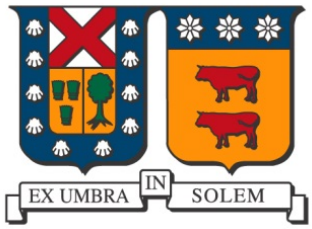
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE




# "LABORATORIO DE SOFTWARE-DEFINED NETWORKING UTILIZANDO MININET"

FELIPE ALEJANDRO PIZARRO MÁRQUEZ

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: JAVIER HERNÁN CAÑAS ROBLES
Profesor Correferente: HORST VON BRAND SKOPNIK

Noviembre - 2019

# DEDICATORIA

A mis padres Juan e Ivonne, que han sido el apoyo fundamental durante toda vida y me han permitido llegar hasta aquí.

Los quiero mucho.

# AGRADECIMIENTOS

A mi profesores y ayudantes que tuve durante mi estadía en la universidad, especialmente a mi profesor guía, Javier Cañas. Gracias por darme la oportunidad de aprender sobre este tema.

A mi hermano Francisco, que me motivó a investigar y siempre estuvo dispuesto a resolver mis dudas que surgían mientras desarrollaba la memoria, eres un capo.

Al resto de mi familia, mis padres Juan e Ivonne, mi hermano Juan, mi cuñada Sofía y sobrino Matías, gracias por aguantarme todos estos años.

A mi amigo Feño, que fue uno de los pocos que se dignó a leer la memoria luego de que la terminara...

A mi perro Canito, que mientras trabajaba en esta memoria durante las frías noches de invierno se acostaba en mi cama y la dejaba calentita, el guatero perfecto.

Y, finalmente, a todas las personas con las que compartí durante mi vida universitaria.

Me gustaría seguir escribiendo, pero soy una persona de pocas palabras y, como diría mi padre, *"ya está todo dicho"*.

# RESUMEN

**Resumen**— *Software-Defined Networking* (SDN) se presenta como un nuevo paradigma con el objetivo de simplificar la creación y gestión de redes. Esta arquitectura reemplaza el modelo tradicional de red distribuido por uno más centralizado, en donde se separa físicamente la capa de control y datos, mediante la introducción de un elemento lógicamente centralizado, denominado controlador. El objetivo de la memoria es implementar un laboratorio de SDN accesible, mediante *software* de código abierto, o, en su defecto, gratuito. Para cumplir con este requisito, se utilizó el emulador de red *Mininet*, junto con el controlador *OpenDaylight* (ODL), ambos de código libre. A lo largo del documento se describe paso a paso cómo implementar un entorno de pruebas para SDN utilizando sólo máquinas virtuales. La memoria culmina con la realización de dos laboratorios que se fundamentan en las herramientas previamente mencionadas. El primero trata sobre cómo programar flujos en *switches* virtualizados con *Mininet*, mediante el protocolo *OpenFlow*. El segundo, en cambio, se enfoca en la creación de redes *overlay* virtual sobre una red *underlay* física, implementada en *Mininet*, en base a flujos programados por el controlador ODL usando el módulo VTN.

**Palabras Clave**— *Software-Defined Networking*, *OpenDaylight*, *OpenFlow*, Virtualización, Experimentación, Emulación, Accesibilidad, Redes *overlay*.

# ABSTRACT

***Abstract***— *Software-Defined Networking* (SDN) is presented as a new paradigm which simplifies the creation and management of networks. This architecture replaces the traditional distributed network model with a centralized one, in which the control and data layers are physically separated, introducing a logically centralized element, called controller. The objective of this paper is to implement an accessible SDN laboratory, through open-source or free software. To accomplish this requisite, *Mininet*, a network emulator, and *OpenDaylight* (ODL), a SDN controller, were used, both open-source. Across this project, a step by step tutorial of how to implement a complete SDN testbed using only virtual machines is described. To culminate this paper, two experiment that utilize the tools mentioned early are implemented. The first one is about programing flow entries in *Mininet*'s virtual switches with the *OpenFlow* protocol. The focus of the second, however, is to create a virtual overlay network over a physical underlay network, implemented on *Mininet*, based on flow entries programed by the ODL controller using the VTN module.

***Keywords***— *Software-Defined Networking*, *OpenDaylight*, *OpenFlow*, Virtualization, Experimentation, Emulation, Accessibility, Overlay networks.

# GLOSARIO

ACL: *Access Control List*
ARP: *Address Resolution Protocol*
API: *Application Programming Interface*
BDDP: *Broadcast Domain Discovery Protocol*
BGP: *Border Gateway Protocol*
CCVPN: *Cross Domain Cross Layer VPN*
DCAN: *Devolved Control of ATM Networks*
DNS: *Domain Name Servers*
DPID: *Datapath-Id*
DSCP: *Differentiated Services Code Point*
FIB: *Forwarding Information Base*
GRE: *Generic Routing Encapsulation*
GSMP: *General Switch Management Protocol*
IaaS: *Infraestructure as a Service*
IDS: *Intrusion Detection Systems*
IETF: *Internet Engineering Task Force*
IP: *Internet Protocol*
IPS: *Intrusion Prevention Systems*
ISP: *Internet Service Provider*
JVM: *Java Virtual Machine*
L1: *Layer 1*
L2: *Layer 2*
L3: *Layer 3*
L4: *Layer 4*
LISP: *Locator ID Separation Protocol*
LLDP: *Link Layer Discovery Protocol*
ML2:*Modular Layer 2*
MPLS: *Multiprotocol Label Switching*
NaaS: *Network as a Service*
NAT: *Network Address Translation*
NETCONF: *Network Configuration Protocol*
NFV: *Network Function Virtualization*
ODL: *OpenDaylight*
ONAP: *Open Network Automation Platform*
ONS: *Open Networking Summit*
ONF: *Open Networking Foundation*
OSI: *Open System Interconnection*
OSGi: *Open Service Gateway Interface*
OSPF: *Open Shortest Path First*
OVS: *Open vSwitch*
OVSDB: *Open vSwitch Database*

QoS: *Quality of Service*
REST: *Representational State Transfer*
SSH: *Secure Shell*
STP: *Spanning Tree Protocol*
SNMP: *Simple Network Management Protocol*
TCAM: *Ternary Content-Addressable Memory*
TCP: *Transmission Control Protocol*
TLS: *Transport Layer Security*
TTL: *Time To Live*
VLAN: *Virtual Local Area Network*
VM: *Virtual Machine*
VRF: *Virtual Routing Forwarding*
VTN: *Virtual Tenant Network*
VPN: *Virtual Private Network*
VXLAN: *Virtual Extensible LAN*
WAN: *Wide Area Network*
XML: *eXtensible Markup Language*
YANG: *Yet Another Next Generation*

# ÍNDICE DE CONTENIDOS

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# INTRODUCCIÓN

Las redes computacionales tradicionales se componen típicamente de *routers*, *switches*, servidores, *middleboxes*, servidores WEB, *firewalls*, balanceadores de carga, IPS, entre otros. Para procesar y gestionar la gran cantidad de datos que se envían a través de la red es imperioso que este proceso sea eficiente, confiable, flexible y robusto. Esto ha provocado que las compañías que manufacturan estos dispositivos de red implementen protocolos cada vez más pesados y complejos, con el solo propósito de poder cumplir con estas premisas.

Los administradores de red son responsables de configurar políticas que respondan a una amplia gama de eventos y aplicaciones. Muchas veces, su tarea consiste en transformar políticas de red de alto nivel en configuraciones de bajo nivel y, al mismo tiempo, adaptarse a los constantes cambios que sufren las redes que administran. Incluso, muchas veces deben realizar estas tareas complejas sin tener acceso a las herramientas más adecuadas. Como resultado, gestionar y optimizar la red se han vuelto un trabajo desafiante y propenso a errores. El hecho que los dispositivos de red tradicionales se comporten como cajas negras integradas verticalmente complejiza aún más esta tarea.

*Software-defined Networking* (SDN) es una arquitectura disruptiva que nace para contrarrestar este problema. En esta aproximación se reemplaza el modelo de red tradicional completamente distribuido por uno más centralizado. La definición formal entregada por [Open Networking Foundation, 2012] enuncia que *"Software-Defined Networking es una arquitectura emergente que es dinámica, manejable, económica y adaptable, haciéndola ideal para la naturaleza dinámica de las aplicaciones actuales que utilizan gran banda ancha. Esta arquitectura separa el plano de control y datos, permitiendo al plano de control ser directamente programable y aplicar una capa de abstracción a la infraestructura subyacente para las aplicaciones y servicios de red".* El plano de datos incluye todos los dispositivos de enrutamiento (*switches* y *routers*), mientras que el plano de control implementa un controlador lógicamente centralizado. Este paradigma permite a los administradores de red utilizar eficientemente todos sus recursos de una manera más centralizada y automatizada.

Grandes empresas como *Google*, *Facebook* y *Microsoft* ya han implementado esta arquitectura en sus *data-centers*. *Google*, en específico, utiliza una red SDN personalizada, llamada B4, para administrar una WAN que interconecta sus *data-centers* y *clusters*. Esta red SDN, construida sobre el protocolo OpenFlow, es capaz de llevar los *links* de la WAN al 70 % de su capacidad (aumentando alrededor de dos y tres veces el uso comparado con un enlace típico) y separar flujos de aplicaciones entre múltiples rutas, basándose en prioridades y demandas [Jain *et al.*, 2013].

Plataformas que proporcionan *infrastructure as a service* (IaaS), también han sido ampliamente beneficiadas con la introducción de SDN. Una IaaS permite a un operador

gestionar el poder de cómputo, almacenamiento, conexión de redes y otros recursos fundamentales, para ofrecer un servicio a un cliente final. *OpenStack* es una de las alterativas de código abierto que permiten implementar este tipo de plataformas. A través del componente *Neutron*, es posible integrar soluciones de SDN en *OpenStack*, logrando así la virtualización de *links*, *switches*, *routers* y otros elementos de red; y de esta manera crear redes virtuales para interconectar los recursos generados en la plataforma [QUITRAL, 2015].

Al año 2017 el valor de mercado de SDN ascendió a $6.600 millones de dólares y se estima que para el año 2021 tendrá un valor de $13.800 millones, lo que se es muestra del continuo aumento de interés por parte de la comunidad. En una encuesta realizada el año 2017 por [IDG Enterprise, 2017], se determinó que, de una muestra de 294 profesionales en el área de redes, el 49 % están considerando o llevando a cabo un programa piloto de SDN; mientras que un 18 % de los encuestados ya implementó SDN.

SDN dará que hablar en los años venideros, razón por la cual es necesario que los administradores de red deban familiarizarse y experimentar con esta tecnología. Lamentablemente, implementar un laboratorio físico de SDN usualmente es caro y no está al alcance de todas las personas. Afortunadamente, existen alternativas gratuitas, que serán mencionadas en la sección 2.6, y que permiten emular y/o simular una red SDN en un computador estándar, siendo *Mininet* la más reconocida. *Mininet* es un emulador de SDN de código abierto que incluye una colección de *host*, *switches*, controladores y enlaces virtuales, ampliamente utilizado por investigadores para experimentar y probar soluciones de SDN, debido a su flexibilidad, interactividad y escalabilidad[Lantz *et al.*, 2017].

# CAPÍTULO 1
# DEFINICIÓN DEL PROBLEMA

A medida que las redes tradicionales crecen, estas, inevitablemente, se hacen más complejas de manejar. Problemas como la dificultad y el alto nivel de experticia necesario a la hora de configurar redes complejas, que en muchos casos se componen de dispositivos de distintos fabricantes, hacen de SDN una alternativa tentadora. Por ejemplo, una de las ventajas que posee SDN por sobre a la administración de redes tradicional, a la hora de manejar redes complejas, es la programabilidad. Esta permite que una organización cualquiera pueda desarrollar o instalar alguna aplicación que se encargue de controlar sus redes, en base a algún comportamiento en específico. Entre las aplicaciones más comunes se encuentran, por ejemplo, herramientas de ingeniería de tráfico, seguridad, balanceo de cargas, QoS, *switching*, *routing*, monitoreo, etc. Sin embargo, nada impide el desarrollo de alguna nueva aplicación que no se haya pensado hasta el momento, lo que convierte a SDN en una tecnología en constante evolución. SDN proporciona otras ventajas por sobre la administración de redes tradicional, entre las cuales se encuentran:

- Reducción de costos usando dispositivos universales.
- Gestión unificada.
- Escalabilidad.
- Seguridad más granular.
- Reducción de costos operacionales.

La arquitectura SDN llegó para quedarse y muchas empresas apostarán por ella en los años venideros. En este punto se identificó el principal problema, no existen cursos ni laboratorios dedicados a enseñar esta tecnología en la UTFSM. Al no existir un ambiente en donde los estudiantes puedan aprender y familiarizarse con SDN, no tendrán la oportunidad de adquirir este conocimiento que en un futuro será indispensable para los administradores de sistemas. A su vez, si no se hace algo al respecto, las empresas no encontrarán egresados capacitados en este ámbito.

Desafortunadamente, implementar un laboratorio SDN con componentes físicos resultaría en una inversión bastante cara. Por ejemplo, una de las alternativas más baratas y básicas de un *switch OpenFlow*, corresponde al modelo *Zodiac FX*[1] de *Northbound Networks*, con un precio aproximado de 90 USD. Para implementar topologías más complejas, con múltiples *hosts*, dispositivos y servicios de red, un *switch* se hace insuficiente, por lo que, inevitablemente, la inversión se acrecentaría. A raíz de esto, es

[1] **Zodiax FX**: https://northboundnetworks.com/collections/zodiac-fx

necesario buscar alternativas que sean más accesibles para el estudiante promedio. Para cumplir con el requisito de accesibilidad, se necesita descartar el uso de *hardware* caro y especializado, en beneficio de la virtualización y emulación, en base a *software* de código abierto o, en su defecto, gratuito. No solo la inversión sería mínima, sino que, además, los entornos virtuales son escalables, lo que permite implementar topologías de cualquier tipo, ya sean simples o complejas.

Otro problema inherente a SDN es que no es un paradigma estandarizado. Existen múltiples formas de implementar una arquitectura SDN, con controladores y protocolos distintos. Esto conlleva a que la literatura asociada a esta materia sea abundante y poco profunda. Es importante definir claramente la manera en cómo se implementará SDN a lo largo de la memoria, por lo que hay que preocuparse de que la bibliografía sea filtrada y, conforme se estandariza, ajustarla como corresponda.

## 1.1. OBJETIVOS

### 1.1.1. OBJETIVO PRINCIPAL

Generar un ambiente virtual de experimentación de *Software-Defined Network*, para que los estudiantes puedan comprender teórica y experimentalmente SDN y llevarlo al aula, sin la necesidad de que el laboratorio posea *hardware* especializado. El entorno virtual debe ser implementado mediante máquinas virtuales, junto con *software* de código abierto o, en su defecto, gratuito, minimizando el uso de recursos físicos.

### 1.1.2. OBJETIVOS ESPECÍFICOS

1. Generar una guía bien definida y estructurada, junto con un repositorio con el *software* utilizado, para que cualquier persona pueda replicar los experimentos.
2. Crear un entorno simulado de SDN utilizado *OpenDaylight* como controlador principal y *Mininet* como emulador de una infraestructura de red SDN.
3. Analizar el comportamiento de *Virtual Tenant Network*, que corresponde a un módulo de *OpenDaylight* que permite implementar redes *overlay* sobre redes SDN físicas, mediante virtualización de funciones de red.

## 1.2. ESTRUCTURA DE LA MEMORIA

Esta memoria se encuentra dividida en las siguientes secciones:

- **Marco Conceptual**: En este capítulo se presentan los conocimientos necesarios para comprender el paradigma SDN y el protocolo *OpenFlow*. Además, se realiza una pequeña retrospectiva sobre las redes tradicionales y los problemas asociados a ellas que pueden ser mitigados por SDN.
- **Software**: En este capítulo se presenta y explican las características del *software* que será utilizado a lo largo de esta memoria. Específicamente, se le dará enfoque al controlador *OpenDaylight* y al emulador de red *Mininet*.
- **Configuración y Conceptos Básicos**: En este capítulo se explica cómo implementar el entorno que servirá como banco de pruebas virtual para los experimentos propuestos, utilizando el *software* presentado en la sección anterior.
- **Experimentación**: En este capítulo se presentan los experimentos realizados. El primero estará enfocado en *Mininet* y redireccionamiento de paquetes mediante el protocolo *OpenFlow*, mientras que el segundo tratará sobre *OpenDaylight* y *Virtual Tenant Network*.
- **Conclusiones**: En este capítulo se presentan las conclusiones generales y específicas, los problemas enfrentados a lo largo de la memoria y, finalmente, se expondrán ideas sobre posibles trabajos a futuro.
- **Anexos**: En este capítulo se presentan cuatro anexos complementarios al documento. Los anexos 1 y 2 proveen información adicional sobre el marco conceptual. Los anexos 3 y 4 contiene códigos *Python* que se utilizaron en los experimentos y respuestas a las consultas hechas a la API REST de *VTN Manager*, respectivamente.

# CAPÍTULO 2
# MARCO CONCEPTUAL

## 2.1. ARQUITECTURA DE UN SWITCH TRADICIONAL

Antes de hablar de *Software-Defined Networking* (SDN), es necesario entender, a grandes rasgos, cómo es la arquitectura de un dispositivo de red tradicional, específicamente de un *switch*, y así comprender sus diferencias con respecto a un *switch* compatible con SDN.

Una infraestructura de red típica se compone de un conjunto de nodos interconectados entre sí, tales como *routers*, *switches*, *switches* virtuales y un gran número de *middleboxes*. Al analizar estos dispositivos desde una perspectiva arquitectónica, es posible separar sus funciones en tres planos. En la figura 1 se muestra la arquitectura dentro un *switch* tradicional, que se separa en los planos de datos, control y gestión. Cada uno de estos planos es capaz de establecer una comunicación horizontal con entidades adyacentes dentro de una mismo topología y vertical entre los diferentes planos.

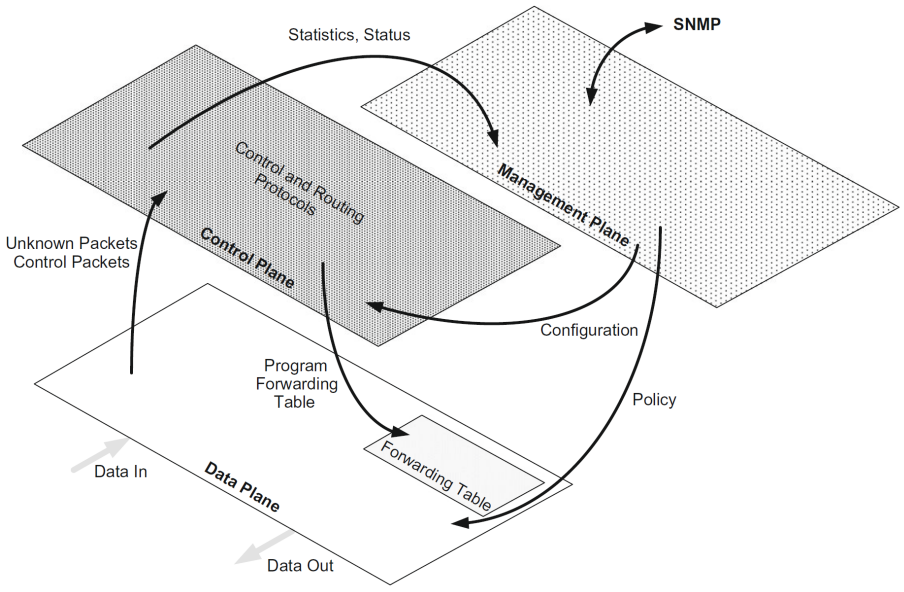



Figura 1: Arquitectura de un *switch* tradicional.
Fuente: [Toghraee, 2017].

El plano de datos consiste en varios puertos que son utilizados para la recepción y transmisión de paquetes y una *forwarding table* (FIB) con su lógica asociada. Este asume la responsabilidad de almacenar paquetes, modificar cabeceras y planificar el direccionamiento de paquetes. Si la información de la cabecera del paquete se encuentra en la FIB, este será redireccionado sin la intervención de los otros dos planos. No todos

los paquetes pueden ser manejados de esta manera ya la información de su cabecera puede que no exista en la FIB, por lo que debe ser procesado por el plano de control, el cual está involucrado en múltiples actividades. Su rol principal es mantener la información de la FIB actualizada para que el plano de datos pueda manejar de forma independiente el mayor número de tráfico posible. El plano de control es responsable de procesar diferentes protocolos de control que afectan el FIB, dependiendo de la configuración y del tipo de *switch*. Juntos, estos protocolos se encargan de gestionar de manera activa la topología de la red. El tercer plano representado en la figura 1 corresponde al plano de gestión. Los administradores de red lo utilizan para extraer información, monitorear y configurar el *switch*, mediante protocolo de gestión de red (como SNMP)[Toghraee, 2017].

### 2.1.1. LIMITACIONES DE LA ADMINISTRACIÓN DE RED TRADICIONAL

Las organizaciones constantemente se enfrentan a las limitaciones inherentes a la administración de redes tradicional, la cual está enfocada en el *hardware*. A continuación, se enuncian algunas que están plenamente identificadas[IP Knowledge, 2014]:

- **La configuración tradicional es lenta y propensa a errores:** Se necesitan muchos pasos a la hora agregar o quitar un dispositivo en una red tradicional. Primero, es necesario configurar manualmente múltiples dispositivos (*switches*, *routers*, *firewalls*) uno por uno. Luego, hay que usar herramientas de gestión a nivel de dispositivo para actualizar numerosas configuraciones, como, por ejemplo, ACLs, VLANs y QoS. Este tipo de configuración hace que sea mucho más complejo desplegar políticas de red consistentes. Como resultado, las organizaciones son más propensas a encontrar brechas de seguridad y sufrir de ineficiencias de rendimiento en sus redes. En síntesis, la molestia de administrar redes tan complejas interfiere con la capacidad de la red de alcanzar estándares empresariales.
- **Entornos con múltiples dispositivos requieren un nivel de experticia alto:** Comúnmente, las organizaciones poseen una variedad de dispositivos de red de diferentes proveedores. Cada uno de estos, usualmente, se comportan como cajas negras integradas verticalmente, lo que representa un desafío a la hora de configurarlos, ya que es necesario tener un conocimiento extensivo de cómo se administra cada uno.
- **La arquitectura tradicional complica la segmentación de la red:** Un aspecto que ha complicado el trabajo con redes en general es que se encuentra en constante evolución. Además de las *tablets*, PCs y *smartphones*, otros dispositivos inteligentes entrarán pronto al mercado. Esta explosión predecible está acompañada con un nuevo desafío para las organizaciones; la incorporación de todos estos dispositivos de diferentes proveedores dentro de su red de una manera segura y

estructurada. En muchas redes estos dispositivos se ubican en la misma *"zona"*. En caso de que la seguridad de uno de éstos se viera comprometida, este diseño riesgoso le podría otorgar acceso total a entidades externas, como *hackers* explotando la conectividad a la red de los dispositivos inteligentes, o fabricantes que pueden ingresar de manera remota a sus dispositivos. De igual manera, no existe una razón de peso para entregarles acceso total a los componentes de la red, sin embargo, implementar una segmentación en la red es proceso complejo, que fácilmente conlleva a errores.

## 2.2. SOFTWARE-DEFINED NETWORKING

El paradigma *Software-Defined Networking* (SDN) permite a los desarrolladores abstraerse de la topología de red subyacente y manejar los recursos de la red de manera sencilla, similar a como funciona la virtualización de un sistema operativo, en donde múltiples máquinas virtuales, que se ejecutan sobre un hipervisor, se abstraen del *hardware* físico, compartiendo los recursos entre sí. Como se aprecia en la comparativa de la figura 2, en SDN se separa el plano de control y datos, centralizando lógicamente la inteligencia de la red en controladores basados en *software* (plano de control), mientras que los nodos de la red se convierten en simples dispositivos de direccionamiento de paquetes (plano de datos), que pueden ser programados mediante interfaces abiertas, como, por ejemplo, SNMP, NETCONF y *OpenFlow*, siendo este el más utilizado.

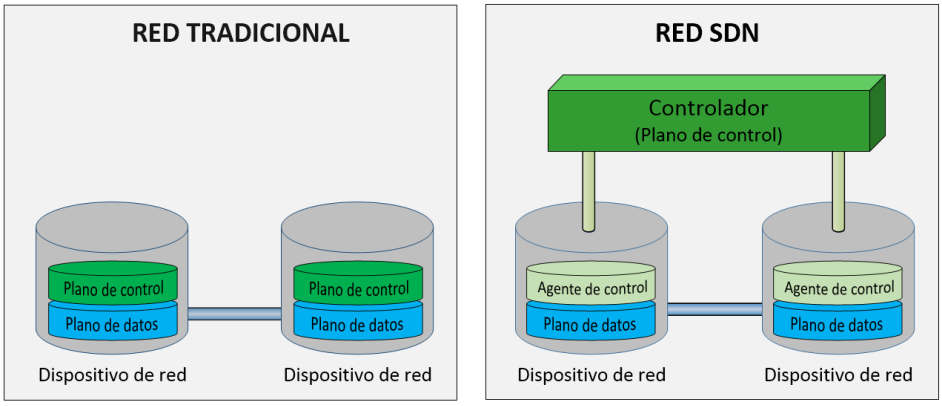



Figura 2: Comparativa entre redes tradicionales y redes SDN.
Fuente: [Álvarez, 2015].

Para complementar esta sección, se recomienda la lectura del anexo 6.1.1, en donde se mencionan algunos proyectos precursores del paradigma SDN.

### 2.2.1. VENTAJAS DE SDN

SDN brinda múltiples beneficios por sobre las redes tradicionales. A continuación, se enumeran algunos de ellos[Horwitz, 2018]:

1. **Redes centralizadas:** Al abstraer el plano de control y datos, SDN puede balancear la carga y distribuir el tráfico de manera eficiente, previniendo cuellos de botella en la red, lo que mejora el rendimiento de las aplicaciones.

2. **Automatización aumentada:** La automatización ayuda a crear entornos más predecibles y consistentes. Además, facilita la escalabilidad del entorno para el manejo de los picos y valles de tráfico en la red.

3. **Gestión integral de la infraestructura:** Permite gestionar las redes de programas que proporcionan *hardware* de aprovisionamiento (almacenamiento, servidores y redes), como *OpenStack*.

4. **Seguridad mejorada:** Mientras que la virtualización ha provocado que la gestión, en general, sea más compleja, un controlador centralizado permite distribuir consistentemente políticas de seguridad a través de la red empresarial.

5. **Reducción de costos operacionales:** Un control centralizado permite aprovechar de mejor manera el *hardware* de la red, optimizando la eficiencia operacional y reduciendo los costos operacionales.

6. **infraestructura compatible con la nube:** A medida que las compañías se mueven a la nube, su infraestructura necesita ser virtualizada y su gestión centralizada, con el fin de que los profesionales puedan desplegar servicios sobre ésta.

7. **Encadenamiento de servicios de red:** También conocido como *Service Function Chaining* (SFC), es una característica de SDN que facilita la conexión de servicios de red (*firewalls*, NAT, IPS, etc.) en una cadena virtual. Esta técnica es utilizada por los administradores de red para definir un catálogo de servicios interconectados, con características distintas, y ofrecerlos mediante una única conexión. Una de las principales ventajas de SFC es automatizar la manera en que las conexiones virtuales manejan los flujos de tráfico de los servicios conectados. Por ejemplo, un controlador SDN puede aplicar una cadena de servicios a distintos flujos de tráfico, dependiendo de la fuente, destino y tipo[SDxCentral, 2019].

### 2.2.2. ARQUITECTURA DE SDN

*Software-Defined Networking* facilita la innovación en la red basándose en cuatro principios fundamentales [Sisov, 2016]:

1. Los planos de control y de datos de la red están desacoplados.
2. Las decisiones de enrutamiento se basan en flujos, en vez de direcciones. Estos flujos se instalan en *switches* compatibles con *OpenFlow*.
3. La lógica de enrutamiento de red se abstrae del *hardware* y se implementa en una capa programable de *software*.
4. Se introduce un controlador para orquestar las decisiones de enrutamiento de la red.

De acuerdo con la figura 3, SDN puede ser representado en una arquitectura de tres capas:

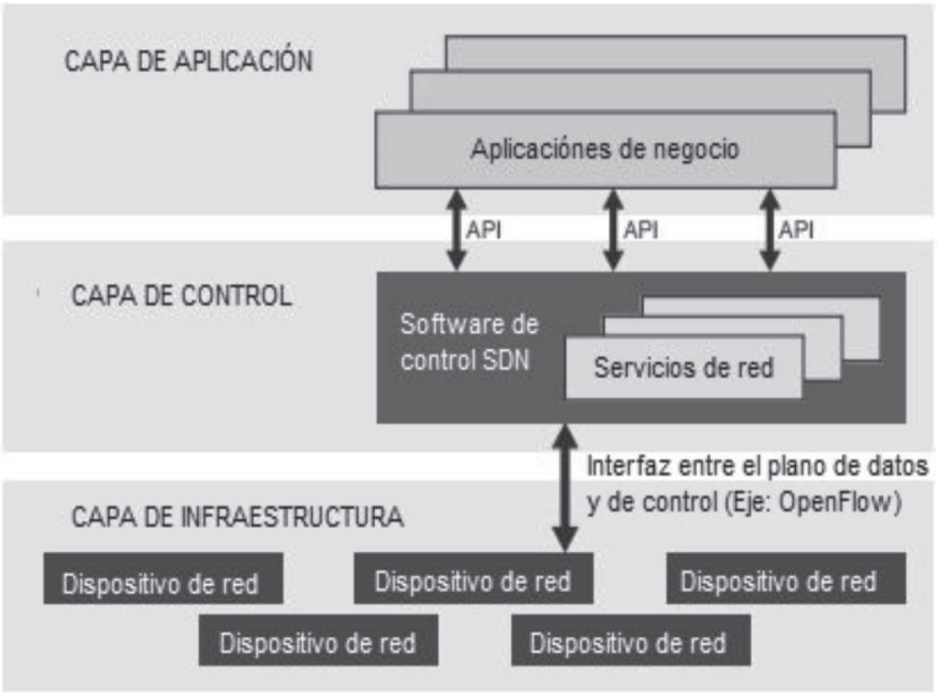



Figura 3: Arquitectura genérica de SDN.
Fuente: [Valencia *et al.*, 2015].

1. **Capa de infraestructura:** También conocida como capa de datos. Incluye todo lo relacionado con el *hardware* de enrutamiento, por ejemplo, *switches* y *routers*, que son configurados por el controlador mediante protocolos, como *OpenFlow*.

2. **Capa de control:** Consiste en un *software* de inteligencia de red que funciona como un controlador SDN lógicamente centralizado, el cual se instala en cualquier sistema operativo compatible. La capa de control administra el *hardware* de la capa de datos e instala las reglas de enrutamiento mediante APIs *southbound*.

3. **Capa de aplicación:** Consiste en un conjunto de aplicaciones y servicios que toman control sobre la capa de infraestructura mediante APIs REST, que se utilizan para personalizar el funcionamiento la red. SDN permite desarrollar fácilmente este tipo de aplicaciones las cuales, usualmente, son desplegadas en computadores separados o en la nube, y pueden ser usadas indistintamente por usuarios u otras aplicaciones.

### 2.2.3. INTERFACES DE SDN

Las APIs en la arquitectura SDN son usualmente llamadas interfaces *southbound* y *northbound*. La interfaz *southbound* se define como la conexión entre las capas de infraestructura y control, mientras que la interfaz *northbound* representa la conexión entre las capas de control y aplicación[Sisov, 2016].

#### 2.2.3.1. INTERFAZ NORTHBOUND

Las interfaces *northbound* son utilizadas por el plano de aplicación para comunicarse con el controlador. Son la parte más crítica en la arquitectura de un controlador SDN ya que el valor de SDN va de la mano con la calidad de las aplicaciones innovadoras que potencialmente podría soportar. Como son tan críticas, las APIs *northbound* deben soportar una amplia variedad de aplicaciones. El rol de estas interfaces es proveer una API de alto nivel entre el controlador y las aplicaciones, para que éstas puedan computar operaciones de red en base a eventos recopilados en el plano de control[Salman *et al.*, 2016].

Las aplicaciones de orquestación utilizan las APIs *northbound* para integrase con el sistema. El objetivo principal de la API es abstraer la infraestructura de red y permitir a los desarrolladores enfocar su atención en el desarrollo de aplicaciones, sin que estén obligados a entender como sus cambios pueden afectar a la red[SDX Central, 2018].

Típicamente, la arquitectura elegida para las APIs *northbound* es REST. Estas proporcionan flexibilidad, funcionalidades, y posibilitan cámbios sin interrumpir al cliente, mediante un mecanismo llamado *"navegación basada en hipertexto"*. En particular, una API REST consiste en una agrupación de diferentes servicios, que proporcionan los recursos REST, en una interfaz uniforme. En el caso de SDN, incluye los servicios de los planos de control y datos, como *switches*, *routers*, controladores, dispositivos NAT, *subnets*, etc.

#### 2.2.3.2. INTERFAZ SOUTHBOUND

Las APIs *southbound* permiten a un controlador SDN gestionar eficientemente la infraestructura de red y hacer cambios dinámicos en los dispositivos de la capa de datos de acuerdo con el tráfico en tiempo real, en base sus demandas y necesidades. Mientras que *OpenFlow* es el protocolo más conocido como interfaz *southbound*, no es el único disponible o en desarrollo, existiendo otras alternativas, tales como NETCONF[2], *OF-Config*[3], OVSDB[4] y *OpFlex*[5] (propietario de *Cisco*). Adicionalmente, hay controladores que dan soporte a protocolos como IS-IS[6], OSPF[7] y BGP[8] como interfaz *southbound* con el objetivo de soportar redes híbridas (SDN y no SDN), o aplicar *networking* tradicional de una manera definida por *software*[Salman *et al.*, 2016].

## 2.3. PROTOCOLO OPENFLOW

*OpenFlow* es la primera interfaz *southbound* estandarizada para SDN, y la más reconocida en la industria. Es un protocolo abierto que define la manera en que un controlador de SDN debe interactuar con el plano de datos para hacer ajustes a los dispositivos red y adaptarse a los requerimientos.

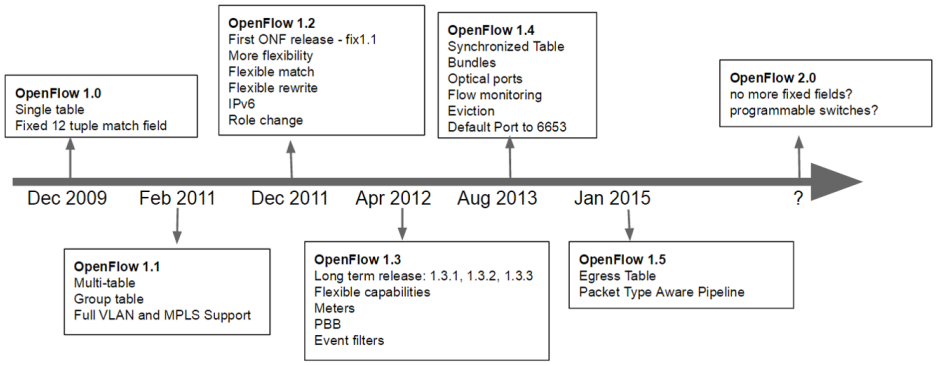



Figura 4: Evolución de *OpenFlow* a través del tiempo.
Fuente: [Ching-Hao *et al.*, 2015].

