Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

ESTUDIO DEL ESTADO DEL ARTE DE LA INGENIERÍA DE TRÁFICO EN REDES SDN. CASO DE ESTUDIO OSHI.

TRABAJO FIN DE MÁSTER

María José Argüello Vélez

2015

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

ESTUDIO DEL ESTADO DEL ARTE DE LA INGENIERÍA DE TRÁFICO EN REDES SDN. CASO DE ESTUDIO OSHI.

Autor
María José Argüello Vélez

Director
David Fernández Cambronero

Departamento de Ingeniería de Sistemas Telemáticos

2015

Resumen

En la actualidad existe una alta demanda en cuanto a fiabilidad y alta capacidad de respuesta de las redes de telecomunicaciones, es por ello que los más interesados en controlar los flujos de datos que atraviesan las redes para poder brindar servicios de calidad y en tiempo real a sus usuarios son los ISPs, quienes han tratado de lidiar con este masivo crecimiento y expansión poniendo en práctica diferentes técnicas que se han ido desarrollando a través de las últimas décadas.

Tomando en cuenta la relevancia del control y la gestión de la red en ambientes tanto académicos como industriales y el gran auge en el que se encuentran las redes SDN actualmente, el presente trabajo tiene como objetivo mostrar dos enfoques de la ingeniería de tráfico sobre redes SDN.

Desde un ámbito general se propone el estudio de su estado del arte buscando profundizar en conceptos básicos y herramientas que han sido creadas para soportar la dirección del flujo de datos a través de estas redes, usando como base para el análisis las prestaciones que MPLS presenta sobre redes IP y las lecciones que este protocolo ya avanzado en la ingeniería de tráfico brinda sobre tecnologías medianamente jóvenes y que siguen en desarrollo como lo son las redes definidas por software. Para este análisis se toman en cuenta dos puntos claves en el desarrollo de TE, la gestión del flujo a través de la red y la capacidad de la red de reponerse ante fallos que puedan sufrir algunos de sus nodos.

Desde un ámbito más específico, se presenta la arquitectura y herramientas utilizadas en la implementación del proyecto de experimentación OSHI (Open Source Hybrid IP/SDN) desplegado sobre escenarios de red formados por máquinas virtuales utilizando Mininet, y se realizan pruebas de conectividad comprobando el funcionamiento detallado en la teoría sobre las topologías propuestas en este proyecto para el despliegue de servicios de líneas arrendadas virtuales (VLL) y Pseudo Wire (PW).

Abstract

There is currently a high demand for reliability and high responsiveness of telecommunications networks, is for that reason that the most interested part in controlling data flows that traverse these networks to provide quality services in real time to its users are ISPs, who have tried to deal with this massive growth and expansion by implementing different techniques that have been developed over the past decades.

Considering the importance of the control and management of the network in academic and industrial environments, and the boom of the SDN networks, this paper aims to show two approaches to SDN traffic engineering.

From a general view it proposes the study of the state of art for looking deeper into basic concepts and tools that have been created to support the direction of data flow through these networks, using as a basis for analyzing the performance of MPLS over IP networks and lessons that this protocol, quite advanced in traffic engineering, provides on young technologies and still in development such as software defined networks. For this analysis, two key points in the development of TE are relevant, flow management across the network and the network capacity to failures recovery in its nodes.

From a specific view, it shows the architecture and tools used in the implementation of the experimental project OSHI (Open Source Hybrid IP / SDN), deployed on network scenarios which are based in virtual machines using Mininet, and connectivity tests are done for checking the operations detailed in theory about topologies proposed in this project for the deployment of Virtual Leased Line (VLL) and Pseudo Wire (PW) services.

Índice general

Índice de figuras

Siglas

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
EU FIRE	Future Internet Research and Experimentation Initiative
GENI	Global Environment for Networking Innovations
iBGP	Internal Border Gateway Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IP FE	IP Forwarding Engine
ISP	Internet Service Provider
MAN	Metropolitan Area Network
MPLS	Multiprotocol Label Switching
MPLS-TE	Multiprotocol Label Switching Traffic Engineering
OFCS	Open Flow Capable Switch
ONF	Open Networking Foundation
OPENSIG	Open Signalling Working Group
OSHI	Open Source Hybrid IP/SDN
SDN	Software Defined Networking
VNX	Virtual Networks over Linux

1 Introducción

En la actualidad existe una alta demanda en cuanto a fiabilidad y alta capacidad de respuesta de las redes de telecomunicaciones, debido a la gran cantidad de información compartida e intercambiada entre los usuarios de diferentes tipos de aplicaciones que cada vez exigen un mejor desempeño del internet y por ende de todas las tecnologías que lo rodean. Es por ello, que los más interesados en controlar los flujos de datos que atraviesan las redes para poder brindar servicios de calidad y en tiempo real a sus usuarios son los ISPs, quienes han tratado de lidiar con este masivo crecimiento y expansión poniendo en práctica diferentes técnicas que se han ido desarrollando a través de las últimas décadas.

Inicialmente, se intentaba predecir el comportamiento del flujo de datos en la red y entorno a este 'supuesto' comportamiento futuro se realizaban las configuraciones de los circuitos de la red, es decir, se realizaba ingeniería de red pura, lo que acarreaba una demora significativa en la instalación de nuevos circuitos. Luego se dio paso a la ingeniería de tráfico [1], que permite manipular el flujo de datos para que se ajuste a la red, es decir, que los administradores de la red configuren sus equipos para que los flujos sigan un camino u otro dependiendo del tipo de datos del flujo; con ello se busca que no existan recursos físicos excesivamente utilizados que provoquen pérdidas de enlaces, mientras que otros queden infrautilizados.

Desde el uso de redes ATM hasta el día de hoy, la ingeniería de tráfico ha sido ampliamente estudiada y puesta en marcha usando técnicas que van desde la conmutación ATM, pasando por el encaminamiento de paquetes IP mediante el ajuste de métricas en las interfaces en redes IPv4 y el proceso de inserción, intercambio y eliminación de etiquetas en redes IP/MPLS que es muy usado actualmente; hasta técnicas en esquemas creados más recientemente para redes SDN que son consideradas como el futuro de las redes.

Dentro de este enfoque de la evolución del control de la red, MPLS ha sido un protocolo de mucha relevancia que se sigue usando aún, tanto para brindar servicios a clientes como para la gestión de los flujos, esto debido a que ha alcanzado un nivel de madurez alto. Sin embargo, debido a la complejidad de la red protocolos de

encaminamiento como OSP e IS-IS han tenido que extenderse (OSFP-TE[1], IS-IS TE[2]) para soportar ingeniería de tráfico con MPLS haciendo más complejo el control.

Por otro lado, mientras se buscaban mecanismos para mejorar el control en las redes tradicionales, desde la década del 90 se realizaron esfuerzos de investigación para hacer estas redes programables con el objetivo de alcanzar ese mismo control. Pero no fue sino hasta inicios de la década del 2000 que se empiezan a crear interfaces abiertas como ForCES[3] para separar el plano de control del plano de datos, buscando independencia del control de la red [2]. Siguiendo esta misma iniciativa se capta la atención de la industria sobre la importancia de la gestión y programabilidad de la red hasta llegar a la creación de grupos de investigación universitarios, organizaciones como la ONF y estandarizaciones como la del protocolo OpenFlow, que es ya bien conocida hoy en día.

Justamente, tomando en cuenta la relevancia del control y la gestión de la red en ambientes tanto académicos como industriales y el gran auge en el que se encuentran las redes SDN actualmente, este trabajo se centra en el estudio del estado actual de la ingeniería de tráfico sobre redes SDN que se ha ido desarrollando a lo largo de los últimos años, y específicamente en el análisis de la arquitectura de los nodos usados en el caso de estudio OSHI [3]. Además, se realizan comprobaciones del funcionamiento interno de los nodos en el despliegue de servicios, mediante pruebas de conectividad y análisis de las tablas de flujo sobre una máquina virtual lista para usar previamente instalada y configurada por el grupo de investigación.

1.1 Contexto

OSHI fue escogido para el análisis debido a que es un proyecto de código abierto que tiene soporte por el grupo que lo desarrolló, ya que está siendo mejorado continuamente. Además, frente a otros enfoques para realizar ingeniería de tráfico sobre redes SDN, este proyecto busca proveer conectividad híbrida, es decir, considera la capacidad de los nodos de lidiar con encaminamiento IP al mismo tiempo que puede gestionar tráfico OpenFlow, buscando interoperabilidad con dispositivos en el núcleo de la red que no hablan OpenFlow y tolerancia a fallos basada en estándares IP.

[1] Extensión de OSPF para ingeniería tráfico, RFC 3630: https://tools.ietf.org/html/rfc3630.
[2] Extensión de IS-IS para ingeniería de tráfico, RFC 3784: https://tools.ietf.org/html/rfc3784.
[3] Forwarding and Control Element Separation define un framework y protocolos asociados para la estandarización del intercambio de información entre el plano de control y el plano de datos, según la RFC 3746

De las dos opciones de despliegue para pruebas que ofrece OSHI, se escogió el despliegue de escenario con máquinas virtuales en Mininet en vez del despliegue en el testbed OFELIA[4], debido a que viene predefinido en los ejemplos que el grupo de investigación propone sobre la virtual lista para usar, lo que facilita el análisis a realizar sobre este caso.

1.2 Objetivos

El objetivo principal de este trabajo consiste en realizar un estudio del estado del arte en ingeniería de tráfico sobre las redes SDN para profundizar en los conceptos básicos y las herramientas que rodean a esta tecnología medianamente nueva y de gran importancia para el desarrollo de las redes y las comunicaciones. Además, se realizará la presentación y análisis del proyecto de experimentación OSHI que usa componentes de código abierto para su implementación.

A partir de este objetivo general, se desarrollarán los siguientes objetivos específicos:

- Profundizar en el funcionamiento de la ingeniería de tráfico con MPLS sobre redes SDN para realizar un análisis acertado del caso de estudio.
- Afianzar los conocimientos sobre ingeniería de tráfico con MPLS en las redes tradicionales para poder realizar una comparación objetiva con su aplicación en redes SDN.
- Analizar el estado actual de la ingeniería de tráfico en redes SDN tomando como base dos enfoques: la gestión de flujo y la tolerancia a fallos.
- Estudiar la arquitectura y las herramientas utilizadas para la implementación del proyecto OSHI.
- Realizar un análisis del funcionamiento del proyecto OSHI en el despliegue de servicios MPLS basándome en la evaluación realizada por el grupo de investigación del proyecto y en las necesidades planteadas para implementaciones de MPLS-TE sobre redes SDN.

1.3 Metodología

Este trabajo de fin de master se realizará en 4 fases que se detallan a continuación:

[4] OFELIA (OpenFlow in Europe: Linking Infraestructure and Applications), es un proyecto colaborativo dentro del programa de trabajo de la comisión Europea que permite a los investigadores experimentar sobre un testbed extendiendo su red dinámicamente, está basado en OpenFlow. Para mayor información, consultar: http://www.fp7-ofelia.eu/about-ofelia/.

1. En la primera fase se pretende profundizar en las bases teóricas de la arquitectura, protocolos y herramientas que forman parte de las redes SDN; así como en el estado del arte de la ingeniería de tráfico y su funcionamiento con MPLS tanto en redes tradicionales como en redes SDN. Esta fase estará dividida en cuatro tareas:
    a. Definición de los temas y subtemas a indagar y profundizar en torno a la ingeniería de tráfico y las redes SDN.
    b. De acuerdo a lo definido, realizar la búsqueda de la información necesaria.
    c. Selección de fuentes de información válidas para la documentación del trabajo.
    d. Realización de un resumen sobre el estado del arte encontrado en la búsqueda documental.
2. En esta fase se investigarán los proyectos emergentes usados para probar ingeniería de tráfico sobre SDN hasta el momento, para determinar cuál de ellos puede ser usado para las pruebas de experimentación. Aquí se realizarán las siguientes tareas:
    a. Levantamiento de información sobre las herramientas o proyectos más actuales que hayan sido probados para ingeniería de tráfico sobre SDN.
    b. Evaluación y selección de una herramienta de virtualización de red sobre la que se realizarán las pruebas. Entre las herramientas conocidas hasta ahora para esta tarea tenemos a Mininet [3], que es un emulador de red que tiene soporte para el protocolo OpenFlow y VNX (Virtual Networks over Linux) [4], que es una herramienta de virtualización de red de código abierto desarrollada en el Departamento de Ingeniería Telemática de esta escuela.
    c. Selección de la tecnología o proyecto a analizar.
    d. Presentación y comprensión del proyecto escogido.
    e. Pruebas de MPLS en el plano de datos SDN.
    f. Análisis de los resultados obtenidos en el caso de estudio.
3. Luego de la fase de experimentación, se realizará una evaluación del proceso de comprobación de la arquitectura y funcionamiento del proyecto. Las conclusiones obtenidas en esta fase serán recogidas en un apartado del documento final.

4. Con la experticia y las bases teóricas obtenidas, se realizará un análisis del proyecto probado y analizado, comparándolo con las demás soluciones y herramientas que permiten realizar TE sobre SDN.

1.4 Estructura de la memoria

A lo largo de este trabajo se describe el marco teórico sobre el que se basa la ingeniería de tráfico aplicada sobre las redes definidas por software, además del análisis de un caso de experimentación usando MPLS-TE basado en el protocolo OpenFlow, para ello se desglosa el contenido en 5 capítulos.

En el capítulo 2, se definen los conceptos relacionados a las redes definidas por software, su evolución, arquitectura y el estándar OpenFlow. Adicionalmente, se presenta una breve descripción del protocolo MPLS, algunas definiciones sobre los servicios que brinda esta tecnología y una mención sobre la ingeniería de tráfico en estas redes.

El capítulo 3 presenta un resumen sobre las ventajas, desventajas, soluciones y campos de investigación abiertos alrededor de la ingeniería de tráfico sobre redes definidas por software, enfocándose principalmente en la gestión de flujo de la red y la capacidad de la red de recuperarse ante fallos.

En el capítulo 4 se presenta la arquitectura y detalles de funcionamiento de los nodos del proyecto OSHI, además de mostrar y describir las pruebas realizadas sobre los escenarios de servicios VLL IP y PW que el grupo de investigación plantea sobre una máquina virtual.

Para finalizar, el capítulo 5 presenta las conclusiones obtenidas sobre el trabajo realizado y detalla las mejoras a realizarse para trabajos futuros.

2 Marco Conceptual

En este apartado se presenta el estado en el que se encuentran las redes definidas por software actualmente y la evolución que han tenido en el tiempo, en conjunto con los conceptos de ingeniería de tráfico sobre redes MPLS.

2.1 Evolución de las redes programables

A pesar de que en estos últimos años se ha escuchado mucho sobre SDN y que parece ser una tecnología que ha emergido recientemente, existe una historia de pasos previos detrás de ella, un empeño por hacer las redes más programables. Este proceso empieza hace veinte años, cuando el Internet recién despegaba, teniendo un gran éxito y presentando grandes desafíos en el manejo y la evolución de la infraestructura de red. Según el enfoque usado en [2], la historia ha sido dividida en tres etapas en las que también se ha ido desarrollando la virtualización de redes, como se observa en la figura 1.

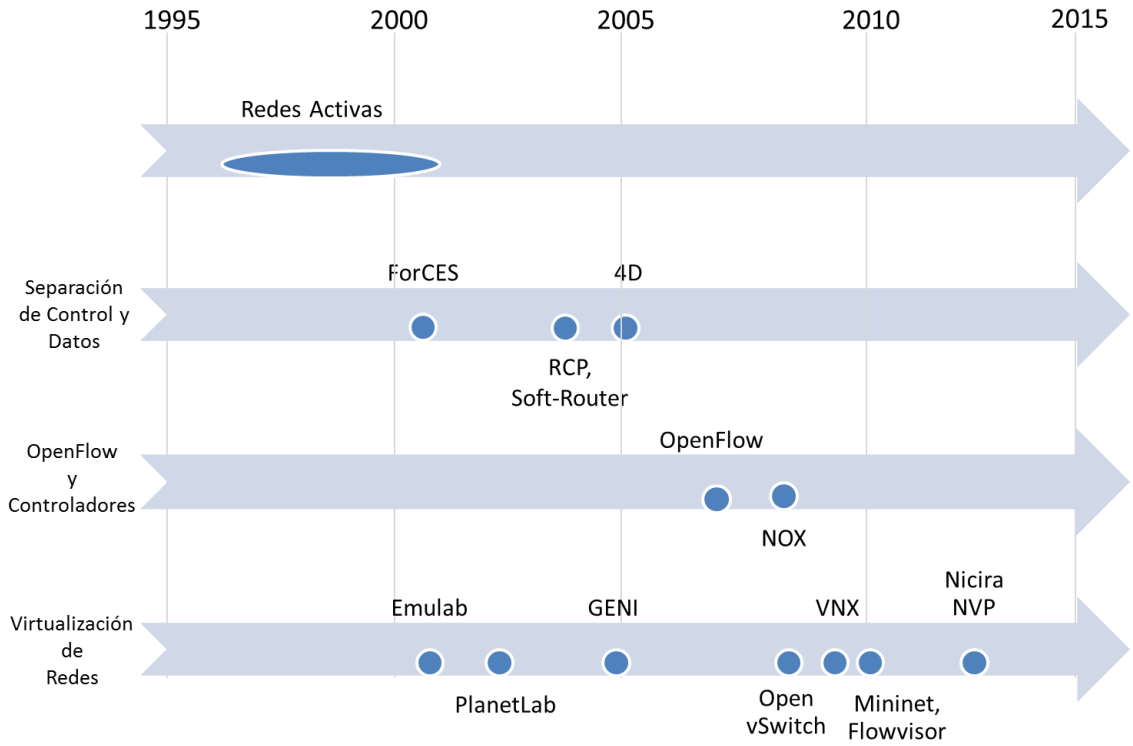


Figura 1. Algunos desarrollos relevantes realizados en los últimos 20 años sobre redes programables y su relación con la virtualización de redes. [2]

A mediados de los años 90 el Grupo de Trabajo de Señalización Abierta, OPENSIG por sus siglas en inglés, empezó sus exploraciones para conseguir un control más abierto de la red mediante la conceptualización de una interfaz de programación o API

de red que básicamente buscaba añadir características de escalabilidad y automatización soportando la construcción de funcionalidades personalizadas que se aplicaban a cada paquete que pasaba por un nodo de la red, a este nuevo enfoque se lo denominó como Redes Activas que se materializó en la creación de programas que apoyaban la investigación de este enfoque como el creado por DARPA[5] en los Estados Unidos.

En la primera década del 2000, un grupo de investigadores cercano a los operadores de red hicieron varios esfuerzos por separar el plano de datos del plano de control y explorar arquitecturas de control centralizado para dar solución a las dificultades encontradas al realizar ingeniería de tráfico usando protocolos convencionales de encaminamiento; como resultado a esta búsqueda y al incremento de la velocidad en los enlaces del *backbone* de las redes junto con los intentos de los ISPs por incrementar el alcance de sus redes para responder a altas demandas de nuevos servicios y mayor fiabilidad, se aceleró el proceso de innovación en el control de la red dando paso a interfaces abiertas entre el plano de control y de datos, tales como ForCES que fue estandarizada por la IETF y Netlink que permite el reenvío de paquetes entre el kernel y los procesos de espacios de usuario en Linux [6].

Además, se crearon arquitecturas como RCP[6] (Plataforma de Control de Encaminamiento) y Soft-Router para centralizar el control de la red, RCP es un *framework* que provee escalabilidad y robustez mediante la selección de las rutas y la comunicación de las mismas a los *routers* dentro de un mismo sistema autónomo usando el protocolo iBGP [7], por su parte Soft-Router separa los elementos de la red clasificándolos en elementos de control o elementos de reenvío de información que luego son asociados mediante enlaces para formar un solo elemento de red [8]. A raíz de esta tendencia de separación del control y los datos en la red, los investigadores empezaron a plantearse arquitecturas que permitieran empezar de cero con nuevas planteamientos, es así que se concibió el proyecto 4D cuyos componentes básicos eran: el plano de decisión, para obtener una visión amplia y realizar un control centralizado de la red; el plano de descubrimiento, para la recolección de datos de topología de red; el plano de diseminación, para un control directo de la red mediante la instalación de reglas de procesamiento de paquetes y el plano de datos, para el reenvío de tráfico.

A mediados de los 2000, diferentes grupos de investigación alentados por el éxito de infraestructuras experimentales como Emulab[7] y PlanetLab[8], empezaron a realizar

[5] Defense Advanced Research Projects Agency.

[6] Routing Control Platform

[7] Emulab es un banco de pruebas de red que permite a los investigadores experimentar en el desarrollo, depuración y evaluación de sus sistemas, es pública y puede ser usada sin cargos.

[8] PlanetLab es un escenario de pruebas a gran escala, conformado por un conjunto de servidores distribuidos geográficamente que permite desarrollar, desplegar y acceder a servicios a escala planetaria.

pruebas con redes a escala debido al gran interés y la disponibilidad de fondos del gobierno estadounidense en ese momento, dando paso a la formación de nuevos grupos como el Entorno Global de Innovaciones de Redes conocido como GENI por sus siglas en inglés, también se crearon programas a este mismo nivel como el GENI Project Office en los Estados Unidos y el EU FIRE en la Unión Europea; por otro lado un grupo de investigadores de la Universidad de Stanford creó el Programa Clean Slate (Borrón y Cuenta Nueva) [2] que se centraba en un entorno de experimentación a una escala más local y accesible, las redes de datos de los campus universitarios.

En este programa se origina la creación del paradigma de las Redes definidas por Software, comúnmente conocido como SDN; un término inicialmente usado para referirse únicamente al protocolo OpenFlow, que es la primera interfaz estándar diseñada específicamente para SDN, la cual será profundizada en las siguientes secciones. De la mano de OpenFlow se diseñan controladores como NOX que impulsan la creación de nuevas aplicaciones de control. Para dar impulso a este nuevo concepto y luego de constatar los resultados obtenidos en el programa Clean Slate, se crea la Fundación de Redes Abiertas conocida como ONF por sus siglas en inglés, una organización sin fines de lucro que fue lanzada inicialmente por la unión de varias empresas interesadas en acelerar la entrega y comercialización de SDN y de productos y servicios derivados de ella, como Deutsche Telekom, Facebook, Google, Microsoft, Verizon y Yahoo [9].

A finales de la primera década del 2000, el grupo de investigación de la Universidad de Stanford demostró las capacidades de OpenFlow a nivel académico empezando a desplegar bancos de pruebas entre varios campus formando un backbone de red WAN que extendió la oportunidad de experimentar y transformar este paradigma en una realidad, a raíz de ello se ampliaron casos de uso en centros de datos y backbones de redes privadas además de algunas investigaciones recientes para configuraciones de redes domésticas, redes empresariales, puntos de intercambio de internet, núcleos de redes celulares y redes de acceso WiFi.

La Virtualización de Redes es una tecnología que ha estado estrechamente vinculada a las redes definidas por software en los últimos años, evolucionando en conjunto aunque sean conceptualmente independientes; ya que la virtualización de red está más relacionada con el uso de una misma infraestructura física compartida por múltiples redes lógicas virtuales que trabajan individualmente y pueden tener una topología diferente a la red física original, sin embargo es un caso de uso de las SDNs. Uno de los usos más extendidos es la puesta en marcha de escenarios para evaluar y

probar la arquitectura de SDN, mediante herramientas como Mininet[9] o VNX que permiten desplegar switches virtuales OpenFlow (Open vSwitch) como parte de la red, además de la división del flujo de tráfico en '*slices*' o porciones diferentes mediante Flowvisor[10] para brindar servicios personalizados a cada porción del tráfico, también podemos añadir el uso de redes virtuales en cloud computing para proveer redes superpuestas sobre una misma infraestructura física compartida a múltiples clientes mediante herramientas como Nicira[11].

2.2 Redes Definidas por Software

Las redes definidas por software tienen como objetivo proporcionar interfaces abiertas que permitan el desarrollo de software que pueda controlar la conectividad provista por un conjunto de recursos de la red y el flujo de tráfico sobre ellos. Esto debido a que su arquitectura permite separar el plano de control del plano de datos, obteniendo una inteligencia y estado de la red centralizados lógicamente y una infraestructura de red que queda enteramente abstraída de las aplicaciones; proporcionando redes altamente escalables y flexibles que se adaptan rápidamente a las necesidades cambiantes de los negocios [10].

