UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Trabajo Fin de Máster

---

# Desarrollo de APIs en escenarios SDN-NFV

---

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

FECHA

| | |
|---|---|
| **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 en escenarios SDN-NFV |


**Resumen:**
POR HACER


| | |
|---|---|
| **Titulación:** | Máster en Ingeniería de Telecomunicación |
| **Departamento:** | Tecnologías de la Información y las Comunicaciones |
| **Fecha de presentación:** | FECHA |

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

## 1.1. Motivaciones

## 1.2. Objetivos

## 1.3. Estructura de la memoria

# Capítulo 2

# Estado del arte

En este capítulo se hablará sobre el contexto en el que se enmarca este proyecto.

En primer lugar, se hará una breve explicación de como las herramientas open-source ayudan a agilizar el desarrollo y la funcionalidad en una red de telecomunicaciones.

Por último, se hace especial mención a los dos paradigmas que motivan este proyecto: SDN y NFV.

## 2.1. 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.

## 2.2. SDN

### 2.2.1. OpenFlow

## 2.3. NFV

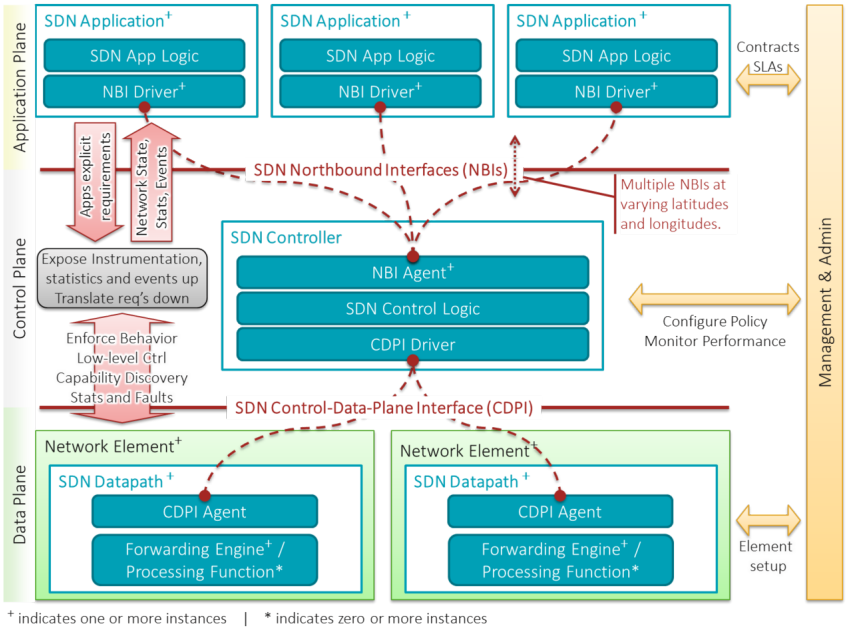


FIGURA 2.1: Arquitectura SDN

# 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 la herramienta open-source que ha servido como base para el desarollo de este proyecto..

Por último, se hablará de las diferentes herramientas/aplicaciones que se han utilizado para llevar a cabo este proyecto, como pueden ser ONOS, ETSI-OSM y OpenStack, y de las librerías JAVA que se han empleado para poder realizar la interacción entre todos los componentes.

## 3.1. Net2Plan

Net2Plan [**?** ] 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, ... Ésto 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, WDM o escenarios de NFV).

Net2Plan tiene dos modos de uso: mediante interfaz gráfica (GUI) y línea de comandos (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. Como se ha hablado antes, ambos modos permiten utilizar Net2Plan en el entorno educativo (investigación o enseñanza) y en el entorno de la industria y la empresa.

