UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Máster

Desarrollo de APIs para escenarios SDN-NFV

Autor: César Francisco San Nicolás Martínez
Director: Pablo Pavón Mariño
Codirector: Francisco Javier Moreno Muro

Julio 2019

Autor:	César Francisco San Nicolás Martínez
E-mail del autor:	cesarfsannicolasmartinez@gmail.com
Director:	Pablo Pavón Mariño
E-mail del director:	pablo.pavon@upct.es
Codirector:	Francisco Javier Moreno Muro
E-mail de codirector:	javier.moreno@upct.es

Título del TFM:	Desarrollo de APIs para escenarios SDN-NFV

Resumen:
El paradigma SDN se ha convertido en una de las maneras más eficientes de gestionar las redes actuales, gracias a la automatización de las funciones de operación y gestión de los diferentes dispositivos de red. A su vez, NFV se ha convertido en un recurso valioso para optimizar recursos de una infraestructura IT. Debido a esto, han ido surgiendo herramientas para la aplicación de las técnicas SDN y NFV. El objetivo de este proyecto es desarrollar diversos clientes para controlar remotamente las herramientas ONOS, OSM y OpenStack e integrarlos en un plugin de la herramienta Net2Plan para realizar una prueba de concepto mediante la interconexión de las diferentes herramientas mencionadas anteriormente.

Titulación:	Máster en Ingeniería de Telecomunicación
Departamento:	Tecnologías de la Información y las Comunicaciones

Índice general

1. Introducción **1**
1.1. Motivaciones 1
1.2. Objetivos 2
1.3. Plan de trabajo 2
1.4. Estructura de la memoria 3

2. Estado del arte **5**
2.1. SDN 5
2.1.1. Arquitectura SDN 6
2.1.2. OpenFlow 7
2.2. NFV 8
2.2.1. Arquitectura NFV 9
2.3. Relación entre SDN y NFV 10
2.4. Herramientas Open-Source en redes de telecomunicación 11

3. Herramientas utilizadas **13**
3.1. Net2Plan 13
3.2. Mininet 14
3.3. ONOS 15
3.3.1. OpenAPI y Swagger 17
3.4. OSM 17
3.4.1. OSMClient 19
3.5. OpenStack 19
3.5.1. OpenStack4Java 20

4. Desarrollo de Aplicaciones **23**
4.1. J-OSM Client 23
4.2. J-ONOS Client 25
4.3. J-OpenStack Client 27
4.4. Net2Plan: NFV Management Plugin 29

5. Prueba de concepto **31**
5.1. Contexto 31
5.2. Arquitectura 31
5.3. Funcionamiento 33

6. Conclusiones **39**
6.1. Fortalezas 39
6.2. Análisis de resultados 39

6.3. Líneas de trabajo futuro y mejoras 40

A. Script Mininet Red Transporte **41**

Índice de figuras

2.1. Arquitectura SDN. Fuente:[1] . . . 7
2.2. Arquitectura NFV. Fuente:[5] . . . 9
2.3. Ejemplo de *Service Chaining*. Fuente:[4] . . . 10

3.1. Plugin *Offline network design and online network simulation* de Net2Plan . . 14
3.2. Arquitectura de ONOS. Fuente: [13] . . . 16
3.3. Interfaz Gráfica de ONOS. Fuente: [14] . . . 16
3.4. Arquitectura de OSM. Fuente: [19] . . . 18
3.5. Ejemplo de uso de OSMClient. Fuente:[22] . . . 19
3.6. Arquitectura de OpenStack. Fuente:[23] . . . 20
3.7. Ejemplo de uso de OpenStack4Java. Fuente:[24] . . . 21

4.1. Estructura de clases de J-OSMClient . . . 24
4.2. Estructura de clases de J-ONOSClient . . . 26
4.3. Estructura de clases de J-OpenStackClient . . . 27
4.4. Autenticación mediante OSClientV3. Fuente:[24] . . . 28
4.5. Interfaz gráfica del Plugin NFV-Management . . . 29

5.1. Arquitectura de la Prueba de Concepto. Fuente:[29] . . . 32
5.2. *Testbed* para la Prueba de Concepto. Fuente:[29] . . . 33
5.3. Interfaz gráfica del Plugin al inicio . . . 34
5.4. Interfaz gráfica del Plugin con la ruta establecida . . . 35
5.5. Interfaz gráfica de Open Source MANO (OSM) con los Virtual Network Functions (VNFs) instanciados . . . 35
5.6. Interfaz gráfica de Open Network Operative System (ONOS) con las reglas de flujo establecidas . . . 36
5.7. Prueba de conectividad en la Graphical User Interface (GUI) de ONOS . . 36

Lista de acrónimos

API Application Programming Interface
AWS Amazon Web Services
CapEx Capital Expenditures
CLI Command Line Interface
CPU Central Processing Unit
ECOC European Conference on Optical Communication
ETSI European Telecommunications Standards Institute
GUI Graphical User Interface
HD Hard Disk
HTTP HyperText Transfer Protocol
ICT Information and Communication Technologies
IP Internet Protocol
JSON JavaScript Object Notation
LA-SCCE Latency-Aware Service Chain Computation Element
MAC Media Access Control
MANO MANager and Orchestrator
NAT Network Address Translation
NBI NorthBound Interface
NIC Network Interface Card
NS Network Service
NSC Network Service Chaining
NSD Network Service Descriptor
NFV Network Function Virtualization
NFVI NFV Infrastructure
NFV-O NFV Orchestrator
OLT Optical Line Termination

ONOS Open Network Operative System
OpEx Operational Expenditures
OSM Open Source MANO
OSS Operative Support System
OTN Optical Transport Network
OTU Optical Transport Unit
PC Personal Computer
RAM Random Access Memory
REST REpresentational State Transfer
ROADM Reconfigurable Optical Add-Drop Multiplexer
SBI SouthBound Interface
SDN Software Defined Networking
SNMP Simple Network Management Protocol
SSH Secure SHell
TAPI Transport API
URL Uniform Resource Locator
VDU Virtual Deployment Unit
VIM Virtual Infrastructure Manager
VLAN Virtual Local Area Network
VLD Virtual Link Descriptor
VNF Virtual Network Function
VNFD Virtual Network Function Descriptor
WDM Wavelength Division Multiplexing
YANG Yet Another Next Generation

Capítulo 1

Introducción

El paradigma Software Defined Networking (SDN) se ha convertido en una de las maneras más eficientes de gestionar las redes de telecomunicación, permitiendo a los administradores de red una gestión y configuración a alto nivel. Todo esto es gracias a la naturaleza de SDN, cuya principal premisa es desacoplar totalmente el plano de control del plano de datos.

El paradigma Network Function Virtualization (NFV) se ha convertido en una de las tecnologías más valiosas para optimizar recursos de una infraestructura Information and Communication Technologies (ICT). Al igual que SDN, NFV permite a los administradores de red gestionar los diferentes recursos a un alto nivel. Gracias a la naturaleza de NFV, que se basa en la idea de virtualizar cualquier función de red, como puede ser un *firewall* o un dispositivo Network Address Translation (NAT), sustituyendo un dispositivo físico por una máquina virtual que realice su misma función de una manera más eficiente.

Debido a esto, han ido surgiendo herramientas para la aplicación de las técnicas SDN y NFV para automatizar y optimizar las redes de telecomunicación.

1.1. Motivaciones

Este proyecto viene motivado por la evolución de las redes de telecomunicación en los últimos años. El concepto de red de telecomunicación se inició como algo puramente físico, constituido exclusivamente por dispositivos *hardware*. Actualmente, una red de telecomunicación está compuesta por dispositivos físicos sustentados por aplicaciones *software* que son utilizadas por los administradores de red para realizar tareas a alto nivel de forma automatizada.

Los paradigmas SDN y NFV han posibilitado esta evolución, permitiendo a los administradores de redes el poder configurar y gestionar una red a un gran alto nivel, así como el sustituir ciertos dispositivos físicos por máquinas virtuales que realicen su misma función dentro de un entorno de red, pero de manera más eficiente.

Para realizar todas estas tareas a alto nivel, son necesarias diferentes herramientas *software* destinadas a tal fin. Cada una de estas herramientas tiene su propia interfaz de comunicación para poder hacer uso de ellas. Un problema que surge es la particularidad de las interfaces de comunicación de las distintas herramientas, ya que cada una de ellas tiene su propia sintáxis.

Así mismo, no existe una entidad central que permita gestionar las interacciones entre las diferentes herramientas de forma óptima. Por esto último surge la idea de desarrollar diferentes clientes **open-source** para interactuar con las distintas herramientas de una forma sencilla y transparente para el usuario, y convertir a Net2Plan en esa entidad central que se pueda comunicar con las diferentes herramientas y gestionar las interacciones entre ellas, para proporcionar optimización al sistema gracias a una visión más amplia de la red.

1.2. Objetivos

El objetivo principal de este trabajo es desarrollar diferentes Application Programming Interfaces (APIs) **open-source** e integrarlas en un plugin de Net2Plan para llevar a cabo una prueba de concepto. Este objetivo se puede desglosar en otros más pequeños:

- Adquisición de conocimientos de las distintas herramientas (ONOS, Mininet, OSM y OpenStack).
- Desarrollo de APIs para entornos SDN (J-ONOS Client).
- Desarrollo de APIs para entornos NFV (J-OSM Client y J-OpenStack Client).
- Integración de todas las APIs en un plugin de Net2Plan para una prueba de concepto.
- Obtención y análisis de resultados.

1.3. Plan de trabajo

Para la consecución de los objetivos marcados, el plan de trabajo del proyecto consta de diferentes fases, cada una de ellas destinada a cumplir un objetivo concreto:

- Estudio y familiarización con ONOS, Mininet, OSM y OpenStack.
- Desarrollo del cliente para interactuar con ONOS (J-ONOS Client).
- Desarrollo del cliente para interactuar con OSM (J-OSM Client).
- Desarrollo del cliente para interactuar con OpenStack (J-OpenStack Client);
- Desarrollo del plugin de Net2Plan integrando los clientes desarrollados.
- Realización la prueba de concepto y obtención de resultados.
- Análisis de resultados y obtención de conclusiones.
- Escritura de la memoria.

1.4. Estructura de la memoria

El resto de la memoria de este Trabajo de Fin de Máster se ha estructurado de la siguiente manera:

- Capítulo 2. Estado del arte

  En este capítulo se lleva a cabo una explicación de como herramientas open-source han cambiado el concepto de red de telecomunicación. Así mismo, se realiza una extensa explicación de los paradigmas SDN y NFV.

- Capítulo 3. Herramientas utilizadas

  En este capítulo se realiza una explicación de todas y cada una de las herramientas y librerías que se han utilizado para llevar a cabo este proyecto.

- Capítulo 4. Desarrollo de APIs

  En este capítulo se realiza una explicación de las APIs desarrolladas para este proyecto (J-OSM Client, J-ONOS Client y J-OpenStack Client), así como del plugin de Net2Plan diseñado para integrar las APIs mencionadas.

- Capítulo 5. Prueba de concepto

  En este capítulo se lleva a cabo una explicación de las diferentes fases de la prueba de concepto, así como de sus resultados finales.

- Capítulo 6. Conclusiones

  En este capítulo se realiza un análisis de los resultados obtenidos. También se establecen futuras líneas de investigación y desarrollo para dar continuidad al proyecto.

Capítulo 2

Estado del arte

En este capítulo se hablará sobre el contexto en el que se enmarca este proyecto, haciendo especial mención a los paradigmas SDN y NFV.

Ambos paradigmas son la base de este Trabajo de Fin de Máster, y por ello se hace una extensa explicación de cada uno de ellos, haciendo mención a sus arquitecturas de trabajo, ya que explican de forma clara y concisa los diferentes elementos que componen dichas arquitecturas y que función desempeñan.

Una vez explicados SDN y NFV, se lleva a cabo una explicación de como han evolucionado las redes de telecomunicación en los últimos años. Esta evolución se debe a la inclusión de herramientas *software*, especialmente las de código abierto, permitiendo una gestión, configuración y optimización mucho más rapida y eficiente de la infraestructura de una red de telecomunicación.

2.1. SDN

SDN es un paradigma que consiste en una nueva forma de configurar y gestionar las redes de telecomunicación. Su principal premisa es la de llevar a cabo un desacoplamiento total del plano de control del plano de datos.[1][2]

El plano de control define como se configura y gestiona la red de telecomunicación, mientras que el plano de datos es el encargado de transferir las órdenes de configuración y gestión a la infraestructura de la red.

SDN pretende cambiar la manera tradicional de configuración de dispositivos usando instrucciones de bajo nivel por una manera novedosa mediante herramientas *software* a un alto nivel.[1]

Dicho paradigma surge principalmente por las limitaciones que tienen las tecnologías de redes actualmente. Las dos principales limitaciones son:

- **Dificultad de escalabilidad:** A medida que el tráfico de una red aumenta, la red debe hacer lo mismo. Esto implica nuevos dispositivos que deben ser configurados y gestionados. La cantidad de dispositivos de una red aumenta de forma progresiva, y debido a ello, la configuración y gestión tradicionales se convierte en un mecanismo tedioso, propenso a errores y poco sostenible a largo plazo.

- **Dependencia del fabricante:** Cada fabricante diseña sus dispositivos de red de una manera particular, con su propio sistema operativo y su propia sintaxis de configuración. En una red de telecomunicaciones es habitual que su infraestructura de red esté compuesta por dispositivos de diferentes vendedores y fabricantes, y es necesario conocer a fondo cada uno de los sistemas operativos.