2.2.1 Arquitectura

La arquitectura de las redes definidas por software fue diseñada siguiendo tres principios básicos, que pretenden solucionar los problemas que presentan las redes tradicionales actualmente:

1. Separación de los controladores y el plano de datos, que permite ejercer control sobre los recursos de la red a pesar de que el plano de datos se encuentre separado del plano de control, para ello se define una interfaz del plano controlador o CPI por sus siglas en inglés, que es la intermediaria entre el controlador SDN y los elementos de la red permitiendo al controlador delegar funcionalidades importantes a los elementos de la red mientras se mantiene el estado del elemento.
2. Control lógicamente centralizado, que permite tener una perspectiva más amplia de los recursos de la red, mejora la capacidad de tomar decisiones sobre

[9] Mininet es un emulador de software que permite la creación de prototipos de una red en una sola máquina.

[10] Flowvisor es una herramienta considerada como un controlador OpenFlow de propósito especial que actúa como un proxy entre los switches y múltiples controladores OpenFlow, permitiendo la división de una red en instancias diferentes.

[11] Nicira Network Virtualization Platform (NVP) es una plataforma de virtualización basada en software que permite el despliegue de redes virtuales en centros de datos proporcionando una capa de abstracción entre sistemas finales y la infraestructura de la red.

el despliegue de los mismos y provee escalabilidad al tener una visión más global y menos detallada de los recursos.

3. Exposición del estado y de los recursos abstractos de la red a aplicaciones externas.

Siguiendo estos principios, la arquitectura SDN está compuesta por tres capas y dos grupos de interfaces de aplicación o APIs abiertas [10], como se observa en la figura 2. La capa de plano de datos está formada por los elementos de red que pueden ser dispositivos programables o tradicionales, entre los programables están los switches OpenFlow, que realizan el trabajo ya conocido de reenvío de tráfico de la red según las indicaciones recibidas en su comunicación con el controlador SDN mediante las interfaces south-bound. Las interfaces south-bound son las encargadas de entregar las órdenes del controlador a cada uno de los switches OpenFlow y de la comunicación entre ellos. El protocolo OpenFlow es la interfaz south-bound más comúnmente usada ya que es la primera diseñada para redes definidas por software.

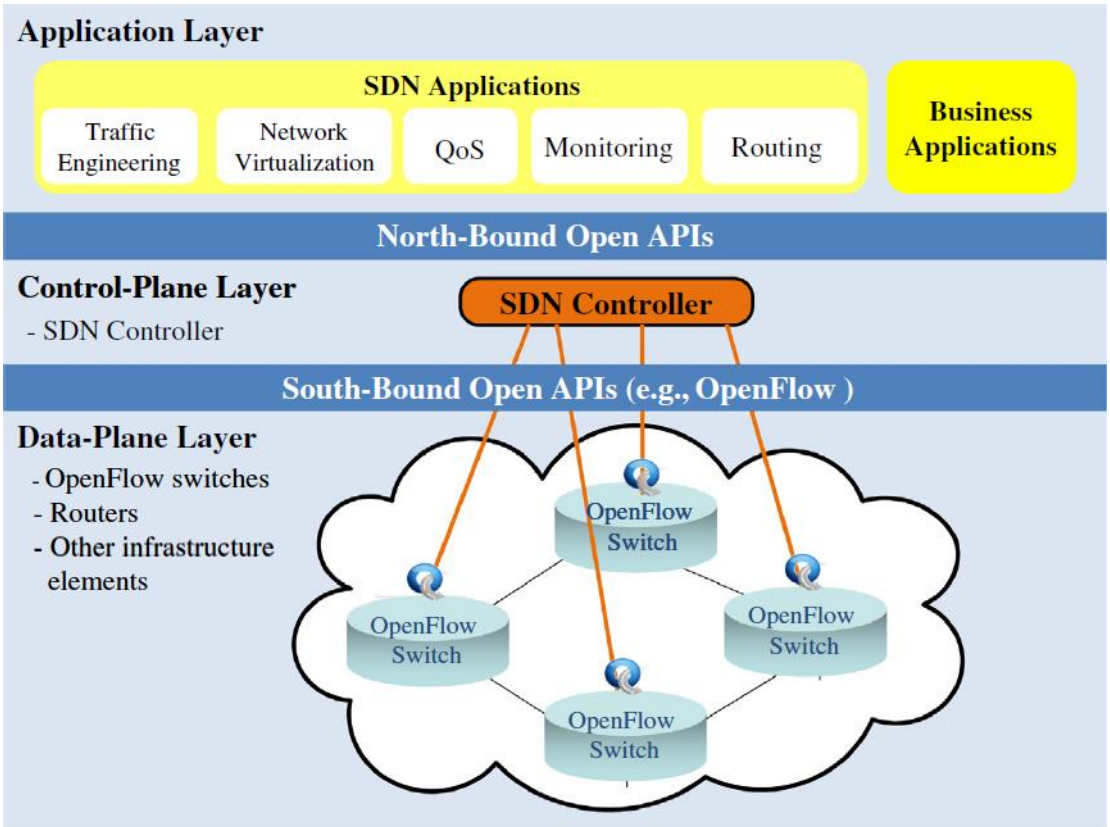


Figura 2. Arquitectura SDN [12].

La capa central es la del plano de control, en la que puede existir más de un controlador, dependiendo de la extensión o el área que cubra la red, los controladores traducen los requerimientos de las aplicaciones y ejercen un control más específico sobre los elementos de la red, mientras proveen información hacia las aplicaciones SDN. Un controlador se comunica con las aplicaciones de red mediante las interfaces

north-bound, que traducen los requerimientos de las aplicaciones y políticas de la red al controlador habilitando servicios en la red, como encaminamiento, ingeniería de tráfico, seguridad, calidad de servicio, uso de la energía, y otros servicios para gestionar la red. Por último, la capa de aplicación alberga a las aplicaciones SDN que envían las órdenes para gestionar la red según las necesidades de los usuarios finales.

Esta arquitectura asegura características de agilidad, automatización, escalabilidad y gestión centralizada de la red que proveen beneficios como ancho de banda bajo demanda para compañías telefónicas y proveedores de servicios, virtualización de red para centros de datos, y control de acceso a la red y monitoreo para campus universitarios o de negocios. Pese a que existe un gran interés sobre las redes SDN por parte de la industria y se han realizado además variedad de desarrollos relativos a las interfaces south-bound y al controlador tanto a nivel industrial como educativo además de la estandarización del protocolo OpenFlow, las interfaces north-bound han recibido menor atención por lo que la ONF ha formado recientemente un grupo para su estudio, el Northbound Working Group, que estará a cargo de escribir código y desarrollar prototipos que originen el establecimiento de estándares para la interfaz [13].

2.2.2 Estándar OpenFlow

OpenFlow es una instancia de la arquitectura SDN, un conjunto de especificaciones establecidas por la ONF (Open Networking Foundation) que consta de tres elementos: un controlador, el protocolo OpenFlow y un dispositivo de red (switch). Por medio del dispositivo de red o *switch* se realiza el procesamiento de paquetes usando una combinación de contenidos de paquetes y estados de configuración del switch, y por medio del controlador se gestionan los estados de configuración de muchos switches y se responde a los eventos usando el protocolo OpenFlow [14].

El protocolo OpenFlow es uno de los primeros diseñados específicamente para trabajar en redes SDN para garantizar el intercambio de datos y permitir la evolución de la red. Este protocolo basa toda la gestión tanto del comportamiento del flujo de datos que circula por la red como del dispositivo de red en el controlador, específicamente maneja los estados de configuración y la recepción de ciertos eventos del switch. El *switch* posee un canal de comunicación que es la interfaz que conecta cada *switch* al controlador y por medio de la cual el controlador configura y gestiona el swicth, usando el protocolo para añadir, actualizar y eliminar entradas de flujo en las tablas de flujo del switch tanto de manera reactiva, en respuesta a paquetes como de manera proactiva, anticipándose al comportamiento del tráfico en la red.

Como se observa en la figura 3, se pueden encontrar cuatro componentes en el protocolo: la capa de mensajes, máquina de estado, la interfaz del sistema y la configuración.

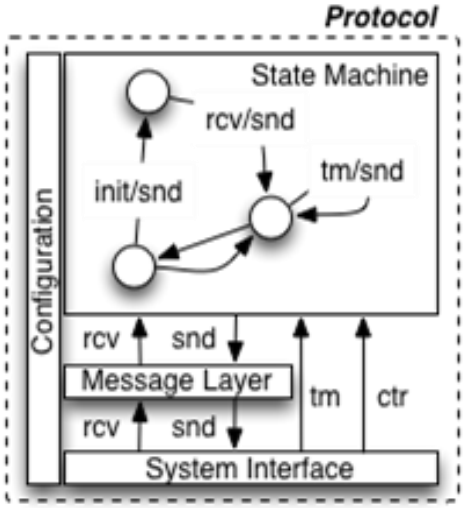


Figura 3. Componentes del Protocolo OpenFlow [14].

2.2.2.1 Capa de Mensajes

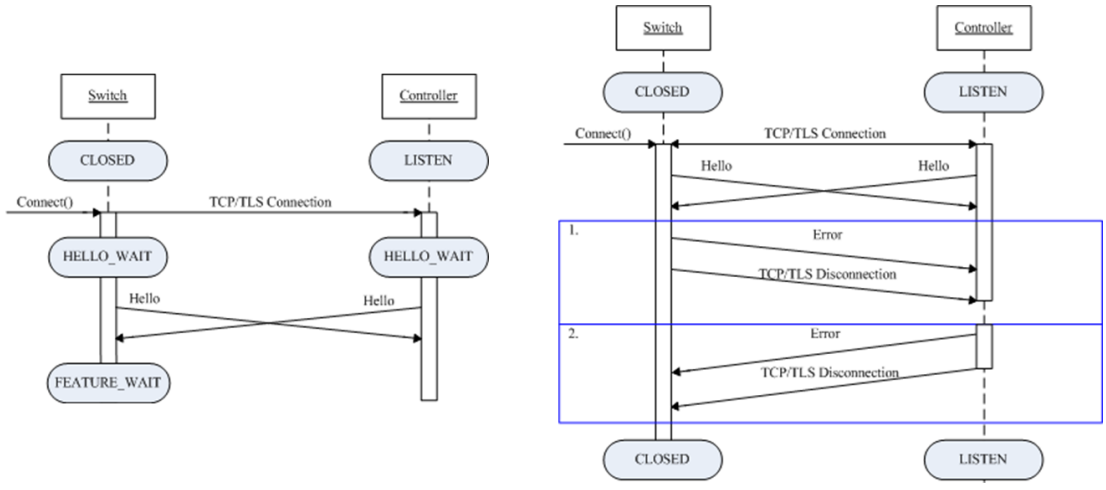
Esta capa es el núcleo de la pila del protocolo, que define la estructura y semántica válida para los mensajes entre el switch y el controlador; soporta tres tipos de mensajes: controlador-a-switch, asíncronos y simétricos, cada uno con múltiples subtipos [15].

- *Controlador-a-Swicth.*- Son mensajes iniciados por el controlador y que pueden o no requerir una respuesta del switch. Estos mensajes permiten consultar parámetros de configuración, enviar estados de los switches o configurar el envío de paquetes escogiendo un puerto específico.
- *Asíncronos.*- Son mensajes enviados sin que el controlador los haya solicitado al switch, los switches los envían al controlador para informar sobre la llegada de un paquete, el cambio de estado de un switch o algún error ocurrido.
- *Simétricos.*- Estos mensajes son enviados sin solicitud, tanto por el controlador como por el switch. Pueden enviarse mensajes de Hello, que son intercambiados entre el switch y el controlador en el inicio de la conexión; mensajes echo request y echo reply, que son utilizados para verificar si una conexión entre controlador y switch se encuentra activa; y mensajes tipo experimenter que sirven para proveer de funcionalidades adicionales a los switches OpenFlow y para futuras revisiones del protocolo. Además están los mensajes error que se utilizan para notificaciones de problemas en las conexiones.

2.2.2.2 *Máquina de Estado*

La máquina de estado es una capa en la que se define el comportamiento a bajo nivel del núcleo del protocolo, es decir, donde se describen los procedimientos de acciones como negociación, descubrimiento, control de flujo o entrega de mensajes. Debido a que en este protocolo casi todos los mensajes son asíncronos, no requieren un estado que manejar, excepto por el procedimiento de establecimiento de la conexión entre los elementos, que es capturado en la máquina de estado para manejar las negociaciones de la versión del protocolo y de la capacidad a usar por los elementos en la comunicación, y que estos mensajes de negociación puedan realizarse antes de que se intercambien los demás mensajes. Para que la conexión sea establecida entre el switch y el controlador, se deben realizar dos fases: la negociación de la versión del protocolo a usar en la comunicación y el descubrimiento de las características del switch por el controlador [14].

En primer lugar, se establece la conexión a más bajo nivel entre los elementos, ya sea TCP o TLS, antes de empezar la negociación de la versión, luego el switch y el controlador intercambian mensajes HELLO y cada dispositivo analiza el mensaje recibido para decidir la versión a usar, es posible que los dispositivos no coincidan en las versión del protocolo que soportan o que falle la recepción del mensaje HELLO en cuyo caso la negociación falla, la conexión física es terminada y ambos dispositivitos son restablecidos a un estado inicial como se observa en la figura 4b; por el contrario si los dispositivos coinciden en una de las versiones del protocolo soportadas y la negociación es exitosa, el estado de máquina de los dos dispositivos entra a la siguiente fase como se observa en la figura 4a.



a) Secuencia Normal

b) Secuencia Fallida

Figura 4. Establecimiento de Conexión OpenFlow. a) Secuencia de conexión cuando las versiones de los protocolos de los elementos coinciden, b) Secuencia de conexión cuando las versiones de los protocolos de los elementos no coinciden [14].

Lo que pretende la fase de descubrimiento de características, también llamada 'Handshake', es que el controlador sea consciente de las capacidades del switch con el que se está comunicando. Para ello el controlador envía un mensaje de petición de características al switch y el switch devuelve un mensaje de respuesta de características al controlador que contiene todas sus capacidades y la conexión queda establecida, en caso de que el mensaje de respuesta no sea recibido el controlador termina la conexión de bajo nivel con el switch.

2.2.2.3 Interfaces del Sistema

El protocolo provee servicios a los otros componentes del sistema (*switch* y controlador) por medio de sus interfaces. Se presentan a continuación cuatro interfaces identificadas en el protocolo:

- La interfaz TCP/TLS, es aquella que interactúa con los protocolos de más bajo nivel en la pila de protocolos y proporciona transmisión de flujo entre el switch y el controlador.
- Una interfaz del agente switch, que interactúa con el kernel del switch OpenFlow, reenviando mensajes del controlador al kernel del switch para que sean procesados, y recibiendo mensajes asíncronos de los switches y reenviándolos a la pila OpenFlow para procesarlos y trasmitirlos al controlador.
- Una interfaz de aplicación del controlador, que comunica a las aplicaciones del controlador que funcionan en el nivel más alto de la pila del protocolo OpenFlow, con el switch mediante el procesamiento y la transmisión de mensajes desde la pila OpenFlow.
- Una interfaz de configuración, por medio de la cual el operador del sistema puede configurar parámetros de la pila OpenFlow antes o durante el uso del sistema.

Por otro lado, la capa de configuración se refiere a todos los aspectos del protocolo que pueden ser configurados o que tienen valores iniciales, desde el tamaño por defecto de un *buffer* hasta los intervalos de respuesta de certificados X.509 y que son manejados mediante un lenguaje y una utilidad de configuración conjuntamente [14].

2.3 Redes MPLS

MPLS es un estándar IP de la IETF, definido en el RFC 3031 [15] que permite la conmutación de paquetes mediante el uso de etiquetas y opera entre la capa de enlace de datos y la capa de red del modelo OSI, combinando las capacidades de gestión y rendimiento de tráfico de la capa 2 con las ventajas de flexibilidad, escalabilidad y rendimiento de encaminamiento de la capa 3. Este protocolo es usado en las redes

actuales por los ISPs principalmente para dos propósitos, brindar servicios de internet y de redes privadas virtuales a los clientes; y dirigir el tráfico de la red cubriendo principalmente cuatro enfoques: la gestión de flujo, la tolerancia a fallos, actualización de la topología y el análisis de tráfico [12].

2.3.1 Definición de Conceptos

A continuación se detallan conceptos importantes sobre algunos servicios que brinda la tecnología MPLS, los cuales será necesario definir para el análisis posterior de las necesidades en el despliegue de MPLS sobre redes SDN.

2.3.1.1 Servicios MPLS

Uno de los servicios de mayor interés, brindado por los ISPs mediante el protocolo MPLS, es la implementación de redes privadas virtuales. Una VPN (Virtual Private Network) es una red privada construida sobre una infraestructura de red pública como Internet, que busca proporcionar el máximo nivel de seguridad posible a través de comunicación cifrada y políticas de seguridad. Es virtual debido a que físicamente no existe una infraestructura dedicada entre los usuarios conectados a los bordes de la misma.

MPLS permite crear redes privadas virtuales que operan en capa 2, además de las que operan en capa 3. En una VPN de capa 3, la naturaleza virtual de la red es aparente, debido a que está realmente formada por routers virtuales, en la que cada cliente puede solamente ver información de sus propias tablas VRF (Virtual Routing and Forwarding). Por otro lado, en las VPN de capa 2, el servicio es brindado sobre una colección de routers pero que no necesitan gestionar el envío de datos con direcciones IP debido a que los routers clientes aparentan estar conectados dentro de una misma gran LAN "virtual", en lados opuestos de un mismo enlace [17].

A continuación se presentan las definiciones de los servicios VLL de capa 3 y PWE3 de capa 2 que se brindan sobre redes MPLS:

- IP Virtual Leased Lines.- Este servicio está basado en el servicio de leased lines o líneas arrendadas físicas que proveían una conexión dedicada física al tráfico de un solo cliente, por lo que son más seguras pero de costo muy elevado. La virtualización de estas entidades permite proveer conexiones punto a punto basadas en Ethernet sobre redes IP/MPLS disminuyendo el coste que conlleva el despliegue de sus predecesoras, son conocidas también como Ip sobre MPLS (IPoMPLS).

- Pseudo Wire Emulation Edge-to-Edge (PWE3).- Según la especificación de la IETF en la RFC 3985 [19], este es un mecanismo que emula los atributos esenciales de un servicio de telecomunicaciones sobre una PSN[12], es decir,

[12]Packet Switched Networks (Red Conmutada por Paquetes).

emula un enlace punto-punto o punto-multipunto que puede proveer un servicio único que el usuario percibe como un enlace no compartido, tal y como si existiera un cable completamente transparente entre ellos.

2.3.2 Ingeniería de Tráfico en Redes MPLS

La capacidad de soportar definición anticipada de rutas para un paquete entre un origen y un destino es una de las características que han hecho de MPLS la tecnología usada por defecto para dirigir los flujos de datos sobre las redes IP, es decir, para realizar ingeniería de tráfico. Debido a que, comparado con el encaminamiento salto a salto tradicional en redes IP, las rutas explícitas ahorran el procesamiento de cabeceras de paquetes para la toma de decisiones de encaminamiento en cada nodo interno haciendo el reenvío de paquetes más ágil a lo largo de la ruta.

Para la creación de rutas en MPLS se definen los LSPs (Label Switched Path), que son caminos específicos de tráfico a través de la red MPLS, mediante la asignación de etiquetas a los paquetes IP y su distribución entre los routers vecinos usando protocolos LDP (Label Distribution Protocols) como RSVP-TE[13] (ReSerVation Protocol – Traffic Engineering). Al inicio del LSP se encuentra el nodo de ingreso LER (Label Edge Router), que es el encargado de clasificar los paquetes en FECs (Forwarding Equivalence Class) y asignar la etiqueta correspondiente a cada paquete que ingresa en la red MPLS según la FEC a la que pertenece; luego el paquete es reenviado a los nodos intermedios de la red denominados LSR (Label Switching Router) que son los encargados de encaminar el paquete a través de la red mediante el intercambio de etiquetas hasta llegar al nodo de egreso LER, que elimina la etiqueta MPLS del paquete y lo conmuta a la red IP.

[13] El protocolo de reserva de recursos (RSVP) es un protocolo genérico de señalización, diseñado originalmente para ser usado por las aplicaciones para solicitar y reservar requerimientos específicos de calidad de servicio (QoS) a través de la interconexión de redes, pero que actualmente es usado para la distribución de etiquetas y el establecimiento de LSPs en redes MPLS.

3 Ingeniería de Tráfico en Redes SDN

Gracias a las ventajas de las redes SDN, los mecanismos de ingeniería de tráfico pueden ser aplicados de mejor manera en comparación con las redes tradicionales ATM, IP, y las basadas en MPLS-TE. Por este motivo y debido a su reciente aparición, los mecanismos de ingeniería de tráfico tienen un alto nivel de relevancia para la evolución de las mismas, destacando 4 enfoques en los que debe proyectarse, como lo indica la siguiente figura.

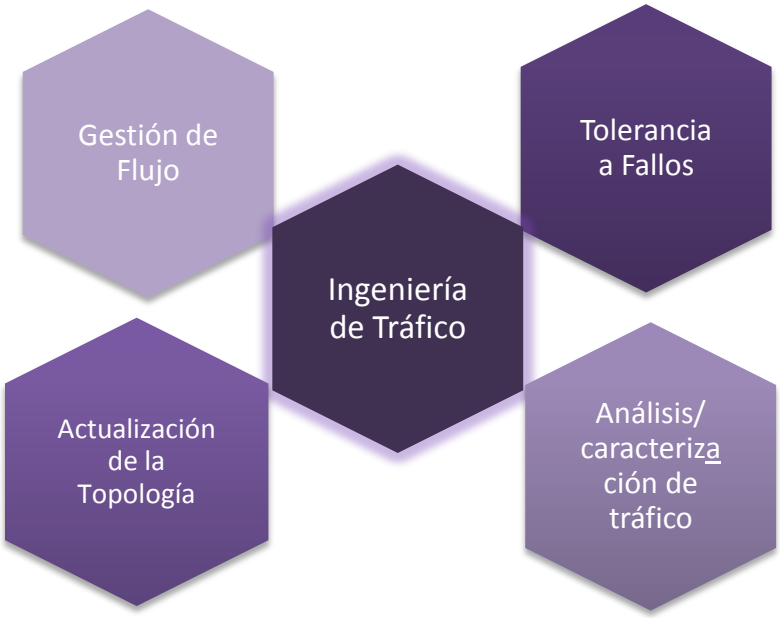


Figura 5. Enfoques de la Ingeniería de tráfico en SDN [12].

Con cada enfoque, se puede añadir especialización en cada uno de los problemas que se mencionarán a continuación:

- **Gestión de Flujo:** Basado en la gestión regular del flujo de datos en redes de este tipo, este enfoque busca encontrar la solución para evitar la sobrecarga de la red debido a la actualización de una gran cantidad de entradas de flujo en los switches involucrados, así como encontrar el equilibrio entre latencia y balanceo de carga.
- **Tolerancia a Fallos:** Busca asegurar la recuperación inmediata de la red cuando ocurre un fallo en alguno de sus nodos, de tal forma que se pueda asegurar la fiabilidad de la red. Sin embargo, se enfrenta a problemas tanto a nivel general en el tiempo para la restauración, como a nivel específico en recursos (memoria limitada y tablas de flujo) de los *switches*, que permitan al controlador central

realizar acciones como: cálculo de nuevas rutas y notificación a los switches afectados sin inconvenientes.

- **Actualización de la topología:** Su objetivo es encontrar la forma de actualizar las políticas de red cambiantes en tiempo real de forma que el controlador SDN permita la configuración dinámica de políticas globales de red y se garantice la aplicación de una u otra política (actual y antigua) para cada flujo, sin permitir la combinación de dichas políticas. Este enfoque se encuentra con inconvenientes como la pérdida o demora de paquetes enviados durante el proceso de actualización de políticas lo cual degrada la calidad del servicio o inutiliza los recursos de red.
- **Análisis y caracterización de tráfico:** Se enfoca en mecanismos que incluyan herramientas para el monitoreo de tráfico de red, software que permita la depuración de errores de programación, estados o flujos de colecciones de datos y análisis de patrones-características, etc. Siendo el análisis de tráfico el prerrequisito más importante, se enfrenta a inconvenientes como: alto consumo de recursos en los switches, incremento de la complejidad en el diseño y un monitoreo sobrecargado a la red. Por ello busca herramientas que permitan baja complejidad, baja sobrecarga y mediciones de tráfico precisas.

El presente trabajo presentará un análisis detallado de dos de los enfoques mencionados, la gestión de flujo y la tolerancia a fallos, explicados en la siguiente sección.

3.1 Requerimientos, soluciones y campos de investigación para un buen desempeño

3.1.1 Gestión de Flujo

Si un switch OpenFlow recibe un flujo del cual no tiene registro en su tabla, pregunta al controlador como gestionarlo, pero esta comunicación y sus respectivas acciones posteriores para la incorporación de los nuevos flujos en las tablas de los switches puede tomar tiempo y provocar picos de retardo en rendimiento.