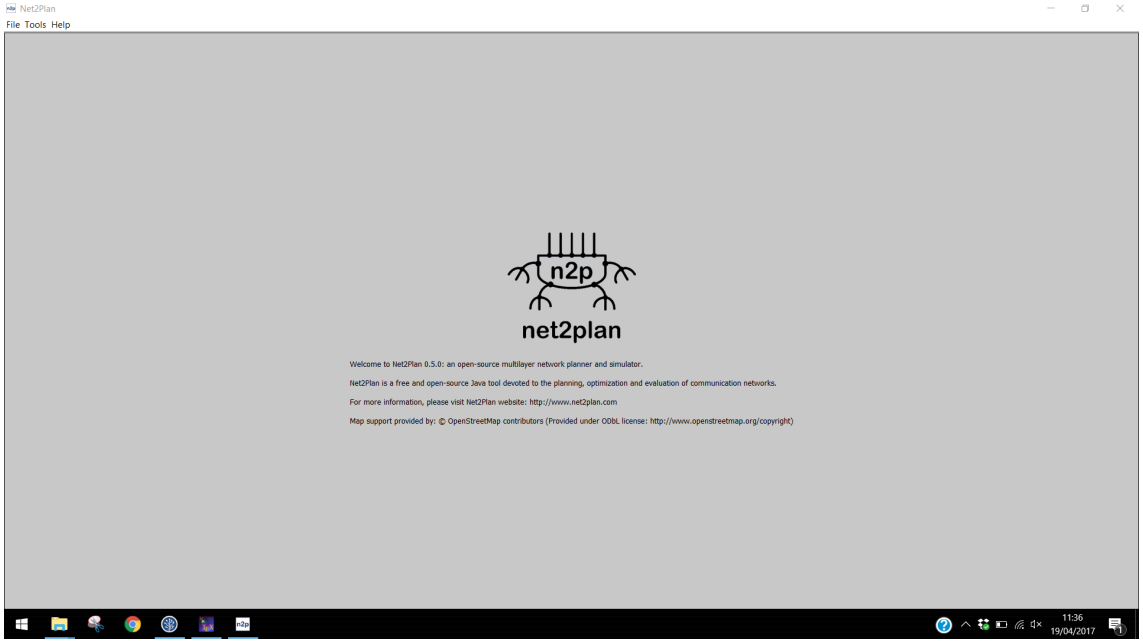


FIGURA 3.1: Ventana de inicio de Net2Plan

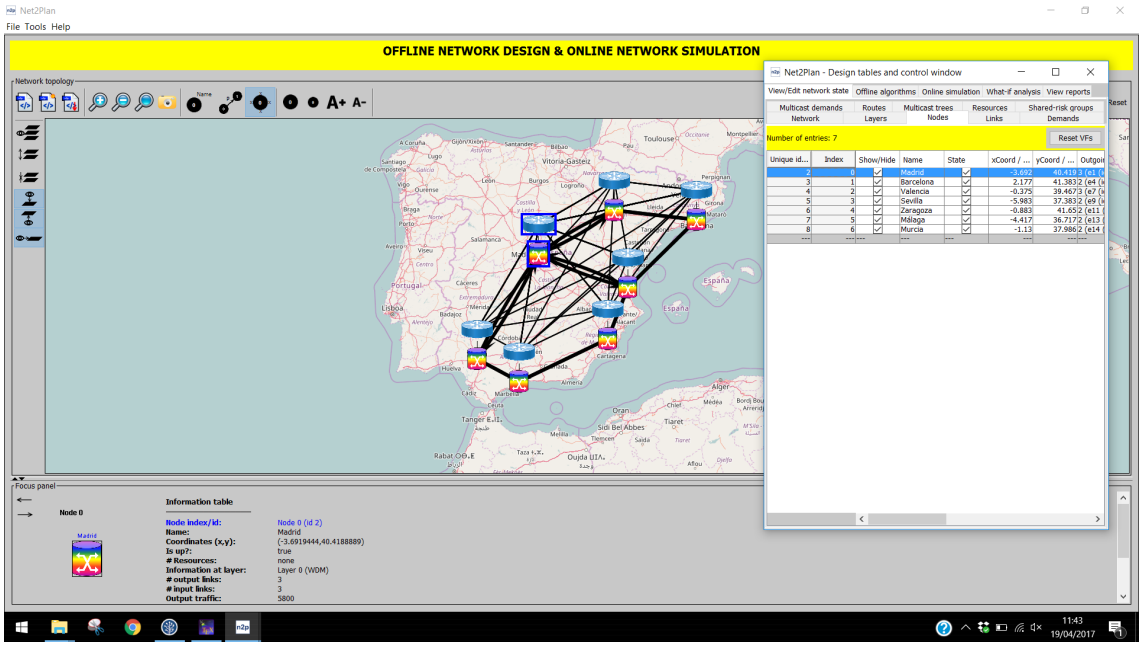


FIGURA 3.2: Ventana *Offline network desing and online network simulation*

## 3.2. Mininet

Mininet es una herramienta de emulación de redes que permite crear redes con *hosts*, *switches*, controladores y enlaces. Los *hosts* de Mininet corren bajo un sistema operativo Linux, mientras que los *switches* soportan el protocolo OpenFlow (ver 2.2.1) para mayor flexibilidad respecto a la configuración del *routing* y para integrarlos dentro de un escenario SDN (ver 2.2).

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 de 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 (Open Network Operative System) 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.
- Alto rendimiento: Se ajusta perfectamente a las especificaciones de los operadores de red.
- Resistencia: Provee la disponibilidad requerida por los operadores de red.
