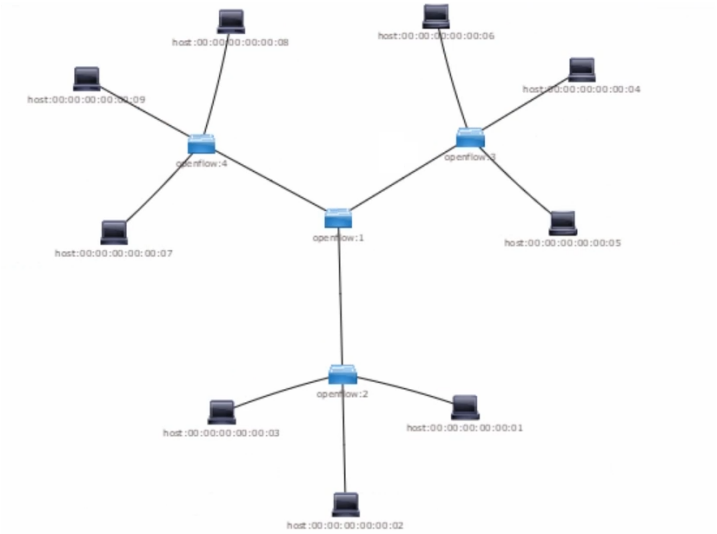
Tree

La topología en árbol o *tree* consta de dos partes: el *fanout* y la profundidad o *depth*. *Fanout* hace referencia al número de conexiones que tiene un *switch*. Por otro lado, *depth* hace referencia al número de *switches* existentes entre un *host* y el *switch* central. A modo de ilustración, en la Figura se muestra una topología en árbol con un *fanout* de 3 y una profundidad de 2. El comando es el siguiente:

Ej.: *fanout*=3, *depth*=2

```
$ sudo mn --topo=tree,depth=2,fanout=3
```

Figura 4.7: Topología *tree*, *fanout* 3, *depth* 2



Controlador

El controlador se puede establecer como remoto o en su defecto usar el que viene integrado en Mininet. En caso de que se quiera usar un controlador remoto se añaden la opciones *controller* y *remote*. El comando es el siguiente:

Ej.: IP controlador: 192.168.56.50

```
$ sudo mn --controller=remote,ip=192.168.56.50
```

Por otro lado, para el caso de el controlador integrado en Mininet no haría especificar la opción *controller*, ya que por defecto se usa este.

Ejemplo

Finalmente, teniendo en cuenta lo analizado anteriormente, en caso de querer realizar una red con una topología en árbol de profundidad 2, con 2 *hosts* por *switch* y con el controlador remoto el comando a ejecutar sería el siguiente:

```
$ sudo mn --controller=remote,ip=192.168.56.50 --topo=tree,
depth=2,fanout=2
```

4.2.3. Comandos de *hosts*

Considerando que como se comentó previamente, los *hosts* están basados en sistemas UNIX, dentro de estos se pueden ejecutar los mismos comandos que en cualquier

Figura 4.8: Información mostrada por la opción *ping* desde el *host* 1

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.44 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.376 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.043 ms
^C
--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4002ms
rtt min/avg/max/mdev = 0.042/0.390/1.446/0.543 ms
mininet>
```

Figura 4.9: Información mostrada por la opción *ifconfig* desde el *host* 1

```
mininet> h1 ifconfig
h1-eth0   Link encap:Ethernet  HWaddr e6:15:39:73:08:cd
          inet addr:10.0.0.10  Bcast:10.0.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1134 (1.1 KB)  TX bytes:1134 (1.1 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:24 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1712 (1.7 KB)  TX bytes:1712 (1.7 KB)

mininet>
```

máquina de este tipo. Si bien es cierto, primero se ha de hacer referencia a la máquina que se le quiera ejecutar comando, por ejemplo h1, y seguido se inserta la opción. Ejemplos de comandos que se pueden ejecutar son los siguientes:

En primer lugar, se puede realizar un *ping* desde una máquina a otra para comprobar la conexión. El resultado se observa en la Figura 4.8.

```
mininet> h1 ping h2
```

En segundo lugar, se puede realizar un *ifconfig* para comprobar el estado de las interfaces en la máquina. El resultado se observa en la Figura 4.9.

```
mininet> h1 ifconfig
```

En tercer lugar, se puede cambiar la dirección IP de la computadora que se desee mediante el siguiente comando:

```
mininet> h1 ifconfig h1-eth0 10.0.0.3 netmask 255.255.255.0
```

El resultado se observa en la Figura 4.10.

En cuarto lugar, se puede realizar un *netstat* para comprobar el estado de las conexiones de la máquina. El resultado se observa en la Figura 4.11.

Figura 4.10: Cambiar la dirección IP de un *host*

```
mininet> h1 ifconfig h1-eth0 10.0.0.3 netmask 255.255.255.0
mininet> h1 ifconfig
h1-eth0   Link encap:Ethernet  HWaddr e6:15:39:73:08:cd
          inet addr:10.0.0.3  Bcast:10.0.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1134 (1.1 KB)  TX bytes:1134 (1.1 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:24 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1712 (1.7 KB)  TX bytes:1712 (1.7 KB)

mininet>
```

Figura 4.11: Información mostrada por la opción *netstat* desde el *host* 1

```
mininet> h1 netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
mininet>
```

```
mininet> h1 netstat
```

Finalmente, se puede aplicar un *clear* para limpiar la terminal.

```
mininet> h1 clear
```

4.2.4. Comandos OpenFlow

Los comandos OpenFlow se integran dentro de los *switches* de la red y es por tanto que si se quiere ejecutar un comando relacionado con OF, este se tendrá que ejecutar a través del *switch*. Al mismo tiempo, es importante mencionar que Mininet utiliza por defecto el protocolo OpenFlow en su primera versión, en caso de querer cambiarla hay que indicarlo como una opción.

Primeramente, atendiendo a la cuestión de la versión de OpenFlow negociada entre en controlador y los diversos *switches*, esta se especifica en el comando de creación de la red de este modo:

```
$ sudo mn --switch=ovsk,protocols=OpenFlow13
```

El *switch* ovsk hace referencia al Open vSwitch, capaz de soportar dicha versión de OF.

Figura 4.12: Tabla de flujos en OF v1.3 usando *sh ovs-ofctl*

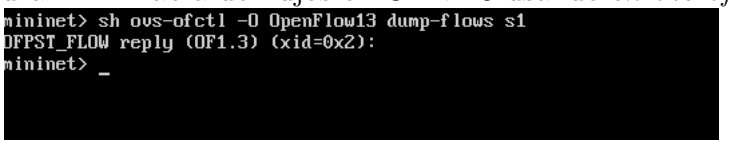
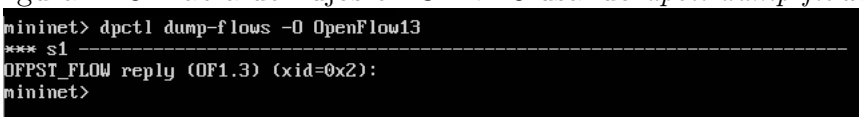


Figura 4.13: Tabla de flujos en OF v1.3 usando *dpctl dump-flows*



Seguidamente, en cuanto al despliegue de información de las diversas entradas de la tabla de flujos OF presentes en los *switches*, dependiendo de la versión de OF utilizada se usará un comando u otro. La información mostrada tras ejecutarlo se encuentra presente en la Figura 4.12.

Ej: Versión OF 1.0

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

Ej: Versión OF 1.3

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

En caso de querer mostrar las entradas de flujo para todos los *switches* presentes en la red, los comandos según la versión son los siguientes:

Ej: Versión OF 1.0

```
mininet> dpctl dump-flows
```

Ej: Versión OF 1.3

```
mininet> dpctl dump-flows -O OpenFlow13
```

El resultado de esta ejecución se observa en la Figura 4.13.

Terceramente, para observar la información acerca de las interfaces y cual es el controlador de cada uno de los *switches* OF presentes en la red se ejecuta el siguiente comando:

```
mininet> sh ovs-vsctl show
```

En la Figura 4.14 se observa el resultado de esta operación.

Figura 4.14: Información acerca de las distintas interfaces del *switch*

```
mininet> sh ovs-vsctl show
97060324-da2c-4cc5-ac47-5e2006b159a7
    Bridge "s1"
        Controller "ptcp:6654"
        Controller "tcp:127.0.0.1:6653"
        fail_mode: secure
        Port "s1-eth2"
            Interface "s1-eth2"
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth1"
            Interface "s1-eth1"
    ovs_version: "2.0.2"
mininet>
```

Figura 4.15: Tráfico según los puertos OF v1.3 usando *sh ovs-ofctl*

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-ports s1
OFPST_PORT reply (OF1.3) (xid=0x2): 3 ports
  port  1: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=969.266s
  port  2: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=969.265s
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=969.266s
mininet>
```

Finalmente, para analizar el tráfico que fluye por los diferentes puertos del *switch* se ha de ejecutar el siguiente comando, que varía según la versión de OF utilizada. La información mostrada tras ejecutarlo se encuentra presente en la Figura 4.15.

Ej: Versión OF 1.0

```
mininet> sh ovs-ofctl dump-ports s1
```

Ej: Versión OF 1.3

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

En caso de querer mostrar el tráfico a través de todos los puertos de todos los *switches* presentes en la red, los comandos según la versión son los siguientes:

Ej: Versión OF 1.0

Figura 4.16: Tráfico según los puertos OF v1.3 usando *dpctl dump-ports*

```
mininet> dpctl dump-ports -O OpenFlow13
*** s1 ------------------------------------------------------------------------
OFPST_PORT reply (OF1.3) (xid=0x2): 3 ports
  port  1: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=1025.536s
  port  2: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=1025.535s
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
           tx pkts=0, bytes=0, drop=0, errs=0, coll=0
           duration=1025.536s
mininet>
```

```
mininet> dpctl dump-ports
```

Ej: Versión OF 1.3

```
mininet> dpctl dump-ports -O OpenFlow13
```

El resultado de esta ejecución se observa en la Figura 4.16.

4.3. Controlador ODL

4.3.1. Introducción

Una vez instalado el controlador según los pasos indicados en el Anexo C.1, se procede a describir la topología de red escogida.

Se ha decidido usar una topología de red en árbol, con 4 *switches* y 3 terminales en los *switches* de los extremos. Esta proporciona la suficiente complejidad para abordar las simulaciones que se quieren realizar y de esta manera, poder comprobar de a través de la práctica las diversas funcionalidades de SDN y OpenFlow.

Para la realización de las simulaciones se ha dividido el apartado en tres puntos principales:

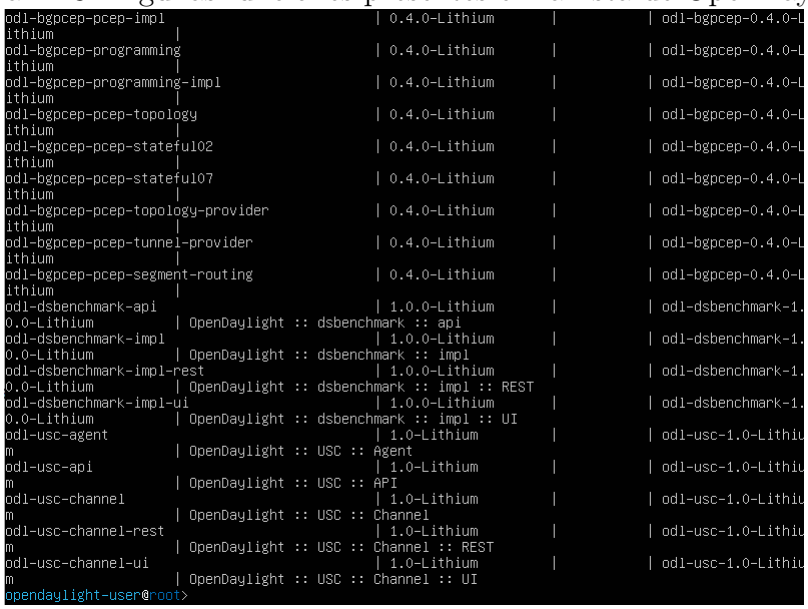
- Las **funciones del controlador**, donde se explican qué es una *feature* en ODL y cuáles se van a utilizar.
- La **creación y análisis de la red**, aquí se marcan los pasos para el montaje de la red en Mininet y se analiza el tráfico que esta genera mediante el uso de la herramienta Wirseshark.
- La utilización de la **App de Cisco**, **OFM**, cuyo procedimiento de instalación se encuentra en el Anexo C.1.1, ayudará a establecer distintas reglas de flujo en los *switches* de la red de manera más sencilla.

4.3.2. Principales funciones

Antes de tratar la funciones del controlador, se ha de conocer como se inicia este. Para ello, se accede al directorio dónde se encuentre el controlador y una vez ahí se ejecuta el siguiente comando:

Figura 4.17: Pantalla inicial en OpenDayLight

Figura 4.18: Algunas funciones presentes en la lista de OpenDayLight




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

Tras ejecutarlo, se cargará el controlador y se mostrará la ventana presente en la Figura 4.17.

Una vez realizado esto, ya el controlador está en funcionamiento, ahora lo único que queda es instalar la diferentes funcionalidades. Previamente, en caso de querer listar la distintas funcionalidades presentes, se ha de ejecutar la instrucción:

```
opendaylight-user@root>feature:list
```

El resultado de esta se observa en la Figura 4.18

Vista la lista de las diferentes *features* o funcionalidades en ODL, se procede a describir las que se van a instalar para el desarrollo de las simulaciones.

En primer lugar, **odl-restconf** instala la funcionalidad que permite el uso de la API RESTCONF. Esta es la utilizada para comunicar el controlador con la aplicación a través de la NBI.

En segundo lugar, **odl-l2switch-switch** asgina a los distintos *switches* de Mininet las tablas de flujo presentes en el controlador ODL y los relaciona con este.

En tercer lugar, **odl-mdsal-apidocs** es una función que sirve para explorar los documentos de la API.

Finalmente, **odl-dlux-all** lidia con lo relacionado con la GUI. Esta función en muchas ocasiones ralentiza la instalación de las distintas funciones, es por ello que también se suele inicializar en distintos componentes. Estos son:

- **odl-dlux-core**. GUI simple que muestra la topología de red.
- **odl-dluxapps-nodes**. Instala la aplicación que lidia con el inventario de los nodos.
- **odl-dluxapps-yangui**. Instala la aplicación YangUI para mostrar la interfaz YANG RESTCONF.
- **odl-dluxapps-yangvisualizer**. Instala la aplicación YangVisualizer para mostrar modelos YANG.

Tras conocer las distintas funcionalidades, el paso siguiente es instalarlas, para ello, se ejecuta la siguiente línea:

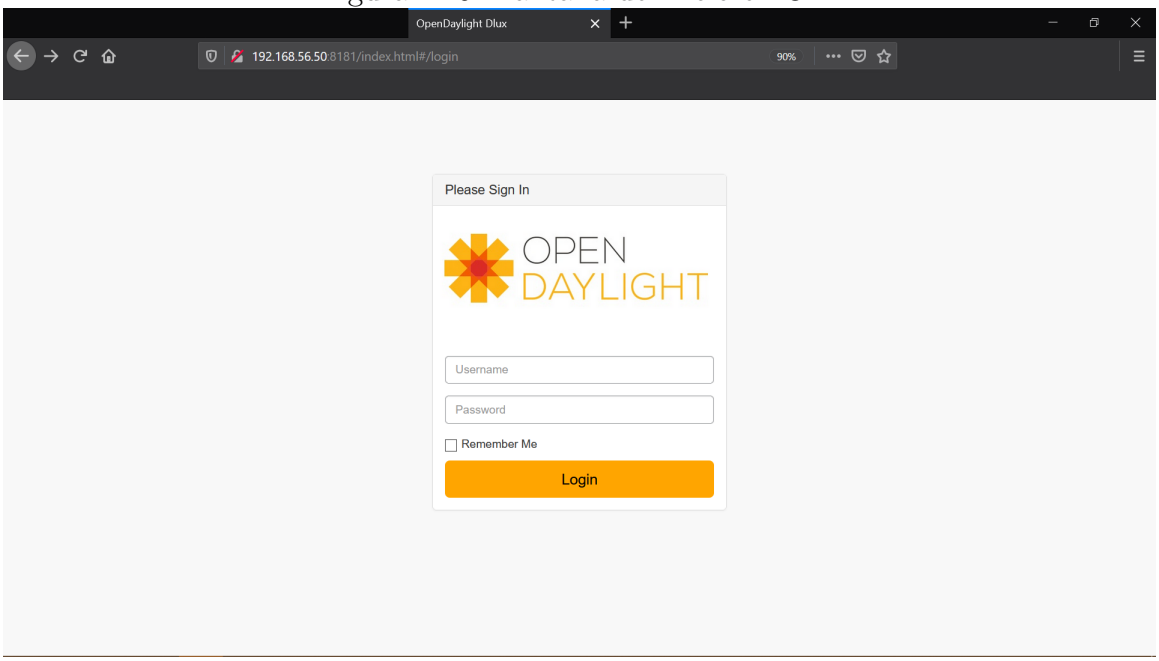
```
opendaylight-user@root>feature:install odl-restconf
odl-l2switch-switch odl-mdsal-apidocs odl-dlux-all
```

Tras instalarlas, el controlador estará listo para ser usado. Esto se comprueba accediendo a la página de inicio que ODL proporciona para el controlador. Para acceder a ella es necesario abrir un navegador e insertar la siguiente dirección en la barra de búsqueda:

DirecciónIPdelControlador:8181/index.html

Como se observa, la aplicación de ODL se conecta a través del puerto 8181 del controlador. El resultado de la dirección anterior se observa en la Figura 4.19.

Figura 4.19: Pantalla de inicio en ODL



Para acceder a la aplicación las credenciales a insertar son:

- *username*→admin
- *password*→admin

Una vez iniciada la sesión se mostrará la pantalla que se observa en la Figura 4.20. En este caso no existe ninguna topología puesto que aún esta no ha sido creada.

4.3.3. Creación y análisis de la red

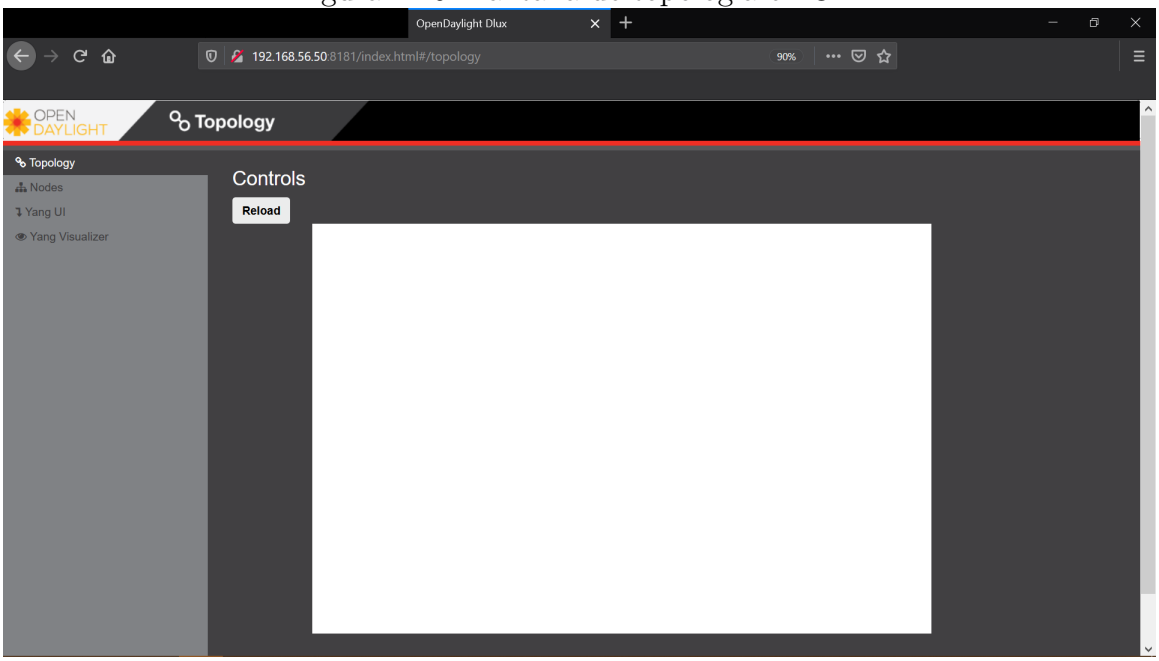
Creación

Para la creación de la red se dispone del controlador ODL ya iniciado (con la dirección IP 192.168.56.50) y de Mininet. El comando a ejecutar en Mininet para crear la red es el siguiente:

```
$ sudo mn --controller=remote,ip=192.168.56.50, --topo=tree,
depth=2, fanout=3 --switch=ovsk, protocols=OpenFlow13 --mac
```

La opción –mac establece direcciones MAC simples a cada uno de los *hosts*, es decir que en lugar de generar una dirección aleatoria, al *host* 1 se le asigna la MAC 00::01 y así sucesivamente para cada uno.

Figura 4.20: Pantalla de topología en ODL



Una vez creada la red, en la GUI de ODL se observan únicamente los distintos *switches* (ver Figura 4.21), ya que el controlador no conoce aún los distintos *hosts* existentes en la red. Para ello, se ejecuta la instrucción *pingall*, tras esto, se muestra la red en su totalidad (ver Figura 4.22).

Análisis

Creada y observada la topología de red, el siguiente paso es el análisis de los distintos elementos que la componen.

Figura 4.21: Topología de red en ODL únicamente con los *switches*

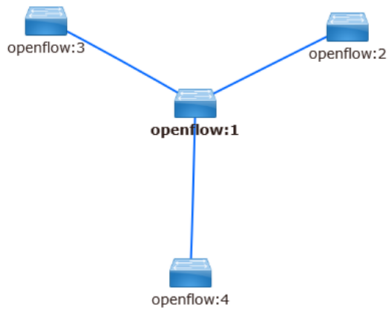
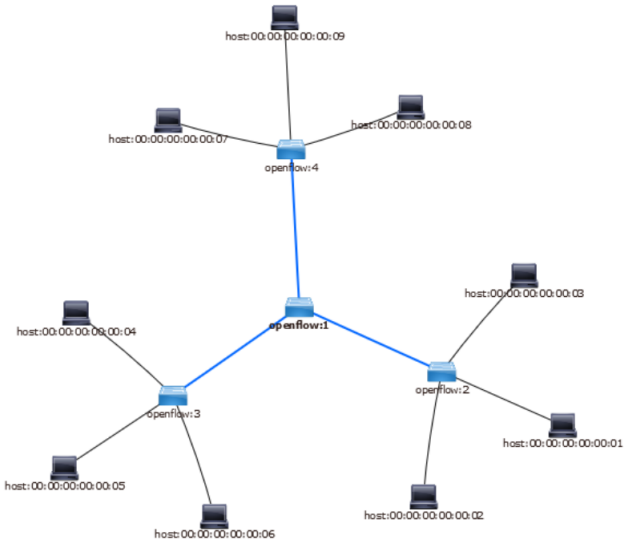


Figura 4.22: Topología de red en ODL completa



Primeramente, atendiendo al establecimiento de la conexión entre el controlador y los distintos *switches*, se tiene el envío de mensajes *Hello* por parte del controlador hacia el *switch* y viceversa. En la Figura 4.23 se observan a modo de ejemplo los mensajes enviados por cada uno de los *switches* al controlador. Paralelamente, también existen mensajes del tipo *Features*, que como ya se estudió en el Capítulo 3, se realizan tras el establecimiento del canal OF y tienen la función dar a conocer al controlador las características de los *switches* (p.ej.: versión de OF). En la Figura 4.24 se observan tanto los mensajes de petición o *request* enviados por parte del *switch* (marcados en azul), como las respuestas o *reply* enviadas por cada uno de los distintos *switches* (marcados en rojo). A parte, se inspecciona el contenido de un paquete en concreto, en el que se resalta en detalle la versión del protocolo OF usado, así como el tipo de mensaje al que se corresponde.

Segundamente, tras el establecimiento de la conexión, los distintos *switches* son inundados con tablas de flujo ya innatas en el controlador. A modo de ejemplo, se proceden a analizar las entradas de flujo de la tabla presente en el *switch* 1 (el nodo central de la red), que se observan en la Figura 4.25.

- La primera entrada tiene una prioridad de 2 y tiene como función el reenviar por los puertos 1 y 2 todos los paquetes recibidos por el puerto 3.
- La segunda entrada tiene una prioridad de 2 y tiene como función el reenviar por los puertos 2 y 3 todos los paquetes recibidos por el puerto 1.