Por ello se han creado esquemas o mecanismos categorizados en tres ámbitos de acción:

1. ***Balanceo de Carga en Switch***.- Presenta las dos siguientes técnicas:
   - Envío de Flujo ECMP (Equal-Cost Multi-Path) basado en Hash.- Esquema que permite el balanceo de carga a través de la división de los flujos en los diferentes caminos disponibles usando una técnica basada en la generación de hashes.
   - Reglas de Envío de Flujo con máscara de subred invertida (Wildcard Rule Flow Forwarding).- OpenFlow utiliza campos de 32 bits y de 64 bits para la máscara de subred invertida, esta técnica permite reducir la carga en el plano de control. Sin embargo, la visibilidad centralizada de los flujos de datos requiere del controlador la configuración de todos los

flujos críticos en la red, lo que hace que un solo controlador genere un cuello de botella y aumente la latencia.

2. ***Balanceo de Carga en Controlador***.-
   - Distribución lógica del despliegue del controlador: Basado en mecanismos que permitan repartir la carga lógica del controlador, permitiendo que otras entidades o nodos de la red tengan algo de participación en la distribución de regla o la toma de decisiones.
   - Distribución Física del despliegue del controlador: Propone mecanismos que ubican al controlador estratégicamente de modo que los flujos de los que el switch no encuentra entrada, puedan viajar rápidamente al controlador.
   - Controladores multi-hilo: Se proponen controladores con arquitecturas de servidores muti-core que permitan mejorar el rendimiento del controlador en el manejo de solicitudes.
   - Controladores Generalizados: Actualmente existe una variedad de controladores que soportan el protocolo OpenFlow, siendo unos cuantos los más usados para realizar prácticas o utilizrse a nivel de pruebas en campos industriales e investigativos, aquellos que son empujados por el mismo estándar a ser usados, como NOX, SOX u otros. Sin embargo, ciertas o solo una versión del protocolo son soportadas por estos controladores llamados generalizados. Es por ello necesario que se cree un controlador que soporte los cambios que se van dando de versión en versión del protocolo, interconectándolas en un conjunto básico de componentes que puedan integrarse.

3. ***Múltiples tablas de flujo.-***

La versión inicial (v1.0) del protocolo OpenFlow solo sorpota un modelo de tabla de flujo individual construida sobre TCAM, que analizando el contenido de cada flujo y las magnitudes que llega a tener cada tabla, queda muy corta para soportar la interacción de los flujos dentro de la misma tabla usando indexación, además de que relentiza el proceso de búsqueda y matching de las entradas en la tabla.

A partir de la versión 1.1 del protocolo, se añadió el soporte para múltiples tablas de flujo y las acciones que se asocian a cada entrada en la tabla. Como se observa en la figura 6, el switch OpenFlow puede tener una o varias tablas de flujo, que se encuentran secuencialmente numeradas empezando con la tabla 0.

El proceso del flujo de un paquete a través de las tablas funciona de la siguiente manera; cuando ingresa un paquete al switch, este es enviado a la tabla 0 para comprobar si coincide con alguna entrada de la primera tabla, del resultado de esta primera comparación dependerá si el paquete es enviado a alguna de las siguientes tablas. Si el paquete coincide con alguna entrada de flujo en la tabla, la instrucción set de la entrada correspondiente es ejecutada y en el caso de que exista una instrucción Goto adicional, el paquete es reenviado a la tabla que la instrucción indique. Por el contrario, si el paquete no coincide con alguna entrada de flujo, es una pérdida en la tabla y el comportamiento a seguir dependerá de la configuración de la tabla, generalemente en este caso el paquete es enviado al controlador sobre el canal de control usando un mensaje Packet-In, la otra opción sería tirar el paquete [15].

Aparte de que la versión 1.1 de OpenFlow añada la funcionalidad de múltiples tablas de flujo, ahora la versión 1.3 soporta tablas de medición que según la

teoría permiten el funcionamiento de requerimientos de QoS en cada nivel de los flujos según la demanda de tráfico de usuarios o de las aplicaciones. Con estas nuevas capacidades del protocolo se están creando switches que las soporten para que puedan gestionar el tráfico, manejando con eficacia los flujos con el objetivo de mejorar el rendimiento de la red.

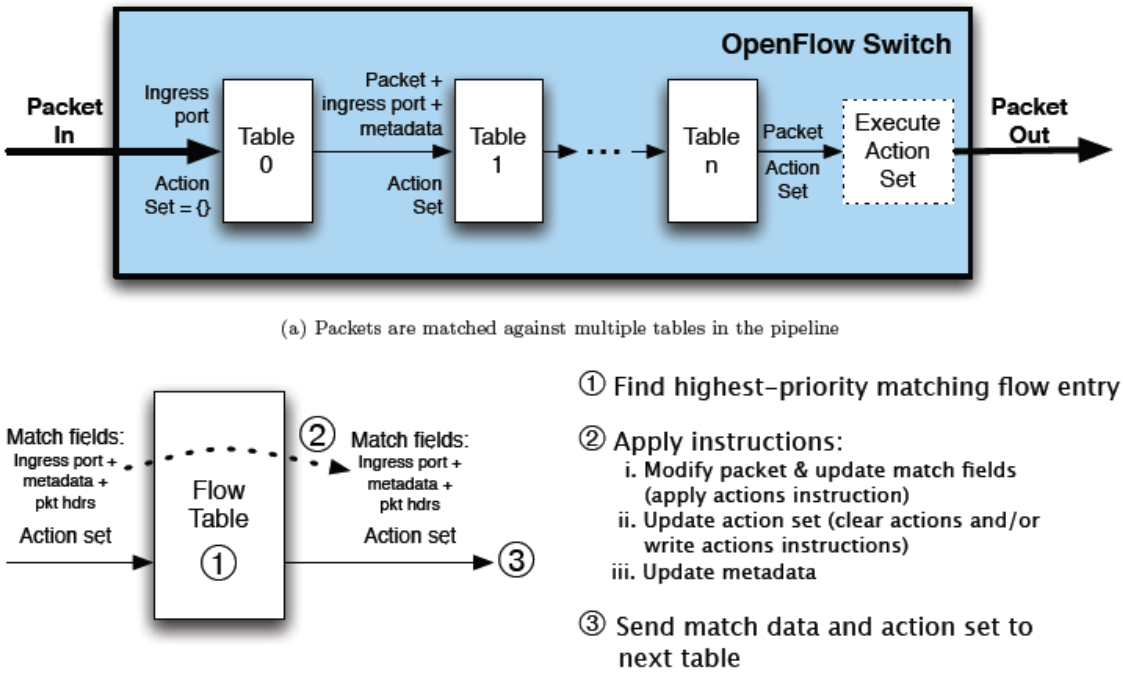


Figura 6. Flujo de un paquete a través de múltiples tablas [15].

A través de la siguiente tabla se presentan las soluciones más relevantes enmarcadas en los tres ámbitos mencionados, que intentan evitar los cuellos de botella en el flujo de datos considerando el equilibrio que debe existir entre latencia y balanceo de carga.

Tabla 1. Soluciones actuales para Gestión de Flujo [12].

Técnica	Soluciones existentes
ECMP basado en Hash	✣ ***Hedera:*** Es un sistema de planificación dinámica y escalable del flujo, cuenta con una vista global del flujo de datos y su encaminamiento, recopila la información del flujo de los switches, calcula rutas que no colapsen para los flujos y dirige a los switches para el re-encaminamiento de los mismos. Resuelve el inconveniente principal del esquema ECMP ya que detecta precisamente los flujos bastante grandes cada 5 segundos y realiza el cálculo del camino más óptimo para ellos. ✣ ***Mahout:*** Gestiona el flujo de tráfico solicitando cada cierto tiempo la detección de precisamente los flujos más grandes, sin embargo a diferencia de Hedera, Mahout monitorea y detecta este tipo de flujo llamado "elephant flows" a través de la capa Shim (*Mahout Shim Layer*) en el sistema operativo de los servidores finales, en vez de monitorearlo en los switches de la red.

	✣ ***MicroTE:*** Es muy parecido a *Mahout* ya que es un esquema de ingeniería de tráfico que detecta los flujos significativos en los hosts finales de forma que los routers *microTE* optimizan su camino.
Reglas de Envío de Flujo con máscara de subred invertida	✣ ***DevoFlow:*** Propone una solución para requerir el menor número de comunicaciones entre el controlador y los switches, basándose en un mecanismo que incluye mascaras de red invertidas en los switches de tal forma que pueden tomar decisiones sobre los flujos de menor tamaño dejándole al controlador únicamente los más significativos. ✣ ***DIFANE:*** Esta solución para evitar los cuellos de botella en el controlador propone una arquitectura de distribución de flujos para redes empresariales usando reglas de máscara de red invertida en los switches y seleccionando switches autoritarios, con la finalidad de enviarle a éste los flujos del plano de datos de los cuales no tenga conocimiento y así evitar el envío de datos al controlador. Es preciso señalar que esta solución carga a los switches principales o anteriormente llamados autoritarios y ellos no proveen balanceo de carga.
Distribución lógica del despliegue del controlador	✣ ***HyperFlow:*** Esta solución ofrece escalabilidad mediante un plano de control distribuido con comunicación cruzada entre controladores basada en eventos que usa el método publish/suscriber para almacenamiento persistente mediante el sistema de archivos WheelFS[14]. HyperFlow delega la toma de decisiones a los controladores individuales minimizando el tiempo de respuesta del plano de control. Se debe mencionar que el método publish/suscriber demanda una sobrecarga debido a la gestión de suscripción de los switches a su controlador más cercano. ✣ ***DIFANE:*** Propone la distribución de reglas a los switches llamados de autoridad desde el controlador. Estos switches de autoridad sirven como desvío para los paquetes logrando así un manejo de los paquetes en el plano de datos, es decir, los switches comunes manipulan los paquetes normalmente y en el momento que se topan con algún paquete que no coincide con sus reglas almacenadas, redirigen el paquete al switch de autoridad que corresponde. Luego de esto, el switch de autoridad responde enviando al swicth de entrada de la red la información necesaria a almacenar en su caché para conservar la regla o reglas necesarias. Las reglas enviadas a un switch para el desvío de paquetes están expresadas en TCAM[15].
Distribución Física del despliegue del	✣ ***Onix:*** Se encarga de gestionar la lógica de control de acceso de programación a la red mediante una plataforma de uno o más servidores físicos distribuidos. Es escalable y fiable, ya que permite reproducir y distribuir los datos de las tablas NIB

[14] WheelFS es un sistema de archivos de área amplia para sistemas distribuidos, que está basado en FUSE (Sistema de Archivos en el espacio de usuarios) y ofrece flexibilidad de almacenamiento para este tipo de aplicaciones.

[15] TCAM (Ternary Content Addressable Memory)

controlador	que contienen el estado de la red, lo que proporciona una vista de la red física y de las aplicaciones de control en todo momento para manejar el estado de los elementos de la red. Sin embargo, este método al igual que HyperFlow demanda una sobrecarga debido a la gestión de suscripción. • ***BalanceFlow:*** Es similar al concepto de controladores distribuidos de Onix, pero está más centrado en el balanceo de carga en el controlador. Propone la existencia de un super controlador que es el responsable de realizar el balanceo de carga de todos los demás controladores, esto sucede cuando se detecta una anomalía en los niveles medios de solicitudes de flujo de uno de los controladores. Aquí existe la probabilidad de sobrecarga en el plano de control debido a que un super controlador maneje el balanceo de carga.
Controladores multi-hilo * Versión OF: 1.0.0	• ***Maestro:*** Usa 7 hilos de procesos con una capacidad de procesamiento de 8 núcleos en cada uno de sus dos procesadores Quad-Core AMD Opteron. Puede gestionar 0.63 millones de rps (respuestas por segundo) con un retraso de hasta 76 ms. 16 GB en memoria. Lenguaje: Java. • ***Bacon:*** Usa 12 hilos de procesos con una capacidad de procesamiento de 16 núcleos en cada uno de sus 2 procesadores Intel Xeon E5-2670. Puede gestionar 12.8 millones de rps con un retraso de hasta 0.02 ms. 60.5 GB de memoria. Lenguaje: Java. • ***NOX-MT:*** Usa 8 hilos de procesos con una capacidad de procesamiento de 8 núcleos en cada uno de sus 2 procesadores de 2GHz. Puede gestionar 1.6 millones de rps con un retraso de hasta 2 ms. Lenguaje: C++.
Controladores Generalizados * Versión OF: 1.3+	• ***SOX:*** Usa 4 hilos de procesos con una capacidad de procesamiento de 4 núcleos en su procesador de 2.4 GHz. Puede gestionar 0.9 millones de peticiones por servidor por segundo, 3.4 o más millones de peticiones por segundo con un cluster de 4 servidores.

* Se debe tomar en cuenta que los datos proporcionados son de referencia de los desarrolladores o fabricantes de estos controladores y que los valores dados no fueron medidos en un ambiente en común por lo que no pueden ser comparados literalmente.

La tabla arriba presentada pretende mostrar el desarrollo que ha habido hasta ahora en los controladores SDN y que tienen resultados directos que pueden aportar o no a la mejora de la gestión de flujo de la red.

3.1.2 Tolerancia a Fallos

Con la finalidad de asegurar fiabilidad de la red SDN frente a fallos ocurridos en su infraestructura (controladores, switches y enlaces) es necesario la aplicación de mecanismos que permitan la detección y recuperación inmediata ante incidentes.

Sin embargo asegurar que el tiempo que le tome a la red SDN recuperarse sea mínimo, es ya un desafío, ya que su arquitectura centralizada y la toma de acciones de recuperación, como el cálculo de nuevas rutas y el envío de notificaciones de acción inmediata a todos los switches afectados pueden aumentar este tiempo.

Para permitir la aplicación de dichos mecanismos, éstos se han dividido de tal forma que puedan asegurar fiabilidad tanto en el plano de control como en el plano de datos.

3.1.2.1 Plano de Datos

En el plano de datos se han establecido dos mecanismos de tolerancia a fallos, uno con estrategias reactivas y otro proactivas:

- ***Restauración:*** En este mecanismo se definen rutas de recuperación pre calculadas o dinámicamente asignadas, sin embargo los recursos no son reservados hasta que el fallo ocurre y es necesario señalización adicional para el establecimiento de dicha ruta.
- ***Protección:*** La ruta en este mecanismo es pre calculada y reservada antes de que el fallo ocurra y no se requiere de señalización adicional para su habilitación.

Sin embargo la aplicación de una u otra alternativa en redes SDN, cuenta con ventajas y desventajas detalladas en la tabla 1 según [12].

Tabla 2. Comparación de Mecanismos de Tolerancia a Fallos en el Plano de Datos

Mecanismo	Ventajas	Desventajas
Restauración	---	Requiere de operaciones de eliminación, modificación y de otras adicionales entre el controlador y los *switches* durante el fallo, lo que aumenta el tiempo de restauración.
Protección	Debido a que las rutas de recuperación son implementadas junto a las de las rutas activas en la tabla de flujos de los *switches*, su recuperación ante el fallo tiene un tiempo inferior ya que no es necesaria una interacción entre *switches* y controlador, con lo cual además ancho de banda y latencia disminuyen durante el fallo.	La preinstalación de la ruta de protección puede ligeramente incrementar el tiempo de configuración de las tablas de flujos ya que dicha información adicional debe ser enviada a los *switches*.
		Consume altos recursos de memoria en la instalación de las rutas de protección.
		Si las políticas de red cambian, las tablas de flujo deben ser modificadas, lo cual produce una comunicación adicional entre el controlador y *switches*

		que pueden causar una sobrecarga en la comunicación establecida.

Sin embargo, no solo el mecanismo seleccionado puede agregar demora en la restauración sino que también ésta puede ser causada por el protocolo *OpenFlow,* debido a su especificación en donde indica que los switches afectados no deben eliminar las entradas de flujo relacionadas a los enlaces caídos hasta que no se haya cumplido el tiempo de sus temporizadores.

Dos alternativas de solución para evitar este inconveniente, han aparecido:

- Las rutas predefinidas para recuperación son instaladas usando funcionalidades de Tablas de Grupo (*Group Table*) de acuerdo a la especificación del protocolo versión 1.1.
- Esquema de Protección de segmento basado en *OpenFlow* (OSP), el cual provee un mecanismo basado en priorización de rutas, dejando con más alta prioridad a las rutas tradicionales y valores menores a las rutas de protección. Este esquema permite la eliminación inmediata de entradas de las rutas caídas sin tener que esperar al temporizador.

Debido a la necesidad de informar a los *switches* que enlaces se encuentran caídos, Maulik Desai y Thyagarajan Nandagopal [16] han desarrollado un algoritmo que permite intercambiar simples mensajes de notificación de fallos con un mínimo de inteligencia en los switches que evita el inundamiento con mensajes innecesarios en la red, únicamente notificando a los switches involucrados y permite el reconocimiento del evento más temprano de lo que el controlador lo detecta y envía la actualización.

Por lo tanto una comunicación anti-fallos oportuna debe cumplir con los siguientes requisitos:

- Comunicación de red mínima necesaria.
- Nula o mínima interacción con controlador SDN.
- Un mínimo de inteligencia en los switches.

Es por ello que se considera un campo de investigación ya que no se encuentra el mecanismo que se apegue idealmente a estos requisitos.

3.1.2.2 Plano de Control

Debido al control centralizado de las redes SDN, el único punto de fallo que dejaría sin "inteligencia" a estas redes es el controlador, por lo tanto se deben tomar medidas que permitan recuperar el control a pesar de una falla en el dispositivo lógico principal.

Con lo cual la primera solución nos lleva a pensar en la necesidad de tener también controladores de respaldo o protección ante fallo, sin embargo esto presenta algunos desafíos detallados en la siguiente tabla según [12].

Tabla 3. Problemas y soluciones propuestas para permitir tolerancia a fallos en el Plano de Control

Problema	Soluciones Propuestas
Protocolos de Coordinación.- A pesar de que mediante las especificaciones del protocolo se permite la configuración de controladores de respaldo, no se provee mecanismos para la coordinación, coherente y consistente actualización entre ellos.	• **CPRecovery.-** Mecanismo de respaldo primario de controladores que actúa sobre el sistema operativo de red y permite que el *switch* se entere de un fallo en el controlador después del envío de un mensaje que no es respondido por el controlador, lo que anuncia su fallo y habilita al controlador de respaldo como el principal. Sin embargo experimentos realizados mostraron que de acuerdo al grado de replicación (más dispositivos de respaldo) el tiempo de respuesta se ve incrementado, detectaron un incremento del 75% en la velocidad de respuesta cuando no se tiene controlador de respaldo (8ms) frente a cuando se posee uno (14ms).
Respaldo de despliegue de controladores.- Encontrar la cantidad de dispositivos y el equilibrio entre fiabilidad y latencias en la distribución de los dispositivos de respaldo.	• Gracias a la investigación realizada sobre Internet2 OS3E, un *testbed* de múltiples controladores para SDN, determinó que el número mínimo de controladores de respaldo es mínimo de 3 y máximo de 11, sin embargo el mínimo de dispositivos optimiza el promedio y el peor caso del índice de latencia, mientras disminuye la fiabilidad, caso contrario con menos dispositivos de respaldo, se incrementa la latencia pero se optimiza la fiabilidad del sistema. Con lo cual se debe encontrar un punto de equilibrio entre ambas*. • La opción de clústeres de controladores en modo equitativo utilizando SOX, un controlador para redes SDN, permite la adaptación dinámica del número de controladores de acuerdo a la demanda de tráfico existente. Con lo que se intenta solucionar el problema de equilibrio entre latencia y fiabilidad de los sistemas de red SDN. • DSOX, este enfoque presenta al controlador SOX pero en forma distribuida en el sistema propuesto anteriormente para ser utilizado en redes MAN o sistemas autónomos (AS).

* Sin embargo, aún se considera un campo de investigación la definición del número idóneo de controladores de respaldo y su localización (controlador principal y de respaldo), de tal forma que se permita tener fiabilidad y baja latencia en la comunicación.

4 Caso de Estudio OSHI

El desarrollo de OSHI nace de la necesidad existente en redes SDN por brindar una infraestructura más sofisticada que permita a los ISPs desplegar servicios como lo hacen en las redes IP/MPLS actuales, mediante la integración de protocolos de encaminamiento IP con la conmutación en capa 2 hasta ahora desarrollada en esta tecnología. OSHI está siendo desarrollado por el grupo de networking de la Universidad "Tor Vergata" de Roma y forma parte del proyecto DREAMER[16], uno de los beneficiarios de la iniciativa de investigación del GÉANT Open Call. En este marco, uno de los objetivos de OSHI con la apertura del enfoque de SDN, es facilitar el desarrollo de nuevos servicios y fomentar la innovación en las telecomunicaciones, más allá de realizar mejoras en el rendimiento que las redes IP/MPLS ya poseen [20].

4.1 Arquitectura Híbrida

La idea de concebir una arquitectura híbrida en redes definidas por software viene de la mano de la simultaneidad con la que han funcionado los protocolos IP y MPLS hasta hoy sobre las redes tradicionales, buscando la determinación de caminos cada vez más óptimos hacia los cuales redirigir el tráfico de la red. Haciendo una semejanza a estos caminos, llamados LSPs (tratado en el apartado 2.3.2) en MPLS; en redes SDN pueden ser creados túneles similares pero de distintos tipos, dando la posibilidad a nodos SDN de coordinar el tráfico basándose en diferentes campos como direcciones MAC o IP, y etiquetas VLAN o MPLS. En OSHI han querido definir a estos caminos como SBPs o Caminos basados en SDN. "Un SBP (SDN Based Paths) es un circuito virtual configurado usando tecnología SDN para reenviar un flujo de paquetes específico entre dos puntos finales a través de un conjunto de nodos que entienden SDN" (Salsano et al., 2015, p. 3).

Es así que podríamos definir a las redes híbridas IP/SDN como un conjunto de nodos capaces de soportar mecanismos de coexistencia de tráfico IP y tráfico SBP, que busca ofrecer una serie de servicios que permitan la clasificación del tráfico al ingreso de la red y el funcionamiento de mecanismos de túneles. De esta definición, nace la necesidad de crear estos nodos o equipos híbridos que analizaremos a continuación.

4.1.1 Nodo OSHI

Un nodo OSHI (Open Source Hybrid IP/SDN) está formado por tres componentes que hacen posible la coexistencia tanto de tráfico IP como SBP; un switch capaz de entender OpenFlow (OFCS), una máquina de reenvío IP (IP FE) y un demonio de

[16] El proyecto Distributed REsilient sdn Architecture Meeting carrier grade Requirements fue fundado por la EU y tiene como objetivo la investigación de redes basadas en OpenFlow/SDN que permitan brindar las mismas funcionalidades que una red IP/MPLS, mediante la creación de prototipos en el plano de control SDN, centrándose principalmente en la capacidad de recuperación y la gestión de fallos. Para más información, tomar como referencia este enlace: http://netgroup.uniroma2.it/twiki/bin/view/DREAMER/WebHome#FactsSheet.

encaminamiento IP. El OFCS es una implementación de Open vSwicth, la máquina de reenvío IP es el kernel de networking de Linux y el demonio de encaminamiento usado es Quagga[17], específicamente un demonio OSPF.

Como se observa en la figura 6, la IP FE interna está conectada al OFCS por medio de los puertos virtuales que el Open vSwitch posee ya definidos, mientras que el OFCS recibe la información de entrada al nodo por medio de las interfaces físicas externas conectadas a él. Es así, que el OFCS clasifica el tráfico diferenciando que paquetes corresponden a IP y cuáles son SBP, los paquetes IP son conmutados de los puertos físicos a los virtuales internos enviándolos a la IP FE para que sean procesados y puedan ser controlados por Quagga, mientras que los SBP son procesados por el Open vSwitch directamente. Esto evita que los paquetes IP que no son soportados por un switch OpenFlow tengan que ser traducidos a reglas SDN para que tengan cabida en las tablas del OFCS. Además, cada puerto físico externo de la red IP/SDN está conectado a uno de los puertos virtuales internos.