- 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 y, ahora también para dispositivos P4.
- Modularidad: Las funcionalidades de ONOS están definidas en modulos localizados, lo que hace más fácil probar y mantener el software en buen estado.

Está escrito en Java y opera como un clúster de nodos idénticos en cuanto al software. Trabaja con modelos y protocolos estandarizados, tales como OpenFlow (ver 2.2.1), NETCONF, OpenConfig, OpenROADM, ...

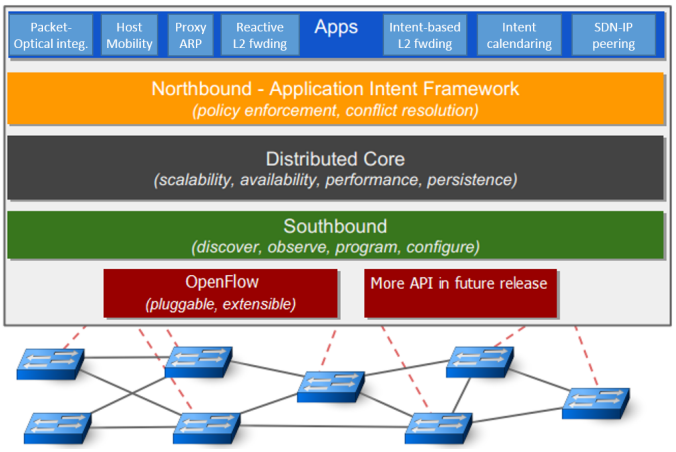


FIGURA 3.3: Arquitectura de ONOS. Fuente: http://sdnhub.org/tutorials/onos/

En la figura 3.3 se observa la arquitectura interna de ONOS. En el *Core* se encuentran los controladores de los diferentes servicios que ofrece (TopologyService, DeviceService, HostService, ...), 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 principalmente de una RestAPI.

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

Para facilitar la interacción con el usuario, ONOS ofrece una GUI (ver figura 3.4) 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.