Figura 4.23: Mensajes OF Hello capturados en Wirseshark (enviados por cada *switch*)

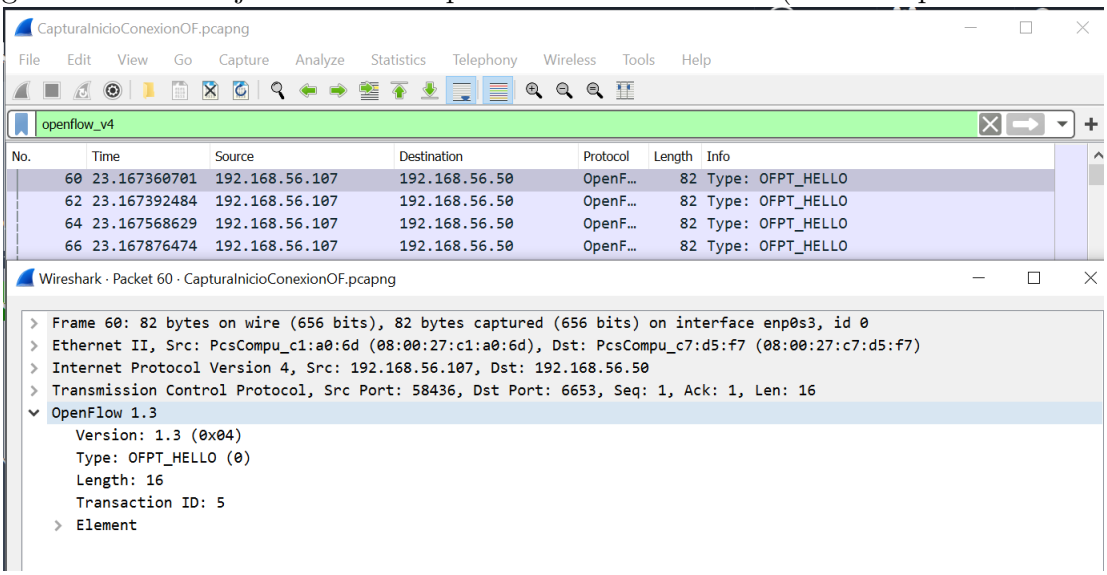


Figura 4.24: Mensajes OF Features capturados en Wireshark

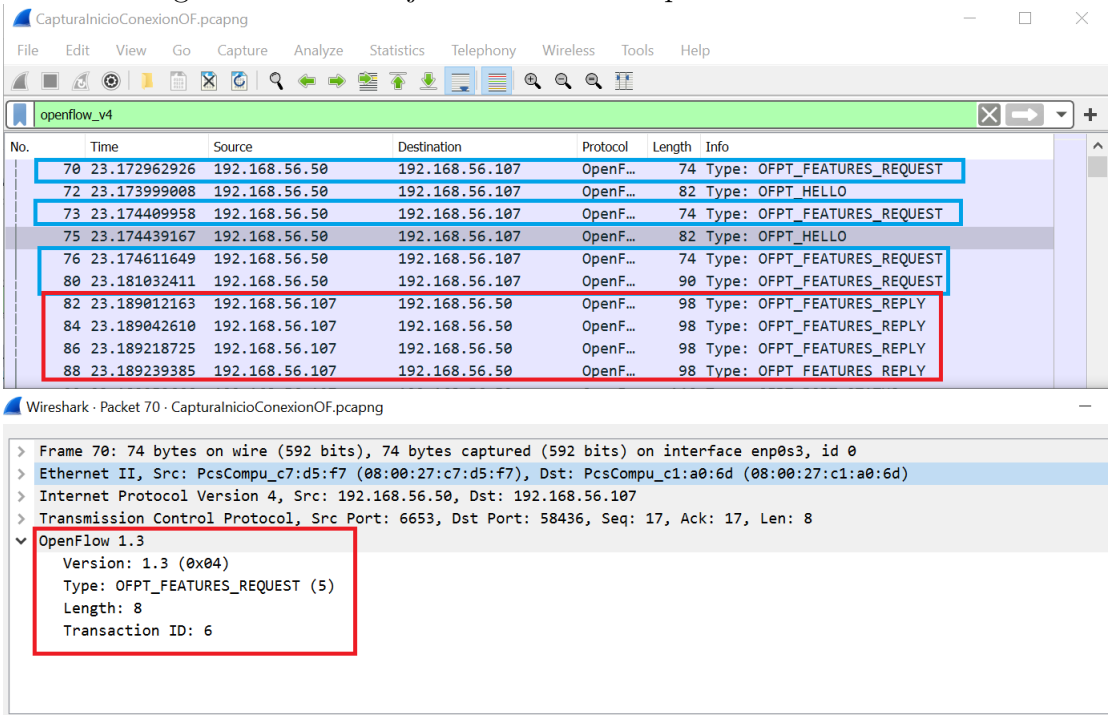


Figura 4.25: Tabla de flujos OF para *switch* 1

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x2b00000000000000, duration=433.135s, table=0, n_packets=90, n_bytes=64
68, priority=2,in_port=3 actions=output:2,output:1
 cookie=0x2b00000000000002, duration=433.135s, table=0, n_packets=93, n_bytes=67
62, priority=2,in_port=1 actions=output:3,output:2
 cookie=0x2b00000000000001, duration=433.135s, table=0, n_packets=93, n_bytes=67
62, priority=2,in_port=2 actions=output:3,output:1
 cookie=0x2b00000000000007, duration=436.975s, table=0, n_packets=264, n_bytes=2
2440, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000006, duration=436.975s, table=0, n_packets=0, n_bytes=0,
priority=0 actions=drop
mininet>
```

- La tercera entrada tiene una prioridad de 2 y tiene como función el reenviar por los puertos 1 y 3 todos los paquetes recibidos por el puerto 2.

- La cuarta entrada tiene una prioridad de 100 y se encarga de reenviar todos los paquetes de tipo LLDP al controlador. Esta tiene la prioridad más alta y por tanto, en caso de que se envíe cualquier paquete del tipo LLDP por la red y pase por este *switch*, se ejecutará esta acción por delante de las demás.

- La quinta y última entrada tiene una prioridad de 0, lo que se llama comúnmente un *table miss*, tiene la función de descartar todos los paquetes.

Para finalizar, en cuanto al análisis de las distintas interfaces presentes en los *switches*, el modo de actuación de estos frente a una interrupción en la conexión con el controlador y el modo de conexión con el controlador, se visualizan en la Figura 4.26 y se describen a continuación.

- Primero, se observa que existen dos controladores asociados a los *switches*, uno conectado al puerto 6655 y otro al 6653. El que se conecta al 6655 es el controlador que viene integrado por defecto en Mininet, y el restante es el controlador ODL, que se conecta vía TCP. El que es está activo es el ODL, ya que se muestra el mensaje *is connected: true*, en el caso de que no lo estuviera, no se mostraría nada, como en el primero.

- Segundo, el modo de actuación frente a una interrupción en el servicio con el controlador es el *secure mode*, es decir que todos los paquetes serían inmediatamente descartados una vez expiren las reglas de flujo del *switch*.

- Tercero, las distintas interfaces presentes en el *switch*, entre las que resalta la que tiene el nombre de *internal*. Esto es debido a que es a través de la cual se comunica con el controlador.

Figura 4.26: Información acerca de los *switches*

```
        Controller "tcp:192.168.56.50:6653"
            is_connected: true
        fail_mode: secure
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1-eth2"
            Interface "s1-eth2"
        Port "s1-eth3"
            Interface "s1-eth3"
    Bridge "s2"
        Controller "ptcp:6655"
        Controller "tcp:192.168.56.50:6653"
            is_connected: true
        fail_mode: secure
        Port "s2-eth2"
            Interface "s2-eth2"
        Port "s2-eth3"
            Interface "s2-eth3"
        Port "s2-eth4"
            Interface "s2-eth4"
        Port "s2-eth1"
            Interface "s2-eth1"
        Port "s2"
            Interface "s2"
                type: internal
    ovs_version: "2.0.2"
```

- Finalmente, la versión del *switch*, en este caso es la versión 2.0.2 de Open vSwitch.

4.3.4. App OFM

Para la creación de entradas de flujo para los distintos *switches* y así demostrar de manera práctica la versatilidad de SDN, se ha utilizado la aplicación de Cisco, OFM, cuyos pasos de instalación se encuentran en el Anexo C.1.1. Esta hace que la labor de insertar, editar y eliminar reglas de flujo sea más sencilla que en ODL puro. Para la adición de este *plugin* en ODL no basta más que instalar la siguiente *feature* a través del comando:

```
opendaylight-user@root>feature:install odl-openflowplugin-all
```

Una vez hecho esto, en la carpeta donde se hayan descargado los archivos pertenecientes a OFM, se ejecuta el comando *grunt* en la terminal. Tras realizarlo, al abrir el navegador e insertar la dirección IP del controlador y el puerto 9000, se muestra la pantalla de inicio de OFM (ver Figura 4.27). En esta se observa la topología de red presente, si se navega por la aplicación, se observan tres pantallas más, *Flow Management*, *Statistics* y *Hosts*, de las que se destaca la de *Flow Management* (ver Figura 4.28), debido a que en esta se van a poder añadir, editar o eliminar las distintas entradas de las tablas.

Figura 4.27: Pantalla inicial de OFM

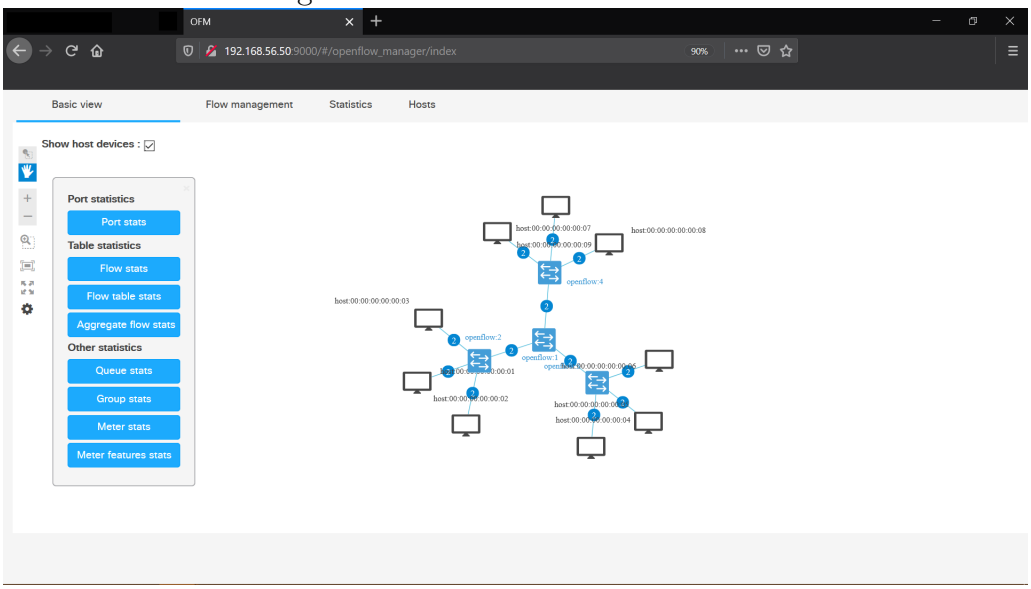


Figura 4.28: Pantalla de gestión de flujos OFM

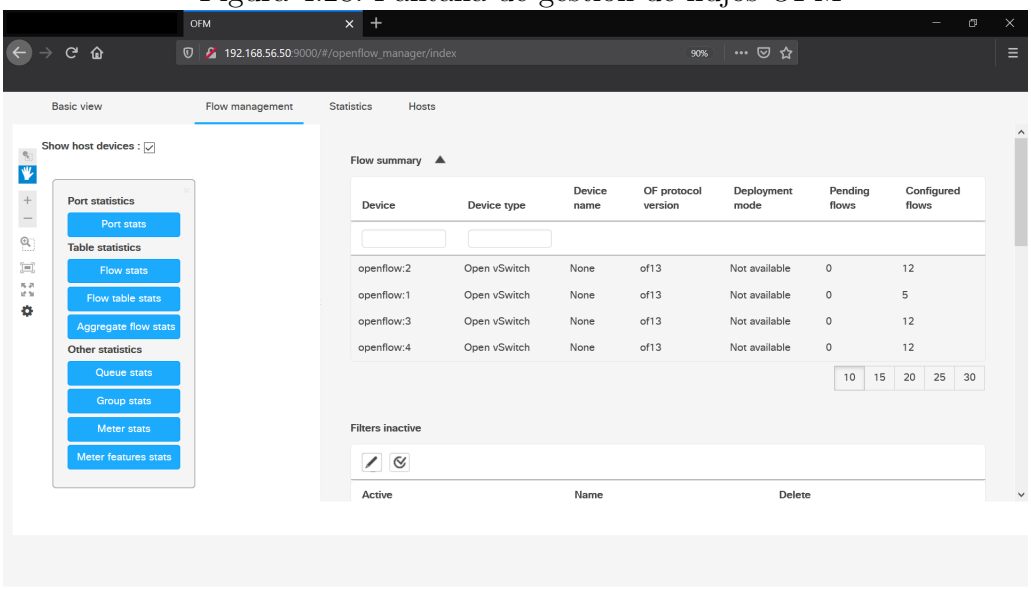
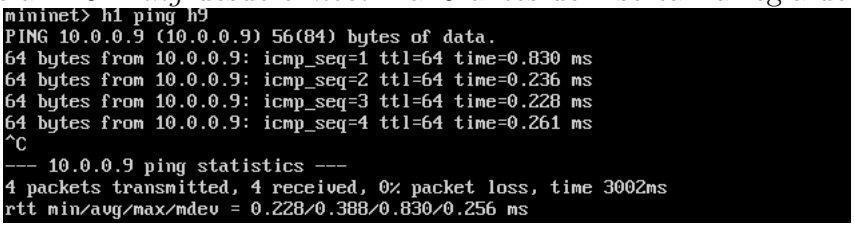


Figura 4.29: *Ping* desde el *host* 1 al 9 antes de insertar la regla de flujo



Una vez introducida la aplicación, para la comprobación y el análisis de SDN, se proponen dos ejemplos de estudio:

1. **La creación de dos "subredes"**. Mediante la inserción de un flujo en la tabla del *switch* central se hace que los *hosts* 1-6 no tengan comunicación con los *hosts* 7-9, pero a su vez, entre 1-6 y entre 7-9 si existe comunicación.

2. **Creación de un servidor web**. En este ejemplo se propone la creación de un servidor web al que se puede acceder mediante conexión TCP pero al que no se le puede hacer un *ping*, es decir se bloquean los paquetes ICMP.

Ejemplo 1

Para comprobar el efecto de la entrada de la regla de flujo en la tabla se comprueba previamente la conexión entre el *host* 1 y el *host* 9 mediante un *ping*. En la Figura 4.29, se muestra que la conexión es satisfactoria y por tanto no se pierde ningún paquete.

Realizado esto, yendo a la aplicación de Cisco, se inserta una regla OF para el *switch* 1, que se añadirá a la primera tabla, con el id drop3, con prioridad 1000, para todos los paquetes que entren por el puerto 3 (donde se encuentra el *switch* que conecta a los *hosts* 7, 8 y 9) y con la acción de descartar los mismos. Esto se observa en la Figura 4.30.

Una vez añadida la regla, en el panel principal se observa que se añade al *switch*, ya que drop3 se encuentra *ON DEVICE* (ver Figura 4.31). Es por tanto que, si ahora se realiza un *ping*, o se intenta cualquier otro tipo de conexión entre los *hosts* 1-6 y los 7-9, está será negativa. A modo de ejemplo, si se realizan diversos *pings*, uno del *host* 1 al *host* 9, otro del 1 al 5 y otro del 9 al 7, se puede comprobar según las Figuras 4.32, 4.33 y 4.34, que el primero, como era de esperar, no ha sido satisfactorio, pero en cambio, los otros dos si lo han sido.

Figura 4.30: Regla de flujo para el ejemplo 1

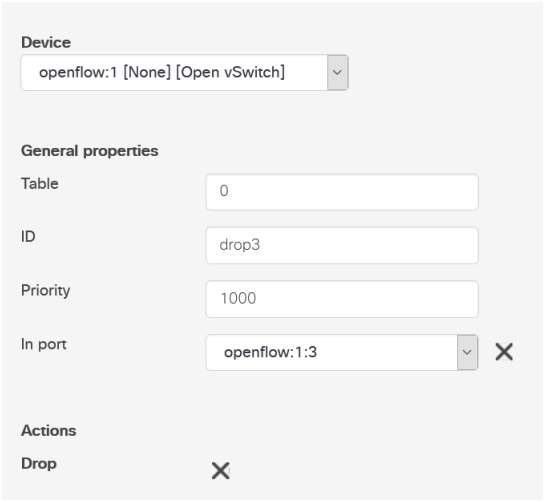


Figura 4.31: Regla de flujo insertada

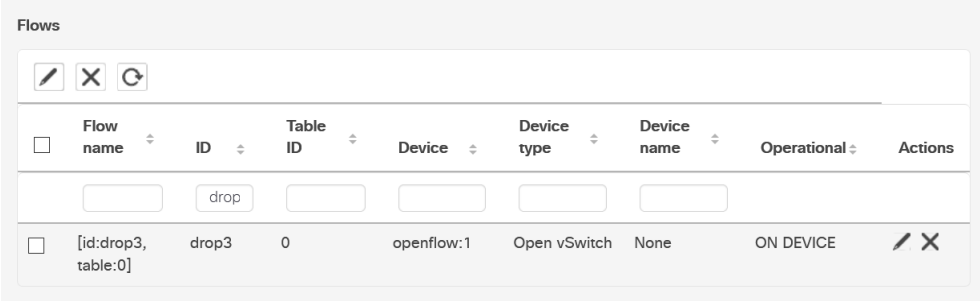


Figura 4.32: *Ping* desde el *host* 1 al 9 tras insertar la regla de flujo

```
mininet> h1 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
^C
--- 10.0.0.9 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6047ms
```

Figura 4.33: *Ping* desde el *host* 1 al 5 tras insertar la regla de flujo

```
mininet> h1 ping h5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=0.345 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=0.182 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=0.126 ms
64 bytes from 10.0.0.5: icmp_seq=4 ttl=64 time=0.130 ms
^C
--- 10.0.0.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
rtt min/avg/max/mdev = 0.126/0.195/0.345/0.090 ms
mininet>
```

Figura 4.34: *Ping* desde el *host* 9 al 7 tras insertar la regla de flujo

```
mininet> h9 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=0.492 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=0.030 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.031 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.037 ms
^C
--- 10.0.0.7 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.030/0.147/0.492/0.199 ms
```

Figura 4.35: Estadísticas de paquetes que cumplen la regla de flujo

Device:

☑ openflow:1 [None] [Open vSwitch] ☐ openflow:3 [None] [Open vSwitch] ☐ openflow:2 [None] [Open vSwitch] ☐ openflow:4 [None] [Open vSwitch]

Device	DeviceType	DeviceName	TableId	flowid	Packetcount
openflow:1	openflow:1	None	0	#UFSTABLE*0 -4	1625
openflow:1	openflow:1	None	0	#UFSTABLE*0 -2	219
openflow:1	openflow:1	None	0	drop3	106
openflow:1	openflow:1	None	0	#UFSTABLE*0 -1	103
openflow:1	openflow:1	None	0	#UFSTABLE*0 -3	103
openflow:1	openflow:1	None	0	#UFSTABLE*0 -5	0

Figura 4.36: *Ping* desde el *host* 1 al 9 tras eliminar la regla de flujo

```
mininet> h1 ping h9
PING 10.0.0.9 (10.0.0.9) 56(84) bytes of data.
64 bytes from 10.0.0.9: icmp_seq=1 ttl=64 time=0.346 ms
64 bytes from 10.0.0.9: icmp_seq=2 ttl=64 time=0.251 ms
64 bytes from 10.0.0.9: icmp_seq=3 ttl=64 time=0.129 ms
64 bytes from 10.0.0.9: icmp_seq=4 ttl=64 time=0.196 ms
64 bytes from 10.0.0.9: icmp_seq=5 ttl=64 time=0.199 ms
^C
--- 10.0.0.9 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999ms
rtt min/avg/max/mdev = 0.129/0.224/0.346/0.072 ms
```

Por otro lado, si se entra en el apartado de estadísticas de la aplicación, se ve perfectamente que la regla drop3 está siendo usada, debido a que, como se visualiza en la Figura 4.35, el número de paquetes que ha contabilizado es mayor que 0.

Finalmente, al eliminar la regla de la tabla de flujo, la conexión entre los *hosts* 1 y 9 inmediatamente vuelve a ser posible, tal y como se puede comprobar en la Figura 4.36.

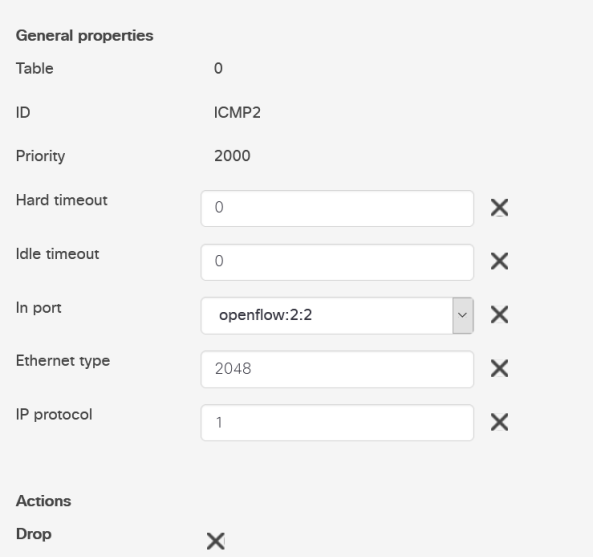
Ejemplo 2

En este ejemplo se muestra como en SDN también se pueden filtrar paquetes según su contenido. Para ello se ha dispuesto de la creación de un servidor web simple en uno de los *hosts* y otro *host* que hiciera las veces de cliente. El comando a ejecutar en Mininet para crear el servidor web es el siguiente:

```
mininet> h1 python -m SimpleHTTPServer 80 &
```

La regla de flujo a añadir en el *switch* 2 (el que conecta los *hosts* 1, 2 y 3) se observa en la Figura 4.37. Analizando esta, se describen las siguientes características;

Figura 4.37: Regla de flujo para el ejemplo 2



- La regla esta ubicada en la tabla 0
- El id es ICMP2
- Tiene prioridad de 2000
- Los *idle* y *hard timeouts* son 0, es decir que la regla no expira tras un tiempo determinado.
- Se aplica a todos los paquetes provenientes del puerto 2 (el que conecta al *host* 1)
- El tipo de paquetes al que se aplica es IPv4 (0x0800 hex., 2048 dec.)
- Dentro de los paquetes IPv4 al protocolo que se aplica es al ICMP (número de protocolo 1 en decimal)
- La acción a realizar para los paquetes que cumplan la regla, es descartarlo.

Tras añadirla, se mostrará, como en el ejemplo 1, en la tabla de las reglas con la característica de *ON DEVICE*, por tanto, desde ese instante ya se estará aplicando. Muestra de ello son las Figuras 4.38 y 4.39, en las que se intenta hacer un *ping* y una petición *wget*. En la primera, como era de esperar, el *ping* no da resultado satisfactorio. En la segunda, al no establecerse la regla para el protocolo TCP, se observa que si se recibe información de la página web alojada en el servidor. El comando para realizar esta última es el siguiente:

Figura 4.38: *Ping* desde el *host* 1 al 2 tras insertar la regla de flujo

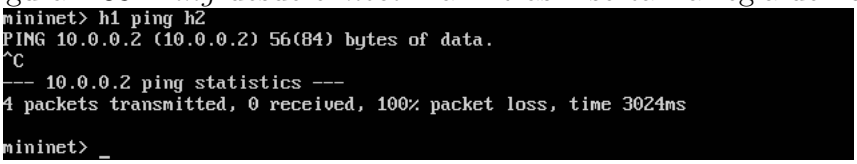
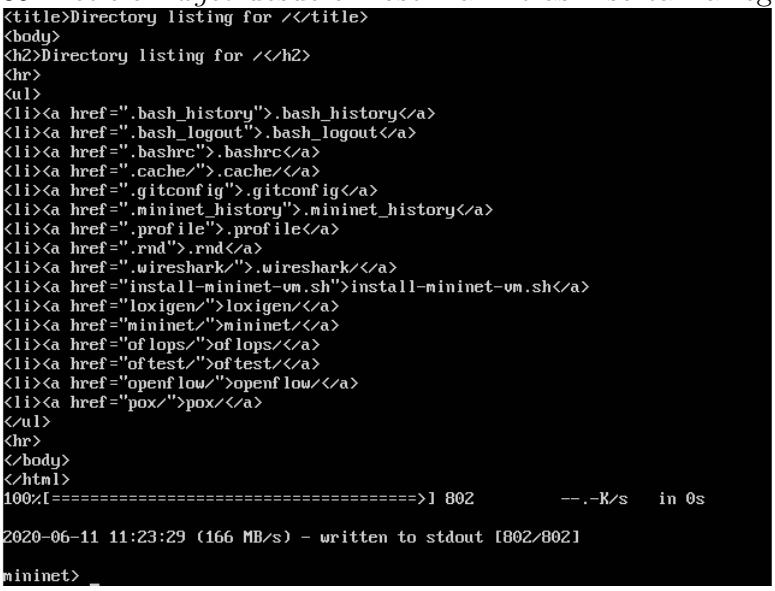


Figura 4.39: Petición *wget* desde el host 2 al 1 tras insertar la regla de flujo




```
mininet> h2 wget -O - h1
```

Tras eliminar la regla, todo vuelve a su origen y ambas máquinas pueden realizarse *pings* sin problema.

4.4. Controlador RYU

4.4.1. Introducción

Una vez instalado el controlador según los pasos indicados en el Anexo C.2, se procede a describir la topología de red escogida.

En caso se ha decidido usar una de red en árbol, con 3 *switches* y 2 terminales en los *switches* de los extremos. Esta proporciona la suficiente información para abordar simulaciones que permitan comprobar las diversas funcionalidades de SDN y OpenFlow. Para su realización se divide el contenido en dos puntos:

Figura 4.40: Listado de archivos de RYU