Una vez identificadas las limitaciones de redes actuales, hay que identificar los beneficios que SDN puede dar a las entidades que opten por esta tecnología en sus redes de telecomunicaciones. Los principales beneficios de SDN son:

- **Reducción de Capital Expenditures (CapEx):** CapEx son los costes asociados a bienes físicos. SDN reduce la necesidad de invertir en nuevos bienes físicos (*hardware*), ya que se lleva a cabo una mejor gestión que induce a planificaciones más eficientes, con menor equipamiento se obtiene igual o menor rendimiento que de forma tradicional.
- **Reducción de Operational Expenditures (OpEx):** OpEx son los costes asociados a operaciones y servicios. SDN reduce este tipo de costes debido a su configuración y gestión de los elementos de red mediante *software*. Este control se realiza de forma más sencilla y automática, lo que permite una reducción del tiempo empleado por los administradores de la red.
- **Agilidad y flexibilidad:** SDN permite a las diferentes entidades desplegar aplicaciones, servicios e infraestructuras de forma rápida para alcanzar objetivos en el menor tiempo posible.

2.1.1. Arquitectura SDN

Las redes definidas por *software* constan de una arquitectura de red específica, compuesta por distintos componentes *hardware* y *software* que interactúan entre sí. Dichos componentes se explican a continuación:

- **Controlador SDN:** Es el cerebro de la red SDN. Forma el núcleo de la arquitectura SDN comunicándose con los *switches* a través de la **Interfaz Sur** (SouthBound Interface (SBI)) y con las distintas aplicaciones a través de la **Interfaz Norte** (NorthBound Interface (NBI)).
- **Interfaz Sur (SBI):** Es la interfaz que conecta al controlador SDN con el plano de datos. Facilita la configuración de la red, transfiriendo dichas configuraciones a los dispositivos de la red. El protocolo más utilizado en esta interfaz es **OpenFlow** (ver 2.1.2).
- **Interfaz Norte (NBI):** Es la interfaz que conecta al controlador SDN con el plano de control. Facilita el proceso de automatización de la red mediante la comunicación del controlador con las diferentes aplicaciones. Para permitir dicha comunicación con las aplicaciones, la interfaz norte exporta una API del tipo REpresentational State Transfer (REST).
- **Aplicaciones SDN:** Son programas que se conectan al controlador SDN mediante la interfaz norte. Gracias a su lógica de aplicaciones, desarrollan un próposito concreto y transfieren datos u órdenes al controlador SDN.

- **Agentes y *Drivers*:** Para establecer la comunicación entre el controlador y los dispositivos mediante la interfaz sur y para la comunicación entre el controlador y las aplicaciones mediante la interfaz norte, es necesario un par agente-*driver*.

  Mientras que en la NBI el driver se encuentra en la aplicación y el agente en el controlador, en la SBI, el driver se encuentra en el controlador y el agente en el dispositivo. Se encargan de realizar la comunicación entre la aplicación y el controlador en la NBI y entre el controlador y los dispositivos de red en la SBI, realizando las conversiones de lenguaje necesarias.

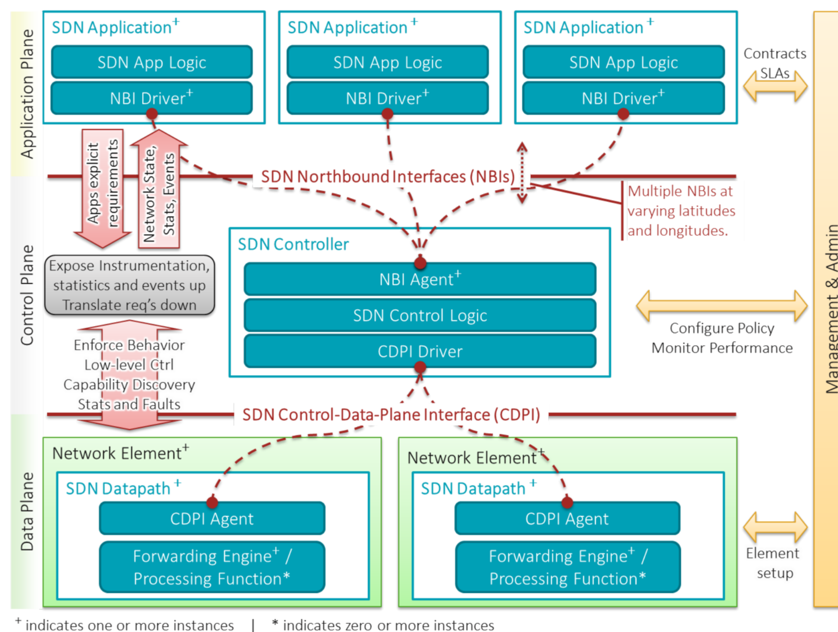

FIGURA 2.1: Arquitectura SDN. Fuente:[1]

En la figura 2.1 se pueden apreciar una visión general de la arquitectura SDN mostrando la conectividad entre los diferentes componentes explicados anteriormente.

2.1.2. OpenFlow

OpenFlow[3] es un protocolo estándar de SDN, siendo el más utilizado para comunicar el controlador SDN con los dispositivos que se encuentran en el plano de datos. Para que esta comunicación sea posible, el controlador SDN debe tener operando un *driver* OpenFlow, mientras que el dispositivo debe tener corriendo un agente OpenFlow.

Un *switch* tradicional utiliza su propia lógica de encaminamiento para decidir como tiene que reenviar los paquetes. Un *switch* OpenFlow es únicamente un dispositivo *hardware* que obedece órdenes que provienen del controlador SDN.

OpenFlow introduce el concepto de flujo, que sustituye a la entrada en la tabla de encaminamiento. Un flujo establece como un *switch* OpenFlow procesa un paquete determinado. Un flujo se compone de dos partes:

- **Selector o regla:** El selector define el conjunto de reglas que debe seguir un determinado paquete. Por ejemplo: tener una dirección MAC origen y/o destino específicas, o una dirección Internet Protocol (IP) origen y/o destino específicas.
- **Tratamiento o acción:** El tratamiento define como se va a procesar un determinado paquete. Por ejemplo: ser reenvíado por un determinado puerto o ser enviado directamente al controlador SDN.

A continuación, se empieza ya a hablar más en detalle de un *switch* OpenFlow. Dicho componente tiene las siguientes partes:

- **Agente o Cliente OpenFlow:** Es el encargado de comunicarse con el driver OpenFlow que se encuentra en el controlador SDN.
- **Tabla de flujos:** Es la tabla donde se almacenan los flujos del *switch*. Mantiene una relación entre el selector y el tratamiento, así como diferentes estadísticas de monitorización.
- **Puerto:** Este concepto es similar al de un *switch* tradicional.

2.2. NFV

NFV es un paradigma que engloba a las redes de telecomunicación, más concretamente a su infraestructura. Su principal premisa es la de desacoplar las funciones de red de dispositivos *hardware* y trasladarlas a servidores virtuales. Con ello se consigue tener múltiples funciones virtualizadas en un único servidor.[5]

Las redes de telecomunicación actuales tienen ciertos problemas que NFV pretende resolver, como son:

- Altos costes y restricciones físicas de fabricación.
- Complejidad *hardware* en las soluciones del fabricante.
- Ciclo de vida corto de los dispositivos *hardware*.

Todos estos problemas o desventajas han desembocado en la adopción de NFV como un estándar para agilizar, facilitar y escalar las infraestructuras de red, así como disminuir costes.

NFV tiene una serie de ventajas importantes que lo convierte en un alternativa óptima:

- **Simplificar la implantación de elementos de red:** Las soluciones de red NFV son flexibles, genéricas y fáciles de implantar, lo que implica que el proceso de actualización es también más rápido y sencillo.
- **Mayor escalabilidad de la red:** Gracias a la naturaleza *software* de las funciones virtualizadas, es mucho más fácil escalar los componentes de la red que si se trataran de dispositivos físicos.
- **Independencia respecto a los fabricantes de dispositivos:** Debido a que las funciones virtualizadas son desarrolladas mediante *software*, se pierde la dependencia

respecto a las particularidades de cada dispositivo, como su sistema operativo o su sintáxis de configuración y gestión, entre otros.

2.2.1. Arquitectura NFV

La virtualización de funciones de red consta de una arquitectura específica, compuesta por diferentes componentes *hardware* y *software* que interactúan entre sí, como se puede ver en la figura 2.2.

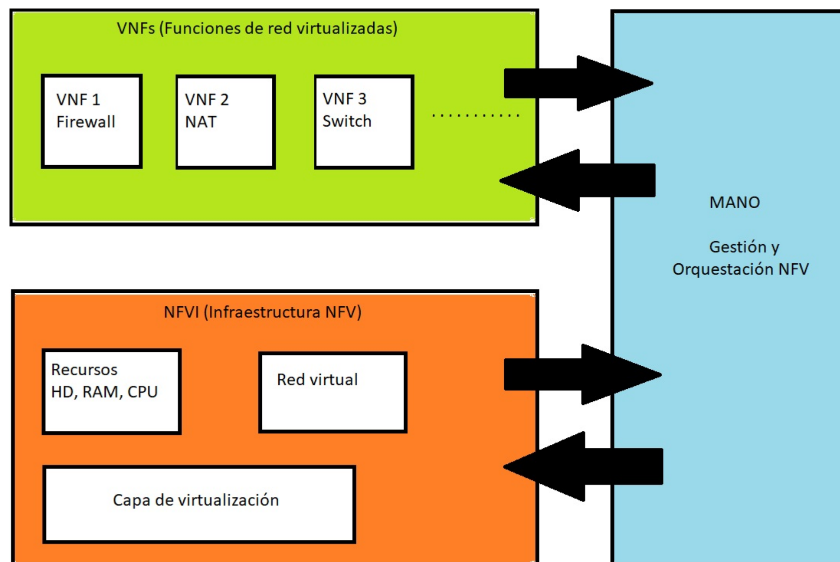

FIGURA 2.2: Arquitectura NFV. Fuente:[5]

La arquitectura NFV se compone principalmente de tres componentes, descritos a continuación:

- **Infraestructura NFV:** Es el componente que constituye la base de la arquitectura. Se trata del conjunto de equipos *hardware* y sus recursos (CPU, RAM, HD entre otros) que se utilizan para alojar las máquinas virtuales que componen las diferentes funciones de red virtualizadas (VNFs).
- **Bloque de VNFs:** Es el componente que engloba al conjunto de VNFs disponibles para instanciar. Una VNF viene descrita como un conjunto de máquinas virtuales que utilizan determinados recursos de la infraestructura.
- **MANO (*Manager and Orchestrator*):** Es el componente que interactúa tanto con la infraestructura NFV (NFV Infrastructure (NFVI)) como con el bloque de VNF. Se encarga de gestionar y orquestar a la propia infraestructura, controlando las acciones que sobre ella se realizan. Dichas acciones pueden ser una nueva instanciación de una VNF o un borrado de una VNF existente, entre otras.

Después de haber explicado cada uno de los tres componentes de la arquitectura NFV, es necesario poner un ejemplo completo de como se instancia una nueva función de red virtualizada, explicando las interacciones entre los componentes:

1. El usuario elige una VNF del bloque de VNFs para instanciar. Dicha orden es transferida al bloque MANager and Orchestrator (MANO).
2. El bloque MANO solicita a la infraestructura NFV información sobre sus recursos disponibles, para comprobar si la VNF podrá ser instanciada o no.
3. En caso afirmativo, el bloque MANO recibirá un OK de la infraestructura y procederá a mandar la orden de instanciación.
4. Una vez enviada la orden de instanciación, la infraestructura NFV reservará los recursos necesarios y arrancará el conjunto de máquinas virtuales pertenecientes a la VNF instanciada.
5. Si la instanciación ha sido correcta, el bloque MANO recibirá un OK, que será retransmitido al usuario.
6. El bloque MANO, gracias a su interacción con la infaestructura NFV (NFVI), proveerá al usuario de datos de monitorización de la VNF instanciada.

2.3. Relación entre SDN y NFV

SDN es un conjunto de técnicas para agilizar la gestión y configuración de redes de telecomunicación gracias a herramientas *software*. En cambio, NFV es un paradigma que persigue un cambio total en la infraestructura de una red de telecomunicación mediante el uso de funciones de red virtualizadas que sustituyan a los dispositivos físicos.

Puede parecer que ambas tecnologías tienen objetivos totalmente diferentes, pero gracias a su uso conjunto, ambos objetivos han convergido a uno solo, el de crear el concepto de Network Service Chaining (NSC).

NSC utiliza técnicas SDN y NFV para definir el concepto de *Service Chain*, una cadena de servicio. Ésta viene definida por un origen y un destino, y por un conjunto de servicios de red que deben ser atravesados en un orden específico.[4]

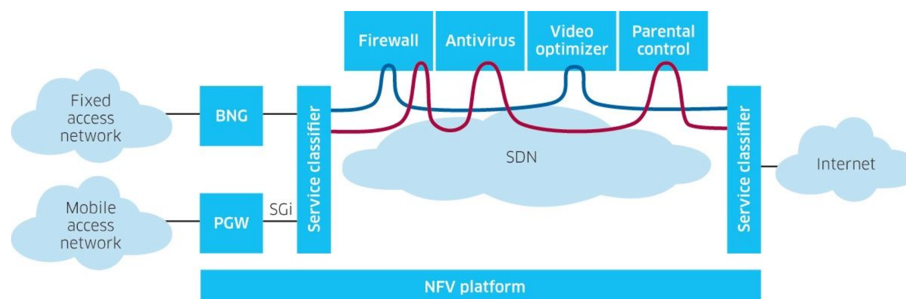

FIGURA 2.3: Ejemplo de *Service Chaining*. Fuente:[4]

En la figura 2.3 se puede observar como está constituida una *Service Chain*. Existen un origen y un destino, y un conjunto de servicios de red (Firewall, Antivirus, Optimizador de vídeo y Control Parental).