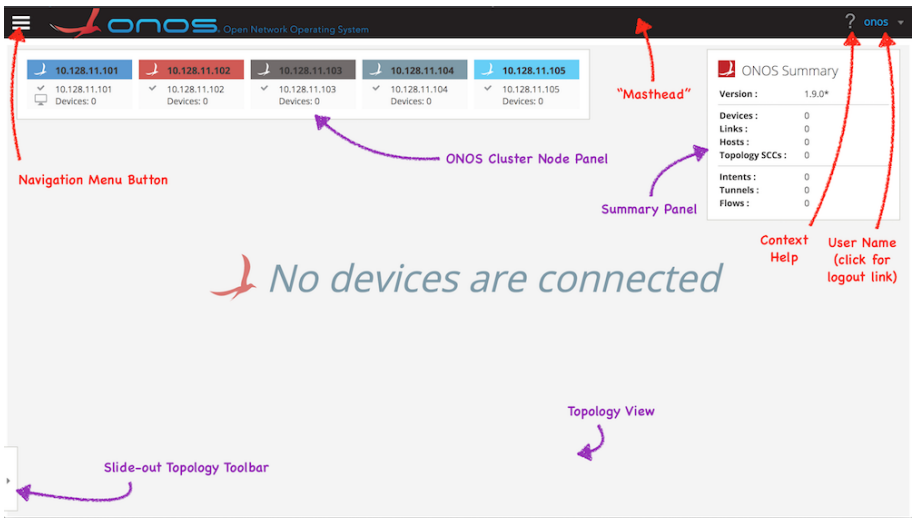


FIGURA 3.4: Interfaz Gráfica de ONOS. Fuente: https://wiki.onosproject.org

### 3.3.1. OpenAPI

OpenAPI es una iniciativa creada por varios expertos de la industria y la investigación para estandarizar las descripciones de las RestAPIs. Su principal objetivo es crear y promover un formato de descripción genérico.

Actualmente, prácticamente todas las aplicaciones utilizar APIs para conectarse con bases de datos, servicios y aplicaciones de terceros,... Gracias a la iniciativa de OpenAPI, las aplicaciones podrán conectarse entre sí de forma más rápida y sencilla, ayudando a tener un mundo realmente comunicado.

## 3.4. OSM

OSM (Open Source MANO) es un software *open-source* cuya función principal es la orquestación de servicios de red avanzados en infraestructuras NFV heterogéneas. Surge como iniciativa de la ETSI para crear una arquitectura NFV común para los operadores de red.

OSM trabaja con una serie de componentes y conceptos que ayudan a definir su arquitectura:

- **VDU (Virtual Deployment Unit):** Es el componente más básico de la arquitectura OSM. Se encarga de definir una máquina virtual.
- **VLD (Virtual Link Descriptor):** Es el componente que se encarga de definir las conexiones entre diferentes componentes de la arquitectura. Hay principamente dos tipos de VLD: VDU-VDU y VNF-VNF.

- **VNFD (Virtual Network Function Descriptor):** Es el componente que se encarga de definir los recursos necesitados para instanciar un VNF. Incluye diferentes componentes: lista de VDUs, lista de VLDs, lista de conexiones,...
- **NSD (Network Service Descriptor):** Es el componente que se encarga de definir la información sobre la configuración de un NS. Incluye diferentes componentes: lista de VNFDs, lista de VLDs, parámetros de configuración iniciales, ...
- **VNF (Virtual Network Function):** Es el componente que define una función de red virtualizada. Ésta puede ser completa, cuando es una función realizada únicamente por él, o parcial, cuando es una función mas compleja que requiere de otros VNFs.
- **NS (Network Service):** Es el componente que se encarga de agrupar diferentes VNFs que realizan una función de red conjunta.