```
jose@jose-VirtualBox:/usr/local/lib/python2.7/dist-packages/ryu/app$ ls
ompstation.py           simple_switch_13.py
ompstation.pyc          simple_switch_13.pyc
cbench.py               simple_switch_14.py
cbench.pyc              simple_switch_14.pyc
conf_switch_key.py      simple_switch_15.py
conf_switch_key.pyc     simple_switch_15.pyc
example_switch_13.py    simple_switch_igmp_13.py
example_switch_13.pyc   simple_switch_igmp_13.pyc
gui_topology            simple_switch_igmp.py
__init__.py             simple_switch_igmp.pyc
__init__.pyc            simple_switch_lacp_13.py
ofctl                   simple_switch_lacp_13.pyc
ofctl_rest.py           simple_switch_lacp.py
ofctl_rest.pyc          simple_switch_lacp.pyc
rest_conf_switch.py     simple_switch.py
rest_conf_switch.pyc    simple_switch.pyc
rest_firewall.py        simple_switch_rest_13.py
rest_firewall.pyc       simple_switch_rest_13.pyc
rest_qos.py             simple_switch_snort.py
rest_qos.pyc            simple_switch_snort.pyc
rest_router.py          simple_switch_stp_13.py
rest_router.pyc         simple_switch_stp_13.pyc
rest_topology.py        simple_switch_stp.py
rest_topology.pyc       simple_switch_stp.pyc
rest_vtep.py            simple_switch_websocket_13.py
rest_vtep.pyc           simple_switch_websocket_13.pyc
simple_monitor_13.py    wsgi.py
simple_monitor_13.pyc   wsgi.pyc
```

- **Las principales funciones**. Se explican en detalle como se implementan las funcionalidades OpenFlow en los *switches* y cuales son los ejemplos de los que se disponen para la simulación.

- La **creación y análisis de la red**. Se marcan las indicaciones para el montaje de la red en Mininet y se analiza el tráfico OpenFlow generado mediante la herramienta Wireshark.

4.4.2. Principales funciones

Dentro del directorio de RYU, como se observa en la Figura 4.40, existen diversos archivos y de entre estos destaca uno redundante en el nombre, *simple switch*. Este es el responsable de establecer las funciones OpenFlow a los distintos *switches* de la red y según el sufijo se pueden destacar varias versiones de OF (desde la 1.0 hasta la 1.5), por ejemplo el archivo *simple switch* 15 tiene lo necesario para implementar un *switch* de versión OF 1.5. Previamente al desarrollo de *simple swtich*, cabe resaltar que en las simulaciones se usará el *simple switch* 13.

Simple Switch

El archivo *simple switch* contiene todo lo necesario para poder ejecutar el controlador RYU y que funcione la red perfectamente, es importante destacar que todo esto

se realiza en 119 líneas de código Python (incluyendo comentarios). A continuación, se procede al análisis del código mencionado y para una mejor comprensión se divide en varias partes.

En primer lugar, el **inicio y registro** consta de la versión de OpenFlow utilizada y de la inicialización de la tabla MAC-puerto.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

En segundo lugar, la sección de **control de eventos** se ejecuta cada vez que se asocia un *switch* al controlador y tiene como labor principal instalar una entrada de flujo al *switch* que haga que este reenvíe los paquetes al controlador.

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
```

En tercer lugar, la siguiente sección de código es la que se encarga de la **inserción de entradas de flujo** en los distintos *switches*. Esto se consigue mediante las llamadas al método OFPFlowMod.

```
def add_flow(self, datapath, priority, match, actions,
             buffer_id=None):
        ofproto = datapath.ofproto
```

```
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions
            (ofproto.OFPIT_APPLY_ACTIONS, actions)]
        if buffer_id:
            mod = parser.OFPFlowMod(datapath=datapath,
                                    buffer_id=buffer_id,
                                    priority=priority, match=match,
                                    instructions=inst)
        else:
            mod = parser.OFPFlowMod(datapath=datapath,
                                    priority=priority,
                                    match=match, instructions=inst)
        datapath.send_msg(mod)
```

En cuarto lugar, en esta sección se observa la función relacionada con el **manejo de paquetes**, concretamente el funcionamiento de PACKET-IN, es decir cuando el controlador recibe un mensaje del *switch*, lo analiza.

```
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]
```

```
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            # ignore lldp packet
            return
        dst = eth.dst
        src = eth.src
```

En quinto lugar, esta sección cubre el **aprendizaje de la dirección MAC y su puerto asociado** para así evitar inundar todos los puertos del *switch* cada vez que se envíe un paquete.

```
    dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid,
                         src, dst, in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port
```

En sexto lugar, en este trozo se observa la importancia del punto anterior ya que aquí se realiza **la asociación MAC-puerto y destino para los paquetes**. Esto hace que los paquetes sean enviados por según que puerto esté asociado a la dirección MAC solicitada.

```
    if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]
```

En séptimo lugar, en esta parte se cubre la **inserción de una entrada de flujo** con el objetivo de evitar que el *switch* envíe constantemente PACKET-INs al controlador y hacer la red más fluida.

```
# install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port,
                                    eth_dst=dst, eth_src=src)
            # verify if we have a valid buffer_id,
            # if yes avoid to send both
            # flow_mod & packet_out
            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
                self.add_flow(datapath, 1, match, actions,
                              msg.buffer_id)
                return
            else:
                self.add_flow(datapath, 1, match, actions)
```

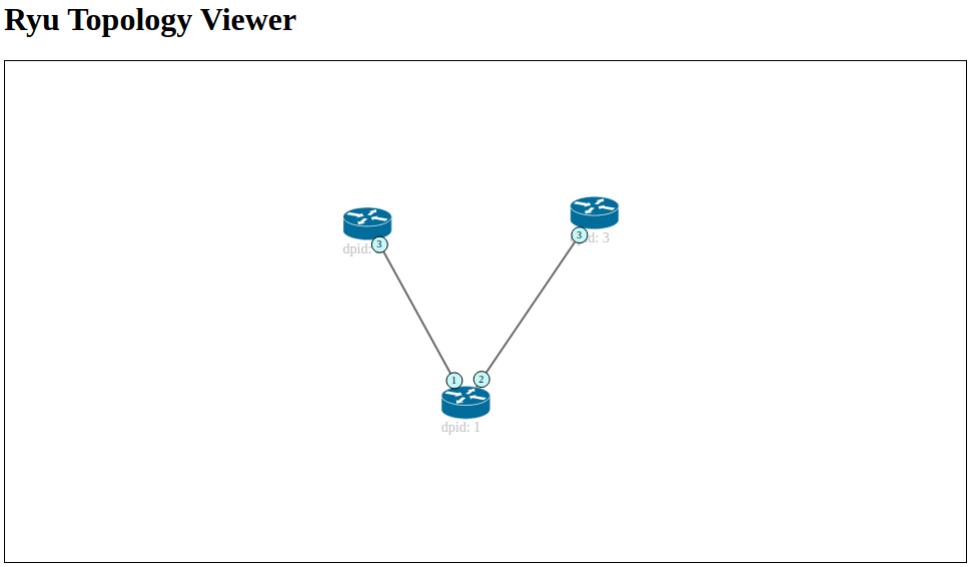
En último lugar, aquí se cubre lo relacionado con **los mensajes PACKET-OUT**, es decir los mensajes reenviados por el controlador al *switch* cuando recibe un PACKET-IN.

```
data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath,
                                  buffer_id=msg.buffer_id,
                                  in_port=in_port,
                                  actions=actions, data=data)
        datapath.send_msg(out)
```

Finalmente, para iniciar el controlador RYU no basta más que con acceder a su directorio principal (/usr/local/lib/python2.7/dist-packages/ryu/app) y ejecutar la siguiente línea en el terminal:

```
$ sudo ryu run simple_switch_13.py
```

Figura 4.41: Topología de red en RYU



4.4.3. Creación y análisis de la red

Creación

Para la creación de la red se dispone del controlador RYU ya iniciado (con la dirección IP 192.168.56.143) y de Mininet. El comando a ejecutar en Mininet para crear la red es el siguiente:

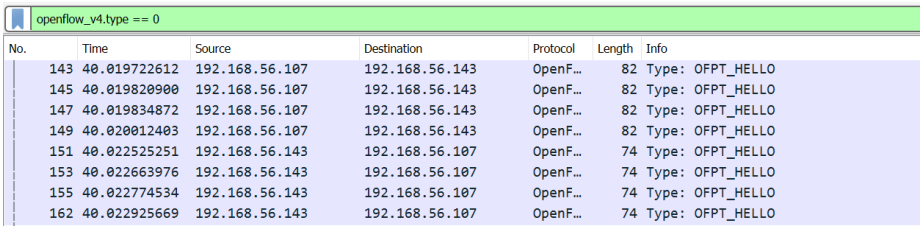
```
$ sudo mn --controller=remote,ip=192.168.56.143, --topo=tree,
depth=2, fanout=2 --switch=ovsk, protocols=OpenFlow13 --mac
```

Una vez creada la red, al contrario que en ODL, no se crea una GUI predefinida donde se pueda observar la topología de red creada. Para ello, a la hora de iniciar el controlador en lugar de escribir ryu run"se escribe lo siguiente (dentro del mismo directorio que ryu run):

```
$ sudo ryu run gui_topology/gui_topology.py simple_switch_13.py
-- observe-links
```

De este modo, al conectarse en el navegador al puerto 8080 de la dirección IP del controlador, se observa la Figura 4.41. En este caso, a diferencia de ODL, se muestran únicamente los *switches*.

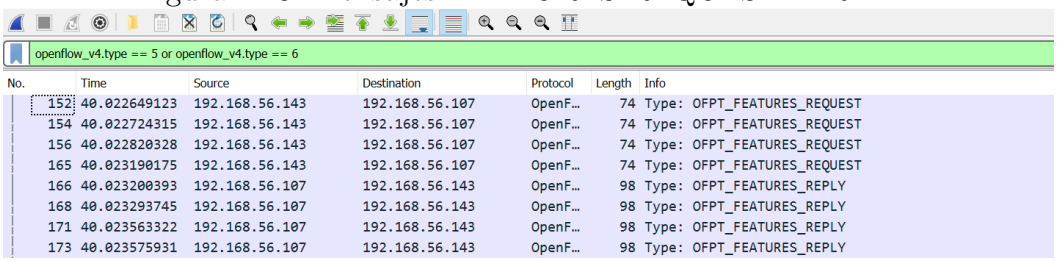
Figura 4.42: Mensajes HELLO



openflow_v4.type == 0

No.	Time	Source	Destination	Protocol	Length	Info
143	40.019722612	192.168.56.107	192.168.56.143	OpenF…	82	Type: OFPT_HELLO
145	40.019820900	192.168.56.107	192.168.56.143	OpenF…	82	Type: OFPT_HELLO
147	40.019834872	192.168.56.107	192.168.56.143	OpenF…	82	Type: OFPT_HELLO
149	40.020012403	192.168.56.107	192.168.56.143	OpenF…	82	Type: OFPT_HELLO
151	40.022525251	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_HELLO
153	40.022663976	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_HELLO
155	40.022774534	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_HELLO
162	40.022925669	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_HELLO

Figura 4.43: Mensajes FEATURES-REQUEST Y REPLY



openflow_v4.type == 5 or openflow_v4.type == 6

No.	Time	Source	Destination	Protocol	Length	Info
152	40.022649123	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_FEATURES_REQUEST
154	40.022724315	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_FEATURES_REQUEST
156	40.022820328	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_FEATURES_REQUEST
165	40.023190175	192.168.56.143	192.168.56.107	OpenF…	74	Type: OFPT_FEATURES_REQUEST
166	40.023200393	192.168.56.107	192.168.56.143	OpenF…	98	Type: OFPT_FEATURES_REPLY
168	40.023293745	192.168.56.107	192.168.56.143	OpenF…	98	Type: OFPT_FEATURES_REPLY
171	40.023563322	192.168.56.107	192.168.56.143	OpenF…	98	Type: OFPT_FEATURES_REPLY
173	40.023575931	192.168.56.107	192.168.56.143	OpenF…	98	Type: OFPT_FEATURES_REPLY

Análisis

El análisis se centrará en el estudio de los mensajes OpenFlows enviados entre el controlador y los distintos *switches* para un *ping* entre dos *hosts*. Los mensajes son:

- HELLO
- FEATURES-REPLY y FEATURES-REQUEST
- PACKET-IN y PACKET-OUT
- FLOW-MOD
- ECHO-REPLY y ECHO-REQUEST

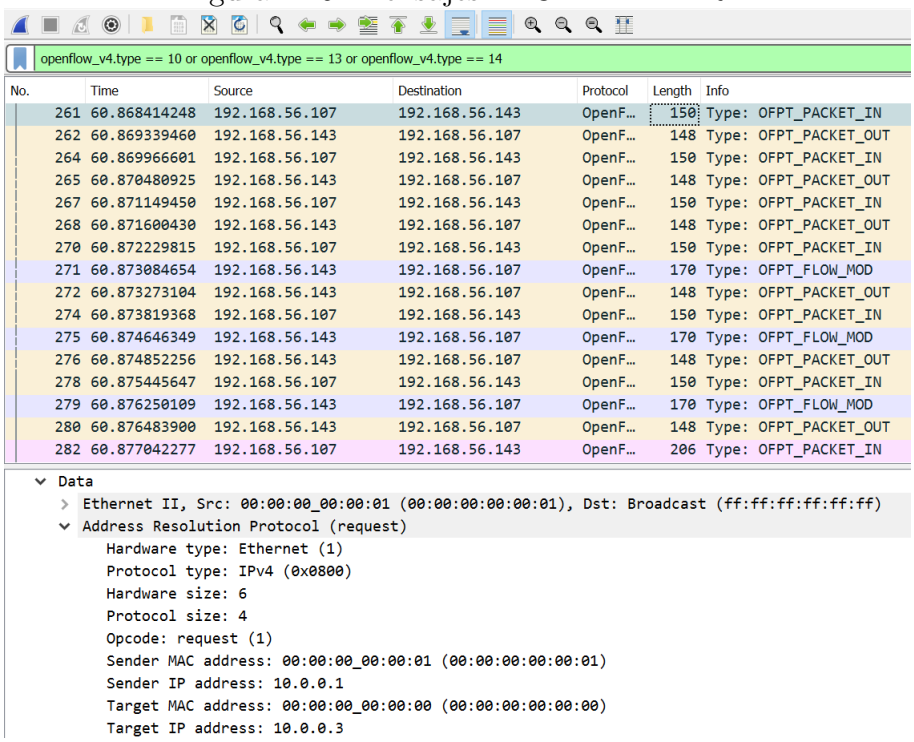
En primer lugar, el mensaje HELLO se envía en el establecimiento de la conexión, y al ser simétrico se envía tanto por el *switch* como por el controlador. En este caso, al haber 4 *switches* habrá 8 mensajes HELLO. Esto se observa en la Figura 4.42.

En segundo lugar, los mensajes FEATURES REQUEST y REPLY se envían por parte del controlador al *switch* (las peticiones o *request*) y del *switch* al controlador (las respuestas o *reply*). Esto se realiza tras el establecimiento de la conexión y tiene como objetivo hacer que el controlador conozca las características del *switch* (p.ej: versión de OF utilizada). Estos mensajes se observan en la Figura 4.43.

Figura 4.44: *Ping* de h1 a h3

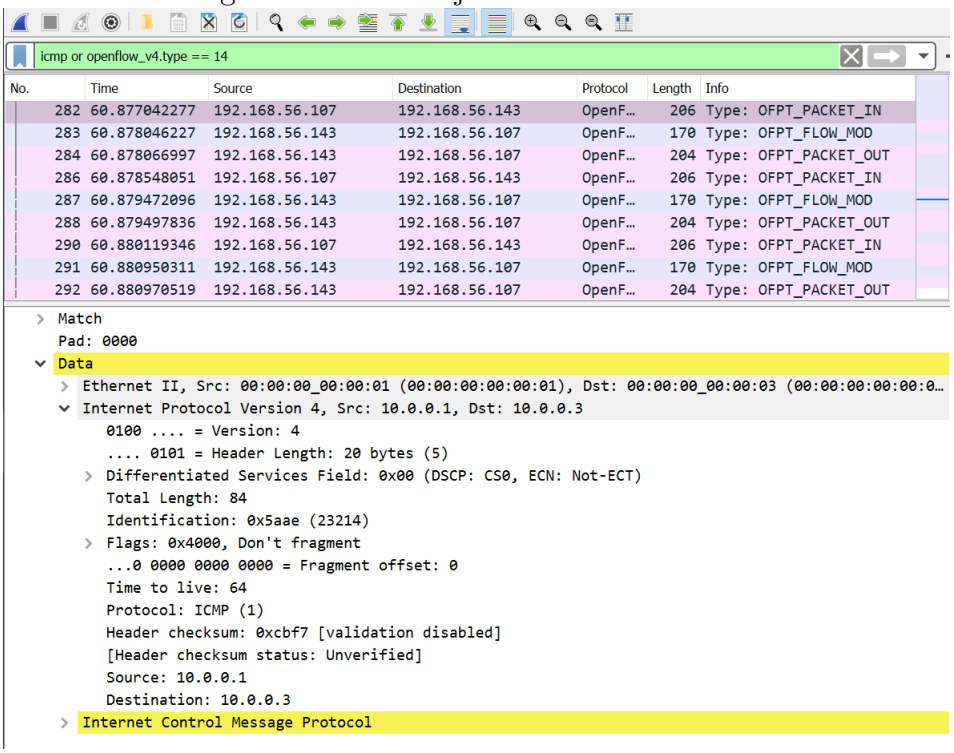
```
mininet> h1 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=13.3 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.313 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.047 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.039 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=0.041 ms
^C
--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.039/2.762/13.371/5.305 ms
```

Figura 4.45: Mensajes PACKET-IN ARP



En tercer lugar, se realiza un *ping* desde el *host* 1 al *host* 3. Esto conlleva la consecuente generación de mensajes ARP para obtener la dirección MAC y mensajes ICMP para realizar el *ping*. Estos se llevan acabo siendo encapsulados en mensajes OpenFlow del tipo PACKET-IN, PACKET-OUT. Como ya se estudió con anterioridad, los mensajes PACKET-IN se envían por parte del *switch* al controlador y los PACKET-OUT del controlador al *switch*. En este caso, se observa la realización del *ping* en la Figura 4.44, y como esta lleva asociada los mensajes PACKET-IN y PACKET-OUT de ARP (en naranja, Figura 4.45) y los de ICMP (en rosa, Figura 4.46).

Figura 4.46: Mensajes PACKET-IN ICMP



En cuarto lugar, entre los mensajes anteriores se encuentra siempre uno llamado FLOW-MOD. Este tiene la función de insertar la entrada de flujo necesaria en el *switch* correspondiente. En este caso, como se observa en la Figura 4.47, la acción a realizar es enviar los paquetes por el puerto 3, lo que hace que el *ping* se realice de manera correcta.

En último lugar, los mensajes ECHO REQUEST y REPLY se envían para verificar que la conexión entre *switch* y controlador sigue activa, al igual que HELLO es un mensaje de tipo simétrico. En la Figura 4.48 se observan estos mensajes.

Por otro lado, en cuanto a las tablas de flujos presentes en los *switches*, a modo de ejemplo, en la Figura 4.49 se observan las entradas para el *switch* 1. En esta se ve claramente como la entrada con mayor prioridad tiene como OUTPUT el controlador, por este motivo todos los mensajes que se envían desde un *host* a otro siempre pasan por el controlador.

Figura 4.47: Mensajes FLOW-MOD

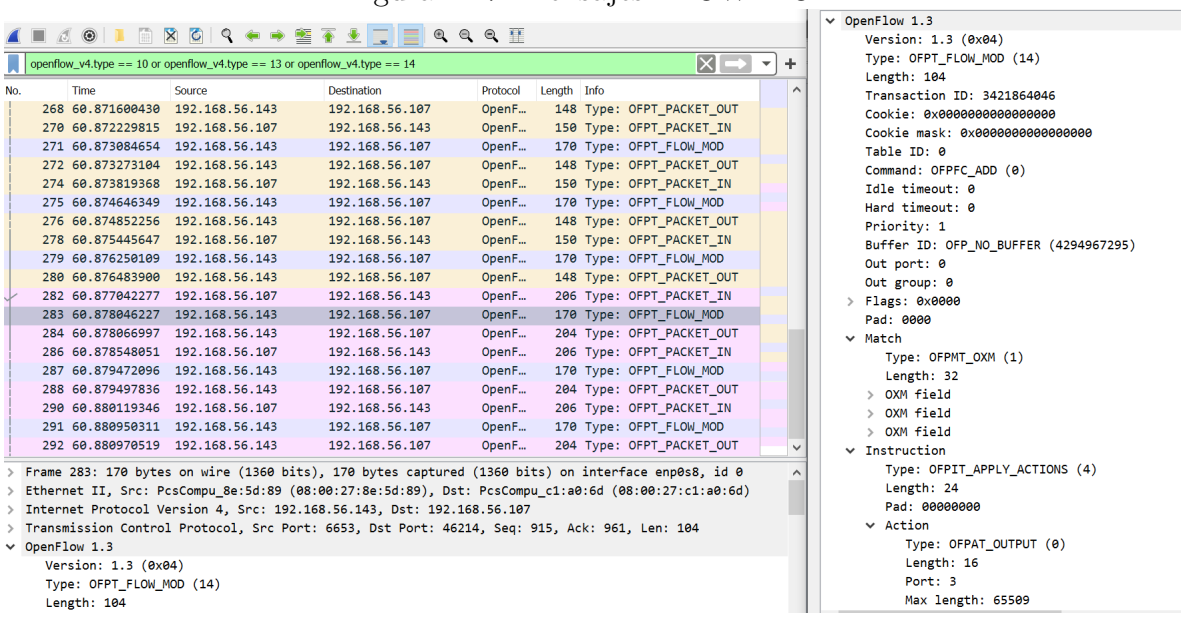


Figura 4.48: Mensajes ECHO-REQUEST Y REPLAY

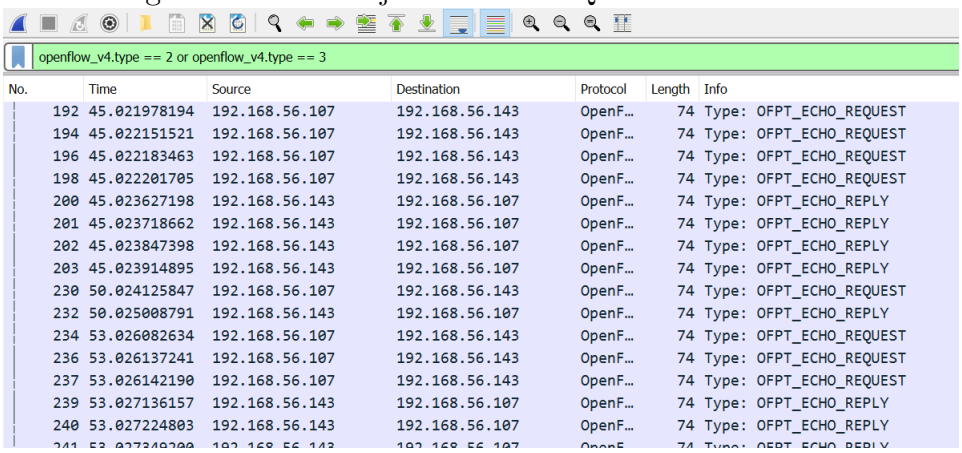


Figura 4.49: Entradas de flujo OF para *switch* 1

```
mininet> sh ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=150.732s, table=0, n_packets=6, n_bytes=532, priority=1,in
_port=2,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=150.727s, table=0, n_packets=5, n_bytes=434, priority=1,in
_port=1,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=output:2
 cookie=0x0, duration=171.571s, table=0, n_packets=4, n_bytes=224, priority=0 ac
tions=CONTROLLER:65535
```

Capítulo 5

Realización de pruebas en entorno simulado. GNS3

5.1. Introducción

GNS3 es una herramienta *software* gratuita que permite la creación, diseño y prueba de distintos escenarios de red. Disponible para todos los sistemas operativos, se apoya en el uso de la tecnología Docker para simular servicios y aplicaciones de dispositivos presentes en la red.

Este capítulo tiene como objetivo el aprendizaje de SDN a través la tecnología Docker y el uso de GNS3. Para lograrlo, a lo largo del capítulo se desarrollan los siguientes aspectos:

- **Explicación de la tecnología Docker.** Cubriendo varios aspectos, tales como: sus fundamentos, una comparación con las máquinas virtuales y cómo se integra dentro de GNS3.
- **Creación de la red**. Se explica como se configuran los distintos dispositivos y como se crea la topología de red escogida.
- **Simulación con controlador ODL**. Se analizan las funciones características de SDN mediante el uso del controlador ODL y de la aplicación OFM para editar reglas de flujo.

Figura 5.1: Evolución del interés por Docker según *Google Trends*

- **Simulación con controlador RYU**. Se analizan las funciones características de SDN mediante el uso del controlador RYU, además esto se complementa con el uso de Wireshark para analizar el tráfico de paquetes OpenFlow en la red.

5.2. Tecnología Docker

5.2.1. Introducción

Docker comenzó su vida a principios de 2013 como un proyecto *open source* en dotCloud, empresa dedicada a ofrecer plataformas como servicio en la nube. Transcurrido un año, se independizó, se formó como empresa (pasándose a llamar Docker Inc.) y se integró dentro de la *Linux Foundation*. Estos hechos sentaron las bases para el desarrollo de la tecnología Docker y su ecosistema. A modo ilustrativo, si se observa la Figura 5.1 (obtenida desde *Google Trends*), se ve perfectamente el crecimiento por el interés en Docker en los últimos años.

En los siguientes apartados se tratan tres aspectos fundamentales para entender la tecnología Docker y aplicarla a GNS3. Estos son:

- **Los fundamentos**. Se define qué es Docker y cómo funciona, destacando los elementos principales de su arquitectura.
- **Una comparación con las máquinas virtuales**. Se realiza una breve comparativa de Docker respecto de estas y cuáles son las ventajas de usarlo.

- **Integración junto a GNS3**. Se relaciona el uso de Docker dentro de GNS3 y como es posible implementar esta tecnología dentro del programa cuando tenemos un S.O. Windows o Mac.

5.2.2. Fundamentos

¿Qué es Docker?

A la pregunta de ¿qué es Docker? se responde accediendo a su web [36], debido a que en esta se encuentra la mejor definición. Según ellos mismos, Docker es una plataforma abierta que permite desarrollar, enviar y ejecutar aplicaciones. Al mismo tiempo, también es visto como un programa de línea de comandos con un demonio ejecutándose en segundo plano. Este programa tiene un conjunto de herramientas que se utilizan para resolver problemas comunes de *software* y simplificar la experiencia del usuario al instalar, ejecutar, publicar y eliminar aplicaciones.