Mediante técnicas SDN se puede calcular la ruta óptima por el que debe pasar una *Service Chain* y transferir de forma fácil a la infraestructura de red las reglas determinadas

para seguir dicho camino. Por otro lado, mediante técnicas NFV se pueden virtualizar dichos servicios de red pertenecientes a la *Service Chain* y elegir los lugares óptimos para instanciarlos.

2.4. Herramientas Open-Source en redes de telecomunicación

Una red de telecomunicación es un conjunto de medios, tecnologías y protocolos que tienen como finalidad el intercambio de información entre diferentes usuarios.

Así mismo, se está produciendo una enorme evolución en el concepto de una red de telecomunicación.

En los orígenes, este concepto era puramente físico, con un conjunto de dispositivos *Hardware*, como pueden ser *routers*, *switches* u ordenadores, interactuando entre sí. Actualmente, la gran mayoría de las redes de telecomunicación utilizan software open-source para diferentes propósitos:

- Sacar el máximo rendimiento a su infraestructura.
- Agilizar el envío y procesamiento del tráfico de la red.
- Acelerar y automatizar la gestión y configuración de los dispositivos.
- Reducir los costes de operación de la red.

Para conseguir los propósitos mencionados anteriormente, existen numerosas herramientas de software desarrolladas por empresas, universidades u organizaciones que están totalmente disponibles para ser usadas por cualquier usuario.

Dichas herramientas forman un gran conjunto heterogéneo, siendo desarrollada cada una de ellas para uno o más propósitos:

- Para sacar el máximo rendimiento a la infraestructura de una red, existen herramientas de virtualización como **OpenStack**[23] o **Docker**[6], para proveer a las aplicaciones de una abstracción e independencia. Este tipo de herramientas se enmarcan dentro de la tecnología **NFV** (ver 2.2).
- Para agilizar el envío y procesamiento del tráfico de una red, existen herramientas que permiten emular el comportamiento de diversos dispositivos hardware a través de la virtualización, como **OpenVSwitch**[7] o **Mininet**[11]. Gracias a este tipo de herramientas, se puede sustituir un *Switch* clásico por un pequeño software que hace las mismas funciones que uno tradicional.
- La gestión de los dispositivos se ha realizado de forma manual, dispositivo a dispositivo. Gracias a herramientas software como **Cacti**[8] o **Nagios**[9], la forma de gestionar las redes de telecomunicación ha dado un giro de 180 grados. Utilizando estas herramientas, el usuario puede gestionar la red desde un terminal de manera remota, gracias al protocolo Simple Network Management Protocol (SNMP).
- Para acelerar y automatizar la configuración de los dispositivos de una red de telecomunicación, existen herramientas enmarcadas en la tecnología *SDN* (ver 2.1). Un ejemplo de estas herramientas es el controlador SDN ONOS (ver 3.3).

- Para reducir los costes de operación de una red, existen herramientas open-source que ayudan a planificar una red de telecomunicación de forma óptima. La más conocida de estas herramientas es **Net2Plan**[10], una herramienta de planificación de redes programada en Java.

Capítulo 3

Herramientas utilizadas

En este capítulo se va a hacer una descripción de cada una de las herramientas utilizadas en el proyecto, haciendo especial mención a Net2Plan, que es una herramienta cuya funcionalidad es la de emular y planificar una red de telecomunicación. Dicha herramienta constituye la base para el desarrollo de este proyecto, más concretamente de la prueba de concepto.

Además de Net2Plan, se han utilizado otras herramientas para complementar la funcionalidad de Net2Plan y permitirle emular escenarios basados en las tecnologías SDN y NFV. Dichas herramientas son Mininet, para emular redes de telecomunicación; ONOS, un controlador SDN; OSM, un gestor y orquestador NFV; y OpenStack, una herramienta para emular una infraestructura IT.

Todas estas herramientas operan en sintonía para llevar a cabo una prueba de concepto exitosa.

3.1. Net2Plan

Net2Plan[10] es una herramienta *open-source* programada en Java dedicada a la planificación, optimización y simulación de redes de comunicaciones desarrollada por el grupo de investigación GIRTEL de la Universidad Politécnica de Cartagena. En sus inicios, fue concebida como una herramienta para docencia sobre redes de comunicaciones. Sin embargo, actualmente se ha convertido en una poderosa herramienta de optimización y planificación de redes, con un repositorio de recursos para la planificación de redes, tanto para el entorno académico como para el entorno de la industria y la empresa.

Net2Plan está basado en una representación de redes con componentes abstractos, tales como nodos, enlaces, demandas o rutas, entre otros. Esto está pensado para poder planificar cualquier tipo de red, sin importar la tecnología que utilice. Para poder personalizar las redes a gusto del usuario, cada componentes permite añadir atributos. Además, hay clases que permiten modelar una tecnología en concreto (redes IP, Wavelength Division Multiplexing (WDM) o escenarios de NFV).

Una ventaja de esta herramienta es que tiene dos modos de uso: mediante interfaz gráfica (GUI) y línea de comandos (Command Line Interface (CLI)). La interfaz gráfica está pensada para utilizar en sesiones de laboratorio como un recurso formativo, o para poder ver más detalladamente la red sobre la que se está trabajando. Por otro lado,

el modo línea de comandos facilita los estudios de investigación, ya que permite automatizar ejecuciones de algoritmos o simulaciones mediante *scripts*. Como se ha hablado antes, ambos modos permiten utilizar Net2Plan en el entorno académico (investigación o enseñanza) y en el entorno de la industria y la empresa.

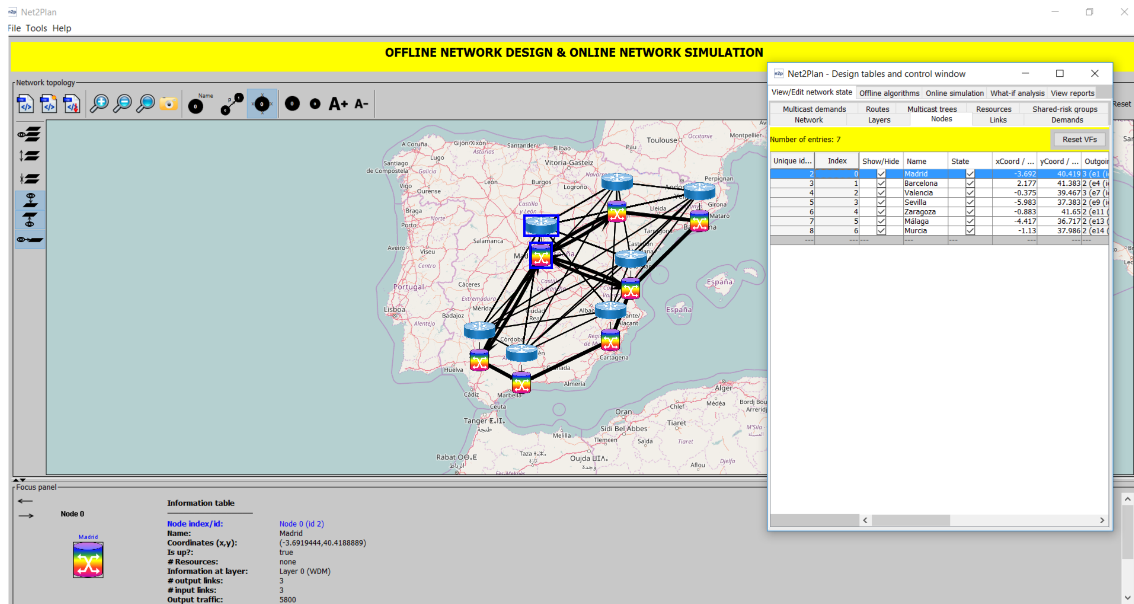

FIGURA 3.1: Plugin *Offline network design and online network simulation* de Net2Plan

En la figura 3.1 se puede ver el aspecto de la interfaz gráfica de Net2Plan, donde se muestra una topología de España con sus respectivas tablas que aportan información detallada de cada uno de los componentes.

3.2. Mininet

Mininet[11] es una herramienta programada en Python cuyo objetivo es el de emular redes de telecomunicación. Permite crear redes con *hosts*, *switches*, controladores y enlaces a un alto nivel. Los *hosts* de Mininet corren bajo un sistema operativo Linux, mientras que los *switches* soportan el protocolo OpenFlow (ver 2.1.2) para mayor flexibilidad respecto a la configuración del *routing* y para integrarlos dentro de un escenario SDN (ver 2.1).

Mininet tiene una gran polivalencia, y eso permite que sea utilizado en diferentes tareas, tales como investigación, desarrollo, aprendizaje o testeo. Gracias a ello, se puede conseguir emular una red con un comportamiento similar a una real.

Sus principales características son:

- Provee un amplio banco de pruebas para desarrollar aplicaciones basadas en OpenFlow.
- Permite que varios desarrolladores trabajen de forma concurrente sobre la misma topología de red.

- Permite realizar tests exhaustivos de topologías sin necesidad de tener una real.
- Incluye una Interfaz de Línea de Comandos que es independiente de la topología emulada y del protocolo que ésta utilice.
- Permite crear desde topologías mas sencillas con un único comando hasta topologías realmente complejas haciendo uso de una API programada en Python para definir los componentes con total detalle.

Las redes emuladas por Mininet ejecutan aplicaciones estandarizadas de Linux, como el kernel del propio sistema Linux. Esto permite que cualquier desarrollo llevado a cabo y testeado en Mininet pueda ser movido a un sistema real realizando las mínimas modificaciones posibles.

3.3. ONOS

ONOS[12] es un proyecto Open-Source perteneciente a The Linux Foundation. Su principal objetivo es el de crear un controlador SDN para proveedores de servicios de comunicaciones.

Sus principales características son:

- **Escalabilidad:** Ofrece replicación ilimitada mediante virtualización para poder añadir y quitar capacidad al plano de control según sea necesario.
- **Resiliencia:** Provee la disponibilidad requerida por los operadores de red en momentos críticos.
- **Retrocompatibilidad:** Permite añadir o configurar dispositivos y servicios con configuración basada en modelos.
- **Soporte a dispositivos de nueva generación:** Ofrece control en *real-time* para dispositivos OpenFlow.
- **Modularidad:** Las funcionalidades de ONOS están definidas en modulos localizados, lo que hace más fácil probar y mantener el software en buen estado.

ONOS está programado en Java y opera como un clúster de nodos idénticos en cuanto al software. Trabaja con modelos y protocolos estandarizados, tales como OpenFlow (ver 2.1.2), NETCONF, OpenConfig, OpenROADM, entre otros.

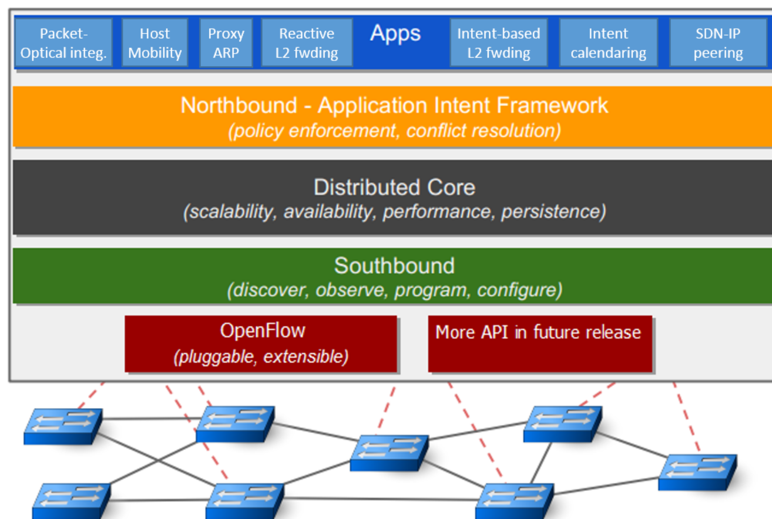

FIGURA 3.2: Arquitectura de ONOS. Fuente: [13]

En la figura 3.2 se observa la arquitectura interna de ONOS. En el *Core* se encuentran los controladores de los diferentes servicios que ofrece (TopologyService, DeviceService, HostService, entre otros), cada uno de ellos destinado a controlar un tipo de componente.

También se puede observar que, para acceder a estos controladores, las aplicaciones necesitan hacer uso de la interfaz *NorthBound*, que se compone de varias subinterfaces que exportan diferentes APIs.

Por otro lado, para que los controladores puedan tener constancia de los dispositivos de la red, la interfaz *SouthBound* (SBI) incluye diferentes *drivers*, genéricos o particulares, para poder comunicarse con dispositivos mediante numerosos protocolos estandarizados, como se explicó anteriormente.

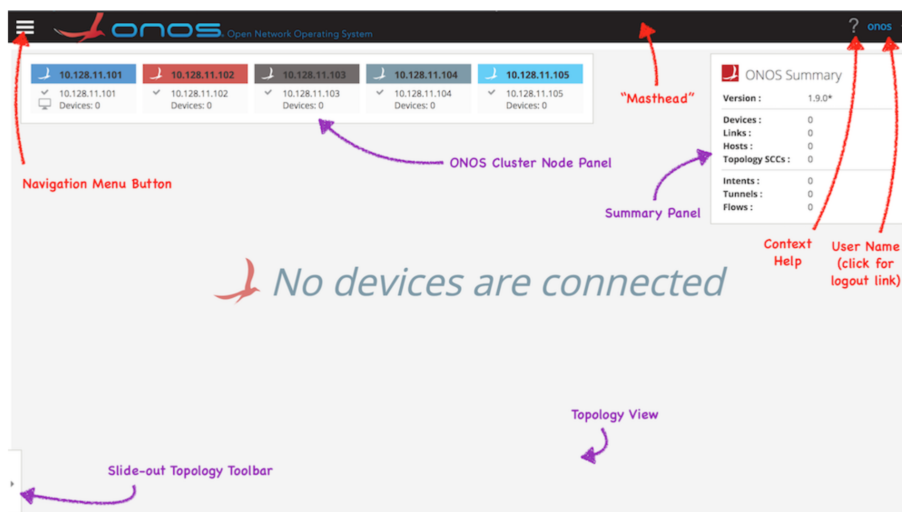

FIGURA 3.3: Interfaz Gráfica de ONOS. Fuente: [14]

Para facilitar la interacción con el usuario, ONOS ofrece una GUI, como se puede ver en la figura 3.3, para ver en más detalle la topología que esta siendo gestionada, así como datos más específicos de cada uno de los dispositivos de la red.