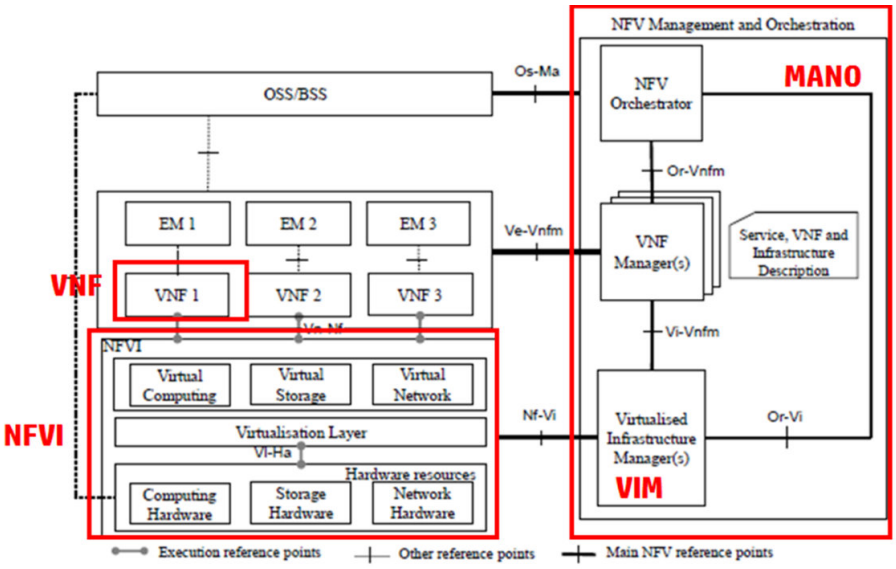


FIGURA 3.5: Arquitectura NFV de la ETSI. Fuente: https://sdn.ieee.org/newsletter/july-2016/opensource-mano

En la figura 3.5 se puede ver un esquema de la arquitectura NFV que propone la ETSI, en la que se puede observar el papel que juega OSM en ella.

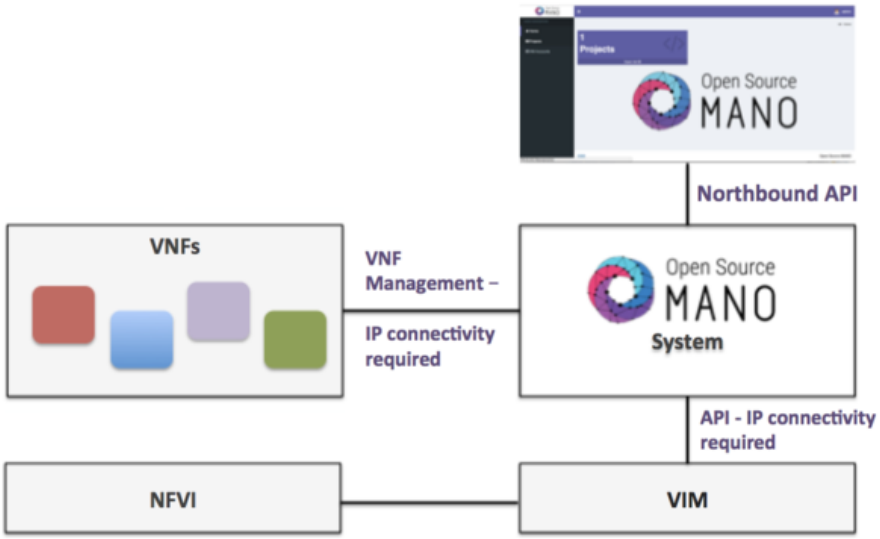


FIGURA 3.6: Arquitectura de OSM. Fuente: https://osm.etsi.org/wikipub

Para ayudar a explicar el funcionamiento de OSM, la figura 3.6 da una visión general de la interaccion entre OSM y los diferentes componentes:

- **Interfaz *NorthBound*:** OSM exporta una RestAPI gracias a su interfaz *NorthBound*. Mediante llamadas HTTP (GET, POST, 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 (OSMClient).

- **Conexión con VIM:** OSM permite la comunicación con múltiples tipos de VIM (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 RestAPI.

- **Conexión VIM-NFVI:** NFV (NFV Infrastructure) es el conjunto de recursos (Memoria RAM, número de CPUs, Memoria de almacenamiento, ...) que son utilizados por un VIM para poder instanciar diferentes máquinas virtuales (VNFs).

  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.