Funcionamiento

Docker permite separar las aplicaciones de su infraestructura, entregando así el *software* de una manera más rápida. Para esta tarea utiliza los llamados contenedores. Estos son una especie de *buckets* o cubos software que incluyen todas las dependencias para ejecutar la aplicación de forma independiente. Se ejecutan directamente sobre el *kernel* de la máquina, es decir, que la aplicación y la máquina anfitriona han de tener el mismo *kernel* para que todo funcione. La mayoría, si no todas las aplicaciones presentes en Docker usan el kernel de Linux, por esta razón Docker se ejecuta únicamente sobre esta arquitectura.

Por otro lado, existen las imágenes. Estas son la referencia que toma Docker para crear los contenedores, es decir que los contenedores son instancias de las imágenes. Al igual que sucede en GitHub, Docker también tiene un repositorio. En este se almacenan las imágenes que hacen referencia a las aplicaciones o servicios que se quieran utilizar.

5.2.3. Comparación con las máquinas virtuales

Para entender por qué Docker ha supuesto una revolución a la hora de ejecutar aplicaciones de manera virtualizada, primero se han de conocer los tipos de virtualizaciones

existentes, son dos: [37]

- **La virtualización ligera**. Es la de tipo 1 o *non-hosted*. En esta, el sistema operativo de la máquina virtual se ejecuta directamente sobre los componentes hardware gracias a un elemento llamado hipervisor. Hay que tener en cuenta que el kernel de la máquina virtual y de la máquina anfitriona tiene que ser el mismo.
- **La virtualización pesada**. Es la de tipo 2 o *hosted*. En esta, las máquinas virtuales actúan como una capa intermedia entre el hardware y el sistema operativo invitado, por tanto no se ejecuta directamente sobre el hardware. Las máquinas son engañadas ya que creen que poseen de forma exclusiva los recursos hardware. Al mismo tiempo, ejecutan una instancia de sistema operativo sobre el que corren servicios o aplicaciones. Ejemplos de esta virtualización serían las de VirtualBox o VMWare Worksation.

Ya sea para el tipo 1 o tipo 2, es necesaria la instalación de un sistema operativo dentro de la máquina virtual para ejecutar cualquier servicio, lo que genera un amplio gasto de recursos y un mayor tiempo de carga.

Como se observa en la Figura 5.2, el proceso de carga y el gasto de recursos es ampliamente menor en Docker. Esto es gracias a la utilización de los contenedores para el uso de aplicaciones. Al compartir el mismo *kernel*, el contenedor lo aprovecha y únicamente añade las librerías y dependencias necesarias para ejecutar el servicio, por lo tanto, no es necesario tener que instalar todo un sistema operativo para realizar el servicio.

El problema de Docker reside cuando se utiliza una máquina anfitriona que no tiene el *kernel* de Linux (Windows o Mac). En este caso, la solución que se aporta consiste en generar una máquina virtual en segundo plano (ver Figura 5.3) que se encargue de ejecutar los contenedores. Esto será lo que se utilice posteriormente en GNS3.

5.2.4. Integración junto a GNS3

GNS3 utiliza Docker a través de sus llamadas *appliances*. En su web [38] se puede encontrar un repositorio con distintas imágenes que al importarlas en la aplicación (ver

Figura 5.2: Comparación Docker vs Virtualización Tipo 2

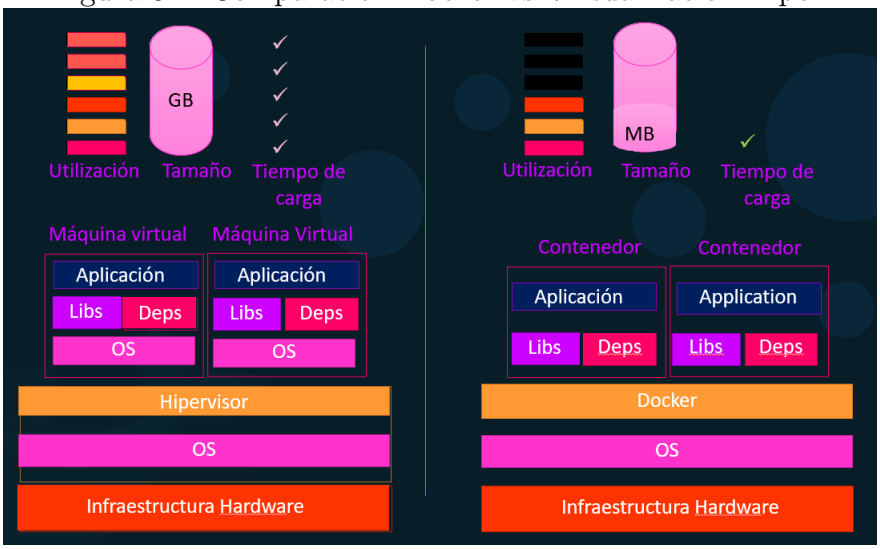
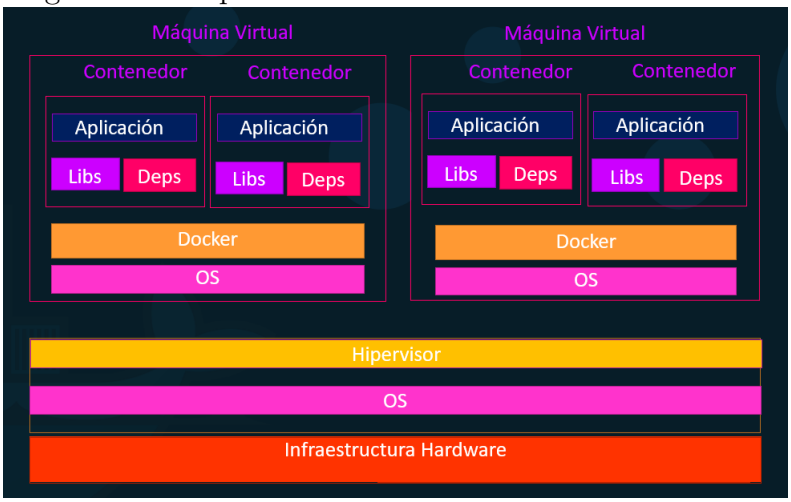


Figura 5.3: Arquitectura Docker en un sistema no Linux



Anexo B.2) crea un dispositivo. A la hora de las simulaciones, al insertar este en la red, generará una instancia de esa imagen, es decir, un contenedor Docker.

Como ya se ha comentado, Docker solo se ejecuta directamente sobre sistemas operativos con *kernel* de Linux. Por este motivo, cuando se quiere utilizar Docker en GNS3 en un Windows o Mac, el propio software obliga a la instalación de una máquina virtual con *kernel* de Linux (GNS3 proporciona su propia MV) para así poder ejecutar esos servicios. Esta máquina virtual puede ejecutarse tanto en VirtualBox como en VMWare Workstation o Player.

5.3. Creación del escenario de red

En este apartado se analizan los distintos componentes de la red, puesto que para ambas simulaciones el escenario utilizado es el mismo y la única diferencia reside en el controlador.

Previo paso a la creación de la red hay que importar las *appliances* o dispositivos correspondientes para la realización de la simulación. En este caso son las relacionadas con los *hosts* y *switches* OpenFlow. Los pasos de instalación se encuentran en el Anexo B.2.

Tras instalar estos elementos es hora de crear la red, para ello se usarán los siguientes dispositivos:

- **Cuatro *hosts***. Tienen la característica principal de ser dispositivos Ubuntu alojados en un contenedor virtual proporcionado por Docker.
- **Tres *switches* OpenFlow**. Son *switches* del tipo OvSwitch cuyas funciones también se encuentran alojadas en un contenedor Docker y que tienen como diferenciación principal ser *switches* de carácter híbrido. Esta función se analizará en el siguiente apartado (ver 5.4.2).
- **Un *switch* tradicional**. Sirve para realizar la conexión de los distintos *switches* OF con la nube.
- **Una nube**. GNS3 tiene a la nube como un dispositivo que se encuentran fuera de la red del programa. En este caso la nube será nuestro controlador, que se encuentra alojado en una máquina virtual Ubuntu corriendo en VirtualBox.

Figura 5.4: Configuración de la dirección IP del *host* 1 en GNS3

```
#
# This is a sample network config uncomment lines to configure the network
#


# Static config for eth0
auto eth0
iface eth0 inet static
                        address 10.0.0.1
                        netmask 255.255.255.0
                        gateway 10.0.0.1
#                       up echo nameserver 192.168.0.1 > /etc/resolv.conf

# DHCP config for eth0
# auto eth0
# iface eth0 inet dhcp
```

Descritos los elementos de red, es turno configurar las direcciones IP para las interfaces de los *switches* OF y los *hosts*. Para ello, se hace clic con el botón derecho sobre el dispositivo que se quiera configurar y se accede a la opción de *Edit Config*. Una vez ahí, se muestra una ventana con texto plano como la de las Figuras 5.4 y 5.5, que corresponden al *host* 1 y *switch* 1, respectivamente. Cabe destacar que la única interfaz configurada para los *switches* es la eth0. Esto se debe a que es la encargada de la administración y gestión local del *switch*, por lo tanto, es la que se conecta al controlador.

Para finalizar, en la Tabla 5.1 se muestran las distintas direcciones IP asociadas a cada dispositivo e interfaz correspondiente y en la Figura 5.6 se observa la red en su totalidad.

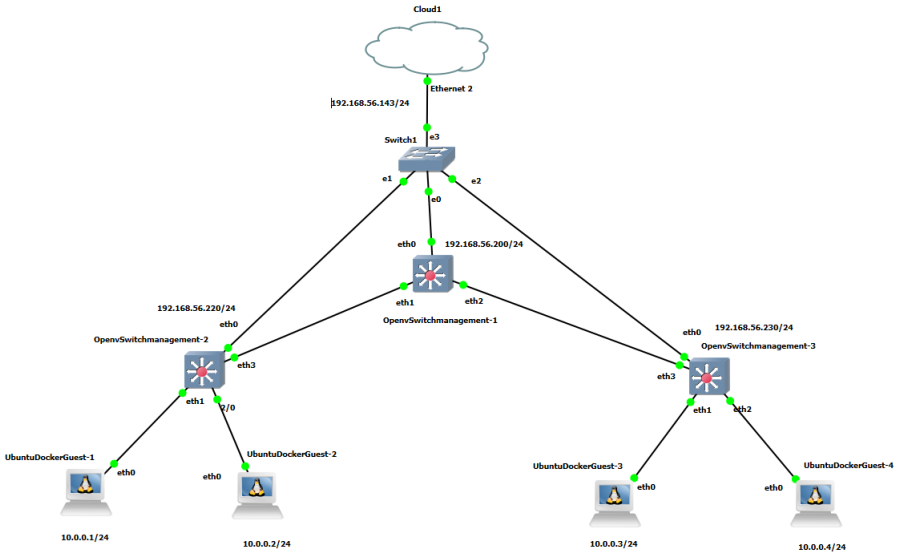
Dispositivo (Interfaz)	**Dirección IP**
Cloud 1 (Ethernet 2)	192.168.56.143/24
Ubuntu Docker Guest 1 (eth0)	10.0.0.1/24
Ubuntu Docker Guest 2 (eth0)	10.0.0.2/24
Ubuntu Docker Guest 3 (eth0)	10.0.0.3/24
Ubuntu Docker Guest 4 (eth0)	10.0.0.4/24
OpenvSwitch 1 (eth0)	192.168.56.200/24
OpenvSwitch 2 (eth0)	192.168.56.220/24
OpenvSwitch 3 (eth0)	192.168.56.230/24

Tabla 5.1: Direcciones IP de los dispositivos GNS3

Figura 5.5: Configuración de la dirección IP del *switch* 1 en GNS3

```
#
# This is a sample network config uncomment lines to configure the network
#


# Static config for eth0
auto eth0
iface eth0 inet static
                        address 192.168.56.200
                        netmask 255.255.255.0
                        gateway 192.168.56.1
#                       up echo nameserver 192.168.0.1 > /etc/resolv.conf

# DHCP config for eth0
# auto eth0
# iface eth0 inet dhcp
# Static config for eth1
#auto eth1
#iface eth1 inet static
#                       address 192.168.56.201
#                       netmask 255.255.255.0
#                       gateway 192.168.56.1
#                       up echo nameserver 192.168.1.1 > /etc/resolv.conf

# DHCP config for eth1
# auto eth1
# iface eth1 inet dhcp
# Static config for eth2
#auto eth2
#iface eth2 inet static
#                       address 192.168.56.202
#                       netmask 255.255.255.0
#                       gateway 192.168.56.1
#                       up echo nameserver 192.168.2.1 > /etc/resolv.conf

# DHCP config for eth2
# auto eth2
# iface eth2 inet dhcp
# Static config for eth3
#auto eth3
#iface eth3 inet static
#                       address 192.168.3.2
#                       netmask 255.255.255.0
#                       gateway 192.168.3.1
#                       up echo nameserver 192.168.3.1 > /etc/resolv.conf

# DHCP config for eth3
# auto eth3
```

Figura 5.6: Escenario de red GNS3



5.4. Controlador ODL

5.4.1. Introducción

En la realización de las simulaciones se ha utilizado el mismo controlador que en el Capítulo 4, pero a diferencia de este, como ya se vio en el apartado anterior (ver 5.3), la red simulada consta de cuatro *hosts*, tres *switches* OpenFlow, un *switch* tradicional y una nube.

Por otro lado, su desarrollo se ha dividido en dos partes para una mejor comprensión. Estas son:

- El **análisis de la red**, dónde se realizan las comprobaciones de las características de los *switches* OF y se establece el controlador para los mismos. Para este caso, el tráfico OpenFlow no se analiza puesto que da el mismo resultado que en el apartado 4.3.3.

- El uso de la **App de Cisco OFM**. Permite establecer distintas reglas de flujo para comprobar la funcionalidad de SDN.

Nota: Es importante aclarar que en este caso no se trata el apartado de las funciones puesto que son las mismas que se utilizaron en la sección 4.3.2.

Figura 5.7: Tabla de flujos OvSwitch previa asociación al controlador

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=249.861s, table=0, n_packets=54, n_bytes=4260, idle_age=0,
 priority=0 actions=NORMAL
/ #
```

5.4.2. Análisis de la red

Antes de asociar los distintos *switches* al controlador se comprueba su característica híbrida. Como ya se vio en el apartado 3.2.1 esta hace posible que en caso de no haber un controlador asociado al *switch*, éste se comporte de manera tradicional. Para comprobarlo, se analizan las tablas de flujo y se observa que únicamente existe una entrada cuya acción es reenviar los paquetes por el puerto NORMAL (ver Figura 5.7). A modo de ejemplo, si se realiza un *ping* desde el *host* 1 hasta el *host* 3, se observa un resultado satisfactorio. Este hecho se puede comprobar en la Figura 5.8.

Figura 5.8: *Ping host* 1 a *host* 3 previa asociación al controlador

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.3
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=2.53 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=2.48 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.77 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=2.30 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 2.307/2.525/2.779/0.179 ms
```

Una vez hecho este análisis y partiendo de la premisa de que el controlador ya está iniciado, es hora de asociar los distintos *switches*. Para ello, primero se hace clic con el botón derecho sobre el dispositivo y se selecciona la opción de *Console*. Esto produce que se muestre una ventana de consola perteneciente al *switch*. Una vez ahí, se comprueba que existe la conexión realizando un *ping* al controlador (ver Figura 5.9). En caso de que sea satisfactorio, se escribe la siguiente orden para asociarlo.

```
$ ovs-vsctl set-controller br0 tcp:192.168.56.143:6633
```

Nota: Para evitar bucles se ha de habilitar el *Spanning Tree Protocol* y para ello se ejecuta la siguiente orden:

```
$ ovs-vsctl set bridge br0 stp_enable=true
```

Figura 5.9: *Ping* desde el *switch* al controlador

```
/ # ping 192.168.56.143
PING 192.168.56.143 (192.168.56.143): 56 data bytes
64 bytes from 192.168.56.143: seq=0 ttl=64 time=2.861 ms
64 bytes from 192.168.56.143: seq=1 ttl=64 time=1.630 ms
64 bytes from 192.168.56.143: seq=2 ttl=64 time=1.757 ms
64 bytes from 192.168.56.143: seq=3 ttl=64 time=1.846 ms
^C
--- 192.168.56.143 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 1.630/2.023/2.861 ms
/ #
```

Figura 5.10: Lista de interfaces *switch* 1

```
d6066151-62c1-4c52-a38b-bea923f837e0
    Bridge "br0"
        Controller "tcp:192.168.56.143:6633"
            is_connected: true
        Port "eth8"
            Interface "eth8"
        Port "eth6"
            Interface "eth6"
        Port "eth7"
            Interface "eth7"
        Port "eth2"
            Interface "eth2"
        Port "eth5"
            Interface "eth5"
        Port "eth1"
            Interface "eth1"
        Port "eth12"
            Interface "eth12"
        Port "eth15"
            Interface "eth15"
        Port "eth4"
            Interface "eth4"
        Port "eth3"
            Interface "eth3"
```

Para comprobar si la asociación ha sido correcta, se revisan las entradas de flujo y la lista de interfaces. Si todo está correcto, en la lista de interfaces se muestra una línea indicando que el controlador está conectado y en la tabla de flujo se observan que las entradas son las asociadas al controlador. Estos resultados se muestran en las Figuras 5.10 y 5.11 respectivamente. Los comandos a ejecutar en la consola son los siguientes:

```
$ ovs-vsctl show # Muestra la lista de interfaces
$ ovs-ofctl dump-flows br0 # Muestra las entradas de flujo
```

Al mismo tiempo, al abrir el navegador y acceder a la interfaz gráfica de ODL, se muestran los distintos dispositivos presentes en la red y que conoce el controlador. Esto se observa en la Figura 5.12.

Figura 5.11: Tabla de flujos tras asociar el controlador

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x2b00000000000001, duration=54.010s, table=0, n_packets=0, n_bytes=0, i
dle_age=54, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2b00000000000000, duration=50.203s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=1 actions=output:7,output:6,output:9,output:8,out
put:3,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000004, duration=50.203s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=8 actions=output:1,output:7,output:6,output:9,out
put:3,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000003, duration=50.203s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=9 actions=output:1,output:7,output:6,output:8,out
put:3,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000001, duration=50.194s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=7 actions=output:1,output:6,output:9,output:8,out
put:3,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000005, duration=50.194s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=3 actions=output:1,output:7,output:6,output:9,out
put:8,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000006, duration=50.194s, table=0, n_packets=1, n_bytes=70,
idle_age=13, priority=2,in_port=2 actions=output:1,output:7,output:6,output:9,ou
tput:8,output:3,output:5,output:4,output:10,output:12,output:11,output:14,output
:13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000007, duration=50.194s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=5 actions=output:1,output:7,output:6,output:9,out
put:8,output:3,output:2,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000008, duration=50.193s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=4 actions=output:1,output:7,output:6,output:9,out
put:8,output:3,output:2,output:5,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000009, duration=50.193s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=10 actions=output:1,output:7,output:6,output:9,ou
tput:8,output:3,output:2,output:5,output:4,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b00000000000002, duration=50.178s, table=0, n_packets=0, n_bytes=0, i
dle_age=50, priority=2,in_port=6 actions=output:1,output:7,output:9,output:8,out
put:3,output:2,output:5,output:4,output:10,output:12,output:11,output:14,output:
13,output:15,CONTROLLER:65535
 cookie=0x2b0000000000000b, duration=50.178s, table=0, n_packets=0, n_bytes=0, i
```

Figura 5.12: Topología de red GNS3 en ODL

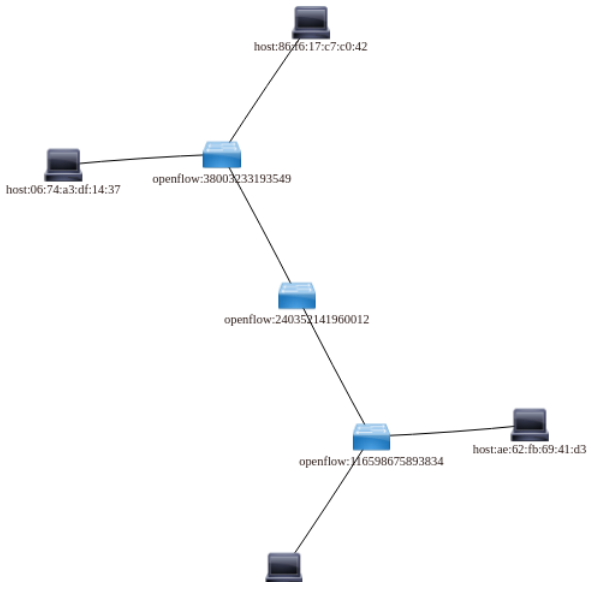


Figura 5.13: *Ping* del *host* 1 al *host* 3 antes de insertar las reglas drop12 y drop21

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.3
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=4.43 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=2.13 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.32 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=1.94 ms
64 bytes from 10.0.0.3: icmp_seq=5 ttl=64 time=2.12 ms
^C
--- 10.0.0.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 1.945/2.592/4.431/0.928 ms
root@UbuntuDockerGuest-1:~#
```

5.4.3. App OFM

La evaluación de las características de la aplicación OFM ya se realizó en el apartado 4.3.4, por lo tanto, ahora se focaliza más en los ejemplos a realizar para demostrar las funcionalidades de SDN. Se realizarán dos:

1. **La creación de dos "subredes"**. Mediante la configuración de dos reglas de flujo en el *switch* central, es posible deshabilitar la conexión entre los *hosts* de los extremos.

2. **Desviar el envío de paquetes ICMP a un servidor web.** En este caso se crea una regla de flujo en uno de los *switches* de los extremos que descarta los paquetes ICMP provenientes del *host* 1.

Ejemplo 1

En este ejemplo se quiere impedir el envío de paquetes entre los *hosts* 1 y 2 y los *hosts* 3 y 4. Antes de nada, se comprueba la conexión entre estos dispositivos mediante un *ping*. Como se observa en la Figura 5.13, el envío de paquetes entre los *hosts* 1 y 3 no falla.

Para conseguir el objetivo propuesto se han de añadir las siguiente reglas de flujo en el *switch* central. Antes de nada, se comprueba cuál es el *switch* central mirando el identificador en la topología de ODL (es el 240352141960012). Una vez se tiene, se pueden añadir las reglas de flujo correspondientes. En este caso las reglas tienen como identificador drop12 y drop21, y tienen como acción descartar todos los paquetes entrantes por el puerto 1 y 2 respectivamente. Estas reglas se observan en las Figuras 5.14 y 5.15.

Figura 5.14: Regla de flujo drop12

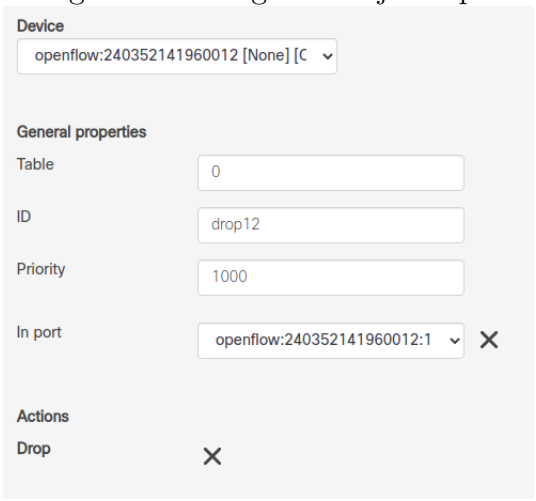
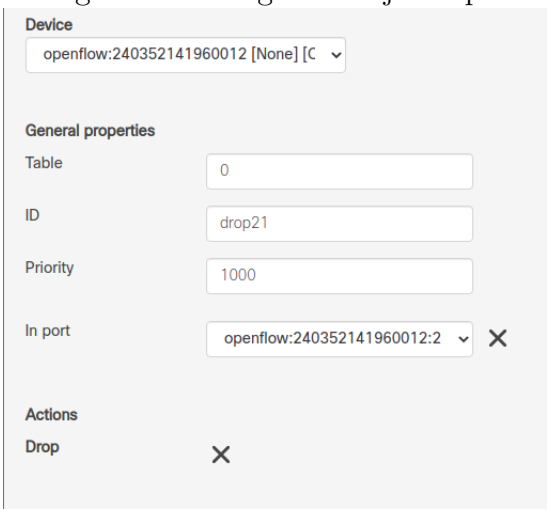


Figura 5.15: Regla de flujo drop21



Una vez añadidas las reglas en el *switch* hay que comprobar que están presentes en su tabla. En la Figura 5.16 se observa como la aplicación muestra que las reglas están en el dispositivo (ON DEVICE) y en la Figura 5.17 se muestra a través de la consola que contiene dichas reglas (resaltadas en rojo).

Para comprobar el funcionamiento de las reglas se realiza un *ping* desde el *host* 1 al 3, y como se observa en la Figura 5.18, no es satisfactorio.

Al eliminar dichas reglas, la conexión entre los *hosts* de los extremos vuelve a ser posible (ver Figura 5.19).