3.3.1. OpenAPI y Swagger

OpenAPI[15] es una iniciativa creada por varios expertos de la industria y la investigación para estandarizar las descripciones de las REST-APIs. Su principal objetivo es crear y promover un formato de descripción genérico.

Swagger[16] es un conjunto de herramientas open-source para definir y documentar REST-APIs.

Mediante la colaboración entre OpenAPI, que establece un modelo de APIs común, y Swagger, que permite diseñar una REST-API de forma simple, las aplicaciones podrán conectarse entre sí de forma sencilla, y ayudará a tener un mundo más comunicado.

3.4. OSM

OSM[17] es un software *open-source* cuya función principal es la orquestación de servicios de red avanzados en infraestructuras NFV heterogéneas. Surge como iniciativa del European Telecommunications Standards Institute (ETSI)[18] para crear una arquitectura NFV común para los operadores de red.

OSM trabaja con una serie de componentes/elementos que ayudan a definir su arquitectura:

- **Virtual Deployment Unit (VDU):** Es el componente más básico de la arquitectura OSM. Se encarga de definir una máquina virtual.
- **VNF:** Es el componente que define una función de red virtualizada. Puede estar compuesto de un único VDU o por más de uno.
- **Network Service (NS):** Se compone de uno o más VNFs que realizan una función de red conjuntamente.
- **Virtual Link Descriptor (VLD):** Define las conexiones directas entre diferentes componentes de la arquitectura. Hay principalmente dos tipos de VLD: VDU-VDU y VNF-VNF.
- **Virtual Network Function Descriptor (VNFD):** Se encarga de definir los recursos necesarios para instanciar un VNF. Incluye diferentes componentes: lista de VDUs que lo definen, lista de VLDs, entre otros.
- **Network Service Descriptor (NSD):** Este componente se encarga de definir la configuración de un NS. Incluye diferentes componentes: lista de VNFDs que lo componen, lista de VLDs, parámetros de configuración iniciales, entre otros.
- **Virtual Infrastructure Manager (VIM):** Este elemento define un controlador de una o más infraestructuras donde se alojaran las diferentes máquinas virtuales. Es el encargado de comunicar a OSM con las diferentes infraestructuras NFV.

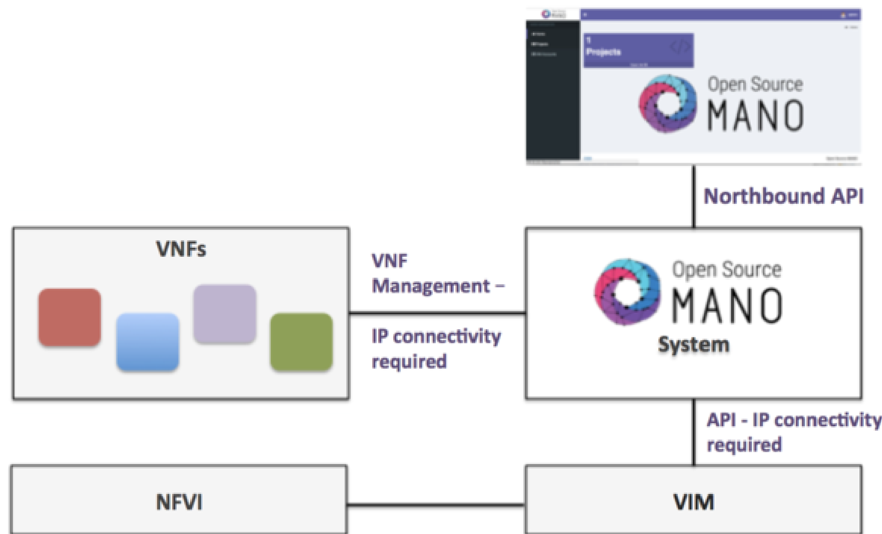

FIGURA 3.4: Arquitectura de OSM. Fuente: [19]

Para ayudar a explicar el funcionamiento de OSM, la figura 3.4 da una visión general de la interaccion entre OSM y los diferentes componentes:

- **Interfaz *NorthBound*:** OSM exporta una REST-API accesible por su interfaz *NorthBound*. Mediante peticiones HyperText Transfer Protocol (HTTP), como GET, POST o DELETE, el usuario es capaz de ejecutar órdenes en OSM, tales como crear un nuevo VIM o instanciar un nuevo VNF, entre otras.

  Para ello, es necesario tener un cliente desde el cuál enviar órdenes. La ETSI ofrece una interfaz gráfica web que se instala al mismo tiempo que OSM y un cliente por línea de comando escrito en Python (ver 3.4.1).

- **Conexión con VIM:** OSM permite la comunicación con múltiples tipos de VIMs (OpenStack, OpenVIM, VMWare y Amazon Web Services). Para ello, es necesaria conectividad IP entre OSM y el propio VIM, ya que las órdenes enviadas por OSM al VIM para realizar operaciones son hechas mediante una REST-API.

- **Conexión VIM-NFVI:** NFVI es el conjunto de recursos (Random Access Memory (RAM), Central Processing Unit (CPU), Hard Disk (HD), entre otros) que son utilizados para instanciar las diferentes máquinas virtuales. El VIM actúa como un controlador de las diferentes NFVIs y gestiona las interacciones entre OSM y ellas.

  En estructuras de trabajo pequeñas, es habitual que un VIM y su NFVI estén en la misma máquina física, aunque para estructuras reales de trabajo, la NFVI de un VIM puede estar distribuida en diferentes máquinas físicas.

- **VNF *Management*:** cuando un VIM instancia un nuevo VNF, se le asigna una dirección IP para poder acceder a la propia máquina virtual y gestionarla. Por ello, es necesario que haya conectividad IP entre OSM y todos los VNFs.

3.4.1. OSMClient

OSMClient[20] es un cliente REST programado en Python por la ETSI que proveé una CLI para interactuar con OSM. Fue introducido en la *release* 2 y permite al usuario ejecutar numerosas acciones, como subir un nuevo descriptor, crear un nuevo VIM, instanciar un nuevo NS o eliminar un NS, entre otras.

En sus orígenes, cuando la *release* 2 de OSM fue dispuesta al público, dicho cliente tenía una funcionalidad limitada y únicamente podía ser instalado junto a la versión completa de OSM.

Cuando la ETSI publicó la *release* 4 de OSM, el cliente sufrió un gran lavado de cara. Se permitió una instalación única de OSMClient sin necesidad de instalar la versión completa de OSM, y se añadió una nueva versión del cliente que seguía el estándar sol005[21], el cual define un nuevo formato de REST-APIs para aplicaciones propias de la ETSI. También se mantuvo el formato correspondiente a las *releases* 2, 3 para tener retrocompatibilidad con ellas.

```
osm vim-create --name openstack-site --user admin --password userpwd \
    --auth_url http://10.10.10.11:5000/v2.0 --tenant admin --account_type openstack
```

FIGURA 3.5: Ejemplo de uso de OSMClient. Fuente:[22]

En la figura 3.5 se puede ver un pequeño ejemplo de uso de OSMClient para crear un nuevo VIM. Se deben introducir diferentes parámetros, como el tipo de VIM, en este caso OpenStack, usuario y contraseña del VIM, y la Uniform Resource Locator (URL) donde se encuentra el controlador del VIM.

3.5. OpenStack

OpenStack[23] es un proyecto basado en la nube que proporciona un *datacenter* para controlar y gestionar grandes cantidades de recursos de computación, almacenamiento y red. Para facilitar la gestión de recursos, OpenStack provee al usuario una interfaz gráfica a la vez que también exporta una REST-API para permitir conectividad con aplicaciones externas.

OpenStack está compuesto de diferentes servicios o bloques, encargándose cada uno de ellos de una funcionalidad concreta dentro de la arquitectura global. La principl característica de dichos servicios es que cada uno de ellos es accesible de forma indepdendiente.

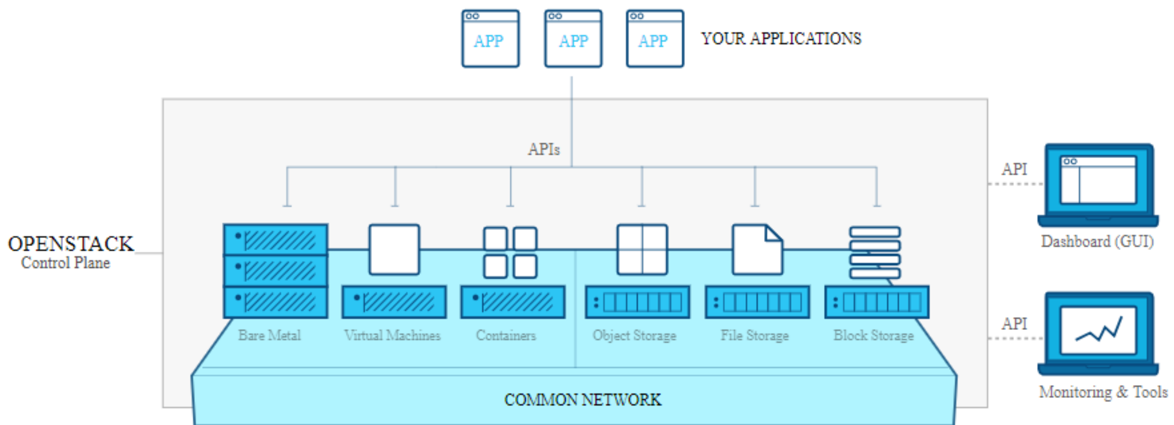

FIGURA 3.6: Arquitectura de OpenStack. Fuente:[23]

En la figura 3.6 se puede apreciar la arquitectura de OpenStack en la que operan los diferentes servicios. Dichos servicios se explican a continuación:

- **Keystone:** Este servicio controla la identificación de los diferentes usuarios que se conecten a la infraestructura de OpenStack, y el acceso a según que aplicaciones de los mismos.
- **Horizon:** Este servicio es el encargado de mostrar la gestión completa de OpenStack mediante una interfaz gráfica. Desde ella se puede observar con todo detalle que está sucediendo en el sistema y poder gestionar los posibles fallos.
- **Nova:** Este servicio está considerado el motor de OpenStack. Es el encargado de desplegar y administrar las diferentes máquinas virtuales instanciadas y otros servicios que se necesiten.
- **Neutron:** Este servicio es el encargado de proporcionar servicios de red entre los diferentes dispositivos. Permite crear y gestionar redes y subredes.
- **Glance:** Este servicio se encarga de gestionar las diferentes imágenes que se usan en la infraestructura.
- **Cinder:** Este servicio se centra en el almacenamiento. Facilita el acceso al contenido alojado en las unidades de disco que se encuentren en la infraestructura.
- **Swift:** Este servicio es el encargado de almacenar los diferentes archivos del sistema, asegurar su integridad y replicarlos por los diferentes volúmenes de la infraestructura, para hacer más dinámicas la accesibilidad y la disponibilidad.

3.5.1. OpenStack4Java

OpenStack4Java[24] es una librería REST *open-source* programada en Java para controlar y gestionar un sistema basado en OpenStack.

Permite al usuario realizar una gestión de OpenStack eficiente gracias a sus múltiples módulos, cada uno de ellos focalizado en gestionar un servicio concreto de OpenStack:

- **Identity:** Este módulo se encarga de gestionar el servicio **Keystone**. Su principal objetivo es el de gestionar el directorio de usuarios, grupos, regiones, servicios y **endpoints**. Así mismo, se encarga de autenticar y autorizar a los diferentes usuarios para utilizar los diferentes servicios.
- **Compute:** Este módulo se encarga de gestionar el servicio **Nova**. Es el encargado de gestionar las diferentes máquinas virtuales que están corriendo en OpenStack.
- **Network:** Este módulo se encarga de gestionar el servicio **Neutron**. Provee conectividad entre diferentes componentes de OpenStack. Permite a los usuarios crear sus propias redes y añadirles interfaces.
- **Image:** Este módulo se encarga de gestionar el servicio **Glance**. Su principal funcionalidad es la de proveer diferentes servicios para la gestión de imágenes. Permite almacenar imágenes personalizadas por el usuario para inicializar máquinas rápidamente.
- **Block Storage:** Este módulo se encarga de gestionar el servicio **Cinder**. Permite al usuario crear y montar volúmenes para escalar el almacenamiento.
- **Object Storage:** Este módulo se encarga de gestionar el servicios **Swift**. Es el encargado de crear almacenamiento persistente para los diferentes archivos alojados en el sistema.

Identity

```
// V2 authentication
OSClientV2 os = OSFactory.builderV2()
                .endpoint("http://127.0.0.1:5000/v2.0")
                .credentials("admin","secret")
                .tenantName("admin")
                .authenticate();

// V3 authentication
OSClientV3 os = OSFactory.builderV3()
                .endpoint("http://127.0.0.1:5000/v3")
                .credentials("admin", "secret", Identifier.byName("Def
                .scopeToProject(Identifier.byName("admin"))
                .authenticate();
```

Compute

```
// Create a Server Model Object
Server server = Builders.server()
                        .name("Ubuntu 2")
                        .flavor("large")
                        .image("imageId")
                        .build();

// Boot the Server
Server server = os.compute().servers().boot(server);

// Create a Snapshot
os.compute().servers().createSnapshot("id", "name");
```

Image

```
// Create an Image
Image image = os.images().create(Builders.image()
               .name("Cirros 0.3.0 x64")
               .isPublic(true)
               .containerFormat(ContainerFormat.BARE)
               .diskFormat(DiskFormat.QCOW2)
               .build()
               ), Payloads.create(new File("cirros.img")));
```

Network

```
// Create a Port
Port port = os.networking().port()
              .create(Builders.port()
              .name("port1")
              .networkId("networkId")
              .fixedIp("52.51.1.253", "subnetId")
              .build());
```

FIGURA 3.7: Ejemplo de uso de OpenStack4Java. Fuente:[24]

En la figura 3.7 se puede ver un breve ejemplo de como utilizar los servicios Identity, Compute, Image y Network.