[2]**NETCONF**:https://www.tail-f.com/what-is-netconf/

[3]**OF-Config**:https://searchnetworking.techtarget.com/definition/OF-Config-OpenFlow-Configuration-and-Management-Protocol

[4]**OVSDB**:https://searchnetworking.techtarget.com/definition/OVSDB-Open-vSwitch-Database-Management-Protocol

[5]**OpFlex**:https://www.sdxcentral.com/networking/sdn/definitions/cisco-opflex/

[6]**IS-IS**:https://www.metaswitch.com/knowledge-center/reference/what-is-intermediate-system-to-intermediate-system-isis

[7]**OSPF**:https://www.ibm.com/support/knowledgecenter/es/ssw_ibm_i_71/rzajw/rzajwospf.htm

[8]**BGP**:https://www.cloudflare.com/learning/security/glossary/what-is-bgp/

La primera versión pública de *OpenFlow* fue lanzada en el año 2009 por la ONF[9], una organización impulsada por usuarios que se dedica a la promoción y adopción de SDN. Al año 2018, *OpenFlow* se encuentra en su versión 1.5.1 y trae bastantes mejoras con respecto a su versión inicial, observables en la figura 4.

## 2.4. SWITCH OPENFLOW

Un *switch* compatible con *OpenFlow* se denomina *"switch OpenFlow"*. Muchos fabricantes han anunciado o dan soporte para este tipo de dispositivos, entre ellos se encuentran *Cisco*, *Juniper*, *Big Switch Networks*, *Brocade*, *Arista*, *Extreme Networks*, IBM, *Dell*, *NoviFlow*, HP, NEC, etc. En esta sección se detallarán los conceptos relacionados directamente con la memoria, sin embargo, para profundizar en el tema se recomienda la lectura del anexo 6.1.2.

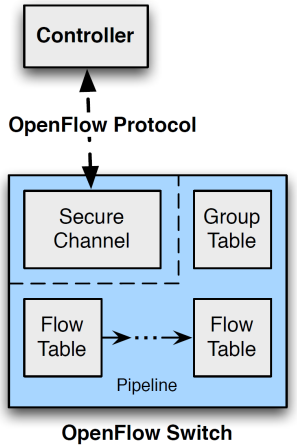



Figura 5: Arquitectura de un *switch OpenFlow* v1.3.0.
Fuente: [Open Networking Foundation, 2012].

En la figura 5 se muestra la arquitectura de un *switch OpenFlow* v1.3.0, caracterizada por cuatro componentes principales:

- **Puertos OpenFlow**: Interfaces de red que permiten distribuir paquetes a través de la red mediante procesamiento *OpenFlow*. Hay tres tipos, los físicos, los lógicos y los reservados.
- **Tabla de Flujos (Flow Table):** Tabla que permite búsqueda y redireccionamiento de paquetes, compuesta por entradas de flujo. Un *switch* puede contener una o muchas tablas de flujos.

[9]**Open Networking Foundation**: `https://www.opennetworking.org/`

- **Tabla de Agrupadores (Group Table):** Tabla que almacena agrupadores *OpenFlow*. Un agrupador *OpenFlow* es una capa de abstracción que facilita las operaciones de paquetes más complejas y especializadas, que no pueden ser realizadas por una entrada de flujo.
- **Canal seguro (Secure Channel):** Permite la comunicación segura entre el *switch* y un controlador mediante el protocolo *OpenFlow*. Generalmente, los mensajes que transitan a través de este canal suelen ser cifrados con TLS, sin embargo, también es posible utilizar directamente TCP asumiendo los riesgos de seguridad.

### 2.4.1. TIPOS DE SWITCH OPENFLOW

Existen dos tipos de *switch OpenFlow*:

- **OpenFlow-only**: Solo soportan operaciones *OpenFlow*, por lo que los paquetes solo pueden ser procesados a través del *processing pipeline* de *OpenFlow*.
- **OpenFlow-hybrid**: Soportan operaciones *OpenFlow* y operaciones normales de *switching*. Esto significa que pueden realizar operaciones del *pipeline* tradicional, como por ejemplo, *switching* de capa 2 (L2) tradicional, aislamiento VLAN y enrutamiento capa 3 (L3), utilizando la capa de control local del *switch*, mientras interactúa con el *OpenFlow pipeline* (ver 2.4.3).

### 2.4.2. PUERTOS OPENFLOW

Los puertos *OpenFlow* son interfaces de red usados para intercambiar paquetes entre un *switch OpenFlow* y el resto de la red. Los *switches* se conectan lógicamente entre ellos mediante puertos *OpenFlow*. Los paquetes son recibidos por un puerto de ingreso, procesados por el *processing pipeline* de *OpenFlow* y luego direccionados a través de un puerto de salida. El puerto de ingreso se utiliza cuando se hace *matching* de paquetes con entradas de flujo, mientras que puerto de salida se utiliza como acción *output* luego de hacer *matching*. Los conceptos como entradas de flujo, *matching* y *processing pipeline* son muy importantes en *OpenFlow* y serán explicados en secciones posteriores.

Un *switch OpenFlow* debe dar soporte tres tipos de puertos; físicos, lógicos y reservados. A su vez, se definen como puertos estándar a todos los que pueden ser utilizados indistintamente para ingreso y salida de paquetes.

- **Puertos Físicos**: Los puertos físicos corresponden a las interfaces del *hardware* del *switch* y no necesariamente los puertos físicos de OpenFlow correspondan a los puertos físicos del hardware, ya que es posible que algunos puertos esten deshabilitados para OpenFlow. Todos los puertos físicos son estándar.

- **Puertos Lógicos**: Los puertos lógicos no corresponden a las interfaces del *hardware* del *switch*. Son una capa de abstracción que puede ser definida sobre un *switch OpenFlow* utilizando métodos no *OpenFlow*, como, por ejemplo, *link aggregation*, túneles e interfaces *loopback*. Los puertos lógicos son trasparentes en el procesamiento *OpenFlow*, es decir, son tratados como puertos físicos. Todos los puertos lógicos son estándar.

- **Puertos Reservados**: Los puertos reservados especifican acciones de redireccionamiento genéricas, tales como el envío de paquetes al controlador, *flooding*, o el redireccionamiento utilizando métodos no *OpenFlow*, como el procesamiento *"normal"* del *switch*. No es obligatoria la implementación de todos los puertos y solo *LOCAL* es considerado como estándar.

  - **(Requerido) ALL**: Representa a todos los puertos a los cuales se le pueden redirigir paquetes y solo puede usarse como puerto de salida. En este caso, una copia del paquete es enviada a todos los puertos estándar, excepto al puerto de ingreso y los que estén bloqueados.
  - **(Requerido) CONTROLLER**: Representa al canal de control con el controlador *OpenFlow*.
  - **(Requerido) TABLE**: Representa el comienzo del *OpenFlow pipeline*.
  - **(Requerido) IN_PORT**: Representa al puerto en que ingresó el paquete. Solo puede ser utilizado como puerto de salida.
  - **(Requerido) ANY**: Un valor especial (*wildcard*) que es asignado por *OpenFlow* cuando no se especifíca ningun puerto en la entrada de flujo. No puede ser utilizado directamente ni como puerto de entrada ni de salida.
  - **(Opcional) LOCAL**: Permite a entidades remotas interactuar de forma directa con un *switch*, sin necesitar una red separada de control. Además, puede ser utilizado para implementar una conexión *in-band* con el controlador. Para más información sobre el uso de este puerto, se recomienda leer el anexo 6.2.1.

  Los siguientes puertos reservados solo pueden ser soportados por *switches OpenFlow-hybrid*:

  - **(Opcional) NORMAL**: Representa el procesamiento tradicional de un *switch*. Solo puede ser utilizado como puerto de salida.
  - **(Opcional) FLOOD**: Representa la acción *flooding* del procesamiento tradicional de un *switch*. Solo puede ser usado como puerto de salida y generalmente envía los paquetes a los puertos estándar, excepto al de ingreso y los que estén bloqueados.

### 2.4.3. PROCESSING PIPELINE DE OPENFLOW

El *processing pipeline* de *OpenFlow* define cómo los paquetes deben interactúan con las tablas de flujos de un *switch OpenFlow*. Estos dispositivos pueden contener muchas tablas de flujos, mientras que cada tabla contiene múltiples entradas de flujo. Un *switch OpenFlow* debe implementar, al menos, una de estas tablas.

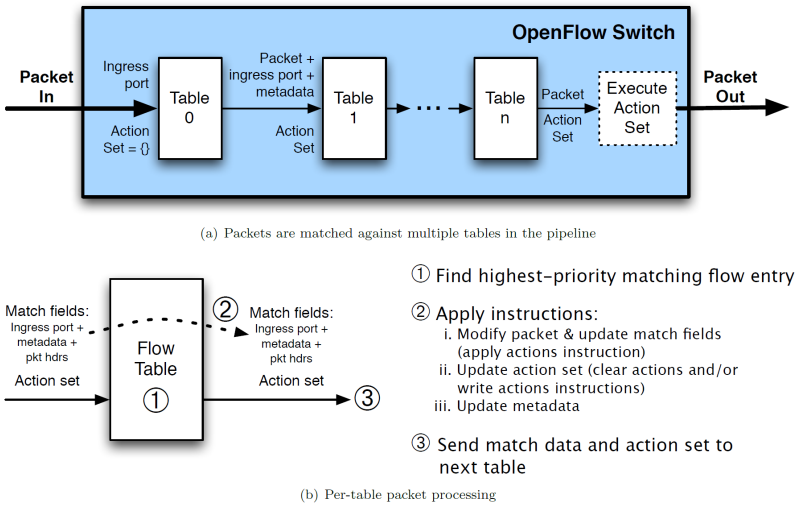



Figura 6: Flujo de paquetes a través del *processing pipeline*.
Fuente: [Open Networking Foundation, 2012].

En la figura 6 se aprecia cómo funciona el *processing pipeline* en un *switch OpenFlow* v1.3.0. Las tablas *OpenFlow* son numeradas secuencialmente, empezando por 0. El procesamiento siempre empieza en la primera tabla y el paquete procesado tendrá asociada una lista de acciones acumulativa denominada *action set*, que almacenará las acciones que debe realizar el *switch* sobre el paquete luego de ser procesado. Esta lista siempre comenzará vacía.

Cuando un paquete es procesado por una tabla de flujos, se hará una comparación entre las cabeceras del paquete y las entradas de flujo. Si aparece la cabecera, se habrá encontrado un *match*. Si las instrucciones de la entrada de flujo que hizo *match* especifican acciones, estas serán agregadas al *action set*. Si las instrucciones señalan que el paquete debe ser direcionado a otra tabla, se repetirá el mismo proceso, pero con el *action set* actualizado. Una entrada de flujo solo puede direccionar a una tabla con un índice superior a la precedente, es decir, el *processing pipeline* solo permite avanzar.

El *processing pipeline* terminará cuando no existan redireccionamientos a tablas subsecuentes, lo que conlleva a la ejecución del *action set*. Entre las múltiples acciones que puede contener un *action set*, la más relevante es la de redirección, ya que si esta no está definida, el paquete será descartado. Un paquete puede ser redireccionado a los puertos físicos, lógicos y reservados.

Para más información sobre las instrucciones y acciones de *OpenFlow*, se recomienda la lectura del anexo 6.1.2.4.

### 2.4.4. TABLA DE FLUJOS

Los flujos son el componente principal de una tabla de flujos de un *switch OpenFlow*. Se define como flujo a la agrupación de paquetes que comparten una serie de características comunes, como, por ejemplo, una pareja de direcciones IP de origen y destino. La primera versión de *OpenFlow* solo soportaba una tabla de flujos, sin embargo, a partir de la versión 1.1, un *switch* puede contener una o muchas. En ella se encuentran un conjunto de entradas que almacenan las instrucciones de enrutamiento, que indican las acciones que se deben realizar sobre un determinado tráfico. Cada una de las entradas de flujo tiene la estructura que se observa en el cuadro 1:

Tabla 1: Componentes principales de una entrada en la tabla de flujos en *OpenFlow* v1.3.0.

Fuente: [Open Networking Foundation, 2012].

| Match fields | Priority | Counters | Instructions | Timeout | Cookies |
|---|---|---|---|---|---|

1. **Match fields:** Conjunto de campos que se comparan con las cabeceras de los paquetes entrantes del *switch*, con el fin de determinar si el paquete pertenece a algún flujo en específico.
2. **Priority:** Orden de precedencia de las entradas en una tabla de flujos.
3. **Counters:** Se actualizan cuando los paquetes hacen *match*.
4. **Instructions:** Corresponde a las instrucciones que son ejecutadas cuando un paquete hace *match* con alguna entrada.
5. **Timeout:** Definen el tiempo máximo de vida de una entrada en la tabla.
6. **Cookies:** Se utilizan exclusivamente por el controlador para identificar las entradas de flujo.

Una entrada de la tabla de flujos se identifica por los campos *match fields* y *priority*. La entrada que hace *match* con cualquier paquete y que tiene una prioridad igual a 0 se denomina *table-miss*.

Existen tres modos en el que un controlador puede instanciar flujos en un *switch OpenFlow*:

- **Instanciación reactiva:** Cuando un flujo de paquetes llega al *switch*, el agente *OpenFlow* realiza una búsqueda en la tabla de flujos. Si no existe ningún *match*, este redirecciona el paquete al controlador a la espera de nuevas instrucciones, que se traducen en nuevas entradas en la tabla de flujos. Posteriormente, cualquier paquete que pertenezca al flujo agregado a la tabla será redireccionado sin la intervención del controlador. En este modo se reacciona al tráfico ya que las tablas se instancian en base al tráfico en tiempo real.
- **Instanciación proactiva:** En este caso, en vez reaccionar al paquete, un controlador *OpenFlow* puede predefinir los flujos y las acciones en las tablas de flujos, adelantándose al tráfico que puede llegar al *switch*, por lo que el envío de paquetes al controlador nunca ocurriría. El modo proactivo elimina la latencia introducida al momento de consultar al controlador por cada flujo.
- **Instanciación híbrida:** Corresponde a una combinación de los dos casos anteriores, lo cual permite un control granular para nuevos flujos mientras se mantiene el enrutamiento de baja latencia para el resto del tráfico.

#### 2.4.4.1. MATCHING

El proceso de *matching* en un *switch OpenFlow* se fundamenta en el paradigma *"match-plus-action"*, en donde un *match* puede ser realizado sobre múltiples campos cabecera de distintos protocolos, pertenecientes a distintas capas del *stack* de protocolos. En la figura 7 se observa una lista de algunos campos con los cuales se puede hacer *match*. La *"action"* se ejecuta luego ocurrido el *match*, y entre ellas se incluyen, por ejemplo, redireccionar paquetes a uno o más puertos de salida (como en *unicast* y *multicast*), reescribir las cabeceras de un paquete (como en NAT) y bloquear paquetes (como un *firewall*) [Kurose y Ross, 2017].

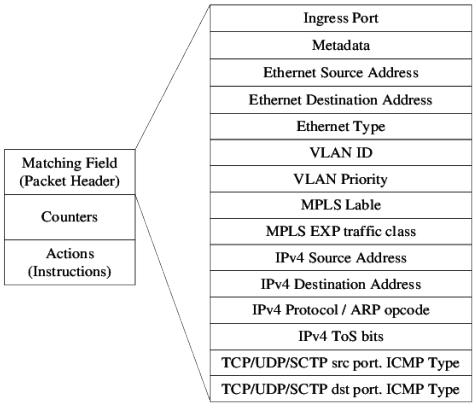



Figura 7: *Match fields* para IPv4.
Fuente: [Bholebawa y Dalal, 2016].

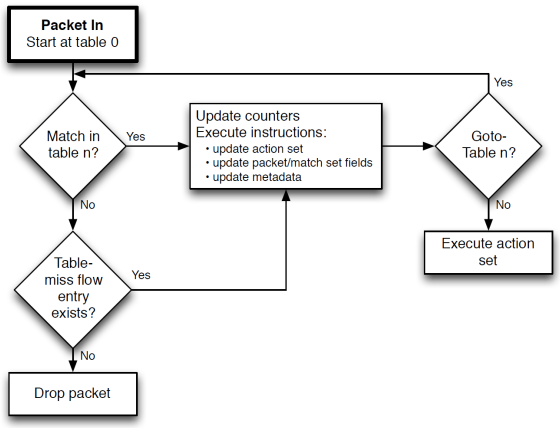



Figura 8: Diagrama de flujo de un paquete en un *switch OpenFlow* v1.3.0. Fuente: [Open Networking Foundation, 2012].

En la figura 8 se muestra un diagrama de flujo del proceso de *matching* en un *switch OpenFlow* v1.3.0. Cuando el *switch* recibe un paquete, este extraerá sus campos cabecera y realizará una búsqueda en la primera tabla de flujos. Si el paquete hace *match* con alguna entrada de flujo, será procesado como lo indican las instrucciones, en base al *processing pipeline*, lo que podría resultar en búsquedas en tablas subsecuentes, o en la ejecución del *action set*. Si el paquete hace *match* con dos o más entradas de flujo, se ejecutará las instrucciones de la que tiene prioridad mayor. Si dos entradas que hacen *match* tienen la misma prioridad, el *switch* elegirá alguna en base a su configuración. Si el paquete no hace ningún *match*, se generará un *table-miss*, que, dependiendo de cómo lo haya definido el controlador, podría resultar en, por ejemplo, el envío del paquete al controlador, el descarte del paquete o en el redireccionamiento del paquete a una tabla subsecuente.

## 2.5. CONTROLADORES

Tal como se observa en la figura 3, los controladores SDN se encuentran en el plano de control. Éstos se comportan como el sistema operativo de la red y desempeñan funciones de gestión sobre los *switches* y *routers* de la red mediante las APIs *southbound*; todos los cálculos computacionales se realizan ahí y muchas aplicaciones y características pueden ser agregadas si son necesarias. Existen una alta gama de controladores disponibles, incluso varios de código abierto, como, por ejemplo, *OpenDaylight*, *FloodLight*, ONOS y RYU. Se recomienda la lectura del anexo 6.2 para profundizar en este tema.

### 2.5.1. ESTRUCTURA GENÉRICA DE UN CONTROLADOR

La estructura genérica de un controlador se compone de los siguientes módulos: *link discovery, topology manager, storage, strategy making, flow table y control data*. Esencialmente, dos módulos son responsables de proveer el servicio de enrutamiento: *topology manager* y *link descovery*. El módulo *link descovery* es responsable del reconocimiento y mantención del estado de los enlaces físicos de la red. Existen dos tipos de *link discovery*: el primero se utiliza para reconocer los enlaces existentes entre los nodos internos (*switches OpenFlow*), mientras que el segundo se utiliza para el reconocimiento de enlaces entre los *host* y los nodos de borde. Este procedimiento es ejecutado por el controlador cuando un tráfico desconocido entra en el dominio de *OpenFlow*. Luego, la información obtenida por los módulos *link discovery* será utilizada para construir la base de datos de los vecinos en el controlador, capturando todos los vecinos *OpenFlow*. Posteriormente, el módulo *topology manager* se encargará de construir y mantener la información de la topología en el controlador y calcular las rutas en la red. Este módulo construirá la base de datos de la topología en el controlador, que contendrá la información sobre el camino más corto (y alternativo) a cualquier nodo *OpenFlow* o *host*[Salman *et al.*, 2016].

### 2.5.2. CARACTERÍSTICAS DE UN CONTROLADOR SDN

Existen muchos controladores disponibles para los usuarios, cada uno con diferentes características y rendimiento, por ejemplo, existen algunos que son pagados de nivel empresarial y otros de código abierto. A continuación, se enumeran seis características deben tomar en cuenta a la hora de elegir un controlador[Salman *et al.*, 2016]:

1. **Soporte para OpenFlow:** El protocolo *OpenFlow* es clave en SDN. Cuando se selecciona un controlador *OpenFlow*, hay que conocer la versión de *OpenFlow* que el controlador soporta y si está contemplado en el plan de desarrollo implementar nuevas versiones del protocolo. La razón de por qué hay que tener esto en cuenta es que funcionalidades como soporte para IPv6, por ejemplo, son solo compatibles con *OpenFlow v1.3.0* en adelante.

2. **Programabilidad de la red:** La programabilidad de los equipos de red es la característica más importante del paradigma SDN. El soporte y la capacidad de programabilidad de un controlador radica esencialmente en su grado de integración con una amplia gama de interfaces *northbound* disponibles, una buena interfaz gráfica y una buena línea de comandos (CLI).

3. **Eficiencia:** La eficiencia de un controlador es un término genérico usado para referirse a diferentes parámetros; rendimiento, escalabilidad, integridad y seguridad. Parámetros como el número de interfaces que un controlador puede manejar, la latencia y el *throughput* definen el rendimiento. Similarmente, existen varias

métricas que definen la escalabilidad, integridad y seguridad. Adicionalmente, la centralización del control en el esquema SDN presentan un gran desafío desde las perspectivas del rendimiento y la integridad. Con el fin de mitigar este problema, algunos controladores soportan un esquema distribuido.

4. **Asociación empresarial:** Si un controlador es apoyado y supervisado por organizaciones de buena reputación, tendrá más chances de ser mejorado y mantenido por un largo tiempo. *Cisco*, *Linux Foundation*, *Intel*, IBM, *Juniper*, etc., son ejemplos de organizaciones que han entrado al mercado de SDN y han participado en el desarrollo de controladores. La experiencia en redes y dominios computacionales, además de la capacidad económica de las organizaciones son un buen criterio a la hora de elegir sus productos.

## 2.6. EXPERIMENTACIÓN CON SDN

La experimentación práctica con un protocolo de red se puede realizar a través de distintos enfoques. Una forma de hacerlo es construyendo un banco de pruebas experimental con *hardware* real, lo que se traduce en resultados más realistas. Sin embargo, esta solución utiliza de dispositivos reales, lo que conlleva a un elevado costo de implementación. Si bien es cierto que existen iniciativas como *PlanetLab*[10], *Emulab*[11] y *OFELIA*[12], que ofrecen banco de pruebas reales con bastantes recursos, normalmente no son accesibles para el usuario común.

El segundo enfoque corresponde a la simulación, en donde las operaciones e interacciones de los dispositivos reales son modeladas y ejecutadas por *software*. Esta aproximación tiene muchas ventajas con respecto al enfoque anteriormente descrito, como el bajo costo, la flexibilidad, controlabilidad, escalabilidad, repetibilidad, accesibilidad a muchos usuarios, y, usualmente, más rápida que la ejecución en tiempo real. No obstante, hay que preocuparse de que la simulación sea lo suficientemente fidedigna, ya que de lo contrario, los resultados simulados podrían diferir de los experimentales. Entre los simuladores de SDN más conocidos se encuentran NS-3[13] y *EstiNet*[14] en modo simulación.

Para superar el problema de una posible incongruencia entre resultados obtenidos en una simulación, es posible utilizar el tercer y último enfoque que corresponde a la emulación. Este difiere de la simulación ya que la emulación se comporta casi como un experimento con *hardware* real, por lo que debe ser ejecutado en tiempo real, mientras que la velocidad de una simulación puede ser mayor o menor al tiempo real. Además,

[10] **PlanetLab**: `https://www.planet-lab.org/`
[11] **Emulab**: `https://www.emulab.net/`
[12] **OFELIA**: `http://www.fibre-ict.eu/index.php/cmf/ofelia`
[13] **NS-3**: `https://www.nsnam.org/`
[14] **Estinet**: `www.estinet.com/es/`

en la emulación algunos dispositivos reales ejecutan sistemas operativos y aplicaciones reales que interactúan con dispositivos simulados, mientras que, en la simulación, generalmente, no hay sistemas operativos ni aplicaciones reales involucrados. Entre los emuladores más conocidos de SDN se encuentran *EstiNet* en modo emulación y *Mininet*[Wang, 2013].

# CAPÍTULO 3
# SOFTWARE

## 3.1. CONTROLADOR OPENDAYLIGHT

*OpenDaylight* (ODL) nació con el objetivo de acelerar la adopción de SDN y crear una base sólida para la virtualización de funciones, colaborando con las compañías más grandes del sector. En abril del 2013, la *Linux Foundation* anunció *OpenDaylight* como un proyecto abierto, con el objetivo de promover la innovación mediante una perspectiva abierta y transparente. Los miembros fundadores fueron compañías importantes del rubro como *Cisco*, *Citrix*, HP, IBM, *Juniper Networks*, *Microsoft*, *Red Hat* y *VMWare*[Projects, 2018b].

La primera versión, denominada *Hydrogen*, fue lanzada en febrero del 2014, que incluyó un controlador de código abierto, capacidades limitadas de virtualización y algunos plugin de protocolos. En noviembre del 2014 fue lanzada la versión *Helium*, que introdujo en ODL un entorno de trabajo más adecuado, basado en *Karaf* y en la gestión de redes en base a modelos YANG. La última versión corresponde a *Neon*, lanzada en marzo del 2019, y trae consigo mejoras importantes en escalabilidad y estabilidad en áreas como la virtualización de redes, configuración y enrutamiento, útiles para el usuario final.

La comunidad de *OpenDaylight* sigue colaborando con otras comunidades de código abierto, incluyendo *OpenStack*, *Kubernetes*, OPNFV, y ONAP. Actualmente, la comunidad ONAP está utilizando ODL en dos de sus proyectos (APP-C y SDN-C), y está en proceso de emplear ODL en uno nuevo (SDN-R), para cubrir casos de uso avanzados como CCVPN y 5G[Projects, 2019].

Todas las versiones de *OpenDaylight* lanzadas a la fecha se pueden obtener desde el siguiente enlace[15].

### 3.1.1. ARQUITECTURA DE OPENDAYLIGHT

*OpenDaylight* es un controlador altamente accesible, extensible, escalable y multi protocolo. Actúa como un *middleware* entre las aplicaciones que requieran servicios de los dispositivos de red y los protocolos que se comunican directamente con los dispositivos de red. El controlador, mediante el *Service Abstraction Layer* (SAL), permite a las aplicaciones ser agnósticas con respecto a las especificaciones de los dispositivos de red, dotando a los desarrolladores de aplicaciones de la posibilidad de enfocarse sólo funcionalidades y abstraerse de lo demás.

[15]**ODL Releases**: `https://docs.OpenDaylight.org/en/latest/downloads.html`

*OpenDaylight* es de código abierto, contenido en su propio JVM, que funciona en cualquier sistema operativo o *hardware* mientras este sea compatible con *Java*[Projects, 2018a]. El controlador se fundamenta en las siguientes tecnologías:

- **Maven:** Es una herramienta de gestión de proyectos que simplifica y automatiza dependencias entre un proyecto o diferentes proyectos. Ayuda a los desarrolladores a gestionar todos los plugin y dependencias requeridas por las aplicaciones.
- **Java:** Es el lenguaje de programación utilizado para desarrollar aplicaciones y módulos en el controlador. El desarrollo en *Java* proporciona seguridad en tiempo de compilación, además ser una manera fácil de implementar servicios definidos.
- **OSGi**[16]**:** El *back-end* de *OpenDaylight*. Le permite al controlador cargar dinámicamente paquetes JAR (que componen a las aplicaciones) y ligar los módulos para intercambiar información.
- **Karaf**: Es un contenedor de aplicaciones implementado sobre OSGi, que simplifica el proceso de empaquetamiento e instalación de aplicaciones.

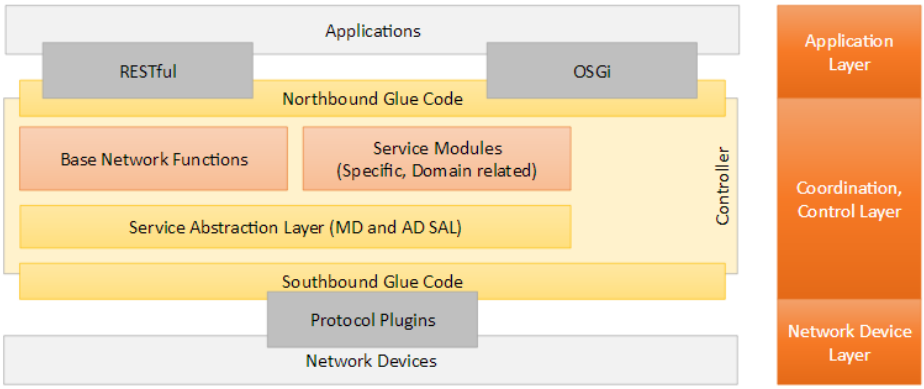



Figura 9: Arquitectura simplificada de *OpenDaylight*.
Fuente: [Rao, 2015a].

En la figura 9 se muestra la arquitectura simplificada del *software*. Como cualquier implementación SDN, *OpenDaylight* sigue la arquitectura común de tres capas[GARCIA, 2015]:

1. **Network Device Layer**: Pueden ser físicos o virtuales, son programados por los protocolos implementados en el controlador. Gracias al *Service Abstraction Layer*, *OpenDaylight* es compatible con la mayoría de los elementos de red, independiente del proveedor.

[16]**Open Services Gateway Initiative**: https://www.osgi.org/

2. **Coordination, Control Layer**: Es la capa central en donde se manifiesta la abstracción de SDN. El controlador también implementa módulos internos que agregan servicios y funcionalidades a la red. Por ejemplo, posee un plugin dinámico que le permite adquirir estadísticas y obtener la topología de la red. Existen tres APIs (DOM, REST y JAVA) que permiten la programación de módulos y aplicaciones.

3. **Application Layer**: En esta capa se encuentran aplicaciones que realizan ingeniería de tráfico de red (orquestan y monitorizan el controlador) o soluciones que usan virtualización de servicios.

Para una descripción más detallada de la arquitectura de ODL, se recomienda la lectura del anexo 6.4.

## 3.2. MÓDULOS DE OPENDAYLIGHT

*OpenDaylight* es una plataforma modular, es decir, es una combinación de múltiples proyectos integrados. Cada proyecto tiene un propósito y existe grupo de personas que se encargan de decidir la prioridad de cada proyecto.

Existen muchos plugin y módulos construidos para ODL, algunos se encuentran en producción, mientras que otros están en desarrollo. A continuación, se presentan los módulos de ODL que se utilizarán a lo largo de la memoria.

### 3.2.1. L2 SWITCH

*L2 switch* es una implementación de un *switch* de auto aprendizaje basada en MD-SAL para ODL, con optimizaciones para redireccionar paquetes. El módulo permite el manejo de tráfico de L2, además de proveer varios servicios reutilizables:

- **PacketHandler**: Proporciona una infraestructura dentro de un controlador ODL para procesar paquetes entrantes y redireccionarlos como corresponda.
- **AddressTracker**: Construye y mantiene una tabla de mapeos que puede tener muchos atributos, tales como:
  - Dirección MAC de un *host*.
  - ID de un *switch* conectado.
  - Número de puerto de un *switch*.
  - VLAN.

    - MPLS *tags*.

- **Path Computation Service**: Proporciona una ruta óptima entre *hosts*. Puede ser la ruta más corta, la ruta con costo menor, etc., similar a los algoritmos de enrutamiento en los protocolos L3.
- **Flow Writer Service**: Este servicio se encarga de programar todos los flujos apropiados en los *switches* intermedios una vez que el camino óptimo entre dos *hosts* haya sido calculado. Mantiene un seguimiento de los mapeos, desde meta flujos hasta flujos individuales, para que nuevos flujos puedan ser reprogramados en caso de que un puerto o un *switch* falle.
- **STP Service**: Corresponde al servicio del *spanning tree protocol* que permite a ODL participar en un *spanning tree*. Es útil cuando ODL se utiliza en redes híbridas, lo cual hace necesario que ODL participe en el *spanning tree*.

### 3.2.2. DLUXAPPS

*Dlux* es una interfaz WEB de ODL basada en *Angular JS*. Consiste en dos partes lógicas, llamadas *core* y *applications*. *Core* es un *framework* que proporciona funcionalidad básicas, como navegación, autentificación, etc. Las *applications* se construyen sobre el *core* y entre ellas se encuentran:

- **YANGMAN**: Corresponde a una interfaz gráfica que permite interactuar con el controlador. Está basada en modelos YANG y permite a los usuarios leer y modificar *data* incluso si no tienen conocimiento de los modelos.
- **YANG Visualizer**: Permite visualizar modelos YANG de manera gráfica.
- **Node**: Maneja un inventario de los nodos de la red.
- **Topology**: Ofrece una representación gráfica de la topología.

### 3.2.3. VIRTUAL TENANT NETWORK

*Virtual Tenant Network* (VTN), representada en la figura 10, es una aplicación que provee una red de múltiples inquilinos sobre una red SDN. La característica principal de VTN es su capa de abstracción lógica. Está permite la completa separación de la capa lógica y física, lo que posibilita a los usuarios implementar cualquier tipo de red, sin la necesidad de conocer la topología física subyacente.

VTN permite a los usuarios definir redes con las mismas características de una red L2/L3 convencional. Una vez que la red sea diseñada en VTN, será automáticamente

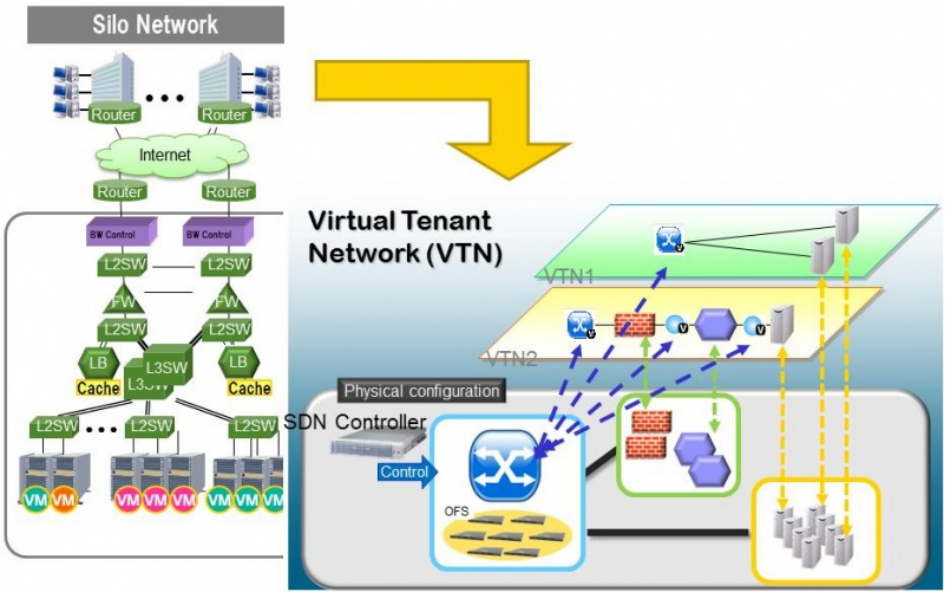



Figura 10: Representación de VTN en *OpenDaylight*.
Fuente: [Project, 2018b].

mapeada a la red física subyacente y luego configurada en los *switches* mediante un protocolo de control *southbound*. La definición del plano lógico no solamente oculta la complejidad de la red física, sino que también permite un mejor manejo de los recursos de red, lo que minimiza los errores de configuración y reduce los tiempos de reconfiguración.