Figura 5.16: Reglas drop12 y drop21 en el dispositivo según OFM

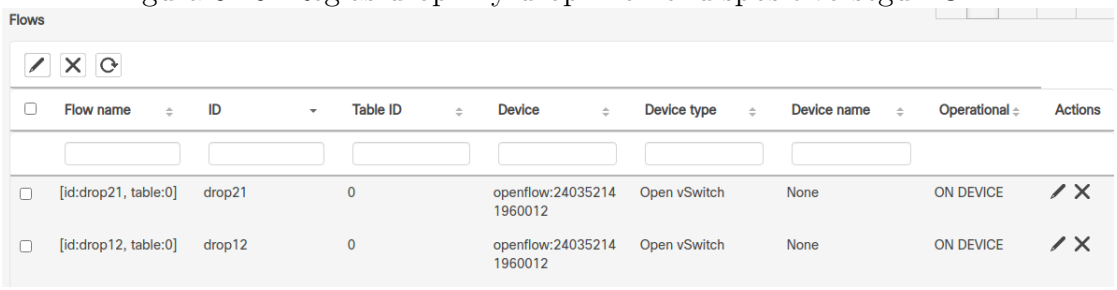
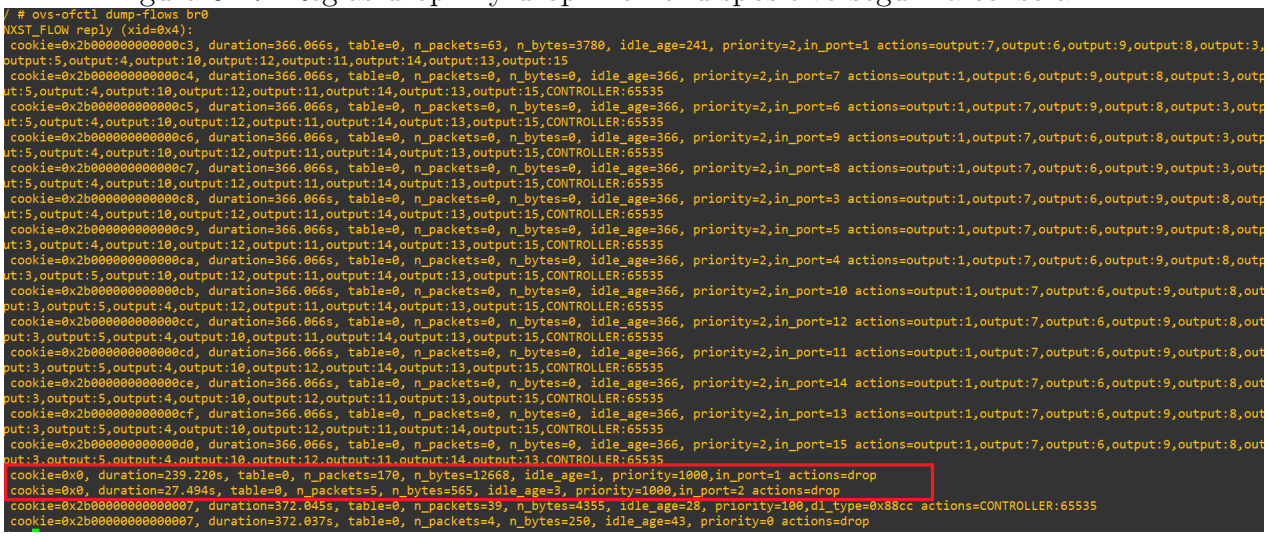


Figura 5.17: Reglas drop12 y drop21 en el dispositivo según la consola



Ejemplo 2

Este ejemplo quiere resaltar la funcionalidad de SDN en la que el *switch* es capaz de descartar los paquetes según su contenido. En este caso, el objetivo es descartar los paquetes ICMP provenientes del *host* 1 con destino al *host* 2, pero permitir la conexión TCP entre los mismos. Los pasos a seguir para su realización son los siguientes.

En primer lugar, dado que los *hosts* no vienen con Python ni con la función wget preinstaladas, se han de conectar a Inernet para descargarlos. Para ello, GNS3 ofrece un dispositivo NAT que permite el acceso a internet conectándose a él. Cambiada la

Figura 5.18: *Ping* del *host* 1 al *host* 3 tras insertar las reglas drop12 y drop21

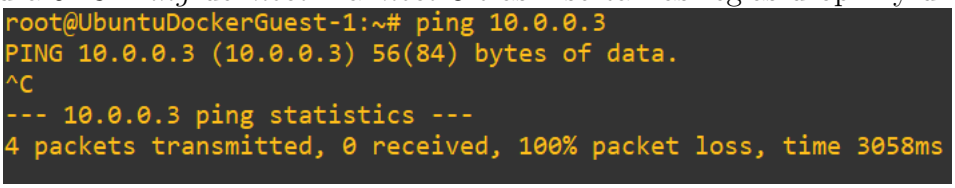


Figura 5.19: *Ping* del *host* 1 al *host* 3 tras eliminar las reglas drop12 y drop21

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.3
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=2.04 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=2.30 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.69 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 2.046/2.347/2.691/0.270 ms
```

Figura 5.20: *Ping* del *host* 1 al *host* 2 antes de insertar la regla dropICMP

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.2
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=1.21 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.921 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.738 ms
^C
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.738/0.959/1.218/0.197 ms
```

configuración de los *hosts* para la asignación de direcciones IP de forma estática a dinámica, se ejecutan los siguientes comandos en el *host* 1 y 2.

```
$ apt upgrade
$ apt install wget # En el host 1
$ apt install python # En el host 2
```

En segundo lugar, se vuelven a cambiar las direcciones IP a las anteriormente asignadas y a conectar los dispositivos como estaban antes. Una vez hecho esto, se comprueba el envío satisfactorio de paquetes ICMP entre el *host* 1 y 2 (ver Figura 5.20).

En tercer lugar, se añade la regla que descarta los paquetes ICMP provenientes del *host* 1 y que tienen como destino el *host* 2. Para ello, dentro de los parámetros de la regla se especifica lo siguiente (ver Figura 5.21):

- **Protocolo IP** en número decimal (ICMP es el 1).
- **Protocolo Ethernet** en hexadecimal (IPv4 o 0x0800).
- **Puerto** del que procede el paquete (el 1).
- **Acción** a realizar (descartar).

En cuarto lugar, se comprueba en la aplicación y en la consola que dicha regla ha sido insertada satisfactoriamente. Como se observa en la Figura 5.22, la regla dropICMP

Figura 5.21: Regla de flujo dropICMP

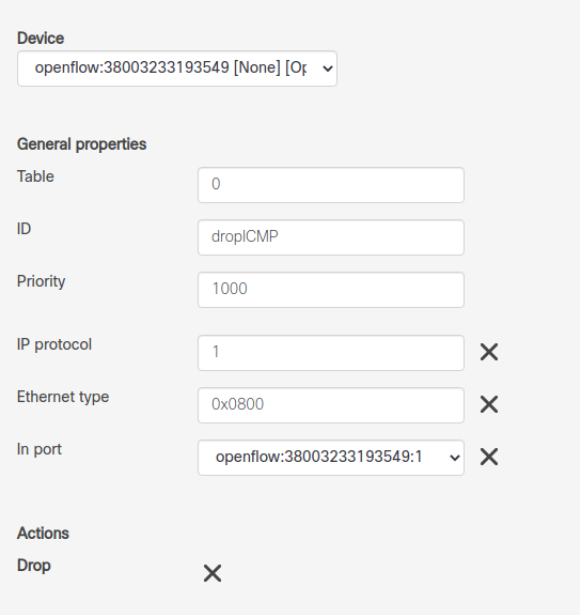
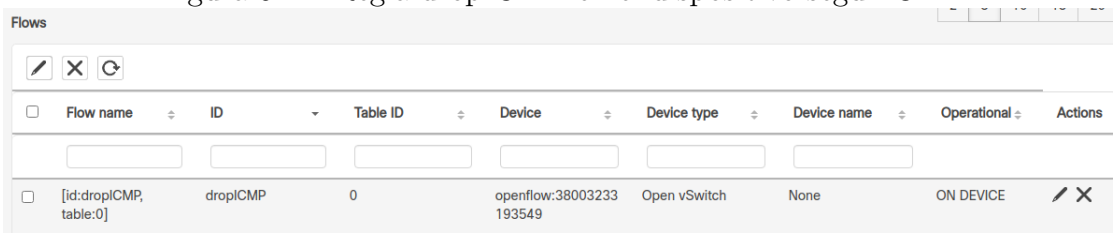


Figura 5.22: Regla dropICMP en el dispositivo según OFM



se encuentra ON DEVICE y como se puede comprobar en la Figura 5.23, en el interior rectángulo rojo se encuentra la regla creada.

En quinto lugar, se crea el servidor web Python en el *host* 2. Para ello se ejecuta el siguiente comando en su terminal.

```
$ python -m SimpleHTTPServer 80 &
```

En sexto lugar, se realiza un *ping* desde el *host* 1 al 2, cuyo resultado es negativo (ver Figura 5.24). Tras esto, se realiza una consulta *wget* desde el *host* 1 al 2, teniendo resultado positivo (ver Figura 5.25). Esto hace indicar que la regla de flujo ha funcionado tal y como se esperaba. Analizando las estadísticas del número de paquetes coincidentes con la regla, se puede demostrar que ha entrado en funcionamiento. Como se observa en la Figura 5.26, el número de paquetes que han cumplido la regla es 4, justamente los paquetes ICMP enviados.

Para finalizar, al eliminar la regla del *switch* los paquetes ICMP vuelven a recibirse, por lo tanto, el *ping* entre el *host* 1 y el 2 vuelve a ser posible (ver Figura 5.27).

Figura 5.23: Regla dropICMP en el dispositivo según la consola

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=272.988s, table=0, n_packets=4, n_bytes=392, idle_age=211,
 priority=1000,icmp,in_port=1 actions=drop
 cookie=0x2b00000000000002, duration=692.733s, table=0, n_packets=139, n_bytes=1
5707, idle_age=4, priority=100,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x2a00000000000000, duration=572.175s, table=0, n_packets=22, n_bytes=18
56, idle_timeout=1800, hard_timeout=3600, idle_age=40, priority=10,dl_src=9a:dc:
df:c2:e3:43,dl_dst=92:bc:94:ed:7a:f4 actions=output:2
 cookie=0x2a00000000000001, duration=572.175s, table=0, n_packets=22, n_bytes=25
36, idle_timeout=1800, hard_timeout=3600, idle_age=40, priority=10,dl_src=92:bc:
94:ed:7a:f4,dl_dst=9a:dc:df:c2:e3:43 actions=output:1
 cookie=0x2b0000000000000f, duration=686.993s, table=0, n_packets=0, n_bytes=0,
idle_age=686, priority=2,in_port=11 actions=output:10,output:13,output:12,output
:1,output:2,output:3,output:4,output:9,output:15,output:14,output:5,output:6,out
put:7,output:8,CONTROLLER:65535
 cookie=0x2b00000000000010, duration=686.988s, table=0, n_packets=0, n_bytes=0,
idle_age=686, priority=2,in_port=10 actions=output:11,output:13,output:12,output
:1,output:2,output:3,output:4,output:9,output:15,output:14,output:5,output:6,out
put:7,output:8,CONTROLLER:65535
 cookie=0x2b00000000000011, duration=686.983s, table=0, n_packets=0, n_bytes=0,
idle_age=686, priority=2,in_port=13 actions=output:11,output:10,output:12,output
:1,output:2,output:3,output:4,output:9,output:15,output:14,output:5,output:6,out
put:7,output:8,CONTROLLER:65535
 cookie=0x2b00000000000012, duration=686.975s, table=0, n_packets=0, n_bytes=0,
idle_age=686, priority=2,in_port=12 actions=output:11,output:10,output:13,output
:1,output:2,output:3,output:4,output:9,output:15,output:14,output:5,output:6,out
```

Figura 5.24: *Ping* del *host* 1 al *host* 2 tras insertar la regla dropICMP

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3056ms
```

Figura 5.25: Wget del *host* 1 al *host* 2 tras insertar la regla dropICMP

```
root@UbuntuDockerGuest-1:~# wget -O - 10.0.0.2
--2020-06-27 17:37:43--  http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 336 [text/html]
Saving to: 'STDOUT'

-                   0%[                    ]       0  --.-KB/s               <
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a>
<li><a href=".bashrc">.bashrc</a>
<li><a href=".gns3_perms">.gns3_perms</a>
<li><a href=".profile">.profile</a>
</ul>
<hr>
</body>
</html>
-                  100%[===================>]     336  --.-KB/s    in 0s

2020-06-27 17:37:43 (92.0 MB/s) - written to stdout [336/336]
```

Figura 5.26: Estadísticas de las reglas de flujo en OFM

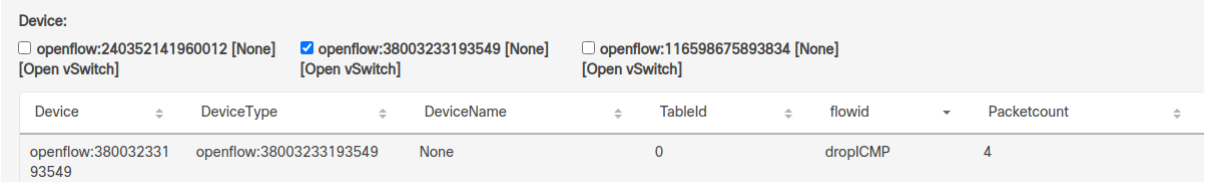


Figura 5.27: *Ping* del *host* 1 al *host* 2 tras eliminar la regla dropICMP

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.2
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=1.27 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.658 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.836 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=1.06 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3045ms
rtt min/avg/max/mdev = 0.658/0.957/1.272/0.233 ms
```

5.5. Controlador RYU

5.5.1. Introducción

Para la simulación de la red que se hace a lo largo del apartado, se utiliza el mismo controlador que en el Capítulo 4. A diferencia del anterior, la red consta de 4 *hosts*, tres *switches* OpenFlow, un *switch* tradicional y una nube. Esto ya se pudo comprobar en el apartado 5.3.

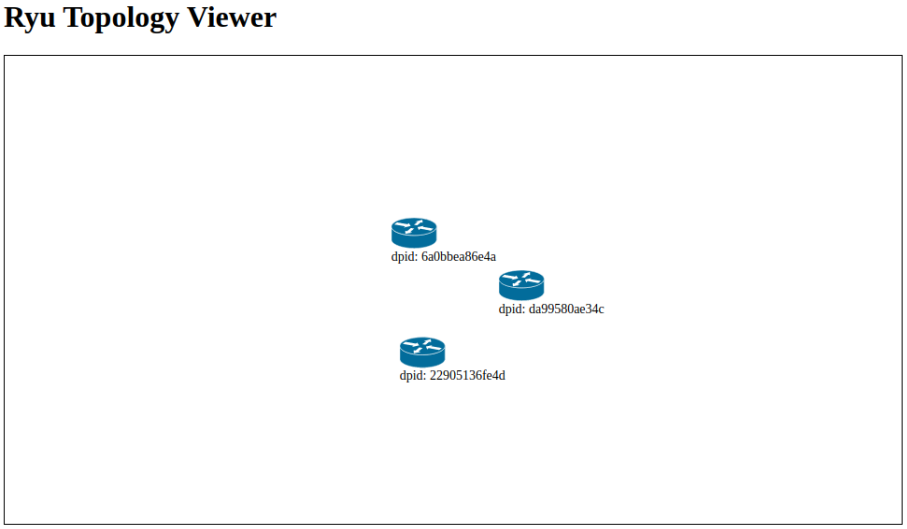
Dado que en el apartado 4.4.2 se analiza la función *simple switch* y puesto que aquí también se va a utilizar, no es necesario volver a explicarla. Además, como el tráfico OpenFlow ya fue analizado exhaustivamente en el apartado 4.4.3, únicamente se destacarán ciertos aspectos diferenciales.

Teniendo en cuenta todo lo anterior, en el siguiente apartado se realizará un análisis de las características del controlador RYU y de su influencia en el flujo de paquetes en la red, poniendo como ejemplo el envío de un paquete ICMP.

5.5.2. Análisis de la red

Asumiendo que el controlador ya se ha iniciado (como en el apartado 4.4.3), se ha de conectar cada *switch* a al controlador, para ello se ejecuta el siguiente comando en

Figura 5.28: GUI de RYU para red GNS3



el terminal.

```
$ ovs-vsctl set-controller br0 tcp:192.168.56.143:6633
```

Nota: Para evitar bucles se ha de habilitar el *Spanning Tree Protocol* y para ello se ejecuta el siguiente comando:

```
$ ovs-vsctl set bridge br0 stp_enable=true
```

Una vez conectado, al acceder al navegador y poner la dirección del controlador y el puerto 8080, se observan los *switches* OF en la GUI de RYU (ver Figura 5.28). Del mismo modo, al ejecutar la siguiente línea en el terminal de un *switch*, tiene que aparecer que el controlador está conectado (ver Figura 5.29).

```
$ ovs-vsctl show
```

Tras esto, se procede a analizar el contenido de las tablas de flujo en cada uno de los *switches*. Para ello, en cada terminal se ejecuta la siguiente línea:

```
$ ovs-ofctl dump-flows br0
```

El resultado de esta acción se observa en las Figuras 5.30, 5.31 y 5.32. Si se observa con detalle, se puede ver que existe una única entrada en la tabla y que esta tiene como acción reenviar los paquetes entrantes al controlador. Esto hace prever que cuando se realice un envío de paquetes (p.ej: tras hacer un *ping*) desde un *host* a otro, el *switch* los reenviará al controlador y este le responderá con la acción a ejecutar.

Figura 5.29: Lista de interfaces del *switch* 1

```
/ # ovs-vsctl show
d6066151-62c1-4c52-a38b-bea923f837e0
    Bridge "br0"
        Controller "tcp:192.168.56.143:6633"
            is_connected: true
        Port "eth8"
            Interface "eth8"
        Port "eth6"
            Interface "eth6"
        Port "eth7"
            Interface "eth7"
        Port "eth2"
            Interface "eth2"
        Port "eth5"
            Interface "eth5"
        Port "eth1"
            Interface "eth1"
        Port "eth12"
            Interface "eth12"
        Port "eth15"
            Interface "eth15"
        Port "eth4"
            Interface "eth4"
        Port "eth3"
```

Figura 5.30: Tabla de flujos del *switch* 1

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=146.304s, table=0, n_packets=81, n_bytes=4930, idle_age=0,
 priority=0 actions=CONTROLLER:65535
/ #
```

Figura 5.31: Tabla de flujos del *switch* 2

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=203.496s, table=0, n_packets=7, n_bytes=490, idle_age=146,
 priority=0 actions=CONTROLLER:65535
```

Figura 5.32: Tabla de flujos del *switch* 3

```
/ # ovs-ofctl dump-flows br0
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=239.581s, table=0, n_packets=129, n_bytes=7830, idle_age=1
, priority=0 actions=CONTROLLER:65535
```

Figura 5.33: *Ping* del *host* 1 al *host* 3

```
root@UbuntuDockerGuest-1:~# ping 10.0.0.3
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=37.3 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=2.57 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=2.11 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=2.17 ms
^C
--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 2.112/11.052/37.350/15.184 ms
```

Figura 5.34: Captura del flujo de mensajes OF

91	27.176735641	192.168.56.220	192.168.56.143	OpenF…	206	Type: OFPT_PACKET_IN
92	27.178168037	192.168.56.143	192.168.56.220	OpenF…	170	Type: OFPT_FLOW_MOD
93	27.179748174	192.168.56.200	192.168.56.143	OpenF…	206	Type: OFPT_PACKET_IN
94	27.180884336	192.168.56.143	192.168.56.200	OpenF…	170	Type: OFPT_FLOW_MOD
95	27.182251593	192.168.56.230	192.168.56.143	OpenF…	206	Type: OFPT_PACKET_IN
96	27.187444135	192.168.56.143	192.168.56.230	OpenF…	170	Type: OFPT_FLOW_MOD

Para demostrar lo anteriormente mencionado se realiza un *ping* desde h1 hasta h3 (ver Figura 5.33) y se analiza el tráfico generado a través de Wirseshark.

A continuación, se procede a describir el flujo de paquetes OF (ver Figura 5.34).

En primer lugar, se observa como el *switch* conectado al *host* 1 envía un mensaje PACKET-IN al controlador. Este le responde con un FLOW-MOD, y la acción a realizar es reenviar el paquete por el puerto 3, es decir el que está conectado al *switch* central(ver Figura 5.35).

En segundo lugar, cuando el *switch* central recibe el paquete envía un mensaje PACKET-IN al controlador. Este le responde con un FLOW-MOD que tiene como acción reenviar el paquete por el puerto 2, es decir, el que está conectado al *switch* 3 (ver Figura 5.36).

Para finalizar, cuando el tercer *switch* recibe el paquete, envía un mensaje PACKET-IN al controlador. Este le responde con un FLOW-MOD que tiene como acción reenviar el paquete por el puerto 1, puerto al que está conectado el *host* 3 (ver Figura 5.37).

Es importante tener en cuenta que previamente al envío de los paquetes ICMP se realiza un envío de mensajes ARP que sirven para localizar donde se encuentra el *host* 3. El funcionamiento para el envío de estas tramas es el mismo que para los paquetes ICMP descritos anteriormente.

Figura 5.35: FLOW-MOD para el *switch* 2

```
> Frame 92: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface enp0s8, id 0
> Ethernet II, Src: PcsCompu_8e:5d:89 (08:00:27:8e:5d:89), Dst: f6:d6:15:dd:92:70 (f6:d6:15:dd:92:70)
> Internet Protocol Version 4, Src: 192.168.56.143, Dst: 192.168.56.220
> Transmission Control Protocol, Src Port: 6633, Dst Port: 41416, Seq: 305, Ack: 721, Len: 104
v OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FLOW_MOD (14)
    Length: 104
    Transaction ID: 1691612828
    Cookie: 0x0000000000000000
    Cookie mask: 0x0000000000000000
    Table ID: 0
    Command: OFPFC_ADD (0)
    Idle timeout: 0
    Hard timeout: 0
    Priority: 1
    Buffer ID: 272
    Out port: 0
    Out group: 0
  > Flags: 0x0000
    Pad: 0000
  > Match
  v Instruction
      Type: OFPIT_APPLY_ACTIONS (4)
      Length: 24
      Pad: 00000000
    v Action
        Type: OFPAT_OUTPUT (0)
        Length: 16
        Port: 3
        Max length: 65509
        Pad: 000000000000
```

Figura 5.36: FLOW-MOD para el *switch* central

```
> Frame 94: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface enp0s8, id 0
> Ethernet II, Src: PcsCompu_8e:5d:89 (08:00:27:8e:5d:89), Dst: MS-NLB-PhysServer-04_c0:7a:17:24 (02:04:c0:7a:17:24)
> Internet Protocol Version 4, Src: 192.168.56.143, Dst: 192.168.56.200
> Transmission Control Protocol, Src Port: 6633, Dst Port: 35084, Seq: 305, Ack: 721, Len: 104
v OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FLOW_MOD (14)
    Length: 104
    Transaction ID: 229630161
    Cookie: 0x0000000000000000
    Cookie mask: 0x0000000000000000
    Table ID: 0
    Command: OFPFC_ADD (0)
    Idle timeout: 0
    Hard timeout: 0
    Priority: 1
    Buffer ID: 272
    Out port: 0
    Out group: 0
  > Flags: 0x0000
    Pad: 0000
  > Match
  v Instruction
      Type: OFPIT_APPLY_ACTIONS (4)
      Length: 24
      Pad: 00000000
    v Action
        Type: OFPAT_OUTPUT (0)
        Length: 16
        Port: 2
        Max length: 65509
        Pad: 000000000000
```

Figura 5.37: FLOW-MOD para el *switch* 3