Capítulo 4

Desarrollo de Aplicaciones

En este capítulo se hablará sobre las diferentes librerías que se han realizado en este proyecto, explicando detalladamente la estructura de cada una de ellas.

Primeramente se explicará J-OSMClient, que es el cliente programado en Java para comunicarse con la REST-API que exporta OSM.

Seguidamente se hablará de J-ONOSClient, que es el cliente encargado de comunicarse con la REST-API de ONOS, y de J-OpenStack Client, que se encarga de comunicarse con la REST-API de OpenStack.

Por último, se hará especial énfasis en el plugin de Net2Plan desarrollado, en el cual se han integrado las diferentes APIs mencionadas anteriormente.

4.1. J-OSM Client

J-OSMClient[25] es una librería programada en Java cuya funcionalidad es la de proporcionar un cliente REST para interactuar con OSM (ver 3.4). Está basado en el cliente programado en Python por la ETSI (ver 3.4.1).

Se compone principalmente de tres clases que realizan la comunicación con OSM y devuelven los resultados de las interacciones al usuario:

- **OSMControllerRelease3:** Es la entidad que se encarga de realizar la comunicación con la *release* 3 de OSM. Mediante llamadas HTTP (GET, POST, PUT, DELETE), dicha clase envía peticiones y procesa internamente las respuestas recibidas.
- **OSMControllerSOL005:** Esta entidad se encarga de realizar la comunicación con la *release* 4 en adelante, utilizando el estándar sol005[21]. Al igual que el *controller* de la *release* 3, establece la comunicación con llamadas HTTP enviando peticiones y procesando las diferentes respuestas recibidas.
- **OSMClient:** Es la clase principal de la aplicación. Actúa como interfaz entre el usuario final y los *controllers*, permitiendo que el usuario obtenga directamente en formato más amigable las respuestas dadas por el servidor de OSM.

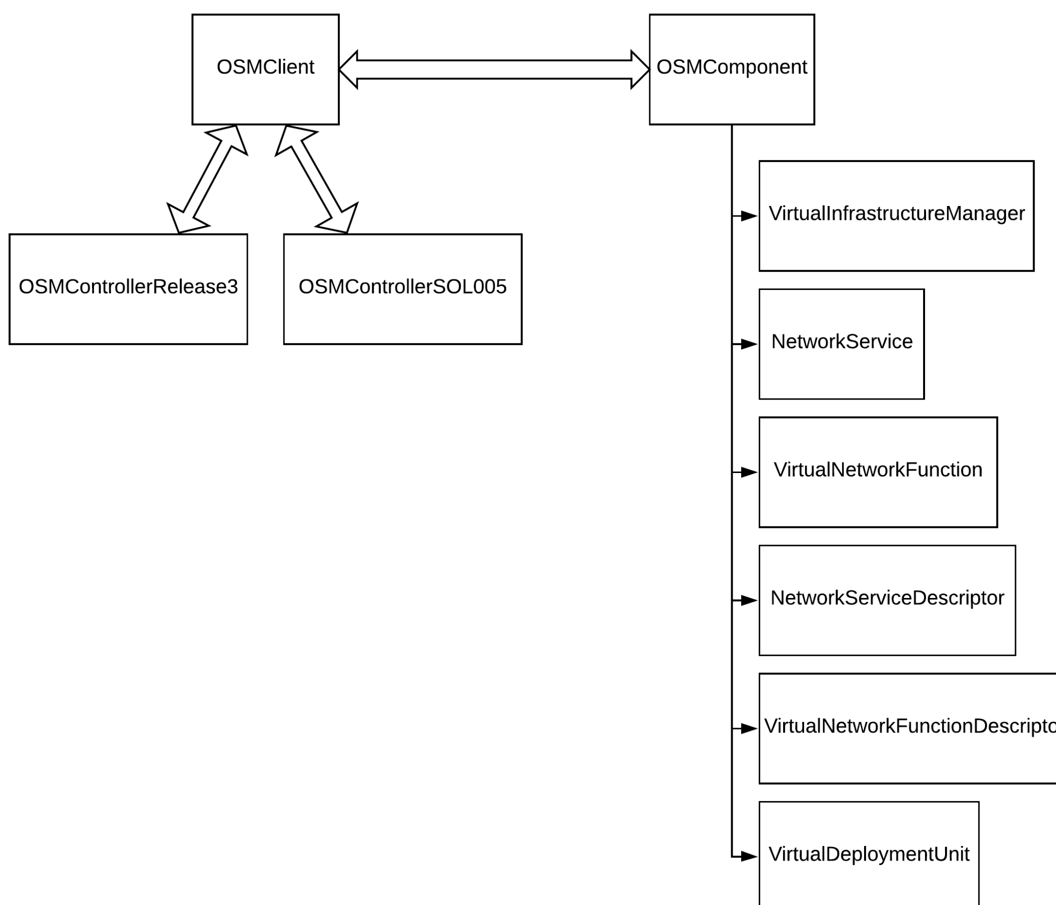

FIGURA 4.1: Estructura de clases de J-OSMClient

En la figura 4.1 se puede ver un esquema detallado de la jerarquía de clases. Se aprecia como OSMClient interactúa directamente con ambos *controllers* para establecer la comunicación con ambas versiones de OSM.

Así mismo, se pueden observar las clases auxiliares que definen los diferentes componentes internos de OSM. Estas clases existen debido a que OSM envía las respuestas HTTP con un formato JavaScript Object Notation (JSON), que no es el formato óptimo para que el usuario final las reciba y las trate.

Estas clases auxiliares se explican a continuación:

- **OSMComponent:** Es la clase genérica que define un componente de OSM. Todas las demás clases heredan de ella, lo que permite un mejor procesamiento interno de las respuestas recibidas de OSM, ya que proveé atributos comunes a todos los componentes, como nombre o identificador interno.
- **VirtualInfraestructureManager:** Esta clase es la que modela el componente VIM. Proveé atributos como su URL y su tipo (OpenStack, OpenVim, VMWare o Amazon Web Services (AWS)).

- **VirtualLinkDescriptor:** Esta clase es la que define el componente VLD. Proporciona un único atributo, que es una lista de conexiones en las que se indica los dos puertos que se conectan gracias a él.
- **VirtualDeploymentUnit:** Esta clase define el componente VDU. Proveé diferentes atributos como la imagen que lo define y los recursos necesarios para su creación (HD, CPU y RAM).
- **VirtualNetworkFunctionDescriptor:** Esta clase es la que modela el componente VNFD. Proporciona únicamente un atributo, que es la lista de VDUs que lo componen.
- **VirtualNetworkFunction:** Esta clase define el componente VNF. Proveé diferentes atributos, como el NS al que pertenece, el VIM donde está instanciado o el VNFD que lo define, entre otros.
- **NetworkServiceDescriptor:** Esta clase es la que modela el componente NSD. Proporciona diferentes atributos, como la lista de VNFDs que lo componen o la lista de VLDs.
- **NetworkService:** Esta clase modela el componente NS. Está compuesto de diferentes atributos, como los VIMs donde está instanciados sus VNFs, el NSD que lo define, la lista de VNFs que lo forman o su estado actual.

4.2. J-ONOS Client

J-ONOSClient[26] es una librería programada en Java cuya funcionalidad es la de proporcionar un cliente REST para interactuar con ONOS (ver 3.3). Esta librería está basada en la representación de objetos que proveé Swagger (ver 3.3.1). Gracias a Swagger, se ha generado un cliente que es capaz de interactuar con ONOS.

Este cliente generado proveé una representación de diferentes componentes, cada uno de ellos con una clase asociada que actúa como interfaz con ONOS para controlar las interacciones que involucran a dicho componente. A continuación se explica cada uno de los componentes mencionados anteriormente:

- **Host:** Este componente representa un usuario final en una red, más concretamente una Network Interface Card (NIC). Proporciona diferentes atributos como su identificador interno, su dirección IP, su dirección Media Access Control (MAC) o el identificador de la Virtual Local Area Network (VLAN) a la que pertenece.
- **Device:** Este componente es una representación de un dispositivo perteneciente a la infraestructura de red. Proporciona los siguientes atributos: su identificador interno, un identificador de *chassis*, un número de serie único, el identificador del fabricante, un número de versión de *hardware*, un número de versión de *software* y el tipo de dispositivo.
- **Port:** Este componente representa un puerto físico de red. Proveé al usuario de distintos atributos, entre los que destacan el dispositivo al que pertenece, el número de puerto, la velocidad que soporta y el tipo de puerto.

- **Link:** Este componente es una representación de un enlace entre dos dispositivos de la infraestructura de red. Proporciona los siguiente atributos: el dispositivo origen y el dispositivo destino, el estado del enlace y el tipo de enlace.
- **Flow:** Este componente representa una regla de flujo que se aplica en un dispositivo concreto. Proveé al usuario de distintos atributos, como su identificador interno, el selector y el tratamiento que lo componen y si es permanente o no, entre otros.
- **Intent:** Este componente es una representación de una combinación origen-destino. Gracias a él, el usuario puede definir un nodo origen y un nodo destino entre los cuales se cursará una demanda de tráfico, y ONOS traduce dicho *intent* a un conjunto de reglas de flujo totalmente transparentes al usuario, sin necesidad de crearlas automáticamente. Proporciona diversos atributos, entre los que destacan su identifcador interno, su prioridad y si ha sido posible llevarlo a cabo (si no existe un camino entre el origen y el destino, la instalación del *intent* fallará).

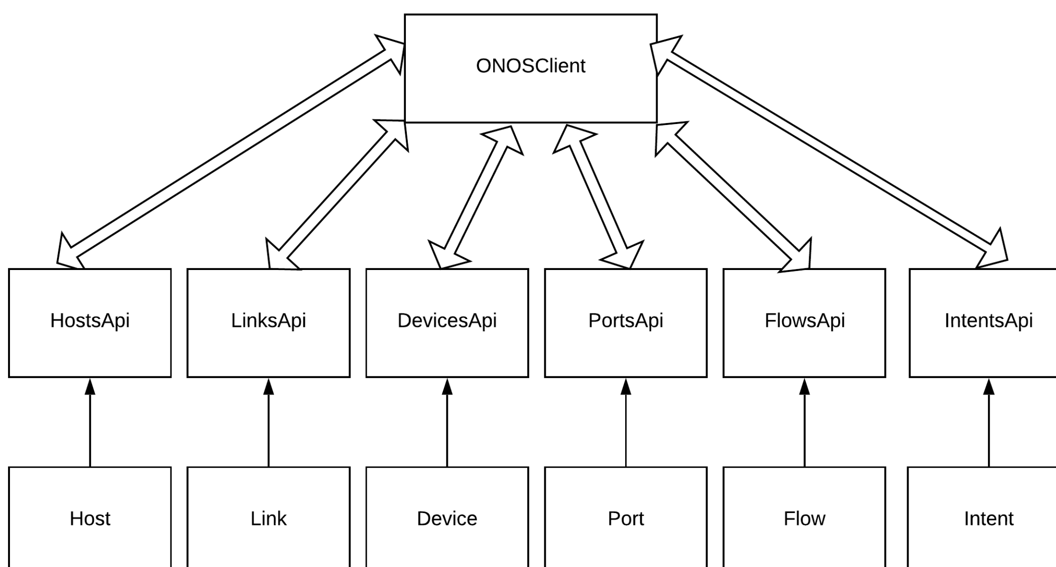

FIGURA 4.2: Estructura de clases de J-ONOSClient

En la figura 4.2 se puede apreciar un esquema de la jerarquía de clases. Se puede observar como cada componente tiene su propia API y todas ellas son utilizadas por la clase ONOSClient.

ONOSClient es la clase principal del cliente que actúa como interfaz entre el usuario y ONOS interactuando con la REST-API de ONOS mediante las APIs de los diferentes componentes.

Dicha clase es necesaria, ya que Swagger genera varias APIs, cada una de ellas para controlar las interacciones que involucran a un componente específico, pero no permite relacionar los componentes entre sí, de manera que, por ejemplo, para replicar una topología completa con todos los componentes, sin la clase ONOSClient sería imposible agruparlos.

Como ejemplo, si el cliente quiere obtener información sobre un *device* en concreto, ONOSClient realizará una llamada a la clase DevicesApi, que internamente realizara una petición HTTP GET a una URL determinada para obtener la información buscada. La propia API realizará una conversión de JSON al objeto *device* para proporcionar un mejor manejo de datos al usuario.

4.3. J-OpenStack Client

J-OpenStackClient[27] es una librería programada en Java cuya funcionalidad es la de proporcionar un cliente REST para interactuar con OpenStack (ver 3.5). Esta librería es un *wrapper* de la librería *open-source* OpenStack4Java (ver 3.5.1).