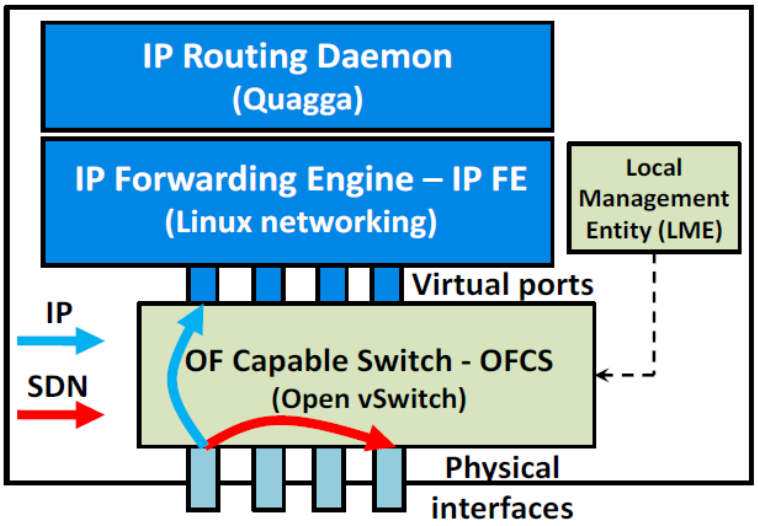


Figura 7. Arquitectura de nodo OSHI (plano de control) [21].

La LME (Local Management Entity), Entidad de Gestión Local, es un subcomponente del nodo que pertenece al OFCS y se encarga de establecer la configuración inicial de las tablas del OFCS para que las conexiones a las interfaces físicas y a los puertos virtuales sean realizadas y el OFCS pueda comunicarse con el controlador SDN normalmente.

Existen dos tipos de nodos OSHI, correspondientes a los nodos de acceso y nodos de core de la terminología en redes MPLS. Los nodos core o núcleo de la red IP/SDN

[17] Quagga es una suite de software libre que mediante la implementación de protocolos de encaminamiento, permite tratar a plataformas Unix (FreeBSD, Linux, Solaris y NetBSD) como si fueran un enrutador.

(COSHI) poseen la misma arquitectura que se observa en la figura 6, mientras que los nodos de acceso a la red (AOSHI) se diferencian debido a funcionalidades adicionales que deben soportar para la clasificación del ingreso de tráfico a la red [19], lo que agrega ciertos elementos a la arquitectura básica de estos nodos que serán detallados en el apartado 4.1.2.2.

4.1.2 Servicios

Dos servicios básicos pueden ser ofrecidos por estas redes híbridas IP/SDN, que son desplegados entre los puntos finales de los routers PE correspondientes a nodos AOSHI. Estos servicios son las redes arrendadas virtuales IP (VLL IP) y los Pseudo-wires de capa 2 (PW). Los puntos finales de la red son interconectados mediante el uso de etiquetas VLAN o etiquetas MPLS formando un SBP para el envío de tráfico entre ellos como se observa en la figura 7.

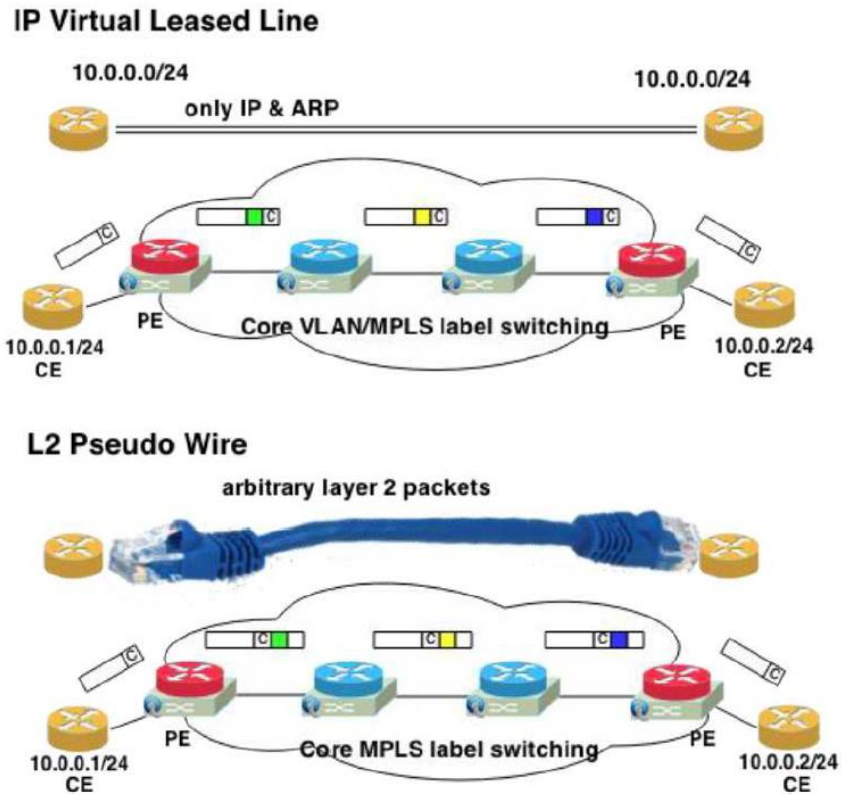


Figura 8. Servicios VLL IP y Pseudo-wire [21].

La diferencia principal entre estos dos servicios es la clase de paquetes que pueden ser procesados para el establecimiento de la conexión, debido a que se ofrecen en capas diferentes del modelo OSI (capa 2 y capa 3). En una VLL IP los nodos del núcleo de la red intercambian solo paquetes IP y ARP como si se encontraran en una misma LAN Ethernet y pueden usar etiquetas VLAN o etiquetas MPLS en el envío de paquetes, mientras que el servicio Pseudo-wire según su especificación [18] puede intercambiar

paquetes con Ethertype arbitrarios de capa 2, aunque en este proyecto el servicio ha sido implementado solo con etiquetas MPLS como se observa en la figura 6.

Con el objetivo de brindar estos servicios, la arquitectura de un nodo PE-OSHI se extiende para soportar mecanismos de coexistencia de tráfico IP y tráfico SBP, como se detalla en la siguiente subsección.

4.1.2.1 Enfoques de Funcionamiento OSHI

Existen dos enfoques, desde los cuales se puede crear un SBP sobre un grupo de nodos OSHI interconectados: un enfoque basado en VLAN y otro basado en MPLS [21]. Desde estos dos enfoques se soportan mecanismos que permitan la coexistencia de tráfico IP y tráfico SBP en estos nodos.

Para el enfoque basado en VLAN, se han encontrado dos mecanismos de coexistencia:

- ***Coexistencia Etiquetada***.- Donde el tráfico IP y el tráfico SBP son transportados con diferentes etiquetas VLAN y cada una específica. Además, el tráfico etiquetado a la entrada con una etiqueta VLAN específica se re-direcciona a un puerto virtual hacia la máquina de reenvío IP (IP FE), mientras el tráfico etiquetado con otras etiquetas VLAN es conmutado a un puerto físico, según la configuración del SBP.
- ***Coexistencia Sin Etiqueta***.- Donde el tráfico IP es transportado sin etiqueta y el tráfico SBP usa una etiqueta VLAN para viajar en la red. El tráfico que ingresa y no se encuentra etiquetado, se envía a la IP FE por un puerto virtual mientras que el tráfico que posee una etiqueta VLAN es conmutado a un puerto físico de salida de acuerdo a la configuración del SBP.

El servicio VLL IP fue implementado usando un enfoque basado en VLAN en el controlador. Usan el script vll_pusher en python para la creación de los SBPs, que se vale de un REST API de la topología implementado en el controlador para devolver la ruta que interconecta los puntos finales del VLL. El script asigna etiquetas VLAN y luego asigna las reglas para el reenvío de paquetes y la conmutación de etiquetas VLAN.

Para el enfoque basado en MPLS se usa el campo Ethertype del protocolo de capa 2 de acuerdo con uno de los mecanismos que puede usarse en el modelo IP/MPLS. Allí, tráfico regular IP es entregado con Ethertype IP (0x800) y tráfico SBP es entregado con Ethertype MPLS (0x8848). En este enfoque es posible el soporte de la coexistencia de los dos tipos de tráfico gracias a la funcionalidad de tablas múltiples del protocolo OpenFlow.

Como se observa en la figura 8, la tabla 0 se usa para el tráfico regular, ya sea IP, ARP, LLDP, etc., mientras que la tabla 1 se usa para los SBPs. La tabla 0 está formada por:

i. Dos reglas para reenviar el tráfico de Ethertype tipo MPLS Unicast (0x8847) y MPLS Multicast (0x8848) hacia la tabla 1;
ii. Un conjunto de reglas que permiten puentear las interfaces físicas con las virtuales y viceversa;
iii. Dos reglas para el reenvío de tráfico LLDP (Link Layer Discovery Protocol)[18] y BLDP al controlador.

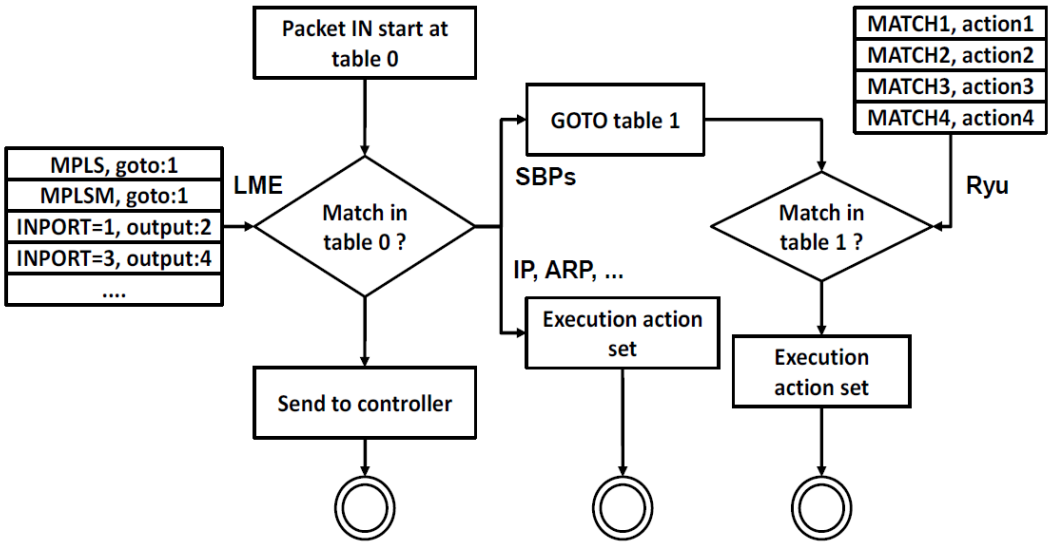


Figura 9. Procesamiento de paquetes dentro de tablas de Flujo de un OFCS [21].

Mientras que la tabla 1 contiene un conjunto de reglas que permiten el reenvío de paquetes de los SBPs de acuerdo con el servicio que se encuentre asociado, ya sea VLL IP o PW. Los diferentes niveles de prioridad de cada entrada en la tabla 0 permiten la coexistencia de los tipos de tráfico. El servicio VLL IP se vale de las dos reglas del Ethertype MPLS, unicast y multicast; mientras que el servicio PW se vale solo de la regla del Ethertype MPLS unicast.

4.1.2.2 Arquitectura Extendida de PE-OSHI

Para brindar el servicio de túnel Pseudowire es necesario introducir una nueva entidad en la arquitectura de los nodos de borde PE. Tal como se observa en la figura 8, esta nueva entidad es el Access Encapsulator (ACE) que está implementado mediante una instancia separada de Open vSwitch que se encuentra corriendo en un namespace

[18] El protocolo de Descubrimiento de Enlace de Datos es un protocolo de descubrimiento de vecindario (NDP), que permite transmitir y recibir información de otros dispositivos adyacentes de la red, es decir, este es un protocolo de un solo salto. Opera en la capa 2 del modelo OSI.

de red de Linux separado y que permite lidiar con el manejo de túneles GRE[19], como se explica en [21]. Esta cabecera es necesaria ya que el protocolo OpenFlow y la mayoría de los OFCS no soportan encapsulación EoMPLS y su proceso inverso por defecto.

EL OFCS solo maneja la asignación y eliminación de etiquetas MPLS y el ACE se encarga de la encapsulación GRE. Para la interconexión de los puertos del OFCS y el ACE han utilizado el concepto de Virtual Ethernet Pair[20] que ofrece Linux en su kernel y que se puede observar en los puertos pintados de color amarillo en la figura. Esta interconexión permite la comunicación entre ellos para el envío de los datos en túneles GRE. En primer lugar, se usan dos puertos del ACE por cada PW creado, un puerto local y uno remoto. El puerto local es aquel que se conecta con el CE que se encuentra conectado localmente al nodo PE, mientras que el puerto remoto es en realidad un puerto GRE que se encuentra conectado al lado remoto del PW.

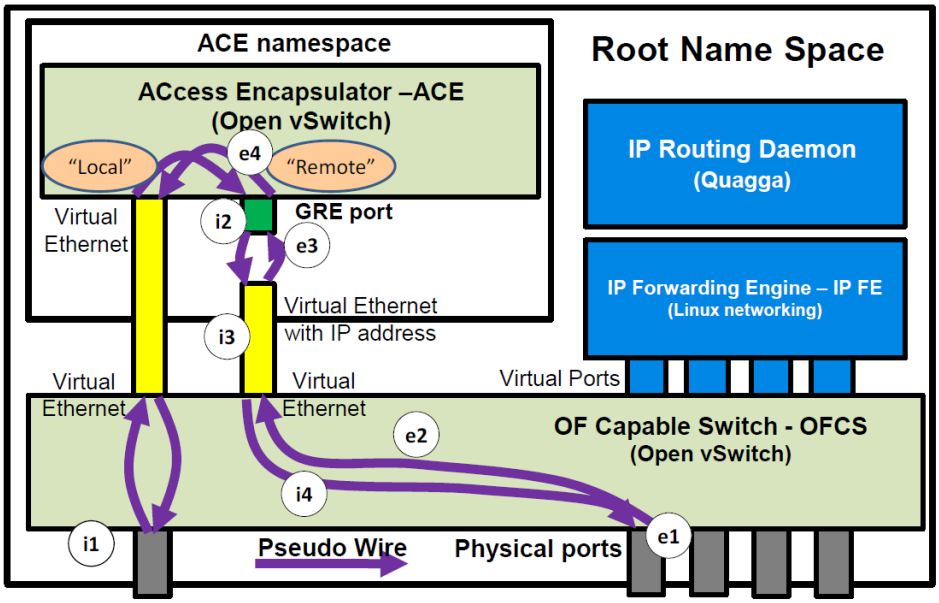


Figura 10. Arquitectura detallada de nodo PE-OSHI [21].

Es así, que el ACE recibe paquetes de capa 2 en los puertos locales y los envía tunelizados con GRE por el puerto remoto hacia el PW, y en el sentido contario para enviarlos hacia el CE localmente conectado [21].

[19] Generic Routing Encapsulation, es un protocolo que permite el establecimiento de túneles a través de Internet, su definición se encuentra en el RFC 1701 y RFC 1702 de la IETF. Lo que lo hace atractivo es que tiene la capacidad de transportar hasta 20 protocolos de nivel de red (capa 3 de OSI) distintos.

[20] Pares Ethernet Virtuales, son interfaces utilizadas por Linux para ser asignadas a los namespaces que se pueden crear sobre el SO, para poder conectarlos hacia afuera del equipo mediante el namespace global donde existen interfaces físicas, estas interfaces siempre vienen en pares.

4.2 Pruebas sobre Escenarios de Servicios

Para realizar pruebas sobre los servicios descritos en el apartado 4.1.2 para una red definida por software, el grupo de investigación propone dos ejemplos que vienen pre-configurados en la máquina virtual OSHI-VM_3 que se descargó desde [23]. El primer ejemplo corresponde a la emulación del despliegue de los servicios de líneas arrendadas virtuales o VLL y de pseudowires.

En este apartado se pretende realizar pruebas en el controlador y pruebas de conectividad para comprobar la comunicación entre los diferentes nodos del despliegue de los dos servicios, además de comprobar los elementos que hacen parte de la arquitectura de cada uno de los nodos. Para ello se utilizará el analizador de protocolos Wireshark, así como las herramientas ping, tcpdump, iperf, y se realizará un análisis de las tablas de flujo en el controlador antes y después de la creación de las VPNs.

4.2.1 Componentes y Versiones de los Escenarios

La máquina virtual a utilizar viene preinstalada y pre-configurada con las siguientes características, que han sido suficientes para realizar las pruebas:

Tabla 4. Características de Máquina Virtual

Sistema Operativo	Ubuntu 13.04
Arquitectura	32 bits
RAM	1.5 GB

La versión de Open vSwitch utilizada en la máquina virtual es la 2.3.90 que implementa la versión 1.1 del protocolo OpenFlow en el OFCS, como se comprueba en la siguiente figura:

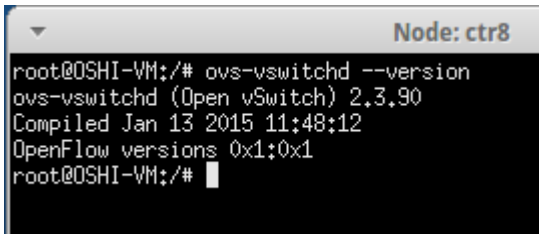


Figura 11. Versión de Open vSwitch.

Esta versión del protocolo fue usada por el grupo de investigación para la implementación de los escenarios de servicios debido a que el OFCS requiere la funcionalidad de uso de múltiples tablas de flujo que es necesaria para el enfoque de etiquetas VLAN utilizado en el servicio VLL IP, que era solo soportado por la versión 1.1 del protocolo en el momento de la implementación de los escenarios,

adicionalmente del soporte para MPLS que fue implantado a partir de la misma versión. En cuanto al controlador actualmente usan ryu para esta implementación (soporta OpenFlow 1.1), aunque para la experimentación de ingeniería de tráfico realizada por el mismo grupo en otra fase del proyecto, utilizaban inicialmente el controlador Floodlight pero este no soportaba la versión 1.1 del protocolo por lo que fue cambiado por ryu.

4.2.2 Servicios VLL y PW

Se realizan pruebas de funcionamiento de los servicios VLL y PW sobre la red OSHI del primer ejemplo propuesto por el grupo de investigación cuyo mapa de red se muestra en la figura 11.

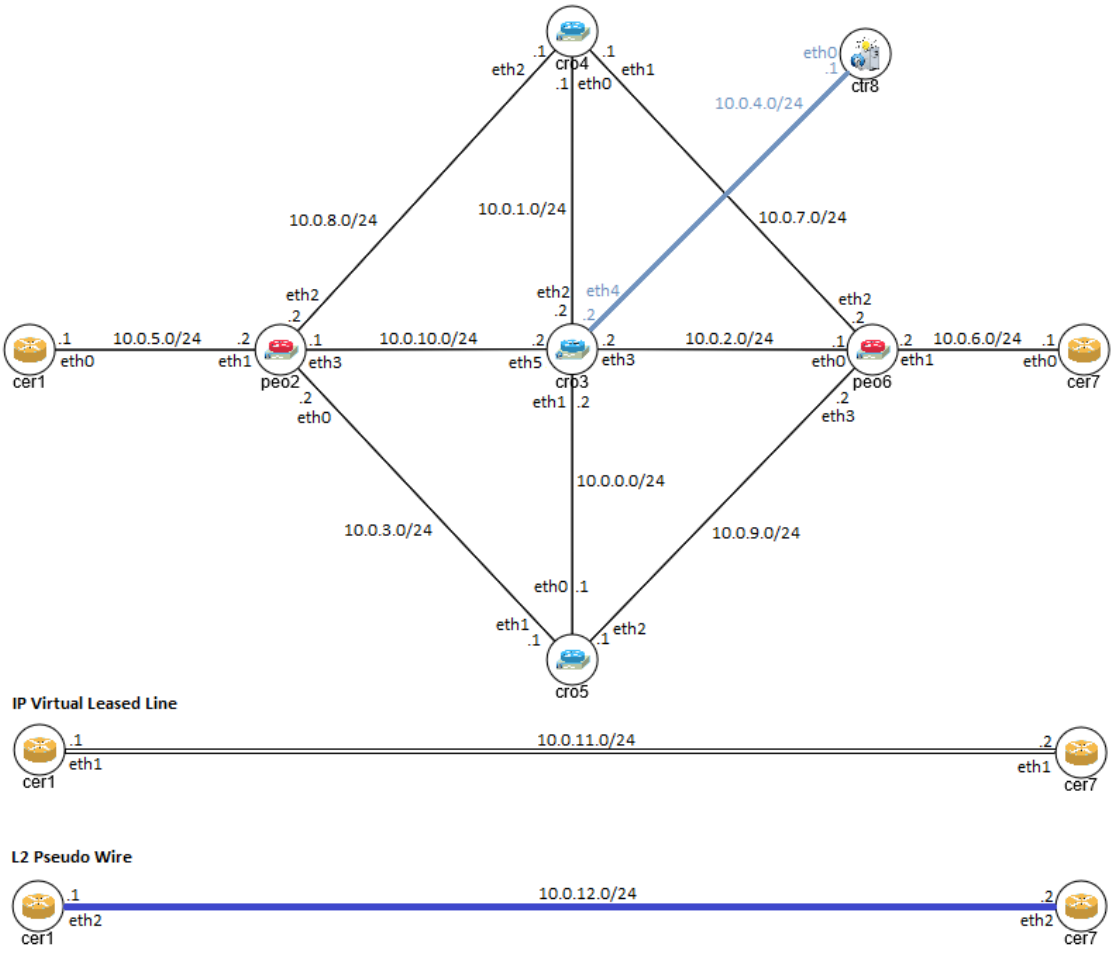


Figura 12. Mapa de Red de Ejemplo - Despliegue de VLL y PW.

Para ello se realizan los siguientes pasos:

1. Se corre el script mininet_deployer.py ubicado en el directorio /home/user/workspace/Dreamer-Mininet-Extensions, como se muestra a continuación:

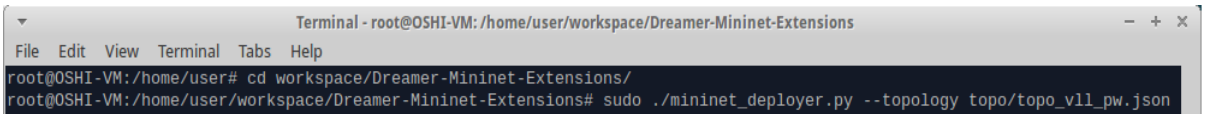


Figura 13. Ejecución del script de despliegue de la topología OSHI.

2. Se espera a que termine la ejecución del script, hasta que se obtiene el mensaje que confirma que los nodos se encuentran corriendo y aparezca el CLI de Mininet.

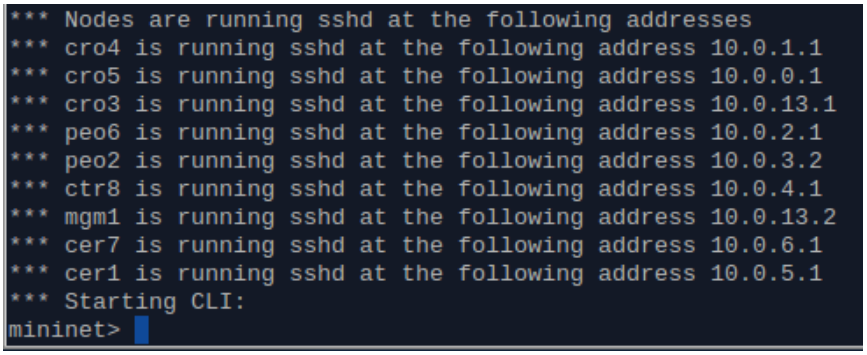


Figura 14. Resultados de Script de Despliegue de Topología.

3. Se realiza una prueba de ping desde la consola de mininet, para comprobar que existe conectividad de extremo a extremo en los dos sentidos entre los routers clientes cer1 y cer7. Así comprobamos que la topología ha sido desplegada correctamente.

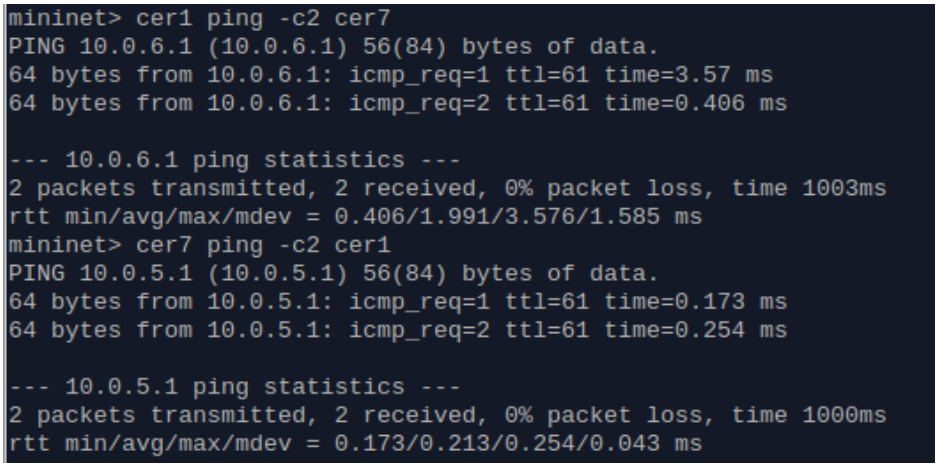


Figura 15. Prueba de ping entre cer1 y cer7 – interfaces para Servicio IP.