```
> Frame 96: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface enp0s8, id 0
> Ethernet II, Src: PcsCompu_8e:5d:89 (08:00:27:8e:5d:89), Dst: fe:24:2c:2c:48:28 (fe:24:2c:2c:48:28)
> Internet Protocol Version 4, Src: 192.168.56.143, Dst: 192.168.56.230
> Transmission Control Protocol, Src Port: 6633, Dst Port: 33932, Seq: 305, Ack: 721, Len: 104
v OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FLOW_MOD (14)
    Length: 104
    Transaction ID: 1250287551
    Cookie: 0x0000000000000000
    Cookie mask: 0x0000000000000000
    Table ID: 0
    Command: OFPFC_ADD (0)
    Idle timeout: 0
    Hard timeout: 0
    Priority: 1
    Buffer ID: 272
    Out port: 0
    Out group: 0
  > Flags: 0x0000
    Pad: 0000
  > Match
  v Instruction
      Type: OFPIT_APPLY_ACTIONS (4)
      Length: 24
      Pad: 00000000
    v Action
        Type: OFPAT_OUTPUT (0)
        Length: 16
        Port: 1
        Max length: 65509
        Pad: 000000000000
```

Capítulo 6

Aplicación práctica con Zodiac FX

6.1. Introducción

En este capítulo se propone crear una plataforma redundante de control SDN-IoT con la intención de demostrar la importancia de esa característica. Para ello se vale de: dos controladores, un sensor de humedad y temperatura y un *script* escrito en Python. Este último se encarga de automatizar el proceso de establecer el controlador en los *switches* y para ello toma como referencia el estado de la conexión con el controlador (si hay o no hay conexión).

Para analizar esta cuestión se ha dividido el capítulo en los siguientes puntos:

- **Análisis de los componentes de red**. Se nombran y explican los distintos dispositivos que forman la red y también las funciones que desempeñan.
- **Creación del escenario de red**. Se describe el conexionado, la topología y las distintas direcciones para los elementos de red.
- **Análisis del escenario de red**. Se examina el comportamiento de la red y se demuestra la importancia del uso de un doble controlador en un esquema de red SDN.

6.2. Análisis de los componentes de la red

Los distintos componentes utilizados para la realización del escenario de red son:

- Tres Raspberry Pi 4
- Una Raspberry Pi 3
- Dos *switches* Zodiac FX
- Dos ordenadores portátiles
- Un *router* TP-Link
- Un sensor de humedad y temperatura

A continuación se procede a describir las funciones de cada uno.

En primer lugar, dos de las tres Raspberry Pi 4 ejercen de controlador. En este caso, el controlador utilizado es RYU debido a que tiene un bajo consumo de recursos y proporciona las funcionalidades básicas para la transmisión de paquetes. La Raspberry restante tiene la función de ejercer como servidor en la conexión con el dispositivo IoT. El archivo que se ejecuta en esta, está programado en Python y se encuentra en el Anexo D.2.

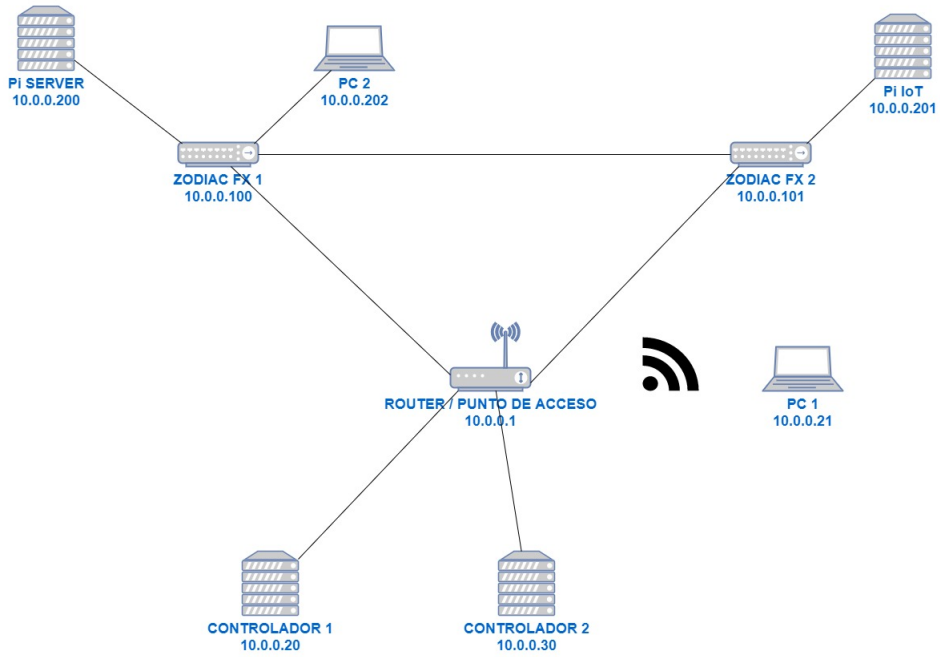
En segundo lugar, la Raspberry Pi 3 ejerce como dispositivo IoT. A ella le llegan los datos del sensor a través de su GPIO, los analiza, los procesa y los envía a través de la red al servidor. Para realizar esta función ejecuta el archivo Python presente en el Anexo D.3.

En tercer lugar, los dos *switches* Zodiac FX ejercen como *switches* OF puros, es decir que si no tienen un controlador asociado no son capaces de reenviar paquetes a través de sus puertos.

En cuarto lugar, los ordenadores portátiles acceden a las distintas Raspberry de manera remota y a los *switches* a través de una conexión serie. Uno de los ordenadores accede a los dos controladores y a los dos *switches* y el otro accede únicamente al servidor y al cliente IoT.

En quinto lugar, el *router* TP-Link ejerce como *switch* para la conexión de las dos Zodiac FX a los controladores y como un punto de acceso inalámbrico para uno de los ordenadores portátiles, concretamente el que accede de manera remota a los controladores.

Figura 6.1: Esquema de red SDN-IoT



Finalmente, el sensor de humedad y temperatura (un DHT11) se encarga de captar los valores y transmitirlos a la Raspberry. Es importante destacar que se encuentra acoplado en una protoboard.

6.3. Creación del escenario de red IoT

Para la realización de la red se ha dispuesto de una topología como la presente en la Figura 6.1. En ella se observa como los *switches* y controladores se conectan al *router*. Esto es como consecuencia de dos aspectos fundamentales.

Por un lado, las Zodiac FX solo tienen habilitado un puerto para conectarse al controlador (el puerto 4), por lo tanto, si se quieren conectar dos controladores a un único *switch*, se ha de utilizar otro dispositivo que lo permita. En este caso el *router* ejerce perfectamente la conmutación de paquetes y hace que la conexión sea posible.

Por otro lado, los controladores disponen únicamente de un puerto Ethernet, lo que imposibilita conectar más de un dispositivo a la vez (p.ej.: No pueden conectarse las dos Zodiac FX a la vez) y hace necesario el uso de un elemento extra.

Paralelamente, el *router* sirve como punto de acceso inalámbrico para el PC 1 dado que este no dispone de un puerto Ethernet.

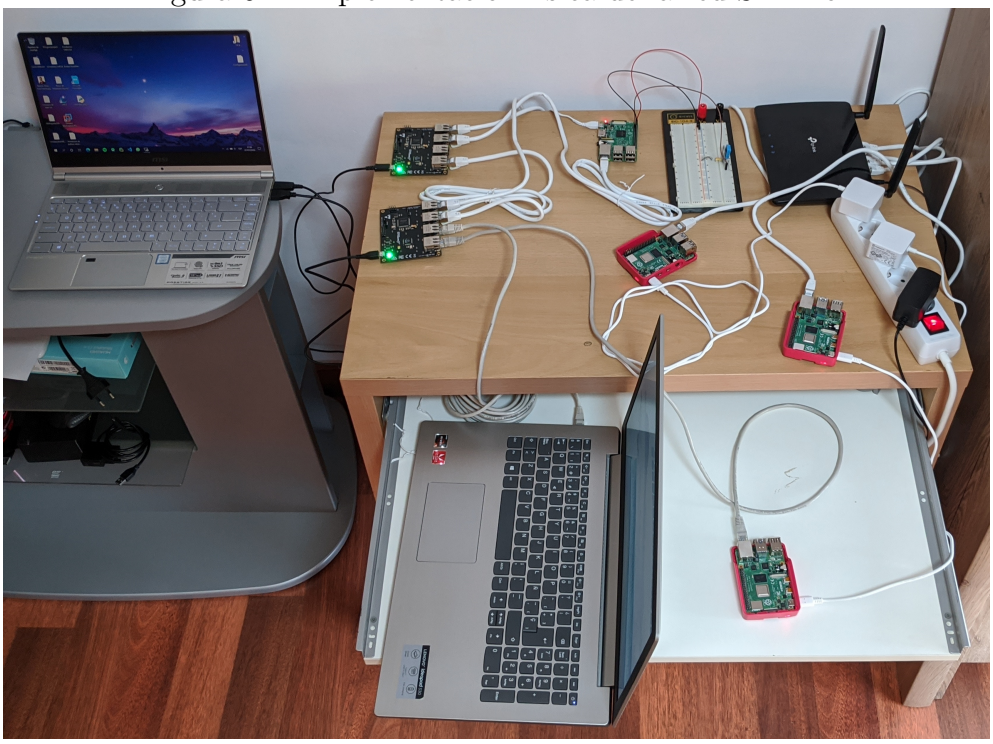
Teniendo en cuenta los anteriores aspectos, se configuran las distintas direcciones IP de manera estática para la red 10.0.0.0/24. La Tabla 6.1 muestra las direcciones y conexiones para cada dispositivo. Sobra mencionar que las conexiones cableadas entre los dispositivos se realizan mediante el uso de cables Ethernet RJ45, excepto la conexión serie entre el PC1 y las Zodiac FX que sea realiza con cables Micro USB.

Dispositivo	**Dirección IP**	**Conexiones**
Router	10.0.0.1	Puerto 1: Controlador 1 Puerto 2: Controlador 2 Puerto 3: Zodiac Fx 1 Puerto 4: Zodiac Fx 2 Wi-Fi: PC 1
Controlador 1	10.0.0.20	Puerto 1: Router
Controlador 2	10.0.0.30	Puerto 1: Router
Zodiac FX 1	10.0.0.100	Puerto 1: Pi Server Puerto 2: PC 2 Puerto 3: Zodiac Fx 2 Puerto 4: Router Puerto Micro USB: PC 1
Zodiac FX 2	10.0.0.101	Puerto 1: Zodiac FX 1 Puerto 3: Pi IoT Puerto 4: Router Puerto Micro USB: PC 1
Pi Server	10.0.0.200	Puerto 1: Zodiac FX 1
Pi IoT	10.0.0.201	Puerto 1: Zodiac FX 2
PC 1	10.0.0.21	Wi-Fi: Router COM3: Zodiac FX 2 COM4: Zodiac FX 1
PC 2	10.0.0.202	Puerto 1: Zodiac FX 1

Tabla 6.1: Conexiones y direcciones IP de los dispositivos

La conexión serie entre los *switches* y el PC 1 sirve además de para ejercer como fuente de alimentación, poder ejecutar el *script* de automatización de los controladores.

Figura 6.2: Implementación física de la red SDN-IoT



Finalmente, el esquema y la asignación de las conexiones da como resultado el escenario de red presente en la Figura 6.2.

6.4. Análisis del escenario de red IoT

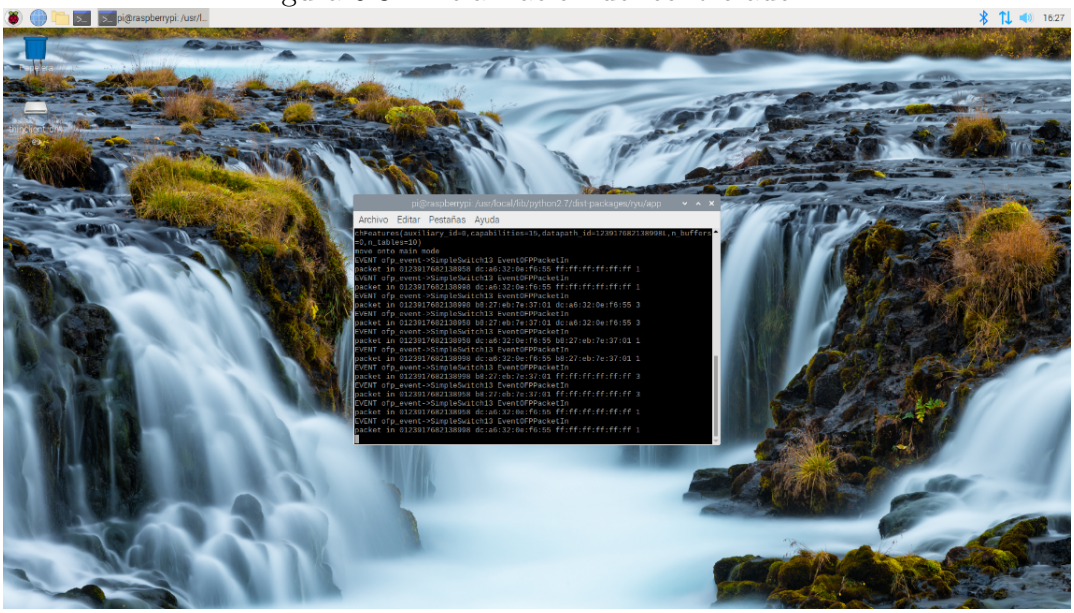
Introducción

Con el objetivo de analizar el comportamiento de una red de con doble controlador SDN y ver su importancia cuando se utilizan *switches* OF puros, se crea el escenario de red mencionado en el punto anterior. Para añadirle más interés se verifica su comportamiento mediante el envío de datos por parte de un sensor a través de la red y el envío de paquetes ICMP.

Para una mejor compresión se divide el análisis en tres puntos:

- **Estado inicial**. Se comprueban la configuración inicial y el tráfico presente en la red.
- **Modificación de la red**. Se hace uso del *script* para la automatización del controlador y se observan las consecuencias de la desconexión de la del controlador que estaba en ejecución.

Figura 6.3: Inicialización del controlador 1



- **Conclusiones**. Se realiza una reflexión final sobre la importancia de la redundancia en SDN.

Estado inicial

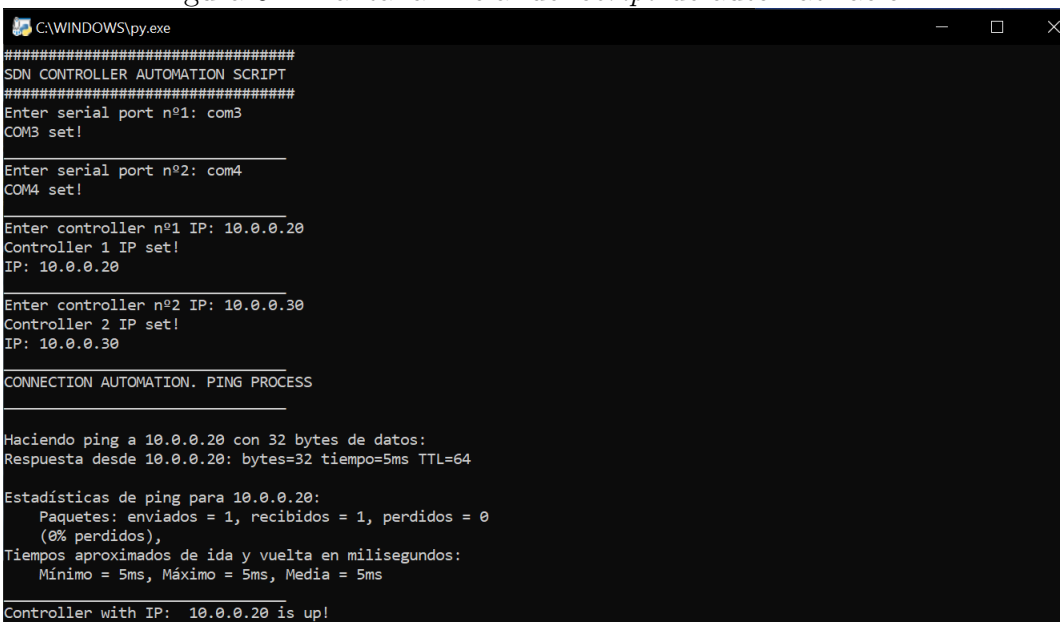
Para el establecimiento de la conexión entre los dispositivos conectados a través de los *switches* OF, se ha de iniciar un controlador. En este caso el controlador utilizado es RYU y se ejecuta en el dispositivo que tiene la IP 10.0.0.20. Este será el controlador por defecto de la red.

Teniendo en cuenta lo anterior se ejecutan el controlador por defecto (ver Figura 6.3) y el *script* para la automatización de los controladores (ver Anexo D.1).

Respecto al funcionamiento del *script*, como se puede observar en la Figura 6.4, dispone de una consola en la que se insertan los distintos parámetros necesarios (IP y puertos serie). Hecho esto, la ejecución se basa en la realización de *pings* cada cinco segundos a la primera IP insertada (en este caso 10.0.0.20). En caso de haber respuesta positiva, se comunica a los *switches* a través de los puertos serie, las instrucciones necesarias para establecer como controlador la dirección IP resultante. En caso negativo, la asignación del controlador sería asociada con la segunda IP insertada.

Finalmente y con el objetivo de comprobar que todo funciona correctamente, se inicia la transmisión de datos por parte del dispositivo IoT al servidor (ver Figura 6.5).

Figura 6.4: Pantalla inicial del *script* de automatización



Al mismo tiempo, se realiza un *ping* sin límite de número de paquetes por parte del PC al servidor. Este tiene el objetivo de comprobar el número de paquetes perdidos al realizar el cambio automático de controlador.

Modificación de la red

Para observar el comportamiento de la automatización de los controladores y la importancia de la redundancia se procede a desconectar el cable Ethernet del puerto del controlador 1 (el que se está ejecutando). Es importante mencionar que antes de la desconexión del controlador 1 hay que iniciar el controlador 2 (ver Figura 6.6) en caso de que no esté iniciado. De esta manera cuando se establezca en los *switches* este controlador, podrán recibir sus reglas.

Al desconectar el cable Ethernet del puerto del controlador 1, cuando el *script* le realice el *ping*, será negativo. Tras esto, se inicia el proceso de automatización y se establece como controlador al controlador 2 (10.0.0.30). Esto se puede observar en la Figura 6.7. Al mismo tiempo, se verifica el correcto funcionamiento del *script* accediendo a la consola de uno de los *switches* y mostrando su configuración (ver Figura 6.8). En este caso se observa que ahora el controlador está asociado a la IP 10.0.0.30.

Figura 6.5: Inicio de la transmisión de datos IoT. Pi Server (izq.) - Pi IoT (der.)

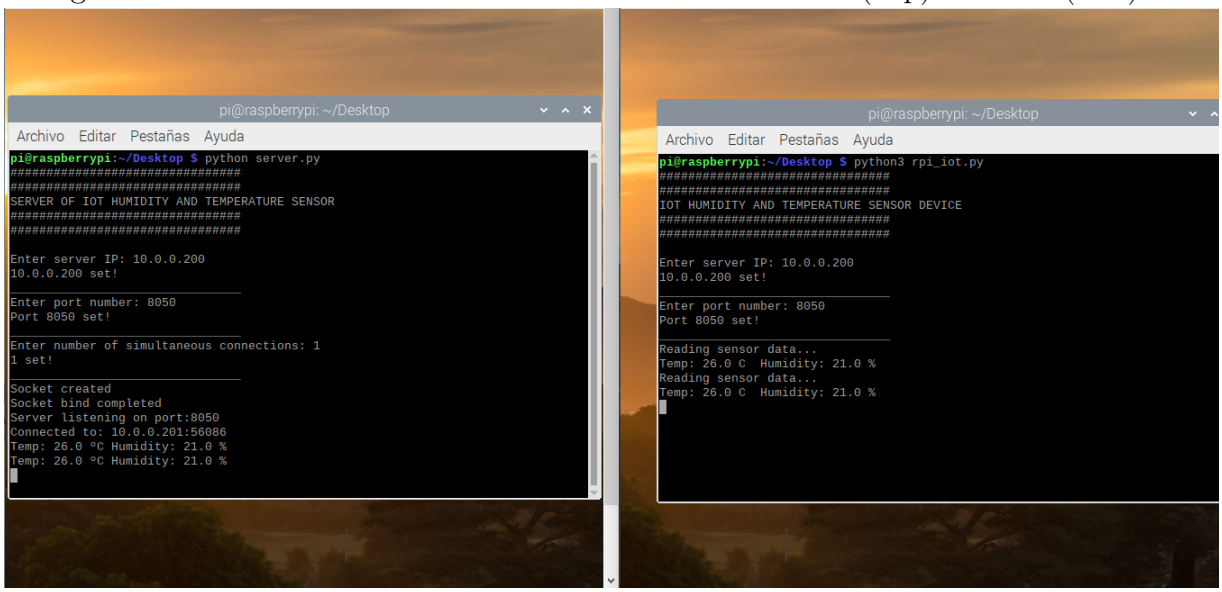


Figura 6.6: Inicialización del controlador 2

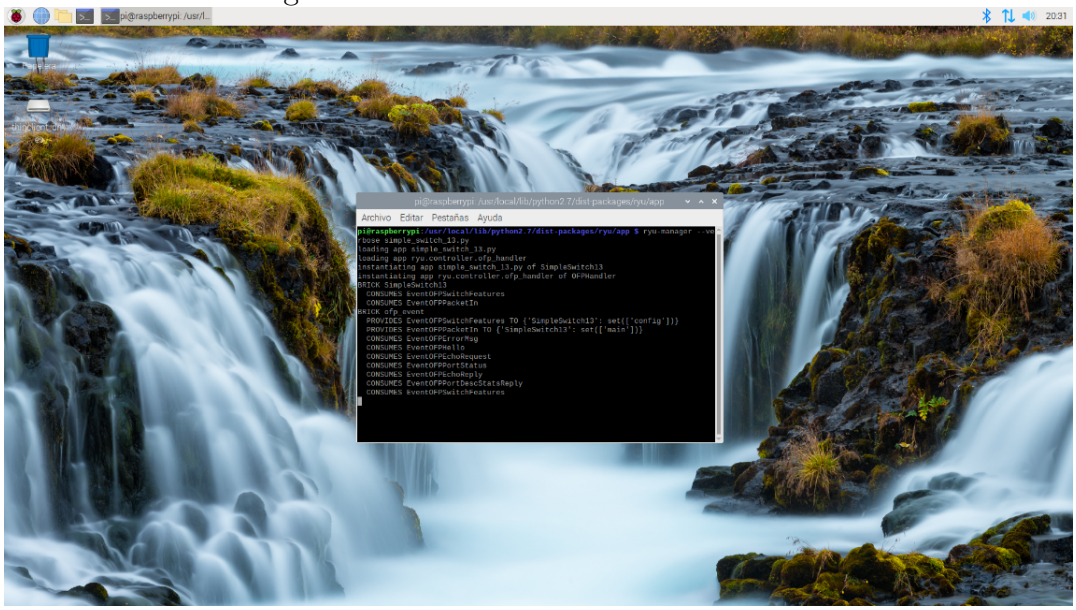


Figura 6.7: *Script* de automatización en acción

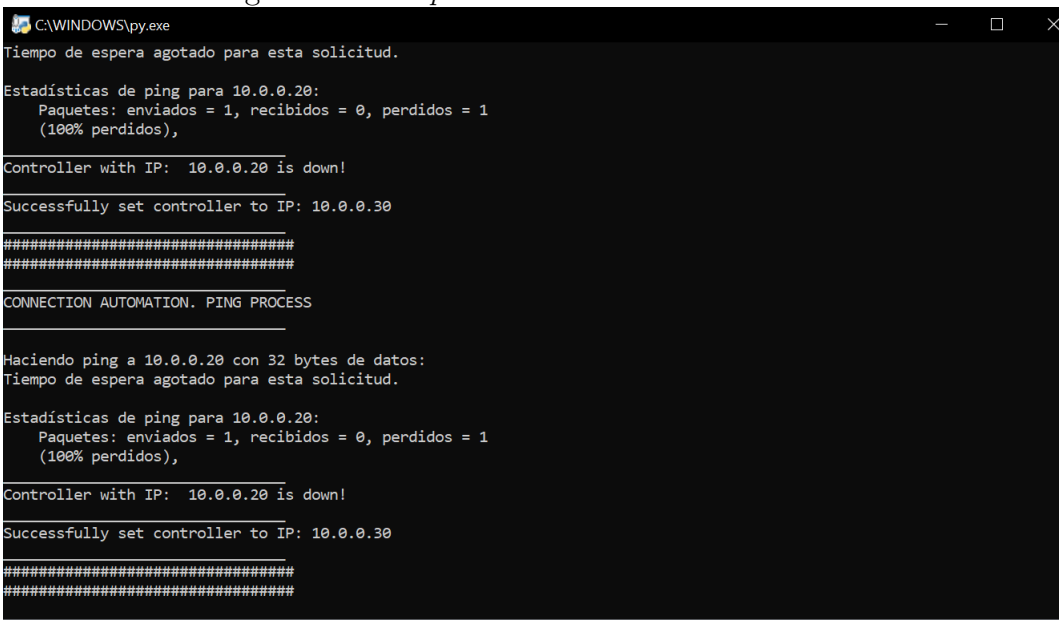


Figura 6.8: Datos de configuración del Zodiac FX 2

```
Zodiac_FX# config
Zodiac_FX(config)# show config

------------------------------------------------------------------------
Configuration
 Name: Zodiac_FX
 MAC Address: 70:B3:D5:6C:DF:4E
 IP Address: 10.0.0.101
 Netmask: 255.255.255.0
 Gateway: 10.0.0.1
 OpenFlow Controller: 10.0.0.30
 OpenFlow Port: 6633
 Openflow Status: Enabled
 Failstate: Secure
 Force OpenFlow version: Disabled
 EtherType Filtering: Disabled

------------------------------------------------------------------------
```

Figura 6.9: Resultado de la realización del *ping* a Pi Server

```
Símbolo del sistema
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo=11ms TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo=372ms TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Tiempo de espera agotado para esta solicitud.
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64
Respuesta desde 10.0.0.200: bytes=32 tiempo<1m TTL=64