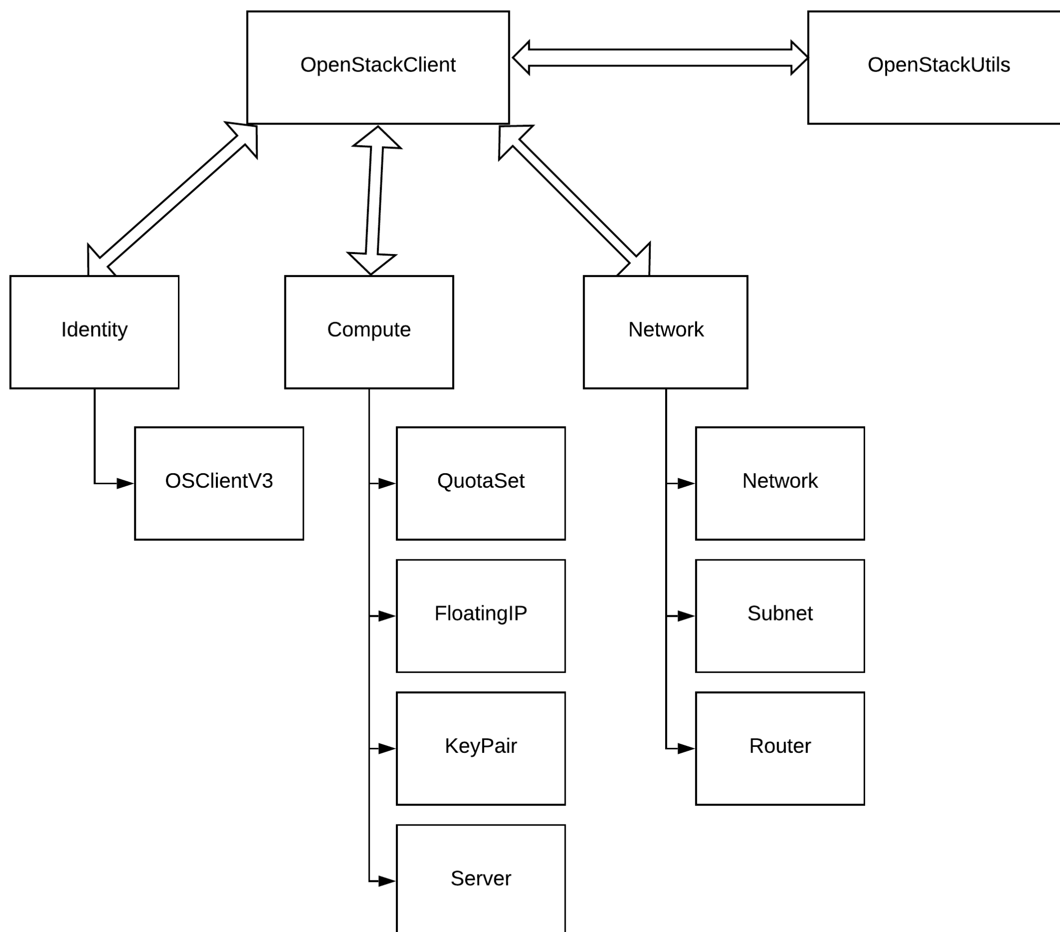

FIGURA 4.3: Estructura de clases de J-OpenStackClient

Cada uno de los servicios de OpenStack tiene una representación en este cliente, como se puede apreciar en la jerarquía de clases mostrada en la figura 4.3. A su vez, cada servicio gobierna una serie de clases que modelan los diferentes componentes internos de OpenStack.

Toda estas funcionalidades de los diferentes servicios son adquiridas de OpenStack4Java. El principal trabajo en este cliente es el de generar una clase central que, utilizando las clases asociadas a los diferentes servicios, permita al usuario controlar las interacciones entre ellos y sus componentes de una manera amigable y sencilla.

A continuación se realiza una explicación de la jerarquía de clases perteneciente a J-OpenStackClient:

- **OpenStackClient:** Es la clase principal de J-OpenStackClient. Actúa como interfaz entre el usuario y las diferentes APIs que controlan a su vez las interacciones con los diferentes componentes.
- **Identity:** Esta clase abstracta es la que modela el servicio KeyStone de OpenStack. Proporciona acceso al resto de servicios de OpenStack mediante autenticación por usuario-contraseña, gracias a laS clases OSClienteV2, la cual se encuentra deprecada y no se usa en las versiones actuales de OpenStack, y OSClientV3.

```
OSClientV3 os = OSFactory.builderV3()
                        .endpoint("http://127.0.0.1:5000/v3")
                        .credentials("admin","sample", domainIdentifier)
                        .authenticate();
```

FIGURA 4.4: Autenticación mediante OSClientV3. Fuente:[24]

En la figura 4.4 se puede apreciar un ejemplo de autenticación mediante la clase mencionada anteriormente.

- **Compute:** Esta clase abstracta es la que modela el servicio Nova de OpenStack. Proporciona funcionalidad de gestión sobre las diferentes máquinas virtuales y su configuración. Para ello, se apoya en diferentes clases, entre las que destacan:
  - **Server:** Esta clase modela una máquina virtual que está corriendo en OpenStack.
  - **FloatingIP:** Esta clase modela una dirección IP flotante, que se utiliza para que una máquina virtual sea accesible desde el exterior (su uso es similar a NAT).
  - **KeyPair:** Esta clase modela un par de claves pública-privada para acceder a una máquina virtual mediante Secure SHell (SSH).
  - **QuotaSet:** Esta clase modela los recursos perteneciente a una máquina virtual, tales como número de CPU, memoria RAM, almacenamiento HD, IPs flotantes, entre otros.
- **Network:** Esta clase abstracta modela el servicio Neutron de OpenStack. Proporciona un control sobre los componentes perteneciente a la infraestructura de red. Para ello, se apoya en las siguientes clases:
  - **Network:** Esta clase modela una red, que está compuesta por diferentes subredes y por *routers* para encaminar el tráfico entre diferentes subredes.
  - **Subnet:** Esta clase modela una subred, que es una subdivisión de una red. Una subred tiene su propia dirección IP y su máscara de red.

- **Router:** Esta clase es la que modela un *router*, que se encarga de encaminar el tráfico entre diferentes subredes.

- **OpenStackUtils:** Es una clase auxiliar que complementa a OpenStackClient. Proporciona métodos amigables para obtener atributos de los diferentes componentes de OpenStack que de forma normal sería tedioso el obtenerlos.

4.4. Net2Plan: NFV Management Plugin

Para llevar a cabo este proyecto, era necesario integrar las APIs mencionadas anteriormente con una herramienta que tenga funcionalidad de planificación de redes. Por ello, se ha desarrollado una extensión de Net2Plan basada en el plugin Network Design para llevar a cabo la prueba de concepto (ver 5).

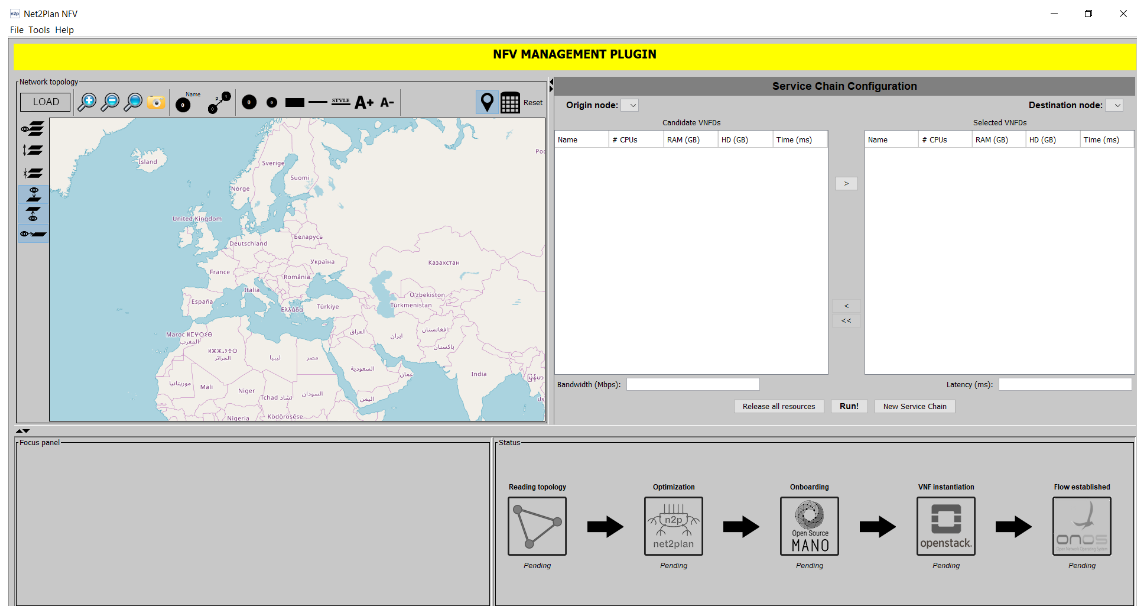

FIGURA 4.5: Interfaz gráfica del Plugin NFV-Management

En la figura 4.5 se puede observar la interfaz gráfica del Plugin NFV-Management, la cual está dividida en diferentes secciones:

- Arriba a la izquierda se encuentra el *TopologyPanel*, que se encarga de dibujar la topología deseada. Esta funcionalidad es heredada del *Plugin Network Design* de Net2Plan.

- Arriba a la derecha se encuentra el *OSMPanel*, que se encarga de obtener información sobre los distintos NSD que se encuentran disponibles en OSM y mostrarla al usuario de una manera amigable, informándole de que recursos (HD, RAM, CPU) son necesarios para su instanciación en un VIM.

  Así mismo, también incluye botones para realizar la ejecución de la prueba de concepto, así como el borrar todos los VNFs instanciados en OSM.

- Abajo a la izquierda se encuentra el *FocusPanel*, que se encarga de mostrar información detallada de un elemento en concreto cuando se selecciona. Dicha funcionalidad es heredada del *Plugin Network Design* de Net2Plan.
- Abajo a la derecha se encuentra el *ServiceChainPanel*, que se encarga de mostrar la fase en la que se encuentra el transcurso de la prueba de concepto.

  Hay cinco fases:

  - La primera indica que se ha cargado la topología de ONOS correctamente.
  - La segunda indica que el algoritmo de NFV se ha ejecutado correctamente.
  - La tercera indica que los VNFs se han instanciado correctamente en OSM.
  - La cuarta muestra cuando OpenStack ha creado las máquinas virtuales asociadas a los VNFs instanciados.
  - La quinta fase indica que se han instalado las reglas de flujo en ONOS y la *Service Chain* se satisface correctamente.

Capítulo 5

Prueba de concepto

En este capítulo se va a hablar de una pequeña prueba de concepto que consiste en un escenario SDN-NFV donde todas las APIs y herramientas mencionadas en los capítulos 3 y 4 trabajan conjuntamente en total sintonía.

Inicialmente, se habla sobre el contexto en el que se ha enmarcado la prueba de concepto, haciendo especial referencia al Proyecto H2020 Metro-Haul[28].

A continuación, se establece una arquitectura de desarrollo, en la que se define el papel de cada API y herramienta, y como interactúan entre sí.

Por último, se realiza una explicación del funcionamiento de la prueba de concepto, así como una vista de los resultados finales.

5.1. Contexto

Esta prueba de concepto está enmarcada dentro del proyecto europeo H2020 Metro-Haul[28]. Su principal objetivo es el de diseñar arquitecturas de redes metro que sean accesibles para 5G, anticipándose a los posibles problemas futuros cuando 5G esté totalmente funcional en las redes de telecomunicación.

Estas nuevas arquitecturas aseguran que algunos parámetros de calidad de servicio, como la latencia o el *jitter* sean lo más bajos posibles, así como una total integración de nuevas tecnologías referentes al campo de ICT, como son SDN y NFV.

Para conseguir todos estos objetivos, Metro-Haul busca integrar diferentes herramientas para diseñar nuevas arquitecturas de red. Esta prueba de concepto se planteó como una demostración de como diferentes herramientas podían operar en sintonía para disminuir la latencia al establecer diferentes *Service Chains*, anticipando la llegada del 5G.[29]

Dicha demostración tuvo éxito y se publicó en el congreso internacional European Conference on Optical Communication (ECOC)[30] en Septiembre de 2018.

5.2. Arquitectura

El principal objetivo de la prueba de concepto es el de integrar todas las APIs y herramientas mencionadas anteriormente para conseguir satisfacer diferentes *Service Chains*

con restricciones de latencia y ancho de banda. Además, se busca demostrar como todas las herramientas *open-source* que componen dicha prueba de concepto trabajan en sintonía para crear un escenario SDN-NFV heterogéneo.

Para conseguir los objetivos mencionados anteriormente, se ha diseñado una arquitectura específica para poder llevar a cabo la prueba de concepto, donde cada uno de los elementos realiza una tarea en concreto.

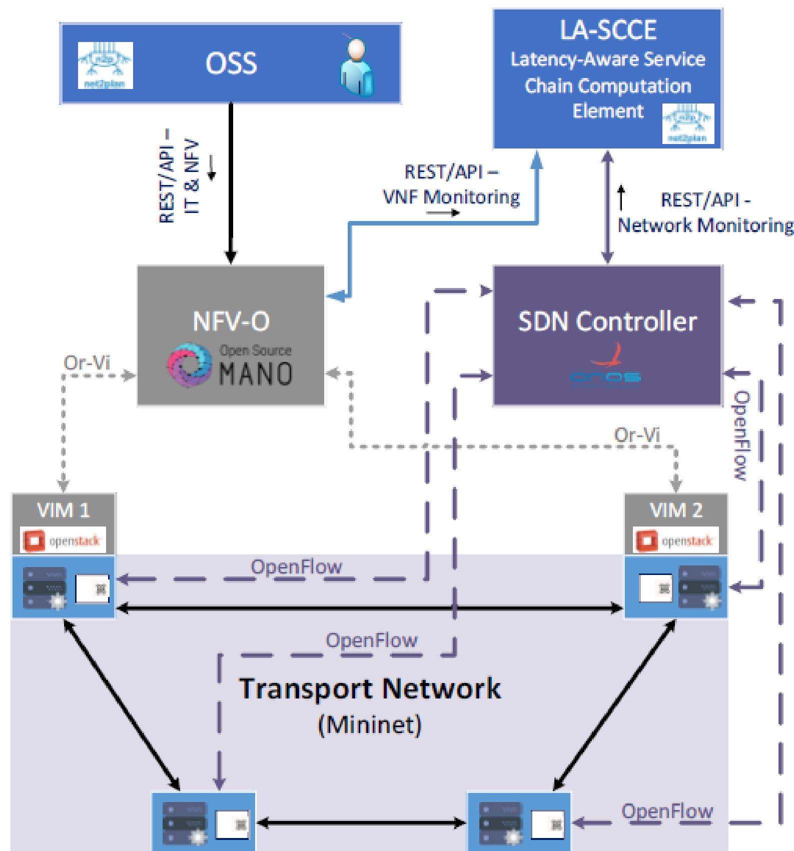

FIGURA 5.1: Arquitectura de la Prueba de Concepto. Fuente:[29]

En la figura 5.1 se puede observar un esquema de la arquitectura, en el que se incluyen todos los elementos que la componen, y hace más fácil de entender como interactúan entre sí los componentes.