Una vez explicadas las interacciones que realiza OSM, se pueden explicar los pasos que sigue OSM para instanciar un nuevo NS en un determinado VIM:

- **Paso 1.** Mediante la interfaz gráfica de OSM u otro cliente, el usuario escoge el NSD que quiere instanciar y en que VIM quiere hacerlo.
- **Paso 2.** OSM se pone en contacto con el VIM para ver si tiene todos los recursos necesarios para poder instanciar el NS.
- **Paso 3.** El VIM se pone en contacto con su NFVI para verificar si tiene todos los recursos necesarios. En caso afirmativo, todo continua de forma normal. En caso contrario, el VIM avisará a OSM de que no es posible llevar a cabo la instanciación.
- **Paso 4.** Una vez reservados los recursos necesarios, el VIM empieza a instanciar el NS. Esto toma un tiempo, ya que el VIM crea una instancia por VNF y tiene que aplicar todos los parámetros de configuración definidos en los descriptores.
- **Paso 5.** Cuando el NS ha sido instanciado satisfactoriamente, el VIM envía a OSM un mensaje de OK y desde OSM ya se pueden gestionar los diferentes VNFs pertenecientes al NS.

## 3.5. OpenStack

OpenStack es una arquitectura basada en el paradigma **Cloud Computing** para controlar y gestionar grandes cantidades de recursos de computación, almacenamiento y red a través de un **Datacenter**. Para facilitar la gestión de recursos, OpenStack provee al usuario una interfaz gráfica a la vez que también exporta una RestAPI para permitir conectividad con aplicaciones externas.

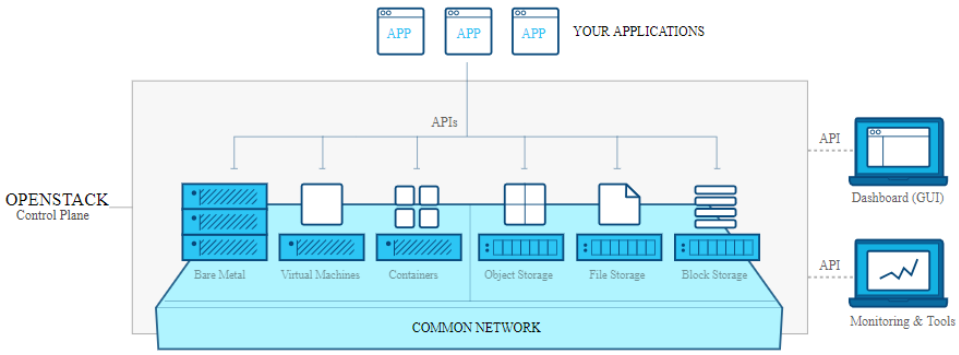


FIGURA 3.7: Arquitectura de OpenStack. Fuente: https://www.openstack.org/software/

Está compuesto de diferentes servicios o bloques, encargándose cada uno de ellos de una funcionalidad concreta dentro de la arquitectura. Los servicios principales de OpenStack son los siguientes:

- **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 que cada componente desplegado en OpenStack se comunique con los demas y estén interrelacionados.
- **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 discos de la infraestructura, para hacer más dinámicas la accesibilidad y la disponibilidad.

### 3.5.1. OpenStack4j

OpenStack4j es una librería REST *open-source* programada en Java para controlar y gestionar un sistema basado en OpenStack.