VTN se implementa mediante dos componentes. El primero es *VTN Manager*, implementado como un módulo de *OpenDaylight*, el cual provee una interfaz REST que permite crear, actualizar o eliminar componentes VTN. El segundo es *VTN Coordinator*, que es aplicación de orquestación externa que provee una interfaz REST para que un usuario pueda aprovechar las características de VTN de *OpenDaylight*. Esta aplicación interactúa directamente con el plugin *VTN Manager*, encargado de implementar la configuración definida por el usuario[Göransson y Black, 2015].

#### 3.2.3.1. ELEMENTOS QUE COMPONEN UNA VTN

El cuadro 2 muestra los elementos que componen una VTN. Una red virtual se construye en base a nodos virtuales (*vBridge*, *vRouter*), interfaces virtuales y enlaces virtuales. Es posible construir redes L2/L3 conectando las interfaces virtuales de los nodos vía *vlinks*, tal como se muestra en la figura 11.

Tabla 2: Elementos de VTN.
Fuente: Elaboración propia.

| Objeto | Función |
|---|---|
| vBridge | Realiza las funciones de un *switch* L2. |
| vTerminal | Similar al *vBridge*, pero solo soporta una *vInterface*. Siempre se utiliza en conjunto con la acción de redirección de un *flow filter*. |
| vRouter | Realiza las funciones de un *router*. |
| vTep | Representa un *Tunnel End Point* (TEP). Se utiliza para dar soporte al protocolo de encapsulación VXLAN. |
| vTunnel | Representa lógica de un *Tunnel*. Se utiliza para dar soporte a tecnologías como VPN. |
| vBypass | Representa la conectividad entre dos redes controladas. |
| Virtual Interface | Representa las interfaces de un nodo virtual. |
| Virtual Link (vLink) | Representación lógica de un enlace L1 entre interfaces virtuales. |

#### 3.2.3.2. MAPEANDO RECURSOS FÍSICOS DE LA RED

Al configurar una red virtual es necesario mapear los recursos físicos de la red. El mapeo identifica a qué red virtual cada paquete transmitido o recibido por el *switch OpenFlow* pertenece. Cuando un paquete llega a un *switch OpenFlow*, se determina si es que existe un mapeo por puertos, luego un mapeo por VLAN, para finalmente enviar el paquete al *vBridge* correspondiente.

A continuación, se detalla cómo funciona cada tipo de mapeo:

- **Mapeo por puertos**: Mapea recursos físicos a una interfaz de un *vBridge* utilizando el *Switch ID, Port ID y VLAN ID* del paquete L2 entrante. Paquetes sin *tag* VLAN también son soportados. Tiene prioridad sobre el mapeo por VLAN.
- **Mapeo por VLAN**: Mapea recursos físicos a un *vBridge* utilizando el *VLAN ID* de un paquete L2 entrante. Mapea los recursos físicos de un *switch* en particular a un *vBridge* utilizando el *Switch ID* y el *VLAN ID* del paquete L2 entrante.
- **Mapeo por MAC**: Mapea recursos físicos a una interfaz del *vBridge* utilizando la dirección MAC del paquete L2 entrante.

#### 3.2.3.3. VIRTUALIZACIÓN DE FUNCIONES DE RED

La definición de una VTN, mapeo y posterior comunicación entre los nodos físicos subyacentes, se realiza mediante un proceso de virtualización de funciones de red (NFV).

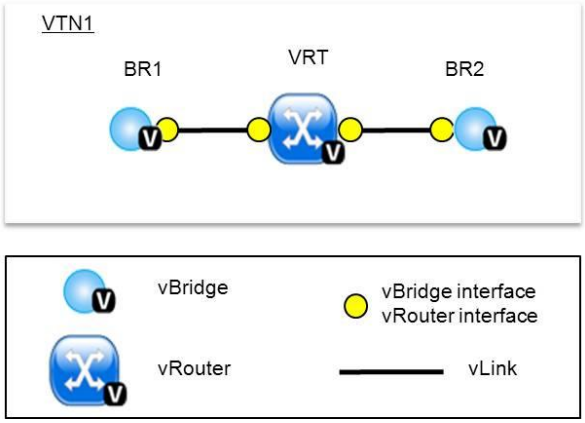



Figura 11: Componentes virtuales de VTN.
Fuente: [Project, 2018b].

VTN, por definición, permite virtualizar funciones de transferencia de paquetes (L2 y L3) y funciones de control de tráfico basado en flujos (filtrado y redireccionamiento). Los componentes, descritos en la sección 3.2.3.1, son los encargados de implementar estas funciones de red. A continuación, se enuncian las más importantes:

- **Funciones vBridge**: Los nodos *vBridge* proporcionan funcionalidades básicas de un *switch*, permitiendo la transmisión de paquetes a interfaces virtuales, de acuerdo a la dirección MAC del destinatario. El *vBridge* revisa la *MAC Address Table* y transmite el paquete a interfaz virtual correspondiente cuando la MAC del destinatario ha sido aprendida. Si la MAC no es reconocida, el *vBridge* hace *flooding* a todas las interfaces, exceptuando la de entrada.

  Un *vBridge* aprende las direcciones MAC de manera tradicional. Cuando un paquete es transmitido hacia alguna interfaz virtual, el *vBridge* se encarga de mapear la dirección MAC de la fuente con su correspondiente interfaz virtual, para luego almacenar estos datos en una *MAC Address Table*, creada exclusivamente para este *vBridge*. Es posible que el *vBridge* aprenda direcciones MAC de manera estática, pero éstas deben ser registradas manualmente.

- **Funciones vRouter**: Los *vRouter* se encargan de transferir paquetes IP entre *vBridges*. El nodo proporciona funciones de enrutamiento, aprendizaje ARP, ARP *aging* y DHCP *relay*.

- **Funciones de filtrado en base flujos (vInterface)**: Las funciones de filtrado en base a flujos son similares a un ACL. Éstas permiten aceptar o denegar el tránsito de un flujo de paquetes que cumplan con condiciones particulares. Además, pueden efectuar redireccionamiento de paquetes, tarea imposible de hacer directamente con ACLs. El filtrado en base a flujos es aplicable a cualquier interfaz de un nodo virtual y controla los paquetes que transitan por esa interfaz. En el cuadro 3 se especifican condiciones de *matching* para un filtrado en base a

flujos, pertenecientes a distintas capas del modelo OSI, las cuales se pueden usar de forma combinada.

Tabla 3: Condiciones de *matching*.
Fuente: [Project, 2018b].

| **Condición** |
|---|
| Source MAC address |
| Destination MAC address |
| MAC ether type |
| VLAN Priority |
| Source IP address |
| Destination IP address |
| DSCP |
| IP Protocol |
| TCP/UDP source port |
| TCP/UDP destination port |
| ICMP type |
| ICMP code |

En el cuadro 4 se muestran las acciones aplicable a los paquetes que hacen *match*. Éstas permiten, por ejemplo, cambiar la ruta de los paquetes provenientes desde un *host* particular, en base a la dirección IP del destinatario.

Tabla 4: Acciones tras *matching*.
Fuente: [Project, 2018b].

| **Acción** | **Función** |
|---|---|
| Pass | Permite el flujo de un paquete que hace *match* con las condiciones especificadas. |
| Drop | Descarta el flujo de un paquete que hace *match* con las condiciones especificadas. |
| Redirection | Redirecciona el paquete a la interfaz virtual deseada. Son soportadas la redirección transparente (manteniendo la dirección MAC) y la redirección de enrutamiento (cambiando la dirección MAC). |

## 3.3. EMULADOR DE RED MININET

El emulador *Mininet* es una plataforma de pruebas de red de código abierto, altamente configurable, siendo reconocido en la ONS del año 2013 como la herramienta de apoyo más utilizada para experimentar con SDN y *OpenFlow*. *Mininet* implementa una red

virtual de *hosts*, *switches*, controladores y enlaces. Cada *host* corre *software* estandar *Linux* de *networking* y los *switches* soportan *OpenFlow*, lo que permite realizar un redireccionamiento personalizado, todo esto sobre una red SDN emulada.

*Mininet* se compone de un conjunto de características integradas en *Linux* que permiten a un sistema ser dividido en pequeños contenedores, cada uno con un cupo fijo de poder de procesamiento, y enlaces virtuales, que garantizan precisión en velocidades y retardos. El resultado final es un sistema eficiente, rápido, altamente escalable y que puede ser configurado según las necesitades del usuario[Lantz *et al.*, 2017].

Una red de *Mininet* consiste en los siguientes componentes:

- **Hosts aislados**: Un *host* emulado en *Mininet* es un grupo de procesos a nivel de usuario, que se ejecutan dentro de un *namespace* para mantener los estados de la red. Los *namespace* de red proporciona un agrupador de procesos con propiedad exclusiva de interfaces, puertos y tablas de enrutamiento (como ARP y IP). Por ejemplo, dos *webservers* en dos *namespace* de red distintos pueden coexistir en un sistema, cada uno escuchando al puerto 80.
- **Links emulados**: Cada *host* emulado tiene sus propias interfaces *Ethernet* virtuales. Un *virtual Ethernet pair* se comporta como un cable que conecta dos interfaces virtuales. Cada interfaz aparece como un puerto *Ethernet* completamente funcional en todo el sistema y aplicaciones.
- **Switch emulados**: Típicamente, *Mininet* utiliza *Open vSwitch* para distribuir paquetes entre sus interfaces. Los *switches* y *routers* emulados pueden ejecutarte en modo *kernel* (para velocidad) o en el espacio de usuario (para ser modificados con facilidad).

### 3.3.1. CARACTERÍSTICAS DE MININET

- **Velocidad:** Crear una red simple emulada solo toma unos pocos segundos, por lo que el *loop* de *"ejecución-edición-depuración"* se hace muy rápido.
- **Topologías personalizadas:** Es posible implementar cualquier tipo de topología utilizando la API de *Python*. Este punto es sumamente importante y será profundizado en la sección 4.3.4.
- **Enrutamiento de paquetes personalizado:** Los *switch* de *Mininet* son programables utilizando el protocolo *OpenFlow*. Estas configuraciones son fácilmente transferibles a *switch OpenFlow* reales. *Mininet* soporta hasta la versión 1.3.0 de *OpenFlow*.
- **Ejecución de programas reales:** Es posible ejecutar en los host emulados en *Mininet* cualquier *software* que funcione en *Linux*, como *Wireshark*.

- **Replicación de resultados:** Cualquier persona puede ejecutar el código en su sistema una vez empaquetado en un script de *Python*.
- **Instalación simple:** Es posible ejecutar *Mininet* en un computador portátil, un servidor, una máquina virtual y en la nube, dependiendo de los requerimientos del usuario.
- **Código abierto:** Cualquier persona puede revisar el código fuente de *Mininet* y modificarlo.
- **Desarrollo activo:** Existe una comunidad de usuarios y desarrolladores que dan soporte a la plataforma de manera periódica.
- **Clustering:** *Mininet* funciona en base a un concepto llamado *"just enough virtualization"*. Sus *hosts* no necesitan correr *software* extra innecesario, como múltiples *kernels* o demonios, ni tampoco requieren sistemas de archivos voluminosos. Para soportar configuraciones que excedan los recursos de un servidor, se puede utilizar *clustering* y distribuir los bancos de pruebas a través de múltiples servidores físicos o virtuales[Lantz y O'Connor, 2015].

## 3.4. SOFWARE COMPLEMENTARIO

El *software* complementario es una parte esencial del laboratorio, ya que su función es integrar todas las herramientas mencionadas en este capítulo, en un entorno virtual fácil de usar. Su uso no es necesariamente obligatorio y si el usuario lo desea puede buscar alternativas. Por ejemplo, el *host* que implementará el entorno virtual está basado en *Windows 10*, por lo que el hipervisor elegido fue *VMware* en su versión gratuita, ya que, desafortunadamente, otras alternativas de código abierto, como *VirtualBox*[17], no entregaron los resultados deseados. Si se desea utilizar otro sistema operativo, como uno en base a *Linux*, existen más alternativas de código abierto como hipervisor, tal como QEMU[18] o similares.

A continuación, se realiza una pequeña mención del software complementario elegido:

- **Hipervisor (VMware Workstation)**[19]: Hipervisor de tipo 2, también denominado *hosted*, el cual puede ser ejecutado en los sistemas operativos *Windows* y *Linux* en sus versiones de *64 bits*. Permite a los usuarios configurar múltiples máquinas virtuales (VMs) en una máquina física y utilizarlas de manera concurrente, junto con la máquina física. Cada VM puede ejecutar su propio sistema operativo

[17] **VirtualBox**: `https://www.virtualbox.org/`
[18] **QEMU**: `https://www.qemu.org/`
[19] **VMware**: `https://www.vmware.com/`

y es compatible con múltiples distribuciones de *Linux*. *VMware Workstation* es desarrollado y vendido por *VMware Inc.*, una división de *Dell Technologies*.

Existen dos versiones de hipervisor, una gratuita y otra comercial. *VMware Workstation Player* es la versión gratuita y está destinada para uso no comercial. La versión comercial, en cambio, corresponde a *VMware Workstation Pro* e incluye todo lo que trae la versión gratuita, junto con otras funciones interesante, como, por ejemplo, la capacidad de guardar y restaurar *snapshots*. Otra herramienta importante es *Virtual Network Editor*, que permite crear y configurar adaptadores de redes virtuales, solo disponible en la versión comercial. *VMware Workstation Pro* tiene un periodo de prueba de 30 días. Trancurrido este periodo se debe adquirir una llave[20], con un costo de 249.99 USD.

- **Clientes REST (Postman y cURL)**: Estas aplicaciones se utilizarán para realizar consultas REST a la *API* de *OpenDaylight*. *Curl* será utilizado para realizar consultas en la misma VM en donde se aloja *OpenDaylight*, mientras *Postman* será utilizado para realizar consultas desde máquinas externas.

  - **cURL**[21]: Es una herramienta de línea de comandos que permite transferir datos con una URL. Soporta DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, *Telnet* y TFTP. No hay muchas limitaciones con *curl*, pero manejar una interfaz basada en texto para enviar un alto flujo de información JSON puede ser desafiante en el contexto de la usabilidad.
  - **Postman**[22]: Compuesta por diferentes herramientas y utilidades gratuitas, *Postman* es un *software* gratuito que facilita el *testing* dentro del universo de las APIs REST, disponible para *Windows* y *Linux*. Este *software* le entrega herramientas a los desarrolladores para crear, compartir, probar y documentar APIs de manera sencilla. Además, *Postman* permite crear y almacenar consultas REST sin importar su nivel de complejidad, junto con sus respectivas respuestas.

- **Cliente SSH (PuTTY)**[23]: Implementación gratuita de SSH y *Telnet* para *Windows* y plataformas *Unix*. Además, provee un emulador de terminales *xterm*.

- **Analizador de paquetes (Wireshark)**[24]: Herramienta gratuita y de código abierto que se utiliza como interfaz gráfica para el programa *tcpdump*, el cual se desempeña como analizador de paquetes. Es usado para solucionar problemas de la red, análisis, desarrollo de protocolos y educación. Dado el alto volumen de tráfico que maneja una red típica, *Wireshark* permite la interceptación y filtrado de tráfico en un formato legible para las personas.

[20] `https://store.vmware.com/store/vmware/en_US/pd/productID.5222154500`
[21] **cURL**: `https://curl.haxx.se/`
[22] **Postman**: `https://www.getpostman.com/`
[23] **PuTTY**: `https://www.putty.org/`
[24] **Wireshark**: `https://www.wireshark.org/`

- **Router virtual (VyOS)**[25] : Es un sistema operativo, basado en *Debian*, que provee un *router* virtual gratuito. Compite directamente con soluciones comerciales de renombrados proveedores de internet. *VyOS* funciona en sistemas *amd64*, *i586* y ARM, por lo que es posible usarlo como *router* virtual para implementaciones en la nube. Al ser de código abierto, es una excelente herramienta para personas que necesiten tener un mayor control sobre su red sin necesidad de *hardware* adicional. Entre sus características se incluye la capacidad de ejecutarse tanto en plataformas físicas o virtuales, soporte para VPN, *firewall*, NAT, BGP y múltiples protocolos de red. *VyOS* es solo una de las alternativas para implementar router virtuales, existen otras como *pfSense*[26], basada en *FreeBSD*, que también pueden cumplir con esta función.

[25] **VyOS Wiki**: `https://vyos.readthedocs.io/en/latest/`
[26] **pfSense**: `https://www.pfsense.org/`

# CAPÍTULO 4
# LABORATORIO: CONFIGURACIÓN Y CONCEPTOS BÁSICOS

## 4.1. CONFIGURACIÓN GLOBAL

En esta sección se explicará como instalar y configurar el *software* que será utilizado para implementar un laboratorio virtual de *Software-Defined Networking*. El *host* en donde se desplegará el entorno virtual posee las siguientes características:

- **CPU**: *Intel Core i5 4690k @4.5Ghz*
- **RAM**: *16 GB DDR3 1600Mhz*
- **GPU**: *Nvidia GTX 1070 8GB VRAM*
- **OS**: *Windows 10 1903*

Como se mencionó en la sección 3.4, el sistema operativo del *host* será *Windows 10*. Si el usuario desea utilizar otro SO como *Linux*, lo puede hacer, solo debe preocuparse de configurar las máquinas en el hipervisor seleccionado de forma análoga a como se configurarán en *VMware* para *Windows*.

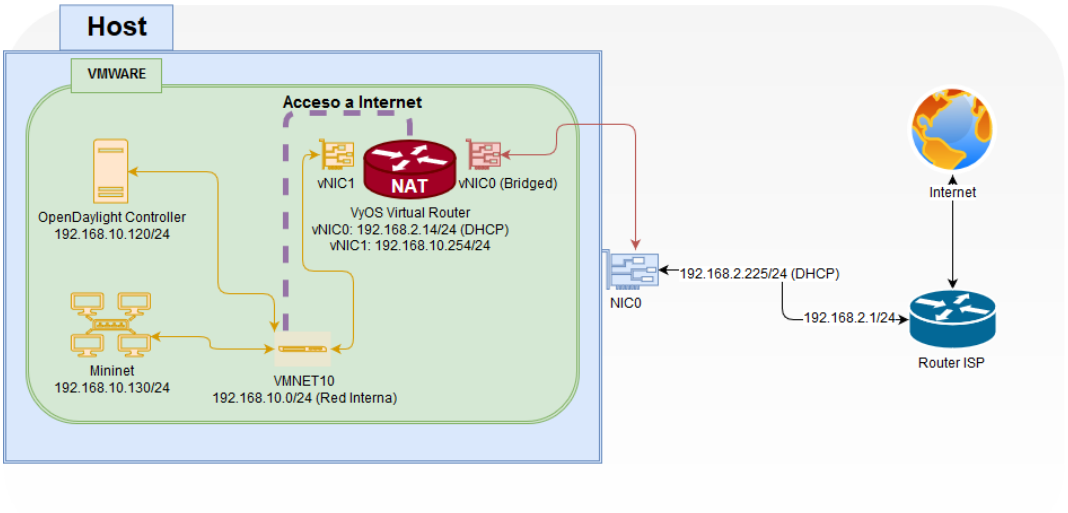



Figura 12: Entorno virtual para el desarrollo del laboratorio.
Fuente: Elaboración propia.

La figura 12 representa un diagrama el entorno virtual. Se configurarán dos VMs y un *router* virtual (*VyOS*) con dos interfaces. Las VMs se conectarán directamente a *VMnet10* (subred 192.168.10.0/24). Una VM tendrá asignada la IP estática 192.168.10.120/24 y correrá *OpenDaylight* sobre *CentOS 7*, mientras que la otra tendrá asignada la IP estática 192.168.10.130/24 y correrá una instancia de *Mininet* sobre *Ubuntu*. La interfaz *eth1* del *router* virtual se conectará directamente a *VMnet10*. La interfaz *eth0* deberá conectar a *vNIC0* con *NIC0* en modo *bridged*, otorgando al *router* virtual acceso a la red física del *host*. Finalmente, será necesario configurar NAT en *VyOS* para darle acceso a internet a las VMs.

En resumen, se deberá instalar en orden el *software* indicado a continuación:

1. **Hipervisor**: *VMware Workstation 15 Pro*.
2. **Router virtual**: *VyOS v1.1.8*.
3. **Emulador de red**: *Mininet 2.2.2-170321*.
4. **Controlador**: *OpenDaylight Nitrogen v0.7.2*.
5. **Aplicación de red**: *VTN Coordinator v6.6.0*.

## 4.2. INSTALAR Y CONFIGURAR EL HIPERVISOR Y EL ROUTER VIRTUAL

El primer paso para implementar el entorno virtual descrito en la sección 4.1 es instalar y configurar el hipervisor y el *router* virtual.

### 4.2.1. INSTALAR Y CONFIGURAR VMWARE WORKSTATION

La instalación de *VMware Workstation 15 Pro*[27] en *Windows 10* es bastante directa y no es necesario ahondar en este tema. Una vez instalado *VMware*, se deberán configurar los adaptadores de red detallados en el cuadro 5, mediante la herramienta *Virtual Network Editor*.

**Nota:** Para instalar nuevas máquinas virtuales, se recomienda seguir el siguiente tutorial[28].

[27] `https://www.vmware.com/cl/products/workstation-pro/workstation-pro-evaluation.html`

[28] `https://www.eukhost.com/kb/vmware-installation-and-configuration-tutorial/`

Tabla 5: Adaptadores de red.
Fuente: Elaboración propia.

| Name | Type | E. C. | H. C. | DHCP | SN. Addr. | SN. Mask |
|---|---|---|---|---|---|---|
| VMnet0 | Bridged | Auto-bridging | - | - | - | - |
| VMnet1 | Host-Only | - | Connected | Enabled | 192.168.76.0 | 255.255.255.0 |
| VMnet8 | NAT | NAT | Connected | Enabled | 192.168.78.0 | 255.255.255.0 |
| VMnet10 | Custom | - | - | - | 192.168.10.0 | 255.255.255.0 |

### 4.2.2. INSTALAR Y CONFIGURAR VYOS

*VyOS* permitirá aislar la red virtual de la red física, otorgando al usuario un mayor control sobre el entorno virtual en donde se desplegarán los experimentos. A continuación, se enumeran los pasos para su instalación y configuración:

1. **Instalar la VM**: Es recomendable utilizar la configuración de *hardware* por omisión, asignada por *VMware*.
2. **Instalar VyOS**: Se recomienda seguir el tutorial de la wiki oficial, disponible en el siguiente enlace[29].
3. **Configurar el adaptador de red**: Agregar un nuevo adaptador de red al hardware de la máquina virtual:
   - **Network Adapter**: Modo *Bridged*.
   - **Network Adapter 2**: Modo *Custom*, seleccionar el adaptador *VMnet10* (cuadro 5).
4. **Ejecutar VyOS**: Las credenciales para ingresar a la VM se asignan automáticamente y son:
   - **login**: vyos
   - **password**: vyos
5. **Configurar VyOS**: Entrar al modo configuración:

   ```
   configure
   ```

   Los siguientes comandos se encargarán de configurar las interfaces virtuales de *VyOS*, junto con la configuración básica de NAT, que dotarán a las VMs de acceso a internet:

[29] `https://vyos.readthedocs.io/en/latest/install.html`

```
#Configuración Inicial
set system host-name VyOS
set system domain-name vmmaster.local
set system time-zone Chile/Continental
set service ssh port '22'

#Configuración Interfaces
set interfaces ethernet eth0 address dhcp
set interfaces ethernet eth0 description 'EXTERNO'
set interfaces ethernet eth1 address '192.168.10.254/24'
set interfaces ethernet eth1 description 'VMNET10'
set system gateway-address 192.168.2.1

#Configuración NAT Source
set nat source rule 10 outbound-interface eth0
set nat source rule 10 source address 192.168.10.0/24
set nat source rule 10 translation address masquerade
```

Guardar la configuración con los siguientes comandos:

```
commit
save
exit
```

#### 4.2.2.1. COMPROBAR VYOS

Para verificar el funcionamiento de *VyOS* hay que comprobar si las interfaces están correctamente configuradas, mediante el comando *show interfaces* (figura 13). En este caso, el DHCP le asignó a *eth0* la IP 192.168.2.14.

```
Codes: S - State, L - Link, u - Up, D - Down, A - Admin Down
Interface        IP Address                     S/L  Description
---------        ----------                     ---  -----------
eth0             192.168.2.14/24                u/u  EXTERNO
eth1             192.168.10.254/24              u/u  VMNET10
lo               127.0.0.1/8                    u/u
                 ::1/128
```

Figura 13: Interfaces *VyOS*.
Fuente: Elaboración propia.

Luego, hay que verificar la configuración del NAT con el comando *show nat source rules*. Tal como se muestra en la figura 14, el tráfico de la subred interna 192.168.10.0/24 saldrá por 192.168.2.14.

```
Codes: X - exclude rule, M - masquerade rule

rule    intf            translation

----    ----            -----------

M10     eth0            saddr 192.168.10.0/24 to 192.168.2.14

        proto-all       sport ANY
```

Figura 14: NAT *Source* en *VyOS*.
Fuente: Elaboración propia.

## 4.3. INSTALAR Y CONFIGURAR EL EMULADOR DE RED MININET

El segundo paso para implementar el entorno virtual es instalar el emulador de red *Mininet*. La forma más fácil y rápida es descargando la imagen[30] preconfigurada y cargarla directamente en el hipervisor. Ésta está basada *Ubuntu 14.04 LTS* y trae preinstalado *Mininet v2.2.2*, junto con un conjunto de herramientas útiles de *networking*.

1. **Instalar la VM**: Es recomendable utilizar la configuración de *hardware* con la que viene preconfigurada la imagen.
2. **Configurar el adaptador de red**: Este paso es importante, ya que permitirá conectar la VM a *VyOS*:
   - **Network Adapter**: Modo *Custom*, seleccionar el adaptador *VMnet10* (cuadro 5).
3. **Ejecutar la VM**: Las credenciales preconfiguradas son las siguientes:
   - **login**: mininet
   - **password**: mininet

   En la figura 15 se muestra la pantalla que debería aparecer al momento de loguearse en la VM.
4. **Configurar la red**: Asignar a la VM la IP estática 192.168.10.130, que pertenece a la subred de *VMnet10*. Para esto se deberá modificar:

   ```
   sudo nano /etc/network/interfaces
   ```

   Con la siguiente configuración:

[30] https://github.com/mininet/mininet/wiki/Mininet-VM-Images

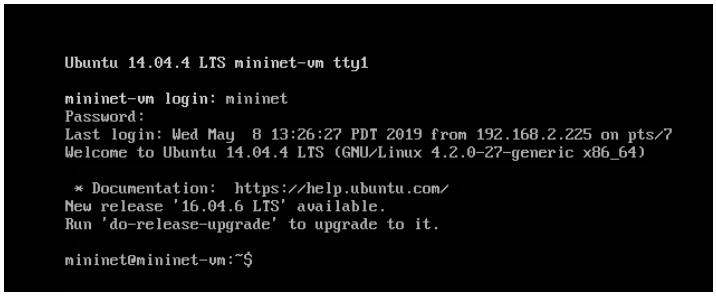



Figura 15: Máquina Virtual con *Mininet*.
Fuente: Elaboración propia.

```
# This file describes the network interfaces available on your
    system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
address 192.168.10.130
netmask 255.255.255.0
gateway 192.168.10.254
dns-nameservers 200.83.1.4 190.160.0.14 200.30.192.15

iface nat0-eth0 inet manual
```

En este caso, los servidores DNS fueron proporcionados por la ISP y pueden ser cambiados si es necesario. Para confirmar los cambios, es necesario reiniciar el servicio de *networking*:

```
sudo /etc/init.d/networking restart
```

5. **Comprobar conectividad**: En este punto, *Mininet* debería tener conectividad con *VyOS* y acceso a internet (figura 16).

6. **(Opcional) Conexión SSH**: Habilitar la conexión SSH desde un *host* externo, mediante el puerto 2223. Para esto, es necesario configurar reglas de *port-forwarding* en *VyOS*:

```
configure
set nat destination rule 101 description 'SSH MININET PORT:2223'
set nat destination rule 101 destination port 2223
set nat destination rule 101 translation port 22
set nat destination rule 101 translation address 192.168.10.130
set nat destination rule 101 inbound-interface eth0
```

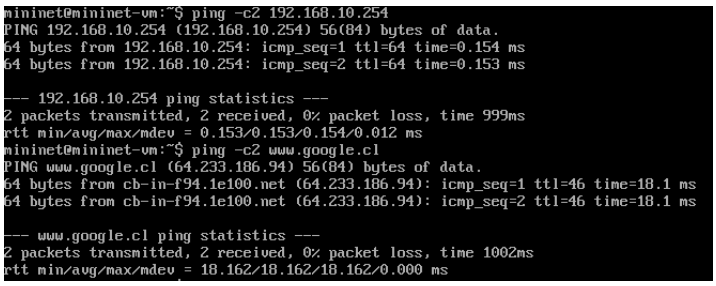



Figura 16: Probando la conectividad de *Mininet*.
Fuente: Elaboración propia.

```
set nat destination rule 101 protocol tcp
commit
save
```

### 4.3.1. COMPROBAR MININET

La manera más efectiva de comprobar la instalación de *Mininet* es creando una topología. Para esto, solo es necesario ejecutar:

```
sudo mn
```

Este comando creará una topología básica, que consiste en dos *hosts* conectados a un *switch*. En la figura 17 se aprecia el *output* de *mn*, mientras que en la figura 18 se muestra la topología creada.

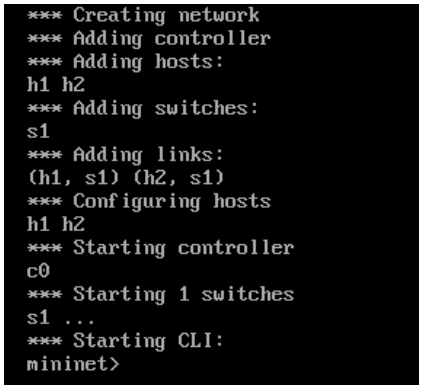



Figura 17: Creación de una topología.
Fuente: Elaboración propia.

*Mininet* normalmente trabaja con *OpenFlow* v1.0, pero esto se puede cambiar mediante parámetros de ejecución. A su vez, si no se especifica un controlador, *Mininet* creará

por omisión un controlador *ovsc* de auto aprendizaje, basado en OVS, que se encargará de llenar las tablas de flujos de cada uno de los *switch* virtualizados.


Figura 18: Visualización de una topología.
Fuente: Elaboración propia.

### 4.3.2. COMANDOS BÁSICOS

En esta sección se describen una serie de comandos básicos y útiles de *Mininet*. Éstos solo pueden ser utilizados luego de haber creado una topología.

- **net**: Permite verificar la configuración de una topología, tal como se aprecia en la figura 19.

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
```

Figura 19: Resultado *net*.
Fuente: Elaboración propia.

- **pingall**: Comprueba si los *hosts* de la topología pueden hacer ping entre si, tal como se muestra en la figura 19.

```
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Figura 20: Resultado *pingall*.
Fuente: Elaboración propia.

- **dpctl dump-flows -O OpenFlow10**: Se utiliza para ver el contenido de las tablas de flujo de los *switches* (figura 21). El parámetro *-O OpenFlow10* indica que la consulta se realiza para tablas compatibles con *OpenFlow* v1.0.
- **help**: Muestra una lista de todos los comandos disponibles en *Mininet*.
- **exit**: Se utiliza para salir de **Mininet**.

```
*** s1 ------------------------------------------------------------------------
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=1.441s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=1, priorit
y=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=16:f3:5b:f1:6d:c0,dl_dst=6e:bd:f8:45:ab:f8,nw_src=10.0
.0.1,nw_dst=10.0.0.2,nw_tos=0,icmp_type=0,icmp_code=0 actions=output:2
 cookie=0x0, duration=1.443s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=1, priorit
y=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=6e:bd:f8:45:ab:f8,dl_dst=16:f3:5b:f1:6d:c0,nw_src=10.0
.0.2,nw_dst=10.0.0.1,nw_tos=0,icmp_type=0,icmp_code=0 actions=output:1
 cookie=0x0, duration=1.441s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=1, priorit
y=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=6e:bd:f8:45:ab:f8,dl_dst=16:f3:5b:f1:6d:c0,nw_src=10.0
.0.2,nw_dst=10.0.0.1,nw_tos=0,icmp_type=8,icmp_code=0 actions=output:1
mininet>
```

Figura 21: Tabla de flujo en OVS.
Fuente: Elaboración propia.

### 4.3.3. CREAR TOPOLOGÍAS

*Mininet*, mediante el parámetro *–topo*, soporta la creación de tres tipos de topologías: *single*, *linear* y *tree*.

- **Topología single**: Consiste en un *switch* conectado una cantidad *X* de *hosts*.

```
sudo mn --topo single,X
```

En la figura 22 se muestra una topología *single* con *X=10 hosts*.

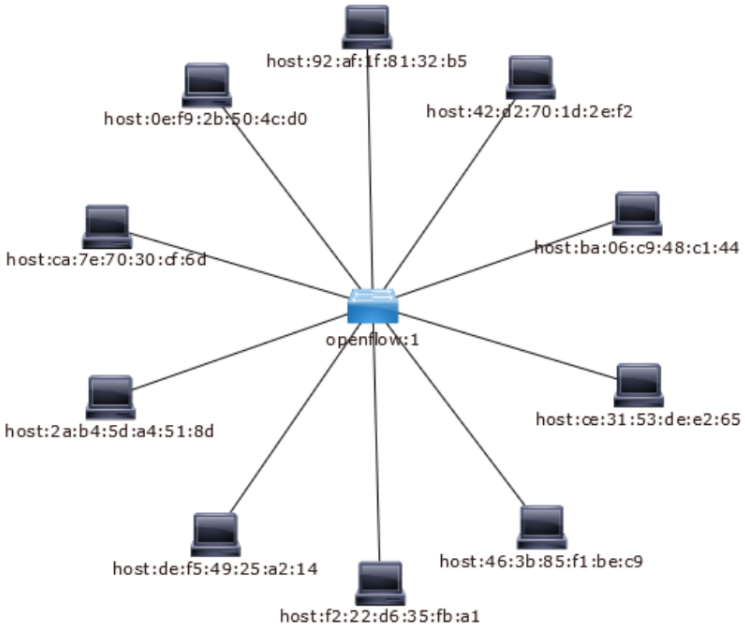



Figura 22: Topología *single (X=10)*.
Fuente: Elaboración propia.

- **Topología linear**: Consiste en *X switches*, uno detrás de otro, y cada uno con asociado a un *host*.

```
sudo mn --topo linear ,X
```

En la figura 23 se muestra una topología *linear* con *X=3 switches*.

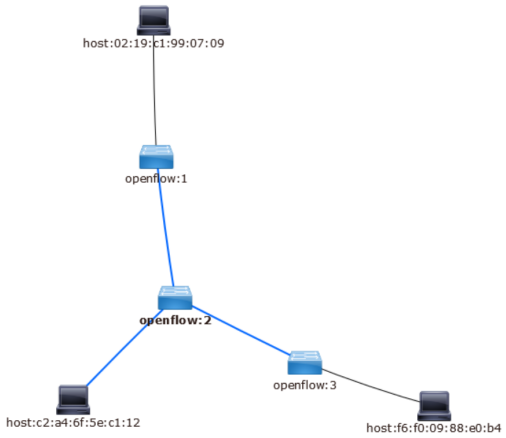



Figura 23: Topología *linear (X=3)*.
Fuente: Elaboración propia.