A continuación vemos una explicación de cada uno de los elementos que componen la prueba de concepto y que componente lleva a cabo esa acción:

- **Operative Support System (OSS):** representa el papel de un operador que despliega un servicio gracias a una aplicación. El operador es emulado mediante la GUI de Net2Plan, más concretamente por su plugin de *NFV Management* (ver 4.4).
- **NFV Orchestrator (NFV-O):** representa el papel de una aplicación que se encarga de gestionar la infraestructura de virtualización necesaria para instanciar diferentes máquinas virtuales. OSM (ver 3.4) es el encargado de dicha función.

- **VIMs:** son los encargados de instanciar y alojar las diferentes máquinas virtuales pertenecientes a los VNF. OpenStack (ver 3.5) es quien realiza este papel.
- **Red de Transporte:** la red de transporte es emulada mediante Mininet (ver 3.2) para establecer flujos de paquetes entre las diferentes VNFs de una *Service Chain*. La red se define mediante un script en Python (ver anexo A).
- **Controlador SDN:** la red de transporte es controlada por una instancia de ONOS (ver 3.3) mediante el envio de paquetes Openflow (ver 2.1.2) a los diferentes switches de la red.
- **Latency-Aware Service Chain Computation Element (LA-SCCE):** Se encarga de decidir el camino a seguir para atravesar una secuencia de VNFs que cumpla con los requisitos de latencia máxima. Este papel lo representa un algoritmo programado en Net2Plan para resolver de forma conjunta tanto el camino como el emplazamiento de las VNFs óptimos.

5.3. Funcionamiento

Para llevar a cabo la prueba de concepto, la arquitectura explicada en la sección anterior se ha traducido en un *testbed* para poder llevarla a cabo, como se puede apreciar en la figura 5.2.

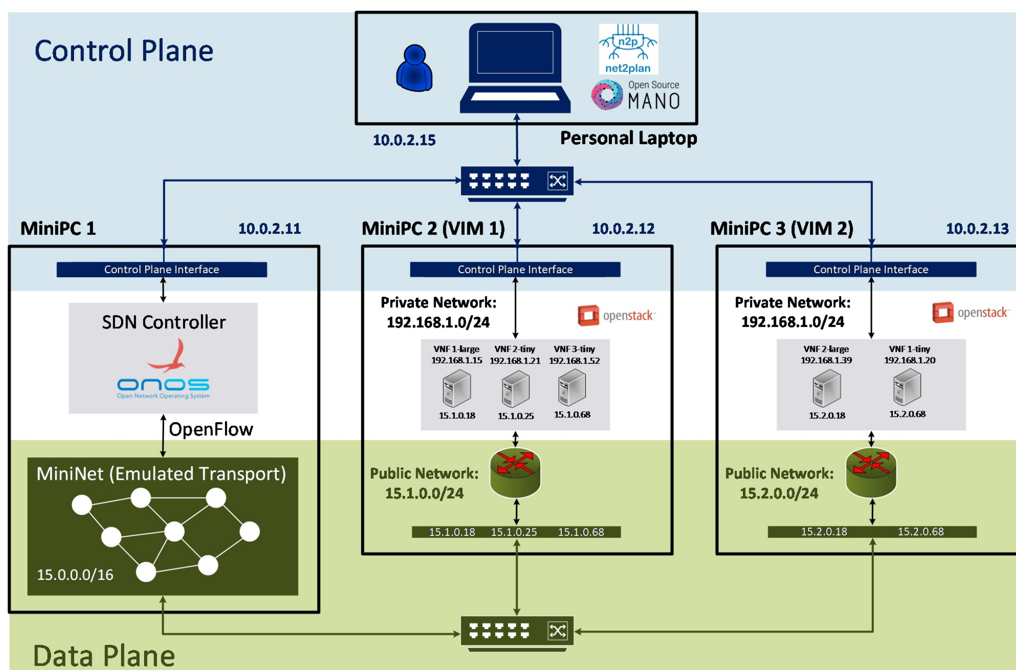

FIGURA 5.2: *Testbed* para la Prueba de Concepto. Fuente:[29]

El *testbed* considerado se compone de cuatro Personal Computers (PCs) y dos *switches*, uno para el plano de control y otro para el plano de datos:

- **MiniPC1:** Este PC se encarga de alojar una instancia del controlador SDN ONOS y de emular la red de transporte gracias a Mininet.
- **MiniPC2:** En este PC se encuentra corriendo una instancia de OpenStack, que actúa como VIM 1.
- **MiniPC3:** En este PC se encuentra corriendo una instancia de OpenStack, que actúa como VIM 2.
- ***Personal Laptop*:** Este dispositivo actúa como entidad central del *testbed*. En él se encuentra corriendo una instancia de OSM y el plugin *NFV Management* de Net2Plan.

Una vez explicado el *testbed*, se explican los diferentes pasos que se realizan para llevar a cabo la prueba de concepto y que APIs intervienen en cada uno de ellos:

- **Paso 1.** Una vez que ONOS, OSM y los VIMs estan listos y corriendo, se arranca el plugin de Net2Plan para comenzar con la demostración.

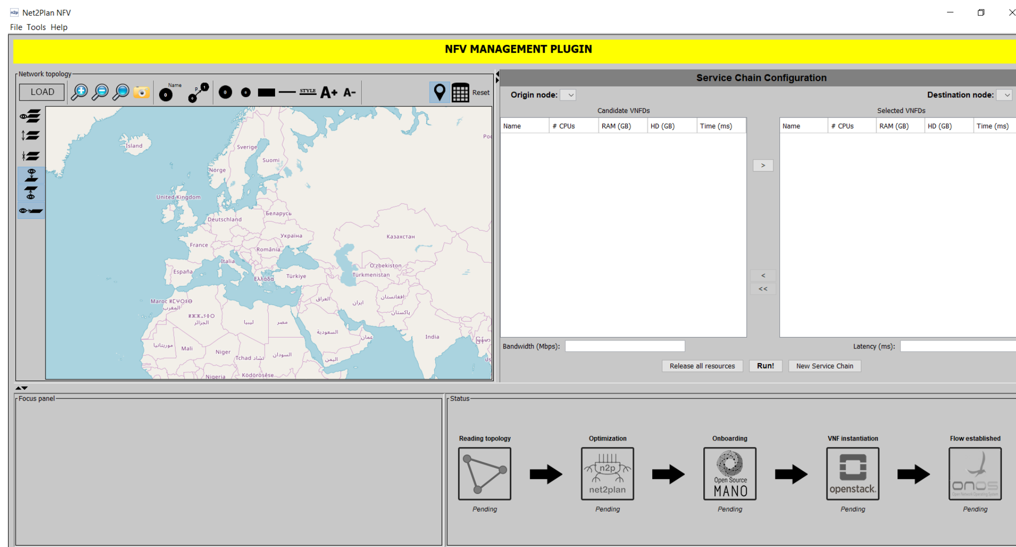

FIGURA 5.3: Interfaz gráfica del Plugin al inicio

- **Paso 2.** Haciendo click en el botón LOAD, Net2Plan recibe la información referente a la red de transporte de ONOS haciendo uso de J-ONOSClient, la información sobre los posibles VNFs a instanciar en OSM haciendo uso de J-OSMClient y la información sobre cada VIM de OpenStack haciendo uso de J-OpenStackClient.
- **Paso 3.** El usuario define la *Service Chain* que se quiere satisfacer (nodo origen, nodo destino, secuencia ordenada de VNFs a atravesar, latencia máxima y ancho de banda) a través de la interfaz gráfica del Plugin.

- **Paso 4.** Net2Plan recibe la información introducida por el usuario y la transfiere al LA-SCCE para que ejecute el algoritmo que devolverá como resultado una ruta de enlaces para la *Service Chain* y el VIM donde cada VNF debe ser instanciado.

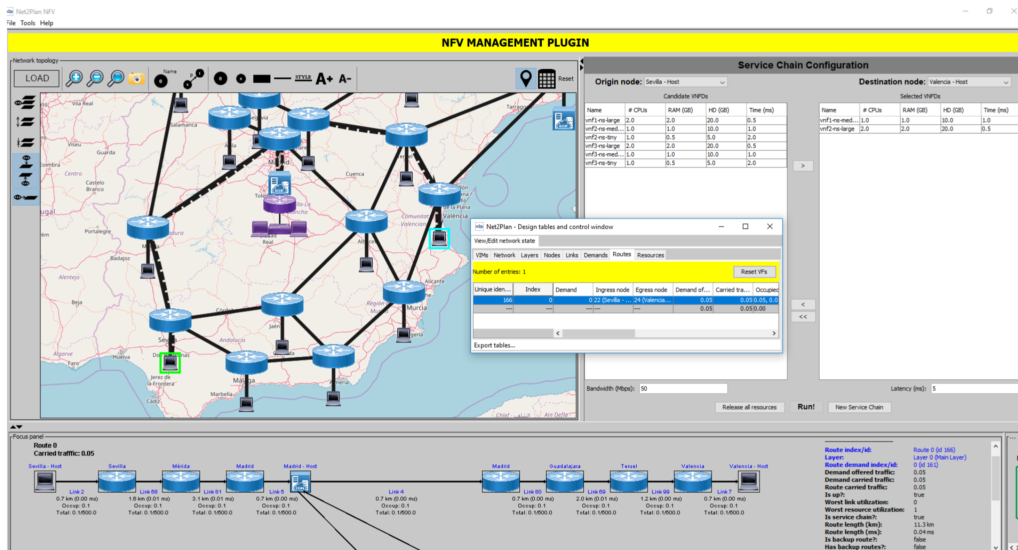

FIGURA 5.4: Interfaz gráfica del Plugin con la ruta establecida

En la figura 5.4 se apreciar la ruta obtenida por el algoritmo en la GUI de Net2Plan.

- **Paso 5.** Net2Plan envía la orden a OSM haciendo uso de J-OSMClient de instanciar los distintos VNFs en los VIMs que el LA-SCCE obtuvo como óptimos.

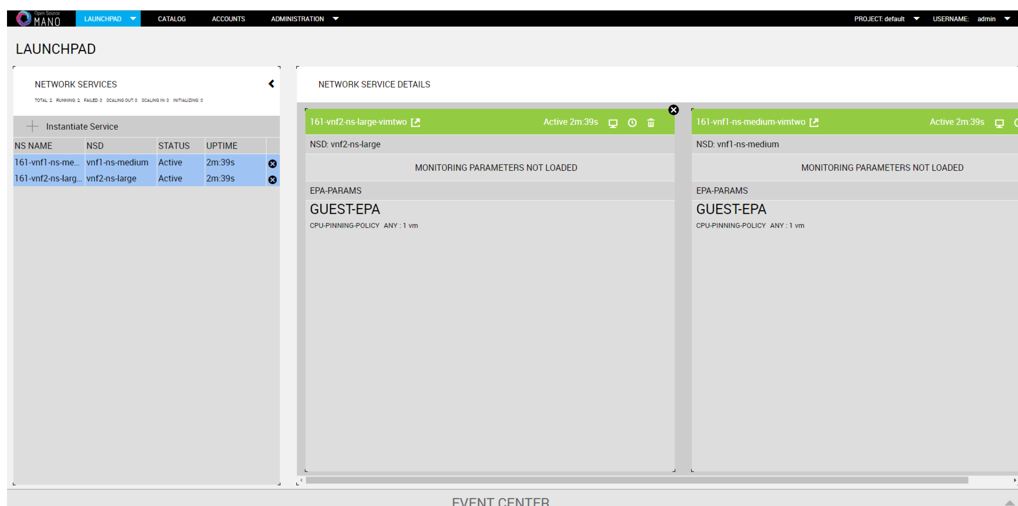

FIGURA 5.5: Interfaz gráfica de OSM con los VNFs instanciados

En la figura 5.5 se puede apreciar como OSM ha instanciados los VNFs en los VIMs según los cálculos del algoritmo.

- **Paso 6.** OSM envia órdenes a los diferentes VIMs para que alojen las diferentes máquinas virtuales correspondientes a los VNFs. La comunicacion entre OSM y OpenStack es transparente al usuario.
- **Paso 7.** Net2Plan envía la orden a ONOS, haciendo uso de J-ONOSClient, con diferentes reglas de flujo para establecer en los diferentes *switches* de la red de transporte, todo según la ruta óptima obtenida por el LA-SCCE.