4. Se accede a dos terminales del nodo ctr8 usando el comando "xterm ctr8 ctr8" desde la consola de mininet. En el primer terminal se inicia el controlador RYU. Tener en cuenta que para que la ejecución del controlador funcione, esta debe ser realizada desde el directorio /home/workspace/ryu/ryu/app tal como se muestra en la siguiente figura.

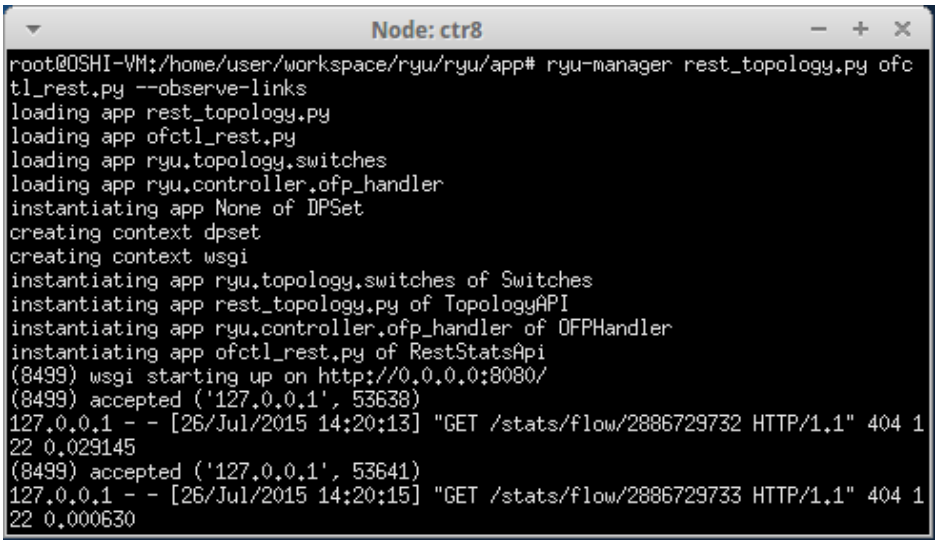


Figura 16. Iniciación de controlador ryu.

5. Una vez iniciado el controlador y sus funciones del REST API, se consultan las tablas de flujo de los switches en el controlador, para comprobar si existen configuraciones iniciales que realiza la LME en las tablas como se comenta en el apartado 4.1.1 correspondiente a la arquitectura del nodo OSHI.
   Para obtener la tabla de flujos de los switches se utiliza la función de REST API (anexo 1): curl –X GET http://localhost:8080/stats/flow/<dpid>, donde dpid se refiere al datapath id de cada switch.
   a. Para obtener los dpid de los switches se utiliza la siguiente función del REST API: curl –X GET http://localhost:8080/stats/switches, y obtenemos los datos que se muestran en la figura 15. Se observa que los dpids obtenidos están formados dígitos decimales.

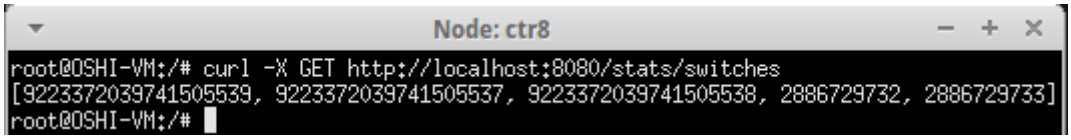


Figura 17. Consulta de dpid de los switches.

   b. Para comprobar los dpids de los switches, se obtiene ahora la configuración de la topología usando una de las funciones de rest_topology (anexo 1) del REST API del controlador: curl –X GET http://localhost:8080/v1.0/topology/switches. Si analizamos la respuesta obtenida se pueden observar los identificadores (dpid) de los switches en notación hexadecimal, además de cada uno de los puertos asignados a los switches con su dirección MAC correspondiente. Para efectos del análisis, se muestra a continuación solo la configuración del peo6. (Para toda la configuración, ver anexo 2)

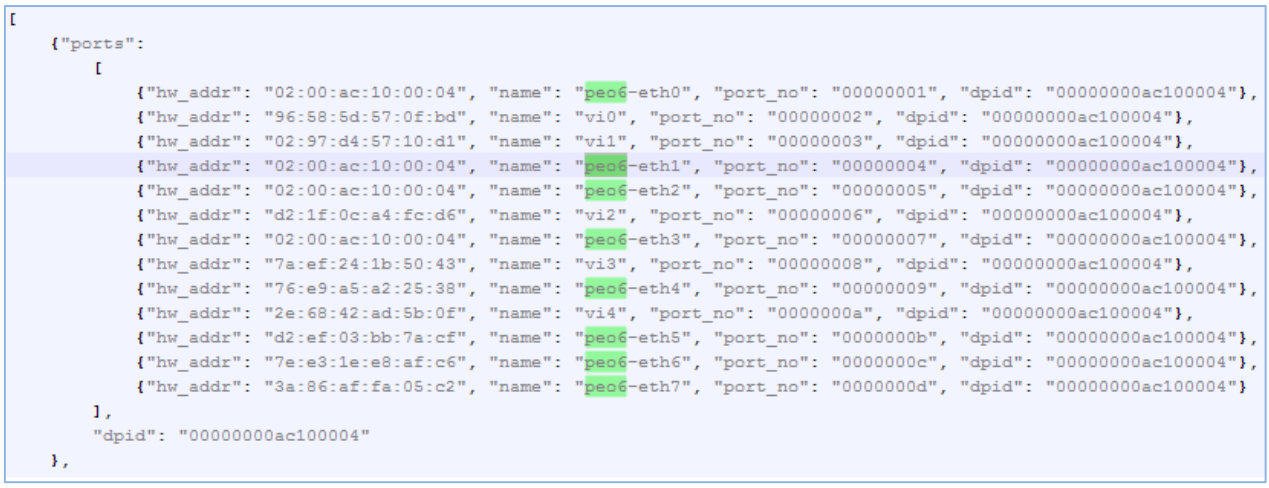


Figura 18. Configuración de topología de Switch peo6.

c. Según la información obtenida del grupo de investigación de OSHI, ellos han realizado la asignación de los dpids como se muestra en la tabla 5, además se encontró que en el script rest_topology.py del proyecto se define que el datapath id está compuesto por 16 dígitos hexadecimales.

Tabla 5. Asignación de dígitos del datapath.

Elemento	Datapath id <dpid>
Core-OSHI	922337203974150553X
PE-OSHI	288672973X

Tomando en cuenta esta información en conjunto con la obtenida en la especificación de OpenFlow [15], que dice que un datapath id está conformado por 64 bits donde los 48 menos significativos se usan para la dirección MAC del switch mientras los 16 más significativos son definidos por el implementador del protocolo; se obtiene la asociación de los dpids con cada nodo correspondiente como se observa en la tabla 6. Esto se puede comprobar convirtiendo el dpid decimal a hexadecimal o viceversa.

Tabla 6. Asociación de datapath id a Elementos de la Red OSHI.

Nodo	dpid en topología (hexadecimal)	dpid en tablas de flujo (decimal)	Dirección MAC
peo2	00000000ac100005	2886729733	02:00:ac:10:00:05
peo6	00000000ac100004	2886729732	02:00:ac:10:00:04
cro3	80000000ac100003	9223372039741505539	02:00:ac:10:00:03
cro4	80000000ac100001	9223372039741505537	02:00:ac:10:00:01
cro5	80000000ac100002	9223372039741505538	02:00:ac:10:00:02

d. Si se consultan las tablas de flujo de los peo2 (2886729733) y peo6 (2886729732) apenas inicia el controlador, no obtenemos ninguna respuesta debido a que las tablas se encuentran vacías al iniciar. Segundos después, se realiza una segunda consulta y se obtienen las tablas de las dos pes, copiándolas en archivos para analizar los resultados posteriormente.

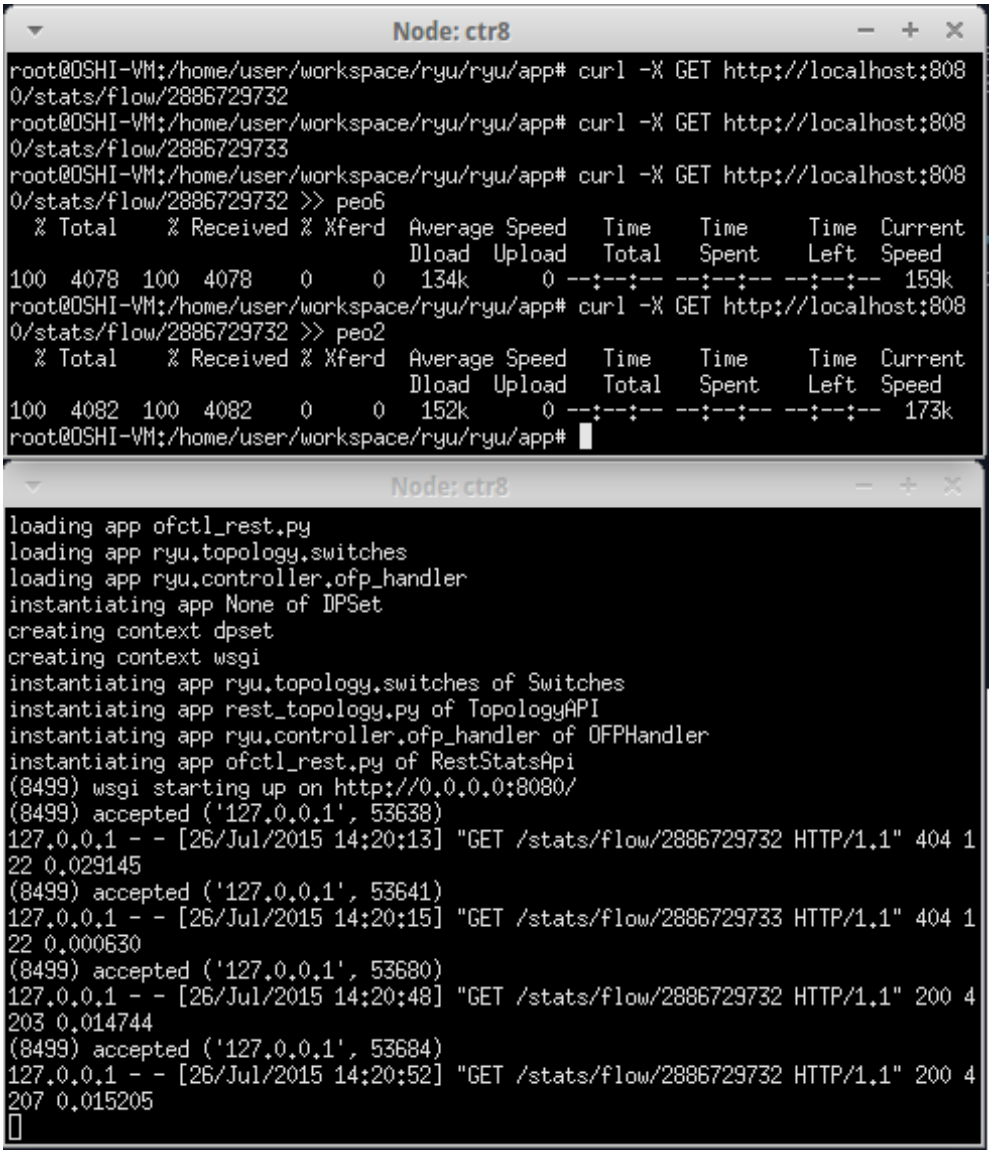


Figura 19. Consulta de tablas de Flujo de switches peo2 y peo6.

e. Se realiza el mismo procedimiento para los routers de núcleo, cro3, cro4 y cro5.

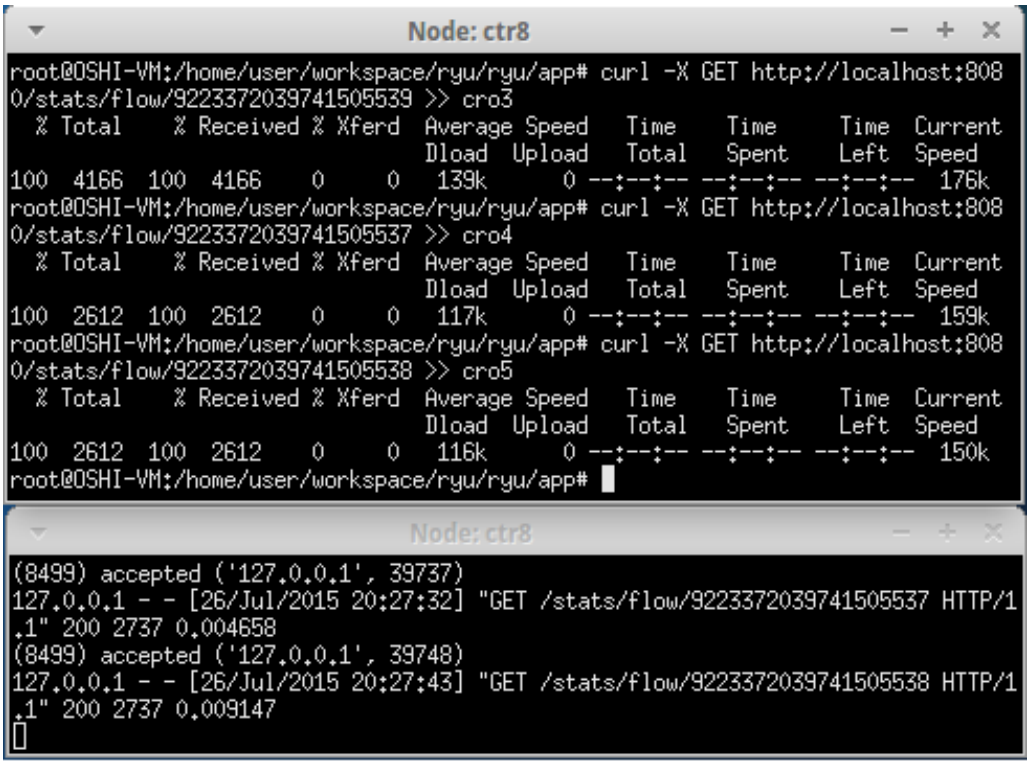


Figura 20. Consulta de tablas de Flujo de switches cro3, cro4 y cro5.

6. Se realiza una prueba de ping desde la consola de cer1 hacia las IP 10.0.11.2 y 10.0.12.2 del cer7 que corresponden a interfaces Ethernet donde se brindarán los servicios de VLL y PW entres los routers clientes. Y se realiza la misma prueba, ahora desde la consola de cer7 hacia las IP 10.0.11.1 y 10.0.12.1 del cer1. Se comprueba que no hay respuesta, debido a que no se encuentran instalados los circuitos MPLS aún.

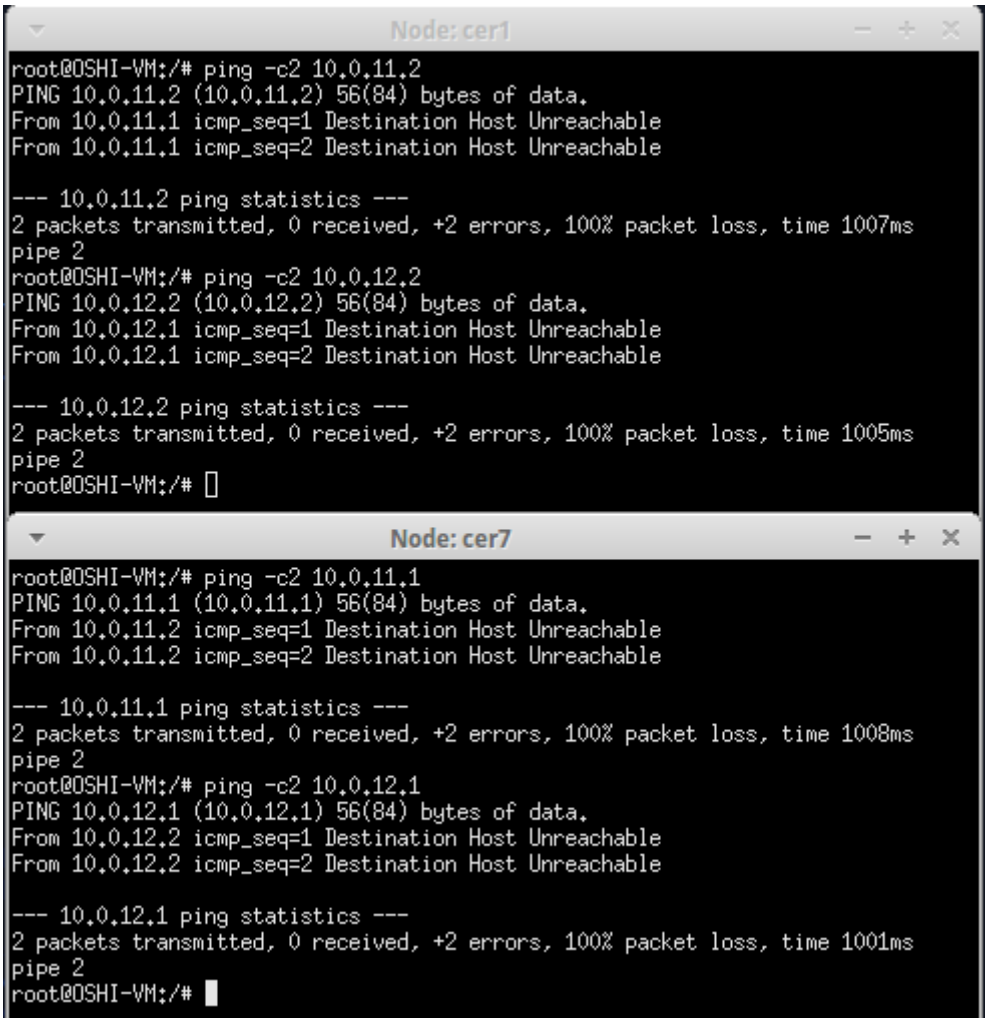


Figura 21. Prueba de ping entre cer1 y cer7 antes de creación de circuitos virtuales – interfaces para Servicios VLL y PW.

7. En el segundo terminal de ctr8 abierto, se copia el archivo de configuración vll_pusher.cfg del directorio Dreamer-Mininet-Extensions al directorio Dreamer-VLL-Pusher, este archivo fue creado por el script mininet_deployer.py en el paso 1 al desplegar la topología.

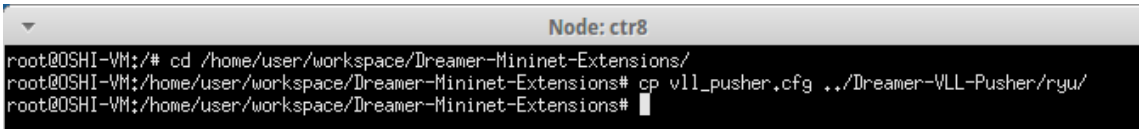


Figura 22. Copia de archivo de configuración vll_pusher.cfg.

8. Se crean los circuitos virtuales VLL y PW, corriendo los siguientes comandos:

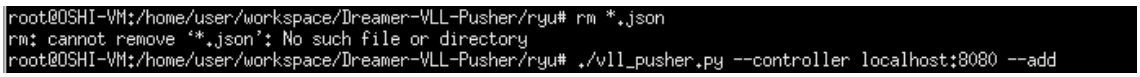


Figura 23. Creación de circuitos virtuales.

9. Se realiza la misma prueba de ping que en el apartado 7 y se comprueba que los circuitos virtuales VLL IP y PW han sido configurados, como se muestra en la siguiente figura:

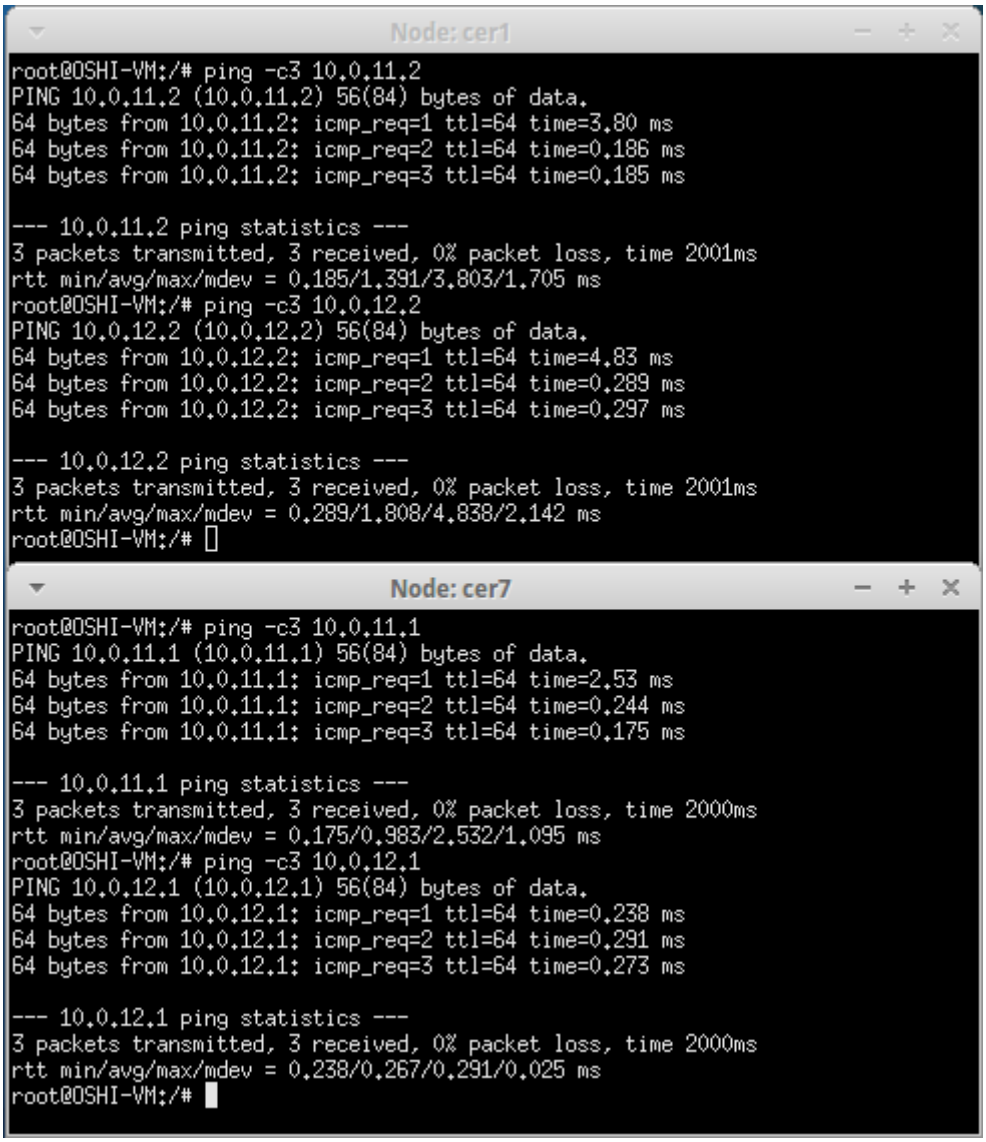


Figura 24. Prueba de ping entre cer1 y cer7 luego de la creación de los circuitos virtuales.