- **Topología tree**: Consiste en un árbol de profundidad *X*, con *Y* hijos por nodo.

```
sudo mn --topo tree ,depth=X,fanout=Y
```

En la figura 24 se muestra una topología *tree* de profundidad *X=2* e *Y=2* hijos por nodo.

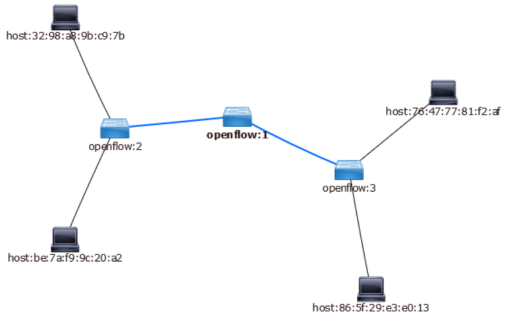



Figura 24: Topología *tree (X=2, Y=2)*.
Fuente: Elaboración propia.

### 4.3.4. TOPOLOGÍAS PARAMETRIZADAS

Una de las características más importantes de *Mininet* es el soporte de topologías parametrizadas. Con una pocas líneas de código *Python*, es posible crear una topología flexible, que puede ser configurada en base a parámetros y reusable en múltiples experimentos. Por ejemplo, el *script* 1 sirve para crear una topología *single*, en donde el parámetro ***n_nodos*** corresponde a la cantidad de nodos conectados al *switch*.

Listing 1: (*Script*) Topología *Single* en Python.

```
#!/usr/bin/python
from mininet.node import Host,CPULimitedHost
from mininet.topo import Topo
from mininet.link import TCLink

n_nodos = 5

#Creando una topologia Single
#Clase sin parametros de rendimiendo
class Topo_Lab(Topo):
        def build(self,n):
                #Agrega Host y Switches
                s1=self.addSwitch('s1')
                for x in xrange(n):
                        h = self.addHost('h%s' %(x + 1))
                        #Agrega links
                        self.addLink(h, s1)

#Clase con parametros de rendimiendo
class Topo_Lab_With_Params(Topo):
    def build(self,n):
        #Add hosts and switches
        s1=self.addSwitch('s1')
        for x in xrange(n):
            # A cada host se le asigna el 50% del CPU del sistema
            h = self.addHost('h%s' %(x + 1), cpu=.5/n)
            #Add links
            # 10 Mbps, 5ms delay, 2% loss, 1000 packet queue
            self.addLink(h, s1,bw=10, delay='5ms', loss=2,
             max_queue_size=1000, use_htb=True)


topos={'topo_example':lambda:Topo_Lab(n=n_nodos),
'topo_example_with_params':lambda:Topo_Lab_With_Params(n=n_nodos)}
```

A continuación, se explican algunas clases, métodos y funciones relevantes:

- **Topo**: La clase base para las topologías *Mininet*.
- **build()**: El método a sobrescribir en la clase personalizada que se hereda de *Topo*.

- **addSwitch()**: Agrega un *switch* a la topología y retorna el nombre del *switch*.
- **addHost()**: Agrega un *host* a la topología y retorna el nombre del *host*.
- **addLink()**: Agrega un *link* bidireccional a la topología.

Para crear la topología parametrizada:

```
sudo mn --custom topo_1.py --topo topo_example
```

### 4.3.5. PARÁMETROS DE RENDIMIENTO

*Mininet* permite establecer parámetros de aislamiento y de limitación de rendimiento en sus *scripts*, a través de las clases *TCLink* y *CPULimitedHost* respectivamente y, de esta manera, emular topologías más realistas. Se recomienda revisar la documentación[31] de la API de *Mininet* para más detalles.

Para ejemplificar estas funciones, nuevamente, se ejecutará el *script* 1, pero con la topología *topo_example_with_params*. A continuación, se describe en qué influyen estos parámetros:

- **self.addHost(name, cpu=f)**: Permite asignar una fracción de los recursos de la CPU al *host* virtual.
- **self.addLink(node1, node2, bw=10, delay='5ms', max_queue_size=1000, loss=10, use_htb=True)**: Agrega un enlace bidireccional con parámetros personalizados. El *bandwith* se mide en *Mbit*, el *delay* es un *string* con unidades de tiempo ('5[*ms*]', '100[*us*]', '1[*s*]'), el *loss* en porcentajes y el *max_queue_size* en cantidad de paquetes.

Para crear la topología, basta con ejecutar (figura 25):

```
sudo mn --custom topo_1.py  --topo topo_example_with_params --link tc --
    host cfs
```

Es importante incluir los parámetros ***–link tc –host cfs***, ya que de lo contrario no funcionará. En definitiva, se obtendrá una topología similar a *topo_example* de la sección 4.3.4, donde la única diferencia radicará en el rendimiento.

[31] `http://mininet.org/api/annotated.html`

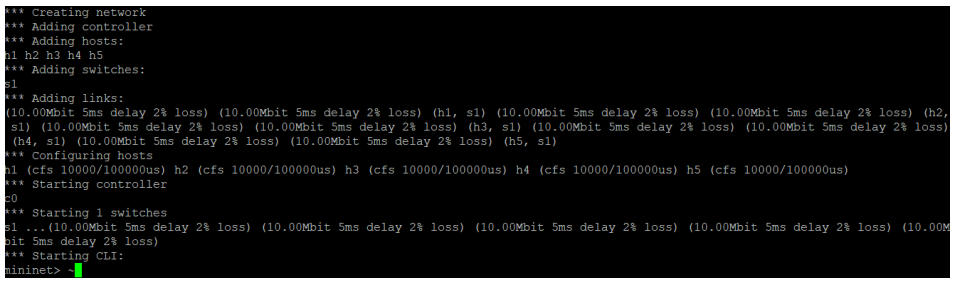



Figura 25: Topología Parametrizada.
Fuente: Elaboración propia.

### 4.3.6. MEDIR RENDIMIENTO

Para medir el rendimiento de una topología parametrizada, es posible hacerlo mediante el comando *iperf*, el cual se encargará de medir el ancho de banda máximo entre dos *hosts*.

```
iperf h1 h2
```

La salida corresponde al *goodput* y *troughtput* medido entre los *host h1* y *h2*. Al comparar las figuras 26 y 27, queda en evidencia cómo afecta el uso de parámetros de rendimiento.


Figura 26: *Iperf* en topología *topo_example*.
Fuente: Elaboración propia.

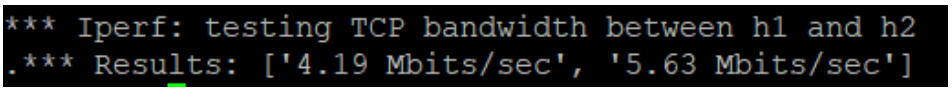



Figura 27: *Iperf* en topología *topo_example_with_params*.
Fuente: Elaboración propia.

### 4.3.7. CREACIÓN MANUAL DE ENTRADAS DE FLUJO

*Mininet* utiliza *Open vSwitch* como *switch* virtual, compatible con *OpenFlow*. Estos *switches* virtuales trabajan bajo el paradigma *match-plus-action*, redireccionando paquetes en base sus tablas de flujo. Típicamente, estos flujos son instalados y configurados por el controlador de la red, sin embargo, si se requiere un control a grano fino, también es posible hacerlo de forma manual.

Para crear entradas de flujo se utilizará el comando *ovs-ofctl*. Los flujos funcionan en base a *match entries* que se configuran mediante parámetros. En la figura 7 se muestra

una lista de campos que pueden hacer *match*. Los campos que especifiquen protocolos deberán hacerlo con su respectivo *etherType* (cuadro 6). Es importante señalar que las entradas en la tabla de flujos deben estar en formato normal. Esto quiere decir que un flujo solo puede especificar un campo de L3 solo si especifica un protocolo particular de L2. La misma regla se aplica para campos de L4, requiriendo la especificación de protocolos particulares de L2 y L3. Si no se sigue esta regla, *ovs-ofctl* lo advertirá y los campos especificados serán tratados como *wildcards*. Se recomienda revisar la documentación[32] de *ovs-ofctl* para más detalles.

Tabla 6: *EtherTypes*.
Fuente: Elaboración propia.

| **Keyword** | **dl_type** | **nw_proto** |
|---|---|---|
| ip | 0x800 | |
| arp | 0x806 | 1 (request), 2 (response) |
| rarp | 0x8035 | |
| icmp | 0x800 | 1 |
| tcp | 0x0800 | 6 |
| udp | 0x0800 | 17 |
| ipv6 | 0x86dd | |
| tcp6 | 0x86dd | 6 |
| udp6 | 0x86dd | 17 |
| icmp6 | 0x86dd | 58 |

#### 4.3.7.1. COMANDO OVS-OFCTL

Para ejemplificar el uso de *ovs-ofctl*, se utilizará una topología básica de *Mininet* sin un controlador asociado:

```
sudo mn --controller=none
```

El parámetro –***controller=none*** le indica a *Mininet* que no instancie un controlador.

Para crear una entrada de flujo de ejemplo en *s1*:

```
sh ovs-ofctl add-flow s1 priority=500,action=normal
```

**Nota**: OVS implementa un *switch OpenFlow-hybrid*, por lo que es compatible con la acción *normal*.

Para ver los flujos instanciados en *s1*:

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

[32] `http://www.openvswitch.org/support/dist-docs/ovs-ofctl.8.txt`

De la figura 28, se obtiene la siguiente información adicional:

- ***duration*=570.75s** - Tiempo en segundos que ha estado la entrada en la tabla de flujos.
- ***n_packets*=24** - Cantidad de paquetes que han hecho *match* con esa entrada.
- ***n_bytes*=1680** - Cantidad de *Bytes* procesados por esa entrada.
- ***idle_age*=519** - Cantidad de tiempo en segundos que ha pasado desde que un paquete hizo *match* con esa entrada.
- ***priority*=500** - Prioridad de la entrada. Si el paquete hace match con dos o más entradas, se utilizará la que tiene una prioridad más alta. La prioridad por omisión es 32768.

```
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=570.759s, table=0, n_packets=24, n_bytes=1680, idle_age=519, priority=500 actions=NORMAL
```

Figura 28: Tabla de flujo del *switch s1*.
Fuente: Elaboración propia.

Para eliminar todos los flujos instalados en *s1*:

```
sh -ovsofctl -delflows s1
```

## 4.4. INSTALAR Y CONFIGURAR EL CONTROLADOR OPENDAYLIGHT

El tercer paso para implementar el entorno virtual es instalar el controlador *OpenDaylight*. Este funciona en base a *Java* y *Apache Maven*, por lo que cualquier distribución de *Linux* serviría como *host*. En este laboratorio se utilizó *CentOS 7* en su versión DVD ISO[33], en conjunto con la versión *Nitrogen* 0.7.2[34] de ODL.

- **Instalar la VM**: Se le asignaron los siguientes recursos para un rendimiento óptimo:
  - **Núclueos CPU**: 4
  - **RAM**: 8GB

[33] `https://www.centos.org/download/`

[34] `https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendaylight/integration/karaf/`

- **Almacenamiento**: 20GB

- **Configurar el adaptador de red**: Este paso es importante, ya que permitirá conectar la VM a *VyOS*:

  - **Network Adapter**: Modo *Custom*, seleccionar el adaptador *VMnet10* (cuadro 5).

- **Configurar la interfaz de red**: La interfaz de red se configurará con la IP estática 192.168.10.120. El nombre de la interfaz dada por el SO es *ens33*, por lo que se debe modificar el archivo correspondiente con los siguientes parámetros. Las direcciones DNS pueden ser cambiadas si se estima conveniente:

```
sudo vi /etc/sysconfig/network-scripts/ifcfg-ens33
```

```
TTYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens33
UUID=41886de0-caa9-49a5-b8f9-e4a0d5a636fe
DEVICE=ens33

#Modificar
ONBOOT=yes
BOOTPROTO=static

#Agregar
IPADDR=192.168.10.120
NETMASK=255.255.255.0
GATEWAY=192.168.10.254
DNS1=200.83.1.4
DNS2=190.160.0.14
DNS3=200.30.192.15
```

  Finalmente, reiniciar el servicio de *networking*:

```
sudo systemctl restart network
```

- **Actualizar el SO**:

```
sudo yum -y update
```

- **Instalar Java**:

```
sudo yum install java-1.8.0-openjdk
```

- **Deshabilitar Firewall**: Para simplificar la configuración, se deshabilitará el servicio *firewalld* de *CentOS 7*, ya que el *router* virtual se encargará de esta tarea:

```
sudo systemctl stop firewalld
sudo systemctl disable firewalld
```

- **Instalar OpenDaylight**: ODL, por conveniencia, se instalará en el directorio */opt/odl/*. Desde la carpeta en donde fue descargado el controlador, ejecutar los siguientes comandos:

```
sudo mkdir /opt/odl
sudo mv  karaf-0.7.2.tar.gz /opt/odl/
cd /opt/odl/
sudo tar -xzf karaf-0.7.2.tar.gz
```

  Por comodidad, se creará un enlace simbólico que apuntará al directorio de instalación:

```
sudo ln -s karaf-0.7.2 nitrogen
```

  Con esta configuración, se podrá utilizar el directorio */opt/odl/nitrogen* como ruta de acceso a la instancia.

- **(Opcional) Conexión SSH**: En primer lugar, hay que inicializar el servicio:

```
sudo systemctl start sshd.service
```

  Luego, hay que habilitar la conexión SSH desde un *host* externo a través del puerto 2222. Para esto, es necesario configurar reglas de *port-forwarding* en *VyOS*:

```
configure
set nat destination rule 100 description 'SSH ODL PORT:2222'
set nat destination rule 100 destination port 2222
set nat destination rule 100 translation port 22
set nat destination rule 100 translation address 192.168.10.120
set nat destination rule 100 inbound-interface eth0
set nat destination rule 100 protocol tcp
commit
save
```

### 4.4.1. COMPROBAR OPENDAYLIGHT

*OpenDaylight* se ejecuta mediante *karaf*. *Apache karaf* es un *runtime* basado en OSGi, que provee un contenedor ligero, en donde conviven varios plugin y aplicaciones.

Al ejecutar *karaf*, se debería obtener un resultado similar a la figura 29:

```
sudo /opt/odl/nitrogen/bin/karaf
```


Figura 29: Consola de comandos *karaf*.
Fuente: Elaboración propia.

Posteriormente, el SO empezará a escuchar los siguientes puertos TCP:

- **44444**: *Karaf rmiServerPort*.
- **8101**: *Karaf SSH shell port*.
- **1099**: *Karaf rmiRegistryPort*.

Para ver todos los plugin disponibles para *karaf*:

```
feature:list
```

La lista de puertos utilizados por otros plugin *karaf*, se encuentra disponible en el siguiente enlace[35].

### 4.4.2. INSTALAR MÓDULOS DE OPENDAYLIGHT

*OpenDaylight* es una plataforma modular, a la cual se le pueden acoplar módulos y aplicaciones. En este proyecto se utilizarán los módulos *L2 Switch*, *DluxApps*, *VTN Manager*.

#### 4.4.2.1. INSTALAR L2 SWITCH

Para instalar el módulo *L2 Switch*, ejecutar el siguiente comando en la consola *karaf* de ODL, que incluye todos los servicios definidos en la sección 3.2.1:

[35] `https://wiki.opendaylight.org/view/Ports`

```
feature:install features-l2switch
```

Cabe destacar, que este módulo funciona *"out of the box"*, a menos que se necesite realizar una configuración avanzada. Al conectar el controlador ODL con el emulador *Mininet*, *L2 Switch* se encargará de instalar automáticamente los flujos necesarios para lograr la interconectividad entre los nodos de red.

#### 4.4.2.2. INSTALAR DLUXAPPS

Para instalar el módulo *DluxApps*, ejecutar el siguiente comando en la consola *karaf* de ODL, que incluye todos los servicios definidos en la sección 3.2.2:

```
feature:install features-dluxapps
```

Esta acción hará que el SO escuche los siguientes puertos TCP:

- **8080**: *Jetty: MD-SAL RESTCONF, Jolokia.*
- **8181**: *Jetty: MD-SAL RESTCONF, Jolokia, AAA auth endpoints, Dlux, DluxApps.*

A partir de ahora se podrá establecer una conexión HTTP al puerto 8181 y acceder a la interfaz WEB, utilizando la URL:

```
192.168.10.120:8181/index.html
```

En la figura 30 se aprecia la interfaz WEB si es que se pudo establecer la conexión. Las credenciales asignadas automáticamente por el módulo son las siguientes:

- **username**: admin
- **password**: admin

**Opcional**: Para establecer conexiones TCP desde un *host* externo a través del puerto 8181, es necesario configurar reglas de *port-forwading* en *VyOS*:

```
configure
set nat destination rule 102 description 'DLUXAPPS PORT:8181'
set nat destination rule 102 destination port 8181
set nat destination rule 102 translation port 8181
set nat destination rule 102 translation address 192.168.10.120
set nat destination rule 102 inbound-interface eth0
set nat destination rule 102 protocol tcp
commit
save
```

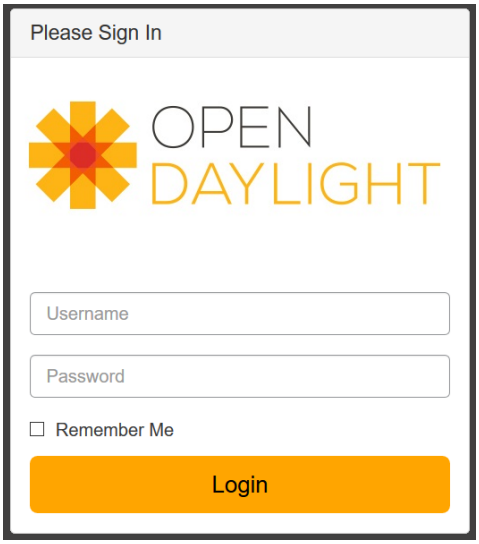



Figura 30: Interfaz gráfica proporcionada por *DluxApps*.
Fuente: Elaboración propia.

#### 4.4.2.3. INSTALAR VTN MANAGER

Es un plugin que interactúa con otros módulos para implementar los componentes del modelo VTN, además de ofrecer una interfaz REST para configurar tales componentes. Adicionalmente, permite la integración con la API de *Networking L2* de *OpenStack*, denominada *Neutron* (referirse al anexo 6.3).

A continuación, se describen los componentes *karaf* de que se instalan con VTN Manager:

- **odl-vtn-manager**: Provee la API JAVA de *VTN Manager*.
- **odl-vtn-manager-rest**: Provee la API REST de *VTN Manager*.
- **odl-vtn-manager-neutron**: Provee la integración con la interfaz *Neutron*.

Para instalar todos estos componentes como un *bundle*, ejecutar el siguiente comando en la consola *karaf* de ODL:

```
feature:install features-vtn-manager
```

## 4.5. INSTALAR Y CONFIGURAR LA APLICACIÓN DE RED VTN COORDINATOR

El último paso para implementar el entorno virtual es instalar *VTN Coordinator*, una aplicación de red externa que interactúa directamente con la API REST de *VTN Manager*, ofreciendo su propia interfaz REST para ser utilizada indistintamente por usuarios y aplicaciones *northbound*, con el objetivo de orquestar la virtualización de VTN entre múltiples instancias de *OpenDaylight*. En la arquitectura *OpenDaylight*, esta aplicación es parte de la capa de aplicaciones de red, orquestación y servicios.

En los experimentos se utilizará *VTN Coordinator v6.6.0*[36]. Esta aplicación será instalada en la máquina virtual con *CentOS 7*, junto con el controlador *OpenDaylight*. Esta configuración no es ideal, ya que se recomienda instalar las aplicaciones en máquinas distintas, sin embargo, será utilizada por simplicidad.

Antes de instalar *VTN Coordinator*, hay que instalar las siguientes dependencias:

```
yum install perl-Digest-SHA uuid libxslt libcurl unixODBC json-c bzip2
rpm -ivh http://yum.postgresql.org/9.3/redhat/rhel-6-x86_64/pgdg-redhat93
    -9.3-3.noarch.rpm
yum install postgresql93-libs postgresql93 postgresql93-server
    postgresql93-contrib postgresql93-odbc
```

Posteriormente, se procederá a instalar *VTN Coordinator* en el directorio */usr/local/vtn*:

```
tar -C/ -jxvf distribution.vtn-coordinator-6.6.0-bin.tar.bz2
```

Luego, hay que inicializar la base de datos de *VTN Coordinator*:

```
sudo /usr/local/vtn/sbin/db_setup
```

En este punto, la aplicación debería estar totalmente configurada para utilizarse por primera vez. Para iniciar *VTN Coordinator*:

```
sudo /usr/local/vtn/bin/vtn_start
```

Para detener *VTN Coordinator*:

```
/usr/local/vtn/bin/vtn_stop
```

[36] https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/vtn/distribution.vtn-coordinator/6.6.0/

### 4.5.1. COMPROBAR VTN COORDINATOR

Para verificar que la instalación se hizo correctamente, basta con ejecutar una consulta REST que debería devolver la versión de la API REST de VTN. Las credenciales por omisión son las siguientes:

- **user**: admin
- **password**: adminpass

Ejecutar la siguiente consulta REST:

```
curl --user admin:adminpass -H 'content-type: application/json' -X GET
    http://127.0.0.1:8083/vtn-webapi/api_version.json
```

Que debería responder lo siguiente:

```
{"api_version":{"version":"V1.4"}}
```

**Opcional**: Para realizar consultas REST desde un *host* externo a través del puerto 8083, es necesario configurar reglas de *port-forwading* en *VyOS*:

```
configure
set nat destination rule 101 description 'VTN COORDINATOR PORT:8083'
set nat destination rule 101 destination port 8083
set nat destination rule 101 translation port 8083
set nat destination rule 101 translation address 192.168.10.120
set nat destination rule 101 inbound-interface eth0
set nat destination rule 101 protocol tcp
commit
save
```

## 4.6. INTEGRAR MININET CON OPENDAYLIGHT

Para culminar con este capítulo y comprobar el funcionamiento del entorno virtual, en esta sección se integrará el emulador *Mininet* con *OpenDaylight*. Antes de empezar, es necesario asegurarse de tener una instancia del controlador funcionando, junto con los módulos *L2 Switch* (requerido) y *DluxApps* (opcional) instalados. La figura 31 representa la topología *out-of-band* a implementar.

Para empezar, se creará la topología en *Mininet*:

```
sudo mn --controller=remote,ip=192.168.10.120 --switch ovsk,protocols=
    OpenFlow13 --mac
```

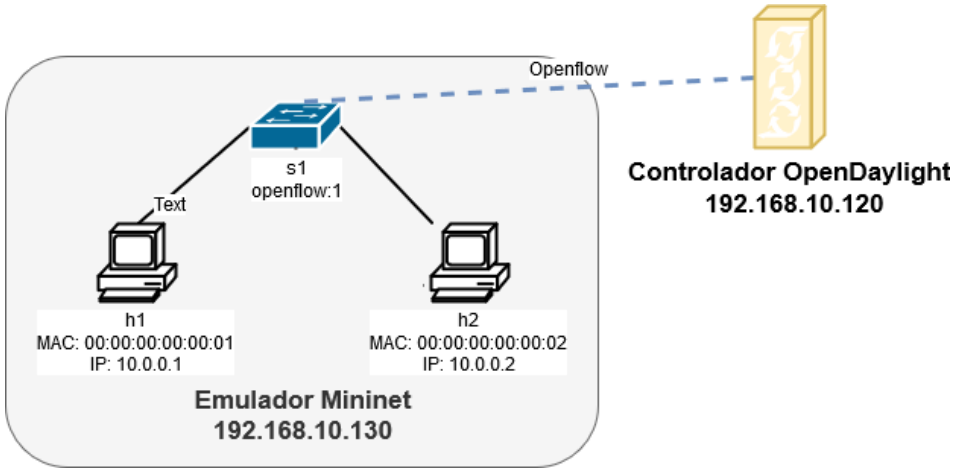



Figura 31: Topología de integración entre SDN y *Mininet*.
Fuente: Elaboración propia.

De este comando destacan dos parámetros:

- **–controller=remote,ip=192.168.10.120**: Se establecerá una conexión TCP con el controlador *OpenDaylight* a través del puerto 6653.
- **–switch ovsk,protocols=OpenFlow13**: El *switch* instanciado se ejecutará en modo *kernel* y será compatible con *OpenFlow* v1.3.

Con el comando *net* se obtendrá la configuración de la topología:

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
```

A partir de este punto, la topología de *Mininet* estará conectada al controlador *OpenDaylight*. El módulo *L2 switch* de ODL se encargará de instalar los flujos necesarios en el *switch* para lograr la interconectividad. Para comprobar esto, bastaría con hacer *pingall*. Si se instaló el módulo *DluxApps*, será posible observar la topología creada a través de la WEB GUI.

# CAPÍTULO 5
# LABORATORIO: EXPERIMENTACIÓN

## 5.1. LABORATORIO 1: MATCHING MANUAL

Este laboratorio tratará sobre el proceso más importante que realiza *OpenFlow*, la creación de entradas de flujo. Normalmente, la entidad encargada de generar estos registros corresponde al controlador, por lo que el usuario se abstrae de este proceso y rara vez está directamente involucrado. No obstante, es importante comprender cómo crear y administrar entradas de flujo de forma manual, para familiarizarse con el protocolo.

El laboratorio se dividirá en dos experiencias. La primera tratará sobe el redireccionamiento de paquetes a través del paradigma *matching-plus-action*, mientras que la segunda involucrará el *processing pipeline* de *OpenFlow*, implementando un *router* simulado, usando múltiples tablas de flujos de un *switch OpenFlow*. Es importante señalar que estos experimentos se realizarán con topologías *Mininet* sin controlador, por lo que todas las entradas deberán ser creadas manualmente.

### 5.1.1. EXPERIENCIA 1: MATCHING-PLUS-ACTION

En esta experiencia se ejemplificará el paradigma *matching-plus-action* utilizando entradas de flujo manuales. El objetivo es familiarizarse con la horizontalidad que proporciona *OpenFlow* al momento de manejar paquetes de red. La experiencia se dividirá en cuatro miniexperimentos independientes, pero relacionados. Cada uno de ellos representará el procesamiento de paquetes *OpenFlow* de cada una de las cuatro primeras capas del modelo OSI y tendrán un objetivo similar, establecer conectividad. Se utilizará la topología de la figura 32, generada por el *script* 2.

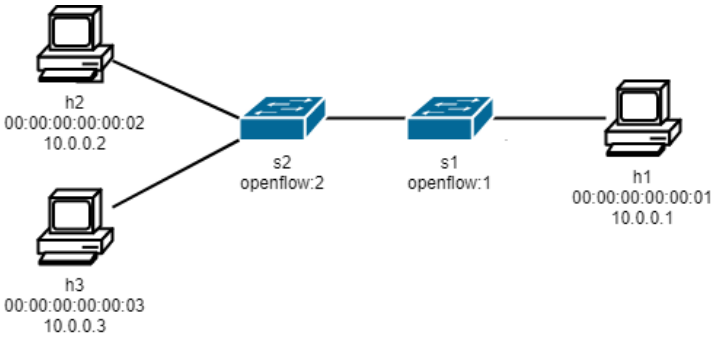



Figura 32: **L1E1**: (Topología) *Matching-plus-action*.
Fuente: Elaboración propia.

## CREANDO LA TOPOLOGÍA

Para crear la topología:

```
sudo mn --custom topo_2.py  --topo topo_mpa --controller=none --mac
```

El parámetro –***mac*** es opcional, pero muy útil, ya que asigna a los *hosts* direcciones MAC pequeñas y únicas, facilitando la lectura.

En la figura 33 se muestra la salida del comando *net*, que representa la configuración de la topología.

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s2-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth3
s2 lo:  s2-eth1:h2-eth0 s2-eth2:h3-eth0 s2-eth3:s1-eth2
```

Figura 33: **L1E1**: Configuración de topología.
Fuente: Elaboración propia.

**Nota**: Será necesario limpiar las entradas de flujo entre experimentos (referirse a la sección 4.3.7.1).

## CREANDO ENTRADAS DE FLUJO

1. **Matching de Capa 1 (L1)**: El objetivo será hacer *ping* entre las máquinas *h1* y *h2* utilizando solo *matching* L1, es decir, redireccionando puertos.

   - Para lograr la conectividad entre *h1* y *h2* se implementarán las siguientes reglas:

     *a*) Todo lo que llege por *s1-eth1*, salga por *s1-eth2*.
     *b*) Todo lo que llege por *s2-eth3*, salga por *s2-eth1*.
     *c*) Todo lo que llege por *s2-eth1*, salga por *s2-eth3*.
     *d*) Todo lo que llege por *s1-eth2*, salga por *s1-eth1*.

   Que se traducen en los siguientes flujos:

   ```
   #Switch s1
   sh ovs-ofctl add-flow s1 in_port=1,actions=output:2
   sh ovs-ofctl add-flow s1 in_port=2,actions=output:1

   #Switch s2
   sh ovs-ofctl add-flow s2 in_port=3,actions=output:1
   sh ovs-ofctl add-flow s2 in_port=1,actions=output:3
   ```

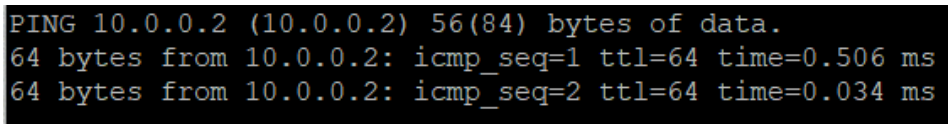



Figura 34: **L1E1**: *Ping h1* y *h2*.
Fuente: Elaboración propia.

- Ejecutando (*h1 ping -c2 h2*) se comprobará que se obtuvo el resultado deseado (figura 34).

2. **Matching de Capa 2 (L2)**: Lograr conectividad entre *h1* y *h2* con *matching* L2, es decir, redireccionando en base a direcciones MAC.

   - Agregar los flujos necesarios, para que los *switches* redireccionen consultas ARP, de lo contrario los *hosts* no podrán aprender las direcciones MAC:

   ```
   sh ovs-ofctl add-flow s1 dl_type=0x806,nw_proto=1,actions=ALL
   sh ovs-ofctl add-flow s2 dl_type=0x806,nw_proto=1,actions=ALL
   ```

   Notar que se han agregado dos nuevos parámetros, *dl_type = 0x806*, que corresponde al *etherType* de ARP y *nw_proto=1* indica que se trata de una solicitud. La *actions=ALL* indica que se debe redireccionar el paquete a todos los puertos, excepto al de entrada.

   - Agregar los siguientes flujos, en donde *dl_src* y *dl_dst* se utilizan para hacer *match* L2 con direcciones MAC:

   ```
   #Switch s1
   sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:01,dl_dst
       =00:00:00:00:00:02,actions=output:2
   sh ovs-ofctl add-flow s1 dl_src=00:00:00:00:00:02,dl_dst
       =00:00:00:00:00:01,actions=output:1
   #Switch s2
   sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:01,dl_dst
       =00:00:00:00:00:02,actions=output:1
   sh ovs-ofctl add-flow s2 dl_src=00:00:00:00:00:02,dl_dst
       =00:00:00:00:00:01,actions=output:3
   ```

3. **Matching de Capa 3 (L3)**: Lograr conectividad entre todos los *hosts* utilizando *matching* L3, es decir, utilizando el protocolo IP. Además, todo el tráfico que provenga desde *h2* será clasificado como *expedited forwarding*, por lo que tendrá un valor DSCP de 46.

   - Resolver el protocolo ARP. Se omitirá el parámetro *nw_proto* para que la entrada resuelva *requests* y *responses*:

   ```
   sh ovs-ofctl add-flow s1 dl_type=0x806,actions=ALL
   sh ovs-ofctl add-flow s2 dl_type=0x806,actions=ALL
   ```

- Agregar los siguientes flujos, en donde *nw_src* y *ns_dst* se utilizan para hacer *match* L3 con direcciones IP.

```
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.0/24,nw_dst
    =10.0.0.0/24,actions=normal
sh ovs-ofctl add-flow s2 priority=500,dl_type=0x800,nw_src
    =10.0.0.0/24,nw_dst=10.0.0.0/24,actions=normal
```

- Para clasificar el paquete como *expedited forwarding*:

```
sh ovs-ofctl add-flow s2 priority=800,dl_type=0x800,nw_src
    =10.0.0.2,actions=mod_nw_tos:184,normal
```

En la figura 35 se comprueba que se están modificando los paquetes enviados desde *h2*.

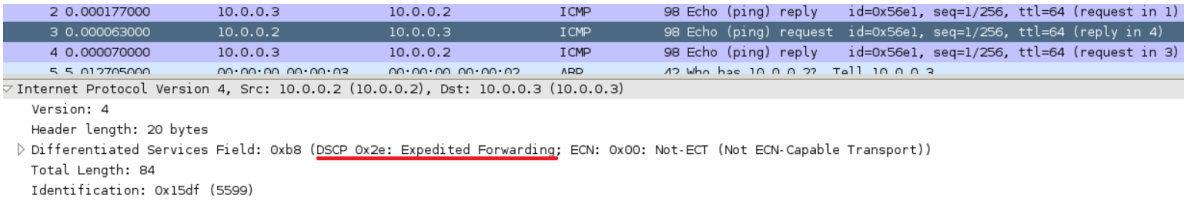



Figura 35: **L1E1**: Análisis *Ping h1* y *h2* en *Wireshark*.
Fuente: Elaboración propia.

4. **Matching de Capa 4 (L4)**: Correr un servidor HTTP en *h3* y permitir consultas solo desde *h2*, bloqueando *h1*. Esta configuración simula el comportamiento de un *firewall*.

- Correr un servidor HTTP, puerto 80 en *h3*:

```
h3 python -m SimpleHTTPServer 80 &
```

- Resolver protocolo ARP:

```
sh ovs-ofctl add-flow s1 dl_type=0x806,actions=ALL
sh ovs-ofctl add-flow s2 dl_type=0x806,actions=ALL
```

- Crear entradas de flujo para permitir redireccionamiento L2:

```
#Switch s1
sh ovs-ofctl add-flow s1 dl_type=0x806,actions=ALL
sh ovs-ofctl add-flow s1 dl_type=0x800,nw_src=10.0.0.0/24,nw_dst
    =10.0.0.0/24,actions=normal
#Switch s2
sh ovs-ofctl add-flow s2 dl_type=0x806,actions=ALL
sh ovs-ofctl add-flow s2 dl_type=0x800,priority=400,nw_src
    =10.0.0.0/24,nw_dst=10.0.0.0/24,actions=normal
```

- Bloquear conexión TCP al puerto 80 entre *h1* y *h3*:

```
sh ovs-ofctl add-flow s2 dl_type=0x800,priority=500,nw_proto=6,
    nw_src=10.0.0.1,nw_dst=10.0.0.3,tp_dst=80,actions=drop
```

  Esta entrada indica que todas las consultas TCP hacia el puerto 80 desde *h1* a *h3* sean descartadas.