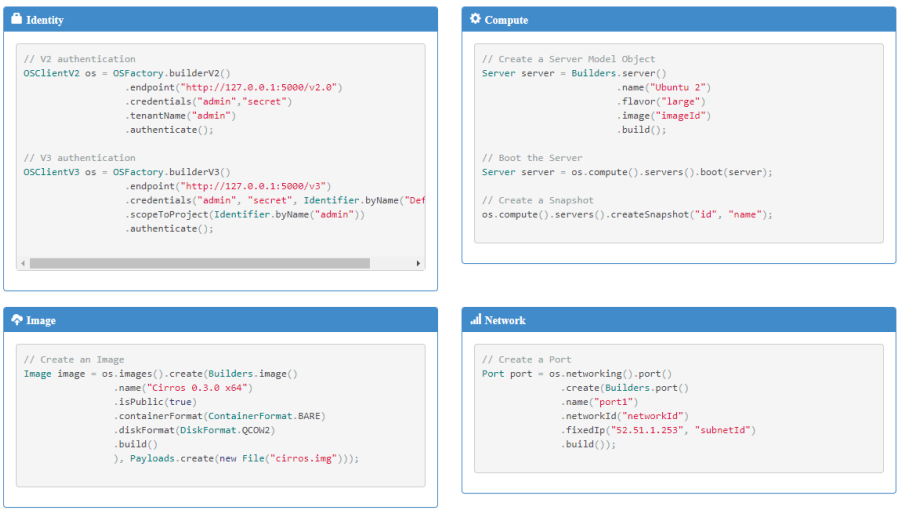


FIGURA 3.8: Ejemplo de uso de OpenStack4j. Fuente: http://www.openstack4j.com/

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.

En la figura 3.8 se puede ver un breve ejemplo de como utilizar los servicios Identity, Compute, Image y Network.

# Capítulo 4

# Desarrollo de APIs

## 4.1. ONOS Client

## 4.2. J-OSM Client

## 4.3. OpenStack Client

## 4.4. Net2Plan: SDN/NFV Management Plugin

En la figura **??** se puede observar la vista que ofrece el plugin de Net2Plan desarrollado para la prueba de concepto.

# Capítulo 5

# Prueba de concepto

En este capítulo se va a hablar de una pequeña prueba de concepto para ver el funcionamiento de todas las APIs y herramientas conjuntamente.

Inicialmente, se establece una arquitectura de desarrollo, en la que se define el papel de cada API y herramienta.

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

## 5.1. Arquitectura

La arquitectura de la demostración muestra los diferentes elementos que la componen, así como las interacciones entre ellos para el intercambio de datos.

A continuación vemos una explicación de cada elemento perteneciente a la demo y que componente lleva a cabo esa acción:

- **Operation Support System (OSS):** representa el papel de un operador que despliega un servicio gracias a una aplicación. El operador es emulado mediante Net2Plan, más concretamente por su plugin de *NFV Management* (ver sección 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. ETSI-OSM (ver 3.4) es el encargado de dicha función.
- **Virtual Infrastructure Managers (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.
- **Controlador SDN:** la red de transporte es controlada por una instancia de ONOS (ver 3.3) mediante el envio de paquetes Openflow (ver 2.2.1) 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 la extensión de Net2Plan mediante la ejecución de un algoritmo de *NFV Placement*.