10. Se obtienen las tablas de flujos como en el paso 5d y 5e.

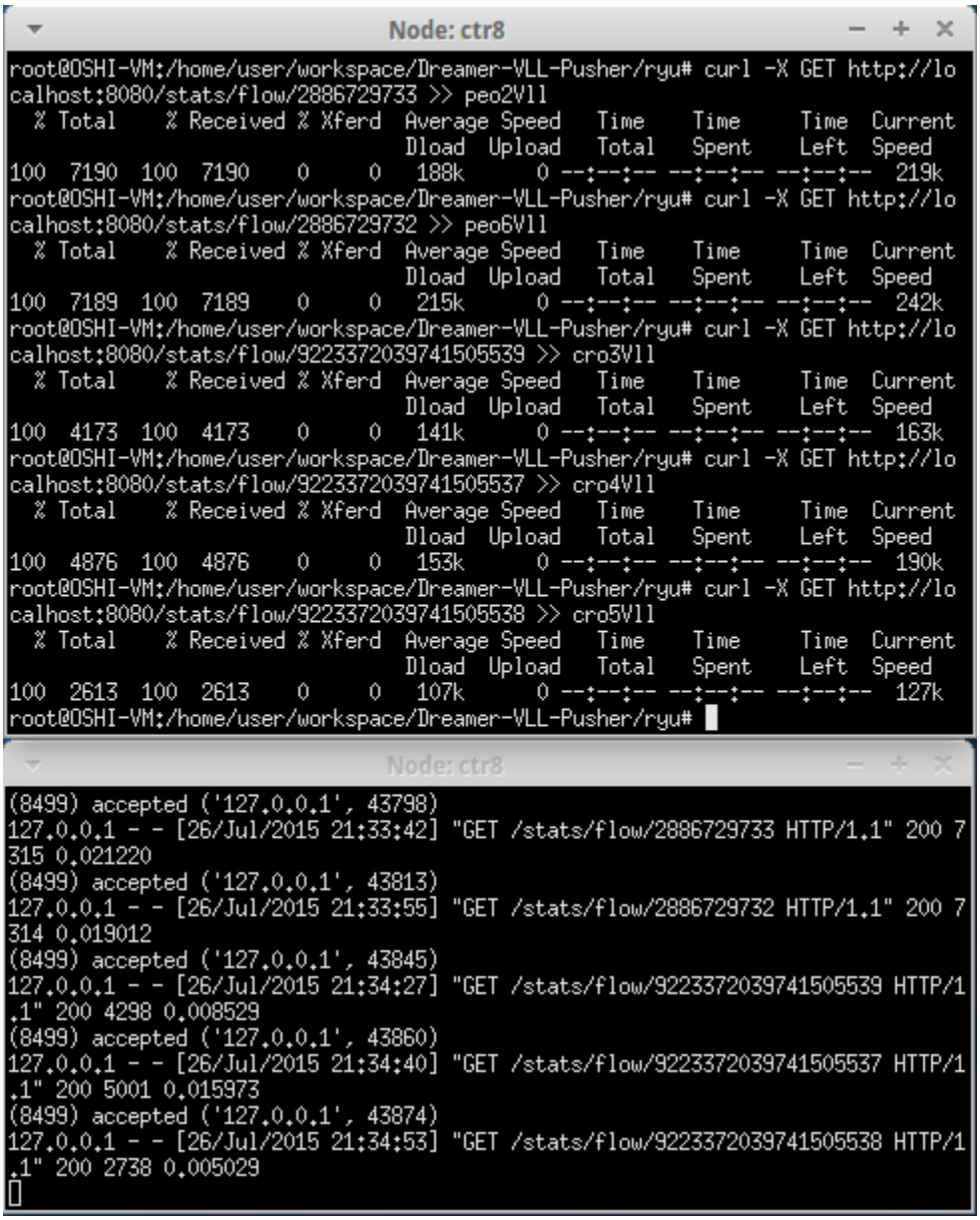


Figura 25. Consultas de tablas de flujo de switches luego de creación de circuitos virtuales.

11. Se ordenan y comparan las tablas de flujo de los pe obtenidas en los apartados 5d y 10. Podemos notar que la diferencia más significativa entre las tablas de flujo antes y después de la creación de los circuitos virtuales (ver anexo 3), son las últimas 6 entradas de las tablas, que se muestran a continuación ordenadas.

```
{"actions":
   ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:16}", "OUTPUT:6"],
   "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6,
   "hard_timeout": 0, "byte_count": 588, "length": 120,
   "duration_nsec":   164000000,   "priority":   32768,   "duration_sec":   961,
   "table_id": 1, "flags": 0,
   "match": {"dl_type": 2048, "in_port": 9}
},

{"actions":
   ["PUSH_MPLS:34888", "SET_FIELD: {mpls_label:16}", "OUTPUT:6"],
   "idle_timeout":   0,   "cookie":   10316443042833537412,   "packet_count":   2,
   "hard_timeout": 0, "byte_count": 84, "length": 120,
   "duration_nsec": 128000000, "priority": 32768,
   "duration_sec": 961, "table_id": 1, "flags": 0,
   "match": {"dl_type": 2054, "in_port": 9}
},

{"actions":
   ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:17}",
       "SET_FIELD: {eth_src:02:00:ac:10:00:05}",
       "SET_FIELD: {eth_dst:02:00:ac:10:00:01}", "OUTPUT:6"],
   "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 386,
   "hard_timeout": 0, "byte_count": 34618, "length": 152,
   "duration_nsec":   481000000,   "priority":   32768,   "duration_sec":   960,
   "table_id": 1, "flags": 0,
   "match": {"dl_type": 2048, "in_port": 13}
},

{"actions":
   ["POP_MPLS:2048", "OUTPUT:9"],
   "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6,
   "hard_timeout": 0, "byte_count": 612, "length": 112,
   "duration_nsec": 89000000, "priority": 32768,
   "duration_sec": 961, "table_id": 1, "flags": 0,
   "match": {"dl_type": 34887, "mpls_label": 16, "in_port": 6}
},

{"actions":
   ["POP_MPLS:2054", "OUTPUT:9"],
   "idle_timeout":   0,   "cookie":   10316443042833537412,   "packet_count":   2,
   "hard_timeout": 0,
   "byte_count": 92, "length": 112, "duration_nsec": 34000000, "priority": 32768,
   "duration_sec": 961, "table_id": 1, "flags": 0,
   "match": {"dl_type": 34888, "mpls_label": 16, "in_port": 6}
},

{"actions":
   ["POP_MPLS:2048", "SET_FIELD: {eth_src:00:00:ac:10:00:02}",
       "SET_FIELD: {eth_dst:00:00:ac:10:00:01}", "OUTPUT:13"],
   "idle_timeout":   0,   "cookie":   1054622872232268153,   "packet_count":   385,
   "hard_timeout": 0, "byte_count": 36069, "length": 152,
   "duration_nsec":   439000000,   "priority":   32768,   "duration_sec":   960,
   "table_id": 1, "flags": 0,
   "match": {"dl_type": 34887, "dl_dst": "02:00:ac:10:00:05", "mpls_label": 17,
       "in_port": 6}
}
```


Flujos MPLS en tablas de flujos de peo2

Estas últimas 6 entradas corresponden al protocolo MPLS, que son definidas por el script vll_pusher.py en el peo2 para permitir la comunicación con el peo6 directamente por los circuitos virtuales VLL IP y PW.

```
{"actions":
       ["POP_MPLS:2048", "OUTPUT:9"],
       "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6,
       "hard_timeout": 0, "byte_count": 612, "length": 112,
       "duration_nsec": 576000000, "priority": 32768, "duration_sec": 973,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 34887, "mpls_label": 16, "in_port": 5}
},

{"actions":
       ["POP_MPLS:2054", "OUTPUT:9"],
       "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2,
       "hard_timeout": 0, "byte_count": 92, "length": 112,
       "duration_nsec": 531000000, "priority": 32768, "duration_sec": 973,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 34888, "mpls_label": 16, "in_port": 5}
},

{"actions":
       ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:16}", "OUTPUT:5"],
       "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6,
       "hard_timeout": 0, "byte_count": 588, "length": 120,
       "duration_nsec": 495000000, "priority": 32768, "duration_sec": 973,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 2048, "in_port": 9}
},

{"actions":
       ["PUSH_MPLS:34888", "SET_FIELD: {mpls_label:16}", "OUTPUT:5"],
       "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2,
       "hard_timeout": 0, "byte_count": 84, "length": 120,
       "duration_nsec": 462000000, "priority": 32768, "duration_sec": 973,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 2054, "in_port": 9}
},

{"actions":
       ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:17}",
              "SET_FIELD: {eth_src:02:00:ac:10:00:04}",
              "SET_FIELD: {eth_dst:02:00:ac:10:00:01}", "OUTPUT:5"],
       "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 390,
       "hard_timeout": 0, "byte_count": 34974, "length": 152,
       "duration_nsec": 929000000, "priority": 32768, "duration_sec": 972,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 2048, "in_port": 13}
},

{"actions":
       ["POP_MPLS:2048", "SET_FIELD: {eth_src:00:00:ac:10:00:01}",
              "SET_FIELD: {eth_dst:00:00:ac:10:00:02}", "OUTPUT:13"],
       "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 391,
       "hard_timeout": 0, "byte_count": 36627, "length": 152,
       "duration_nsec": 978000000, "priority": 32768, "duration_sec": 972,
       "table_id": 1, "flags": 0,
       "match": {"dl_type": 34887, "dl_dst": "02:00:ac:10:00:04",
              "mpls_label": 17, "in_port": 5}
}
```


Flujos MPLS en tablas de flujos de peo6

Estas últimas 6 entradas corresponden al protocolo MPLS, que son definidas por el script vll_pusher.py en el peo6 para permitir la comunicación con el peo2 directamente por los circuitos virtuales VLL IP y PW.

Para comprobar si estas entradas en las tablas de flujo crean rutas entre el peo2 y el peo6 en la topología de la red para los dos servicios, se realizó una consulta a la función REST API: curl -X GET http:localhost:8080/v1.0/topology/route/2886729733/00000003/2886729732/00000004 del script rest_topology.py como se muestra en la siguiente figura.

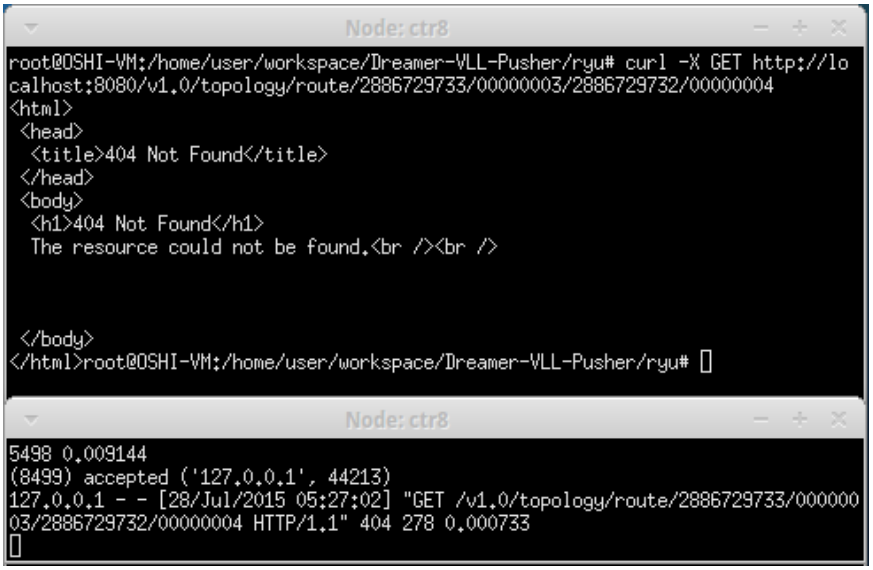


Figura 26. Consulta de rutas entre peo2 y peo6.

Como se observa, se obtiene una respuesta HTTP 404 de error, lo que significa que no existe el recurso que estamos consultando. Esto quiere decir que la ruta no es encuentra guardada estáticamente sino que el controlador gestiona las tablas de los nodos de manera automática.

De las pruebas realizadas, cabe destacar que se realizaron también pruebas con las herramientas iperf, tcpdump y capturas de tráfico con Wireshark, para comprobar el tráfico existente en los circuitos virtuales entre los clientes cer1 y cer7 conectados a los pes, como se muestra en las siguientes figuras.

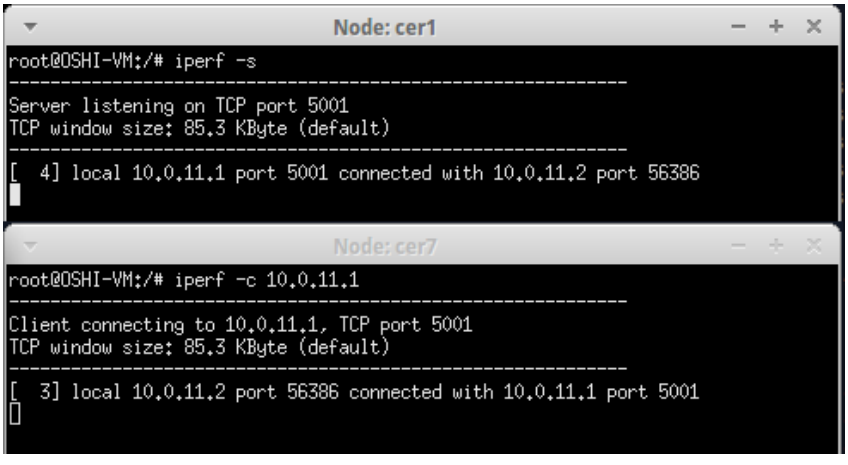


Figura 27. Pruebas de tráfico TCP entre cer1 y cer7.

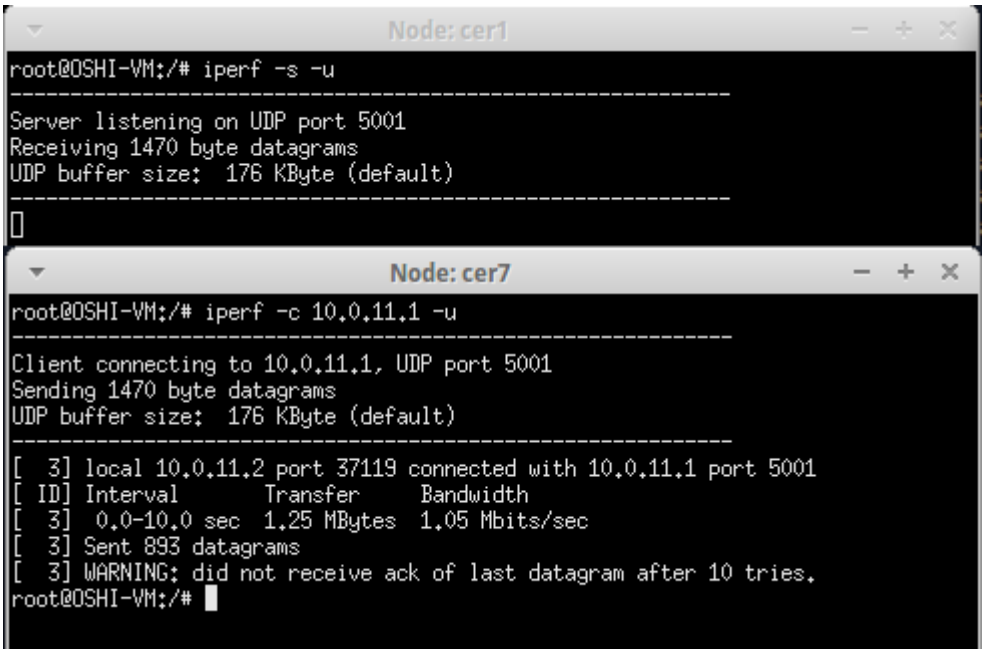


Figura 28. Pruebas de tráfico UDP entre cer1 y cer7.

Pero ninguna de estas pruebas fue favorable, aún cuando los dos nodos logran alcanzarse uno al otro luego de la creación de los circuitos.

Según el análisis realizado, se cree que esta puede ser una desventaja del emulador Mininet, debido a que es una prueba sencilla de conectividad que debería arrojar resultados. Tampoco pudimos ingresar a la configuración interna de cada nodo para acceder a sus componentes y comprobar la arquitectura de los nodos OSHI, esto debido a que en el diseño se usan Namespaces para alojar cada componente del nodo, y de alguna manera se los "oculta" a la vista del usuario.

Si bien para efectos de montar un ejemplo rápido y comprobar que una configuración o arquitectura propuesta funciona, Mininet ofrece agilidad en el proceso, realmente no se conoce bien que aplicaiones o sistemas son los que corren por debajo, y si lo que se quiere es realizar pruebas de funcionamiento a muy bajo nivel, Mininet resulta poco didáctico para llegar a la comprensión total de los escenarios.

5 Conclusiones y Trabajo Futuro

5.1 Conclusiones

De la investigación realizada en la presente memoria, se puede concluir que los esfuerzos por lograr la implementación de mejoras en ingeniería de tráfico sobre redes definidas por software son cada vez mayores, y que la inversión a nivel de industria en este tipo de redes siempre se da teniendo en la mira el ofrecimiento de mejoras por medio de la ingeniería de tráfico. Estos esfuerzos dan como resultado la búsqueda por dotar de un mínimo de inteligencia (cada vez se busca más) a los switches o nodos de la red para que sean capaces de realizar un poco más que solo funciones de reenvío de paquetes. Sin embargo, esta búsqueda conlleva a que la complejidad de la composición del switch aumente. Esto se da sobre todo cuando se busca poder recuperarse rápidamente ante fallos en la red como se presentó en el apartado 3.1.2.

Con respecto al plano de control, se proponen mecanismos que buscan eliminar el único punto de fallo en la red mediante la implementación de un enfoque distribuido, sin embargo este enfoque puede llegar a complicar el funcionamiento de la red, provocando sobrecarga en la comunicación entre los controladores, ya que siempre uno se propone como el controlador principal que gestione la carga entre todos.

El desarrollo de ingeniería de tráfico sobre redes SDN está sólo iniciando con cada una de las herramientas y técnicas presentadas en el apartado 3. Para que estas iniciativas y las soluciones a problemas ya conocidos en redes IP se vayan aplicando en redes SDN, es necesario que se interactúe mucho más entre los diferentes gestores de la innovación en esta área, por ejemplo, dando paso a la colaboración entre soluciones ya realizadas y potenciando las capacidades de las mismas.

En cuanto a las pruebas realizadas en los escenarios de servicio, se pudo comprobar como están formados los dpids según la especificación del protocolo OpenFlow y como en el caso específico de OSHI son implementados para la asociación con cada nodo.

Si bien la parte teórica del estudio del caso OSHI quedó muy clara, no se pudo alcanzar el ojetivo de comprobar los componentes que hacen parte de la arquitectura de OSHI, es decir, no se pudo ingresar a cada componente, aunque si se comprobó las versiones de las herramientas necesarias para que se ejecuten estos componentes. Esto debido a que se usaron Namespaces para la implementación de los mismos y al parecer fueron ocultados con algún mecanismo del kernel de Linux.

Se atribuye parte de la falta de éxito total en el entedimiento del funcionamiento del caso de estudio al ambiente de emulación en Mininet, debido a que es poco didáctico para efectos de este tipo de análisis de funcionalidades y que al usar virtualización compartida de recursos llega a limitar el estudio de cada nodo en su totalidad.

5.2 Trabajo Futuro

A través de la experiencia obtenida con este trabajo, se propone a futuro la integración de la herramienta de virtualización VNX como base para el despliegue de escenarios sobre redes OSHI. Para ello, es necesario estudiar los scripts de despliegue que usan en el proyecto y la compatibilidad de versiones de protocolos y demás herramientas de OSHI con VNX.

Teniendo en cuenta que el objetivo planteado en este proyecto fue estudiar el estado del arte de TE sobre redes SDN y que en el caso de estudio OSHI se ha realizado un escenario de experimentación muy reciente usando el protocolo Segment Routing, se propone como extensión de este trabajo realizar un análisis de esta experimentación cuando la máquina virtual facilitada por el grupo de investigación sea publicada.

Bibliografía

[1] Erick D Osborne and Ajay Simha, *Traffic Engineering with MPLS*. Indianapolis, USA: Cisco Press, 2004.

[2] Nick Feamster, Jennifer Rexford, and Ellen Zegura, "The Road to SDN: An Intellectual History," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87-98, Abril 2014.

[3] University of Rome "Tor Vergata" Networking Group. (2015, Julio) Web Home Oshi. [Online]. http://netgroup.uniroma2.it/twiki/bin/view/Oshi

[4] Mininet Team. (2015) Mininet: An Instant Virtual Network on your Laptop. [Online]. http://mininet.org/

[5] DIT, UPM. (2015, Junio) VNX. [Online]. http://web.dit.upm.es/vnxwiki/index.php/Main_Page

[6] J Salim, H Khosravi, A Kleen, and A Kuznetsov. (2003, Julio) Linux Netlink as an IP Services Protocol. RFC 3549.

[7] N Feamster, H Balakrishnan, J Rexford, A Shaikh, and K van der Merwe, "The case for separating routing from routers," *ACM SIGCOMM Workshop on Future Directions in Network*, Septiembre 2004.

[8] T. V Lakshman, T Nandagopal, R Ramjee, K Sabnani, and T Woo, "The SoftRouter Architecture," *Proc. 3nd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, Noviembre 2004.

[9] Open Networking Foundation. (2013, Septiembre) Open Networking Foundation. [Online]. https://www.opennetworking.org/images/stories/downloads/about/onf-what-why.pdf

[10] Open Networking Foundation. (2014, Noviembre) SDN Architecture Overview, Versión 1.1. [Online]. https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN-ARCH-Overview-1.1-11112014.02.pdf

[11] Hyojoon Kim and Nick Feamster, "Improving Network Management with Software Defined Networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114 - 119, Febrero 2013.

[12] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1-30, Octubre 2014.

[13] SDxCentral. (2015, Junio) What are SDN Northbound APIs? [Online]. https://www.sdxcentral.com/resources/sdn/north-bound-interfaces-api/

[14] Frowgrammable. (2014, Noviembre) SND / OpenFlow / Frowgrammable. [Online]. http://flowgrammable.org/sdn/openflow/#tab_protocol

[15] Open Networking Foundation. (2011, Febrero 28) openflow-spec-v1.1.0. [Online]. http://archive.openflow.org/documents/openflow-spec-v1.1.0.pdf

[16] R Callon, A Viswanathan, and R Callon, "Multiprotocol Label Switching Architecture," *IETF RFC 3031*, Enero 2001.

[17] Walter Goralski, "MPLS-Based Virtual Private Networks," in *The Illustrated Network*. Burlington, USA: Morgan Kaufmann Publishers, 2009, ch. 26, pp. 672-676.

[18] S Bryant and P Pate. (2015, Marzo) Pseudo Wire Emulation Edge-to-Edge (PWE3) Architecture. RFC 3985.

[19] M Desai and T Nandagopal, "Coping with link failures in centralized control plane architectures," *Proceedings of 2010 Second International Conference on Communication Systems and Networks, COMSNETS'10*, pp. 1-10, Enero 2010.

[20] Pier Luigi Ventre, "Scholarship "Orio Carlini" Quarterly Reports QR.2 (January 2014 - March 2014)," Università degli Studi di Roma «Tor Vergata», Roma, Scholarship Report 2014.

[21] Stefano Salsano et al., "OSHI - Open Source Hybrid IP/SDN networking and Mantoo - a set of management tools for controlling SDN/NFV experiments," *Cornell Univesity Library*, Mayo 2015.

[22] P L Ventre, S Salsano, G Siracusano, L Prete, and M Gerola, "OSHI - Open Source Hybrid IP/SDN networking (and its emulation on Mininet and on distributed SDN testbeds)," Consortium GARR / CNIT / University of Rome "Tor Vergata", Roma, Technical Report - Version 1.03 2014.