- Para probar la configuración, se puede utilizar el comando *curl*:

```
h2 curl -v h3
h1 curl -v h3
```

  Como se aprecia en la figura 36, sólo la consulta desde *h2* obtendrá respuesta, la otra fallará.

```
mininet> h2 curl -v h3
* Rebuilt URL to: 10.0.0.3/
* Hostname was NOT found in DNS cache
*   Trying 10.0.0.3...
* Connected to 10.0.0.3 (10.0.0.3) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: 10.0.0.3
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/2.7.6
< Date: Fri, 06 Sep 2019 23:55:52 GMT
< Content-type: text/html
< Content-Length: 33
< Last-Modified: Fri, 06 Sep 2019 23:15:33 GMT
<
SimpleHTTPServer Funcionando =)
* Closing connection 0
```

```
mininet> h1 curl -v h3
* Rebuilt URL to: 10.0.0.3/
* Hostname was NOT found in DNS cache
*   Trying 10.0.0.3...
* connect to 10.0.0.3 port 80 failed: Connection timed out
* Failed to connect to 10.0.0.3 port 80: Connection timed out
* Closing connection 0
curl: (7) Failed to connect to 10.0.0.3 port 80: Connection timed out
```

Figura 36: **L1E1**: Consultas al servidor HTTP en *h3* con *curl*.
Fuente: Elaboración propia.

## ANÁLISIS DE RESULTADOS

En cada uno de estos miniexperimentos se logró establecer un resultado similar utilizando distintas condiciones de *match*, a través del paradigma *matching-plus-action*. Dada la horizontalidad en el manejo de paquetes en *OpenFlow*, los *switches* son capaces de resolver cualquier tipo de protocolo de red (como ARP) si es que tienen instaladas las entradas de flujo indicadas.

Esta experiencia sirve como una pequeña demostración de la posibilidades que entrega *OpenFlow* como protocolo *southbound*. Para alcanzar el verdadero potencial de *OpenFlow*, es necesario utilizarlo como API *southbound* en conjunto con un controlador, como se verá, posteriormente, en los experimentos de la sección 5.2.

### 5.1.2. EXPERIENCIA 2: MATCHING CON MÚLTIPLES TABLAS

El objetivo de esta experiencia será implementar la topología de la figura 37, generada por el *script* 3. En ella se describe uno topología similar a la experiencia anterior, con la diferencia de que, en este caso, se simulará el funcionamiento de un *router* básico en *s1* en base a entradas de flujo y múltiples tablas. En este caso en particular, se utilizará *OpenFlow* v1.3.0. El *switch s1* se encargará de hacer ACL, NAT y enrutamiento usando el *processing pipeline* de *OpenFlow*. Se configurarán dos subredes distintas, en donde *h2* y *h3* pertenecerán a 10.0.0.0/24, mientras que *h1* pertenecerá a 10.0.1.0/24. La subred 10.0.0.0/24 estará detrás de un NAT simulado y *h3* no podrá comunicarse con *h1*.

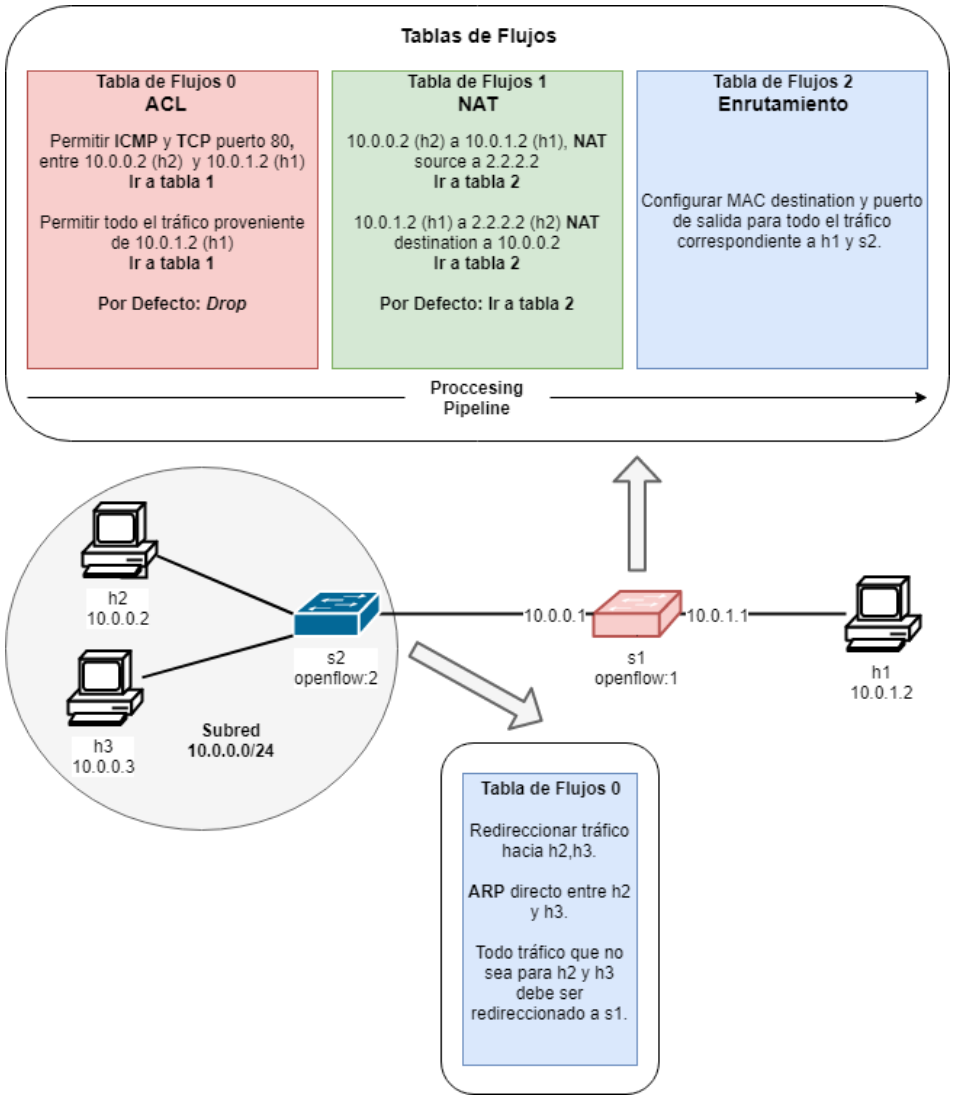



Figura 37: **L1E2**: (Topología) *Matching* con múltiples tablas.
Fuente: Elaboración propia.

## CREANDO LA TOPOLOGÍA

1. Crear la topología en *Mininet*:

```
sudo mn --custom topo_5.py --topo topo_multitable --controller=
    none --switch ovsk,protocols=OpenFlow13 --mac
```

Utilizando el comando *net* se obtendrá la configuración de la topología:

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s2-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth3
s2 lo:  s2-eth1:h2-eth0 s2-eth2:h3-eth0 s2-eth3:s1-eth2
```

## CREANDO ENTRADAS DE FLUJO

2. Agregar la entradas de flujo al *switch s1*, simulando el comportamiento de un *router*:

```
#Tabla 0 - ACL
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,icmp,nw_src
    =10.0.0.2,nw_dst=10.0.1.2, actions=resubmit(,1)"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,tcp,nw_src=10.0.0.2,
    nw_dst=10.0.1.2,tp_dst=80,actions=resubmit(,1)"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,ip,nw_src=10.0.1.2,
    actions=resubmit(,1)"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=0,priority=0,actions=
    drop"

#Tabla 1 - NAT
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,ip,nw_src=10.0.0.2,
    nw_dst=10.0.1.2,actions=mod_nw_src=2.2.2.2,resubmit(,2)"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,ip,nw_src=10.0.1.2,
    nw_dst=2.2.2.2,actions=mod_nw_dst=10.0.0.2,resubmit(,2)"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=1,priority=0,actions=
    resubmit(,2)"

#Tabla 2 - Enrutamiento
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=2,ip,nw_dst=10.0.1.2,
    actions=mod_dl_dst=00:00:00:00:00:01,output:1"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=2,ip,nw_dst=10.0.0.2,
    actions=mod_dl_dst=00:00:00:00:00:02,output:2"
sh ovs-ofctl -O OpenFlow13 add-flow s1 "table=2,ip,nw_dst=10.0.0.3,
    actions=mod_dl_dst=00:00:00:00:00:03,output:2"
```

La acción *resubmit* permite redireccionar paquetes para que sean procesados por tablas de flujo subsecuentes. Además, en vez de usar el *etherType* hexadecimal de los protocolos a hacer *match*, se utilizó directamente su identificador.

3. Agregar las entradas de flujo al *switch s2*, para que se redireccione correctamente el tráfico:

```
sh ovs-ofctl -O OpenFlow13 add-flow s2 "table=0,dl_dst
    =00:00:00:00:00:02,actions=output:1"
sh ovs-ofctl -O OpenFlow13 add-flow s2 "table=0,dl_dst
    =00:00:00:00:00:03,actions=output:2"
sh ovs-ofctl -O OpenFlow13 add-flow s2 "table=0,priority=1,arp,
    nw_dst=10.0.0.2,actions=output:1"
sh ovs-ofctl -O OpenFlow13 add-flow s2 "table=0,priority=1,arp,
    nw_dst=10.0.0.3,actions=output:2"
sh ovs-ofctl -O OpenFlow13 add-flow s2 "table=0,priority=0,actions=
    output:3"
```

## ANÁLISIS DE RESULTADOS

En este punto el *router* simulado debería tener las funcionalidades esperadas. Para comprobar, por ejemplo, que las reglas de NAT funcionan, basta con realizar ping entre *h2* y *h1*. Al analizar con *wireshark* tal acción (figura 38), se observan dos perspectivas distintas.

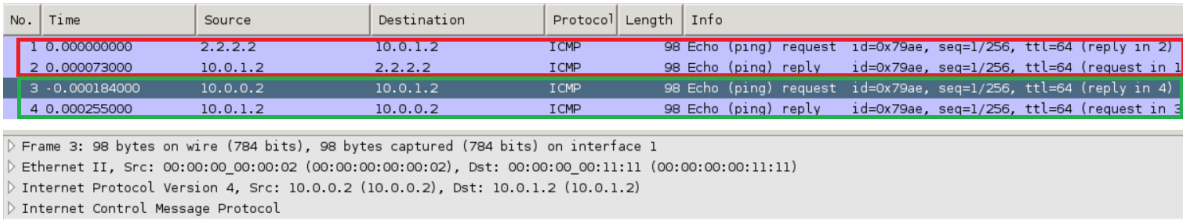



Figura 38: **L1E2**: *Ping* entre *h2* y *h1* en *Wireshark*.
Fuente: Elaboración propia.

Los registros que se encuentran dentro del conjunto verde corresponden a la perspectiva de *h2*. En ellos, se aprecia que la conexión que se realiza entre 10.0.0.2 (*h2*) y 10.0.1.2 (*h1*) se realiza de forma transparente, lo cual es esperable ya que en este caso se trata de pre NAT. Los registros dentro del conjunto rojo, en cambio, corresponden a la perspectiva de *h1*. En ellos se observa que el la IP *source* cambia a 2.2.2.2, la cual es asignada por el *switch s2*, confirmando que las reglas de NAT funcionan correctamente.

Las entradas de flujo configuradas en la tabla de ACL indican que es posible establecer una conexión TCP hacia *h1* a través del puerto 80. Para comprobar esto, se puede crear un servidor simple, que escuche el puerto 80 en *h1*, y luego hacer peticiones hacia *h1* con *curl*:

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

Como se aprecia en la figura 39, para probar esta configuración se realizaron peticiones *curl* desde *h2* y *h3* hacia *h1*, no obstante, solo las provenientes de *h2* obtuvieron respuesta.

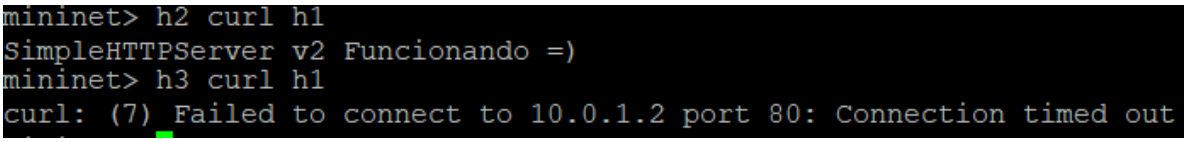



Figura 39: **L1E2**: Consultas al servidor HTTP en *h1* con *curl*.
Fuente: Elaboración propia.

## 5.2. LABORATORIO 2: VIRTUAL TENANT NETWORK

El controlador *OpenDaylight* soporta múltiples aplicaciones que, a través de interfaces *northbound*, pueden ser manipulados para programar la red física, siendo VTN una de las más interesantes. Como se mencionó en la sección 3.2.3, VTN es una aplicación de ODL que permite implementar redes de múltiples inquilinos sobre una red SDN física.

Al igual que el caso anterior, el laboratorio también se dividirá en experiencias separadas. En la primera se implementará una red *overlay* utilizando mapeo de puertos, con soporte para configuración de rutas. En la segunda se implementará una red *overlay* que contendrá dos VLANs, utilizando un mapeo por VLAN. Finalmente, en la tercera y última experiencia se implementará una red *overlay* para simular el encadenamiento de servicios de red. Para realizar este laboratorio, es necesario tener instalados el módulo *VTN Manager* y el programa *VTN Coordinator*. Cabe destacar, que el módulo no es compatible con *L2 Switch*, por lo que se recomienda desinstalarlo antes de utilizar *VTN Manager*.

### 5.2.1. EXPERIENCIA 1: MAPEANDO PUERTOS CON VTN

En esta experiencia se utilizará el *script* 4 para implementar la red descrita en la figura 40. En esta topología la conexión entre *s1* y *s2* tiene un *delay* de 1000[*ms*]. El objetivo será crear una VTN que permita la conectividad entre *h1* y *h2*, utilizando la ruta más rápida.

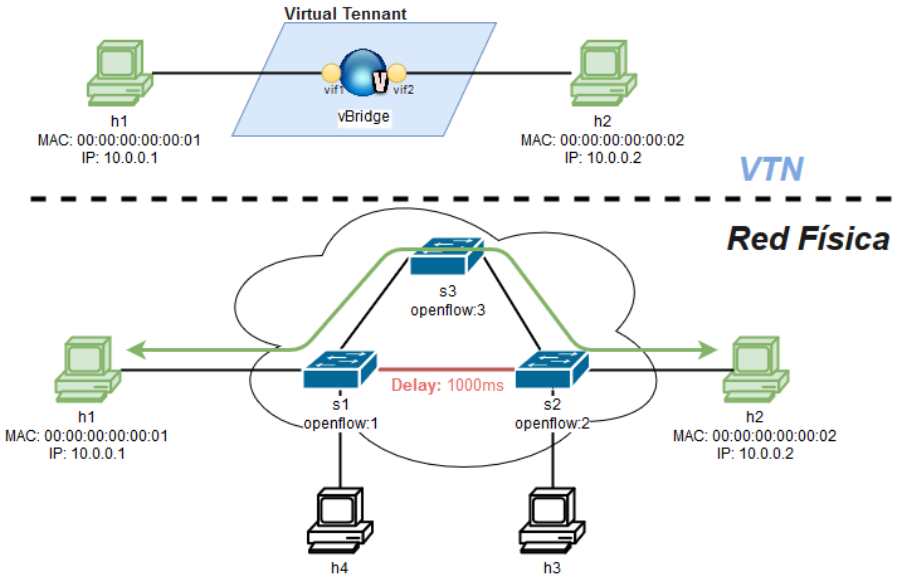



Figura 40: **L2E1**: (Topología) Mapeando puertos con VTN.
Fuente: Elaboración propia.

## CREANDO LA TOPOLOGÍA

1. Crear la topología en *Mininet*:

```
sudo mn --controller=remote,ip=192.168.10.120 --custom topo_3.py --
    topo topo_route --link tc --mac
```

Ejecutar *net* para obtener la configuración de la topología:

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s2-eth2
h4 h4-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h4-eth0 s1-eth3:s2-eth3 s1-eth4:s3-
    eth1
s2 lo:  s2-eth1:h2-eth0 s2-eth2:h3-eth0 s2-eth3:s1-eth3 s2-eth4:s3-
    eth2
s3 lo:  s3-eth1:s1-eth4 s3-eth2:s2-eth4
c0
```

2. Inicializar *VTN Coordinator* en *CentOS 7*:

```
sudo /usr/local/vtn/bin/vtn_start
```

## CREANDO LA VTN CON VTN COORDINATOR

3. Crear un controlador llamado *controladorUno*. En *ipaddr* se pondrá la IP del controlador ODL:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"controller": {"controller_id": "controladorUno", "
    ipaddr":"192.168.10.120","username":"admin","password":"admin", "
    type": "odc", "version": "1.0","auditstatus":"enable"}}' http
    ://127.0.0.1:8083/vtn-webapi/controllers.json
```

4. Crear una VTN llamada *vtn1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vtn" : {"vtn_name":"vtn1","description":"VTN EXP 1"
    }}' http://127.0.0.1:8083/vtn-webapi/vtns.json
```

## CREANDO VBRIDGE Y VINTERFACES

5. Crear un *vBridge* (*vBridge1*) y asignarlo a *vtn1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vbridge" : {"vbr_name":"vBridge1","controller_id":"
    controladorUno","domain_id":"(DEFAULT)" }}' http
    ://127.0.0.1:8083/vtn-webapi/vtns/vtn1/vbridges.json
```

6. Crear una interfaz *vif1* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif1","description": "h1
    gateway"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn1/vbridges/
    vBridge1/interfaces.json
```

La interfaz virtual *vif1* será mapeada al puerto *s1-eth1* del *switch openflow:1* de *Mininet*. El *host h1* está conectado directamente al puerto *s1-eth1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:1-s1-eth1
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1
    /interfaces/vif1/portmap.json
```

7. Crear una interfaz *vif2* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif2","description": "h2
    gateway"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn1/vbridges/
    vBridge1/interfaces.json
```

La interfaz virtual *vif2* será mapeada al puerto *s2-eth1* del *switch openflow:2* de *Mininet*. El *host h2* está conectado directamente al puerto *s2-eth1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth1
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1
    /interfaces/vif2/portmap.json
```

## CREANDO CONDICIONES DE FLUJO Y POLÍTICAS DE RUTA

8. Crear una nueva condición de flujo (*flow condition*) con el nombre *cond_1*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    condition:set-flow-condition -d '{"input":{"operation":"SET","
    present":"false","name":"cond_1", "vtn-flow-match":[{"vtn-ether-
    match":{},"vtn-inet-match":{"source-network":"10.0.0.1/32","
    protocol":1,"destination-network":"10.0.0.2/32"},"index":"1"},{"
    vtn-ether-match":{},"vtn-inet-match":{"source-network
    ":"10.0.0.2/32","protocol":1,"destination-network
    ":"10.0.0.1/32"},"index":"2"}]}}'
```

9. Crear una nueva política de ruta (*path policy*):

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-path-
    policy:set-path-policy -d '{"input":{"operation":"SET","id":
    "1","default-cost": "1000","vtn-path-cost": [{"port-desc":"
    openflow:1,3,s1-eth3","cost":"10000"},{"port-desc":"openflow:2,3,
    s2-eth3","cost":"10000"}]}}'
```

10. Crear un mapeo de rutas (*pathmap*) que mapee la condición *cond_1* con la política de ruta 1 y asignarla a *vtn1*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-path-map:
    set-path-map -d '{"input":{"tenant-name":"vtn1","path-map-list
    ":[{"condition":"cond_1","policy":"1","index": "1","idle-timeout
    ":"300","hard-timeout":"0"}]}}'
```

**LIMPIEZA**

11. Eliminar la VTN creada (*vtn1*):

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn:remove-
    vtn -d '{"input":{"tenant-name":"vtn1"}}'
```

```
sudo /usr/local/vtn/bin/vtn_stop
```

## ANÁLISIS DE RESULTADOS

En el anexo 7 se muestra la configuración final de *vtn1*. Hasta el punto 7, *vtn1* estará parcialmente configurado, es decir, permitirá la interconectividad entre sus *hosts*.

En la figura 41 se representa un ping entre *h1* y *h2* con un *delay* significativo de 2000[*ms*]. Esto ocurre porque VTN automáticamente asigna el camino más corto, el cual no es necesariamente el más rápido. En este caso, el camino por omisión incluye el enlace *s1-s2*, el cual tiene un *delay* de 1000[*ms*]. Esto se puede corroborar al analizar los *dataFlows* del puerto *s1-eth1*, disponibles en el anexo 8.

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2001 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=2001 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=2000 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=2000 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=2000 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4024ms
rtt min/avg/max/mdev = 2000.313/2000.876/2001.560/1.359 ms, pipe 2
```

Figura 41: **L2E1**: *Ping* con *delay* entre *h1* y *h2*.
Fuente: Elaboración propia.

A partir del punto 8, se implementaron políticas de ruta para que el costo de utilizar *s1-s2* sea mayor al de *s1-s3-s2* y así evitar este *delay*. Con esta nueva configuración, VTN

seleccionará *s1-s3-s2* para enrutar tráfico entre *h1* y *h2*. Los *dataflows* del puerto *s1-eth1*, disponibles en el anexo 9, lo corroboran. Al hacer *ping* entre *h1* y *h2*, se obtendrá el resultado de la figura 42, en donde se aprecia la ausencia de un *delay* significativo.

```
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.175 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.038 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.050 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.038 ms

--- 10.0.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.038/0.069/0.175/0.053 ms
```

Figura 42: **L2E1**: *Ping* sin *delay* entre *h1* y *h2*.
Fuente: Elaboración propia.

### 5.2.2. EXPERIENCIA 2: IMPLEMENTANDO UNA VLAN CON VTN

En esta experiencia se utilizará el *script* 5 para implementar la red descrita en la figura 43. El objetivo será crear dos VLAN (100 y 200) y asignar *h2*, *h4* y *h6* a la VLAN 100 y *h1*, *h3* y *h5* a la VLAN 200. El *script* se encargará de crear interfaces VLAN en los *hosts*, con el objetivo de que asignen un *tag 802.1q* a los *frames* que serán enviados a través de éstas, por lo que cada paquete tendrá asignada la VLAN-ID de la VLAN a la que pertenece. Por ejemplo, todos los paquetes generados por *h1* tendrán una VLAN-ID 200.

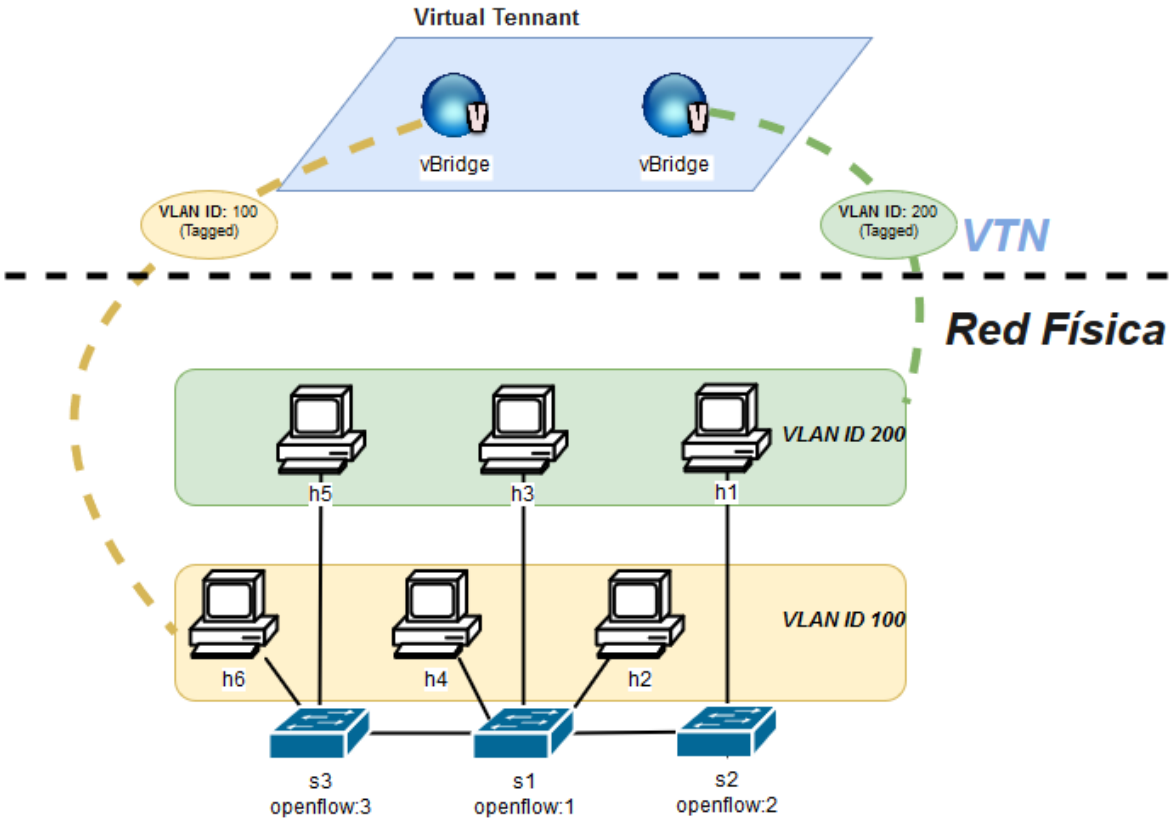



Figura 43: **L2E2**: (Topología) Implementando una VLAN con VTN.
Fuente: Elaboración propia.

#### CREANDO LA TOPOLOGÍA

1. Crear la topología en *Mininet*:

```
sudo mn --controller=remote,ip=192.168.10.120 --custom topo_4.py --
    topo topo_vlan  --mac
```

Ejecutar *net* para obtener la configuración de la topología:

```
h1 h1-eth0.200:s2-eth1
h2 h2-eth0.100:s1-eth1
h3 h3-eth0.200:s1-eth2
h4 h4-eth0.100:s1-eth3
h5 h5-eth0.200:s3-eth1
h6 h6-eth0.100:s3-eth2
s1 lo:  s1-eth1:h2-eth0.100 s1-eth2:h3-eth0.200 s1-eth3:h4-eth0.100
    s1-eth4:s2-eth2
s2 lo:  s2-eth1:h1-eth0.200 s2-eth2:s1-eth4 s2-eth3:s3-eth3
s3 lo:  s3-eth1:h5-eth0.200 s3-eth2:h6-eth0.100 s3-eth3:s2-eth3
c0
```

2. Inicializar *VTN Coordinator* en *CentOS 7*:

```
sudo /usr/local/vtn/bin/vtn_start
```

### CREANDO LA VTN CON VTN COORDINATOR

3. Crear un controlador llamado *controladorUno*. En *ipaddr* se pondrá la IP del controlador ODL:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"controller": {"controller_id": "controladorUno", "
    ipaddr":"192.168.10.120","username":"admin","password":"admin", "
    type": "odc", "version": "1.0", "auditstatus":"enable"}}' http
    ://127.0.0.1:8083/vtn-webapi/controllers.json
```

4. Crear una VTN llamada *vtn2*:

```
curl -X POST -H 'content-type: application/json' -H 'username: admin
    ' -H 'password: adminpass' -d '{"vtn" : {"vtn_name":"vtn2","
    description":"VTN EXP 2" }}' http://127.0.0.1:8083/vtn-webapi/
    vtns.json
```

### CREANDO VBRIDGES

5. Crear dos *vBridges* (*vBridge1* y *vBridge2*) y asignarlos a *vtn2*:

```
curl -X POST -H 'content-type: application/json' -H 'username: admin
    ' -H 'password: adminpass' -d '{"vbridge" : {"vbr_name":"vBridge1
    ", "controller_id":"controladorUno","domain_id":"(DEFAULT)" }}'
    http://127.0.0.1:8083/vtn-webapi/vtns/vtn2/vbridges.json
```

```
curl -X POST -H 'content-type: application/json' -H 'username: admin
    ' -H 'password: adminpass' -d '{"vbridge" : {"vbr_name":"vBridge2
    ","controller_id":"controladorUno","domain_id":"(DEFAULT)" }}'
    http://127.0.0.1:8083/vtn-webapi/vtns/vtn2/vbridges.json
```

### CONFIGURANDO LOS MAPEOS POR VLAN

6. Configurar dos mapeos de VLAN (*vlan map*), uno con VLAN-ID 100 para *vBridge1* y otro con VLAN-ID 200 para *vBridge2*:

```
curl –X POST –H 'content–type: application/json' –H 'username: admin
    ' –H 'password: adminpass' –d '{"vlanmap" : {"vlan_id": 100 }}'
    http://127.0.0.1:8083/vtn–webapi/vtns/vtn2/vbridges/vBridge1/
    vlanmaps.json
```

```
curl –X POST –H 'content–type: application/json' –H 'username: admin
    ' –H 'password: adminpass' –d '{"vlanmap" : {"vlan_id": 200 }}'
    http://127.0.0.1:8083/vtn–webapi/vtns/vtn2/vbridges/vBridge2/
    vlanmaps.json
```

### LIMPIEZA

7. Eliminar la VTN creada (*vtn2*):

```
curl ––user "admin":"admin" –H "Content–type: application/json" –X
    POST http://192.168.10.120:8181/restconf/operations/vtn:remove–
    vtn –d '{"input":{"tenant–name":"vtn2"}}'
```

```
sudo /usr/local/vtn/bin/vtn_stop
```

## ANÁLISIS DE RESULTADOS

En el anexo 10 se muestra la configuración final de *vtn2*. Para verificar las VLANs creadas, bastaría con utilizar el comando *pingall*, que debería entregar el resultado de la figura 44. En este caso, se comprueba que solo los *hosts* que pertenecen a la misma VLAN tienen interconectividad.

```
*** Ping: testing ping reachability
h1 -> X h3 X h5 X
h2 -> X X h4 X h6
h3 -> h1 X X h5 X
h4 -> X h2 X X h6
h5 -> h1 X h3 X X
h6 -> X h2 X h4 X
*** Results: 60% dropped (12/30 received)
```

Figura 44: **L2E2**: *Pingall* entre todos los *hosts*.
Fuente: Elaboración propia.

### 5.2.3. EXPERIENCIA 3: ENCADENAMIENTO DE SERVICIOS DE RED CON VTN

En esta última experiencia se utilizará el *script* 6 para implementar la red descrita en la figura 45. El objetivo será probar el encadenamiento de servicios de red con VTN. Se simularán dos servicios; *srv_fw*, que representará un *firewall* (***delay***: 100[*ms*] ± 10[*ms*]) y *srv_ids*, que representará un IDS (***delay***: 200[ms] ± 20[*ms*]). En la topología existirán los *hosts h2*, *h3* y *h4*, y un *NAT Gateway* que permitirá a *Mininet* conectarse a internet. En base a estos recursos, se implementarán los siguientes casos:

- El tráfico originado desde internet con destino a *h2* deberá pasar por *srv_fw* y *srv_ids*.
- El tráfico originado desde internet con destino a *h3* sólo deberá pasar por *srv_fw*.
- El tráfico originado desde internet con destino a *h4* no pasará por ningún servicio de red.

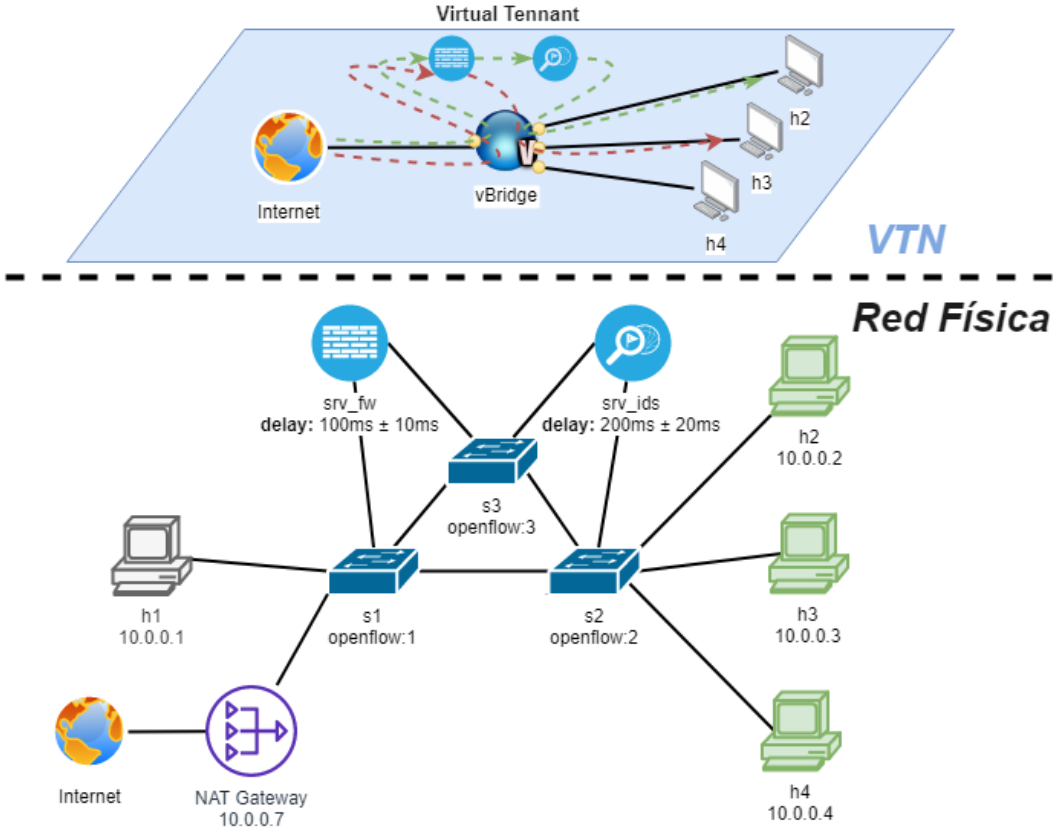



Figura 45: **L2E3**: (Topología) Encadenamiento de servicios de red con VTN. Fuente: Elaboración propia.

## CREANDO LA TOPOLOGÍA

1. Asegurarse de tener instalado el paquete *bridge-utils* en la máquina en donde se aloja *Mininet*:

```
sudo apt-get install bridge-utils
```

2. Crear la topología en *Mininet*. El parámetro ***–nat*** permite a *Mininet* conectarse a la LAN de la VM mediante un *NAT Gateway*:

```
sudo mn --controller=remote,ip=192.168.10.120 --custom topo_6.py --
    topo topo_sfc  --mac --nat
```

Ejecutar *–net* para obtener la configuración de la topología:

```
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s2-eth2
h4 h4-eth0:s2-eth3
srv_fw srv_fw-eth0:s1-eth4 srv_fw-eth1:s3-eth3
srv_ids srv_ids-eth0:s3-eth4 srv_ids-eth1:s2-eth6
nat0 nat0-eth0:s1-eth5
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth4 s1-eth3:s3-eth1 s1-eth4:
    srv_fw-eth0 s1-eth5:nat0-eth0
s2 lo:  s2-eth1:h2-eth0 s2-eth2:h3-eth0 s2-eth3:h4-eth0 s2-eth4:s1-
    eth2 s2-eth5:s3-eth2 s2-eth6:srv_ids-eth1
s3 lo:  s3-eth1:s1-eth3 s3-eth2:s2-eth5 s3-eth3:srv_fw-eth1 s3-eth4:
    srv_ids-eth0
c0
```

3. Inicializar *VTN Coordinator* en *CentOS 7*:

```
sudo /usr/local/vtn/bin/vtn_start
```

## CREANDO LA VTN CON VTN COORDINATOR

4. Crear un controlador llamado *controladorUno*. En *ipaddr* se pondrá la IP del controlador ODL:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"controller": {"controller_id": "controladorUno", "
    ipaddr":"192.168.10.120","username":"admin","password":"admin", "
    type": "odc", "version": "1.0", "auditstatus":"enable"}}' http
    ://127.0.0.1:8083/vtn-webapi/controllers.json
```

5. Crear una VTN llamada *vtn3*:

```
curl -X POST -H 'content-type: application/json' -H 'username: admin
    ' -H 'password: adminpass' -d '{"vtn" : {"vtn_name":"vtn3","
    description":"VTN EXP 3" }}' http://127.0.0.1:8083/vtn-webapi/
    vtns.json
```

### CREANDO VBRIDGE Y VINTERFACES

6. Crear un *vBridge* (*vBridge1*) y asignarlo a *vtn3*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vbridge" : {"vbr_name":"vBridge1","controller_id":"
    controladorUno","domain_id":"(DEFAULT)" }}' http
    ://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges.json
```

7. Crear una interfaz llamada *vif1* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif1","description": "NAT"}}'
     http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/vBridge1/
    interfaces.json
```

La interfaz virtual *vif1* será mapeada al puerto *s1-eth5* (*NAT Gateway*):

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:1-s1-eth5
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/vBridge1
    /interfaces/vif1/portmap.json
```

8. Crear una interfaz llamada *vif2* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif2","description": "h2
    gateway"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/
    vBridge1/interfaces.json
```

La interfaz virtual *vif2* será mapeada al puerto *s2-eth1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth1
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/vBridge1
    /interfaces/vif2/portmap.json
```

9. Crear una interfaz llamada *vif3* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif3","description": "h3
    gateway"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/
    vBridge1/interfaces.json
```

La interfaz virtual *vif3* será mapeada al puerto *s2-eth2*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth2
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/vBridge1
    /interfaces/vif3/portmap.json
```

10. Crear una interfaz llamada *vif4* en *vBridge1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "vif4","description": "h4
    gateway"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/
    vBridge1/interfaces.json
```

La interfaz virtual *vif4* será mapeada al puerto *s2-eth3*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth3
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vbridges/vBridge1
    /interfaces/vif4/portmap.json
```

## CREANDO VTERMINALS

11. Definir los puertos de los *switches* que serán mapeados a los servicios como *static-edge*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    PUT http://192.168.10.120:8181/restconf/config/vtn-static-
    topology:vtn-static-topology/static-edge-ports -d '{"static-edge-
    ports": {"static-edge-port": [ {"port": "openflow:1:4"}, {"port":
     "openflow:3:3"}, {"port": "openflow:3:4"}, {"port": "openflow
    :2:6"}]}}'
```

12. Crear un *vTerminal* llamada *srv_fw_1* que simulará la interfaz de entrada del servicio de *firewall*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vterminal": {"vterminal_name": "srv_fw_1","
    controller_id": "controladorUno","domain_id": "(DEFAULT)","
    description": "vterminal firewall"}}' http://127.0.0.1:8083/vtn-
    webapi/vtns/vtn3/vterminals.json
```

Crear una interfaz llamada *sif11* en *srv_fw_1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "sif11","description": "
    vterminal sif11"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/
    vterminals/srv_fw_1/interfaces.json
```

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-
    vinterface:update-vinterface -d '{"input":{"update-mode":"UPDATE
    ","operation":"ADD","tenant-name":"vtn3","terminal-name":"
    srv_fw_1","interface-name":"sif11","enabled":"true"}}'
```

La interfaz virtual *sif11* será mapeada al puerto *s1-eth4*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:1-s1-eth4
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vterminals/
    srv_fw_1/interfaces/sif11/portmap.json
```

13. Crear un *vTerminal* llamada *srv_fw_2* que simulará la interfaz de salida del servicio de *firewall*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vterminal": {"vterminal_name": "srv_fw_2","
    controller_id": "controladorUno","domain_id": "(DEFAULT)","
    description": "vterminal firewall"}}' http://127.0.0.1:8083/vtn-
    webapi/vtns/vtn3/vterminals.json
```

Crear una interfaz llamada *sif12* en *srv_fw_2*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "sif12","description": "
    vterminal sif12"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/
    vterminals/srv_fw_2/interfaces.json
```

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-
    vinterface:update-vinterface -d '{"input":{"update-mode":"UPDATE
    ","operation":"ADD","tenant-name":"vtn3","terminal-name":"
    srv_fw_2","interface-name":"sif12","enabled":"true"}}'
```

La interfaz virtual *sif12* será mapeada al puerto *s3-eth3*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:3-s3-eth3
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vterminals/
    srv_fw_2/interfaces/sif12/portmap.json
```

14. Crear un *vTerminal* llamada *srv_ids_1* que simulará la interfaz de entrada del servicio de IDS.

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vterminal": {"vterminal_name": "srv_ids_1","
    controller_id": "controladorUno","domain_id": "(DEFAULT)","
    description": "vterminal ids"}}' http://127.0.0.1:8083/vtn-webapi
    /vtns/vtn3/vterminals.json
```

Crear una interfaz llamada *sif21* en *srv_ids_1*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "sif21","description": "
    vterminal sif21"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/
    vterminals/srv_ids_1/interfaces.json
```

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-
    vinterface:update-vinterface -d '{"input":{"update-mode":"UPDATE
    ","operation":"ADD","tenant-name":"vtn3","terminal-name":"
    srv_ids_1","interface-name":"sif21","enabled":"true"}}'
```

La interfaz virtual *sif21* será mapeada al puerto *s3-eth4*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:3-s3-eth4
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vterminals/
    srv_ids_1/interfaces/sif21/portmap.json
```

15. Crear un *vTerminal* llamada *srv_ids_2* que simulará la interfaz de salida del servicio de IDS.

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"vterminal": {"vterminal_name": "srv_ids_2","
    controller_id": "controladorUno","domain_id": "(DEFAULT)","
    description": "vterminal ids"}}' http://127.0.0.1:8083/vtn-webapi
    /vtns/vtn3/vterminals.json
```

Crear una interfaz llamada *sif22* en *srv_ids_2*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    POST -d '{"interface": {"if_name": "sif22","description": "
    vterminal sif22"}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/
    vterminals/srv_ids_2/interfaces.json
```

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-
    vinterface:update-vinterface -d '{"input":{"update-mode":"UPDATE
    ","operation":"ADD","tenant-name":"vtn3","terminal-name":"
    srv_ids_2","interface-name":"sif22","enabled":"true"}}'
```

La interfaz virtual *sif22* será mapeada al puerto *s2-eth6*:

```
curl --user admin:adminpass -H 'content-type: application/json' -X
    PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth6
    "}}' http://127.0.0.1:8083/vtn-webapi/vtns/vtn3/vterminals/
    srv_ids_2/interfaces/sif22/portmap.json
```

## CREANDO CONDICIONES Y FILTROS DE FLUJO

16. Crear una *flow condition* llamada *cond_1*. Esta condición hará *match* con todos los paquetes con destino a *h2*, originados en internet:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    condition:set-flow-condition -d '{"input":{"operation":"SET","
    present":"false","name":"cond_1","vtn-flow-match":[{"index":1,"
    vtn-ether-match":{},"vtn-inet-match":{"destination-network
    ":"10.0.0.2/32"}}]}}'
```

Crear una *flow condition* llamada *cond_2*. Esta condición hará *match* con todos los paquetes con destino a *h3*, originados en internet:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    condition:set-flow-condition -d '{"input":{"operation":"SET","
    present":"false","name":"cond_2","vtn-flow-match":[{"index":1,"
    vtn-ether-match":{},"vtn-inet-match":{"destination-network
    ":"10.0.0.3/32"}}]}}'
```

Crear una *flow condition* llamada *cond_todas*. Esta condición hará *match* con cualquier paquete:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    condition:set-flow-condition -d '{"input":{"operation":"SET","
    present":"false","name":"cond_todas","vtn-flow-match":[{"index
    ":1}]}}'
```

17. Crear un *flow filter* en la interfaz virtual *sif1* que redireccione el tráfico hacia *vif3*, en base a la condición de flujo *cond_2*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    filter:set-flow-filter -d '{"input":{"output":"false","tenant-
    name":"vtn3","bridge-name":"vBridge1","interface-name":"vif1","
    vtn-flow-filter":[{"condition":"cond_2","index":11,"vtn-redirect-
    filter":{"redirect-destination":{"terminal-name":"srv_fw_1","
    interface-name":"sif11"},"output":"true"}}]}}'
```

Crear un *flow filter* en la interfaz virtual *vif3* que redireccione el tráfico hacia *sif1*, en base a la condición de flujo *cond_todas*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    filter:set-flow-filter -d '{"input":{"output":"false","tenant-
    name":"vtn3","terminal-name":"srv_fw_2","interface-name":"sif12
    ","vtn-flow-filter":[{"condition":"cond_todas","index":11,"vtn-
    redirect-filter":{"redirect-destination":{"bridge-name":"vBridge1
    ","interface-name":"vif3"},"output":"true"}}]}}'
```

18. Crear un *flow filter* en la interfaz virtual *vif1* que redireccione el tráfico hacia *sif1*, en base a la condición de flujo *cond_1*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    filter:set-flow-filter -d '{"input":{"output":"false","tenant-
    name":"vtn3","bridge-name":"vBridge1","interface-name":"vif1","
    vtn-flow-filter":[{"condition":"cond_1","index":10,"vtn-redirect-
    filter":{"redirect-destination":{"terminal-name":"srv_fw_1","
    interface-name":"sif11"},"output":"true"}}]}}'
```

Crear un *flow filter* en la interfaz virtual *sif1* que redireccione el tráfico hacia *sif2*, en base a la condición de flujo *cond_1*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    filter:set-flow-filter -d '{"input":{"output":"false","tenant-
    name":"vtn3","terminal-name":"srv_fw_2","interface-name":"sif12
    ","vtn-flow-filter":[{"condition":"cond_1","index":10,"vtn-
    redirect-filter":{"redirect-destination":{"terminal-name":"
    srv_ids_1","interface-name":"sif21"},"output":"true"}}]}}'
```

Crear un *flow filter* en la interfaz virtual *sif2* que redireccione el tráfico hacia *vif2*, en base a la condición de flujo *cond_todas*:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow-
    filter:set-flow-filter -d '{"input":{"output":"false","tenant-
    name":"vtn3","terminal-name":"srv_ids_2","interface-name":"sif22
    ","vtn-flow-filter":[{"condition":"cond_todas","index":10,"vtn-
    redirect-filter":{"redirect-destination":{"bridge-name":"vBridge1
    ","interface-name":"vif2"},"output":"true"}}]}}'
```

**LIMPIEZA**

19. Eliminar la VTN creada (*vtn3*):

```
curl --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn:remove-
    vtn -d '{"input":{"tenant-name":"vtn3"}}'
```