Estadísticas de ping para 10.0.0.200:
    Paquetes: enviados = 90, recibidos = 89, perdidos = 1
    (1% perdidos),
Tiempos aproximados de ida y vuelta en milisegundos:
    Mínimo = 0ms, Máximo = 372ms, Media = 18ms
Control-C
^C
```

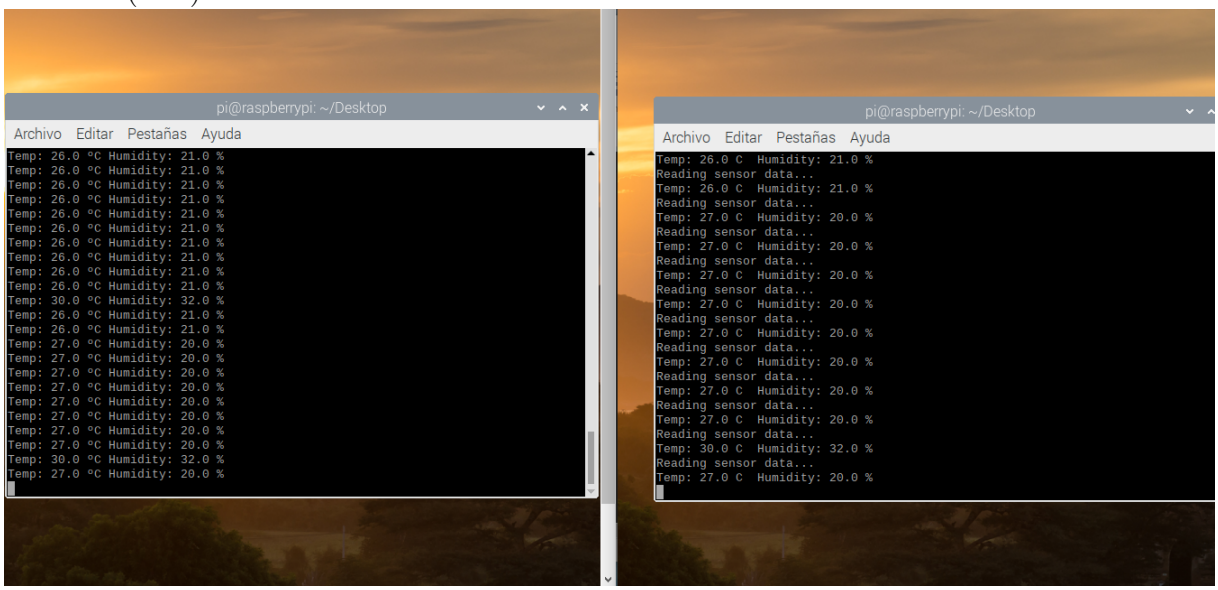
Para ver como ha afectado este proceso al flujo de paquetes, se observa el *ping* a Pi Server por parte del PC 2 que se realizó en el apartado anterior. Como se puede ver en la Figura 6.9, gracias al doble controlador y a la automatización, únicamente se perdió un paquete ICMP de los noventa enviados.

Finalmente, al analizar los datos enviados entre el servidor y el cliente IoT, se observa que no se ha perdido ningún paquete y que la conexión se mantuvo estable entre ambos (ver Figura 6.10).

Conclusiones

Tras analizar este experimento, se demuestra la importancia de la redundancia en una red SDN para *switches* OF puros. En caso de que surgiera algún problema con el controlador principal (p.ej.: caída de la conexión), entra en acción un controlador de sustitución, lo que hace posible que el tráfico en la red no se vea afectado.

Figura 6.10: Transmisión de datos IoT tras el cambio de controlador. Pi Server (izq.) - Pi IoT (der.)



Capítulo 7

Conclusiones y líneas futuras

7.1. Conclusiones

El Trabajo Fin de Grado presentado a lo largo de esta memoria tiene como objetivo principal el estudio de SDN, una tecnología que permite flexibilizar y agilizar el comportamiento de la red. Este objetivo se consigue a través del desarrollo de simulaciones y la implementación física de un escenario de red SDN-IoT con doble controlador.

Por una parte, para el desarrollo de las simulaciones se han utilizado dos plataformas (Mininet y GNS3) y dos controladores (ODL y RYU). A continuación se destacan los pros y contras de cada uno.

- De Mininet destacan su sencillez y fácil instalación, a la par que su bajo consumo de recursos para la creación y simulación de redes. Contrariamente, debido a su sencillez provoca que no se puedan simular entornos más específicos, ya que por ejemplo, las topologías de red están especificadas según unos parámetros y no puedes salirte de ellos.

- Para GNS3 resaltan los aspectos de tener una mayor capacidad para simulaciones de redes más específicas, ya que gracias a sus *appliances* puedes tener exactamente los mismos dispositivos en la red simulada que en una red física. Al mismo tiempo, gracias al uso de la tecnología Docker hace que una simulación de una red sea menos pesada que si esta se hiciera con máquinas virtuales. Por el contrario, es un *software* que consume muchos recursos, y por tanto si no se dispone de amplios recursos de computación la experiencia de simulación no es buena. Del

mismo modo, la instalación de la máquina virtual sobre la que se ejecutan los contenedores Docker es bastante compleja si se utiliza VirtualBox, debido a que la página oficial no proporciona ayuda (el tutorial para la instalación únicamente está disponible para VMWare Workstation) y se ha de investigar por cuenta propia como hacerlo y buscar el complemento necesario para ello.

- ODL destaca en el aspecto de que es un controlador bastante completo. Esto se debe a que dispone de bastantes *plugins* que permiten tener desde una GUI hasta integrar otra aplicación como OFM para la edición de las reglas de flujo en los *switches* y así realizar redes más complejas (se pueden filtrar y direccionar paquetes según el contenido de los mismos) y poder aprovechar al máximo las características de SDN. Por el contrario, es un controlador que consume bastantes recursos, ya que si quieres editar los flujos de manera sencilla se ha de usar conjuntamente con OFM. Al mismo tiempo también tiene el defecto de ser bastante tediosa su instalación, debido a que no muchas versiones son compatibles (por temas de la JVM) y se ha de comprobar cual es la correcta de manera empírica.

- RYU es un controlador bastante recomendable si lo que se quiere es únicamente comprobar la funcionalidad de SDN y gastar pocos recursos computacionales. Del mismo modo, su instalación es bastante sencilla, no hay problemas de compatibilidad y está detallada en su web. De manera contraria, si se quiere tener una experiencia más personalizable existen dos opciones, o bien se añaden *plugins* a RYU (p.ej.: Faucet) o se utiliza otro controlador (p.ej.: ODL), ya que RYU únicamente se encarga de decirle a los *switches* que reenvíen los paquetes por el puerto que sea necesario, dando igual el tipo de información que fuera en ellos, es decir, no tiene la capacidad de filtrar los paquetes según su contenido.

A modo de resumen, la Tabla 7.1 muestra las principales ventajas y desventajas de cada uno de ellos.

Por otra parte, para la implementación física de un escenario de red SDN-IoT con doble controlador se demuestra lo siguiente:

- La importancia de la redundancia en SDN para redes formadas por *switches* OF puros, puesto que en el caso de existiese únicamente un controlador, en el

Elemento	Pros	Contras
Mininet	▪ Sencillez. ▪ Fácil instalación. ▪ Bajo consumo de recursos.	▪ No permite simulaciones específicas. ▪ Esquemas de red fijos.
GNS3	▪ Alto nivel de personalización de la red. ▪ Uso de Docker.	▪ Complejidad para iniciar. ▪ Difícil instalación de la máquina virtual. ▪ Alto consumo de recursos para redes simples.
ODL	▪ Integración de numerosos *plugins* de manera sencilla. ▪ Capacidad para personalizar reglas de flujo de forma sencilla junto a OFM.	▪ Problemas de versiones. ▪ Dificultad para la instalación sin previos conocimientos. ▪ Alto consumo de recursos.
RYU	▪ Fácil instalación. ▪ Uso sencillo a través de plantillas. ▪ Consumo bajo de recursos.	▪ Difícil capacidad de personalización de reglas de flujo. ▪ Para un uso básico de SDN.

Tabla 7.1: Pros y contras de los elementos usados en las simulaciones

supuesto de que se desconectara de la red, esta quedaría totalmente inutilizable ya que los *switches* no sabrían como direccionar los paquetes entrantes.

- La automatización como elemento diferencial en la gestión de redes. Se demuestra como gracias a la automatización la red sigue funcionando correctamente y prácticamente no se pierden paquetes, lo que hace también que el usuario apenas note diferencia. Esto quedó demostrado para el caso de la transmisión de datos entre el dispositivo IoT y el servidor, puesto que esta no se vio afectada en ningún momento.

Finalmente, gracias al estudio teórico y la posterior aplicación práctica de los conocimientos adquiridos durante el desarrollo de este T.F.G. se puede llegar a tener una idea del funcionamiento de la nueva manera de implementar redes de comunicación y como estas van a jugar un papel importante en los próximos años debido al constante auge de dispositivos conectados y a la demanda continua de la sociedad por una baja latencia y alta velocidad de conexión.

7.2. Líneas Futuras

Tras la realización de este Trabajo Fin de Grado se sugieren las siguientes de estudio para el futuro:

- La implementación de NFV en la red a través del uso de *switches* presentes en máquinas virtuales y no con el uso de las Zodiac Fx.
- El envío de los datos del dispositivo IoT a un servidor web y posteriormente descargarlos y leerlos desde cualquier dispositivo.
- El uso de otros controladores como ONOS para la realización de las simulaciones, puesto que ofrece muchas funcionalidades y versatilidad sin necesidad de depender de otras aplicaciones. Al mismo tiempo, no trata directamente con el protocolo OpenFlow sino que lo encapsula creando así una capa común en el controlador y las aplicaciones.
- El análisis del concepto de *Telco-Cloud*, complemento del 5G, presente cada vez más en los operadores móviles debido a la necesidad de redes ágiles y versátiles.

Bibliografía

[1] D. Evans, «The internet of things: How the next evolution of the internet is changing everything», *CISCO white paper*, vol. 1, n.º 2011, págs. 1-11, 2011.

[2] *Public cloud market revenue worldwide from 2012 to 2027*, nov. de 2018. dirección: `https://www.statista.com/statistics/477702/public-cloud-vendor-revenue-forecast/`.

[3] Ericsson, «Ericsson mobility report», *Ericsson, Sweden, Tech. Rep.*, jun. de 2019.

[4] *SDN Versus Traditional Networking Explained*, ago. de 2019. dirección: `https://www.ibm.com/services/network/sdn-versus-traditional-networking`.

[5] J.F.R.Báez, «Diseño de una plataforma SDN/IoT mediante OpenFlow», Universidad de las Palmas de Gran Canaria. IDeTIC, 2019.

[6] *Software-Defined Networking (SDN) Definition*. dirección: `https://www.opennetworking.org/sdn-definition/`.

[7] M. Boucadair y C. Jacquenet, «Software-defined networking: A perspective from within a service provider environment», en *RFC 7149*, 2014.

[8] D. Sánchez Álvarez, R. Diego y G. David, «Diseño e implementación de una fuente de datos tipo SNORT basada en la arquitectura SELFNET», Tesis de mtría., Universidad Complutense de Madrid, 2017.

[9] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer y O. Koufopavlou, «Software-defined networking (SDN): Layers and architecture terminology», en *RFC 7426*, IRTF, 2015.

[10] M. Betts, S. Fratini, N. Davis, D. Hoods, R. Dolin, M. Joshi y Z. Dacheng, «SDN Architecture, Issue 1, Open Networking Foundation», ONF TR-502, inf. téc., 2014.

[11] S. Makam, *SDN Openflow commercial applications – Part 1*, ago. de 2014. dirección: `https://sreeninet.wordpress.com/2014/08/03/sdn-commercial-applications/`.

[12] ——, *SDN Openflow commercial applications – Part 2*, ago. de 2014. dirección: `https://sreeninet.wordpress.com/2014/08/09/sdn-openflow-commercial-applications-part-2/`.

[13] A. Feghali, R. Kilany y M. Chamoun, «SDN security problems and solutions analysis», en *2015 International Conference on Protocol Engineering (ICPE) and International Conference on New Technologies of Distributed Systems (NTDS)*, IEEE, 2015, págs. 1-5.

[14] *SDN Security - Challenges Implementing SDN Network Security in SDN Environments*, nov. de 2013. dirección: `https://www.sdxcentral.com/networking/sdn/definitions/security-challenges-sdn-software-defined-networks/`.

[15] J. Gil Herrera y J. F. Botero, «Resource Allocation in NFV: A Comprehensive Survey», *IEEE Transactions on Network and Service Management*, vol. 13, n.º 3, págs. 518-532, 2016.

[16] N. ETSI, «Network Functions Virtualization-Introductory White Paper», en *SDN and OpenFlow World Congress*, 2012.

[17] G. ETSI, «Network functions virtualisation (nfv): Architectural framework», *ETsI Gs NFV*, vol. 2, n.º 2, pág. V1, 2013.

[18] Y. Li y M. Chen, «Software-Defined Network Function Virtualization: A Survey», *IEEE Access*, vol. 3, págs. 2542-2553, 2015.

[19] R. Minerva, A. Biru y D. Rotondi, «Towards a definition of the Internet of Things (IoT)», *IEEE Internet Initiative*, vol. 1, n.º 1, págs. 1-86, 2015.

[20] D. Navani, S. Jain y M. S. Nehra, «The Internet of Things (IoT): A Study of Architectural Elements», en *2017 13th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*, 2017, págs. 473-478.

[21] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu y D. Qiu, «Security of the Internet of Things: perspectives and challenges», *Wireless Networks*, vol. 20, n.º 8, págs. 2481-2501, 2014.

[22] P. Raj y A. C. Raman, «The Internet of Things: Enabling technologies, platforms, and use cases», en. CRC Press, 2017, cap. 3, págs. 61-64.

[23] K. Kalkan y S. Zeadally, «Securing internet of things with software defined networking», *IEEE Communications Magazine*, vol. 56, n.º 9, págs. 186-192, 2017.

[24] P. Martinez-Julia, A. F. Skarmeta y col., «Empowering the internet of things with software defined networking», *White Paper, IoT6-FP7 European research project*, 2014.

[25] B. Pfaff y B. Davie, «The open vswitch database management protocol», *IETF RFC 7047*, 2013.

[26] D. Bansal, S. Bailey, T. Dietz, C. Moberg, J. Quittek, A. Ramaiah y A. Shaikh, «OF-CONFIG 1.2, OpenFlow Management and Configuration Protocol», ONF TS-016, inf. téc., 2014.

[27] Cisco, *OpFlex: An Open Policy Protocol White Paper*, ene. de 2019. dirección: `https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-731302.html`.

[28] A. Stancu, A. Avram, M. Skorupski, A. Vulpe y S. Halunga, «Enabling SDN application development using a NETCONF mediator layer simulator», en *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2017, págs. 658-663.

[29] W. Stallings, «Software-Defined Networks and OpenFlow-The Internet Protocol Journal, Volume 16, No. 1», *vol*, vol. 16, pág. 11, 2016.

[30] B. Pfaff, B. Lantz, B. Heller y col., «Openflow switch specification, version 1.3.0», *Open Networking Foundation*, jun. de 2012.

[31] Z. Latif, K. Sharif, F. Li, M. M. Karim, S. Biswas e Y. Wang, «A comprehensive survey of interface protocols for software defined networks», *Journal of Network and Computer Applications*, feb. de 2019.

[32] L. F. Projects, *ODL Home*, mayo de 2020. dirección: `https://www.opendaylight.org/`.

[33] ——, *ODL Lithium Overview*, sep. de 2017. dirección: https://www.opendaylight.org/what-we-do/current-release/lithium.

[34] ——, *Platform Overview*, mar. de 2019. dirección: https://www.opendaylight.org/what-we-do/odl-platform-overview.

[35] D. Pemberton, A. Linton y S. Russell, «RYU OpenFlow Controller», University of Oregon, inf. téc., 2014.

[36] D. Inc, *Docker Overview*, jun. de 2020. dirección: https://docs.docker.com/get-started/overview/.

[37] G. Bhatia, A. Choudhary y V. Gupta, «The road to Docker: a survey», *International Journal of Advanced Research in Computer Science*, vol. 8, n.º 8, 2017.

[38] Gns3, *The Gns3 marketplace*, jun. de 2020. dirección: https://www.gns3.com/marketplace/appliances.

[39] ULPGC, *Boletín Oficial de la Universidad de Las Palmas de Gran Canaria*, jun. de 2019. dirección: https://www.ulpgc.es/sites/default/files/ArchivosULPGC/boulpgc/BOULPGC/boulpgc_junio_2019_3_junio.pdf.

[40] COITT, *Derechos de visado 2018 COITT*, 2018. dirección: https://www.coit.es/sites/default/files/151123_tarifas_2018_visado.pdf.

[41] M. Team, *Download/Get Started With Mininet*, 2018. dirección: http://mininet.org/download/.

[42] ——, *Mininet releases*, 2019. dirección: https://github.com/mininet/mininet/releases.

[43] G. Team, *Gns3 version 2.2.10*, 2020. dirección: https://www.gns3.com/software.

[44] OpenDayLight, *ODL Litihum*, 2015. dirección: https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/integration/distribution-karaf/0.3.0-Lithium/.

[45] Cisco, *CiscoDevNet Openflow App*, 2016. dirección: https://github.com/CiscoDevNet/OpenDaylight-Openflow-App.

[46] R. S. F. Community, *Build SDN Agilely*, 2017. dirección: https://ryu-sdn.org/.

Parte II

PRESUPUESTO

Presupuesto

P.1. Introducción

Para el cálculo del presupuesto se ha dividido el capítulo en los siguientes puntos:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material.
  - Amortización del material *hardware*.
  - Amortización del material *software*.
- Redacción del documento.
- Derechos de visado del COITT.
- Gastos de tramitación y envío.
- Material fungible.

Finalmente, tras analizar cada uno de estos puntos se calcula el coste final del proyecto incluyendo impuestos.

P.2. Trabajo tarifado por tiempo empleado

El trabajo tarifado por tiempo empleado es el coste de la mano de obra asociada al trabajo de un graduado en ingeniería de telecomunicaciones. Para el cálculo se ha tenido en cuenta que el proyecto se desarrolla en la Universidad de Las Palmas de Gran Canaria, por lo tanto, la tarifa que se aplica es la correspondiente al personal técnico (grado) según la tabla de clasificación y retribución del personal contratado con cargo a proyectos, convenios y contratos correspondiente al BOULPGC del 3 de junio de

2019 [39]. Al mismo tiempo, también se ha tenido en cuenta la duración del proyecto (4 meses) y el tiempo empleado diariamente (4 horas). El resultado final se encuentra en la Tabla P.2.

Personal	**Coste total mensual**	**Tiempo**	**Total**
Ingeniero técnico	711.90 €	4 meses	2847.60 €

Tabla P.2: Trabajo tarifado por tiempo empleado

El coste final del trabajo tarifado por tiempo empleado es de DOS MIL OCHOCIENTOS CUARENTA Y SIETE EUROS Y SESENTA CÉNTIMOS.

P.3. Amortización del inmovilizado material

Para la amortización del inmovilizado material se tienen en cuenta el conjunto de recursos *hardware* y *software* empleados en el desarrollo del presente T.F.G.

El coste de la amortización se calcula para un periodo de 3 años y el sistema utilizado es de carácter lineal. Esto da como resultado que el inmovilizado material se deprecie uniformemente a lo largo de su vida útil.

Para el cálculo de la cuota de amortización anual se tiene en cuenta que el valor residual (valor teórico que supuestamente tendrá el elemento después de su vida útil) es nulo. La fórmula utilizada para su cálculo es P.1:

$$C = \frac{V_{ad} - V_{res}}{N} \tag{P.1}$$

Donde:

- C = Cuota de amortización anual
- V_{ad} = Valor de adquisición
- V_{res} = Valor residual
- N = Número de años

P.3.1. Amortización del material *hardware*

Dado que el trabajo se ha elaborado en un periodo inferior a 3 años (que es el estipulado para la amortización), se realiza una amortización equiparable al periodo de duración del trabajo (4 meses). Los resultados finales se observan en la Tabla P.3.

Descrip.	**Unid.**	**Val. de adq.**	**Tpo. de uso**	**C. anual**	**C. final**
Raspberry Pi 4 Model B - 4GB	3	243.60 €	4 meses	81.20 €	27.07 €
Raspberry Pi 3 Model B - 1GB	1	39,90 €	4 meses	13.30 €	4.43 €
Zodiac FX OpenFlow *Switch*	2	199.98 €	4 meses	66.66 €	22.22 €
MSI PS42 Modern 8RA	1	1021.10 €	4 meses	340.37 €	113.46 €
Lenovo Ideapad 330	1	500.00 €	4 meses	166.67 €	55.56 €
TP-Link TL-MR6400	1	78.99 €	4 meses	26.33 €	8.78 €
Total:					**231.52 €**

Tabla P.3: Precios y costes de la amortización *hardware*

El coste final para el material *hardware* es de DOSCIENTOS TREINTA Y UN EUROS Y CINCUENTA Y DOS CÉNTIMOS.

P.3.2. Amortización del material *software*

Para la realización del presente T.F.G se utilizaron los materiales *software* presentes en la Tabla P.4. Al igual que en la amortización material *hardware*, el trabajo se ha

elaborado en un periodo inferior a 3 años (que es el estipulado para la amortización) y por ello se realiza una amortización equiparable al periodo de duración del trabajo (4 meses). Es importante destacar que las licencias de Microsoft Office y de PyCharm (Jet-Brains) son proporcionadas por la ULPGC, por lo tanto, se referencian con un valor de adquisición de cero euros.

Descrip.	**Unid.**	**Val. de adq.**	**Tpo. de uso**	**C. anual**	**C. final**
Microsoft Office	1	0.00 €	4 meses	0.00 €	0.00 €
PyCharm	1	0.00 €	4 meses	0.00 €	0.00 €
Raspberry Pi OS	4	0.00 €	4 meses	0.00 €	0.00 €
Oracle Virtual Box	1	0.00 €	4 meses	0.00 €	0.00 €
GNS3	1	0.00 €	4 meses	0.00 €	0.00 €
Mininet	1	0.00 €	4 meses	0.00 €	0.00 €
Open Day Light	1	0.00 €	4 meses	0.00 €	0.00 €
RYU	1	0.00 €	4 meses	0.00 €	0.00 €
VMWare Worksation 15.5 Pro	1	221.26 €	4 meses	73.75 €	25.58 €
S.O. Windows 10	2	256.74 €	4 meses	86.58 €	28.52 €
Total:					54.10 €

Tabla P.4: Precios y costes de la amortización *software*

El coste final para el material *software* es de CINCUENTA Y CUATRO EUROS Y DIEZ CÉNTIMOS.

P.4. Redacción del documento

El coste de la redacción del documento se calcula a través de la ecuación P.2, donde P es el presupuesto y C_n el coeficiente de ponderación del presupuesto. Para este caso, C_n vale uno debido a que el coste total del proyecto no supera los 30,050.00 €.

$$R = 0{,}07 \times P \times C_n \tag{P.2}$$

El presupuesto se calcula como la suma del coste de las amortizaciones y el trabajo tarifado por tiempo empleado. El resultado de esta operación se observa en la Tabla P.5.

Concepto	**Coste asociado**
Tarifado por tiempo empleado	2847.60 €
Amortización *hardware*	231.52 €
Amortización *software*	54.10 €
Total:	3133.22 €

Tabla P.5: Suma de las amortizaciones y el tarifado por tiempo empleado

Finalmente, tras hallar el precio del presupuesto, se aplica la ecuación P.2 y se obtiene lo siguiente:

$$R = 0{,}07 \times 3133{,}22 \times 1 = 219{,}33 \tag{P.3}$$

El coste asociado a la redacción del documento es de DOSCIENTOS DIECINUEVE EUROS Y TREINTA Y TRES CÉNTIMOS.

P.5. Derechos de visado del COITT

El COITT establece en [40] que el precio del visado para los proyectos de carácter general se calcula en base a la ecuación P.4:

$$R = 0,006 \times P_1 \times C_1 + 0,003 \times P_2 \times C_2 \tag{P.4}$$

Donde:

- P_1 es el presupuesto general para este proyecto.
- P_2 es el presupuesto de ejecución material que corresponde a la obra civil.
- C_1 es el coeficiente reductor correspondiente a P_1.
- C_2 es el coeficiente reductor correspondiente a P_2.

El valor del presupuesto se obtiene a partir de los costes de los elementos anteriores (el trabajo tarifado por tiempo empleado, la amortización del inmovilizado material y la redacción del documento). La Tabla P.6 muestra este resultado.

Concepto	**Coste asociado**
Tarifado por tiempo empleado	2847.60 €
Amortización *hardware*	231.52 €
Amortización *software*	54.10 €
Redacción del documento	219.33 €
Total:	3352.55 €

Tabla P.6: Valor del presupuesto para el cálculo del visado del COITT

Tras hallar el valor del presupuesto, se calcula el precio del visado del COITT según la expresión P.4. Hay que tener en cuenta que el C_1 vale uno debido a que el coste total del proyecto es inferior a 30,050.00 € y que P_2 vale cero puesto que no se ha requerido obra civil para el desarrollo del trabajo.

$$R = 0,006 \times 3352{,}55 \times 1 = 20{,}12 \tag{P.5}$$

El coste final de los derechos de visado del COITT es de VEINTE EUROS Y DOCE CÉNTIMOS.

P.6. Gastos de tramitación y envío

Los gastos de tramitación y envío están estipulados según [40] en SEIS EUROS por cada documento visado de forma telemática.

P.7. Material fungible

Aparte de los previamente mencionados, se han empleado otros materiales para el desarrollo de este trabajo, tales como: material de oficina, impresión del documento, CDs, etc. Estos abarcan lo que se denomina material fungible y sus costes están representados en la Tabla P.7.

Material	**Coste asociado**
Material de papelería	10.00 €
CD-ROM	5.00 €
Encuadernación	5.00 €
Impresión del documento	30.00 €
Total:	50.00 €

Tabla P.7: Coste total del material fungible

El coste final del material fungible asciende a CINCUENTA EUROS.

P.8. Aplicación de impuestos y coste final

La Tabla P.8 muestra con todos los apartados desglosados el coste total del proyecto.

El presupuesto total teniendo en cuenta los impuestos asciende a TRES MIL SEISCIENTOS SESENTA Y OCHO EUROS Y SESENTA Y OCHO CÉNTIMOS.

Fdo.: José Daniel Padrón Pérez

En Las Palmas de Gran Canaria a 14 de julio de 2020

Concepto	**Coste asociado**
Tarifado por tiempo empleado	2847.60 €
Amortización del inmovilizado material:	
- Amortización *hardware*	231.52 €
- Amortización *software*	54.10 €
Total inmovilizado material:	285.62 €
Redacción del documento	219.33 €
Visado del COITT	20.12 €
Gastos de tramitación y envío	6.00 €
Material fungible:	
- Material de papelería	10.00 €
- CD-ROM	5.00 €
- Encuadernación	5.00 €
- Impresión del documento	30.00 €
Total material fungible:	50 €
Total antes de impuestos:	3428.67 €
Total (IGIC 7 %):	3668.68 €

Tabla P.8: Coste total del proyecto

Parte III

ANEXOS

Anexo A

Instalación de Mininet

Mininet es un software que se ejecuta sobre sistemas Linux y para su instalación es necesario o bien una máquina que tenga este S.O. de manera nativa o un programa que permita su emulación (p. ej: VirtualBox). En nuestro caso, se ha escogido la segunda opción, debido a que la máquina anfitriona es Windows 10. Las especificaciones de esta son:

- Procesador: Intel® Core i7-8550U con 8 núcleos.
- Dispositivo de almacenamiento: 512GB NVMe PCIe SSD.
- Memoria RAM: 16GB DDR4.
- Unidad de procesamiento gráfico: NVIDIA GeForce® MX250, 2GB GDDR5.

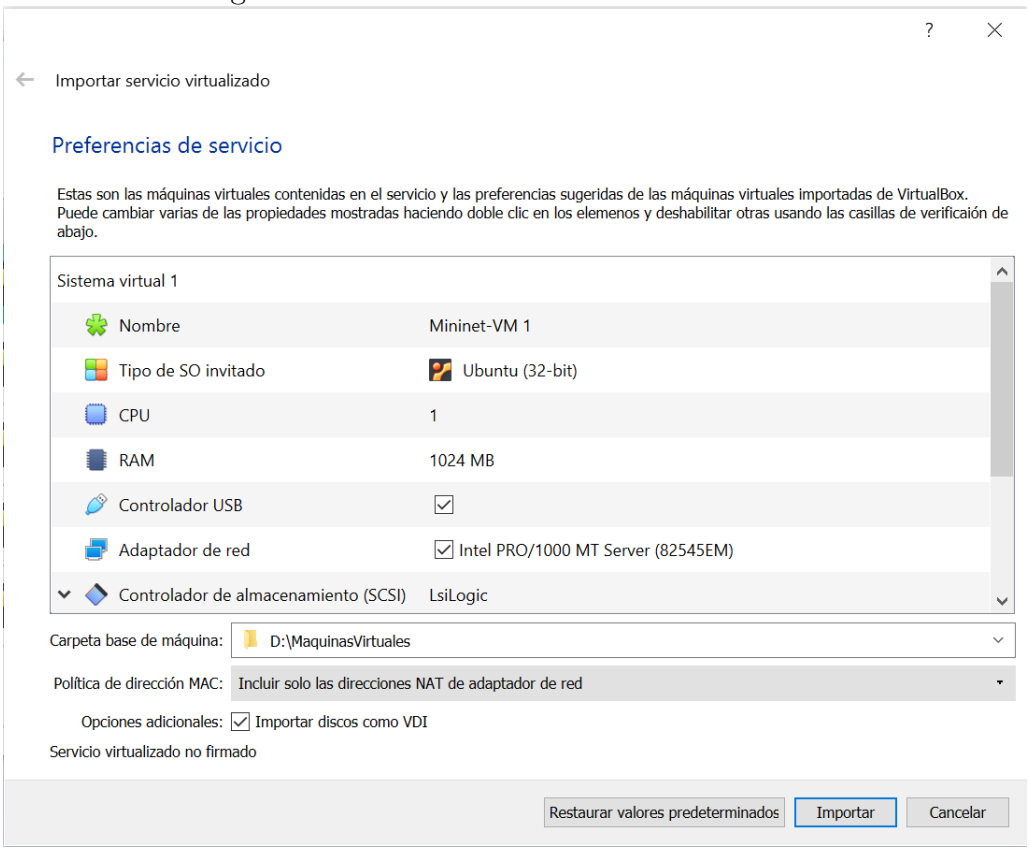
Esto hace que sea totalmente compatible con los requerimientos de VirtualBox y en su defecto, Mininet. Para este, son necesarios como mínimo 1 núcleo de CPU, 1 GB de RAM y 10GB de almacenamiento, pero para nuestro caso se disponen de 2 núcleos de CPU y 2 GB de RAM para un mejor rendimiento.

A continuación, se describen los pasos a seguir para la instalación (se parte la base de que ya se ha instalado VirtualBox).

En primer lugar, se accede la web principal de Mininet, [41]. Ahí se observan varias opciones de instalación, en nuestro caso optaremos por la primera, ya que es la más sencilla para VirtualBox.

En segundo lugar, tras seleccionar la primera opción, se procede a descargar la imagen necesaria para la máquina virtual. Esta se encuentra en su repositorio de GitHub,

Figura A.1: Pantalla de instalación de Mininet



[42] . Una vez ahí, se selecciona la última versión hasta la fecha (v2.2.2) y se descarga el archivo .zip que se llama *mininet-2.2.2-170321-ubuntu-14.04.4-server-i386*. Como se observa ya por el nombre del archivo, Mininet se ejecutará sobre un servidor Ubuntu de versión 14.04.

En tercer lugar, una vez descargado el zip, se extrae y se observan dos archivos, uno .ovf y otro .vmdk. El primer archivo hace referencia a las características de instalación de la máquina virtual de Mininet. El segundo es el disco duro virtual sobre el que va a correr la máquina. Al hacer doble click sobre el .ovf, se abre una ventana en VirtualBox como la mostrada en la Figura A.1, aquí únicamente será necesario darle a importar y ya estará la instalación completada.

Finalmente, tras ejecutar la máquina virtual, se requerirán de un usuario y una contraseña para su inicio, ambos son mininet.

Anexo B

Instalación de GNS3

GNS3 es una herramienta *software* gratuita que permite la creación, diseño y prueba de distintos escenarios de red. Disponible para todos los sistemas operativos, se apoya en el uso de la tecnología Docker para simular servicios y aplicaciones de dispositivos presentes en la red. Según se especifica en su web [43], los requerimientos mínimos y los recomendados son lo que se encuentran en la Tabla B.1.

	Req. mínimos	**Req. recomendados**
Sistema Operativo	Win.7 (64 bit) o más, Mavericks (10.9) o más, cualquier Linux Distro - Debian/Ubuntu	Win.7 (64 bit) o más, Mavericks (10.9) o más, cualquier Linux Distro - Debian/Ubuntu
Procesador	2 núcleos o más. AMD-V / RVI Series o Intel VT-X / EPT - extensiones para la virtualización presentes y habilitadas en la BIOS	4 núcleos o más. AMD-V / RVI Series o Intel VT-X / EPT - extensiones para la virtualización presentes y habilitadas en la BIOS
Memoria RAM	4 GB	8 GB
Almacenamiento	1 GB (Instalación Windows menos de 200MB)	SSD - 35 GB

Tabla B.1: Requerimientos para GNS3 [43]

Teniendo esto en cuenta, para descargarse el programa en necesario registrarse en la web. Una vez registrados, se selecciona la descarga para Windows. Al mismo tiempo, se descarga también la máquina virtual de GNS3 para VMWare Worksation. Este es el programa que se usará para la máquina virtual dentro del marco de su licencia gratuita de 30 días. También es posible utilizar la opción de VirtualBox, pero esta opción solo es recomendada en caso de no poder realizar la primera, ya que el rendimiento de GNS3 disminuye en consideración.

Una vez descargados los dos archivos, se procede a la instalación de GNS3. Se siguen los pasos por defecto marcados por el instalador y se consigue instalar sin mayor dificultad.

B.1. Instalación de la máquina virtual

La máquina virtual es necesaria debido a que la instalación se va a realizar sobre un sistema operativo Windows, además la página recomienda instalarla sea cual sea el sistema para un mejor rendimiento.

Para instalar la máquina virtual en los pasos a seguir son los siguientes (en el caso de VMWare Workstation):

1. Extraer el zip donde se encuentra el archivo .ova

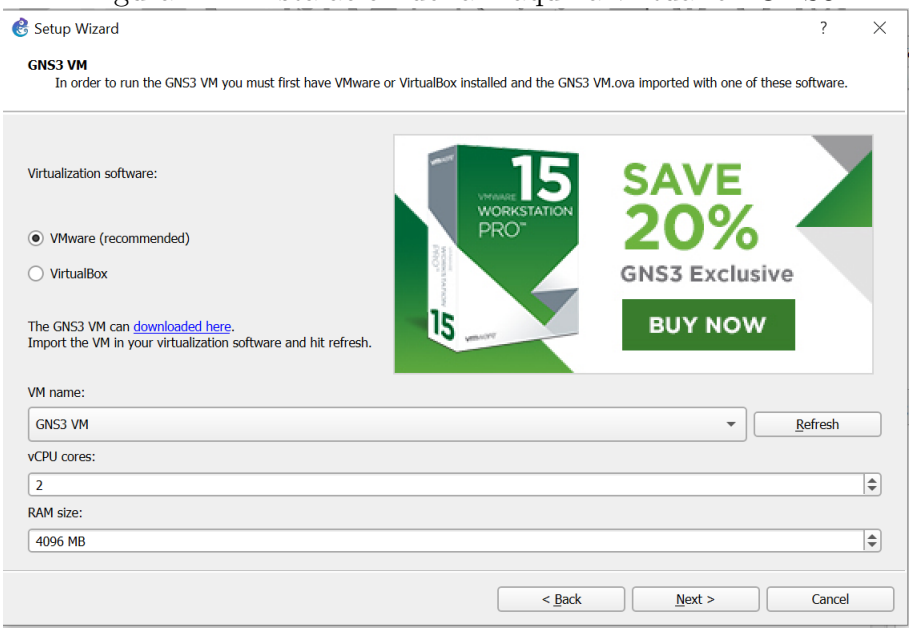
2. Abirir el archivo .ova con VMWare Workstation. La máquina virtual se importa automáticamente.

3. Abrir GNS3 y se muestra una ventana de configuración como la de la Figura B.1. En este caso al tener instalado VMWare Workstation la reconoce automáticamente.

4. Finalizar la configuración clickando en next.

En el caso de querer usar VirtualBox, la instalación es un poco más compleja. Esto se debe a que hay que instalar un archivo llamado vboxmanage.exe y configurar dentro de las preferencias de GNS3, en el apartado de VirtualBox, la ruta de este archivo. Al mismo tiempo, sobra mencionar que también hay que importar el archivo .ova en VirtualBox en el caso de querer instalarla ahí.

Figura B.1: Instalación de la máquina virtual en GNS3



B.2. Instalación de las *appliances*

Una *appliance* viene a ser un dispositivo que no viene integrado en GNS3 por defecto y que se necesita descargarlo para tenerlo. En nuestro caso, las *appliances* utilizadas son la Ubuntu Docker Guest y la OvSwitch. La instalación de ambas es de la misma manera, y en este caso se escoge el ejemplo de instalación de la primera.

Para la instalación del dispositivo Ubuntu los pasos a seguir son los siguientes:

1. Descargar la *appliance* de GNS3 en el *Marketplace* [38].

2. Importar la *appliance* en el programa, para ello se accede a *File* y dentro de ahí a *Import Appliance*.

3. Se selecciona la *appliance* descargada. Esta no es más que un fichero de texto en un formato específico (ver Figura B.2) para que GNS3 realice las instrucciones necesarias para descargar la imagen Docker.

4. Creado el dispositivo, para integrarlo en la red no basta más que arrastrarlo y soltarlo. En la Figura B.3, se observa que se descarga la imagen (ya que no se encuentra descargada previamente) para crear el contenedor Docker de Ubuntu.

Figura B.2: Contenido del archivo .gns3a relativo al dispositivo Ubuntu

ubuntu-docker.gns3a