[23] Networking Group, University of Rome "Tor Vergata". (2015, Junio) WebHome < Oshi < TWiki. [Online].

http://netgroup.uniroma2.it/twiki/bin/view/Oshi/WebHome#AnchorSoftDown

[24] (2015, Junio) Leased line - Wikipedia, the free encyclopedia. [Online]. https://en.wikipedia.org/wiki/Leased_line

Anexos

Anexo 1: Funciones REST API del controlador

1. Función para obtener y modificar información de la configuración de la topología: rest_topology.py

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import json
from webob import Response
import networkx as nx
from random import randrange


from ryu.app.wsgi import ControllerBase, WSGIApplication, route
from ryu.base import app_manager
from ryu.lib import dpid as dpid_lib
from ryu.lib import port_no as port_no_lib
from ryu.topology.api import get_switch, get_link

# REST API for switch configuration
#
# get all the switches
# GET /v1.0/topology/switches
#
# get the switch
# GET /v1.0/topology/switches/<dpid>
#
# get all the links
# GET /v1.0/topology/links
#
# get the links of a switch
# GET /v1.0/topology/links/<dpid>
#
# XXX
# get the route that interconnects input switches
# GET /v1.0/topology/route/<srcdpid>/<srcport>/<dstdpid>/<dstport>
#
# where
# <dpid>: datapath id in 16 hex
```

```
class TopologyAPI(app_manager.RyuApp):
    _CONTEXTS = {
        'wsgi': WSGIApplication
    }

    def __init__(self, *args, **kwargs):
        super(TopologyAPI, self).__init__(*args, **kwargs)

        wsgi = kwargs['wsgi']
        wsgi.register(TopologyController, {'topology_api_app': self})


class TopologyController(ControllerBase):
    def __init__(self, req, link, data, **config):
        super(TopologyController, self).__init__(req, link, data,
**config)
        self.topology_api_app = data['topology_api_app']

    @route('topology', '/v1.0/topology/switches',
           methods=['GET'])
    def list_switches(self, req, **kwargs):
        return self._switches(req, **kwargs)

    @route('topology', '/v1.0/topology/switches/{dpid}',
           methods=['GET'], requirements={'dpid':
dpid_lib.DPID_PATTERN})
    def get_switch(self, req, **kwargs):
        return self._switches(req, **kwargs)

    @route('topology', '/v1.0/topology/links',
           methods=['GET'])
    def list_links(self, req, **kwargs):
        return self._links(req, **kwargs)

    @route('topology', '/v1.0/topology/links/{dpid}',
           methods=['GET'], requirements={'dpid':
dpid_lib.DPID_PATTERN})
    def get_links(self, req, **kwargs):
        return self._links(req, **kwargs)

    @route('topology',
'/v1.0/topology/route/{srcdpid}/{srcport}/{dstdpid}/{dstport}',
           methods=['GET'], requirements={'srcdpid':
dpid_lib.DPID_PATTERN,
          'srcport': port_no_lib.PORT_NO_PATTERN, 'dstdpid':
dpid_lib.DPID_PATTERN,
          'dstport': port_no_lib.PORT_NO_PATTERN})
    def get_route(self, req, **kwargs):
        return self._route(req, **kwargs)

    def _switches(self, req, **kwargs):
        dpid = None
        if 'dpid' in kwargs:
            dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
        switches = get_switch(self.topology_api_app, dpid)
        body = json.dumps([switch.to_dict() for switch in switches])
        return Response(content_type='application/json', body=body)

    def _links(self, req, **kwargs):
```

```
        dpid = None
        if 'dpid' in kwargs:
            dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
        links = get_link(self.topology_api_app, dpid)
        body = json.dumps([link.to_dict() for link in links])
        return Response(content_type='application/json', body=body)

    def _route(self, req, **kwargs):
        srcdpid=dpid_lib.str_to_dpid(kwargs['srcdpid'])
        srcport=port_no_lib.str_to_port_no(kwargs['srcport'])
        dstdpid=dpid_lib.str_to_dpid(kwargs['dstdpid'])
        dstport=port_no_lib.str_to_port_no(kwargs['dstport'])
        links = get_link(self.topology_api_app, None)

        topology = nx.MultiDiGraph()
        for link in links:
            print link
            topology.add_edge(link.src.dpid, link.dst.dpid,
src_port=link.src.port_no, dst_port=link.dst.port_no)

        try:
            shortest_path = nx.shortest_path(topology, srcdpid,
dstdpid)
        except (nx.NetworkXError, nx.NetworkXNoPath):
            body = json.dumps([])
            print "Error"
            return Response(content_type='application/json',
body=body)

        ingressPort = NodePortTuple(srcdpid, srcport)
        egressPort = NodePortTuple(dstdpid, dstport)
        route = []
        route.append(ingressPort)

        for i in range(0, len(shortest_path)-1):
            link = topology[shortest_path[i]][shortest_path[i+1]]
            index = randrange(len(link))
            dstPort = NodePortTuple(shortest_path[i],
link[index]['src_port'])
            srcPort = NodePortTuple(shortest_path[i+1],
link[index]['dst_port'])
            route.append(dstPort)
            route.append(srcPort)

        route.append(egressPort)
        body = json.dumps([hop.to_dict() for hop in route])
        return Response(content_type='application/json', body=body)


class NodePortTuple(object):

    def __init__(self, dpid, port_no):
        self.dpid = dpid_lib.dpid_to_str(dpid)
        self.port_no = port_no_lib.port_no_to_str(port_no)

    def to_dict(self):
        return {'switch': self.dpid, 'port':self.port_no}

    def __str__(self):
        return 'NodePortTuple<switch=%s, port=%s>' % (self.dpid,
self.port_no)
```

2. Función para obtener y modificar información de los estados de los switches: ofctl_rest.py

```
# Copyright (C) 2012 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions
and
# limitations under the License.

import logging

import json
import ast
from webob import Response

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller import dpset
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0
from ryu.ofproto import ofproto_v1_2
from ryu.ofproto import ofproto_v1_3
from ryu.lib import ofctl_v1_0
from ryu.lib import ofctl_v1_2
from ryu.lib import ofctl_v1_3
from ryu.app.wsgi import ControllerBase, WSGIApplication


LOG = logging.getLogger('ryu.app.ofctl_rest')

# REST API
#

# Retrieve the switch stats
#
# get the list of all switches
# GET /stats/switches
#
# get the desc stats of the switch
# GET /stats/desc/<dpid>
#
# get flows stats of the switch
# GET /stats/flow/<dpid>
#
# get flows stats of the switch filtered by the fields
```

```
# POST /stats/flow/<dpid>
#
# get aggregate flows stats of the switch
# GET /stats/aggregateflow/<dpid>
#
# get aggregate flows stats of the switch filtered by the fields
# POST /stats/aggregateflow/<dpid>
#
# get ports stats of the switch
# GET /stats/port/<dpid>
#
# get queues stats of the switch
# GET /stats/queue/<dpid>
#
# get meter features stats of the switch
# GET /stats/meterfeatures/<dpid>
#
# get meter config stats of the switch
# GET /stats/meterconfig/<dpid>
#
# get meters stats of the switch
# GET /stats/meter/<dpid>
#
# get group features stats of the switch
# GET /stats/groupfeatures/<dpid>
#
# get groups desc stats of the switch
# GET /stats/groupdesc/<dpid>
#
# get groups stats of the switch
# GET /stats/group/<dpid>
#
# get ports description of the switch
# GET /stats/portdesc/<dpid>

# Update the switch stats
#
# add a flow entry
# POST /stats/flowentry/add
#
# modify all matching flow entries
# POST /stats/flowentry/modify
#
# modify flow entry strictly matching wildcards and priority
# POST /stats/flowentry/modify_strict
#
# delete all matching flow entries
# POST /stats/flowentry/delete
#
# delete flow entry strictly matching wildcards and priority
# POST /stats/flowentry/delete_strict
#
# delete all flow entries of the switch
# DELETE /stats/flowentry/clear/<dpid>
#
# add a meter entry
# POST /stats/meterentry/add
#
```

```
# modify a meter entry
# POST /stats/meterentry/modify
#
# delete a meter entry
# POST /stats/meterentry/delete
#
# add a group entry
# POST /stats/groupentry/add
#
# modify a group entry
# POST /stats/groupentry/modify
#
# delete a group entry
# POST /stats/groupentry/delete
#
# modify behavior of the physical port
# POST /stats/portdesc/modify
#
#
# send a experimeter message
# POST /stats/experimenter/<dpid>


class StatsController(ControllerBase):
    def __init__(self, req, link, data, **config):
        super(StatsController, self).__init__(req, link, data,
**config)
        self.dpset = data['dpset']
        self.waiters = data['waiters']

    def get_dpids(self, req, **_kwargs):
        dps = self.dpset.dps.keys()
        body = json.dumps(dps)
        return (Response(content_type='application/json',
body=body))

    def get_desc_stats(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            desc = ofctl_v1_0.get_desc_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            desc = ofctl_v1_2.get_desc_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            desc = ofctl_v1_3.get_desc_stats(dp, self.waiters)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(desc)
        return (Response(content_type='application/json',
body=body))

    def get_flow_stats(self, req, dpid, **_kwargs):
        if req.body == '':
            flow = {}
```

```
        else:
            try:
                flow = ast.literal_eval(req.body)
            except SyntaxError:
                LOG.debug('invalid syntax %s', req.body)
                return Response(status=400)

        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            flows = ofctl_v1_0.get_flow_stats(dp, self.waiters,
flow)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            flows = ofctl_v1_2.get_flow_stats(dp, self.waiters,
flow)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            flows = ofctl_v1_3.get_flow_stats(dp, self.waiters,
flow)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(flows)
        return (Response(content_type='application/json',
body=body))

    def get_aggregate_flow_stats(self, req, dpid, **_kwargs):
        if req.body == '':
            flow = {}
        else:
            try:
                flow = ast.literal_eval(req.body)
            except SyntaxError:
                LOG.debug('invalid syntax %s', req.body)
                return Response(status=400)

        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            flows = ofctl_v1_0.get_aggregate_flow_stats(dp,
self.waiters, flow)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            flows = ofctl_v1_2.get_aggregate_flow_stats(dp,
self.waiters, flow)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            flows = ofctl_v1_3.get_aggregate_flow_stats(dp,
self.waiters, flow)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(flows)
        return Response(content_type='application/json', body=body)
```

```
    def get_port_stats(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            ports = ofctl_v1_0.get_port_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ports = ofctl_v1_2.get_port_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ports = ofctl_v1_3.get_port_stats(dp, self.waiters)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(ports)
        return (Response(content_type='application/json',
body=body))

    def get_queue_stats(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            queues = ofctl_v1_0.get_queue_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            queues = ofctl_v1_2.get_queue_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            queues = ofctl_v1_3.get_queue_stats(dp, self.waiters)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(queues)
        return Response(content_type='application/json', body=body)

    def get_meter_features(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            meters = ofctl_v1_3.get_meter_features(dp,
self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION or
\
                dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(meters)
        return (Response(content_type='application/json',
body=body))
```

```
    def get_meter_config(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            meters = ofctl_v1_3.get_meter_config(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION or
\
                dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(meters)
        return (Response(content_type='application/json',
body=body))

    def get_meter_stats(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            meters = ofctl_v1_3.get_meter_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION or
\
                dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(meters)
        return (Response(content_type='application/json',
body=body))

    def get_group_features(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            groups = ofctl_v1_2.get_group_features(dp,
self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            groups = ofctl_v1_3.get_group_features(dp,
self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
```

```
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(groups)
        return Response(content_type='application/json', body=body)

    def get_group_desc(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            groups = ofctl_v1_2.get_group_desc(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            groups = ofctl_v1_3.get_group_desc(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(groups)
        return Response(content_type='application/json', body=body)

    def get_group_stats(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            groups = ofctl_v1_2.get_group_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            groups = ofctl_v1_3.get_group_stats(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(groups)
        return Response(content_type='application/json', body=body)

    def get_port_desc(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            groups = ofctl_v1_0.get_port_desc(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            groups = ofctl_v1_2.get_port_desc(dp, self.waiters)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            groups = ofctl_v1_3.get_port_desc(dp, self.waiters)
```

```
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        body = json.dumps(groups)
        return Response(content_type='application/json', body=body)

    def mod_flow_entry(self, req, cmd, **_kwargs):
        try:
            flow = ast.literal_eval(req.body)
        except SyntaxError:
            LOG.debug('invalid syntax %s', req.body)
            return Response(status=400)

        dpid = flow.get('dpid')
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if cmd == 'add':
            cmd = dp.ofproto.OFPFC_ADD
        elif cmd == 'modify':
            cmd = dp.ofproto.OFPFC_MODIFY
        elif cmd == 'modify_strict':
            cmd = dp.ofproto.OFPFC_MODIFY_STRICT
        elif cmd == 'delete':
            cmd = dp.ofproto.OFPFC_DELETE
        elif cmd == 'delete_strict':
            cmd = dp.ofproto.OFPFC_DELETE_STRICT
        else:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            ofctl_v1_0.mod_flow_entry(dp, flow, cmd)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ofctl_v1_2.mod_flow_entry(dp, flow, cmd)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ofctl_v1_3.mod_flow_entry(dp, flow, cmd)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        return Response(status=200)

    def delete_flow_entry(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        flow = {'table_id': dp.ofproto.OFPTT_ALL}

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            ofctl_v1_0.delete_flow_entry(dp)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ofctl_v1_2.mod_flow_entry(dp, flow,
dp.ofproto.OFPFC_DELETE)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
```

```
            ofctl_v1_3.mod_flow_entry(dp, flow,
dp.ofproto.OFPFC_DELETE)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        return Response(status=200)

    def mod_meter_entry(self, req, cmd, **_kwargs):
        try:
            flow = ast.literal_eval(req.body)
        except SyntaxError:
            LOG.debug('invalid syntax %s', req.body)
            return Response(status=400)

        dpid = flow.get('dpid')
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if cmd == 'add':
            cmd = dp.ofproto.OFPMC_ADD
        elif cmd == 'modify':
            cmd = dp.ofproto.OFPMC_MODIFY
        elif cmd == 'delete':
            cmd = dp.ofproto.OFPMC_DELETE
        else:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ofctl_v1_3.mod_meter_entry(dp, flow, cmd)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION or
\
                dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        return Response(status=200)

    def mod_group_entry(self, req, cmd, **_kwargs):
        try:
            group = ast.literal_eval(req.body)
        except SyntaxError:
            LOG.debug('invalid syntax %s', req.body)
            return Response(status=400)

        dpid = group.get('dpid')
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
```

```
            return Response(status=501)

        if cmd == 'add':
            cmd = dp.ofproto.OFPGC_ADD
        elif cmd == 'modify':
            cmd = dp.ofproto.OFPGC_MODIFY
        elif cmd == 'delete':
            cmd = dp.ofproto.OFPGC_DELETE
        else:
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ofctl_v1_2.mod_group_entry(dp, group, cmd)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ofctl_v1_3.mod_group_entry(dp, group, cmd)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        return Response(status=200)

    def mod_port_behavior(self, req, cmd, **_kwargs):
        try:
            port_config = ast.literal_eval(req.body)
        except SyntaxError:
            LOG.debug('invalid syntax %s', req.body)
            return Response(status=400)

        dpid = port_config.get('dpid')

        port_no = int(port_config.get('port_no', 0))
        port_info = self.dpset.port_state[int(dpid)].get(port_no)

        if 'hw_addr' not in port_config:
            if port_info is not None:
                port_config['hw_addr'] = port_info.hw_addr
            else:
                return Response(status=404)

        if 'advertise' not in port_config:
            if port_info is not None:
                port_config['advertise'] = port_info.advertised
            else:
                return Response(status=404)

        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        if cmd != 'modify':
            return Response(status=404)

        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            ofctl_v1_0.mod_port_behavior(dp, port_config)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ofctl_v1_2.mod_port_behavior(dp, port_config)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ofctl_v1_3.mod_port_behavior(dp, port_config)
```

```
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

    def send_experimenter(self, req, dpid, **_kwargs):
        dp = self.dpset.get(int(dpid))
        if dp is None:
            return Response(status=404)

        try:
            exp = ast.literal_eval(req.body)
        except SyntaxError:
            LOG.debug('invalid syntax %s', req.body)
            return Response(status=400)

        if dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            ofctl_v1_2.send_experimenter(dp, exp)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            ofctl_v1_3.send_experimenter(dp, exp)
        elif dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            LOG.debug('Request not supported in this OF protocol
version')
            return Response(status=501)
        else:
            LOG.debug('Unsupported OF protocol')
            return Response(status=501)

        return Response(status=200)


class RestStatsApi(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION,
                    ofproto_v1_2.OFP_VERSION,
                    ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {
        'dpset': dpset.DPSet,
        'wsgi': WSGIApplication
    }

    def __init__(self, *args, **kwargs):
        super(RestStatsApi, self).__init__(*args, **kwargs)
        self.dpset = kwargs['dpset']
        wsgi = kwargs['wsgi']
        self.waiters = {}
        self.data = {}
        self.data['dpset'] = self.dpset
        self.data['waiters'] = self.waiters
        mapper = wsgi.mapper

        wsgi.registory['StatsController'] = self.data
        path = '/stats'
        uri = path + '/switches'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_dpids',
                       conditions=dict(method=['GET']))

        uri = path + '/desc/{dpid}'
```

```
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_desc_stats',
                       conditions=dict(method=['GET']))

        uri = path + '/flow/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_flow_stats',
                       conditions=dict(method=['GET', 'POST']))

        uri = path + '/aggregateflow/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
                       action='get_aggregate_flow_stats',
                       conditions=dict(method=['GET', 'POST']))

        uri = path + '/port/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_port_stats',
                       conditions=dict(method=['GET']))

        uri = path + '/queue/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_queue_stats',
                       conditions=dict(method=['GET']))

        uri = path + '/meterfeatures/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_meter_features',
                       conditions=dict(method=['GET']))

        uri = path + '/meterconfig/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_meter_config',
                       conditions=dict(method=['GET']))

        uri = path + '/meter/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_meter_stats',
                       conditions=dict(method=['GET']))

        uri = path + '/groupfeatures/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_group_features',
                       conditions=dict(method=['GET']))

        uri = path + '/groupdesc/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_group_desc',
                       conditions=dict(method=['GET']))
```

```
        uri = path + '/group/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_group_stats',
                       conditions=dict(method=['GET']))

        uri = path + '/portdesc/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='get_port_desc',
                       conditions=dict(method=['GET']))

        uri = path + '/flowentry/{cmd}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='mod_flow_entry',
                       conditions=dict(method=['POST']))

        uri = path + '/flowentry/clear/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='delete_flow_entry',
                       conditions=dict(method=['DELETE']))

        uri = path + '/meterentry/{cmd}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='mod_meter_entry',
                       conditions=dict(method=['POST']))

        uri = path + '/groupentry/{cmd}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='mod_group_entry',
                       conditions=dict(method=['POST']))

        uri = path + '/portdesc/{cmd}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='mod_port_behavior',
                       conditions=dict(method=['POST']))

        uri = path + '/experimenter/{dpid}'
        mapper.connect('stats', uri,
                       controller=StatsController,
action='send_experimenter',
                       conditions=dict(method=['POST']))

    @set_ev_cls([ofp_event.EventOFPStatsReply,
                 ofp_event.EventOFPDescStatsReply,
                 ofp_event.EventOFPFlowStatsReply,
                 ofp_event.EventOFPAggregateStatsReply,
                 ofp_event.EventOFPPortStatsReply,
                 ofp_event.EventOFPQueueStatsReply,
                 ofp_event.EventOFPMeterStatsReply,
                 ofp_event.EventOFPMeterFeaturesStatsReply,
                 ofp_event.EventOFPMeterConfigStatsReply,
```

```
                ofp_event.EventOFPGroupStatsReply,
                ofp_event.EventOFPGroupFeaturesStatsReply,
                ofp_event.EventOFPGroupDescStatsReply,
                ofp_event.EventOFPPortDescStatsReply
                ], MAIN_DISPATCHER)
    def stats_reply_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        if dp.id not in self.waiters:
            return
        if msg.xid not in self.waiters[dp.id]:
            return
        lock, msgs = self.waiters[dp.id][msg.xid]
        msgs.append(msg)

        flags = 0
        if dp.ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
            flags = dp.ofproto.OFPSF_REPLY_MORE
        elif dp.ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION:
            flags = dp.ofproto.OFPSF_REPLY_MORE
        elif dp.ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
            flags = dp.ofproto.OFPMPF_REPLY_MORE

        if msg.flags & flags:
            return
        del self.waiters[dp.id][msg.xid]
        lock.set()

    @set_ev_cls([ofp_event.EventOFPSwitchFeatures],
MAIN_DISPATCHER)
    def features_reply_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath

        if dp.id not in self.waiters:
            return
        if msg.xid not in self.waiters[dp.id]:
            return
        lock, msgs = self.waiters[dp.id][msg.xid]
        msgs.append(msg)

        del self.waiters[dp.id][msg.xid]
        lock.set()
```

Anexo 2: Configuración de la Topología de los Switches


```
[
```

PEO6

```
{"ports":
        [
                {"hw_addr": "02:00:ac:10:00:04", "name": "peo6-eth0", "port_no": "00000001", "dpid": "00000000ac100004"},
                {"hw_addr": "96:58:5d:57:0f:bd", "name": "vi0", "port_no": "00000002", "dpid": "00000000ac100004"},
                {"hw_addr": "02:97:d4:57:10:d1", "name": "vi1", "port_no": "00000003", "dpid": "00000000ac100004"},
                {"hw_addr": "02:00:ac:10:00:04", "name": "peo6-eth1", "port_no": "00000004", "dpid": "00000000ac100004"},
                {"hw_addr": "02:00:ac:10:00:04", "name": "peo6-eth2", "port_no": "00000005", "dpid": "00000000ac100004"},
                {"hw_addr": "d2:1f:0c:a4:fc:d6", "name": "vi2", "port_no": "00000006", "dpid": "00000000ac100004"}
                {"hw_addr": "02:00:ac:10:00:04", "name": "peo6-eth3", "port_no": "00000007", "dpid": "00000000ac100004"},
                {"hw_addr": "7a:ef:24:1b:50:43", "name": "vi3", "port_no": "00000008", "dpid": "00000000ac100004"},
                {"hw_addr": "76:e9:a5:a2:25:38", "name": "peo6-eth4", "port_no": "00000009", "dpid": "00000000ac100004"},
                {"hw_addr": "2e:68:42:ad:5b:0f", "name": "vi4", "port_no": "0000000a", "dpid": "00000000ac100004"},
                {"hw_addr": "d2:ef:03:bb:7a:cf", "name": "peo6-eth5", "port_no": "0000000b", "dpid": "00000000ac100004"},
                {"hw_addr": "7e:e3:1e:e8:af:c6", "name": "peo6-eth6", "port_no": "0000000c", "dpid": "00000000ac100004"},
                {"hw_addr": "3a:86:af:fa:05:c2", "name": "peo6-eth7", "port_no": "0000000d", "dpid": "00000000ac100004"}
        ],
        "dpid": "00000000ac100004"
},
```

CRO4

```
{"ports":
        [
                {"hw_addr": "02:00:ac:10:00:01", "name": "cro4-eth0", "port_no": "00000001", "dpid": "80000000ac100001"},
                {"hw_addr": "32:ad:04:19:a4:d2", "name": "vi0", "port_no": "00000002", "dpid": "80000000ac100001"},
                {"hw_addr": "02:00:ac:10:00:01", "name": "cro4-eth1", "port_no": "00000003", "dpid": "80000000ac100001"},
                {"hw_addr": "ea:5d:e4:1e:c1:67", "name": "vi1", "port_no": "00000004", "dpid": "80000000ac100001"},
                {"hw_addr": "aa:c3:34:e4:6a:00", "name": "vi2", "port_no": "00000005", "dpid": "80000000ac100001"},
                {"hw_addr": "02:00:ac:10:00:01", "name": "cro4-eth2", "port_no": "00000006", "dpid": "80000000ac100001"}
        ],
        "dpid": "80000000ac100001"
},
```

CRO5

```
{"ports":
        [
                {"hw_addr": "02:00:ac:10:00:02", "name": "cro5-eth0", "port_no": "00000001", "dpid": "80000000ac100002"},
```

```
            {"hw_addr": "86:e1:a3:b9:11:af", "name": "vi0", "port_no": "00000002", "dpid": "80000000ac100002"},
            {"hw_addr": "02:00:ac:10:00:02", "name": "cro5-eth1", "port_no": "00000003", "dpid": "80000000ac100002"},
            {"hw_addr": "be:7b:04:70:49:0a", "name": "vi1", "port_no": "00000004", "dpid": "80000000ac100002"},
            {"hw_addr": "02:00:ac:10:00:02", "name": "cro5-eth2", "port_no": "00000005", "dpid": "80000000ac100002"},
            {"hw_addr": "22:c9:2a:ef:fc:b2", "name": "vi2", "port_no": "00000006", "dpid": "80000000ac100002"}
      ],
      "dpid": "80000000ac100002"
},
```


CRO3


```
{"ports":
      [
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth0", "port_no": "00000001", "dpid": "80000000ac100003"},
            {"hw_addr": "1a:64:c6:66:04:bd", "name": "vi0", "port_no": "00000002", "dpid": "80000000ac100003"},
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth1", "port_no": "00000003", "dpid": "80000000ac100003"},
            {"hw_addr": "fa:7a:01:e0:d0:55", "name": "vi1", "port_no": "00000004", "dpid": "80000000ac100003"},
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth2", "port_no": "00000005", "dpid": "80000000ac100003"},
            {"hw_addr": "f6:8e:bb:1c:a9:17", "name": "vi2", "port_no": "00000006", "dpid": "80000000ac100003"},
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth3", "port_no": "00000007", "dpid": "80000000ac100003"},
            {"hw_addr": "ba:41:63:70:f4:d1", "name": "vi3", "port_no": "00000008", "dpid": "80000000ac100003"},
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth4", "port_no": "00000009", "dpid": "80000000ac100003"},
            {"hw_addr": "aa:dc:c6:35:35:ef", "name": "vi4", "port_no": "0000000a", "dpid": "80000000ac100003"},
            {"hw_addr": "a6:10:00:18:9b:2b", "name": "vi5", "port_no": "0000000b", "dpid": "80000000ac100003"},
            {"hw_addr": "02:00:ac:10:00:03", "name": "cro3-eth5", "port_no": "0000000c", "dpid": "80000000ac100003"}
      ],
      "dpid": "80000000ac100003"
},
```


PEO2


```
{"ports":
      [
            {"hw_addr": "02:00:ac:10:00:05", "name": "peo2-eth0", "port_no": "00000001", "dpid": "00000000ac100005"},
            {"hw_addr": "92:44:b3:48:21:0e", "name": "vi0", "port_no": "00000002", "dpid": "00000000ac100005"},
            {"hw_addr": "02:00:ac:10:00:05", "name": "peo2-eth1", "port_no": "00000003", "dpid": "00000000ac100005"},
            {"hw_addr": "c2:3e:b1:c4:2f:1f", "name": "vi1", "port_no": "00000004", "dpid": "00000000ac100005"},
            {"hw_addr": "02:00:ac:10:00:05", "name": "peo2-eth2", "port_no": "00000005", "dpid": "00000000ac100005"},
            {"hw_addr": "4e:f3:62:be:43:64", "name": "vi2", "port_no": "00000006", "dpid": "00000000ac100005"},
            {"hw_addr": "02:00:ac:10:00:05", "name": "peo2-eth3", "port_no": "00000007", "dpid": "00000000ac100005"},
            {"hw_addr": "de:86:8a:89:39:99", "name": "vi3", "port_no": "00000008", "dpid": "00000000ac100005"},
            {"hw_addr": "f2:bc:c7:70:60:b0", "name": "peo2-eth4", "port_no": "00000009", "dpid": "00000000ac100005"},
            {"hw_addr": "ee:b1:4a:69:b3:05", "name": "vi4", "port_no": "0000000a", "dpid": "00000000ac100005"},
```

```
            {"hw_addr": "1a:9b:d8:be:8d:06", "name": "peo2-eth5", "port_no": "0000000b", "dpid": "00000000ac100005"},
            {"hw_addr": "b2:d2:8f:5a:88:60", "name": "peo2-eth6", "port_no": "0000000c", "dpid": "00000000ac100005"},
            {"hw_addr": "96:84:4e:5f:89:8b", "name": "peo2-eth7", "port_no": "0000000d", "dpid": "00000000ac100005"}
        ],
        "dpid": "00000000ac100005"}
]
```

Anexo 3: Tablas de Flujo de los PE

1. Tabla de Flujo de peo2 antes de creación de circuitos virtuales.