```
sudo /usr/local/vtn/bin/vtn_stop
```

## ANÁLISIS DE RESULTADOS

En el anexo 11 se muestra la configuración final de la *vtn3*. En la figura 46 se muestran los resultados de las pruebas de conectividad echas con *ping*. En **(B)**, se aprecia que el *ping* entre *www.google.cl* y *h2* tiene un *delay* aproximado de 300[*ms*]. Esto se correlaciona con el *delay* sumado que introducen los servicios simulados de *firewall* e IDS, lo que demuestra que el tráfico transita por ambos servicios antes de llegar a *h2*.

En **(A)**, a su vez, se muestra que el *ping* entre *www.google.cl* y *h3* tiene un *delay* aproximado de 100[*ms*]. Similar al caso anterior, este *delay* simboliza que el tráfico que fluye desde *www.google.cl* hacia *h3* sólo transita por el servicio de *firewall*, sin tomar en cuenta el IDS. De esta manera, queda demostrada la efectividad del encadenamiento de servicios de red en base a *flow filters*.


Figura 46: **L2E3**: Pruebas de encadenamiento de servicios de red.
Fuente: Elaboración propia.

# CAPÍTULO 6
# CONCLUSIONES

*Software-Defined Networking* es una tecnología disruptiva que ha dado que hablar durante los últimos años y que brinda muchas ventajas por sobre las redes tradicionales. La reducción de gastos operacionales, la programabilidad de la red, la automatización y la seguridad mejorada son solo algunos ejemplos que hacen que SDN y el protocolo *OpenFlow* sean apoyados por grandes organizaciones, como *Linux Foundation*, *Intel*, *Cisco*, *Juniper*, IBM, entre otras.

A medida que pasa el tiempo, una organización que siga operando mediante el esquema de red tradicional, no solamente generará un gasto operacional innecesario, sino que además limitará su capacidad de proporcionar servicios de calidad a sus clientes, lo cual a la larga llevaría a una inevitable pérdida de competitividad. Sin embargo, muchas organizaciones todavía no dimensionan esta realidad, muchas veces por conformidad, miedo al cambio o, simplemente, por ignorancia. Pese a esto, es seguro que a medida que SDN evolucione y madure con el correr de los años, las organizaciones estarán más abiertas a invertir en esta tecnología. Cuando esta situación empiece a ocurrir, los profesionales con conocimiento en SDN se convertirán en un recurso valioso para las empresas.

Implementar un laboratorio de redes físico, indistintamente de la arquitectura, comúnmente resulta muy caro y no siempre se encuentra al alcance de las personas. Afortunadamente, existen programas, usualmente de código abierto, que simulan y/o emulan una red SDN dentro de máquinas virtuales. *Mininet* es el *software* de este tipo más reconocido y utilizado por los investigadores para experimentar con SDN, y se considera como la herramienta ideal para investigar, desarrollar, aprender, prototipar, probar, depurar o cualquier tarea que podría realizarse al tener una red experimental completa emulada en un PC. Lo mejor de todo, es que *Mininet* no necesita *hardware* especializado ni de gran poder de procesamiento para funcionar, por lo que, actualmente, SDN está al alcance de todos.

Para que *Mininet* alcance su verdadero potencial como plataforma de estudio, es necesario que interactúe con un controlador real. *OpenDaylight* es un controlador de código abierto que nació con el objetivo de acelerar la adopción de SDN, respaldado por grandes compañias de *networking*. Si bien es cierto que existen otras alternativas, también de código abierto, se eligió ODL debido a su amplia documentación, continuo soporte y que probablemente en el futuro se convertirá en una parte esencial en el área de las telecomunicaciones.

Estas herramientas fueron utilizadas como base para los experimentos implementados en la secciones 5.1 y 5.2, ambos con objetivos similares, pero con distinto enfoque. Mientras que los experimentos de la sección 5.1 trataban sobre la programación de flujos

directamente en *switches* emulados en *Mininet* sin la intervención de un controlador, los de la sección 5.2 dependieron completamente de la interacción con el controlador, abstrayéndose de la programación de los dispositivos de red. Este paralelismo permitió comprender de manera empírica que existen muchas formas de cómo programar un *switch OpenFlow* para lograr un mismo objetivo, y cómo el uso de un controlador ayuda a simplificar esta tarea, permitiéndole al usuario abstraerse del cómo hacerlo y solo preocuparse del qué se necesita.

## 6.1. CONCLUSIONES ESPECÍFICAS

### OBJETIVO GENERAL

**Generar un ambiente virtual de experimentación de *Software-Defined Networking*, para que los estudiantes puedan comprender teórica y experimentalmente SDN y llevarlo al aula, sin la necesidad de que el laboratorio posea *hardware* especializado. El entorno virtual debe ser implementado mediante máquinas virtuales, junto con *software* de código abierto o, en su defecto, gratuito, minimizando el uso de recursos físicos.**

Se logró generar un ambiente virtual de experimentación de *Software-Defined Networking* de manera satisfactoria. En la sección 4.1, se hizo un resumen de todas las herramientas necesarias construir uno y, al mismo tiempo, cumplir con los requisitos estipulados en el objetivo general. A grandes rasgos, el usuario necesita los siguientes componentes; un *host* (PC de escritorio, *notebook*, etc.), y el siguiente conjunto de herramientas: hipervisor (*VMware Worstation Pro*), *router* virtual (*VyOS*), emulador de red (*Mininet*) y un controlador (*OpenDaylight*). Cabe destacar, que casi todas estas herramientas son de código abierto, a excepción del hipervisor, que solo es de prueba. Existen otros hipervisores de código abierto que podrían cumplir con la misma función, sin embargo, con *VMware Worstation Pro* se obtuvieron los mejores resultados en conjunto con el sistema operativo del *host* (*Windows 10*). El último paso fue armar el ambiente virtual con los componentes mencionados, proceso que culmina en la visión global representada en la figura 12.

### OBJETIVOS ESPECIFICOS

1. **Generar una guía bien definida y estructurada, junto con un repositorio con el *software* utilizado, para que cualquier persona pueda replicar los experimentos.**

   El capítulo 4 está dedicado exclusivamente a cumplir con este objetivo. En él se explicó, paso a paso, cómo instalar, configurar y usar cada uno de los programas descritos en el capítulo 3. Teniendo claras todas las piezas del rompecabezas, el

resto del capítulo se destinó a describir cómo armarlo. Todos los códigos *python*, junto con el *software* utilizado, puede descargado desde el siguiente repositorio personal[37].

2. **Crear un entorno simulado de SDN utilizado *OpenDaylight* como controlador principal y *Mininet* como emulador de una infraestructura de red SDN.**

   En la sección 4.6 se muestra con un ejemplo como hacer que ODL y *Mininet* interactuen. Como ODL es una plataforma modular, es necesario instalar el módulo *L2 Switch* para que ODL proporcione la funcionalidad mínima de redireccionamiento de paquetes a través de la red. A partir de este punto, es posible instalar otros módulos que implementen otras funcionalidades, todo esto dependiendo de las necesidades del usuario. Cabe destacar, que muchas veces los módulos no son compatibles entre sí, por lo que hay que tener cuidado a la hora de instalarlos.

3. **Analizar el comportamiento de *Virtual Tenant Network*, que corresponde a un módulo de *OpenDaylight* que permite implementar redes *overlay* sobre redes SDN físicas, mediante virtualización de funciones de red.**

   *Virtual Tenant Network*, como se mencionó en la sección 3.2.3, es una aplicación de ODL que provee una red de múltiples inquilinos (*overlay*) sobre una red SDN (*underlay*), mediante la virtualización de funciones de red. En la sección 3.2.3.1 se mencionaron los elementos que se pueden virtualizar con esta aplicación, mientras que en la sección 3.2.3.3 se describieron las funciones que realizan algunos de estos componentes virtualizados. Cabe destacar, que la programación de los *switches* se realiza a través de la API REST expuesta por *VTN Manager*, por lo que solo se puede hacer de manera proactiva.

   En la sección 5.2 se implementaron tres experiencias que representan algunos casos de uso de VTN. La primera consistió en configurar una red *overlay* utilizando los elementos *vBridge* y *vInterface* y, posteriormente, se configuraron políticas de ruta para redirigir el tráfico de la red *underlay*, sin afectar la configuración de la red *overlay* virtualizada. La segunda consistió en definir un mapeo por VLAN, utilizando *vBridge* y *vInterface*, lo cual permite automatizar la creación de VLANs, dependiendo de los *tags 802.1q* de los paquetes entrantes. Finalmente, en la tercera y última experiencia, se simuló un encadenamiento de servicios de red sobre una red *overlay*, utilizando *vBridge*, *vInterface*, *vTerminals* y un par de servicios simulados.

   Para programar estos laboratorios se utilizó la API REST de la aplicación *VTN Coordinator*, la cual se comunica directamente con la API REST de *VTN Manager*. Pese a que su uso no es estrictamente necesario, ya que la programación se podría haber realizado directamente con la API REST de *VTN Manager*, su inclusión trajo algunos beneficios, como la simplificación de sintaxis en algunas

[37] `https://gitlab.labcomp.cl/fpizarro/mininet`

peticiones REST. Éstas se hicieron a través de *Postman*, ya que su interfaz gráfica permite organizar y almacenar las peticiones REST para facilitar el despliegue y repetición de pruebas.

## 6.2. TRABAJO A FUTURO

A lo largo de la memoria se logró establecer una metodología para levantar un laboratorio de SDN de manera satisfactoria, sin embargo, al ser un tema tan amplio, existen muchos conceptos que no se desarrollaron a plenitud. A continuación, se mencionan algunas aristas relacionadas directamente con la investigación:

- **Mejora continua**: Cada uno de los componentes se encuentra en un constante proceso de mejora continua, con actualizaciones relativamente frecuentes, por lo que una más importantes es mantener al día el laboratorio. Por otra parte, las experiencias propuestas solo fueron un ejemplo del potencial de SDN y se espera que sirvan como base para nuevas experiencias propuestas de mayor complejidad, como, por ejemplo, una experiencia en donde se coordinen múltiples instancias de *OpenDaylight* a través de *VTN Coordinator*.
- **Integrar hardware real con Mininet**: *Mininet* permite emular completamente una red SDN virtualizada, sin embargo, trabajar con *hardware* real también es un requisito para comprender de una manera más integral este ecosistema. Afortunadamente, *Mininet* permite integrar *hardware* real, como *hosts* y *switches*, en sus redes virtuales. Incluso, es posible implementar *switches OpenFlow* físicos utilizando una *Raspberry Pi* y *Open vSwitch* (PiOVS)[38], por lo que no es necesario invertir en *hardware* caro y privativo.
- **Integrar Mininet con otro controlador**: A lo largo de la memoria se trabajó solo con controlador *OpenDaylight*, cuando en realidad existen muchos otros, con distintas características y funcionalidades, privativos o de código abierto. Sería interesante investigar cómo es la integración de esas alternativas con *Mininet*, las ventajas y desventajas, etc.
- **Experimentar con otros módulos de ODL**: Además de los módulos presentados en la sección 3.2, existen muchos otros servicios de plataforma que expanden las funcionalidades del controlador. Por ejemplo, el módulo BGP[39] permite emparejar un dominio SDN con un *router* externo e intercambiar rutas mediante el protocolo BGP, con el fin de establecer una conexión con el mundo exterior. El *router* virtual *VyOS* es compatible con el protocolo BGP, por lo que también puede ser utilizado como *router* externo para realizar este tipo de experimentos.

[38] **PiOVS**: `https://www.telematika.org/post/piovs-raspberry-pi-open-vswitch/`

[39] **BGP**: `https://docs.opendaylight.org/en/stable-nitrogen/user-guide/bgp-user-guide.html`

- **Integrar ODL con OpenStack**: Como fue mencionado en el caso de uso del anexo 6.3.2, un controlador SDN puede ser integrado al componente *Neutron* de *OpenStack* a través del plugin ML2, con el fin de superar sus deficiencias y soportar múltiples tecnologías de virtualización de red. ODL, mediante el módulo VTN, es capaz de proveer un entorno de red basado completamente en *OpenFlow*. En la sección 5.2 se realizaron múltiples experimentos con VTN, por lo que un complemento ideal sería realizarlos en una plataforma *OpenStack*, para así verificar empíricamente el beneficio que proporciona utilizar el paradigma SDN en plataformas de IaaS. Una forma accesible de realizar estas pruebas es utilizar *DevStack*, que son, básicamente, una serie de *scripts* extensibles que se utilizan para levantar un entorno completo de *OpenStack* en una máquina virtual, basado en las últimas versiones disponibles en su *git*.

- **Desarrollar de componentes OSGi para ODL**: En los experimentos de la sección 5.2, los *switches* de *Mininet* fueron programados con la API REST expuesta por VTN. Este tipo de programación es simple y conveniente, pero tiene una limitación importante, solo puede instanciar flujos de manera proactiva. Si por alguna razón, se necesita establecer un comportamiento reactivo personalizado, es decir, que los flujos sean instanciados dependiendo de la reacción del controlador a los eventos *packet-in*, será necesario desarrollar un componente OSGi propio, que se integrará al ecosistema *OpenDaylight* mediante el MD-SAL.

## 6.3. PROBLEMAS ENCONTRADOS

A lo largo de la memoria se encontraron dos situaciones que obstaculizaron su desarrollo, ambas relacionadas con el controlador *OpenDaylight*:

- **Documentación inconsistente**: ODL es uno de los controladores más documentados en Internet, sin embargo, muchas veces la documentación a menudo estaba obsoleta o incompleta, incluso la oficial. Esto puede entenderse debido a la rápida evolución del *software*, incluso con nuevas versiones cada seis meses, pero de igual manera debe ser mejorada en el futuro si es que se espera que sea utilizado de manera masiva.

- **Incompatibilidad de módulos entre versiones de ODL**: Como se menciona en la sección 3.2, ODL es una combinación de múltiples proyectos integrados, cada uno con un propósito y personas encargadas. Cuando una nueva versión del controlador es liberada, cada uno de los módulos debe ser adaptado e integrado. Si el módulo no está contemplado, o no está listo para una nueva versión, este no vendrá en el paquete básico y tendrá que ser compilado de forma manual. La última versión a la fecha, *Neon*, no es compatible con los módulos VTN y *L2 Switch*, mientras que *DluxApps* no está contemplado. Luego de algunas pruebas, se

determinó que *Nitrogen* era la versión del controlador más adecuada para trabajar, debido a su alta compatibilidad y estabilidad.

# REFERENCIAS BIBLIOGRÁFICAS

[Astuto *et al.*, 2014] Astuto, B. N., Mendonça, M., Nguyen, X. N., Obraczka, K., y Turletti, T. (2014). A survey of software-defined networking: Past, present, and future of programmable networks.

[Bholebawa y Dalal, 2016] Bholebawa, I. Z. y Dalal, U. D. (2016). *Design and Performance Analysis of OpenFlow-Enabled Network Topologies Using Mininet*. Tesis doctoral, Sardar Vallabhbhai National Institute of Technology.

[Ching-Hao *et al.*, 2015] Ching-Hao, Chang, y Lin, Y.-D. (2015). Openflow version roadmap.

[GARCIA, 2015] GARCIA, B. R. (2015). *OpenDaylight SDN controller platform*. Tesis doctoral, Universitat Politècnica de Catalunya.

[Göransson y Black, 2015] Göransson, P. y Black, C. (2015). *Software Defined Networks A Comprehensive Approach*.

[Guide, 2019] Guide, D. (2019). ¿qué es iaas? https://www.ionos.es/digitalguide/servidores/know-how/que-es-iaas/. [Online] 2019-07-28.

[Horwitz, 2018] Horwitz, L. (2018). Benefits of software-defined networking. https://www.cisco.com/c/en/us/solutions/software-defined-networking/benefits.html. [Online] 2018-07-21.

[IDG Enterprise, 2017] IDG Enterprise (2017). Network world: State of the network survey.

[IP Knowledge, 2014] IP Knowledge (2014). Traditional vs software defined networking.

[Izard, 2018] Izard, R. (2018). How to work with fast-failover openflow groups. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/7995427/How+to+Work+with+Fast-Failover+OpenFlow+Groups. [Online] 2019-05-20.

[Jain *et al.*, 2013] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., S.Venkata, Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., y Vahdat, A. (2013). B4: Experience with a globally deployed software defined wan. *ACM SIGCOMM*.

[Kurose y Ross, 2017] Kurose, J. F. y Ross, K. W. (2017). *Computer Networking A Top-Down Approach*. Pearson, seventh edici n.

[Lantz *et al.*, 2017] Lantz, B., Handigol, N., Heller, B., y Jeyakumar, V. (2017). Introduction to mininet. https://github.com/mininet/mininet/wiki/Introduction-to-Mininet. [Online] 2018-06-15.

[Lantz y O'Connor, 2015] Lantz, B. y O'Connor, B. (2015). A mininet-based virtual testbed for distributed sdn development.

[Álvarez, 2015] Álvarez, R. (2015). Estudio de las redes definidas por software mediante el desarrollo de escenarios virtuales basados en el controlador opendaylight.

[Natarajan, 2015] Natarajan, S. (2015). Openflow version 1.3 tutorial. http://sdnhub.org/tutorials/openflow-1-3/. [Online] 2019-08-12.

[Open Networking Foundation, 2012] Open Networking Foundation (2012). *OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04)*, pp. 7–37.

[Project, 2018a] Project, O. (2018a). Openstack with virtual tenant network. https://docs.opendaylight.org/en/stable-fluorine/opendaylight-with-openstack/openstack-with-vtn.html. [Online] 2019-08-01.

[Project, 2018b] Project, O. (2018b). Virtual tenant network (vtn). https://docs.opendaylight.org/en/stable-fluorine/user-guide/virtual-tenant-network-(vtn).html. [Online] 2019-05-12.

[Projects, 2018a] Projects, L. (2018a). Opendaylight controller overview. https://docs.opendaylight.org/en/stable-oxygen/user-guide/opendaylight-controller-overview.html. [Online] 2018-06-30.

[Projects, 2018b] Projects, L. (2018b). Project members. https://www.opendaylight.org/support/members. [Online] 2019-06-25.

[Projects, 2019] Projects, L. (2019). Odl neon is here! https://www.opendaylight.org/what-we-do/current-release/neon. [Online] 2019-07-10.

[QUITRAL, 2015] QUITRAL, D. V. (2015). *IMPLEMENTACIÓN DE UN SOFTWARE-DEFINED DATACENTER USANDO OPENSTACK*. Tesis doctoral, UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA.

[Rao, 2015a] Rao, S. (2015a). Sdn series part six: Opendaylight, the most documented controller. https://thenewstack.io/sdn-series-part-vi-opendaylight/. [Online] 2019-07-28.

[Rao, 2015b] Rao, S. (2015b). Sdn's scale out effect on openstack neutron. https://thenewstack.io/sdn-controllers-and-openstack-part1/. [Online] 2019-07-28.

[Salman *et al.*, 2016] Salman, O., Elhajj, I. H., y Chehab, A. (2016). Sdn controllers: A comparative study.

[SDX Central, 2018] SDX Central (2018). What are sdn northbound apis (and sdn rest apis)? https://www.sdxcentral.com/sdn/definitions/north-bound-interfaces-api/. [Online] 2018-06-20.

[SDxCentral, 2019] SDxCentral (2012-2019). What is network service chaining or service function chaining. https://www.sdxcentral.com/networking/virtualization/definitions/what-is-network-service-chaining/.

[Sharma *et al.*, 2013] Sharma, S., Staessens, D., Colle, D., Pickavet, M., y Demeester, P. (2013). Automatic bootstrapping of openflow networks. pp. 1–6.

[Sisov, 2016] Sisov, M. (2016). Building a software-defined networking system with opendaylight controller.

[Toghraee, 2017] Toghraee, R. (2017). *Learning OpenDaylight*, pp. 28–47.

[Valencia *et al.*, 2015] Valencia, B., Santacruz, S., Becerra, L., y Padilla, J. (2015). Mininet: a versatile tool for emulation and prototyping of software defined networking. *Entre Ciencia e Ingeniería*, pp. 62–70.

[Wang, 2013] Wang, S.-Y. (2013). Estinet openflow network simulator and emulator. *TOPICS IN NETWORK TESTING*, p. 110–117.

[Yazıcı, 2013] Yazıcı, V. (2013). Discovery of the switches. https://vlkan.com/blog/post/2013/08/06/sdn-discovery/. [Online] 2019-08-02.

# ANEXOS

## 6.1. ANEXO 1: PROFUNDIZACIÓN DEL MARCO CONCEPTUAL

### 6.1.1. REDES PROGRAMABLES

La idea de redes programables se propuso para mitigar los problemas inherentes de las redes tradicionales, y facilitar su innovación y evolución. Los múltiples beneficios que brindan las redes programables van desde la centralización del control, la simplificación de algoritmos, ahorro al no necesitar dispositivos de red caros y propietarios, eliminación de *midleboxes* y permitir el diseño y despliegue de aplicaciones de terceros en la red. Pese a todo, la idea de implementar redes programables y desacoplar lógica de control lleva rondando desde hace tiempo. A continuación, se hará un resumen de las primeras aproximaciones a las redes programables, precursoras del paradigma SDN[Astuto *et al.*, 2014]:

1. **Open Signaling:** En el año 1995, el grupo OPENSIG trabajó en una serie *workshops* dedicados a *"hacer que ATM, internet y las redes móviles sean más abiertas, extensibles y programables"*. Ellos creían que la separación entre el *hardware* de comunicación y el *software* de control era necesaria, pero bastante difícil de realizar, principalmente por la integración vertical existente en los *routers* y *switches*, en donde su comportamiento como caja negra hacía que desplegar de rápidamente nuevos servicios de red y entornos fuera imposible. La base de sus propuestas radicaba en la idea de dar acceso a *hardware* de red mediante interfaces de red abiertas y programables; esto permitiría el despliegue de nuevos servicios a través de entornos de programación distribuida. Motivados por estas ideas, se creó un equipo de trabajo en la *Internet Engineering Task Force* (IETF), el cual creó el protocolo GSMP. En el año 2002 el trabajo concluyó con la publicación de GSMPv3.

2. **Active Networking:** A mediados de la década de los 90, se propuso la idea de una infraestructura de red programable, para servicios personalizados. Se consideraron dos aproximaciones principales:

   *a*) *Switches* programables por el usuario, con transferencia de datos *in-band* y canales de gestión *out-of-band*.

   *b*) Capsulas que contenían fragmentos de programas embebidos a los mensajes de los usuarios. Posteriormente, Estas capsulas serían interpretadas y ejecutadas por los *routers*.

*Active Networking* nunca tuvo la fama necesaria para que su uso fuera extendido en la industria, principalmente por problemas de seguridad y rendimiento.

3. **DCAN:** Otra iniciativa que tuvo lugar a mediados de la década de los 90 es el DCAN. El objetivo era diseñar y desarrollar la infraestructura necesaria para el control y la gestión de redes ATM. Básicamente, la premisa radica en que el control y las funciones de gestión de muchos dispositivos (*switches* ATM en el caso de DCAN) deben ser desacopladas de los dispositivos, y delegadas a entidades externas dedicadas a ese propósito, lo que es básicamente el concepto principal detrás de SDN. DCAN asume un protocolo minimalista entre el gestionador y la red, muy parecido a lo que se hace con *OpenFlow* en la actualidad.

4. **4D Project:** En el año 2004, 4D Project se abocó en idear un diseño que enfatizara la separación entre la lógica de decisiones de enrutamiento y los protocolos que gobernaban la interacción entre los elementos de la red. Propuso que el plano de *"decisión"* tuviera una mirada global de la red, apoyado por los planos de *"diseminación"* y *"descubrimiento"*, para el enrutamiento de tráfico. Estas ideas sirvieron como inspiración para trabajos posteriores como NOX, que propuso un *"sistema operativo para redes"*, en el contexto de una red habilitada para *OpenFlow*.

5. **NETCONF:** En el año 2006, la IETF propuso NETCONF como un protocolo de gestión para modificar la configuración de los dispositivos de red. El protocolo permite que los dispositivos de la red sean configurables a través de una API de la cual datos de configuración extensible pueden ser enviados y recuperados. Es importante destacar que NETCONF cumple con la meta de simplificar la configuración y actúa como pilar para la gestión, no existe una separación entre los planos de control y datos, por lo que esta no debe ser catalogada como totalmente programable.

6. **Ethane:** El predecesor inmediato de *OpenFlow* fue el proyecto *SANE / Ethane*. Surgió en el año 2006 y definió una nueva arquitectura para las redes empresariales. El foco de *Ethane* era utilizar un controlador centralizado para gestionar las políticas y la seguridad de la red. Un ejemplo sería proporcionar control de acceso basado en la identidad. Similar a SDN, *Ethane* emplea dos componentes; un controlador que decide si un paquete debe ser enviado, y un *switch Ethane* que consiste en una tabla de flujos y un canal seguro al controlador. Haciendo un paralelismo con SDN, el control de acceso basado en la identidad sería implementado como una aplicación por sobre un controlador SDN.

### 6.1.2. SWITCH OPENFLOW: CONCEPTOS ADICIONALES

#### 6.1.2.1. TABLA DE AGRUPADORES

Corresponde a una tabla de agrupadores *OpenFlow*. Un agrupador *OpenFlow* es una capa de abstracción que facilita las operaciones de paquetes más complejas y especializadas. Cada agrupador recibe paquetes como entrada y realiza las acciones *OpenFlow* correspondientes. Un agrupador no es capaz de realizar cualquier acción *OpenFlow*, como, por ejemplo, enviar paquetes a otra tabla de flujos. Además, se espera que cada paquete haya hecho *match* de manera apropiada antes de entrar al agrupador, ya que los agrupadores no soportan *matching* de paquetes. En resumen, los agrupadores son un mecanismo para realizar acciones avanzadas, que no pueden ser ejecutadas fácilmente por una entrada de flujo.

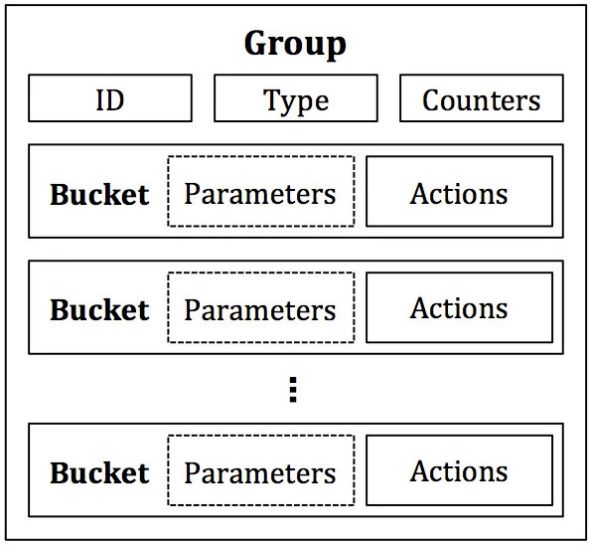



Figura 47: Componentes de un agrupador.
Fuente: [Izard, 2018].

En la figura 47 se aprecian los componentes de un agrupador, detallados a continuación:

- **ID**: Un entero de 32 bits sin signo que se desempeña como identificador único para el agrupador.
- **Type**: El tipo de agrupador. Los tipos definidos son *ALL*, *SELECT*, *INDIRECT* y *FAST-FAILOVER*.
- **Counters**: Se actualiza cada vez que un paquete es procesado por el agrupador.
- **Buckets**: Una lista ordenada de *action buckets*, en donde cada *bucket* contiene una lista de acciones *OpenFlow* a ejecutar. Los parámetros de un *bucket* solo están definidos para un cierto tipo de agrupadores.

#### 6.1.2.2. TIPOS DE AGRUPADORES

Un *switch OpenFlow* no requiere soportar todos los tipos de agrupadores:

- **(Requerido) ALL**: Los agrupadores de tipo *ALL* toman el paquete recibido como entrada y lo multiplican, para ser operado de manera independiente por cada *bucket* de su lista de *action buckets*. Distintas acciones pueden estar definidas en cada *bucket*, por lo que cada copia del paquete tendrá distinto comportamiento.
- **(Opcional) SELECT**: El tipo *SELECT* fue diseñado originalmente para balanceo de cargas. En los agrupadores del tipo *SELECT*, cada *bucket* de su lista de *action buckets* tiene asignado un peso, y cada paquete es enviado a un único *bucket*. La selección del bucket no está definida y depende de la implementación del *switch*, sin embargo, *weighted round robin* es talvez la opción más simple para distribuir paquetes. El peso del paquete se asigna como un parámetro especial a cada *bucket*.
- **(Requerido) INDIRECT**: El agrupador de tipo *INDIRECT* solo contiene un único *bucket* y todos los paquetes entrantes son enviados a este. El propósito de este tipo de agrupador es encapsular una lista de acciones comunes, utilizadas por muchos flujos. Este agrupador es comúnmente utilizado para simplificar una implementación *OpenFlow*.
- **(Opcional) FAST-FAILOVER**: El tipo *FAST-FAILOVER* fue diseñado específicamente para detectar y superar fallos de conectividad a nivel de capa de datos, sin la necesidad de que el controlador se vea involucrado. Al igual que para el tipo *SELECT*, los agrupadores del tipo *FAST-FAILOVER* poseen una lista de *action buckets* y solo un *bucket* puede ser utilizado. La diferencia radica en que cada *bucket* posee un parámetro especial llamado *port/group*, encargado de almacenar el estado de ese *port/group*. El agrupador seleccionará el primer *bucket* que encuentre con estado *UP* y este no cambiará a menos que su estado pase a *DOWN*. Si ocurre este evento, el agrupador rápidamente seleccionará el siguiente *bucket* de la lista de *action buckets* que esté en estado *UP*. Si ningún *bucket* está en estado *UP*, el paquete será descartado.

#### 6.1.2.3. TABLA DE MÉTRICAS

Una tabla de métricas consiste en entradas que definen métricas por flujo. Las métricas por flujo permiten a *OpenFlow* implementar operaciones de *QoS* simples, como un limitador de *rate*, y pueden ser combinadas con *queues* por puerto para implementar *frameworks* QoS complejos, como *DiffServ*.

Una métrica mide el *rate* de paquetes asignados a ella, permitiendo controlar el *rate* de esos paquetes. Las métricas se adjuntan directamente a las entradas de flujo. Cada

entrada de flujo puede especificar una métrica en su *set* de instrucciones; la métrica mide y controla el *rate* de paquetes que hacen *match* con todas las entradas de flujos a las que fue adjuntada. Múltiples métricas pueden usarse en una misma tabla de manera exclusiva, es decir, en conjuntos de entradas de flujo excluyentes, pero pueden ser usadas sobre un mismo conjunto de entradas si es que se utilizan en tablas de flujos subsecuentes.

Tabla 7: Componentes principales de una tabla de métricas en *OpenFlow* v1.3.0. Fuente: [Open Networking Foundation, 2012].

| Meter Identifier | Meter Bands | Counters |
|---|---|---|

En el cuadro 7 se observan los componentes principales de una tabla de métricas:

- **Meter Identifier**: Un identificador único para cada métrica.
- **Meter Bands**: Una lista no ordenada de bandas de métricas, en donde cada banda especifíca el *rate* y la manera en que se procesarán los paquetes.
- **Counters**: Se actualiza cada vez que un paquete es procesado por una métrica.

#### 6.1.2.4. INSTRUCCIONES

Cada entrada de flujo de una tabla de flujos posee un conjunto de instrucciones a aplicar a todos los paquetes que hagan *match*. Estas instrucciones pueden ser usadas para modificar el estado de un paquete, redireccionar un paquete a un puerto en particular o enviar el paquete a otra tabla (flujos, métrica o agrupadora), para un procesamiento más específico. Las instrucciones que implementa *OpenFlow* v1.3.0 se detallan a continuación, notar que algunas instrucciones son opcionales y que el orden de ejecución es correlativo con el orden de la lista:

- **(Opcional) Meter**: Direcciona el paquete a una tabla de métricas. Una tabla de métricas permite a *OpenFlow* implementar operaciones QoS simples y complejas.
- **(Opcional) Apply**: Aplica una lista de acciones inmediatamente al paquete, ignorando el *action set*.
- **(Optional) Clear**: Elimina todas las acciones del *action set*.
- **(Requerido) Write Action**: Agrega una lista de acciones al *action set* asociado al paquete.
- **(Opcional) Write Metadata**: Actualiza la meta data del paquete.
- **(Requerido) Goto**: Continua con el *processing pipeline* en la tabla subsecuente indicada.

#### 6.1.2.5. ACCIONES

El *action set* se compone de acciones detalladas a continuación. De la misma manera que las instrucciones, algunas acciones son opcionales:

- **(Requerido) Output**: La acción *output* redirecciona el paquete al puerto *OpenFlow* especificado.
- **(Opcional) Set-Queue**: Encola un paquete en una cola específica de un puerto de salida.
- **(Requerido) Drop**: No existe una representación explicita para descartar un paquete. Si el *action set* no especifica un *output*, el paquete se descartará.
- **(Requerido) Group**: Procesa el paquete a través de un agrupador específico de la tabla de agrupadores.
- **(Opcional) Push-Tag/Pop-Tag**: Algunos *switches* permiten hacer agregar o remover *tags* (como VLAN-ID) de los paquetes.
- **(Opcional) Set-Field**: Modifica los campos de la cabecera del paquete en base al *field type*.
- **(Opcional) Change-TTL**: Modifica el valor del TTL en IPv4 y en MPLS.

#### 6.1.2.6. MENSAJES OPENFLOW

*OpenFlow* soporta tres tipos de mensajes denominados *controlador-switch*, asincrónicos y simétricos, cada uno con sus respectivos subtipos. Éstos se intercambian a través del canal seguro.

##### CONTROLADOR-SWITCH

Los mensajes *controlador-switch* son inicializados por el controlador y se utilizan para gestionar o inspeccionar un *switch*. Algunos de estos mensajes necesitan obtener una respuesta del *switch*.

- **Features**: Se utiliza para consultar la características del *switch*. El *switch* debe responder esta consulta con la información requerida.
- **Configuration**: Permite al controlador consultar o modificar parámetros de configuración en el *switch*.

- **Modify-State**: Tambien denominados *"flow mod"*, se utilizan para agregar, borrar o modificar entradas en las tablas de flujos y de agrupadores.
- **Read-States**: Sirve para obtener estadísticas del *switch*.
- **Packet-Out**: El controlador envía paquetes al *switch*. Los mensajes del tipo *Packet-Out* deben encapsular al paquete completo, o un identificador que haga referencia a un paquete almacenado en el *switch*.
- **Barrier**: Utilizados por el controlador para asegurarse que las dependencias de los mensajes del controlador se han cumplido, o para recibir notificaciones de operaciones completadas.
- **Role-Request**: Son usados por el controlador para establecer o consultar el rol del canal *OpenFlow*. Son especialmente útiles cuando un *switch* se conecta a múltiples controladores.
- **Asynchronous-Configuration**: Son usados por el controlador para aplicar un filtro adicional sobre los mensajes asincrónicos que desea recibir por el canal *OpenFlow*. Son especialmente útiles a la hora de conectarse con múltiples controladores, y normalmente, se realizan al momento de establecer el canal *OpenFlow*. También posibilita consulta por el filtro aplicado.

### ASINCRÓNICOS

Los mensajes asincrónicos son inicializados por el *switch* sin que el controlador los haya solicitado. Son utilizados para alertar al controlador sobre cambios de estado en el *switch*.

- **Packet-In**: Transfiere el control del paquete al controlador.
- **Flow-Removed**: Informa al controlador que un flujo ha sido borrado.
- **Port Status**: Informa al controlador cambios en la configuración de un puerto.
- **Error**: Informa al controlador de problemas utilizando mensajes de error.

### SIMÉTRICOS

Los mensajes simétricos son enviados en ambos sentidos, sin solicitación previa.

- **Hello**: Mensajes intercambiados entre el controlador y el *switch* al momento que inicia la conexión.

- **Echo**: Mensajes que pueden ser enviados por el *switch* o el controlador y deben retornar una respuesta. Son utilizados principalmente para verificar la conectividad entre el *switch* y el controlador. También pueden ser usados para medir la latencia o el ancho de banda de ésta.
- **Experimenter**: Una manera estándar para ofrecer funcionalidades adicionales dentro del espacio de mensajes del protocolo.

## 6.2. CONTROLADORES: CONCEPTOS ADICIONALES

### 6.2.1. CONTROL IN-BAND VS CONTROL OUT-OF-BAND

En SDN, es requerido el intercambio de mensajes *OpenFlow* entre los *switches* y el controlador. Estos pueden ser intercambiados en modo *in-band* o en modo *out-of-band*. En *in-band*, los mensajes de control son enviados a través del mismo canal usado para la red de transporte de datos entre *switches*. En *out-of-band*, en cambio, los mensajes de control son enviados por un canal distinto. Como se muestra en la figura 48, en el modo *in-band* **(A)**, los *switches* A, B, C y D comparten el mismo canal para el tráfico de control y datos. En cambio, en el modo *out-of-band* **(B)**, los *switches* A, B, C y D utilizan canales distintos.

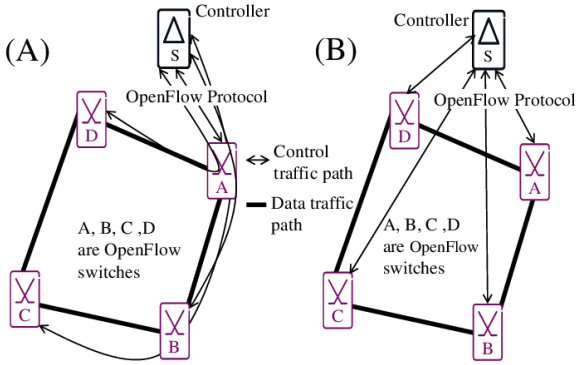



Figura 48: *In-Band vs Out-of-Band*.
Fuente: [Sharma *et al.*, 2013].

El modo *out-of-band* es más simple y fácil de diseñar ya que cada *switch* está conectado físicamente con el controlador, sin embargo, puede que no sea posible implementarlo en algunos escenarios. A su vez, un *switch* requerirá un puerto exclusivo, por lo que implementar una conexión *out-of-band* en el mundo real es más caro. En el modo *in-band*, los *switches* no necesitan tener un puerto físico extra para el tráfico de control. *OpenFlow* puede utilizar el puerto reservado *LOCAL* para implementar este tipo de conexiones, sin embargo, el protocolo no describe cómo establecer rutas para el tráfico

de control a través de la red *OpenFlow*, por lo que estás deberán ser implementadas mediante entradas de flujos.

### 6.2.2. DESCUBRIMIENTO DE TOPOLOGÍAS

El descubrimiento de topologías en SDN se realiza a través del descubrimiento de *switches*, enlaces y *hosts* [Yazıcı, 2013]:

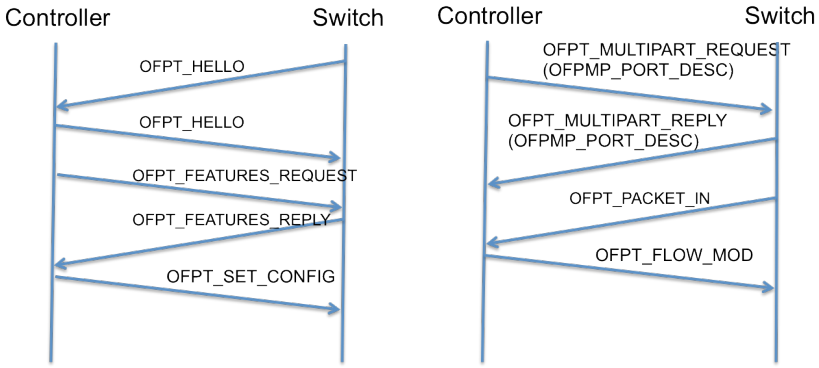



Figura 49: Cronología de mensajes *OpenFlow*.
Fuente: [Natarajan, 2015].

- **Switch Discovery**: En la figura 49 se aprecia el flujo de mensajes cronológico enviados entre el controlador y un *switch OpenFlow*. El *switch* establece una conexión estándar TCP (o TLS) con el controlador. Cuando una conexión es establecida, el *switch* debe enviar un mensaje simétrico del tipo *OFPT_HELLO* anunciando la versión más alta de *OpenFlow* que soporta el *switch*. Luego de establecer una conexión, el controlador envía un mensaje *controlador-switch* del tipo *OFPT_FEATURES_REQUEST*, que solicita una serie de características al *switch*. La otra entidad responde con un mensaje *OFPT_FEATURES_REPLY*, que, entre otras cosas, incluye el identificador *datapath-id* (DPID). Este campo es único, y se utiliza para identificar un *switch* dentro de una topología *OpenFlow*.

- **Link Discovery**: Luego de establecer una conexión entre un *switch* y el controlador, esté, periódicamente, le indica al *switch* que envíe paquetes del tipo *Link Layer Discovery Protocol* (LLDP) y *Broadcast Domain Discovery Protocol* (BDDP) a través de todos sus puertos. Un paquete de descubrimiento típicamente contiene el DPID, junto con el puerto del *switch* que originó el mensaje. El MAC de destinación reservado y el *ethertype* de los paquetes le permiten al controlador diferenciarlos. LLDP se utiliza para descubrir enlaces directos entre *switches*, mientras que BDDP se usa para descubrir enlaces indirectos entre *switches* que se encuentren en el mismo dominio *broadcast*. LLDP y BDDP se diferencian por el *ethertype*.

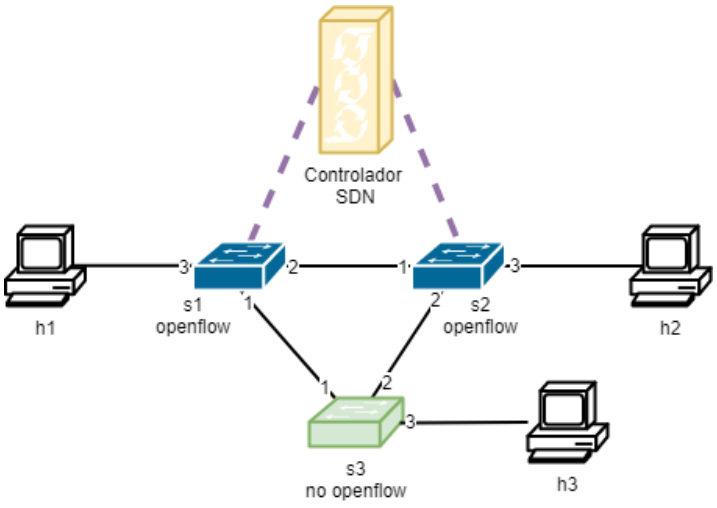



Figura 50: Descubriendo topologías en SDN.
Fuente: Elaboración propia.

Por ejemplo, en la figura 50 se presenta una topología SDN simple. Suponer que el controlador desea conocer los enlaces directos del *switch s1*. El controlador le ordena a *s1* que haga *flooding* de paquetes LLDP a través de todos sus puertos. El paquete LLDP enviado por el puerto *s1-1* es ignorado por *s3* (switch no *OpenFlow*), mientras que el paquete enviado por el puerto *s1-2*, en cambio, es recibido por *s2-1* y reenviado al controlador. De esta forma, el controlador identifica un enlace unidireccional entre *s1-2* y *s2-1*. Esta misma situación ocurre para *s2*.

Para identificar enlaces entre *switches* que se encuentren en el mismo dominio *broadcast*, el proceso es diferente. Por ejemplo, el controlador le indica a *s1* y *s2* que hagan *flooding* de paquetes BDDP. Estos paquetes llegan, eventualmente, al puerto *s3-1* y *s3-2* respectivamente. Luego, los paquetes BDDP, a diferencia de los LLDP, son redireccionados por *s3* mediante *broadcast*. Al recibir *s1* y *s2* los paquetes BDDP de *s3*, estos son reenviados al controlador. De esta manera, el controlador es capaz de identificar un enlace indirecto entre *s1-1* y *s2-2*.

- **Host Discovery**: Típicamente, los *host* son descubiertos de manera reactiva. Por ejemplo, suponer que *h1* quiere hacer ping con *h2*. El *host h1* envía un mensaje ICMP hacia *h2*, el cual llega a *s1* a través de *s1-3*. Como *s1* no tiene flujos que redireccionen a *h2*, el paquete es enviado al controlador. De esta manera, el controlador es capaz de descubrir un enlace entre *h1* y *s1-3*. En este punto, el controlador todavía desconoce la ubicación de *h2*, por lo que le indica a *s1* que haga *flooding* con el paquete. El paquete eventualmente llega a *s2* y, nuevamente, el paquete es reenviado al controlador ya que *s2* no sabe cómo redireccionar a *h2*. El controlador desconoce la ubicación de *h2*, y le indica a *s2* que haga *flooding* con el paquete. El paquete llega a *h2* y procede a responder el mensaje ICMP de *h1*. Para llegar a *h1*, la respuesta sigue un flujo similar. El paquete llega a

*s2* a través de *s2-3*. Como *s2* desconoce cómo llegar a *s1*, el paquete es enviado al controlador. En este punto, el controlador descubre el enlace entre *h2* y *s2-3*. Como el controlador sabe cómo llegar a *h1*, este instala las reglas de flujo necesarias y envía el paquete a *s1* por el puerto correspondiente. Lo mismo ocurre en *s1* y, finalmente, el paquete llega a *h1*.

## 6.3. CASO DE USO: INFRAESTRUCTURE AS A SERVICE

*Infrastructure as a Service* (IaaS) es un modelo de servicios que proporciona infraestructura computacional a organizaciones que la requieran. Las empresas ofrecen IaaS cuentan, por norma general, con centros de datos propios, donde se aloja el *hardware* necesario para ello, ocupándose de su administración y su mantenimiento. De este modo, los proveedores de IaaS pueden garantizar a sus clientes el acceso a recursos de computación (procesador, memoria RAM, disco duro) y estructuras de red integradas (incluyendo *firewalls*, *routers* y sistemas de seguridad y *backup*) en función de sus necesidades. La arquitectura SDN, específicamente, puede ser utilizada para simplificar todo lo relacionado con la configuración de la red y proporcionar *network as a service* (NaaS) dentro de una IaaS.

Una vez alquilados, los recursos de IaaS pueden escalarse verticalmente y, por normal general, los proveedores permiten a los clientes pagar solo por los componentes que utilizan. Esta elevada flexibilidad se debe a que las ofertas de IaaS no dependen de un *hardware* dedicado, lo que permite un reparto óptimo de recursos disponibles entre todos los clientes del proveedor. Para poder garantizar la fiabilidad y la seguridad de su servicio a largo plazo, el proveedor también se ocupa del mantenimiento y la modernización del *hardware* en sus centros de datos, así como de la instalación de los dispositivos y sistemas de seguridad necesarios, incluyendo también la sustitución de piezas defectuosas[Guide, 2019].

### 6.3.1. OPENSTACK

*OpenStack* es un proyecto de código abierto, diseñado para implementar soluciones de IaaS. El objetivo de *OpenStack* es permitir a las organizaciones construir y desplegar sus propias nubes (sean públicas, privadas o híbridas). La principal característica de este sistema es su estructuración: *OpenStack* está dividido en módulos que son independientes y que interactúan a través de sus APIs correspondientes, basándose en una arquitectura tipo *share nothing* (componentes asíncronos independientes, sin punto único de contención). Esta arquitectura permite la elasticidad y la escalabilidad del sistema; al no compartir nada entre los componentes es posible replicar ciertas partes para mejorar el rendimiento (permitiendo el escalamiento horizontal)[QUITRAL, 2015].

*OpenStack* es una integración de múltiples componentes y proyectos; los importantes

son los siguientes:

- **Nova compute**: Virtualización de servidores.
- **Swift object storage and Cinder block storage**: Virtualización de almacenamiento.
- **Neutron**: Virtualización de la red.

Existen otros más componentes y proyectos, sin embargo, los mencionados son los componentes clave de *OpenStack* para virtualización de servidores, almacenamiento y red.

*OpenStack* puede servir a múltiples inquilinos que, por ejemplo, pueden ser diferentes compañías. Los inquilinos pueden crear su propia infraestructura, incluyendo máquinas virtuales y NFVs, como *firewalls* y balanceadores de carga. Cada inquilino tendrá acceso a sus propias herramientas de monitoreo y gestión de su infraestructura y red virtuales, dándoles la oportunidad de crear sus propias redes que interconectarán a las VMs.

#### 6.3.1.1. NEUTRON

*Neutron* es un proyecto modular de *OpenStack* que proporciona la funcionalidad *network as a service* (NaaS) entre las interfaces de los dispositivos gestionados por otros servicios de *OpenStack*. Puede ser integrado con otros productos de *networking*, como *OpenDaylight*, *Cisco ACI*, *VMware NSX* y otros productos comerciales o de código libre. Antes de *Neutron*, *Openstack* utilizaba una infraestructura de red plana, sin soporte para L3 o *firewalls*, contenida dentro del servidor *Nova*, que hacía dificultoso acomodar los constantes cambios que ocurrían en la red.

*Neutron* se basa en *Open vSwitch*, un *switch* virtual que corre en el *host* del hipervisor, el cual proporciona servicios de red a las máquinas virtuales instaladas en el *host*. OVS es compatible con *OpenFlow*, por lo que puede ser integrado con un controlador SDN que soporte OVSDB como protocolo *southbound*. Esta integración se hace a través del plugin denominado *Modular Layer 2* (ML2), que se utiliza para controlar plataformas de redes externas o redes *underlay*.

#### 6.3.1.2. MODULAR LAYER 2 (ML2)

*Modular Layer 2 Core Plugin* es un *framework* que permite a *Neutron* a utilizar simultáneamente una variedad de tecnologías de red de L2 que se encuentran en los *data-centers* más complejos.

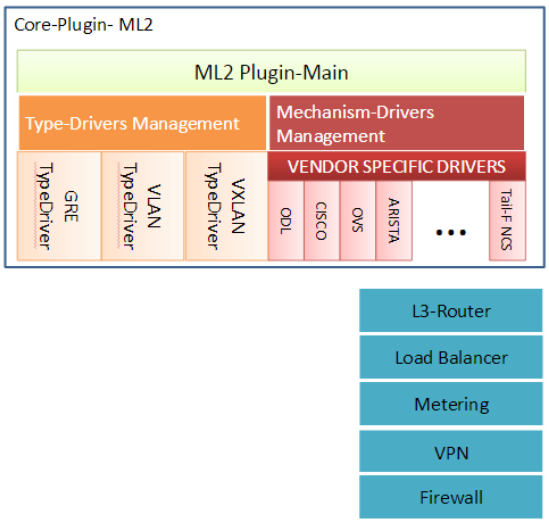



Figura 51: Plugin ML2.
Fuente: [Rao, 2015b].

En la figura 51 se aprecia la modularidad de la arquitectura de ML2. Ésta se logra separándola en dos categorías de drivers: tipo (*type*) y mecanismo (*mechanism*). Los drivers tipo definen un tipo (como flat, VLAN, GRE y VXLAN) de L2 en particular, en donde cada tipo de red disponible es gestionada por su *driver* tipo correspondiente. El *driver* mantiene la información específica del estado del tipo y conoce cómo se aíslan las redes de inquilinos. Por otra parte, los drivers mecanismo (específico de proveedores) se basan en el *driver* tipo para crear, actualizar y borrar redes, subredes y puertos.

### 6.3.2. INTEGRAR SDN CON OPENSTACK

SDN fue integrado a *Neutron* para superar sus deficiencias y soportar múltiples tecnologías de virtualización de red (un plano de control centralizado, creando redes aisladas virtuales de inquilinos). Con esta integración, *Neutron* soporta la naturaleza dinámica de entornos en la nube de gran escala, alta densidad y compatibles con múltiples inquilinos.

Técnicamente, el soporte de múltiples inquilinos en una red legado se realiza a través de políticas y listas de acceso. Herramientas disponibles en el *networking* tradicional, como VLANs (aislamiento L2), VRF (aislamiento L3) y ACL de L2/L3/L4 suelen ser utilizadas para lograr este aislamiento, sin embargo, en una red SDN, implementar este tipo de aislamiento y restricciones es mucho más sencillo ya que el controlador SDN decide como deben comunicarse lo dispositivos de red.

*Neutron*, a través de su arquitectura en base a *plugins*, permite la integración de controladores SDN en el entorno de *OpenStack*. Controladores como *OpenDaylight*, *Ryu* y

*Floodlight* utilizan el plugin ML2 con su correspondiente *driver* mecanismo para permitir la comunicación entre el controlador y *Neutron*.

En la arquitectura SDN, los cambios que se ejecutan en el plano de infraestructura son realizados por aplicaciones que corren al norte de los controladores SDN, mediante las APIs *nortbound*. Con la integración entre *Neutron* y los controladores SDN, los cambios hechos a la red y a los elementos de red también pueden ser realizados por los usuarios de *OpenStack*. Estas peticiones son traducidas a las *Neutron* APIs y gestionadas por los plugin de *Neutron* y su correspondiente agente que corre en el controlador SDN. Por ejemplo, *OpenDaylight* se comunica con *Neutron* a través de API REST *northbound*.

### 6.3.3. INTEGRAR ODL Y VTN CON OPENSTACK

Pese a que la integración práctica no se contempla en esta memoria, es importante mencionarla de manera conceptual. ODL puede ser utilizado para proveer *networking* a *Openstack*. Esto significa que es capaz de soportar múltiples inquilinos y aislación. Existen varias alternativas para lograr esta integración, pero una de las más interesantes se realiza mediante el módulo VTN, ya le permite a *OpenStack* administrar sus redes en un entorno *OpenFlow* puro. Esta integración se logra a través del plugin ML2 de *Neutron*, mediante el driver mecanismo ODL.

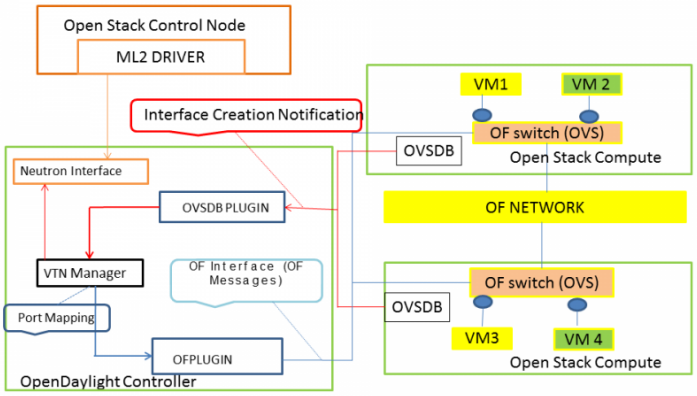



Figura 52: Integración de ODL y VTN con *OpenStack*.
Fuente: [Project, 2018a].

En la figura 52 se muestra la integración de VTN en un modelo de ejemplo, que describe la comunicación del controlador ODL y dos nodos *compute* conectados por un *switch OpenFlow*, la cual utiliza OVSDB como protocolo *southbound* y *Neutron* como API *northbound*. Una vez establecida esta conexión, cualquier inquilino creado en *OpenStack* será mapeado a VTN. Luego, VTN mapeará automáticamente la red creada a la

red física subyacente, inyectando la configuración requerida a cada *switch* de manera individual, utilizando el protocolo *southbound* soportado por el dispositivo, comúnmente *OpenFlow*.

VTN puede ser considerado como un conjunto de políticas que viven dentro de ODL. Éstas indican como los dispositivos de la capa de datos deben comunicarse entre sí, cuáles son las rutas de redirección de tráfico y cómo se deben encadenar las funciones de red. Pueden ser definidas directamente mediante la API REST de *VTN Manager*, o mediante *software* de orquestación como *VTN Coordinator*.

## 6.4. ANEXO 2: ARQUITECTURA DETALLADA DE OPENDAYLIGHT

En este anexo se describirán los componentes en detalle de ODL, representados en la figura 53, en un formato *botton-up*[Rao, 2015a].

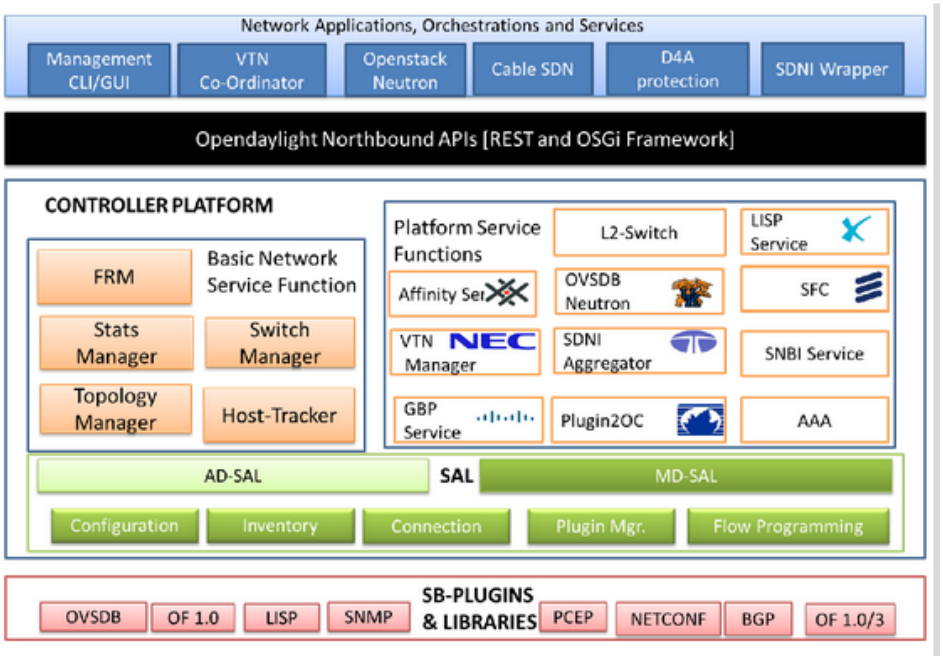



Figura 53: Arquitectura detallada de *OpenDaylight*.
Fuente: [Rao, 2015a].

1. **Southbound APIs**: Soporta múltiples protocolos (como plugin separados) como, por ejemplo, *OpenFlow* v1.0 y v1.3, BGP-LS, LISP, SNMP, etc. Estos módulos son dinámicamente enlazados al SAL, que determina cómo cumplir con los servicios requeridos por las aplicaciones, independiente del protocolo utilizado entre el controlador y los dispositivos de red.

2. **Service Abstraction Layer (SAL)**: El *Service Abstraction Layer* es la clave que permite la abstracción de servicios entre productores y consumidores. El SAL actúa como un gran registro de servicios anunciados por varios módulos y los liga a las aplicaciones que los requieran. Los módulos que proporcionan servicios (productores) pueden registrar sus APIs en el registro. Cuando una aplicación (consumidor) solicita un servicio mediante una API genérica, el SAL es responsable de ensamblar la solicitud mediante la vinculación entre el productor y consumidor en un contrato negociado y servido por el SAL. Existen dos formas de implementar el registro del SAL; en base a aplicaciones (AD-SAL) y en base a modelos (MD-SAL).

3. **Servicios Básicos de Red**: El controlador incluye varios servicios básicos de red como base. Entre ellos se incluyen servicios de *topology discovery and dissemination*, un *forwarding manager* para gestionar reglas básicas de redirección y un *switch manager* para identificar los elementos de red en la topología física subyacente. El SAL actúa como un registro activo que negocia contratos entre proveedores de servicios, como plugin de protocolos y de servicios básicos de red, y consumidores de servicios, como las aplicaciones. Estos contratos son respetados por el SAL, sin una dependencia directa con el plugin respectivo.

4. **Servicios de Plataforma**: El controlador contiene una colección de módulos que agregan funcionalidades SDN avanzadas al controlador, y que se pueden añadir o quitar de manera dinámica. Todos los servicios utilizan el SAL para registrar las funcionalidades implementadas en ellos, mediante la exposición de interfaces. El SAL es responsable de enlazar llamadas originadas desde las aplicaciones *northbound* solicitando las funcionalidades específicas implementadas en los módulos.

5. **Northbound APIs**: ODL ofrece modularidad de servicios y aplicaciones, ambos gestionados por el SAL. El controlador expone las *northbound* APIs que son utilizadas por las aplicaciones. ODL soporta el *framework* OSGI y APIs REST bidireccionales como *northbound* APIs. OSGI es utilizado por aplicaciones que corren en el mismo espacio de direcciones que el controlador, mientras que las APIs REST son utilizadas por aplicaciones que pueden correr en la misma máquina que el controlador o en un equipo diferente. Las aplicaciones *northbound* usan al controlador para obtener inteligencia de red, ejecutar algoritmos analíticos y luego usar el controlador para orquestar nuevas reglas, si es que existen, a través de la red.

6. **Aplicaciones**: La capa más alta de *OpenDaylight* consiste en las aplicaciones de lógica y negocio que controlar y monitorean el comportamiento de la red. La mayoría de las aplicaciones están mapeadas a su correspondiente *platform service*, como *VTN Coordinator* a *VTN Manager*, *SDNI whapper* a *SDNI aggregator*, etc. Las aplicaciones de red incluyen lógica de orquestación que realiza ingeniería de tráfico de red de acuerdo con las necesidades del entorno como, por ejemplo, la nube.

### 6.4.1. ABSTRACCIÓN DE SERVICIOS (SAL)

La abstracción es una de las filosofías que ha modelado *OpenDaylight*. El SAL distingue entre varios plugin, basado en los servicios que proveen (productores) y que consumen (consumidores), junto con la abstracción necesaria para que interactúen. Existen dos formas distintas para implementar estos plugin; AD-SAL y MD-SAL.

#### 6.4.1.1. SAL BASADO EN APLICACIONES (AD-SAL)

*OpenDaylight* debe proporcionar un entorno para desarrollar aplicaciones de red, abstrayéndose las especificaciones de los dispositivos de red. AD-SAL fue diseñado para resolver este problema, proporcionando una abstracción a través del uso de un conjunto de APIs genéricas, que permiten acceder a todas las funciones de los dispositivos de red. Los dispositivos se comunican con el controlador ODL a través de sus respectivos módulos de protocolo. Estos plugin, en cambio, se comunican con las APIs expuestas por el SAL. El SAL convierte el idioma hablado por estos plugin en APIs específicas para aplicaciones, mientras mantiene las funcionalidades requeridas por la lógica de negocios de las aplicaciones.

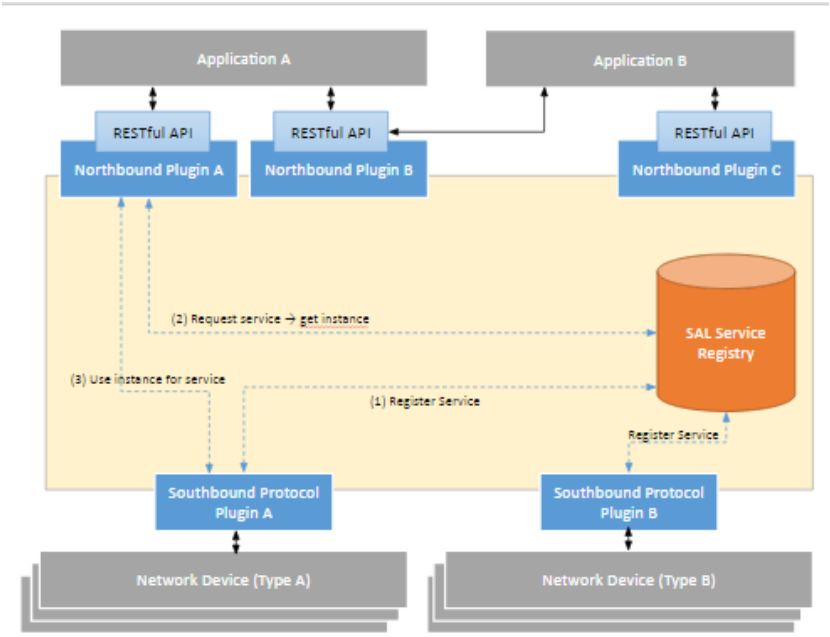



Figura 54: Arquitectura AD-SAL para un plugin.
Fuente: [Rao, 2015a].

En la figura 54 se aprecia la arquitectura de un plugin para AD-SAL. Por ejemplo, considerar un plugin *southbound* para SNMP. Si se desarrolla mediante la metodología AD-SAL, implementaría varias funciones básicas esperadas por el controlador, incluyendo la extracción del estado de los puertos, *switch health* y la topología OSPF. Estas funciones básicas son expuestas a los desarrolladores de plugin *northbound*. Un plugin SNMP puede exportar algunas funciones específicas, como el monitoreo de ancho de banda, por lo que el desarrollador de plugin *northbound* tendrá que implementar directamente Java APIs abstractas para importar esta funcionalidad.

#### 6.4.1.2. SAL BASADO EN MODELOS (MD-SAL)

Si AD-SAL permite independencia en el desarrollo para manejar complejidades a nivel de dispositivos, MD-SAL va un paso más allá, otorgándole a los desarrolladores la posibilidad de trabajar en un silo, agnóstico a las especificaciones de las APIs de servicios expuestas por los módulos que proporcionan los servicios. Mientras que AD-SAL fue diseñado para que exista transparencia norte-sur, MD-SAL fue diseñado para unir los módulos de manera horizontal, permitiéndole al desarrollador utilizar interfaces genéricas para el descubrimiento y consumo de servicios.

La diferencia radica en cómo las APIs son utilizadas por los productores y consumidores. En el caso de MD-SAL, el productor, generalmente un plugin *southbound*, crea un modelo de la data o servicios que va a exponer. Estos modelos están en formato YANG. Posteriormente, un compilador YANG es usado para crear APIs uniformes para los consumidores, que luego se hacen parte del plugin. Estas APIs son generadas por herramientas, permitiendo un alto nivel de uniformidad entre ellas, en términos de definición y uso.

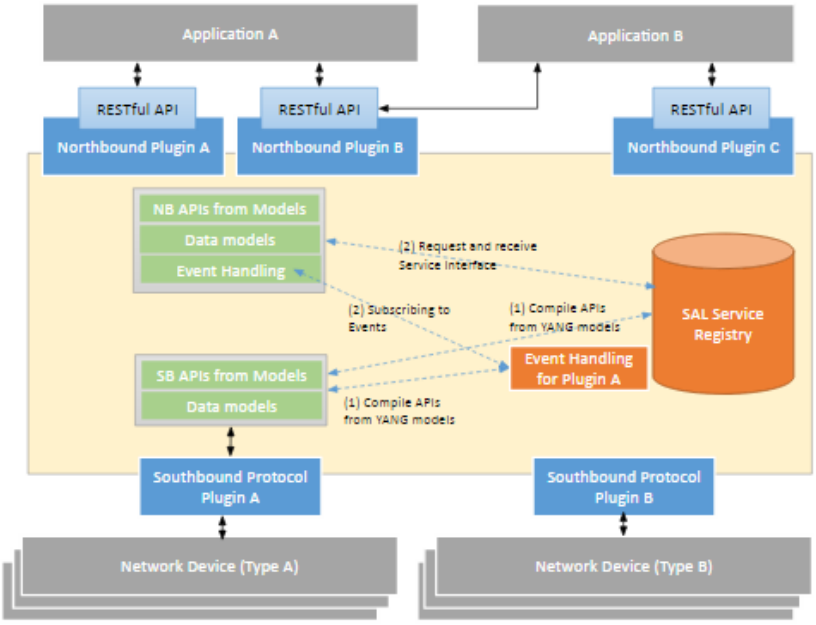



Figura 55: Arquitectura MD-SAL para un plugin.
Fuente: [Rao, 2015a].

En la figura 55 se aprecia la arquitectura de un plugin para MD-SAL. Por ejemplo, un plugin *southbound* para *OpenFlow* sería definido en un modelo YANG que contendría una descripción de servicios, como *packet_in*, *packet_out*, y entrega de paquetes. Posteriormente, al compilar los modelos, generará APIs para que el plugin *OpenFlow* interactúe con los *switches* y extraer data, APIs para acceder a la data extraída desde el

plugin *northbound*, incluyendo RPCs, interfaces RESTful, DOM APIs y APIs escuchas de notificaciones.

El SAL ayuda a los plugin generados para MD-SAL a registrar consumidores interesados en recibir eventos y notificaciones generadas por los dispositivos de red.

## 6.5. ANEXO 3: SCRIPTS PYTHON

Listing 2: **L1E1**: (*Script*) *Matching-plus-action.*