```
{
    "name": "Ubuntu Docker Guest",
    "category": "guest",
    "description": "Ubuntu is a Debian-based Linux operating system, w
    "vendor_name": "Canonical",
    "vendor_url": "http://www.ubuntu.com",
    "product_name": "Ubuntu",
    "registry_version": 3,
    "status": "stable",
    "maintainer": "GNS3 Team",
    "maintainer_email": "developers@gns3.net",
    "symbol": "linux_guest.svg",
    "docker": {
        "adapters": 1,
        "image": "gns3/ubuntu:xenial",
        "console_type": "telnet"
    }
}
```

Figura B.3: Descarga de la imagen Docker por primera vez para la máquina Ubuntu

Console

```
Pulling image gns3/ubuntu:xenial:e8db7bf7c39f: [==========================================>       ]   426B/513B
Pulling image gns3/ubuntu:xenial:e8db7bf7c39f: [==================================================>]   513B/513B
Pulling image gns3/ubuntu:xenial:f8b845f45a87: [==========================>                ]   426B/820B
Pulling image gns3/ubuntu:xenial:f8b845f45a87: [==================================================>]   820B/820B
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [>                                  ] 507.9kB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [=>                                 ] 1.522MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [==>                                ] 2.55MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [===>                               ] 3.07MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [====>                              ] 4.107MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [======>                            ] 5.63MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [=======>                           ] 6.138MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [=========>                         ] 8.174MB/50.43MB
Pulling image gns3/ubuntu:xenial:9654c40e9079: [================================>            ]   424B/677B
Pulling image gns3/ubuntu:xenial:9654c40e9079: [==================================================>]   677B/677B
Pulling image gns3/ubuntu:xenial:6d9ef359eaaa: [==================================================>]   161B/161B
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [==========>                        ] 9.202MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [===========>                       ] 10.24MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [============>                      ] 11.26MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [============>                      ] 11.77MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [=============>                     ] 12.27MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [==============>                    ] 13.29MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [================>                  ] 14.31MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [================>                  ] 14.82MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [=================>                 ] 15.34MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [==================>                ] 16.37MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [===================>               ] 17.91MB/50.43MB
Pulling image gns3/ubuntu:xenial:d54efb8db41d: [====================>              ] 19.45MB/50.43MB
```

Anexo C

Instalación de los controladores

C.1. Instalación de OpenDayLight

Para la instalación de ODL se requiere de previamente de un S.O Linux, integrado en una máquina virtual o de forma nativa. En nuestro caso, se optará por la máquina virtual, con un sistema operativo Ubuntu Desktop v18.04. Es importante mencionar que en un principio la instalación se realizó para la versión servidor de Ubuntu, pero al no poder ejecutar Wireshark se optó por utilizar la versión de escritorio.

La versión de ODL que se va a instalar es la 0.3.0 o Lithium, pese a haber versiones posteriores, al dar problemas a la hora de la ejecución se optó por esta, ya que es la versión más alta que no los da. Los recursos usados en su ejecución son los de la máquina virtual: 4GB de RAM, 2 núcleos de CPU y 10GB de almacenamiento. Con esta configuración el controlador se ejecuta sin ningún problema. A continuación se describen los pasos a seguir:

En primer lugar, se instala la versión 1.8 de Java, necesaria ya que ODL se ejecuta sobre una JVM que corre sobre esta versión. El comando a ejecutar en el terminal para ello es:

```
$ sudo apt install openjdk-8-jdk
```

En segundo lugar, una vez instalado el JDK, se añade al PATH de las variables del entorno para que desde el terminal se pueda ejecutar. Esto se realiza a través de:

```
$ export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64
```

```
$ export PATH=$JAVA_HOME/bin:$PATH
$ sudo source ~/.bashrc
```

En tercer lugar, accediendo a la página de versiones de ODL, dentro de la versión de Lithium [44], se copia el link de descarga del archivo .tar y se descarga desde el terminal con la función wget.

```
$ wget https://nexus.opendaylight.org/content/repositories/public
/org/opendaylight/integration/distribution-karaf/0.3.0-Lithium
/distribution-karaf-0.3.0-Lithium.tar.gz
```

Finalmente, se extrae el archivo tar descargado, y accediendo a la carpeta extraída con el comando ./bin/karaf se ejecuta ODL.

```
$ sudo tar -xzf distribution-karaf-0.3.0-Lithium.tar.gz
$ cd distribution-karaf-0.3.0-Lithium
$ ./bin/karaf
```

C.1.1. Instalación de OFM

OpenFlow Manager es una aplicación de Cisco que se encuentra en GitHub [45] y que se ejecuta utilizando la herramienta Grunt de Node.js. A continuación, se describen los pasos a realizar para la instalación:

1. Instalación de git.

   ```
   $ sudo apt install git
   ```

2. Instalación de Node.js ya que es la plataforma sobre la que se va ejecutar la aplicación.

   ```
   $ sudo apt install nodejs
   ```

3. Clonación del repositorio de Git de la aplicación.

   ```
   $ git clone https://github.com/CiscoDevNet/OpenDaylight-
   Openflow-App.git
   ```

4. Configuración del archivo env.module.js, responsable de configurar toda la gestión de la aplicación.

```
$ sed -i 's/localhost/direccionIP/g' ./OpenDaylight-
Openflow-App/ofm/src/common/config/env.module.js
```

5. Instalación del sistema de gestión de paquetes de Node.js (npm).

```
$ sudo apt install npm
```

6. Instalación de la herramienta Grunt, necesaria para la ejecución de OFM.

```
$ npm install -g grunt-cli
```

Realizados todos estos pasos, accediendo a la carpeta de /OpenDaylight-Openflow-App y ejecutando grunt, el servicio comienza a ejecutarse.

```
$ cd OpenDaylight-Openflow-App/
$ grunt
```

C.2. Instalación de RYU

Para la instalación de RYU se requiere de previamente de un S.O Linux, integrado en una máquina virtual o de forma nativa. En nuestro caso, se optará por la máquina virtual, con un sistema operativo Ubuntu Desktop v18.04. Los recursos usados en su ejecución son los de la máquina virtual: 4GB de RAM, 2 núcleos de CPU y 10GB de almacenamiento.

Según la página web de RYU [46] se pueden seguir dos formas para la instalación: utilizar la herramienta pip de Python (la manera más sencilla y rápida) o clonar directamente el repositorio y configurar todo de manera manual (usada cuando falla pip). Para nuestro caso se usará la segunda, debido a que la primera falla a la hora de ejecutar el script. Los pasos para la instalación son los siguientes:

Por un lado, se han de instalar Python (en muchas distribuciones viene ya preinstalado) y sus herramientas. Las herramientas que se van a descargar son: pip, dev, eventlet, routes, webob y paramiko (las dos últimas no son estrictamente necesarias para RYU). Los comandos a ejecutar son:

```
$ sudo apt upgrade #Actualizacion de las librerias
$ sudo apt install git #Instalacion de git
$ sudo apt install python-pip
$ sudo apt install python-dev
$ sudo apt install python-eventlet
$ sudo apt install python-routes
$ sudo apt install python-webob
$ sudo apt install python-paramiko
```

Por otro lado, una vez instaladas todas las herramientas de Python el siguiente paso es instalar RYU mediante git. Para ello se ejecutan las siguientes línea en el terminal:

```
$ sudo git clone git://github.com/osrg/ryu.git
$ cd ryu
$ sudo python setup.py install
$ ryu-manager --version #Para comprobar la correcta instalacion
```

En caso de que al ejecutar el comando de ryu-manager y producirse un error, lo que hay que hacer es lo siguiente: estando en el directorio ryu, se descargan los paquetes requeridos y se reinstala de nuevo el controlador, para ello se ejecutan las siguientes líneas:

```
$ sudo pip install -r tools/pip-requires
$ sudo python setup.py install
```

Finalmente, tras instalar RYU, todos los ejemplos de RYU se encuentran en el directorio /usr/local/lib/python2.7/dist-packages/ryu/app/.

Anexo D

Código Python para el escenario de red real

D.1. *Script* para la automatización del controlador

```
import serial  # For treating with serial connections
import os  # For executing a shell command
import platform  # For getting the operating system name
import time  # For setting the interval of doing pings

param = '-n' if platform.system().lower() == 'windows' else '-c'


def set_controller(controller_ip, serial_1, serial_2):
    """Open the serial ports and
    write the commands to set the controller
    in the Zodiac Fx SDN switch
     """
    try:
        # Open ports serial 1 AND serial 2
        serial_port_1 = serial.Serial(serial_1, 9600)
        serial_port_2 = serial.Serial(serial_2, 9600)

        # Automation for setting de IP of the controller
        serial_port_1.write(b"config \r\n")
        serial_port_2.write(b"config \r\n")
        # Variable to concatenate the IP of the controller
```

```
            order = "set of-controller " + controller_ip + " \r\n"
            serial_port_1.write(str.encode(order))
            serial_port_1.write(b"save \r\n")
            serial_port_1.write(b"exit \r\n")
            serial_port_2.write(str.encode(order))
            serial_port_2.write(b"save \r\n")
            serial_port_2.write(b"exit \r\n")
            print("--------------------------------")
            print("Successfully set controller to IP: " + controller_ip)
            print("--------------------------------")
            serial_port_1.close()
            serial_port_2.close()
        except serial.SerialException as e:
            # There is no serial port connected
            print("--------------------------------")
            print("SERIAL PORT EXCEPTION: There is a switch that is not connected")
            print("--------------------------------")
            exit()
            return None


def is_controller_up(controller):
    """Checks if the controller is up
    by pinging it. If the response is ok returns true
    if its not ok return false
    """
    print("--------------------------------")
    print("CONNECTION AUTOMATION. PING PROCESS")
    print("--------------------------------")
    response = os.system("ping " + param + " 1 " + controller)
    # Check the response
    if response == 0:  # Ping is successful
        print("--------------------------------")
        print("Controller with IP: ", controller, 'is up!')
        return True
    else:
        print("--------------------------------")
        print("Controller with IP: ", controller, 'is down!')
        return False
```

```
def ping_automation_controller(controller_1, controller_2, serial_1, serial_2):
    """Set the controller if it is up, by default if
    both connections are ok the controller set is 10.0.0.20"""

    if is_controller_up(controller_1):
        set_controller(controller_1, serial_1, serial_2)
    else:
        set_controller(controller_2, serial_1, serial_2)


def main():
    """Main function. Automates the process of pinging
    and sleep for 5 seconds before doing the previous stuff again"""

    # Console to enter the values needed

    print("#####################################")
    print("SDN CONTROLLER AUTOMATION SCRIPT")
    print("#####################################")

    # Set serial 1
    serial_1 = input("Enter serial port n 1: ")
    # Requests for data until user inputs something
    while not serial_1:
        serial_1 = input("Enter serial port n 1: ")
    print(serial_1.upper().strip() + " set!")

    print("--------------------------------")

    # Set serial 2
    serial_2 = input("Enter serial port n 2: ")
    # Requests for data until user inputs something
    while not serial_2:
        serial_2 = input("Enter serial port n 2: ")
    print(serial_2.upper().strip() + " set!")

    print("--------------------------------")
```

```
    # Set controller 1 IP
    controller_1 = input("Enter controller n 1 IP: ")
    # Requests for data until user inputs something
    while not controller_1:
        controller_1 = input("Enter controller n 1 IP: ")
    print("Controller 1 IP set!")
    print("IP: " + controller_1.strip())

    print("--------------------------------")

    # Set controller 2 IP
    controller_2 = input("Enter controller n 2 IP: ")
    # Requests for data until user inputs something
    while not controller_2:
        controller_2 = input("Enter controller n 2 IP: ")
    print("Controller 2 IP set!")
    print("IP: " + controller_2.strip())

    # Infinite while loop
    while True:
        ping_automation_controller(controller_1.strip(), controller_2.strip(),
                                   serial_1.strip().upper(),
                                   serial_2.strip().upper())
        print("#####################################")
        print("#####################################")
        time.sleep(5)


main()
```

D.2. *Script* para el servidor

```
import socket


def set_up_connection(server_ip, port_number, sim_connections):
    server = set_up_server(server_ip, port_number)
```

```
    # Listen to x simultaneous connections
    server.listen(int(sim_connections))
    print("Server listening on port:" + str(port_number))

    # Create a client socket connection and accept it
    cli, addr = server.accept()
    # Print the connection is established
    print("Connected to: " + addr[0] + ":" + str(addr[1]))

    return cli, server


def set_up_server(server_ip, port_number):

    # Initialize an object to work with the socket
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print("Socket created")

    try:
        server.bind((server_ip, int(port_number)))  # Server IP and port to listen
    except socket.error as msg:
        print(msg)

    print("Socket bind completed")

    return server


def transfer_data(cli, server):
    """Function that is responsible of dealing
    with the data from the IoT device. Processes
    and prints on console the humidity
    and temperature"""
    while True:
        data = cli.recv(1024)
        data = data.decode('utf-8')
        data_message = data.split(":")
        temperature = data_message[0]
```

```
        humidity = data_message[1]
        print("Temp: " + temperature + " C" + " Humidity: " + humidity + " %")

    # Unreachable code with this algorithm.
    # Close the client connection
    cli.close()
    # Close the server connection
    server.close()


def main():
    """Main function that executes the whole file.
    Creates a console to interact with the user
    in order to create a dynamic program"""

    print("#####################################")
    print("#####################################")
    print("SERVER OF IOT HUMIDITY AND TEMPERATURE SENSOR")
    print("#####################################")
    print("#####################################")
    print("")

    # Set server IP
    server_ip = input("Enter server IP: ")
    # Requests for data until user inputs something
    while not server_ip:
        server_ip = input("Enter server IP: ")
    print(server_ip.upper().strip() + " set!")

    print("--------------------------------")

    # Set port number
    port = input("Enter port number: ")
    # Requests for data until user inputs something
    while not port:
        port = input("Enter port number: ")
    print("Port " + port.upper().strip() + " set!")

    print("--------------------------------")
```

```
    # Set number of sim. conn.
    sim_connections = input("Enter number of simultaneous connections: ")
    # Requests for data until user inputs something
    while not sim_connections:
        sim_connections = input("Enter number of simultaneous connections: ")
    print(sim_connections.upper().strip() + " set!")

    print("--------------------------------")

    cli, server = set_up_connection(server_ip, port, sim_connections)
    # Call the function to set the connection
    transfer_data(cli, server)


main()
```

D.3. *Script* para el cliente IoT

```
import time
import RPi.GPIO as GPIO
import Adafruit_DHT
import socket

DHTSensor = Adafruit_DHT.DHT11
GPIO_PIN = 17


def read_sensor_data():
    humidity, temperature = Adafruit_DHT.read_retry(DHTSensor, GPIO_PIN)

    # Check if the values are not empty
    if humidity is not None and temperature is not None:
        print('Reading sensor data...')
        return humidity, temperature
    else:
        print('No sensor data available')
        return "no data", "no data"
```

```
def set_up_connection(server_ip, port_number):
    # Create a socket object
    client_socket = socket.socket()

    # Establishing the connection with the server
    client_socket.connect((server_ip, int(port_number)))

    return client_socket


def send_msg(client_socket):
    while True:
        humidity, temperature = read_sensor_data()

        # Check the data retrieved by the sensor
        print('Temp: {0:0.1f} C  Humidity: {1:0.1f} %
        .format(temperature, humidity))

        humid = str(humidity)
        temp = str(temperature)

        message = temp + ":" + humid

        client_socket.send(str.encode(message))

        time.sleep(2)  # Sends the data every 2 seconds


def main():

    print("#####################################")
    print("#####################################")
    print("IOT HUMIDITY AND TEMPERATURE SENSOR DEVICE")
    print("#####################################")
    print("#####################################")
    print("")
```

```
    # Set server IP
    server_ip = input("Enter server IP: ")
    # Requests for data until user inputs something
    while not server_ip:
        server_ip = input("Enter server IP: ")
    print(server_ip.upper().strip() + " set!")

    print("--------------------------------")

    # Set port number
    port = input("Enter port number: ")
    # Requests for data until user inputs something
    while not port:
        port = input("Enter port number: ")
    print("Port " + port.upper().strip() + " set!")

    print("--------------------------------")

    send_msg(set_up_connection(server_ip, port))

    # print('Temp: {0:0.1f} C  Humidity: {1:0.1f} %'.format(temperature, humidity))


main()
```