```
{
	"2886729733":
	[
		{"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0,
"byte_count": 0, "length": 88, "duration_nsec": 692000000, "priority": 301, "duration_sec": 256, "table_id": 0, "flags": 0,
"match": {"dl_type": 35138}},

		{"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 124, "hard_timeout": 0,
"byte_count": 6324, "length": 88, "duration_nsec": 682000000, "priority": 301, "duration_sec": 256, "table_id": 0, "flags": 0,
"match": {"dl_type": 35020}},

		{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0,
"length": 72, "duration_nsec": 325000000, "priority": 32768, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"dl_type":
34888}},

		{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0,
"length": 72, "duration_nsec": 316000000, "priority": 32768, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"dl_type":
34887}},

		{"actions": ["OUTPUT:9"], "idle_timeout": 0, "cookie": 0, "packet_count": 6, "hard_timeout": 0, "byte_count": 468,
"length": 88, "duration_nsec": 651000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
10}},

		{"actions": ["OUTPUT:10"], "idle_timeout": 0, "cookie": 0, "packet_count": 2, "hard_timeout": 0, "byte_count": 160,
"length": 88, "duration_nsec": 611000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
9}},

		{"actions": ["OUTPUT:7"], "idle_timeout": 0, "cookie": 0, "packet_count": 728, "hard_timeout": 0, "byte_count":
62969, "length": 88, "duration_nsec": 575000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match":
{"in_port": 8}},

		{"actions": ["OUTPUT:8"], "idle_timeout": 0, "cookie": 0, "packet_count": 740, "hard_timeout": 0, "byte_count":
97920, "length": 88, "duration_nsec": 538000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match":
{"in_port": 7}},
```

```
            {"actions": ["OUTPUT:6"], "idle_timeout": 0, "cookie": 0, "packet_count": 78, "hard_timeout": 0, "byte_count": 6952,
"length": 88, "duration_nsec": 505000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
5}},

            {"actions": ["OUTPUT:5"], "idle_timeout": 0, "cookie": 0, "packet_count": 83, "hard_timeout": 0, "byte_count": 8074,
"length": 88, "duration_nsec": 471000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
6}},

            {"actions": ["OUTPUT:3"], "idle_timeout": 0, "cookie": 0, "packet_count": 60, "hard_timeout": 0, "byte_count": 4584,
"length": 88, "duration_nsec": 435000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
4}},

            {"actions": ["OUTPUT:4"], "idle_timeout": 0, "cookie": 0, "packet_count": 1, "hard_timeout": 0, "byte_count": 70,
"length": 88, "duration_nsec": 401000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
3}},

            {"actions": ["OUTPUT:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 75, "hard_timeout": 0, "byte_count": 6762,
"length": 88, "duration_nsec": 365000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
2}},

            {"actions": ["OUTPUT:2"], "idle_timeout": 0, "cookie": 0, "packet_count": 73, "hard_timeout": 0, "byte_count": 6850,
"length": 88, "duration_nsec": 333000000, "priority": 300, "duration_sec": 256, "table_id": 0, "flags": 0, "match": {"in_port":
1}},

            {"actions": ["OUTPUT:12"], "idle_timeout": 0, "cookie": 0, "packet_count": 1, "hard_timeout": 0, "byte_count": 70,
"length": 88, "duration_nsec": 306000000, "priority": 32768, "duration_sec": 255, "table_id": 0, "flags": 0, "match": {"in_port":
11}},

            {"actions": ["OUTPUT:11"], "idle_timeout": 0, "cookie": 0, "packet_count": 2, "hard_timeout": 0, "byte_count": 160,
"length": 88, "duration_nsec": 272000000, "priority": 32768, "duration_sec": 255, "table_id": 0, "flags": 0, "match": {"in_port":
12}}

      ]
}
```


2. Tabla de Flujo de peo2 después de creación de circuitos virtuales.


```
{
      "2886729733":
```

```
[
        {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0,
"byte_count": 0, "length": 88, "duration_nsec": 596000000, "priority": 301, "duration_sec": 18992, "table_id": 0, "flags": 0,
"match": {"dl_type": 35138}},

        {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 21238, "hard_timeout": 0,
"byte_count": 1083138, "length": 88, "duration_nsec": 586000000, "priority": 301, "duration_sec": 18992, "table_id": 0, "flags":
0, "match": {"dl_type": 35020}},

        {"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 2, "hard_timeout": 0, "byte_count":
92, "length": 72, "duration_nsec": 229000000, "priority": 32768, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"dl_type": 34888}},

        {"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 391, "hard_timeout": 0, "byte_count":
36681, "length": 72, "duration_nsec": 220000000, "priority": 32768, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"dl_type": 34887}},

        {"actions": ["OUTPUT:9"], "idle_timeout": 0, "cookie": 0, "packet_count": 6, "hard_timeout": 0, "byte_count": 468,
"length": 88, "duration_nsec": 555000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port":
10}},

        {"actions": ["OUTPUT:10"], "idle_timeout": 0, "cookie": 0, "packet_count": 5, "hard_timeout": 0, "byte_count": 286,
"length": 88, "duration_nsec": 515000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port":
9}},

        {"actions": ["OUTPUT:7"], "idle_timeout": 0, "cookie": 0, "packet_count": 117875, "hard_timeout": 0, "byte_count":
9812273, "length": 88, "duration_nsec": 479000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"in_port": 8}},

        {"actions": ["OUTPUT:8"], "idle_timeout": 0, "cookie": 0, "packet_count": 117936, "hard_timeout": 0, "byte_count":
16212226, "length": 88, "duration_nsec": 442000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"in_port": 7}},

        {"actions": ["OUTPUT:6"], "idle_timeout": 0, "cookie": 0, "packet_count": 3954, "hard_timeout": 0, "byte_count":
328124, "length": 88, "duration_nsec": 409000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"in_port": 5}},

        {"actions": ["OUTPUT:5"], "idle_timeout": 0, "cookie": 0, "packet_count": 4033, "hard_timeout": 0, "byte_count":
334742, "length": 88, "duration_nsec": 375000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match":
{"in_port": 6}},
```

{"actions": ["OUTPUT:3"], "idle_timeout": 0, "cookie": 0, "packet_count": 3818, "hard_timeout": 0, "byte_count": 297724, "length": 88, "duration_nsec": 339000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port": 4}},

{"actions": ["OUTPUT:4"], "idle_timeout": 0, "cookie": 0, "packet_count": 13, "hard_timeout": 0, "byte_count": 1022, "length": 88, "duration_nsec": 305000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port": 3}},

{"actions": ["OUTPUT:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 3961, "hard_timeout": 0, "byte_count": 328906, "length": 88, "duration_nsec": 269000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port": 2}},

{"actions": ["OUTPUT:2"], "idle_timeout": 0, "cookie": 0, "packet_count": 4023, "hard_timeout": 0, "byte_count": 333366, "length": 88, "duration_nsec": 237000000, "priority": 300, "duration_sec": 18992, "table_id": 0, "flags": 0, "match": {"in_port": 1}},

{"actions": ["OUTPUT:12"], "idle_timeout": 0, "cookie": 0, "packet_count": 12, "hard_timeout": 0, "byte_count": 868, "length": 88, "duration_nsec": 210000000, "priority": 32768, "duration_sec": 18991, "table_id": 0, "flags": 0, "match": {"in_port": 11}},

{"actions": ["OUTPUT:11"], "idle_timeout": 0, "cookie": 0, "packet_count": 387, "hard_timeout": 0, "byte_count": 20059, "length": 88, "duration_nsec": 176000000, "priority": 32768, "duration_sec": 18991, "table_id": 0, "flags": 0, "match": {"in_port": 12}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6, "hard_timeout": 0, "byte_count": 588, "length": 80, "duration_nsec": 230000000, "priority": 32768, "duration_sec": 961, "table_id": 0, "flags": 0, "match": {"dl_type": 2048, "in_port": 9}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2, "hard_timeout": 0, "byte_count": 84, "length": 80, "duration_nsec": 199000000, "priority": 32768, "duration_sec": 961, "table_id": 0, "flags": 0, "match": {"dl_type": 2054, "in_port": 9}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 386, "hard_timeout": 0, "byte_count": 34618, "length": 80, "duration_nsec": 511000000, "priority": 32768, "duration_sec": 960, "table_id": 0, "flags": 0, "match": {"dl_type": 2048, "in_port": 13}},

{"actions": ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:16}", "OUTPUT:6"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6, "hard_timeout": 0, "byte_count": 588, "length": 120, "duration_nsec": 164000000, "priority": 32768, "duration_sec": 961, "table_id": 1, "flags": 0, "match": {"dl_type": 2048, "in_port": 9}},

```
            {"actions": ["PUSH_MPLS:34888", "SET_FIELD: {mpls_label:16}", "OUTPUT:6"], "idle_timeout": 0, "cookie":
10316443042833537412, "packet_count": 2, "hard_timeout": 0, "byte_count": 84, "length": 120, "duration_nsec": 128000000,
"priority": 32768, "duration_sec": 961, "table_id": 1, "flags": 0, "match": {"dl_type": 2054, "in_port": 9}},

            {"actions": ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:17}", "SET_FIELD: {eth_src:02:00:ac:10:00:05}", "SET_FIELD:
{eth_dst:02:00:ac:10:00:01}", "OUTPUT:6"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 386, "hard_timeout":
0, "byte_count": 34618, "length": 152, "duration_nsec": 481000000, "priority": 32768, "duration_sec": 960, "table_id": 1,
"flags": 0, "match": {"dl_type": 2048, "in_port": 13}},

            {"actions": ["POP_MPLS:2048", "OUTPUT:9"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6,
"hard_timeout": 0, "byte_count": 612, "length": 112, "duration_nsec": 89000000, "priority": 32768, "duration_sec": 961,
"table_id": 1, "flags": 0, "match": {"dl_type": 34887, "mpls_label": 16, "in_port": 6}},

            {"actions": ["POP_MPLS:2054", "OUTPUT:9"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2,
"hard_timeout": 0, "byte_count": 92, "length": 112, "duration_nsec": 34000000, "priority": 32768, "duration_sec": 961,
"table_id": 1, "flags": 0, "match": {"dl_type": 34888, "mpls_label": 16, "in_port": 6}},

            {"actions": ["POP_MPLS:2048", "SET_FIELD: {eth_src:00:00:ac:10:00:02}", "SET_FIELD: {eth_dst:00:00:ac:10:00:01}",
"OUTPUT:13"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 385, "hard_timeout": 0, "byte_count": 36069,
"length": 152, "duration_nsec": 439000000, "priority": 32768, "duration_sec": 960, "table_id": 1, "flags": 0, "match":
{"dl_type": 34887, "dl_dst": "02:00:ac:10:00:05", "mpls_label": 17, "in_port": 6}}
      ]
}
```


3. Tabla de Flujo de peo6 antes de creación de circuitos virtuales.


```
{
      "2886729732":
      [
            {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0,
"byte_count": 0, "length": 88, "duration_nsec": 110000000, "priority": 301, "duration_sec": 161, "table_id": 0, "flags": 0,
"match": {"dl_type": 35138}},

            {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 10, "hard_timeout": 0,
"byte_count": 510, "length": 88, "duration_nsec": 102000000, "priority": 301, "duration_sec": 161, "table_id": 0, "flags": 0,
"match": {"dl_type": 35020}},

            {"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0,
"length": 72, "duration_nsec": 752000000, "priority": 32768, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"dl_type":
34888}},
```

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0, "byte_count": 0, "length": 72, "duration_nsec": 744000000, "priority": 32768, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"dl_type": 34887}},

{"actions": ["OUTPUT:9"], "idle_timeout": 0, "cookie": 0, "packet_count": 6, "hard_timeout": 0, "byte_count": 468, "length": 88, "duration_nsec": 70000000, "priority": 300, "duration_sec": 161, "table_id": 0, "flags": 0, "match": {"in_port": 10}},

{"actions": ["OUTPUT:10"], "idle_timeout": 0, "cookie": 0, "packet_count": 1, "hard_timeout": 0, "byte_count": 70, "length": 88, "duration_nsec": 33000000, "priority": 300, "duration_sec": 161, "table_id": 0, "flags": 0, "match": {"in_port": 9}},

{"actions": ["OUTPUT:7"], "idle_timeout": 0, "cookie": 0, "packet_count": 58, "hard_timeout": 0, "byte_count": 5460, "length": 88, "duration_nsec": 998000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 8}},

{"actions": ["OUTPUT:8"], "idle_timeout": 0, "cookie": 0, "packet_count": 61, "hard_timeout": 0, "byte_count": 5874, "length": 88, "duration_nsec": 963000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 7}},

{"actions": ["OUTPUT:5"], "idle_timeout": 0, "cookie": 0, "packet_count": 60, "hard_timeout": 0, "byte_count": 5748, "length": 88, "duration_nsec": 928000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 6}},

{"actions": ["OUTPUT:6"], "idle_timeout": 0, "cookie": 0, "packet_count": 61, "hard_timeout": 0, "byte_count": 5794, "length": 88, "duration_nsec": 894000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 5}},

{"actions": ["OUTPUT:3"], "idle_timeout": 0, "cookie": 0, "packet_count": 41, "hard_timeout": 0, "byte_count": 3102, "length": 88, "duration_nsec": 861000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 4}},

{"actions": ["OUTPUT:4"], "idle_timeout": 0, "cookie": 0, "packet_count": 1, "hard_timeout": 0, "byte_count": 70, "length": 88, "duration_nsec": 827000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 3}},

{"actions": ["OUTPUT:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 109, "hard_timeout": 0, "byte_count": 10448, "length": 88, "duration_nsec": 793000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match": {"in_port": 2}},

```
            {"actions": ["OUTPUT:2"], "idle_timeout": 0, "cookie": 0, "packet_count": 124, "hard_timeout": 0, "byte_count":
12534, "length": 88, "duration_nsec": 761000000, "priority": 300, "duration_sec": 160, "table_id": 0, "flags": 0, "match":
{"in_port": 1}},

            {"actions": ["OUTPUT:12"], "idle_timeout": 0, "cookie": 0, "packet_count": 1, "hard_timeout": 0, "byte_count": 70,
"length": 88, "duration_nsec": 873000000, "priority": 32768, "duration_sec": 158, "table_id": 0, "flags": 0, "match": {"in_port":
11}},

            {"actions": ["OUTPUT:11"], "idle_timeout": 0, "cookie": 0, "packet_count": 2, "hard_timeout": 0, "byte_count": 160,
"length": 88, "duration_nsec": 835000000, "priority": 32768, "duration_sec": 158, "table_id": 0, "flags": 0, "match": {"in_port":
12}}
      ]
}
```


4. Tabla de Flujo de peo6 después de creación de circuitos virtuales.


```
{
      "2886729732":
      [
            {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 0, "hard_timeout": 0,
"byte_count": 0, "length": 88, "duration_nsec": 167000000, "priority": 301, "duration_sec": 19006, "table_id": 0, "flags": 0,
"match": {"dl_type": 35138}},

            {"actions": ["OUTPUT:4294967293"], "idle_timeout": 0, "cookie": 0, "packet_count": 21253, "hard_timeout": 0,
"byte_count": 1083903, "length": 88, "duration_nsec": 159000000, "priority": 301, "duration_sec": 19006, "table_id": 0, "flags":
0, "match": {"dl_type": 35020}},

            {"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 2, "hard_timeout": 0, "byte_count":
92, "length": 72, "duration_nsec": 809000000, "priority": 32768, "duration_sec": 19005, "table_id": 0, "flags": 0, "match":
{"dl_type": 34888}},

            {"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 397, "hard_timeout": 0, "byte_count":
37239, "length": 72, "duration_nsec": 801000000, "priority": 32768, "duration_sec": 19005, "table_id": 0, "flags": 0, "match":
{"dl_type": 34887}},

            {"actions": ["OUTPUT:9"], "idle_timeout": 0, "cookie": 0, "packet_count": 6, "hard_timeout": 0, "byte_count": 468,
"length": 88, "duration_nsec": 127000000, "priority": 300, "duration_sec": 19006, "table_id": 0, "flags": 0, "match": {"in_port":
10}},
```

{"actions": ["OUTPUT:10"], "idle_timeout": 0, "cookie": 0, "packet_count": 4, "hard_timeout": 0, "byte_count": 196, "length": 88, "duration_nsec": 90000000, "priority": 300, "duration_sec": 19006, "table_id": 0, "flags": 0, "match": {"in_port": 9}},

{"actions": ["OUTPUT:7"], "idle_timeout": 0, "cookie": 0, "packet_count": 3957, "hard_timeout": 0, "byte_count": 327702, "length": 88, "duration_nsec": 55000000, "priority": 300, "duration_sec": 19006, "table_id": 0, "flags": 0, "match": {"in_port": 8}},

{"actions": ["OUTPUT:8"], "idle_timeout": 0, "cookie": 0, "packet_count": 4018, "hard_timeout": 0, "byte_count": 333284, "length": 88, "duration_nsec": 20000000, "priority": 300, "duration_sec": 19006, "table_id": 0, "flags": 0, "match": {"in_port": 7}},

{"actions": ["OUTPUT:5"], "idle_timeout": 0, "cookie": 0, "packet_count": 3957, "hard_timeout": 0, "byte_count": 328866, "length": 88, "duration_nsec": 985000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 6}},

{"actions": ["OUTPUT:6"], "idle_timeout": 0, "cookie": 0, "packet_count": 4039, "hard_timeout": 0, "byte_count": 334718, "length": 88, "duration_nsec": 951000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 5}},

{"actions": ["OUTPUT:3"], "idle_timeout": 0, "cookie": 0, "packet_count": 3820, "hard_timeout": 0, "byte_count": 297880, "length": 88, "duration_nsec": 918000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 4}},

{"actions": ["OUTPUT:4"], "idle_timeout": 0, "cookie": 0, "packet_count": 13, "hard_timeout": 0, "byte_count": 1022, "length": 88, "duration_nsec": 883000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 3}},

{"actions": ["OUTPUT:1"], "idle_timeout": 0, "cookie": 0, "packet_count": 118008, "hard_timeout": 0, "byte_count": 9824277, "length": 88, "duration_nsec": 849000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 2}},

{"actions": ["OUTPUT:2"], "idle_timeout": 0, "cookie": 0, "packet_count": 118076, "hard_timeout": 0, "byte_count": 16225764, "length": 88, "duration_nsec": 817000000, "priority": 300, "duration_sec": 19005, "table_id": 0, "flags": 0, "match": {"in_port": 1}},

{"actions": ["OUTPUT:12"], "idle_timeout": 0, "cookie": 0, "packet_count": 12, "hard_timeout": 0, "byte_count": 868, "length": 88, "duration_nsec": 929000000, "priority": 32768, "duration_sec": 19003, "table_id": 0, "flags": 0, "match": {"in_port": 11}},

{"actions": ["OUTPUT:11"], "idle_timeout": 0, "cookie": 0, "packet_count": 393, "hard_timeout": 0, "byte_count": 20365, "length": 88, "duration_nsec": 891000000, "priority": 32768, "duration_sec": 19003, "table_id": 0, "flags": 0, "match": {"in_port": 12}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6, "hard_timeout": 0, "byte_count": 588, "length": 80, "duration_nsec": 418000000, "priority": 32768, "duration_sec": 973, "table_id": 0, "flags": 0, "match": {"dl_type": 2048, "in_port": 9}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2, "hard_timeout": 0, "byte_count": 84, "length": 80, "duration_nsec": 385000000, "priority": 32768, "duration_sec": 973, "table_id": 0, "flags": 0, "match": {"dl_type": 2054, "in_port": 9}},

{"actions": ["GOTO_TABLE:1"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 390, "hard_timeout": 0, "byte_count": 34974, "length": 80, "duration_nsec": 888000000, "priority": 32768, "duration_sec": 972, "table_id": 0, "flags": 0, "match": {"dl_type": 2048, "in_port": 13}},

{"actions": ["POP_MPLS:2048", "OUTPUT:9"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6, "hard_timeout": 0, "byte_count": 612, "length": 112, "duration_nsec": 576000000, "priority": 32768, "duration_sec": 973, "table_id": 1, "flags": 0, "match": {"dl_type": 34887, "mpls_label": 16, "in_port": 5}},

{"actions": ["POP_MPLS:2054", "OUTPUT:9"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2, "hard_timeout": 0, "byte_count": 92, "length": 112, "duration_nsec": 531000000, "priority": 32768, "duration_sec": 973, "table_id": 1, "flags": 0, "match": {"dl_type": 34888, "mpls_label": 16, "in_port": 5}},

{"actions": ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:16}", "OUTPUT:5"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 6, "hard_timeout": 0, "byte_count": 588, "length": 120, "duration_nsec": 495000000, "priority": 32768, "duration_sec": 973, "table_id": 1, "flags": 0, "match": {"dl_type": 2048, "in_port": 9}},

{"actions": ["PUSH_MPLS:34888", "SET_FIELD: {mpls_label:16}", "OUTPUT:5"], "idle_timeout": 0, "cookie": 10316443042833537412, "packet_count": 2, "hard_timeout": 0, "byte_count": 84, "length": 120, "duration_nsec": 462000000, "priority": 32768, "duration_sec": 973, "table_id": 1, "flags": 0, "match": {"dl_type": 2054, "in_port": 9}},

{"actions": ["PUSH_MPLS:34887", "SET_FIELD: {mpls_label:17}", "SET_FIELD: {eth_src:02:00:ac:10:00:04}", "SET_FIELD: {eth_dst:02:00:ac:10:00:01}", "OUTPUT:5"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 390, "hard_timeout": 0, "byte_count": 34974, "length": 152, "duration_nsec": 929000000, "priority": 32768, "duration_sec": 972, "table_id": 1, "flags": 0, "match": {"dl_type": 2048, "in_port": 13}},

{"actions": ["POP_MPLS:2048", "SET_FIELD: {eth_src:00:00:ac:10:00:01}", "SET_FIELD: {eth_dst:00:00:ac:10:00:02}", "OUTPUT:13"], "idle_timeout": 0, "cookie": 1054622872232268153, "packet_count": 391, "hard_timeout": 0, "byte_count": 36627,

```
"length": 152, "duration_nsec": 978000000, "priority": 32768, "duration_sec": 972, "table_id": 1, "flags": 0, "match":
{"dl_type": 34887, "dl_dst": "02:00:ac:10:00:04", "mpls_label": 17, "in_port": 5}}
      ]
}
```