```
#!/usr/bin/python
from mininet.topo import Topo
from mininet.node import Host


#Creando una topologia Linear con 3 hosts
class Topo_Lab(Topo):
        def build(self):
                #Agregar Host
                h1=self.addHost('h1')
                h2=self.addHost('h2')
                h3=self.addHost('h3')
                #Agregar Switches
                s1=self.addSwitch('s1')
                s2=self.addSwitch('s2')
                #Agregar Links
                self.addLink(h1, s1)
                self.addLink(h2, s2)
                self.addLink(h3, s2)
                self.addLink(s1, s2)

topos={'topo_mpa':lambda:Topo_Lab()}
```

Listing 3: **L1E2**: (*Script*) Matching con múltiples tablas.

```
#!/usr/bin/python
from mininet.topo import Topo
from mininet.node import Host

class IPConfigHost( Host ):
        def config( self, ip,gw,**params ):
                r = super( Host, self ).config( **params )
                intf = self.defaultIntf()
                # Quitar la IP de la interfaz por defecto
                self.cmd( 'ifconfig %s inet %s' %(intf,ip) )
                # Agregar gateway dummy
                self.cmd( 'route add default gw %s' %(gw) )
                # Agregar reglas ARP para llegar a un gateway dummy
                self.cmd( 'arp -s %s 00:00:00:00:11:11' %(gw) )
                return r

#Creando una topologia Linear con 3 hosts
class Topo_Lab(Topo):
        def build(self):
                #Agregar Host
                h1=self.addHost('h1', cls=IPConfigHost,ip='10.0.1.2/24',
                    gw='10.0.1.1')
                h2=self.addHost('h2', cls=IPConfigHost,ip='10.0.0.2/24',
                    gw='10.0.0.1')
```

```
                h3=self.addHost('h3', cls=IPConfigHost,ip='10.0.0.3/24',
                    gw='10.0.0.1')

                #Agregar Switches
                s1=self.addSwitch('s1')
                s2=self.addSwitch('s2')
                #Agregar Links
                self.addLink(h1, s1)
                self.addLink(h2, s2)
                self.addLink(h3, s2)
                self.addLink(s1, s2)

topos={'topo_multitable':lambda:Topo_Lab()}
```

Listing 4: **L2E1**: (*Script*) Mapeando puertos con VTN.

```
#!/usr/bin/python
from mininet.node import Host,CPULimitedHost
from mininet.topo import Topo
from mininet.link import TCLink

#Creando una topologia con 4 hosts y 3 switches
class Topo_Lab(Topo):
        def build(self):
                #Agregar Host
                h1=self.addHost('h1')
                h2=self.addHost('h2')
                h3=self.addHost('h3')
                h4=self.addHost('h4')
                #Agregar Switches
                s1=self.addSwitch('s1')
                s2=self.addSwitch('s2')
                s3=self.addSwitch('s3')
                #Agregar Links
                self.addLink(h1, s1)
                self.addLink(h4, s1)
                self.addLink(h2, s2)
                self.addLink(h3, s2)
                #link s1-s2 tendra un delay de 1000ms
                self.addLink(s1, s2, delay='1000ms')
                self.addLink(s1, s3)
                self.addLink(s2, s3)

topos={'topo_route':lambda:Topo_Lab()}
```

Listing 5: **L2E2**: (*Script*) Implementando una VLAN con VTN.

```
#!/usr/bin/python
from mininet.node import Host, RemoteController
from mininet.topo import Topo

class VLANHost( Host ):
        def config( self, vlan=1, **params ):
```

```
                r = super( Host, self ).config( **params )
                intf = self.defaultIntf()
                # Quitar la IP de la interfaz por defecto
                self.cmd( 'ifconfig %s inet 0' %intf )
                # crear la interfaz Vlan
                self.cmd( 'vconfig add %s %d' %( intf, vlan ) )
                # asignar la IP del host a la interfaz Vlan
                self.cmd( 'ifconfig %s.%d inet %s' %( intf, vlan,params[
                    'ip'] ) )
                # Actualizar el nombre de inf y el inf map del host
                newName = '%s.%d' %( intf, vlan )
                # Actualizar la interfaz de Mininet para que
                # apunte al nombre de la interfaz Vlan
                intf.name = newName
                self.nameToIntf[ newName ] = intf
                return r
class Topo_Lab(Topo):

        def __init__( self ):
# Inicializar Topologia
                Topo.__init__( self )
# Agregar Host y switches
                h1=self.addHost( 'h1', cls=VLANHost, vlan=200)
                h2=self.addHost( 'h2', cls=VLANHost, vlan=100)
                h3=self.addHost( 'h3', cls=VLANHost, vlan=200)
                h4=self.addHost( 'h4', cls=VLANHost, vlan=100)
                h5=self.addHost( 'h5', cls=VLANHost, vlan=200)
                h6=self.addHost( 'h6', cls=VLANHost, vlan=100)
                s1 = self.addSwitch( 's1' )
                s2 = self.addSwitch( 's2' )
                s3 = self.addSwitch( 's3' )

                self.addLink(s1,h2)
                self.addLink(s1,h3)
                self.addLink(s1,h4)
                self.addLink(s2,h1)
                self.addLink(s3,h5)
                self.addLink(s3,h6)
                self.addLink(s1,s2)
                self.addLink(s2,s3)

topos = {'topo_vlan': ( lambda: MyTopo() ) }
```

Listing 6: **L2E3**: (*Script*) Encadenamiento de servicios de red con VTN.

```
#!/usr/bin/python
from mininet.topo import Topo
from mininet.node import Host

class ServiceHost( Host ):
        def config( self,ld,**params ):
                r = super( Host, self ).config( **params )
                self.cmd('ip addr flush dev %s-eth0' %(self.name))
```

```
                self.cmd('brctl addbr br0')
                self.cmd('brctl addif br0 %s-eth0' %(self.name))
                self.cmd('brctl addif br0 %s-eth1' %(self.name))
                self.cmd('ifconfig br0 up')
                self.cmd('tc qdisc add dev %s-eth1 root netem delay %s' %
                    (self.name,ld))
                return r

#Creando una topologia Linear con 4 hosts, 2 servicios y 3 switches
class Topo_Lab(Topo):
        def build(self):
                #Agregar Host
                h1=self.addHost('h1')
                h2=self.addHost('h2')
                h3=self.addHost('h3')
                h4=self.addHost('h4')
                fw=self.addHost('srv_fw',cls=ServiceHost,ld='100ms 10ms')
                ids=self.addHost('srv_ids',cls=ServiceHost,ld='200ms 20ms
                    ')
                #Agregar Switches
                                s1=self.addSwitch('s1')
                s2=self.addSwitch('s2')
                s3=self.addSwitch('s3')
                #Agregar Links
                self.addLink(h1, s1)
                self.addLink(h2, s2)
                self.addLink(h3, s2)
                self.addLink(h4, s2)
                self.addLink(s1, s2)
                self.addLink(s1, s3)
                self.addLink(s2, s3)
                self.addLink(fw, s1)
                self.addLink(fw, s3)
                self.addLink(ids, s3)
                self.addLink(ids, s2)

topos={'topo_sfc':lambda:Topo_Lab()}
```

## 6.6. ANEXO 4: CONSULTAS REST

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-
    type: application/json" -X GET http://192.168.10.120:8181/restconf/
    operational/vtn:vtns/
```

Listing 7: **L2E1**: Configuración de la VTN.

```
{
    "vtns": {
        "vtn": [
            {
                "name": "vtn1",
                "vbridge": [
                    {
                        "name": "vBridge1",
                        "vinterface": [
                            {
                                "name": "vif1",
                                "vinterface-status": {
                                    "state": "UP",
                                    "mapped-port": "openflow:1:1",
                                    "entity-state": "UP"
                                },
                                "port-map-config": {
                                    "node": "openflow:1",
                                    "port-name": "s1-eth1",
                                    "vlan-id": 0
                                },
                                "vinterface-config": {
                                    "description": "h1 gateway",
                                    "enabled": true
                                }
                            },
                            {
                                "name": "vif2",
                                "vinterface-status": {
                                    "state": "UP",
                                    "mapped-port": "openflow:2:1",
                                    "entity-state": "UP"
                                },
                                "port-map-config": {
                                    "node": "openflow:2",
                                    "port-name": "s2-eth1",
                                    "vlan-id": 0
                                },
                                "vinterface-config": {
                                    "description": "h2 gateway",
                                    "enabled": true
                                }
                            }
                        ],
                        "vbridge-config": {
```

```
                        "age-interval": 600
                    },
                    "bridge-status": {
                        "state": "UP",
                        "path-faults": 0
                    }
                }
            ],
            "vtenant-config": {
                "description": "VTN Escenario 1",
                "idle-timeout": 300,
                "hard-timeout": 0
            },
            "vtn-path-maps": {
                "vtn-path-map": [
                    {
                        "index": 1,
                        "condition": "cond_1",
                        "policy": 1,
                        "idle-timeout": 300,
                        "hard-timeout": 0
                    }
                ]
            }
        }
    ]
  }
}
```

```
curl -v --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow:get-data-
    flow -d '{"input":{"tenant-name":"vtn1","mode":"DETAIL","node":"
    openflow:1","data-flow-port":{"port-id":1,"port-name":"s1-eth1"}}}'
```

Listing 8: **L2E1**: Rutas sin políticas personalizadas.

```
{
    "output": {
        "data-flow-info": [
            {
                "hard-timeout": 0,
                "data-egress-node": {
                    "tenant-name": "vtn1",
                    "bridge-name": "vBridge1",
                    "interface-name": "vif1"
                },
                "physical-route": [
                    {
                        "physical-ingress-port": {
                            "port-id": "1",
                            "port-name": "s2-eth1"
                        },
                        "physical-egress-port": {
```

```
                "port-id": "3",
                "port-name": "s2-eth3"
            },
            "order": 0,
            "node": "openflow:2"
        },
        {
            "physical-ingress-port": {
                "port-id": "3",
                "port-name": "s1-eth3"
            },
            "physical-egress-port": {
                "port-id": "1",
                "port-name": "s1-eth1"
            },
            "order": 1,
            "node": "openflow:1"
        }
    ],
    "idle-timeout": 300,
    "data-flow-stats": {
        "packet-count": 1,
        "byte-count": 98,
        "duration": {
            "nanosecond": 880000000,
            "second": 2
        }
    },
    "data-ingress-port": {
        "port-id": "1",
        "node": "openflow:2",
        "port-name": "s2-eth1"
    },
    "virtual-route": [
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif2"
            },
            "reason": "PORTMAPPED",
            "order": 0
        },
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif1"
            },
            "reason": "FORWARDED",
            "order": 1
        }
    ],
```

```
        "data-egress-port": {
            "port-id": "1",
            "node": "openflow:1",
            "port-name": "s1-eth1"
        },
        "flow-id": 8,
        "data-ingress-node": {
            "tenant-name": "vtn1",
            "bridge-name": "vBridge1",
            "interface-name": "vif2"
        },
        "creation-time": 1558653486181,
        "data-flow-match": {
            "vtn-ether-match": {
                "destination-address": "00:00:00:00:00:01",
                "source-address": "00:00:00:00:00:02",
                "vlan-id": 0
            }
        }
    },
    {
        "hard-timeout": 0,
        "data-egress-node": {
            "tenant-name": "vtn1",
            "bridge-name": "vBridge1",
            "interface-name": "vif2"
        },
        "physical-route": [
            {
                "physical-ingress-port": {
                    "port-id": "1",
                    "port-name": "s1-eth1"
                },
                "physical-egress-port": {
                    "port-id": "3",
                    "port-name": "s1-eth3"
                },
                "order": 0,
                "node": "openflow:1"
            },
            {
                "physical-ingress-port": {
                    "port-id": "3",
                    "port-name": "s2-eth3"
                },
                "physical-egress-port": {
                    "port-id": "1",
                    "port-name": "s2-eth1"
                },
                "order": 1,
                "node": "openflow:2"
            }
        ],
```

```
"idle-timeout": 300,
"data-flow-stats": {
    "packet-count": 2,
    "byte-count": 196,
    "duration": {
        "nanosecond": 918000000,
        "second": 2
    }
},
"data-ingress-port": {
    "port-id": "1",
    "node": "openflow:1",
    "port-name": "s1-eth1"
},
"virtual-route": [
    {
        "virtual-node-path": {
            "tenant-name": "vtn1",
            "bridge-name": "vBridge1",
            "interface-name": "vif1"
        },
        "reason": "PORTMAPPED",
        "order": 0
    },
    {
        "virtual-node-path": {
            "tenant-name": "vtn1",
            "bridge-name": "vBridge1",
            "interface-name": "vif2"
        },
        "reason": "FORWARDED",
        "order": 1
    }
],
"data-egress-port": {
    "port-id": "1",
    "node": "openflow:2",
    "port-name": "s2-eth1"
},
"flow-id": 7,
"data-ingress-node": {
    "tenant-name": "vtn1",
    "bridge-name": "vBridge1",
    "interface-name": "vif1"
},
"creation-time": 1558653486166,
"data-flow-match": {
    "vtn-ether-match": {
        "destination-address": "00:00:00:00:00:02",
        "source-address": "00:00:00:00:00:01",
        "vlan-id": 0
    }
}
```

```
            }
        ]
    }
}
```

```
curl -v --user "admin":"admin" -H "Content-type: application/json" -X
    POST http://192.168.10.120:8181/restconf/operations/vtn-flow:get-data-
    flow -d '{"input":{"tenant-name":"vtn1","mode":"DETAIL","node":"
    openflow:1","data-flow-port":{"port-id":1,"port-name":"s1-eth1"}}}'
```

Listing 9: **L2E1**: Rutas con políticas personalizadas.

```
{
    "output": {
        "data-flow-info": [
            {
                "hard-timeout": 0,
                "data-egress-node": {
                    "tenant-name": "vtn1",
                    "bridge-name": "vBridge1",
                    "interface-name": "vif2"
                },
                "physical-route": [
                    {
                        "physical-ingress-port": {
                            "port-id": "1",
                            "port-name": "s1-eth1"
                        },
                        "physical-egress-port": {
                            "port-id": "4",
                            "port-name": "s1-eth4"
                        },
                        "order": 0,
                        "node": "openflow:1"
                    },
                    {
                        "physical-ingress-port": {
                            "port-id": "1",
                            "port-name": "s3-eth1"
                        },
                        "physical-egress-port": {
                            "port-id": "2",
                            "port-name": "s3-eth2"
                        },
                        "order": 1,
                        "node": "openflow:3"
                    },
                    {
                        "physical-ingress-port": {
                            "port-id": "4",
                            "port-name": "s2-eth4"
                        },
                        "physical-egress-port": {
```

```
                "port-id": "1",
                "port-name": "s2-eth1"
            },
            "order": 2,
            "node": "openflow:2"
        }
    ],
    "idle-timeout": 300,
    "data-flow-stats": {
        "packet-count": 2,
        "byte-count": 196,
        "duration": {
            "nanosecond": 687000000,
            "second": 2
        }
    },
    "data-ingress-port": {
        "port-id": "1",
        "node": "openflow:1",
        "port-name": "s1-eth1"
    },
    "virtual-route": [
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif1"
            },
            "reason": "PORTMAPPED",
            "order": 0
        },
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif2"
            },
            "reason": "FORWARDED",
            "order": 1
        }
    ],
    "data-egress-port": {
        "port-id": "1",
        "node": "openflow:2",
        "port-name": "s2-eth1"
    },
    "flow-id": 5,
    "data-ingress-node": {
        "tenant-name": "vtn1",
        "bridge-name": "vBridge1",
        "interface-name": "vif1"
    },
    "creation-time": 1558660283977,
```

```
        "data-flow-match": {
            "vtn-ether-match": {
                "ether-type": 2048,
                "source-address": "00:00:00:00:00:01",
                "vlan-id": 0,
                "destination-address": "00:00:00:00:00:02"
            },
            "vtn-inet-match": {
                "source-network": "10.0.0.1/32",
                "destination-network": "10.0.0.2/32",
                "protocol": 1
            }
        }
    },
    {
        "hard-timeout": 0,
        "data-egress-node": {
            "tenant-name": "vtn1",
            "bridge-name": "vBridge1",
            "interface-name": "vif1"
        },
        "physical-route": [
            {
                "physical-ingress-port": {
                    "port-id": "1",
                    "port-name": "s2-eth1"
                },
                "physical-egress-port": {
                    "port-id": "4",
                    "port-name": "s2-eth4"
                },
                "order": 0,
                "node": "openflow:2"
            },
            {
                "physical-ingress-port": {
                    "port-id": "2",
                    "port-name": "s3-eth2"
                },
                "physical-egress-port": {
                    "port-id": "1",
                    "port-name": "s3-eth1"
                },
                "order": 1,
                "node": "openflow:3"
            },
            {
                "physical-ingress-port": {
                    "port-id": "4",
                    "port-name": "s1-eth4"
                },
                "physical-egress-port": {
                    "port-id": "1",
```

```
                "port-name": "s1-eth1"
            },
            "order": 2,
            "node": "openflow:1"
        }
    ],
    "idle-timeout": 300,
    "data-flow-stats": {
        "packet-count": 2,
        "byte-count": 196,
        "duration": {
            "nanosecond": 925000000,
            "second": 2
        }
    },
    "data-ingress-port": {
        "port-id": "1",
        "node": "openflow:2",
        "port-name": "s2-eth1"
    },
    "virtual-route": [
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif2"
            },
            "reason": "PORTMAPPED",
            "order": 0
        },
        {
            "virtual-node-path": {
                "tenant-name": "vtn1",
                "bridge-name": "vBridge1",
                "interface-name": "vif1"
            },
            "reason": "FORWARDED",
            "order": 1
        }
    ],
    "data-egress-port": {
        "port-id": "1",
        "node": "openflow:1",
        "port-name": "s1-eth1"
    },
    "flow-id": 6,
    "data-ingress-node": {
        "tenant-name": "vtn1",
        "bridge-name": "vBridge1",
        "interface-name": "vif2"
    },
    "creation-time": 1558660283980,
    "data-flow-match": {
```

```
                    "vtn-ether-match": {
                        "ether-type": 2048,
                        "source-address": "00:00:00:00:00:02",
                        "vlan-id": 0,
                        "destination-address": "00:00:00:00:00:01"
                    },
                    "vtn-inet-match": {
                        "source-network": "10.0.0.2/32",
                        "destination-network": "10.0.0.1/32",
                        "protocol": 1
                    }
                }
            }
        ]
    }
}
```

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-
    type: application/json" -X GET http://192.168.10.120:8181/restconf/
    operational/vtn:vtns/
```

Listing 10: **L2E2**: Configuración de la VTN.

```
{
    "vtns": {
        "vtn": [
            {
                "name": "vtn2",
                "vtenant-config": {
                    "description": "VTN Escenario 2",
                    "idle-timeout": 300,
                    "hard-timeout": 0
                },
                "vbridge": [
                    {
                        "name": "vBridge2",
                        "vlan-map": [
                            {
                                "map-id": "ANY.200",
                                "vlan-map-status": {
                                    "active": true
                                },
                                "vlan-map-config": {
                                    "vlan-id": 200
                                }
                            }
                        ],
                        "vbridge-config": {
                            "age-interval": 600
                        },
                        "bridge-status": {
                            "state": "UP",
                            "path-faults": 0
```

```
                        }
                    },
                    {
                        "name": "vBridge1",
                        "vlan-map": [
                            {
                                "map-id": "ANY.100",
                                "vlan-map-status": {
                                    "active": true
                                },
                                "vlan-map-config": {
                                    "vlan-id": 100
                                }
                            }
                        ],
                        "vbridge-config": {
                            "age-interval": 600
                        },
                        "bridge-status": {
                            "state": "UP",
                            "path-faults": 0
                        }
                    }
                ]
            }
        ]
    }
}
```

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-
    type: application/json" -X GET http://192.168.10.120:8181/restconf/
    operational/vtn:vtns/
```

Listing 11: **L2E3**: Configuración de la VTN.

```
{
    "vtns": {
        "vtn": [
            {
                "name": "vtn3",
                "vbridge": [
                    {
                        "name": "vBridge1",
                        "vinterface": [
                            {
                                "name": "vif1",
                                "vinterface-status": {
                                    "state": "UP",
                                    "mapped-port": "openflow:1:5",
                                    "entity-state": "UP"
                                },
                                "port-map-config": {
                                    "node": "openflow:1",
```

```
            "port-name": "s1-eth5",
            "vlan-id": 0
        },
        "vinterface-config": {
            "description": "NAT",
            "enabled": true
        },
        "vinterface-input-filter": {
            "vtn-flow-filter": [
                {
                    "index": 11,
                    "condition": "cond_2",
                    "vtn-redirect-filter": {
                        "output": true,
                        "redirect-destination": {
                            "terminal-name": "
                                srv_fw_1",
                            "interface-name": "
                                sif11"
                        }
                    }
                },
                {
                    "index": 10,
                    "condition": "cond_1",
                    "vtn-redirect-filter": {
                        "output": true,
                        "redirect-destination": {
                            "terminal-name": "
                                srv_fw_1",
                            "interface-name": "
                                sif11"
                        }
                    }
                }
            ]
        }
    },
    {
        "name": "vif4",
        "vinterface-status": {
            "state": "UP",
            "mapped-port": "openflow:2:3",
            "entity-state": "UP"
        },
        "port-map-config": {
            "node": "openflow:2",
            "port-name": "s2-eth3",
            "vlan-id": 0
        },
        "vinterface-config": {
            "description": "h4 gateway",
            "enabled": true
```

```
                    }
                },
                {
                    "name": "vif2",
                    "vinterface-status": {
                        "state": "UP",
                        "mapped-port": "openflow:2:1",
                        "entity-state": "UP"
                    },
                    "port-map-config": {
                        "node": "openflow:2",
                        "port-name": "s2-eth1",
                        "vlan-id": 0
                    },
                    "vinterface-config": {
                        "description": "h2 gateway",
                        "enabled": true
                    }
                },
                {
                    "name": "vif3",
                    "vinterface-status": {
                        "state": "UP",
                        "mapped-port": "openflow:2:2",
                        "entity-state": "UP"
                    },
                    "port-map-config": {
                        "node": "openflow:2",
                        "port-name": "s2-eth2",
                        "vlan-id": 0
                    },
                    "vinterface-config": {
                        "description": "h3 gateway",
                        "enabled": true
                    }
                }
            ],
            "vbridge-config": {
                "age-interval": 600
            },
            "bridge-status": {
                "state": "UP",
                "path-faults": 0
            }
        }
    ],
    "vtenant-config": {
        "description": "VTN Escenario 3",
        "idle-timeout": 300,
        "hard-timeout": 0
    },
    "vterminal": [
        {
```

```
            "name": "srv_fw_1",
            "vinterface": [
                {
                    "name": "sif11",
                    "vinterface-status": {
                        "state": "UP",
                        "mapped-port": "openflow:1:4",
                        "entity-state": "UP"
                    },
                    "port-map-config": {
                        "node": "openflow:1",
                        "port-name": "s1-eth4",
                        "vlan-id": 0
                    },
                    "vinterface-config": {
                        "description": "vterminal sif11",
                        "enabled": true
                    }
                }
            ],
            "vterminal-config": {
                "description": "vterminal firewall"
            },
            "bridge-status": {
                "state": "UP",
                "path-faults": 0
            }
        },
        {
            "name": "srv_ids_2",
            "vinterface": [
                {
                    "name": "sif22",
                    "vinterface-status": {
                        "state": "UP",
                        "mapped-port": "openflow:2:6",
                        "entity-state": "UP"
                    },
                    "port-map-config": {
                        "node": "openflow:2",
                        "port-name": "s2-eth6",
                        "vlan-id": 0
                    },
                    "vinterface-config": {
                        "description": "vterminal sif22",
                        "enabled": true
                    },
                    "vinterface-input-filter": {
                        "vtn-flow-filter": [
                            {
                                "index": 10,
                                "condition": "cond_todas",
                                "vtn-redirect-filter": {
```

```
                                        "output": true,
                                        "redirect-destination": {
                                            "bridge-name": "
                                                vBridge1",
                                            "interface-name": "
                                                vif2"
                                        }
                                    }
                                }
                            ]
                        }
                    }
                ],
                "vterminal-config": {
                    "description": "vterminal ids"
                },
                "bridge-status": {
                    "state": "UP",
                    "path-faults": 0
                }
            },
            {
                "name": "srv_fw_2",
                "vinterface": [
                    {
                        "name": "sif12",
                        "vinterface-status": {
                            "state": "UP",
                            "mapped-port": "openflow:3:3",
                            "entity-state": "UP"
                        },
                        "port-map-config": {
                            "node": "openflow:3",
                            "port-name": "s3-eth3",
                            "vlan-id": 0
                        },
                        "vinterface-config": {
                            "description": "vterminal sif12",
                            "enabled": true
                        },
                        "vinterface-input-filter": {
                            "vtn-flow-filter": [
                                {
                                    "index": 11,
                                    "condition": "cond_todas",
                                    "vtn-redirect-filter": {
                                        "output": true,
                                        "redirect-destination": {
                                            "bridge-name": "
                                                vBridge1",
                                            "interface-name": "
                                                vif3"
                                        }
```

```
                                }
                            },
                            {
                                "index": 10,
                                "condition": "cond_1",
                                "vtn-redirect-filter": {
                                    "output": true,
                                    "redirect-destination": {
                                        "terminal-name": "
                                            srv_ids_1",
                                        "interface-name": "
                                            sif21"
                                    }
                                }
                            }
                        ]
                    }
                }
            ],
            "vterminal-config": {
                "description": "vterminal firewall"
            },
            "bridge-status": {
                "state": "UP",
                "path-faults": 0
            }
        },
        {
            "name": "srv_ids_1",
            "vinterface": [
                {
                    "name": "sif21",
                    "vinterface-status": {
                        "state": "UP",
                        "mapped-port": "openflow:3:4",
                        "entity-state": "UP"
                    },
                    "port-map-config": {
                        "node": "openflow:3",
                        "port-name": "s3-eth4",
                        "vlan-id": 0
                    },
                    "vinterface-config": {
                        "description": "vterminal sif21",
                        "enabled": true
                    }
                }
            ],
            "vterminal-config": {
                "description": "vterminal ids"
            },
            "bridge-status": {
                "state": "UP",
```

```
                        "path-faults": 0
                    }
                }
            ]
        }
    ]
  }
}
```