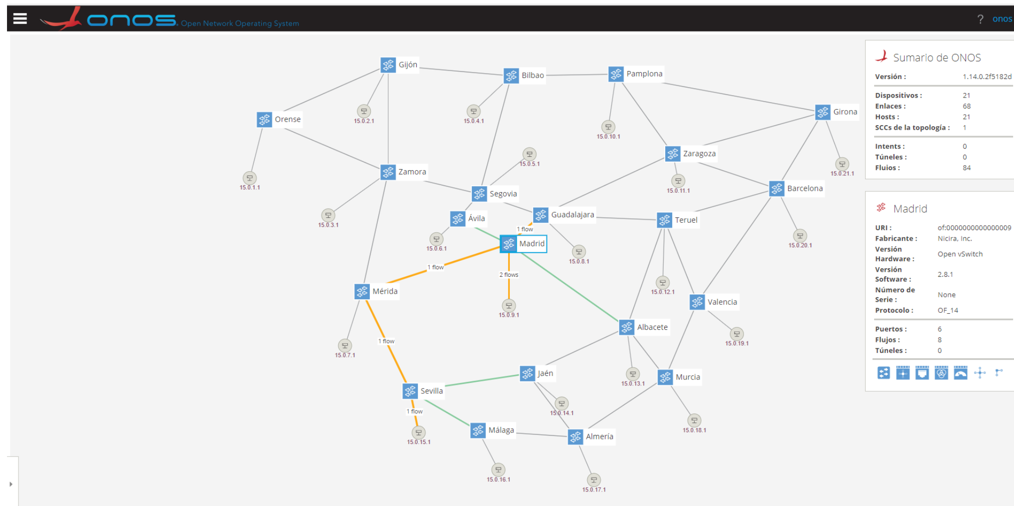

FIGURA 5.6: Interfaz gráfica de ONOS con las reglas de flujo establecidas

En la figura 5.6 se puede apreciar como la ruta calculada en Net2Plan se traslada a ONOS.

- **Paso 8.** Una vez establecidas las reglas de flujo en los diferentes *switches* mediante OpenFlow, se realiza una prueba de conexión para asegurar que la *Service Chain* está establecida y se satisface correctamente.

onos Open Network Operating System

%title_flows%of:0000000000000009 (8 %total%)

%STATE%	%PACKETS%	%DURATION%	%PRIORITY%	%TABLENAME%	%SELECTOR%	%TREATMENT%	%APPNAME%
Added	206	3.385	5	0	ETH_TYPE:ipv4	imm[OUTPUT:CONTROLLER], cleared:true	*core
Added	4.372	3.385	40000	0	ETH_TYPE:bddp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Added	4.372	3.385	40000	0	ETH_TYPE:lldp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Added	829	3.385	40000	0	ETH_TYPE:arp	imm[OUTPUT:CONTROLLER], cleared:true	*core
Added	21	1.094	25000	0	IN_PORT:4, ETH_DST:D2:7B:DD:51:83:41, ETH_SRC:02:F1:44:55:31:05	imm[OUTPUT:1], cleared:false	*fwd
Added	104	1.094	25000	0	IN_PORT:3, ETH_DST:D2:7B:DD:51:83:41, ETH_SRC:DE:E1:5C:2B:97:98	imm[OUTPUT:1], cleared:false	*fwd
Added	104	1.094	25000	0	IN_PORT:1, ETH_DST:DE:E1:5C:2B:97:98, ETH_SRC:D2:7B:DD:51:83:41	imm[OUTPUT:3], cleared:false	*fwd
Added	21	1.094	25000	0	IN_PORT:1, ETH_DST:02:F1:44:55:31:05, ETH_SRC:D2:7B:DD:51:83:41	imm[OUTPUT:4], cleared:false	*fwd

FIGURA 5.7: Prueba de conectividad en la GUI de ONOS

En la figura 5.7 se puede apreciar la instalación de reglas de flujo en ONOS que sirven para satisfacer la *Service Chain*.

Tras llevar a cabo la prueba de concepto, hay que mencionar los resultados obtenidos. Una vez instanciados los VNFs correspondientes a la *Service Chain* y habiendo instalado las reglas de flujo correspondientes a la ruta óptima obtenida en los *switches*, se realizan una prueba de conexión enviando un ping entre el origen y el destino.

En la figuras 5.4 y 5.6 se puede observar como la ruta es la misma en ambos casos (en el plugin de Net2Plan y en ONOS), y en la figura 5.7 como las reglas de flujo aplicadas indican que han procesado un paquete, que corresponde al ping realizado anteriormente para validad la conectividad.

Capítulo 6

Conclusiones

El objetivo inicialmente propuesto fue el desarrollo de diferentes APIs para comunicar diferentes herramientas *open-source* entre sí para poder diseñar un escenario SDN-NFV heterogéneo.

Una vez establecidos los objetivos, se comenzó con el desarrollo de J-OSMClient, J-ONOSClient y J-OpenStackClient para poder establecer la comunicación con OSM, ONOS y OpenStack, gracias a las APIs REST que exportan cada uno de ellos.

Una vez los clientes estuvieron totalmente funcionales, se comenzó con el desarrollo del plugin *NFV Management*, para convertir a Net2Plan en la entidad central que permitierá comunicar a las diferentes herramientas entre sí y hacerlas trabajar en sintonía.

Cuando el plugin estuvo totalmente desarrollado, se diseño una prueba de concepto para ser presentada en el congreso ECOC en Septiembre de 2018 para demostrar como las diferentes herramientas trabajaban conjuntamente para satisfacer diferentes *Service Chains* con requisitos de latencia y ancho de banda, anticipándo la llegada del 5G.

6.1. Fortalezas

Inicialmente, se pretendía desarrollar un conjunto de APIs para poder diseñar un entorno heterogéneo donde las tecnologías SDN y NFV cumplieran un papel fundamental en él, mediante diferentes herramientas que siguen dichas tecnologías operando en total sintonía.

Una vez acabado el proyecto, cabe decir que la principal fortaleza de este Trabajo de Fin de Máster es la creación de clientes *open-source* que permiten tener interfaces de comunicación con distintos módulos SDN-NFV para proporcionar optimización a entornos de red heterogéneos, permitiendo reducir la latencia en el establecimiento de una *Service Chain*, algo básico en el contexto de 5G.

6.2. Análisis de resultados

Una vez acabado el proyecto y habiendo obtenido resultados para su correspondiente análisis, se puede observar como, gracias a J-ONOSClient y a J-OpenStackClient, el plugin de Net2Plan puede obtener información exhaustiva sobre la red de transporte controlada por ONOS y de los recursos internos de los VIMs, permitiendo utilizar dicha

información como parámetros de entrada del algoritmo de planificación ejecutado por el LA-SCCE, lo que conlleva unos resultados más realistas.

En referencia a J-OSMClient, hay que mencionar que es el único cliente *open-source* programado en Java que existe para establecer comunicación con OSM. Aunque existe el cliente en Python desarrollado por la ETSI, dicho cliente no permite utilizarse de manera gráfica, debido a su naturaleza de CLI.

Por ello, la creación de J-OSMClient proporciona una amplio abanico de trabajo, permitiendo que OSM sea gestionado por una aplicación externa.

6.3. Líneas de trabajo futuro y mejoras

Aunque el objetivo de este proyecto se ha cumplido con creces, siempre se puede mejorar. Por ello, se proponen las siguientes líneas de trabajo futuro y mejoras:

- Emular una red de transporte multicapa (IP sobre WDM) para conseguir un escenario más realista. Para ello, evaluar herramientas como LINC-OE[31] o incluso utilizar agentes basados en modelos Yet Another Next Generation (YANG).
- Complementar la prueba de concepto con herramientas que sirvan para monitorizar el estado interno de los VIMs para obtener información interna con más nivel de detalle.
- Actualmente, J-ONOSClient y J-OpenStackClient se encuentran dentro del código del plugin *NFV Management*, y este se encuentra bajo un repositorio Git privado. Sería útil exportar ambos clientes a GitHub para que estuvieran disponibles para utilizar en cualquier otra herramienta para otros escenarios.

Anexo A

Script Mininet Red Transporte

```
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.link import Intf
from mininet.cli import CLI
from mininet.nodelib import NAT
import time

class RedEspana():

def __init__(self):

net = Mininet()

controller_onos = net.addController('controller',
controller = RemoteController, ip = '10.0.2.11')

numberElements = 21

switches = []
hosts = []
vimIndexes = [8,19]

for s in range(numberElements):
        switch = net.addSwitch('s'+str(s+1))
        switches.append(switch)


for h in range(numberElements):
        if h not in vimIndexes:
                host = net.addHost('h'+str(h+1),
                ip = '15.0.0.'+str(h+1+50)+'/24')
                hosts.append(host)
                net.addLink(host, switches[h])
```

```
net.addLink(switches[0], switches[1])
net.addLink(switches[0], switches[2])
net.addLink(switches[1], switches[2])
net.addLink(switches[1], switches[3])
net.addLink(switches[2], switches[4])
net.addLink(switches[2], switches[6])
net.addLink(switches[3], switches[4])
net.addLink(switches[3], switches[9])
net.addLink(switches[4], switches[5])
net.addLink(switches[4], switches[7])
net.addLink(switches[5], switches[8])
net.addLink(switches[6], switches[8])
net.addLink(switches[6], switches[14])
net.addLink(switches[7], switches[8])
net.addLink(switches[7], switches[10])
net.addLink(switches[7], switches[11])
net.addLink(switches[8], switches[12])
net.addLink(switches[9], switches[10])
net.addLink(switches[9], switches[20])
net.addLink(switches[10], switches[19])
net.addLink(switches[10], switches[20])
net.addLink(switches[11], switches[12])
net.addLink(switches[11], switches[18])
net.addLink(switches[11], switches[19])
net.addLink(switches[12], switches[13])
net.addLink(switches[12], switches[17])
net.addLink(switches[13], switches[14])
net.addLink(switches[13], switches[16])
net.addLink(switches[14], switches[15])
net.addLink(switches[15], switches[16])
net.addLink(switches[16], switches[17])
net.addLink(switches[17], switches[18])
net.addLink(switches[18], switches[19])
net.addLink(switches[19], switches[20])

intf_vimone = Intf('enx0050b6253bb0',switches[8])
intf_vimtwo = Intf('enx0050b6253baf',switches[19])

print('Added physical interfaces '+str(intf_vimone)+' and '+str(intf_vimtwo))

controller.start()

net.start()

time.sleep(3)
```

```
net.pingAll()

for host in hosts:
        host.cmd('ip route add 15.0.2.0/24 via 15.0.0.59')
        host.cmd('ip route add 15.0.3.0/24 via 15.0.0.70')
        host.cmd('/usr/sbin/sshd -D&')
        ping_vimone = host.cmd('ping 15.0.0.59 -c 1')
        print(str(ping_vimone))
        ping_vimtwo = host.cmd('ping 15.0.0.70 -c 1')
        print(str(ping_vimtwo))

CLI(net)



topos = { 'mytopo' : ( lambda : RedEspana() ) }
```

Bibliografía

[1] Software-Defined Networking [En línea].
Disponible: https://en.wikipedia.org/wiki/Software-defined_networking

[2] Open Networking Foundation. Software Defined Networking: The New Norm for Networks, White Paper, 2012

[3] OpenFlow [En línea].
Disponible: https://en.wikipedia.org/wiki/OpenFlow

[4] What is Network Service Chaining [En línea].
Disponible: https://www.sdxcentral.com/networking/virtualization/definitions/what-is-network-service-chaining/

[5] Network Function Virtualization [En línea].
Disponible: https://en.wikipedia.org/wiki/Network_function_virtualization

[6] Docker [En línea].
Disponible: https://www.docker.com/

[7] Open vSwitch [En línea].
Disponible: https://www.openvswitch.org/

[8] Cacti [En línea].
Disponible: https://www.cacti.net/

[9] Nagios [En línea].
Disponible: https://www.nagios.org/

[10] Net2Plan: The open-source network planner [En línea].
Disponible: http://www.net2plan.com/

[11] Mininet [En línea].
Disponible: http://mininet.org/overview/

[12] ONOS - A new carrier-grade SDN network operating system [En línea].
Disponible: https://onosproject.org/

[13] ONOS tutorials [En línea].
Disponible: http://sdnhub.org/tutorials/onos/

[14] ONOS Wiki [En línea].
Disponible: https://wiki.onosproject.org/

[15] OpenAPI Initiative [En línea].
Disponible: https://www.openapis.org/about

[16] Swagger [En línea].
Disponible: https://swagger.io/

[17] OSM [En línea].
Disponible: https://osm.etsi.org/

[18] ETSI [En línea].
Disponible: https://www.etsi.org/

[19] OSM Wiki [En línea].
Disponible: https://osm.etsi.org/wikipub

[20] OSMClient [En línea].
Disponible: https://osm.etsi.org/wikipub/index.php/OSM_client

[21] ETSI GS NFV-SOL 005 - V2.5.1.
White Paper

[22] OSM Release FIVE [En línea].
Disponible: https://osm.etsi.org/wikipub/index.php/OSM_Release_FIVE

[23] OpenStack [En línea].
Disponible: https://www.openstack.org/

[24] OpenStack4Java - Fluent OpenStack client for Java [En línea].
Disponible: http://www.openstack4j.com/

[25] J-OSM Client [En línea].
Disponible: https://github.com/girtel/J-OSMClient

[26] ONOS Java API - 1.15.0 [En línea].
Disponible: http://api.onosproject.org/1.15.0/apidocs/

[27] OpenStack4Java API - 3.0.5 [En línea].
Disponible: http://www.openstack4j.com/javadoc/

[28] Metro-Haul Project [En línea].
Disponible: https://metro-haul.eu/

[29] F.J. Moreno-Muro, *et. al*, "Latency-aware Optimization of Service Chain Allocation with joint VNF instantiation and SDN metro network control", ECOC, 2018

[30] ECOC [En línea].
Disponible: https://www.ecocexhibition.com/

[31] LINC-OE [En línea].
Disponible: https://github.com/FlowForwarding/LINC-Switch