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

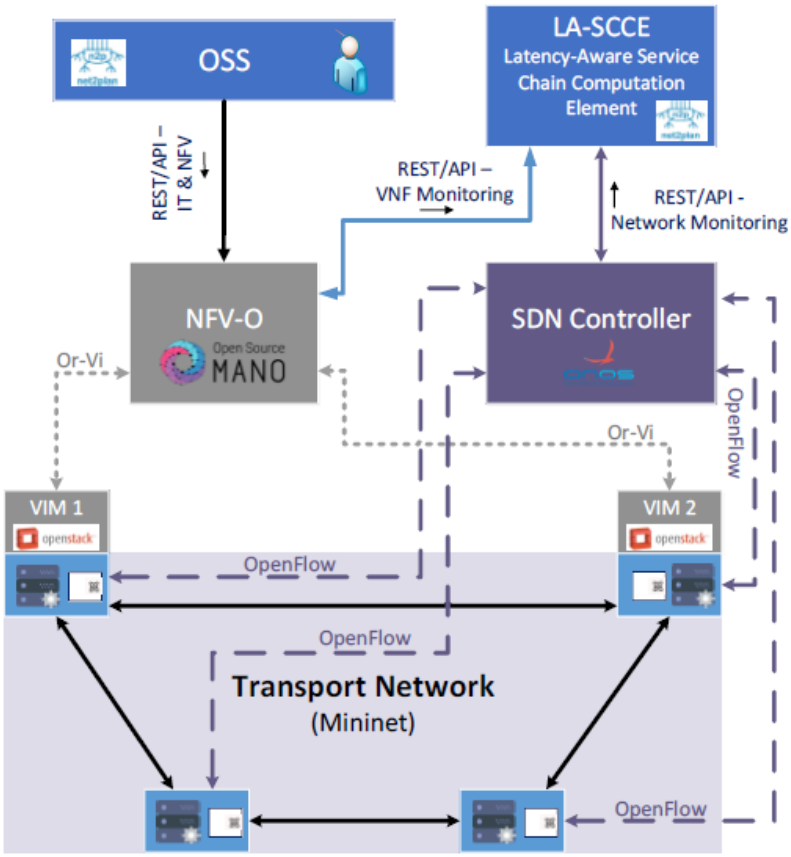


FIGURA 5.1: Arquitectura de la Prueba de Concepto

## 5.2. Desarrollo

- Paso 1. Haciendo click en el botón LOAD, Net2Plan recibe la información referente a la red de transporte de ONOS, la información sobre los posibles VNFs a instanciar de ETSI-OSM y la información sobre cada VIM de OpenStack, todo mediante llamadas a sus respectivas Rest-APIs.
- Paso 2. El usuario define la Service Chain que quiere satisfacer (nodo origen, nodo destino, secuencia ordenada de VNFs a atravesar, latecia máxima y ancho de banda) a través de la interfaz gráfica del Plugin.
- Paso 3. 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 una serie de VNFs instanciadas en diferentes VIMs.

- Paso 4. Net2Plan envía la orden a ETSI-OSM de instanciar las VNFs en los VIMs que el LA-SCCE obtuvo como óptimos.
- Paso 5. ETSI-OSM envia órdenes a los diferentes VIMs (OpenStack) para que alojen las diferentes máquinas virtuales correspondientes a los VNFs.
- Paso 6. Net2Plan envía la orden a ONOS con diferentes reglas de flujo para establecer en los diferentes switches de la red de transporte, todo según lo obtenido por el LA-SCCE.
- Paso 7. Una vez establecidos las reglas de flujo mediante OpenFlow, se realiza una prueba de conexión para asegurar que la Service Chain está establecida.

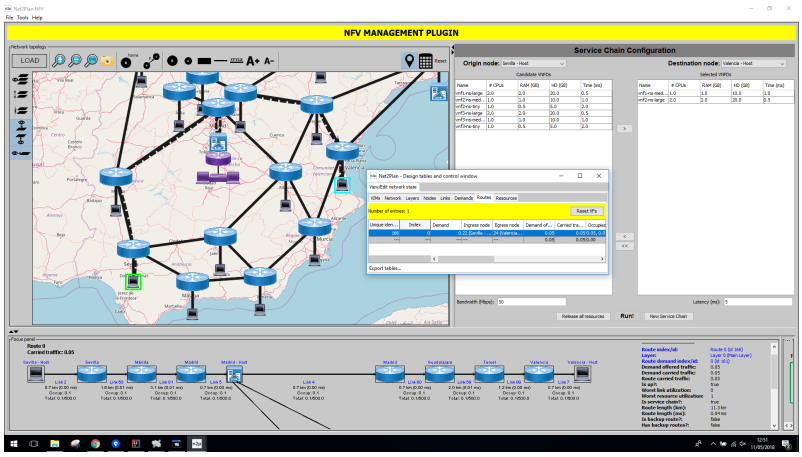


FIGURA 5.2: Estado final de la prueba de concepto

# Capítulo 6

# Conclusiones

# 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() ) }
```